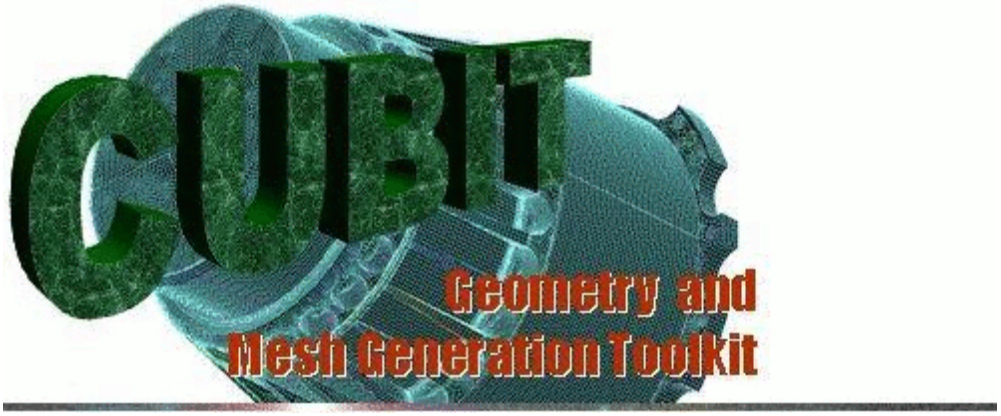
SAND Number: SAND2024-07256W






# CUBIT® 16.18 User Documentation

**Introduction** - A quick overview of some of the main features and goals of the CUBIT Mesh Generation Toolkit, licensing and distribution, hardware requirements, and where to go for help.

**Environment Control** - A description of the CUBIT user environment, including using the graphical user interface, session control, command line syntax, journal files, graphics, entity picking, saving and restoring etc..

**Geometry** - A description of CUBIT's geometry features including building geometry from scratch, manipulating geometry in CUBIT, importing and exporting geometry formats, etc...

**Mesh Generation** - A description of CUBIT's mesh generation capabilities, including how to mesh geometry, meshing and smoothing schemes, setting sizes and intervals, importing a mesh, etc...

**Finite Element Model** - How to set up the finite element model for analysis, including defining boundary conditions, material properties, exporting the finite element model, etc.

**Boundary Layer Meshing** - How to set up boundary layers.

**Immersive Topology Environment for Meshing (ITEM)** - A description of Cubit's interactive meshing wizard including how to use the wizard, and a guide to geometry clean-up, setting up the finite element model, mesh generation in ITEM, etc.

**Step-By-Step Tutorials**

**Appendix**

**Credits**

**Quick Reference**

**Official CUBIT Web Page**

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Table of Contents


CUBIT® 16.18 User Documentation ..... 1
Introduction ..... 5
Introduction ..... 5
CUBIT Mailing Lists ..... 5
CUBIT® users mailing list ..... 5
CUBIT® Announcements mailing list ..... 6
Hardware Requirements ..... 6
How to Use This Manual ..... 6
Key Features ..... 6
Geometry Creation, Modification, and Healing ..... 6
Non-Manifold Topology ..... 7
Geometry Decomposition ..... 7
Mesh Generation ..... 7
Boundary Conditions ..... 7
Element Types ..... 8
Graphics Display Capabilities ..... 8
Graphical User Interface ..... 8
Command Line Interface ..... 8
Licensing, Distribution and Installation ..... 8
Problem Reports and Enhancement Requests ..... 8
Trademark Notice ..... 8
Environment Control ..... 11
Environment Control ..... 11
session control ..... 11
Session Control ..... 11
Command Line Help ..... 11
Command Syntax ..... 12
Environment Commands ..... 14
Environment Variables ..... 17
Execution Command Syntax ..... 19
Initialization Files ..... 23
Interrupting Running Tasks ..... 23

Saving and Restoring a Cubit Session ..... 24
Starting and Exiting a CUBIT Session ..... 25
recording and playback ..... 27
Command Recording and Playback ..... 27
Automatic Journal File Creation ..... 27
Controlling Playback of Journal Files ..... 28
Journal File Creation and Playback ..... 29
Idless Journal Files ..... 30
location direction specification ..... 31
Location, Direction and Axis Specification ..... 31
Drawing a Location, Direction, or Axis ..... 32
Specifying an Axis ..... 33
Specifying a Direction ..... 35
Specifying a Location ..... 38
Specifying a Location on a Curve ..... 43
Specifying a Plane ..... 46
listing information ..... 52
Listing Information ..... 52
List Cubit Environment ..... 53
List Geometry ..... 56
List Mesh ..... 57
List Model Summary ..... 58
List Special Entities ..... 58
gui ..... 59
Graphical User Interface ..... 59
tree view ..... 59
graphics window ..... 118
drop down menus ..... 126
control panel ..... 133
CUBIT Application Window ..... 140
Command Line Workspace ..... 144
Journal File Editor ..... 146
Property Editor ..... 148
Toolbars ..... 151

Toolbar Customization ..... 154
graphics window control ..... 170
Graphics Window Control ..... 170
Graphics Clipping Plane ..... 171
Colors ..... 173
Drawing, Locating, and Highlighting Entities ..... 176
Drawing Locations, Lines and Polygons ..... 181
Entity Labels ..... 182
Graphics Camera ..... 184
Graphics Modes ..... 186
Graphics Window Size and Position ..... 188
Hardcopy Output ..... 189
Graphics Lighting Model ..... 190
Mesh Visualization ..... 191
Miscellaneous Graphics Options ..... 192
Mouse Based View Navigation: Zoom, Pan and Rotate ..... 197
Saving Graphics Views ..... 200
Updating the Display ..... 202
Geometry, Mesh, and BC Entity Visibility ..... 202
Command Line View Navigation: Zoom, Pan and Rotate ..... 204
entity selection and filtering ..... 205
Entity Selection ..... 205
Extended Selection Dialog ..... 205
Command Line Entity Specification ..... 214
Extended Command Line Entity Specification ..... 217
Selecting Entities with the Mouse ..... 223
Part Classification with Machine Learning ..... 228
Training CAD Operations with Machine Learning ..... 232
Geometry ..... 237
Geometry ..... 237
model definitions ..... 237
CUBIT Geometry Formats ..... 237
ACIS Geometry Kernel ..... 239
Mesh-Based Geometry ..... 239

geom creation ..... 243
Geometry Creation ..... 243
primitive geometry ..... 244
bottom up creation ..... 249
transforms ..... 277
Geometry Transforms ..... 277
Align Command ..... 278
Copy Command ..... 280
Move Command ..... 281
Reflect Command ..... 282
Rotate Command ..... 283
Scale Command ..... 283
booleans ..... 284
Geometry Booleans ..... 284
Intersect ..... 284
Subtract ..... 284
Unite ..... 285
decomposition ..... 286
Geometry Decomposition ..... 286
web cutting ..... 286
splitting geometry ..... 294
Section Command ..... 311
Separating Surfaces from Bodies ..... 312
Separating Multi-Volume Bodies ..... 312
cleanup and defeaturing ..... 312
Geometry Cleanup and Defeaturing ..... 312
healing ..... 313
tweaking geometry ..... 317
removing geometric features ..... 338
auto clean ..... 340
reduce ..... 345
Debugging Geometry ..... 410
Finding Surface Overlap ..... 411
Geometry Accuracy ..... 414

Regularizing Geometry ..... 414
Stitching Sheet Bodies ..... 414
Trimming and Extending Curves ..... 415
Validating Geometry ..... 417
Blunt Tangency ..... 418
imprint merge ..... 422
Geometry Imprinting and Merging ..... 422
Examining Merged Entities ..... 423
Imprinting Geometry ..... 423
Merge Tolerance ..... 425
Merging Geometry ..... 426
Using Geometry Merging to Verify Geometry ..... 428
Unmerging ..... 428
virtual geometry ..... 429
Virtual Geometry ..... 429
collapse geometry ..... 430
composite geometry ..... 436
partitioned geometry ..... 438
Deleting Virtual Geometry ..... 444
Simplify Geometry ..... 446
groups ..... 449
Groups ..... 449
Basic Group Operations ..... 449
Groups in Graphics ..... 451
Propagated Hex Groups ..... 451
Quality Groups ..... 458
Propagated Groups ..... 459
Seeded Mesh Groups ..... 459
Naming Convention for Propagated Hex Groups ..... 461
attributes ..... 462
Geometry Attributes ..... 462
persistent attributes ..... 462
Entity IDs ..... 465
Entity Names ..... 467

metadata ..... 470
Parts, Assemblies, and Metadata ..... 470
Importing and Exporting Metadata ..... 471
Metadata Attributes ..... 472
Working With Parts and Assemblies ..... 475
import ..... 479
Importing Geometry ..... 479
Importing ACIS Files ..... 480
Importing Facet Files ..... 482
Importing FASTQ Files ..... 486
Importing Granite Files ..... 486
Importing IGES Files ..... 486
Importing STEP Files ..... 487
Importing SGM Files ..... 488
export ..... 489
Exporting Geometry ..... 489
Exporting ACIS Files ..... 489
Exporting Facet Files ..... 490
Exporting IGES Files ..... 491
Exporting STEP Files ..... 491
Geometry Deletion ..... 492
Geometry Orientation ..... 492
Adjusting Orientation ..... 493
Entity Measurement ..... 494
Measure Between ..... 494
Measure Small ..... 495
Measure Angle ..... 495
Measure Void ..... 495
Measure Volume ..... 495
Measure Surface ..... 496
Mesh Generation ..... 497
Mesh Generation ..... 497
Element Types ..... 497
Mesh Generation Process ..... 497

Meshing the Geometry ..... 498
Default Scheme and Interval Selection ..... 499
Continuing Meshing After a Mesh Failure ..... 499
interval assignment ..... 499
Interval Assignment ..... 499
Additional Interval Constraints ..... 500
Vertex Sizing and Automatic Curve Biasing ..... 501
Explicit Specification of Intervals ..... 502
Explicit Specification of Intervals Using Interval Size ..... 502
Automatic Specification of Interval Size ..... 503
Interval Matching ..... 506
Interval Firmness ..... 508
Relative Intervals ..... 509
Mesh Interval Preview ..... 509
Periodic Intervals ..... 510
meshing schemes ..... 510
Meshing Schemes ..... 510
duplication ..... 512
conversion ..... 515
traditional ..... 521
parallel ..... 582
Automatic Scheme Selection ..... 723
free ..... 726
mesh quality assessment ..... 731
Automatic Mesh Quality Assessment ..... 731
Coincident Node Check ..... 732
Controlling Mesh Quality ..... 733
Metrics for Edge Elements ..... 734
Metrics for Hexahedral Elements ..... 735
Mesh Quality Assessment ..... 737
Mesh Quality Example Output ..... 738
Mesh Quality Command Syntax ..... 740
Metrics for Quadrilateral Elements ..... 743
Metrics for Tetrahedral Elements ..... 745

Mesh Topology Check ..... 748
Metrics for Triangular Elements ..... 751
Metrics for Wedge Elements ..... 753
Metrics supporting higher-order element types ..... 753
Measuring Number of Tets Through the Thickness ..... 754
Finding Intersecting Mesh ..... 755
mesh modification ..... 758
Mesh Modification ..... 759
mesh smoothing ..... 759
Align Mesh ..... 771
Collapsing Mesh ..... 772
Creating and Merging Mesh Elements ..... 774
Mesh Cleanup ..... 777
Remeshing ..... 779
Edge Swapping ..... 783
Matching Tetrahedral Meshes ..... 784
Mesh Coarsening ..... 784
Mesh Refinement ..... 786
Mesh Scaling ..... 802
Block Repositioning ..... 810
Node and Nodeset Repositioning ..... 810
Mesh Pillowing ..... 811
Mesh Column Operations ..... 813
Removing Intersecting Mesh ..... 817
mesh import ..... 818
Importing a Mesh ..... 818
Importing Fluent Files ..... 819
Importing 2D Exodus Files ..... 819
Importing Abaqus Files ..... 820
Importing Exodus II Files ..... 821
Importing I-DEAS Files ..... 832
Importing Nastran Files ..... 833
Importing Patran Files ..... 833
adaptivity and sizing functions ..... 834

Geometry Adaptive Sizing for TriMesh and TetMesh Schemes ..... 834
Mesh Adaptivity and Sizing Functions ..... 834
Bias Sizing Function ..... 836
Constant Sizing Function ..... 842
Curvature Sizing Function ..... 843
Exodus II-based Field Function ..... 844
Geometry Adaptive Sizing Function (Skeleton Sizing) ..... 847
Interval Sizing Function ..... 853
Inverse Sizing Function ..... 854
Linear Sizing Function ..... 855
Free Meshes ..... 856
Creating a free mesh ..... 857
Creating Mesh-Based Geometry to fit a Free Mesh ..... 857
Merging a free mesh ..... 858
Free Mesh Transformation Operations ..... 858
Smoothing a free mesh ..... 863
Mesh quality on a free mesh ..... 864
Mesh refinement on a free mesh ..... 864
Cleaning up a free mesh ..... 864
Assigning boundary conditions ..... 864
Skinning a free mesh ..... 866
Deleting free mesh elements ..... 866
Bottom-up element creation ..... 866
Exporting free meshes ..... 867
Mesh Deletion ..... 867
Automatic Mesh Deletion ..... 867
Mesh Validity ..... 867
Skinning a Mesh ..... 868
Lite Meshes ..... 869
Creating a lite mesh ..... 869
Graphics ..... 870
Information ..... 871
Modification to lite mesh ..... 871
Exporting a lite mesh ..... 871

Equivalencing of nodes ..... 871
Finite Element Model ..... 873
Finite Element Model ..... 873
Global Element IDs ..... 873
Cubit Mesh Entity ID Spaces ..... 873
Global Element IDs ..... 873
export ..... 874
Export Mesh and Its Geometry Association ..... 874
Exporting Sierra Files ..... 874
Defining PARAMS for NASTRAN ..... 875
Exporting ABAQUS ..... 875
Exporting an Exodus II File ..... 876
Exporting the Finite Element Model ..... 879
Exporting Fluent Grid Files ..... 883
Transforming Mesh Coordinates ..... 884
Coordinate Frames ..... 885
Exporting GDF Files ..... 886
exodus ..... 886
Element Block Specification ..... 886
Exodus II File Specification ..... 899
Exodus II Model Title ..... 900
Defining Materials and Media Types ..... 900
Exodus Boundary Conditions ..... 903
Nodeset and Sideset Specification ..... 905
non exodus ..... 916
CUBIT Initial Conditions ..... 916
Using Constraints ..... 917
Cubit Boundary Conditions ..... 918
CFD Boundary Conditions ..... 919
Using Contact Surfaces ..... 920
Using Loads ..... 921
Miscellaneous Boundary Condition Commands ..... 924
Using Restraints ..... 925
Boundary Condition Sets ..... 928

Boundary Layer Meshing ..... 931
Boundary Layer Meshing ..... 931
Step-by-Step Tutorials ..... 939
Step-By-Step Tutorials ..... 939
Additional Tutorials ..... 939
item ..... 940
ITEM Tutorial ..... 940
ITEM Tutorial Step 1 ..... 941
ITEM Tutorial Step 2 ..... 942
ITEM Tutorial Step 3 ..... 944
ITEM Tutorial Step 4 ..... 950
ITEM Tutorial Step 5 ..... 953
ITEM Tutorial Step 6 ..... 957
ITEM Tutorial Step 7 ..... 962
ITEM Tutorial Step 8 ..... 963
ITEM Tutorial Step 9 ..... 968
power tools ..... 969
Power Tools GUI Tutorial ..... 969
Power Tools GUI Tutorial Step 1 ..... 971
Power Tools GUI Tutorial Step 2 ..... 976
Power Tools GUI Tutorial Step 3 ..... 980
Power Tools GUI Tutorial Step 4 ..... 983
Power Tools GUI Tutorial Step 5 ..... 985
Power Tools GUI Tutorial Step 6 ..... 989
Power Tools GUI Tutorial Step 7 ..... 996
Power Tools GUI Tutorial Step 8 ..... 1002
Power Tools GUI Tutorial Step 9 ..... 1005
Power Tools GUI Tutorial Step 10 ..... 1006
Power Tools GUI Tutorial Step 11 ..... 1015
decomposition ..... 1021
Decomposition Tutorial ..... 1021
Example 1. Sweeping multiple adjacent volumes ..... 1027
Example 2. Interlocking rings ..... 1030
Example 3. Webcutting using the sweep option ..... 1032

Example 4. Using the Loft command ..... 1034
Example 5. Multiple sweep directions ..... 1037
Example 6. Employing Symmetry ..... 1039
Example 7. Using virtual geometry in geometry decomposition ..... 1052
Example 8. Sweeping volumes with narrow angles and surfaces ..... 1063
gui ..... 1078
GUI Basic Tutorial ..... 1078
GUI Basic Tutorial Step 1 ..... 1080
GUI Basic Tutorial Step 2 ..... 1082
GUI Basic Tutorial Step 3 ..... 1085
GUI Basic Tutorial Step 4 ..... 1086
GUI Basic Tutorial Step 5 ..... 1087
GUI Basic Tutorial Step 6 ..... 1089
GUI Basic Tutorial Step 7 ..... 1094
GUI Basic Tutorial Step 8 ..... 1098
GUI Basic Tutorial Step 9 ..... 1100
GUI Basic Tutorial Step 10 ..... 1102
GUI Basic Tutorial Step 11 ..... 1105
command line ..... 1106
Command Line Basic Tutorial ..... 1106
CL Basic Tutorial Step 1 ..... 1107
CL Basic Tutorial Step 2 ..... 1108
CL Basic Tutorial Step 3 ..... 1109
CL Basic Tutorial Step 4 ..... 1110
CL Basic Tutorial Step 5 ..... 1112
CL Basic Tutorial Step 6 ..... 1113
CL Basic Tutorial Step 7 ..... 1115
CL Basic Tutorial Step 8 ..... 1117
CL Basic Tutorial Step 9 ..... 1118
CL Basic Tutorial Step 10 ..... 1123
CL Basic Tutorial Step 11 ..... 1123
Geometry Cleanup Process Flow ..... 1124
ITEM ..... 1127
Immersive Topology Environment for Meshing (ITEM) ..... 1127

Guiding the user through the workflow. ..... 1127
Providing the user with smart options. ..... 1128
Automating geometry and meshing tasks. ..... 1128
How to Use the ITEM Wizard ..... 1129
The ITEM Workflow ..... 1129
Using an ITEM Panel ..... 1130
Undo Button ..... 1135
Magic Mesh Button ..... 1136
Getting Help ..... 1136
Setting up the Finite Element Model ..... 1137
Defining the Geometric Model ..... 1138
Generating a Mesh in ITEM ..... 1139
ITEM Meshing Suggestions ..... 1140
Validating the Mesh in ITEM ..... 1144
clean up ..... 1144
Recognizing Nearly Sweepable Regions ..... 1144
Blend Surfaces ..... 1146
Clean Up the Geometry ..... 1146
Resolving Problems with Conformal Assemblies ..... 1148
Contact Surfaces ..... 1152
Geometry Decomposition ..... 1152
Forced Sweepability ..... 1154
Bad geometry representation ..... 1155
Determining an Appropriate Merge Tolerance ..... 1156
Building a Sweepable Topology ..... 1158
Small details in the model ..... 1158
Determining the Small Feature Size ..... 1163
Appendix ..... 1167
Appendix ..... 1167
alpha ..... 1167
Alpha Commands ..... 1167
Creating ACIS Geometry From Mesh ..... 1168
Cohesive Elements ..... 1168
Deleting Mesh Elements ..... 1171

FeatureSize ..... 1172
Importing Abaqus Files ..... 1173
Importing Meshed Based Geometry Files (MBG) ..... 1173
Exporting Meshed Based Geometry Files (MBG) ..... 1174
Mesh Cutting ..... 1174
Mesh Grafting ..... 1181
Optimize Jacobian ..... 1185
Randomize ..... 1185
Refine Mesh Boundary ..... 1186
Remove Tiny Edge Length ..... 1187
Super Sizing Function ..... 1188
Test Sizing Function ..... 1189
Transition ..... 1190
Triangle Mesh Coarsening ..... 1192
Higher Order Element Metrics ..... 1194
aprepro ..... 1197
APREPRO ..... 1197
Using APREPRO in CUBIT ..... 1197
APREPRO Functions ..... 1201
APREPRO Journaling ..... 1210
python ..... 1212
Python Interface ..... 1212
Importing Cubit into Python ..... 1212
Python Cubit Enhancement Scripts ..... 1213
Classes ..... 1213
Functions ..... 1215
CubitInterface Control ..... 1300
Detailed Description ..... 1301
Function Documentation ..... 1301
◆ add_entities_to_group() ..... 1301
◆ add_entity_to_group() ..... 1302
◆ add_filename_to_recent_file_list() ..... 1302
◆ add_filter_type() ..... 1302
◆ are_adjacent_curves() ..... 1302

- are_adjacent_surfaces() ..... 1302
- auto_size_needs_to_be_calculated() ..... 1303
- best_edge_to_collapse_interior_node() ..... 1303
- body() ..... 1303
- brick() ..... 1303
- calculate_timestep_estimate() ..... 1303
- calculate_timestep_estimate_with_props() ..... 1304
- clear_drawing_set() ..... 1304
- clear_highlight() ..... 1304
- clear_picked_list() ..... 1305
- clear_preview() ..... 1305
- cmd() ..... 1305
- cmd_single() ..... 1305
- command_is_python_mode() ..... 1305
- compare_geometry_and_mesh() ..... 1305
- complete_filename() ..... 1306
- contains_virtual() ..... 1306
- convection_is_on_shell_area() ..... 1307
- convection_is_on_solid() ..... 1307
- copy_body() ..... 1307
- create_arc_curve() ..... 1307
- create_curve() ..... 1308
- create_new_group() ..... 1308
- create_spline() ..... 1308
- create_surface() ..... 1308
- create_vertex() ..... 1308
- cubit_or_python_cmd() ..... 1309
- cubit_or_python_cmds() ..... 1309
- cubit_or_python_cmds_as_file() ..... 1309
- current_selection_count() ..... 1309
- curve() ..... 1309
- cylinder() ..... 1309
- delete_all_groups() ..... 1310
- delete_group() ..... 1310

- destroy() ..... 1310
- developer_commands_are_enabled() ..... 1310
- enable_signal_handling() ..... 1310
- ensure_init() ..... 1310
- entity_exists() ..... 1311
- estimate_merge_tolerance() ..... 1311
- evaluate_exterior_angle() ..... 1312
- evaluate_exterior_angle_at_curve() ..... 1312
- evaluate_surface_angle_at_vertex() ..... 1312
- exodus_sizing_function_file_exists() ..... 1312
- find_floating_volumes() ..... 1313
- find_nonmanifold_curves() ..... 1313
- find_nonmanifold_vertices() ..... 1313
- flush_graphics() ..... 1314
- gather_surfaces_by_orientation() ..... 1314
- get_2D_sheet_volumes() ..... 1314
- get_3D_thin_volume() ..... 1314
- get_acceleration_combine_type() ..... 1314
- get_acceleration_dof_signs() ..... 1315
- get_acceleration_dof_values() ..... 1315
- get_acis_version() ..... 1315
- get_acis_version_as_int() ..... 1315
- get_adjacent_surfaces() ..... 1315
- get_adjacent_volumes() ..... 1316
- get_all_cfd_bcs() ..... 1316
- get_all_exodus_times() ..... 1316
- get_all_exodus_variable_names() ..... 1317
- get_all_geometric_owners() ..... 1317
- get_all_ids_from_name() ..... 1317
- get_aprepro_numeric_value() ..... 1318
- get_aprepro_value() ..... 1318
- get_aprepro_value_as_string() ..... 1318
- get_aprepro_vars() ..... 1319
- get_arc_length() ..... 1319

- get_assembly_children() ..... 1319
- get_assembly_classification_category() ..... 1319
- get_assembly_classification_level() ..... 1319
- get_assembly_description() ..... 1319
- get_assembly_file_format() ..... 1319
- get_assembly_instance() ..... 1320
- get_assembly_items() ..... 1320
- get_assembly_level() ..... 1320
- get_assembly_material_description() ..... 1320
- get_assembly_material_specification() ..... 1320
- get_assembly_metadata() ..... 1320
- get_assembly_name() ..... 1321
- get_assembly_path() ..... 1321
- get_assembly_type() ..... 1321
- get_assembly_units() ..... 1321
- get_assembly_weapons_category() ..... 1322
- get_auto_size() ..... 1322
- get_bad_geometry() ..... 1322
- get_bc_id_list() ..... 1322
- get_bc_name() ..... 1323
- get_bc_temperature() ..... 1323
- get_blend_chain_collections() ..... 1323
- get_blend_chains() ..... 1324
- get_blend_surfaces() ..... 1324
- get_block_attribute_count() ..... 1324
- get_block_attribute_name() ..... 1324
- get_block_attribute_value() ..... 1325
- get_block_children() ..... 1325
- get_block_count() ..... 1326
- get_block_curves() ..... 1326
- get_block_edges() ..... 1326
- get_block_element_attribute_count() ..... 1326
- get_block_element_attribute_names() ..... 1327
- get_block_element_type() ..... 1327

- get_block_elements_and_nodes() ..... 1327
- get_block_faces() ..... 1328
- get_block_hexes() ..... 1328
- get_block_id() ..... 1328
- get_block_id_list() ..... 1328
- get_block_ids() ..... 1329
- get_block_material() ..... 1329
- get_block_nodes() ..... 1329
- get_block_pyramids() ..... 1329
- get_block_surfaces() ..... 1329
- get_block_tets() ..... 1330
- get_block_tris() ..... 1330
- get_block_vertices() ..... 1330
- get_block_volumes() ..... 1330
- get_block_wedges() ..... 1330
- get_blocks_with_materials() ..... 1331
- get_blunt_tangency_default_depth() ..... 1331
- get_body_count() ..... 1331
- get_bolt_axis() ..... 1331
- get_bolt_clamped_members() ..... 1331
- get_bolt_coordinate_system() ..... 1332
- get_bolt_diameter() ..... 1332
- get_bolt_holes() ..... 1332
- get_bolt_holes_info() ..... 1333
- get_bolt_shigley_radius() ..... 1333
- get_bolts_in_clamped_members() ..... 1333
- get_bool_sculpt_default() ..... 1334
- get_boundary_layer_algorithm() ..... 1334
- get_boundary_layer_aspect_first_parameters() ..... 1334
- get_boundary_layer_aspect_last_parameters() ..... 1334
- get_boundary_layer_continuity() ..... 1334
- get_boundary_layer_curve_intersection_types() ..... 1334
- get_boundary_layer_curve_surface_pairs() ..... 1335
- get_boundary_layer_id_list() ..... 1335

- get_boundary_layer_surface_volume_pairs() ..... 1335
- get_boundary_layer_uniform_parameters() ..... 1335
- get_boundary_layer_vertex_intersection_types() ..... 1335
- get_boundary_layers_by_base() ..... 1335
- get_boundary_layers_by_pair() ..... 1336
- get_bounding_box() ..... 1336
- get_build_number() ..... 1336
- get_cavity_surfaces() ..... 1336
- get_center_point() ..... 1337
- get_cfd_type() ..... 1337
- get_chamfer_chain_collections() ..... 1337
- get_chamfer_chains() ..... 1338
- get_chamfer_surfaces() ..... 1338
- get_close_loop_thickness() ..... 1338
- get_close_loops() ..... 1338
- get_close_loops_with_thickness() ..... 1339
- get_close_vertex_curve_pairs() ..... 1339
- get_closed_narrow_surfaces() ..... 1339
- get_closest_node() ..... 1340
- get_closest_vertex_curve_pairs() ..... 1340
- get_coincident_entity_pairs() ..... 1340
- get_coincident_vertex_curve_pairs() ..... 1342
- get_coincident_vertex_surface_pairs() ..... 1342
- get_coincident_vertex_vertex_pairs() ..... 1343
- get_coincident_vertices() ..... 1343
- get_command_from_history() ..... 1344
- get_command_history_count() ..... 1344
- get_common_curve_id() ..... 1344
- get_common_vertex_id() ..... 1344
- get_cone_surfaces() ..... 1344
- get_connected_surfaces() ..... 1345
- get_connectivity() ..... 1345
- get_constraint_dependent_entity_point() ..... 1345
- get_constraint_reference_point() ..... 1346

- get_constraint_type() ..... 1346
- get_contact_pair_exterior_state() ..... 1346
- get_contact_pair_friction_value() ..... 1346
- get_contact_pair_general_state() ..... 1346
- get_contact_pair_tied_state() ..... 1347
- get_contact_pair_tol_lower_value() ..... 1347
- get_contact_pair_tolerance_value() ..... 1347
- get_continuous_curves() ..... 1347
- get_continuous_surfaces() ..... 1347
- get_convection_coefficient() ..... 1348
- get_coordinate_systems_id_list() ..... 1348
- get_copy_block_on_geometry_copy_setting() ..... 1348
- get_copy_nodeset_on_geometry_copy_setting() ..... 1348
- get_copy_sideset_on_geometry_copy_setting() ..... 1348
- get_cubit_digits_setting() ..... 1349
- get_cubit_message_handler() ..... 1349
- get_current_ids() ..... 1349
- get_current_journal_file() ..... 1349
- get_curve_bias_coarse_size() ..... 1349
- get_curve_bias_fine_size() ..... 1349
- get_curve_bias_first_interval_fraction() ..... 1349
- get_curve_bias_first_interval_length() ..... 1350
- get_curve_bias_first_last_ratio1() ..... 1350
- get_curve_bias_first_last_ratio2() ..... 1350
- get_curve_bias_from_start() ..... 1350
- get_curve_bias_from_start_set() ..... 1350
- get_curve_bias_geometric_factor() ..... 1351
- get_curve_bias_geometric_factor2() ..... 1351
- get_curve_bias_last_first_ratio1() ..... 1351
- get_curve_bias_last_first_ratio2() ..... 1351
- get_curve_bias_start_vertex_id() ..... 1352
- get_curve_bias_type() ..... 1352
- get_curve_center() ..... 1352
- get_curve_count() ..... 1352

- get_curve_count_in_volumes() ..... 1352
- get_curve_edges() ..... 1352
- get_curve_length() ..... 1353
- get_curve_mesh_scheme_curvature() ..... 1353
- get_curve_mesh_scheme_pinpoint_locations() ..... 1353
- get_curve_mesh_scheme_stretch_values() ..... 1353
- get_curve_nodes() ..... 1354
- get_curve_radius() ..... 1354
- get_curve_type() ..... 1354
- get_dbl_sculpt_default() ..... 1354
- get_default_auto_size() ..... 1354
- get_default_element_type() ..... 1355
- get_default_geometry_engine() ..... 1355
- get_displacement_combine_type() ..... 1355
- get_displacement_dof_signs() ..... 1355
- get_displacement_dof_values() ..... 1355
- get_distance_between() ..... 1356
- get_distance_between_entities() ..... 1356
- get_distance_from_curve_start() ..... 1356
- get_edge_count() ..... 1357
- get_edge_global_element_id() ..... 1357
- get_edges_to_swap() ..... 1357
- get_elem_quality_stats() ..... 1357
- get_element_block() ..... 1359
- get_element_budget() ..... 1359
- get_element_count() ..... 1359
- get_element_exists() ..... 1359
- get_element_type() ..... 1359
- get_element_type_id() ..... 1360
- get_entities() ..... 1360
- get_entity_color() ..... 1360
- get_entity_color_index() ..... 1361
- get_entity_modeler_engine() ..... 1361
- get_entity_name() ..... 1361

- get_entity_names() ..... 1362
- get_entity_sense() ..... 1362
- get_error_count() ..... 1362
- get_exodus_element_count() ..... 1362
- get_exodus_entity_description() ..... 1363
- get_exodus_entity_name() ..... 1363
- get_exodus_entity_type() ..... 1364
- get_exodus_id() ..... 1364
- get_exodus_sizing_function_file_name() ..... 1364
- get_exodus_sizing_function_variable_name() ..... 1364
- get_exodus_variable_count() ..... 1365
- get_exodus_variable_names() ..... 1365
- get_exodus_version() ..... 1365
- get_expanded_connectivity() ..... 1365
- get_force_direction_vector() ..... 1366
- get_force_magnitude() ..... 1366
- get_force_moment_vector() ..... 1366
- get_gaps_between_volumes() ..... 1366
- get_geometric_owner() ..... 1366
- get_geometry_node_count() ..... 1367
- get_geometry_owner() ..... 1367
- get_global_element_id() ..... 1368
- get_graphics_version() ..... 1368
- get_group_bodies() ..... 1368
- get_group_children() ..... 1368
- get_group_curves() ..... 1369
- get_group_edges() ..... 1369
- get_group_groups() ..... 1370
- get_group_hexes() ..... 1370
- get_group_nodes() ..... 1370
- get_group_pyramids() ..... 1370
- get_group_quads() ..... 1370
- get_group_spheres() ..... 1371
- get_group_surfaces() ..... 1371

- get_group_tets() ..... 1371
- get_group_tris() ..... 1371
- get_group_vertices() ..... 1371
- get_group_volumes() ..... 1372
- get_group_wedges() ..... 1372
- get_heatflux_on_area() ..... 1372
- get_hex_count() ..... 1372
- get_hex_global_element_id() ..... 1372
- get_hex_sheet() ..... 1373
- get_hole_surfaces() ..... 1373
- get_hydraulic_radius_surface_area() ..... 1373
- get_hydraulic_radius_volume_area() ..... 1373
- get_id_from_name() ..... 1374
- get_id_string() ..... 1374
- get_idless_signature() ..... 1374
- get_idless_signatures() ..... 1374
- get_int_sculpt_default() ..... 1375
- get_interface() ..... 1375
- get_label_type() ..... 1375
- get_last_id() ..... 1375
- get_list_of_free_ref_entities() ..... 1375
- get_material_name() ..... 1376
- get_material_name_list() ..... 1376
- get_material_property() ..... 1376
- get_media_name_list() ..... 1376
- get_media_property() ..... 1376
- get_merge_setting() ..... 1377
- get_merge_tolerance() ..... 1377
- get_mergeable_curves() ..... 1377
- get_mergeable_entities() ..... 1378
- get_mergeable_surfaces() ..... 1378
- get_mergeable_vertices() ..... 1378
- get_mesh_edge_length() ..... 1379
- get_mesh_element_type() ..... 1379

◆ get_mesh_error_count() ..... 1379
◆ get_mesh_error_solutions() ..... 1379
◆ get_mesh_errors() ..... 1379
◆ get_mesh_geometry_approximation_angle() ..... 1380
◆ get_mesh_group_parent_ids() ..... 1380
◆ get_mesh_interval_firmness() ..... 1380
◆ get_mesh_intervals() ..... 1381
◆ get_mesh_scheme() ..... 1381
◆ get_mesh_scheme_firmness() ..... 1381
◆ get_mesh_size() ..... 1382
◆ get_mesh_size_type() ..... 1382
◆ get_meshed_volume_or_area() ..... 1383
◆ get_meshgems_version() ..... 1383
◆ get_ML_classification() ..... 1383
◆ get_ML_classification_categories() ..... 1383
◆ get_ML_classification_models() ..... 1384
◆ get_ML_classifications() ..... 1384
◆ get_ML_feature_distance() ..... 1384
◆ get_ML_feature_importances() ..... 1384
◆ get_ML_model_ID() ..... 1384
◆ get_ML_model_name() ..... 1385
◆ get_ML_operation() ..... 1385
◆ get_ML_operation_feature_names() ..... 1385
◆ get_ML_operation_feature_size() ..... 1385
◆ get_ML_operation_feature_types() ..... 1386
◆ get_ML_operation_features() ..... 1386
1. 2. 3. ml_op_name Entity1 Entity2 Params ..... 1387
◆ get_ML_operation_label_size() ..... 1388
◆ get_ML_predictions() ..... 1388
◆ get_ML_regression_models() ..... 1389
◆ get_moment_magnitude() ..... 1389
◆ get_n_largest_distances_between_meshes() ..... 1389
◆ get_narrow_regions() ..... 1389
◆ get_narrow_surfaces() ..... 1390

- get_nearby_entities() ..... 1390
- get_nearby_volumes_at_volume() ..... 1390
- get_next_block_id() ..... 1391
- get_next_boundary_layer_id() ..... 1391
- get_next_command_from_history() ..... 1391
- get_next_group_id() ..... 1391
- get_next_nodeset_id() ..... 1391
- get_next_sideset_id() ..... 1391
- get_nodal_coordinates() ..... 1392
- get_node_constraint() ..... 1392
- get_node_constraint_smart_metric() ..... 1392
- get_node_constraint_smart_threshold() ..... 1392
- get_node_constraint_value() ..... 1392
- get_node_count() ..... 1392
- get_node_edges() ..... 1392
- get_node_exists() ..... 1393
- get_node_faces() ..... 1393
- get_node_global_id() ..... 1393
- get_node_position_fixed() ..... 1393
- get_node_tris() ..... 1394
- get_nodeset_children() ..... 1394
- get_nodeset_count() ..... 1394
- get_nodeset_curves() ..... 1394
- get_nodeset_id_list() ..... 1395
- get_nodeset_id_list_for_bc() ..... 1395
- get_nodeset_node_count() ..... 1395
- get_nodeset_nodes() ..... 1395
- get_nodeset_nodes_inclusive() ..... 1396
- get_nodeset_surfaces() ..... 1396
- get_nodeset_vertices() ..... 1396
- get_nodeset_volumes() ..... 1396
- get_num_volume_shells() ..... 1396
- get_overlap_max_angle() ..... 1397
- get_overlap_max_gap() ..... 1397

- get_overlap_min_gap() ..... 1397
- get_overlapping_curves() ..... 1397
- get_overlapping_surfaces() ..... 1398
- get_overlapping_surfaces_at_surface() ..... 1398
- get_overlapping_surfaces_in_bodies() ..... 1399
- get_overlapping_surfaces_in_volumes() ..... 1399
- get_overlapping_volumes() ..... 1400
- get_overlapping_volumes_at_volume() ..... 1400
- get_owning_body() ..... 1401
- get_owning_volume() ..... 1401
- get_owning_volume_by_name() ..... 1401
- get_owning_volume_ids() ..... 1402
- get_parent_assembly_instance() ..... 1402
- get_parent_assembly_path() ..... 1402
- get_periodic_data() ..... 1402
- get_pick_filters() ..... 1403
- get_pick_type() ..... 1403
- get_pressure_function() ..... 1403
- get_pressure_value() ..... 1403
- get_previous_command_from_history() ..... 1403
- get_pyramid_count() ..... 1404
- get_pyramid_global_element_id() ..... 1404
- get_python_version() ..... 1404
- get_quad_count() ..... 1404
- get_quad_global_element_id() ..... 1404
- get_quality_stats() ..... 1405
- get_quality_stats_at_geometry() ..... 1406
- get_quality_value() ..... 1407
- get_quality_values() ..... 1408
- get_reduce_bolt_core_default_dimensions() ..... 1408
- get_relatives() ..... 1408
- get_rendering_mode() ..... 1409
- get_requested_mesh_interval_firmness() ..... 1409
- get_requested_mesh_intervals() ..... 1409

- get_requested_mesh_size() ..... 1410
- get_requested_mesh_size_type() ..... 1410
- get_revision_date() ..... 1410
- get_rubberband_shape() ..... 1410
- get_selected_id() ..... 1411
- get_selected_ids() ..... 1411
- get_selected_type() ..... 1411
- get_sharp_angle_vertices() ..... 1411
- get_sharp_curve_angles() ..... 1411
- get_sharp_surface_angles() ..... 1412
- get_sideset_area() ..... 1413
- get_sideset_children() ..... 1413
- get_sideset_count() ..... 1414
- get_sideset_curves() ..... 1414
- get_sideset_edges() ..... 1414
- get_sideset_element_type() ..... 1414
- get_sideset_id_list() ..... 1414
- get_sideset_id_list_for_bc() ..... 1414
- get_sideset_quads() ..... 1415
- get_sideset_surfaces() ..... 1415
- get_sideset_tris() ..... 1415
- get_similar_curves() ..... 1416
- get_similar_surfaces() ..... 1416
- get_similar_volumes() ..... 1416
- get_sizing_function_name() ..... 1417
- get_small_and_narrow_surfaces() ..... 1417
- get_small_curves() ..... 1417
- get_small_radius_blend_surfaces() ..... 1418
- get_small_surfaces() ..... 1418
- get_small_surfaces_HR() ..... 1418
- get_small_surfaces_hydraulic_radius() ..... 1419
- get_small_volumes() ..... 1419
- get_small_volumes_hydraulic_radius() ..... 1420
- get_smallest_curves() ..... 1420

- get_smallest_features() ..... 1420
- get_smooth_scheme() ..... 1421
- get_solutions_for_bad_geometry() ..... 1421
- get_solutions_for_blends() ..... 1422
- get_solutions_for_bolt() ..... 1422
- get_solutions_for_bolt_hole() ..... 1422
- get_solutions_for_cavity_surface() ..... 1423
- get_solutions_for_classified_surface() ..... 1423
- get_solutions_for_classified_volume() ..... 1423
- get_solutions_for_close_loop() ..... 1424
- get_solutions_for_cone_surface() ..... 1424
- get_solutions_for_decomposition() ..... 1424
- get_solutions_for_forced_sweepability() ..... 1424
- get_solutions_for_imprint_merge() ..... 1425
- get_solutions_for_near_coincident_vertex_and_curve() ..... 1425
- get_solutions_for_near_coincident_vertex_and_surface() ..... 1426
- get_solutions_for_near_coincident_vertices() ..... 1426
- get_solutions_for_overlapping_surfaces() ..... 1426
- get_solutions_for_overlapping_volumes() ..... 1427
- get_solutions_for_sharp_angle_vertex() ..... 1427
- get_solutions_for_sheet_volume_connection() ..... 1428
- get_solutions_for_sheet_volumes() ..... 1428
- get_solutions_for_small_curves() ..... 1428
- get_solutions_for_small_surfaces() ..... 1429
- get_solutions_for_source_target() ..... 1429
- get_solutions_for_surfaces_with_narrow_regions() ..... 1430
- get_solutions_for_thin_volume() ..... 1430
- get_solutions_for_volumes() ..... 1430
- get_source_surfaces() ..... 1431
- get_sphere_count() ..... 1431
- get_sphere_global_element_id() ..... 1431
- get_string_sculpt_default() ..... 1431
- get_sub_elements() ..... 1432
- get_submap_corner_types() ..... 1432

- get_surface_area() ..... 1432
- get_surface_cavity_collections() ..... 1433
- get_surface_centroid() ..... 1433
- get_surface_count() ..... 1433
- get_surface_element_count() ..... 1433
- get_surface_hole_collections() ..... 1434
- get_surface_loop_nodes() ..... 1434
- get_surface_nodes() ..... 1434
- get_surface_normal() ..... 1434
- get_surface_normal_at_coord() ..... 1435
- get_surface_num_loops() ..... 1435
- get_surface_principal_curvatures() ..... 1435
- get_surface_quads() ..... 1435
- get_surface_sense() ..... 1436
- get_surface_tris() ..... 1436
- get_surface_type() ..... 1436
- get_surfs_with_narrow_regions() ..... 1436
- get_tangential_intersections() ..... 1436
- get_target_surfaces() ..... 1437
- get_target_timestep() ..... 1437
- get_tet_count() ..... 1437
- get_tet_global_element_id() ..... 1437
- get_tetmesh_growth_factor() ..... 1438
- get_tetmesh_insert_mid_nodes() ..... 1438
- get_tetmesh_minimize_interior_points() ..... 1438
- get_tetmesh_minimize_slivers() ..... 1438
- get_tetmesh_num_anisotropic_layers() ..... 1438
- get_tetmesh_optimization_level() ..... 1438
- get_tetmesh_optimize_mid_nodes() ..... 1438
- get_tetmesh_optimize_overconstrained_edges() ..... 1439
- get_tetmesh_optimize_overconstrained_tets() ..... 1439
- get_tetmesh_parallel() ..... 1439
- get_tetmesh_proximity_flag() ..... 1439
- get_tetmesh_proximity_layers() ..... 1439

- get_tetmesh_relax_surface_constraints() ..... 1439
- get_tight_bounding_box() ..... 1439
- get_top_level_assembly_items() ..... 1440
- get_total_bounding_box() ..... 1440
- get_total_volume() ..... 1440
- get_tri_count() ..... 1441
- get_tri_global_element_id() ..... 1441
- get_trimesh_geometry_sizing() ..... 1441
- get_trimesh_num_anisotropic_layers() ..... 1441
- get_trimesh_ridge_angle() ..... 1441
- get_trimesh_split_overconstrained_edges() ..... 1442
- get_trimesh_surface_gradation() ..... 1442
- get_trimesh_surface_proximity_ratio() ..... 1442
- get_trimesh_target_min_size() ..... 1442
- get_trimesh_tiny_edge_length() ..... 1442
- get_trimesh_use_surface_proximity() ..... 1442
- get_trimesh_volume_gradation() ..... 1442
- get_undo_enabled() ..... 1443
- get_unmerged_curves_on_shells() ..... 1443
- get_valence() ..... 1443
- get_valid_block_element_types() ..... 1443
- get_velocity_combine_type() ..... 1443
- get_velocity_dof_signs() ..... 1444
- get_velocity_dof_values() ..... 1444
- get_version() ..... 1444
- get_vertex_count() ..... 1444
- get_vertex_node() ..... 1444
- get_vertex_type() ..... 1445
- get_view_at() ..... 1445
- get_view_distance() ..... 1445
- get_view_from() ..... 1445
- get_view_up() ..... 1445
- get_vol_sphere_params() ..... 1445
- get_volume_area() ..... 1446

- get_volume_count() ..... 1446
- get_volume_element_count() ..... 1446
- get_volume_gap_solutions() ..... 1446
- get_volume_gaps() ..... 1447
- get_volume_hexes() ..... 1447
- get_volume_nodes() ..... 1448
- get_volume_pyramids() ..... 1448
- get_volume_tets() ..... 1448
- get_volume_volume() ..... 1448
- get_volume_wedges() ..... 1449
- get_volumes_for_node() ..... 1449
- get_wedge_count() ..... 1449
- get_wedge_global_element_id() ..... 1449
- get_wrt_entity() ..... 1449
- group_list() ..... 1450
- group_names_ids() ..... 1450
- has_valid_size() ..... 1450
- heatflux_is_on_shell_area() ..... 1450
- highlight() ..... 1451
- init() ..... 1451
- is_acis_engine_available() ..... 1451
- is_assembly_metadata_attached() ..... 1451
- is_blend_surface() ..... 1451
- is_boundary_layer_id_available() ..... 1451
- is_catia_engine_available() ..... 1451
- is_cavity_surface() ..... 1452
- is_chamfer_surface() ..... 1452
- is_close_loop_surface() ..... 1452
- is_command_echoed() ..... 1452
- is_command_journaled() ..... 1452
- is_cone_surface() ..... 1453
- is_continuous_surface() ..... 1453
- is_cylinder_surface() ..... 1453
- is_geometry_visibility_on() ..... 1453

- is_hole_surface() ..... 1453
- is_interval_count_odd() ..... 1454
- is_merged() ..... 1454
- is_mesh_element_in_group() ..... 1454
- is_mesh_visibility_on() ..... 1455
- is_meshed() ..... 1455
- is_modified() ..... 1455
- is_multi_volume() ..... 1455
- is_narrow_surface() ..... 1455
- is_occlusion_on() ..... 1456
- is_on_thin_shell() ..... 1456
- is_opencascade_engine_available() ..... 1456
- is_part_of_list() ..... 1456
- is_performing_undo() ..... 1456
- is_periodic() ..... 1457
- is_perspective_on() ..... 1457
- is_playback_paused() ..... 1457
- is_playback_paused_on_error() ..... 1457
- is_point_contained() ..... 1457
- is_scale_visibility_on() ..... 1458
- is_select_partial_on() ..... 1458
- is_sheet_body() ..... 1458
- is_surface_meshable() ..... 1458
- is_surface_planar() ..... 1458
- is_surface_planer() ..... 1458
- is_type_filtered() ..... 1459
- is_undo_save_needed() ..... 1459
- is_virtual() ..... 1459
- is_visible() ..... 1459
- is_volume_meshable() ..... 1460
- is_working_dir_set() ..... 1460
- journal_commands() ..... 1460
- load_ML() ..... 1460
- measure_between_entities() ..... 1460

- ML_train() ..... 1461
- move() ..... 1461
- number_undo_commands() ..... 1461
- override_journal_stream() ..... 1461
- parse_cubit_list() ..... 1462
- parse_locations() ..... 1462
- pause_playback() ..... 1462
- plugin_manager() ..... 1462
- print_cmd_options() ..... 1462
- print_current_selections() ..... 1462
- print_currently_selected_entity() ..... 1462
- print_info() ..... 1463
- print_raw_help() ..... 1463
- print_surface_summary_stats() ..... 1463
- print_volume_summary_stats() ..... 1463
- prism() ..... 1463
- process_input_files() ..... 1464
- project_unit_square() ..... 1464
- pyramid() ..... 1464
- reflect() ..... 1465
- release_interface() ..... 1465
- remove_entity_from_group() ..... 1465
- remove_filter_type() ..... 1465
- replace_progress_handler() ..... 1466
- report_usage() ..... 1466
- reset() ..... 1466
- reset_camera() ..... 1466
- resume_playback() ..... 1466
- scale() ..... 1466
- set_copy_block_on_geometry_copy_setting() ..... 1466
- set_copy_nodeset_on_geometry_copy_setting() ..... 1467
- set_copy_sideset_on_geometry_copy_setting() ..... 1467
- set_cubit_interrupt() ..... 1467
- set_cubit_message_handler() ..... 1467

- set_element_variable() ..... 1467
- set_entity_name() ..... 1467
- set_exit_handler() ..... 1468
- set_filter_types() ..... 1468
- set_label_type() ..... 1468
- set_max_group_id() ..... 1468
- set_ML_base_user_dir() ..... 1469
- set_modified() ..... 1469
- set_nodal_variable() ..... 1469
- set_overlap_max_angle() ..... 1469
- set_overlap_max_gap() ..... 1470
- set_overlap_min_gap() ..... 1470
- set_pick_type() ..... 1470
- set_playback_paused_on_error() ..... 1470
- set_progress_handler() ..... 1470
- set_rendering_mode() ..... 1470
- set_undo_saved() ..... 1471
- silent_cmd() ..... 1471
- snap_locations_to_geometry() ..... 1471
- sphere() ..... 1472
- step_next_possible_selection() ..... 1472
- step_previous_possible_selection() ..... 1472
- stop_playback() ..... 1472
- string_from_id_list() ..... 1472
- subtract() ..... 1473
- surface() ..... 1473
- sweep_curve() ..... 1473
- temperature_is_on_shell_area() ..... 1474
- temperature_is_on_solid() ..... 1474
- torus() ..... 1474
- tweak_curve_offset() ..... 1475
- tweak_curve_remove() ..... 1475
- tweak_surface_offset() ..... 1475
- tweak_surface_remove() ..... 1476

◆ tweak_vertex_fillet() ..... 1476
◆ unite() ..... 1476
◆ unload_ML() ..... 1477
◆ unselect_entity() ..... 1477
◆ vertex() ..... 1477
◆ volume() ..... 1477
◆ volume_contains_tets() ..... 1478
◆ was_last_cmd_undoable() ..... 1478
◆ write_to_journal() ..... 1478
Variable Documentation ..... 1478
◆ CI_ERROR ..... 1478
Public Member Functions ..... 1479
Protected Attributes ..... 1479
Detailed Description ..... 1479
Constructor & Destructor Documentation ..... 1479
◆ ~Entity() ..... 1479
◆ Entity() [1/3] ..... 1480
◆ Entity() [2/3] ..... 1480
◆ Entity() [3/3] ..... 1480
Member Function Documentation ..... 1480
◆ bounding_box() ..... 1480
◆ center_point() ..... 1480
◆ destroy_cubit_entity() ..... 1480
◆ entity_ptr() ..... 1480
◆ id() ..... 1481
◆ operator=() ..... 1481
Member Data Documentation ..... 1481
◆ cubitWatcher ..... 1481
◆ mEntityPtr ..... 1481
Public Member Functions ..... 1481
Protected Member Functions ..... 1484
Additional Inherited Members ..... 1484
Detailed Description ..... 1484
Constructor & Destructor Documentation ..... 1484

◆ GeomEntity() [1/2] ..... 1484
◆ GeomEntity() [2/2] ..... 1484
Member Function Documentation ..... 1485
◆ bodies() ..... 1485
◆ curves() ..... 1485
◆ dimension() ..... 1485
◆ entity_name() [1/2] ..... 1485
◆ entity_name() [2/2] ..... 1485
◆ entity_names() ..... 1486
◆ is_meshed() ..... 1486
◆ is_transparent() [1/2] ..... 1486
◆ is_transparent() [2/2] ..... 1486
◆ is_visible() [1/2] ..... 1487
◆ is_visible() [2/2] ..... 1487
◆ mesh() ..... 1487
◆ num_names() ..... 1487
◆ remove_entity_name() ..... 1487
◆ remove_entity_names() ..... 1487
◆ remove_mesh() ..... 1488
◆ set_entity_name() ..... 1488
◆ set_transparent() ..... 1488
◆ set_visible() ..... 1488
◆ smooth() ..... 1488
◆ surfaces() ..... 1489
◆ vertices() ..... 1489
◆ volumes() ..... 1489
Public Member Functions ..... 1490
Additional Inherited Members ..... 1493
Detailed Description ..... 1493
Constructor & Destructor Documentation ..... 1493
◆ Body() [1/3] ..... 1493
◆ Body() [2/3] ..... 1493
◆ ~Body() ..... 1493
◆ Body() [3/3] ..... 1494

Member Function Documentation ..... 1494
◆ get_mass_props() ..... 1494
◆ is_sheet_body() ..... 1494
◆ point_containment() ..... 1494
◆ volume() ..... 1495
Public Member Functions ..... 1495
Additional Inherited Members ..... 1499
Detailed Description ..... 1499
Constructor & Destructor Documentation ..... 1499
◆ Volume() [1/3] ..... 1499
◆ ~Volume() ..... 1499
◆ Volume() [2/3] ..... 1499
◆ Volume() [3/3] ..... 1499
Member Function Documentation ..... 1499
◆ centroid() ..... 1499
◆ color() ..... 1500
◆ principal_axes() ..... 1500
◆ principal_moments() ..... 1500
◆ set_color() ..... 1501
◆ volume() ..... 1501
Public Member Functions ..... 1501
Additional Inherited Members ..... 1506
Detailed Description ..... 1506
Constructor & Destructor Documentation ..... 1506
◆ Surface() [1/3] ..... 1506
◆ ~Surface() ..... 1506
◆ Surface() [2/3] ..... 1506
◆ Surface() [3/3] ..... 1507
Member Function Documentation ..... 1507
◆ area() ..... 1507
◆ closest_point_along_vector() ..... 1507
◆ closest_point_trimmed() [1/2] ..... 1508
◆ closest_point_trimmed() [2/2] ..... 1508
◆ color() ..... 1508

◆ get_param_range_U() ..... 1508
◆ get_param_range_V() ..... 1509
◆ is_cylindrical() ..... 1509
◆ is_planar() ..... 1509
◆ normal_at() ..... 1510
◆ ordered_loops() ..... 1510
◆ point_containment() ..... 1510
◆ position_from_u_v() ..... 1511
◆ principal_curvatures() ..... 1511
◆ set_color() ..... 1512
◆ u_v_from_position() ..... 1512
Public Member Functions ..... 1513
Additional Inherited Members ..... 1517
Detailed Description ..... 1517
Constructor & Destructor Documentation ..... 1517
◆ Curve() [1/3] ..... 1518
◆ ~Curve() ..... 1518
◆ Curve() [2/3] ..... 1518
◆ Curve() [3/3] ..... 1518
Member Function Documentation ..... 1518
◆ closest_point() ..... 1518
◆ closest_point_trimmed() ..... 1518
◆ color() ..... 1519
◆ curvature() ..... 1519
◆ curve_center() ..... 1519
◆ end_param() ..... 1519
◆ fraction_from_arc_length() ..... 1520
◆ is_periodic() ..... 1520
◆ length() ..... 1520
◆ length_from_u() ..... 1521
◆ point_from_arc_length() ..... 1521
◆ position_from_fraction() ..... 1522
◆ position_from_u() ..... 1522
◆ set_color() ..... 1522

◆ start_param() ..... 1523
◆ tangent() ..... 1523
◆ u_from_arc_length() ..... 1523
◆ u_from_position() ..... 1524
Public Member Functions ..... 1524
Additional Inherited Members ..... 1528
Detailed Description ..... 1528
Constructor & Destructor Documentation ..... 1528
◆ Vertex() [1/3] ..... 1528
◆ ~Vertex() ..... 1528
◆ Vertex() [2/3] ..... 1528
◆ Vertex() [3/3] ..... 1528
Member Function Documentation ..... 1528
◆ color() ..... 1528
◆ coordinates() ..... 1529
◆ operator=() ..... 1529
◆ set_color() ..... 1529
Public Member Functions ..... 1530
Detailed Description ..... 1530
Member Function Documentation ..... 1530
◆ what() ..... 1530
Public Member Functions ..... 1530
Detailed Description ..... 1530
Member Function Documentation ..... 1530
◆ what() ..... 1530
Public Member Functions ..... 1531
Detailed Description ..... 1531
Member Function Documentation ..... 1531
◆ what() ..... 1531
Public Member Functions ..... 1531
Detailed Description ..... 1532
Constructor & Destructor Documentation ..... 1532
◆ MeshErrorFeedback() ..... 1532
◆ ~MeshErrorFeedback() ..... 1532

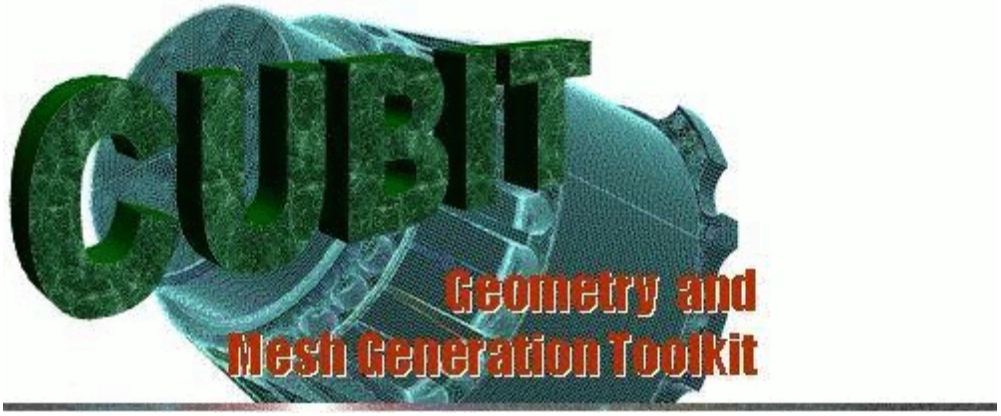
Member Function Documentation ..... 1532
◆ get_entity_id() ..... 1532
◆ get_entity_type() ..... 1532
◆ get_error_code() ..... 1532
◆ get_error_text() ..... 1532
◆ set_entity_id() ..... 1532
◆ set_entity_type() ..... 1533
◆ set_error_code() ..... 1533
◆ set_error_text() ..... 1533
Functions ..... 1533
Detailed Description ..... 1533
Function Documentation ..... 1534
◆ get_elem_connectivity() ..... 1534
◆ get_mesh_data_size() ..... 1534
◆ get_node_coordinates() ..... 1535
◆ get_side_and_centroid_data() ..... 1535
◆ init_mesh_access() ..... 1535
Functions ..... 1535
Detailed Description ..... 1536
Function Documentation ..... 1537
◆ clear_window() ..... 1537
◆ create_free_nodes() ..... 1537
◆ create_free_tris() ..... 1537
◆ draw_line() ..... 1537
◆ draw_point() ..... 1537
◆ flush() ..... 1538
◆ graphics_transforms() ..... 1538
◆ init() ..... 1538
◆ notify_mesh_modified() ..... 1538
◆ set_node_coordinates() ..... 1538
◆ transfer_mesh() ..... 1538
Public Member Functions ..... 1539
Public Attributes ..... 1540
Detailed Description ..... 1540

Constructor & Destructor Documentation ..... 1540
◆ AssemblyItem() ..... 1540
◆ ~AssemblyItem() ..... 1540
Member Function Documentation ..... 1540
◆ get_id() ..... 1540
◆ get_instance() ..... 1540
◆ get_level() ..... 1540
◆ get_name() ..... 1540
◆ get_path() ..... 1541
◆ get_type() ..... 1541
◆ set_id() ..... 1541
◆ set_instance() ..... 1541
◆ set_level() ..... 1541
◆ set_name() ..... 1541
◆ set_path() ..... 1541
◆ set_type() ..... 1541
Member Data Documentation ..... 1541
◆ volume_id_list ..... 1541
Public Member Functions ..... 1541
Detailed Description ..... 1542
Constructor & Destructor Documentation ..... 1542
◆ CFD_BC_Entity() ..... 1542
◆ ~CFD_BC_Entity() ..... 1542
Member Function Documentation ..... 1542
◆ get_id() ..... 1542
◆ get_name() ..... 1542
◆ get_type() ..... 1542
◆ set_id() ..... 1542
◆ set_name() ..... 1542
◆ set_type() ..... 1542
Public Member Functions ..... 1543
Public Attributes ..... 1544
Detailed Description ..... 1544
Constructor & Destructor Documentation ..... 1544

◆ Dir() ..... 1545
◆ ~Dir() ..... 1545
Member Function Documentation ..... 1545
◆ angle() ..... 1545
◆ cross() ..... 1545
◆ dir_print() ..... 1545
◆ distance() ..... 1545
◆ dot() ..... 1545
◆ get_xyz() ..... 1545
◆ length() ..... 1545
◆ normalize() ..... 1546
◆ orthogonal_vectors() ..... 1546
◆ set() ..... 1546
◆ set_x() ..... 1546
◆ set_y() ..... 1546
◆ set_z() ..... 1546
◆ x() ..... 1546
◆ y() ..... 1547
◆ z() ..... 1547
Member Data Documentation ..... 1547
◆ xVal ..... 1547
◆ yVal ..... 1547
◆ zVal ..... 1547
Public Member Functions ..... 1547
Public Attributes ..... 1548
Detailed Description ..... 1548
Constructor & Destructor Documentation ..... 1548
◆ Loc() ..... 1548
◆ ~Loc() ..... 1548
Member Data Documentation ..... 1548
◆ xVal ..... 1548
◆ yVal ..... 1548
◆ zVal ..... 1548
Public Attributes ..... 1548

Member Data Documentation ..... 1549
◆ Centroid ..... 1549
◆ ParentId ..... 1549
◆ ParentSide ..... 1549
Navigation XML Files ..... 1549
Periodic Space Filling Models (Tile) ..... 1551
Initial setup ..... 1551
Creating Nodesets ..... 1552
Smoothing ..... 1552
Example ..... 1553
References ..... 1554
Available Colors ..... 1557
Element Numbering ..... 1561
Node Numbering ..... 1561
Side Numbering ..... 1561
Triangular Shell Element Numbering ..... 1562
FASTQ ..... 1563
FullHex vs. NodeHex Representation ..... 1567
Generating a Finite Element Mesh from Level-set Data ..... 1567
Hex Mesh Generation Using Sculpt ..... 1567
Tet Mesh Generation by Remeshing Mesh Based Geometry (MBG) ..... 1569
Tet Mesh Generation Using Level-set Triangulation ..... 1570
Credits ..... 1573
Credits ..... 1573
Quick Reference ..... 1575
Quick Reference ..... 1575
Index ..... 1581

SAND Number: SAND2024-07256W






# CUBIT® 16.18 User Documentation

**Introduction** - A quick overview of some of the main features and goals of the CUBIT Mesh Generation Toolkit, licensing and distribution, hardware requirements, and where to go for help.

**Environment Control** - A description of the CUBIT user environment, including using the graphical user interface, session control, command line syntax, journal files, graphics, entity picking, saving and restoring etc..

**Geometry** - A description of CUBIT's geometry features including building geometry from scratch, manipulating geometry in CUBIT, importing and exporting geometry formats, etc...

**Mesh Generation** - A description of CUBIT's mesh generation capabilities, including how to mesh geometry, meshing and smoothing schemes, setting sizes and intervals, importing a mesh, etc...

**Finite Element Model** - How to set up the finite element model for analysis, including defining boundary conditions, material properties, exporting the finite element model, etc.

**Boundary Layer Meshing** - How to set up boundary layers.

**Immersive Topology Environment for Meshing (ITEM)** - A description of Cubit's interactive meshing wizard including how to use the wizard, and a guide to geometry clean-up, setting up the finite element model, mesh generation in ITEM, etc.

**Step-By-Step Tutorials**

**Appendix**

**Credits**

**Quick Reference**

**Official CUBIT Web Page**

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND Number: SAND2024-07256W

# CUBIT®

Geometry and Mesh Generation Toolkit
16.18 User Documentation

Michael Skroch, Steven J. Owen, Matthew L. Staten, Roshan W. Quadros, Byron Hanks, Brett Clark, Trevor Hensley, Corey Ernst, Randy Morris, Corey McBride, Clinton Stimpson, James Downer, Mark Richardson, Karl Merkley

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories



U.S. DEPARTMENT OF ENERGY



# Introduction

## Introduction


- Key Features
- Hardware Requirements
- Licensing and Distribution
- Trademark Notice
- How to Use this Manual
- Cubit Mailing Lists
- Problem Reports and Enhancement Requests


Welcome to CUBIT, the Sandia National Laboratory automated mesh generation toolkit. CUBIT is a full-featured software toolkit for robust generation of two- and three-dimensional finite element meshes (grids) and geometry preparation. Its main goal is to reduce the time to generate meshes, particularly large hex meshes of complicated, interlocking assemblies. It is a solid-modeler based preprocessor that meshes volumes and surfaces for finite element analysis. Mesh generation algorithms include quadrilateral and triangular paving, 2D and 3D mapping, hex sweeping and multi-sweeping, tetrahedral meshing, and various special purpose primitives. CUBIT contains many algorithms for controlling and automating much of the meshing process, such as automatic scheme selection, interval matching, sweep grouping, and also includes state-of-the-art smoothing algorithms

The CUBIT environment is designed to provide the user with a powerful toolkit of meshing algorithms that require varying degrees of input to produce a complete finite element model. Many CUBIT users want to experiment with capabilities as soon as possible. Hence, CUBIT releases often contain algorithms which are not quite ready for production use. These features are listed in the Appendix, and are accessible to the user by specifying a developer flag.

The overall goal of the CUBIT project is to reduce the time it takes a person to generate an analysis model. Generating meshes for complex, solid model-based geometries requires a variety of tools. Many CUBIT tools are completely automatic, while others require user input. Usually, the automatic choices can be over-ridden by the user if necessary. Most meshing capabilities are integrated into the common CUBIT framework; there are also stand-alone tools like Verde. The user is encouraged to become familiar with all of the available tools, so that he can choose the right one for the job.

## CUBIT Mailing Lists

### CUBIT® users mailing list

We maintain a mailing list intended for our users to discuss CUBIT® issues and to use for a help resource. It also carries announcements that we feel are of interest to our user community. To subscribe to or unsubscribe from this list, send an email to cubit-support@sandia.gov from your organizational email address inserting **subscribe cubit** or **unsubscribe cubit** in the subject line.

To send mail to the list, use cubit@sandia.gov

### CUBIT® Announcements mailing list

We also maintain a mailing list that will **only** broadcast announcements we feel will be of interest to our users. If you are already subscribed to our cubit@sandia.gov mailing list, you do NOT need to subscribe to the cubit-announce@sandia.gov list, as announcements will be echoed on it also. To subscribe to or unsubscribe from the announcements-only mailing list, send an email to cubit-support@sandia.gov from your organizational email address inserting **subscribe cubit-announce** or **unsubscribe cubit-announce** in the subject line.

# Hardware Requirements

Cubit is available on the following platforms:

- Red Hat 7, 64 bit (or similar system with at least glibc 2.17 and libstdc++ 4.8)
- Windows 10, 64 bit
- Mac 10.14+, 64 bit only

The Graphical User Interface version is available on all platforms.
For best results, local displays supporting OpenGL 3.2 or newer is recommended.

# How to Use This Manual

This manual provides specific information about the commands and features of CUBIT. It is divided into chapters, which roughly follow the process in which a finite element model is created, from geometry creation to mesh generation to boundary condition application. Examples are provided in the tutorial chapter. Appendices contain advanced topics, alpha commands, summary of APREPRO functions, FASTQ reference, a troubleshooting guide, and references.

| | |
|---|---|
|  | Integrated in CUBIT are algorithms and tools, which are in a user-beware state. As they are further tested (often with the assistance of users) and improved, the tool becomes more stable and production-worthy. Since documentation of the tool is necessary for actual use, we have included the documentation of all available tools. However, a "hammer" icon is placed next to some capabilities as a warning. |
|  | Certain portions of this manual contain information that is vital for understanding and effectively using CUBIT. These portions are highlighted with a "key" icon. |

# Key Features

### Geometry Creation, Modification, and Healing

CUBIT usually relies on the ACIS solid modeling kernel for geometry representation; there is also mesh-based geometry. Other solid model kernels are planned. Geometry is imported or created within CUBIT. Geometry is created bottom-up or through primitives.

CUBIT can also read STEP, IGES, and FASTQ files and convert them to the ACIS kernel. SolidWorks, AutoCAD, and some other commercial CAD systems can write SAT files directly.
Once in CUBIT, an ACIS model is modified through Booleans, or tweaking curves and surfaces. Without changing the geometric definition of the model, the topology of the model may be changed using virtual geometry. For example, virtual geometry can be used to composite two surfaces together, erasing the curve dividing them.
Sometimes, an ACIS model is poorly defined. This often happens with translated models. The model can be healed inside CUBIT.

## Non-Manifold Topology

Typical assembly meshes require contiguous mesh across multiple parts in an assembly. CUBIT accomplishes this by taking the two touching surfaces of neighboring volumes, and merging them into a single surface. There will be only one mesh of the surface, and both volume meshes will share that surface mesh. (In contrast, some meshing packages keep two surfaces, and take steps to ensure their mesh connectivity and positions match.) These shared surfaces are called non-manifold topology. Geometric models are usually imported into CUBIT as manifold (non-shared) models; then, surfaces which pass a geometric and topological comparison are "merged". A similar technique is used to merge model edges and vertices across parts. These comparisons are performed automatically, and can optionally be restricted to subsets of the model (to allow representations of such features as slide lines).

## Geometry Decomposition

Solid models often require decomposition to make them amenable to hexahedral meshing. CUBIT contains a wide variety of tools for interactive geometry decomposition, and a capability for performing automatic geometry decomposition is also under development.

## Mesh Generation

CUBIT contains a variety of tools for generating meshes in one, two and three dimensions. While the primary focus of CUBIT is on generating unstructured quadrilateral and hexahedral meshes, algorithms are also available for structured mesh generation and triangle/tetrahedral mesh generation. Several algorithms for generating mixed hex-tet meshes are also being developed.

## Boundary Conditions

CUBIT uses different boundary conditions for EXODUS-II format and Non-Exodus formats such as ABAQUS, for importing and exporting mesh data. EXODUS represents boundary conditions on meshes using Element Blocks, Nodesets, and Sidesets. Element Blocks are used to group elements by material type. Nodesets are used to group nodes. Other analysis programs can apply nodal boundary conditions to these sets, such as enforced displacement or nodal temperature values. Sidesets are used to group sides of elements, such as faces of hexes or edges of quads. Other analysis programs can apply face-based and edge-based boundary conditions to these sets, for example pressure or heat flux.
Using Element Blocks, Nodesets and Sidesets, a mesh and boundary conditions can be specified in an analysis-independent manner. Typically this specification is combined with an additional data file which designates the specific type of boundary condition (temperature, displacement, pressure, etc.), along with boundary condition values.

Non-Exodus export formats such as Abaqus support more specific boundary condition sets. These sets may include displacements, temperatures, forces, heatflux, pressure, or contact pairs.

### Element Types

CUBIT supports a wide variety of element types, including 1d, 2d, and 3d elements of various orders. Each block has a unique element type. The element type is specified after the block is created, and after mesh generation (recommended). Higher order nodes are generated when the element type is specified. Higher order nodes are projected to curved geometry, depending on the user-settable node constraint flag.

### Graphics Display Capabilities

CUBIT uses the VTK package for its graphics and rendering engine. CUBIT can display geometric and mesh entities in several modes, including hidden line, shaded, transparent or wireframe modes. CUBIT supports screen picking of geometric and mesh entities, as well as mouse-controlled view transformations like rotate, pan, and zoom. VTK takes advantage of hardware acceleration on most supported platforms. Image files of any displayed image can also be generated. CUBIT can also be run without graphics, to allow execution in batch mode or over slow network connections.

### Graphical User Interface

A full graphical user interface (GUI) with the standard look and feel consistent with major platforms is available on all supported Cubit platforms. The GUI version can improve productivity, making new users aware of the wide range of CUBIT capabilities, and freeing new and experienced users from having to remember esoteric syntax. The GUI and non-GUI versions create and play back identical journal files, making it easier to switch from one environment to the other.

### Command Line Interface

In the command line interface, commands are specified by text rather than mouse clicks. Commands can be entered interactively or in batch mode by playing back a journal file. The command line interface is available in the GUI through a window. The non-GUI version supports graphical picking and echoing to the command line, and also mouse-driven view transformations, but no menus and dialog boxes. The command line and GUI dialog boxes support the APREPRO preprocessor, which allows parameterization of input. The non-GUI version is available on all platforms, including Windows.

# Licensing, Distribution and Installation

Please refer to https://cubit.sandia.gov/licensing for information on licensing and distribution.

# Problem Reports and Enhancement Requests

CUBIT bugs, problem reports and enhancement requests should be reported via the Help Desk Portal or sent to cubit-help@sandia.gov or cubit-dev@sandia.gov. The CUBIT production meshing team or development team will review the email quickly. Users should expect some type of response within two days. Bugs are usually entered by a developer into CUBIT's bug tracking system.

# Trademark Notice

ACIS™ is a proprietary format developed by Spatial Corporation.
Granite is a proprietary format developed by Parametric Technology Corporation
All other trademarks are the property of their respective owners.

# Environment Control

## Environment Control


- Session Control
- Graphical User Interface
- Command Recording and Playback
- Graphics Window Control
- Entity Selection and Filtering
- Location, Direction, and Axis Specification
- Listing Information


The CUBIT user interface is designed to fill multiple meshing needs throughout the design to analysis process. The user interface options include a full graphical user interface, a modern command line interface as well as no-graphics and batch mode operation. This chapter covers the interface options as well as the use of journal files, control of the graphics, a description of methods for obtaining model information, and an overview of the help facility.

## session control

Session Control


- Starting and Exiting a CUBIT Session
- Execution Command Syntax
- Initialization Files
- Environment Variables
- Command Syntax
- Command Line Help
- Environment Commands
- Saving and Restoring a CUBIT Session
- Interrupting Running Tasks


This section provides an overview to session control in CUBIT. This includes information on starting and exiting a CUBIT session, running CUBIT in batch mode, initialization files, how to enter commands, file manipulation, changing the working directory, memory manipulation and more. Much of your ability to use CUBIT effectively depends on mastery of concepts in this section. Even experienced users will find it useful to review this section periodically.

### Command Line Help

In addition to the documentation you are currently viewing, CUBIT can give help on command syntax from the command line. For help on a particular command or keyword, the user can simply type **help <keyword>**. If the user is uncertain of the keyword, an asterisk ***** may be added to the end of the entered characters and help for all keywords

that start with the entered characters will be printed. In addition, if the user has typed part of a command and is uncertain of the syntax of the remainder of the command, they can type a question mark **?** and help will be printed for the sequence of keywords currently entered. It is important to note that if the user has typed the keywords out of order, then no help will be found. If the user is not sure of the correct order of the keywords, the ampersand **&** key will search on all occurrences of whatever keywords are entered, regardless of the order. The results of this type of command are shown in the following listing.

**CUBIT>help degenerate***
**Help for words: degenerate*.**
**set Block Mixed Element Output { OFFSET | Degenerate }**
**set Degenerates [on|off]**
**CUBIT> volume 3 label ?**
**Completing commands starting with: volume, label.**
**Help not found for the specified word order.**
**CUBIT> volume 3 label &**
**Help for words: volume & label**
**Label Volume [ on | off | name [only|id] | id | interval | size | scheme | merge | firmness ]**
**CUBIT> label volume 3 ?**
**Completing commands starting with: label, volume.**
**Label Volume [on|off|name [only|ids]|ids|interval|size|scheme|merge|firmness]**

## Command Syntax

The execution of CUBIT is controlled either by entering commands from the command line or by reading them in from a journal file. The commands can be either Cubit commands or Python statements. Similarly, a journal file may contain both Cubit commands and Python statements. By default, a command is interpreted as a Cubit command when starting the processing of a journal file or at the command line. A **#!python** statement can be given to indicate the subsequent commands are to be interpreted as Python statements. A **#!cubit** will similarly indicate the subsequent commands are to be interpreted as Cubit commands.

Throughout this document, each function or process will have a description of the corresponding CUBIT command; in this section, general conventions for command syntax will be described. The user can obtain a quick guide to proper command format by issuing the **<keyword> help** command; see Command Line Help for details.

CUBIT commands are described in this manual and in the help output using the following conventions. An example of a typical CUBIT command is:

**Volume <range> Scheme Sweep [Source [Surface] <range>] [Target [Surface] <range>] [Rotate {on | OFF}]**

The commands recognized by CUBIT are free-format and abide by the following syntax conventions.

1. Case is not significant.

2. The "#" character in any command line begins a comment. The "#" and any characters following it on the same line are ignored. Although note that the "#" character can also be used to start an Aprepro statement. See the Aprepro documentation for more information.
3. Commands may be abbreviated as long as enough characters are used to distinguish it from other commands.
4. The meaning and type of command parameters depend on the keyword. Some parameters used in CUBIT commands are:

   **Numeric**: A numeric parameter may be a real number or an integer. A real number may be in any legal C or FORTRAN numeric format (for example, 1, 0.2, -1e-2). An integer parameter may be in any legal decimal integer format (for example, 1, 100, 1000, but not 1.5, 1.0, 0x1F).
   **String**: A string parameter is a literal character string contained within single or double quotes. For example, 'This is a string'.
   **Filename**: When a command requires a filename, the filename must be enclosed in single or double quotes. If no path is specified, the file is understood to be in the current working directory. After entering a portion of a filename, typing a '?' will complete the filename, or as much of the filename as possible if there is more than one possible match.
   A filename parameter must specify a legal filename on the system on which CUBIT is running. The filename may be specified using either a relative path (**../cubit/mesh.jou**), a fully-qualified path (**/home/jdoe/cubit/mesh.jou**), or no path; in the latter case, the file must be in the working directory (See Environment Commands for details.) Environment variables and aliases may also be used in the filename specification; for example, the C-Shell shorthand of referring to a file relative to the user's login directory (**~jdoe/cubit/mesh.jou**) is valid.
   **Toggle**: Some commands require a "toggle" keyword to enable or disable a setting or option. Valid toggle keywords are "**on**", "**yes**", and "**true**" to enable the option; and "**off**", "**no**", and "**false**" to disable the option.

5. Each command typically has either:

   * an action keyword or "verb" followed by a variable number of parameters. For example:

   ```
   Mesh Volume 1
   ```

   Here **Mesh** is the verb and **Volume 1** is the parameter.
   * or a selector keyword or "noun" followed by a name and value of an attribute of the entity indicated. For example:

   ```
   Volume 1 Scheme Sweep Source 1 Target 2
   ```

   Here **Volume 1** is the noun, **Scheme** is the attribute, and the remaining data are parameters to the **Scheme** keyword.

The notation conventions used in the command descriptions in this document are:

- The command will be shown in a format that **looks like this**:
- A word enclosed in angle brackets ( **<parameter>** ) signifies a user-specified parameter. The value can be an integer, a range of integers, a real number, a string, or a string denoting a filename or toggle. The valid value types should be evident from the command or the command description.
- A series of words delimited by a vertical bar ( **choice1 | choice2 | choice3** ) signifies a choice between the parameters listed.
- A toggle parameter listed in **ALL CAPS** signifies the default setting.
- A word that is not enclosed in any brackets, or is enclosed in curly brackets ( **{required}** ) signifies required input.
- A word enclosed in square brackets ( **[optional]** ) signifies optional input which can be entered to modify the default behavior of the command.
- A curly bracket that is inside a square bracket (e.g. **[Rotate {on|OFF}]** ) is only required if that optional modifier is used.

## Environment Commands


- Working Directory
- File Manipulation
- CPU Time
- Comment
- History
- Error Logging
- Determining the CUBIT Version
- Echoing Commands
- Digits Displayed


*Working Directory*

The working directory is the current directory where journal files are saved. To list the current directory type

```
pwd
```

The current path will be echoed to the screen. By default, the current directory is the directory from which CUBIT was launched. The command to change the current directory is

```
cd "<new_path>"
```

The new path may be an absolute reference, or relative to the current directory. The <TAB> key will complete unique file references.

*File Manipulation*

A helpful addition is the ability to do a directory listing of a directory. The command for this is

**ls ['<file_name>']**

**or**

**dir ['<file_name>']**

Note also that you can delete files from the command line. The command for this is

**Delete File ['<file_name>']**

The file name may include the wildcard character *, but not the wildcard character ?, since the ? is used for command completion. File deletion from the command line can also be disabled. If deletions are set to **off** files cannot be deleted from the cubit command line.

**Set Deletions [ON|Off]**

The **mkdir** command is used to create a new directory. The syntax for this command is:

**Mkdir "<directory_name>"**

This creates a new directory with the specified name and path. The command accepts an absolute path, a relative path, or no path. If a relative path is specified, it is relative to the current working directory, which can be seen by typing 'pwd' at the cubit command prompt. If no path is specified, the new directory is created in the current working directory.
The command succeeds if the specified directory was successfully created, or if the specified directory already exists. The command fails if the new directory's immediate parent directory does not exist or is not a directory.

*CPU Time*

At times it is important to see how much cpu time is being used by a command. One function available to do this is the timer command. The syntax for this command is:

**Timer [Start|Stop]**

The start option will start a CPU timer that will continue until the stop command is issued. The elapsed time will be printed out on the command line. If no arguments are given, the command will act like a toggle.

*Comment*

This keyword allows you to add comments without affecting the behavior of CUBIT.

**Comment ['<text_to_print>'] [<aprepro_var>] [<numeric_value>]**

The comment command can take multiple arguments. If an argument is an unquoted word, it is treated as an aprepro variable and its value is printed out. Quoted strings are printed verbatim, and numbers are printed as they would be in a journal string. For example:

```
CUBIT> #{x=5}
CUBIT> #{s="my string"}
CUBIT> comment "x is" x "and s is" s

User Comment: x is 5 and s is my string

Journaled Command: comment "x is" x "and s is" s
```

*History*

This command allows you to display a listing of your previous commands.

**History <number_of_lines>**

For example, if you type history 10, the most recent 10 commands will be echoed to the input window.

*Error Logging*

**[set] Logging Errors {Off | On File '<filename>'[Resume]}**

This setting will allow users to echo error messages to a separate log file. The resume option will allow output to be appended to existing files instead of overwriting them. For more information on CUBIT environment settings see List Cubit Environment.

*Determining the CUBIT Version*

To determine information on version numbers, enter the command **Version**. This command reports the CUBIT version number, the date and time the executable was compiled, and the version numbers of the ACIS solid modeler and the VTK library linked into the executable. This information is useful when discussing available capabilities or software problems with CUBIT developers.

*Echoing Commands*

By default, commands entered by the user will be echoed to the terminal. The echo of commands is controlled with the command:

**[Set] Echo {On | Off}**

*Digits Displayed*

CUBIT uses all available precision internally, but by default will only print out a certain number of digits in order for columns to line up nicely. The user can override that with the "set digits" command:

**Set Digits [<num_to_list=-1>]**

If the digits are set to -1, then the default number of digits for pretty formatting are used. If the digits are set to a specific number, such as 15, more digits of accuracy can be displayed. This may be useful when checking the exact position and size of geometric features.
The number of digits used for listing positions, vectors and lengths can be listed using the following command:

**List Digits**

**Examples:**
**CUBIT> set digits 6**
Coordinates and lengths will be listed with up to 6 digits.
**CUBIT> set digits 20**
For this platform, max digits = 15. Coordinates and lengths will be listed with up to 15 digits.
**CUBIT> set digits -1**
To reset digits to default, use 'set digits -1'
The number of coordinate and length digits listed will vary depending on the context.

## Environment Variables

CUBIT can interpret the following environment variables. These settings are only applicable to the Command Line Version of CUBIT and do not apply to the Graphical User Interface. See also the CUBIT_STEP_PATH and CUBIT_IGES_PATH environment variables. See also the CUBIT_DIR, HOMEDRIVE and HOMEPATH settings.

| | |
|---|---|
| **DISPLAY** | The graphics window or GUI will pop-up on the specified X-Window display. This is useful for running CUBIT across a network, or on a machine with more than one monitor. Unix only. |
| **CUBIT_OPT** | Execution command line parameter options. Any option that is valid from the command line may be used in this environment variable. See Execution Command Syntax. |

| | |
|---|---|
| **CUBIT_Journal** | Specifies path and name to use for journal file. The specified path may contain the following %-escape sequences:<br>%a - abbreviated weekday name<br>%A - full weekday name<br>%b - abbreviated month name<br>%B - full month name<br>%d - date of the month [01,31]<br>%H - hour (24-hour clock) [00,23]<br>%I - hour (12-hour clock) [01,12]<br>%j - day of the year [1,366]<br>%m - month number [1,12]<br>%M - minute [00,59]<br>%n - replaced with the next available number between 01 and 999.<br>%p - "a.m." or "p.m."<br>%S - seconds [00,61]<br>%u - weekday [1,7], 1 is Monday<br>%U - week of year [00,53]<br>%w - weekday [0,6], 0 is Sunday<br>%y - year without century [00,99]<br>%Y - year with century (e.g. 1999)<br>%% - a '%' character<br>The default value is "cubit%n.jou". This creates journal files in the current directory named "cubit00.jou", "cubit01.jou", "cubit02.jou", etc. To keep the same naming scheme but create the files the /tmp directory, set CUBIT_JOURNAL to "/tmp/cubit%n.jou"<br>To create journal files in directories according to the day of the week, first create directories named "Mon", "Tues", etc. CUBIT will not create them for you. Next set CUBIT_JOURNAL to "%a/%n.jou". This will create journal files named "01.jou" through "999.jou" in the appropriate directory for the current day of the week. |

## Execution Command Syntax

To run CUBIT from the command line:

**cubit [options and args] [journalFile(s)]**

**claro [options and args] [journalFile(s)]**

**Cubit** is the command line version of Cubit and **Claro** is the GUI version of Cubit.
Command options for the command line are:

**cubit**
**-help** (Print this summary)
**-Include <$val>** (Specify a journal file)
**-workingdir <$val>** (Directory to use as working directory)
**-input $val** (Playback commands in file $val)
**-solidmodel <$val>** (Read .sat, .cub or .exo from file $val)
**-nogeom** (Read .exo from -solidmodel $val without creating geometry)
**-lite** (Read .exo from -solidmodel $val in lite mode)
**-fastq <$val>** (Read FASTQ file $val)

**-initfile <$val>** (Read $val as initialization file instead of $HOME/.cubit)
**-batch** (Batch Mode - No Interactive Command Input)
**-nographics** (Do not display graphics windows)
**-nogui** (Do not display graphical user interface)
**-noinitfile** (Do not read .cubit file)
**-noecho** (Do not echo commands to console)
**-nojournal** (Do not write journal file)
**-nodeletions** (Do not allow file deletions)
**-journalfile <$val>** (Name of journal file, will be overwritten)
**-restore [$val]** (Name of restore file (default = cubit_geom.save.sat))
**-maxjournal [$val]** (Maximum number of journal files to write)
**-warning [$val]** (Warning Messages On/Off)
**-information [$val]** (Informational Messages On/Off)
**-debug <$val>** (Set specified flags on, e.g. 1,3,7-9 enables 1,3,7,8,9))
**-display <$val>** (Specify display to be used for graphics window)
**-driver <$val>** (Specify the type of driver to be used for graphics display)
**-nooverwritecheck** (Do not perform file export overwrite check)
**-nobanner** (Suppress printing of startup information)
**-version** (Prints version information)
**-log <$val>** (Copy all output to specified file)
**-python_version <version>** (The major version of python to use: 2 or 3
**APREPRO variable pair** (Quoted name value pair)

Each of these is optional. If specified, the quantities in square brackets, **[$val]**, are optional and the quantities in angle brackets, **<$val>**, are required.
Options are summarized in more detail below:

| | |
|---|---|
| **-help** | Print a short usage summary of the command syntax to the terminal and exit. |
| | |
| **-workingdir** | Set the working directory to be used at startup. Journal files will be written to this directory. |
| **-initfile <$val>** | Use the file specified by **<$val>** as the initialization file instead of the default set of initialization files. See Initialization Files |
| **-noinitfile** | Do not read any initialization file. This overrides the default behavior described in Initialization Files |
| **-solidmodel <$val>** | Read the ACIS solid model geometry or .cub file information from the file |

| | |
|---|---|
| | specified by **<$val>** prior to prompting for interactive input. |
| **-batch** | Specify that there will be no interactive input in this execution of CUBIT. CUBIT will terminate after reading the initialization file, the geometry file, and the **input_file_list.** |
| **-nographics** | Run CUBIT without graphics. This is generally used with the -batch option or when running CUBIT over a line terminal. |
| **-nogui** | Run CUBIT without the graphical user interface. |
| **-display** | Sets the location where the CUBIT graphics system will be displayed, analogous to the **-display** environment variable for the X Windows system. Unix only. |
| **-driver <type>** | Sets the **<type>** of graphics display driver to be used. Available drivers depend on platform, hardware, and system installation. Typical drivers include *X11* and *OpenGL*. |
| **-nojournal** | Do not create a journal file for this execution of CUBIT. This option performs the same function as the Journal Off command. The default behavior is to create a new journal file for every execution of CUBIT. |
| **-journalfile <file>** | Write the journal entries to **<file>**. The file will be overwritten if it already exists. |
| **-maxjournal <$val>** | Only create a maximum of **<$val>** default journal files. Default journal files are of the form cubit#.jou where # is a number in the range 01 to 999. |
| **-nodeletions** | Turn off the ability to delete files with the delete file '<filename>' command. |
| **-nooverwritecheck** | Turn off the file overwrite check flag. Files that are written may then overwrite (erase) old files with the same name with no warning. This is typically useful when re-running journal files, in order to overwrite existing output files. See the |

| | |
|---|---|
| | **set File Overwrite Check [ON\|off]** command. |
| **-restore** | Restore the specified filename (or "cubit_geom") mesh and ACIS files, e.g. cubit_geom.save.g and cubit_geom.save.sat. |
| **-noecho** | Do not echo commands to the console. This option performs the same function as the Echo Off command. The default behavior is to echo commands to the console. |
| **-debug <$val>** | Set to "on" the debug message flags indicated by **<$val>**, where **<$val>** is a comma-separated list of integers or ranges of integers, e.g. 1,3,8-10. |
| **-information {on\|off}** | Turn **{on\|off}** the printing of information messages from CUBIT to the console. |
| **-warning {on\|off}** | Turn **{on\|off}** the printing of warning messages from CUBIT to the console. |
| **-Include <path>** | Allows the user to specify a journal file from the command line. |
| **-fastq <file>** | Read the mesh and geometry definition data in the FASTQ file **<file>** and interpret the data as FASTQ commands. See T. D. Blacker, FASTQ Users Manual Version 1.2, SAND88-1326, Sandia National Laboratories, (1988). for a description of the FASTQ file format. |
| **<input_file_list>** | Input files to be read and executed by CUBIT. Files are processed in the order listed, and afterwards interactive command input can be entered (unless the -batch option is used.) |
| **-log <file>** | Copies all output to the specified file. |
| **-python_version <version>** | The major version of python to use: 2 or 3. |
| **<variable=value>** | APREPRO variable-value pairs to be used in the CUBIT session. Values can be either doubles or character type (character values must be surrounded by double quotes.). Command options can |

| | |
|---|---|
| | also be specified using the **CUBIT_OPT** environment variable. (See Environment Variables .) |

*Passing Variables into a CUBIT Session*

To pass an aprepro variable into a CUBIT Session, start cubit with the variable defined in quotes i.e. cubit **"some_var=2.3"**

## Initialization Files

CUBIT can execute commands on startup, before interactive command input, through initialization files. This is useful if the user frequently uses the same settings.
The following files are played back in order, if they exist, at startup:

**$(cubit install directory)/.cubit.install**
**$HOME/.cubit**
**$(current working directory)/.cubit**

The **$(cubit install directory)** is determined by the location the program is installed. On Linux and Windows, it'll be the bin directory of the installation and on macOS it'll be the Cubit.app/Contents/MacOS directory.
**$HOME** is an environment variable pointing to the location of the user's home directory. On Windows, the **HOMEDRIVE** and **HOMEPATH** environment variables will be used instead of the **HOME** environment variable.
The **$(current working directory)** is determined by where the user starts the program itself.
If the **-initfile <filename>** option is used on the command that starts cubit, then the other init files are skipped and only the specified filename is played back.
These files are typically used to perform initialization commands that do not change from one execution to the next, such as turning off journal file output, specifying default mouse buttons, setting geometric and mesh entity colors, and setting the size of the graphics window.

## Interrupting Running Tasks

Many operations in the command line version of CUBIT can be interrupted using **<Control>-C**. Pressing **<Control>-C** will attempt to interrupt the running process as soon as feasible, returning the user back to the command line. Not all operations may be interrupted, and many can only be interrupted at certain stages. Any current tasks are canceled as soon as it is feasible to do so, including playback of journal files. The playback of a journal file is always stopped, even if the current running task cannot be interrupted. The journal file will stop at the next opportunity, when the current task is completed. Interrupted journal files may be resumed at the next command. See the section titled Controlling Playback of Journal Files for more information on controlling playback of journal files.
The **GUI** has a cancel button that can be used to interrupt the current command. The cancel button will turn red when a command can be interrupted. The cancel button has an 'x' on it, and is located on the status bar, which is at the bottom of the application.

## Saving and Restoring a Cubit Session

There are currently two ways to save/restore a model in CUBIT. A file can be saved with either the Exodus or CUBIT File method. The method of choice is determined by a set command. The CUBIT method is the default.

**Set Save [exodus|CUBIT] [Backups <number>]**

*CUBIT File Method*

- New
- Open
- Save
- Import
- Export

The CUBIT file is a binary cross-platform compatible file for the storage of a Cubit model that is compact in size and efficient to access. It includes both the geometry and the associated mesh, groups, blocks, sidesets, and nodesets. Mesh and geometry are restored from the Cubit file in exactly the same state as when saved. For example, element faces and edges are persistent, as well as mesh and geometry ids. The Graphical User Interface version of CUBIT also provides a toolbar with direct access to file operations using the CUBIT File method described here.

**New**

Creates a new blank model with default name, closing the current model. The New command essentially acts like the reset command.

**Open '<filename>'**

Opens an existing *.cub file, closing the current model.

**Save**

A default file name is assigned when CUBIT is started (in very much the same way the journal files are assigned on startup) in the form cubit01.cub, for example. The current model filename is displayed on the title bar of the CUBIT window. Typing save at any time during your session will save the current model to the assigned *.cub file. The *.cub file includes the *.sat file and the mesh. Groups, blocks, sidesets and nodesets are also saved within the *.cub file. To change the name of the current model, or to save the model's current geometry to a different file, use the save as command. Note that 'save <file.cub>' is NOT a valid command.

**Save**

**Save As 'filename.cub' [Overwrite]**

The set file overwrite command can be toggled on and off to allow overwriting when using the save as command. The command is defaulted to not allow overwriting.

**Set File Overwrite [On|OFF]**

A backup file is created by default, allowing access to previous states of the model. The backup files are named *.cub.1, *.cub.2... The user can set the total number of backups created per model with the following command (the default number of backups is 99,999):

**Set Save Backups <number>**

As soon as the number of model backups reaches the maximum, the lowest numbered backup file will be removed upon subsequent backup creation.
To check on the status of a 'set' command, type in the command in question without any options. For example, to check which save method is currently toggled, type:

**Set Save**

**Import**

Appends a *.cub file to an existing model.

**Import Cubit 'filename.cub' [merge_globally]**

**Export**

In addition to saving an entire model, one can use the export command to save only a portion of a model. The geometry and associated mesh, groups, blocks, sidesets and nodesets are exported. Only bodies or free surfaces, curves or vertices can be exported to a Cubit file.

**Export Cubit 'filename.cub' entity-list**

## Starting and Exiting a CUBIT Session

The following commands are used to control CUBIT execution.

*Starting the Session*

The command line version of CUBIT can be started on UNIX machines by typing "**cubit**" at the command prompt from within the CUBIT directory. If you have not yet installed CUBIT, instructions for doing so can be found in Licensing, Distribution and Installation. A CUBIT console window will appear which tells the user which CUBIT version is being run and the most recent revision date. A graphics window will also appear unless you are running with the **-nographics** option. For a complete list of startup options see the

Execution Command Syntax section of this document. CUBIT can also be run with initialization files or in batch mode.

*Windows File Association*

Windows users have the option to associate .cub, .sat, and .jou files with CUBIT. This means that double-clicking on one of these files will open it automatically in CUBIT. This option is available during the installation process

*Exiting the Session*

The CUBIT session can be discontinued with either of the following commands

**Exit**

**Quit**

*Resetting the Session*

A reset of CUBIT will clear the CUBIT database of the current geometry and mesh model, allowing the user to begin a new session without exiting CUBIT. This is accomplished with the command

**Reset [Genesis | Block | Nodeset | Sideset | QA_Records]**

A subset of portions of the CUBIT database to be reset can be designated using the qualifiers listed. Advanced options controlled with the Set command are not reset.
QA Records are stored in exodus, genesis, or cub files. If your file contains an excessive amount of qa records and you don't need them, it is beneficial to reset them for faster file I/O.
You can also reset the number of errors in the current Cubit session, using the command

**Reset Errors <value>**

which will set the error count to the specified value, or zero if the value is left blank.

*Abort Handling*

In the event of a crash, Cubit will attempt to save the current mesh as "crashbackup.cub" in the current working directory just before it exits.

To disable saving of the crashbackup.cub file set an environment variable **CUBIT_NO_CRASHSAVE** equal to true. Or, use the following command:

**Set Crash Save [On|Off]**

This command will turn on or off crashbackup.cub creation during a crash on a per-instance basis. To minimize the effects of unexpected aborts, use Cubit's automatic journaling feature, and remember to save your model often.

# recording and playback

## Command Recording and Playback

Sequences of CUBIT commands can be recorded and used as a means to control CUBIT from ASCII text files. Command or "journal" files can be created within CUBIT, or can be created and edited directly by the user outside CUBIT.

- Journal File Creation & Playback
- Controlling Playback of Journal Files
- Automatic Journal File Creation
- IDless Journal Files

## Automatic Journal File Creation

*Controlling Automatic Journal File Creation*

By default, CUBIT automatically creates a journal file each time it is executed. The file is created in the current directory, and its name begins with the word "cubit", followed by a number starting with cubit01.jou and continuing up to a maximum of cubit999.jou. It is recommended that the user keep no more than around 100 journal files in any directory, to avoid using up disk space and causing confusion. To that end, when the journal name increments to more than cubit99.jou, a warning will be given on startup telling the user that there are at least 99 journal files, and to please clean out unused files. If the user has up through cubit999.jou, then the user is warned that there are too many journal files in the current directory, and cubit999.jou will be re-used, destroying the previous contents.
When starting cubit, the choice of journal file name will depend on the existence of other cubitXX.jou files. Cubit will fill in gaps, starting with the lowest number. For example, if there are already journal files with names cubit01.jou, cubit02, jou, and cubit04.jou, then Cubit will use cubit03.jou as the current journal file.
Journal file names end with a ".jou" extension, though this is not strictly required for user-generated journal files. If no journaling is desired, the user may start CUBIT with the -nojournal command line option or use the command :

**[Set] Journal {Off | On}**

Turning journaling back on resumes writing commands to the same journal file.
Most CUBIT commands entered during a session are journaled; the exceptions are commands that require interactive input (such as Zoom Cursor), some graphics related commands, and the **Playback** command.

*Recording Graphics Commands*

All graphics related commands may be enabled or disabled with the command:

**Journal Graphics {On | Off}**

The default is **Journal Graphics Off** .

*Recording Entity IDs and Names*

When an entity is specified in a command using its name, the command may be journaled using the entity name, or by using the corresponding entity type and id. The method used to journal commands using names is determined with the command:

**Journal Names {On | Off}**

The default is **Journal Names On** .
If an entity is referred to using its entity type and id, the command will be journaled with the entity type and id, even if the entity has been named.

*Recording APREPRO Commands*

APREPRO commands may be echoed to the journal file using the following command

**[set] Journal [Graphics|Names|Aprepro|Errors] [on|off]**

See APREPRO Journaling for more information.

*Recording Errors*

The default mode for CUBIT is to not journal any command that does not execute successfully. To turn this mode off and echo all commands to the journal file, regardless of the success status, use the following command:

**Journal Errors {On|OFF}**

If a command did not execute successfully and the journal errors status is ON, then the unsuccessful command will be written as a comment to the file. For example an unsuccessful command might look like the following in the journal file

**## create brick x 10 x 10 z 10**

Since CUBIT recognizes this as erroneous syntax, it will issue an error when the command is issued, but will still write the command to the journal file as a comment, prefixing the command with "##".
This option may be useful when tracking or documenting program errors.

## Controlling Playback of Journal Files

The following commands control the playback of Journal Files:

```
Stop
Pause
Sleep <duration_in_seconds>
Resume [<n>]
Where
Next [<n>]
```

The playback of a journal file can be interrupted in three ways. Pressing **ctrl-c** while the journal file is playing will halt playback of the journal file. (This only works in the command line version of CUBIT. See Interrupting Running Tasks for more information ). Alternately, if the **stop** or **pause** commands are encountered in the journal file and CUBIT is reading commands from a terminal (as opposed to a redirected file), playback of the journal file will halt after that command.

The **sleep** command pauses execution for the specified number of seconds. It can be used to build a delay into journal files during presentations.

In the command line version of CUBIT you can resume playback of a journal file with the **resume** command. If playback was interrupted because **ctrl-c** was pressed, it will resume at the next command after the one that was interrupted. If playback stopped because of a **stop** or **pause** command in the journal file, it will resume at the next line after the **stop** or **pause** command. If the file was paused because of a **sleep** command in the file, it will resume automatically after the specified duration.

If journal files that are playing back contain **playback** commands themselves, there may be multiple current journal files. The **where** lists all current journal files and where the journal files have paused. Each line contains the stack position (a number), the filename and the current line in the file. Unless CUBIT is running in batch mode, the first line is always **<stdin>**. This just means that CUBIT will return to the command prompt after the top-most journal file has completed.

The remaining portion of any active journal file may be skipped by specifying the stack position (first number on each line of the output from the **where** command) of the file where you want to resume. Any remaining commands in active journal files with lower stack positions will be skipped.

The **next** command steps through interrupted journal files line-by-line. The argument to the **next** command is the number of lines to read before halting playback again. If no number is specified, the command will advance one line.

Journal playback can also be set to stop automatically when it encounters an error during playback. The command syntax is:

```
Set Stop Error {On|OFF}
```

Setting the stop error to "on" will cause the file to halt for each error. The setting is turned off by default.

## Journal File Creation and Playback

*Recording a Session*

Command sequences can be written to a text file, either directly from CUBIT or using a text editor. CUBIT commands can be read directly from a file at any time during CUBIT execution, or can be used to run CUBIT in batch mode. To begin and end writing commands to a file from within CUBIT, use the command

**Record '<filename>'**

**Record Stop**

Once initiated, all commands are copied to this file after their successful execution in CUBIT.

*Replaying a Session*

To replay a journal file, issue the command

**Playback '<filename>'**

Journal files are most commonly created by recording commands from an interactive CUBIT session, but can also be created using automatic journaling or even by editing an ASCII text file.
Commands being read from a file can represent either the entire set of commands for a particular session, or can represent a subset of commands the user wishes to execute repeatedly.
Two other commands are useful for controlling playback of CUBIT commands from journal files. **Playback** from a journal file can be terminated by placing the **Stop** command after the last command to be executed; this causes CUBIT to stop reading commands from the current journal file. Playback can be paused using the Pause command; the user is prompted to hit a key, after which playback is resumed.
Journal files are most useful for running CUBIT in batch mode, often in combination with the parameterization available through the APREPRO capability in CUBIT. Journal files are also useful when a new finite element model is being built, by saving a set of initialization commands then iteratively testing different meshing strategies after playing that initialization file.

## Idless Journal Files

Journal files can also be created without reference to entity IDs. The purpose of this command is to enable journal files created in earlier versions of CUBIT to be played back in newer versions of CUBIT. Using the "IDless" method, commands entered with an entity ID will be journaled with an alternative way of referring to the entity. Changes in CUBIT or ACIS often lead to changes in entity IDs. For example, a webcut may result in volume 3 on the left and volume 4 on the right. In another version of CUBIT, those entity IDs may be swapped (4 on the left and 3 on the right). Playing an IDless journal file makes the actual ID of an entity irrelevant. The syntax for this command is:

**[set] Journal IDless {on|off|reverse}**

The **on** option will enable idless journaling, and commands will be journaled without entity IDs. For example, "mesh volume 1" may be journaled as "mesh volume at 3.42 5.66 6.32 ordinal 2".
Selecting the **off** option will cause commands to be journaled in the traditional manner (i.e., as they are entered).
The **reverse** option allows you to convert idless journal files back into an ID-based journal file where the new journal file will reflect current numbering standards for IDs.
If you issue the command **Journal IDless** without any additional options, then the current status of ID journaling is printed. At startup, this should be "off".
The most likely scenario for converting older journal is to use the record command during playback. The following is an example.

```
journal                                idless                                on
record                                                          "my_idless.jou"
playback                                                       "my_journal.jou"
record                                                                    stop
journal idless off
```

To record an *idless* journal file back into an *id-based* journal file you might use the following sequence.

```
journal                              idless                              reverse
record                                                        "new_id_based.jou"
playback                                                         "my_idless.jou"
record                                                                    stop
journal idless off
```

**Note**: IDless conversions of APREPRO expressions are partially supported.
When IDless mode is set to **ON**, APREPRO functions such as Vx(id), that take an ID as an argument, are converted to use (x, y, z, ord) as arguments such as Vx(x, y, z, ord), where (x, y, z) is the center point coordinates and ord is the ordinal value. The ordinal values, 1..n, identifies each entity in a set of n entities that have a common center point. An entity's ordinal value is based on its creation order with respect to the other entities within the same set.
When IDless mode is set to **REVERSE** (using the above example) Vx(x, y, z, ord) will be converted to Vx(id). Outside these APREPRO functions, APREPRO expressions are not modified when converting a journal file to or from its IDless form. Hence, expressions reduced to an entity ID, such as in the command "volume {x} size 10," are not modified. Therefore, when moving a journal file from one version of CUBIT to another, it may be necessary to manually update IDs in APREPRO expressions.

# location direction specification

## Location, Direction and Axis Specification


- Specifying a Location
- Specifying a Location on a Curve
- Specifying a Direction
- Specifying an Axis
- Specifying a Plane
- Drawing a Location, Direction, or Axis

Many commands require that a location or a direction be specified. Although entering the three floating point numbers required to uniquely define a vector is perfectly acceptable, it may be more convenient to specify the direction or location with respect to existing entities in the model.

For example, the following commands might be used for creating straight curves using location and direction specification described here:

**Create Curve [From] Location {options} Location {options}**

**Create Curve [From] Location {options} Direction {options} Length <val>**

## Drawing a Location, Direction, or Axis

Some commands require you to specify a location on a curve (i.e., webcutting with a plane normal to a curve). This location can be previewed with the following options:

1. A fraction along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
2. A distance along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
3. An xyz position that is moved to the closest point on the given curve.
4. The position of a vertex that is moved to the closest point on the given curve.

**Draw Location On Curve <curve id> {Fraction <f> | Distance <d> | Position <xval><yval><zval> | Close_To Vertex <vertex_id>} [[From] Vertex <vertex_id> (optional for 'Fraction' & 'Distance')]**

Some commands require a specified axis (such as webcut with a cylinder) and it is sometimes advantageous to view an axis before modifying geometry. To draw a preview of an axis use the following command:

**Draw Axis {options}**

Some commands require a specified location or point (such as create curve spline) and it is sometimes advantages to view a location before modifying or creating geometry. To draw a preview of a location use the following command:

**Draw Location {options} [color <color_name>] [no_flush]**

Some commands require a direction. To draw a preview of a direction, use the following command:

**Draw Direction {options} [from] [location {options}]**

Similar commands for drawing lines and polygons may also be useful.

## Specifying an Axis

Some commands require a specified axis (such as webcut with a cylinder) and it is sometimes advantageous to view an axis before modifying geometry. An axis is simply a vector with a specified origin. The following options determine an axis specification:

- Last
- Direction {options} [Origin [Location] {options}] [Length <val>] [Angle <val>]
- Surface <id>
- Curve <id>
- [Revolve [About] Axis {options} Angle <val>]]

*Last*

**Last**

The last option recalls the last axis used in an axis command. The last axis does not carry over from CUBIT session to CUBIT session.

*Specify a direction and a location*

**Direction {options} [Origin [Location] {options}] [Length <val>] [Angle <val>]**

To specify an axis simply specify a vector (a direction) and an origin (a location). Notice that the command requires the axis direction first because the origin defaults to 0 0 0 when not specified. An example of specifying an axis to draw a location using the swing command is as follows:

**draw location 1 0 0 swing about axis direction z ang 45**

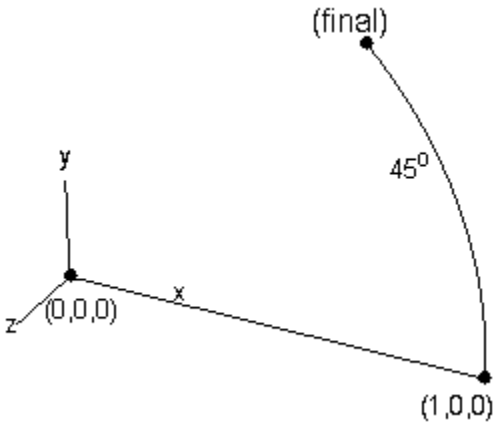



**Figure 1 - Swinging a point about the z-axis**

The location 1 0 0 was swung 45 degrees about an axis defined by a vector in the z direction and an origin at 0 0 0.

*Specify a surface*

**Surface <id>**

If a surface is specified, it must be a cone type surface. The axis of the cone surface is used. If a non-cone type surface is specified, an error will result.

*Specify a curve*

**Curve <id>**

If a curve is specified, it must be an arc, helix, or spline of constant curvature. All other curve types will result in an error.

*Option to revolve an axis about an axis*

**[Revolve [About] Axis {options} Angle <val>]**

To revolve one axis around another use the revolve keyword. The following example revolves the first axis (defined by the y-axis and origin) around the second axis (defined by the z-axis and origin) by 45 degrees and draws the result.

**draw axis direction y revolve axis direction z angle 45**

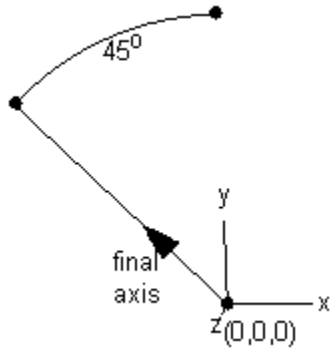



**Figure 2 - Revolving an axis about another axis**

*Previewing an Axis*

Sometimes it is helpful to preview an axis before using it in a command. An axis may be previewed using the Draw command. The options for previewing an axis are the same as the ones described above.

**Draw Axis {options}**

## Specifying a Direction

Some commands require a specified a direction or vector for the command. A direction is basically a xyz vector in the model. The following options determine a direction specification:

- [Vector] <xval yval zval>
- Last
- X|Y|Z|Nx|Ny|Nz
- [On] | [Tangent] [At] Curve <id> {location on curve options}
- [On] | [Normal] [At] Surface <id> [Location {options}]
- [From] { Location {options} | {Node|Vertex} <id> }[Project] {Location {options} | [Entity] {Node|Vertex|Curve|Surface} <id> }
- [Rotate {options}]
- [Cross [With] Direction {options}]
- [Reverse]

*Vector (XYZ values)*

**[Vector] <xval yval zval>**

The most basic way to specify a direction is to just give the vector x-y-z components of the direction. The given vector need not be a unit vector. The following three commands simply draw a direction in the x-direction (1, 0, 0) as the Vector keyword is optional and unit vectors are not required:

```
draw direction vector 1 0 0
draw direction 1 0 0
draw direction 10 0 0
```

*Last Direction Used*

**Last**

The last option recalls the last direction used in a command. For example, if the following command is entered after the above vector commands a direction location would be drawn in the x-direction (1, 0, 0).

```
draw direction last
```

Last directions do not carry over from CUBIT session to CUBIT session. The last direction defaults to (1, 0, 0) if no direction has been used during the session.

*Positive or Negative X,Y,Z Direction Vectors*

**X|Y|Z|Nx|Ny|Nz**

The x|y|z|nx|ny|nz options assign the x direction, y direction, z direction, negative x direction, negative y direction and negative z direction respectively.

*On Curve Tangent*

**[On] | [Tangent] [At] Curve <id> {location on curve options}**

The curve option simply finds a tangent vector on a curve. Note that the **on**, **tangent** and **at** keywords are optional, as well as the location on the curve. If no location is specified, the tangent at the start vertex of the curve is found. See the section above, Specifying a Location on a Curve, for details on how to specify where along the curve the tangent vector is found.

**draw direction curve 1**
**draw direction on curve 1**
**draw direction tangent at curve 1**
**draw direction tangent at curve 1 distance 3**
**draw direction tangent at curve 1 fraction .5**
**draw direction tangent at curve 1 distance 2 reverse**

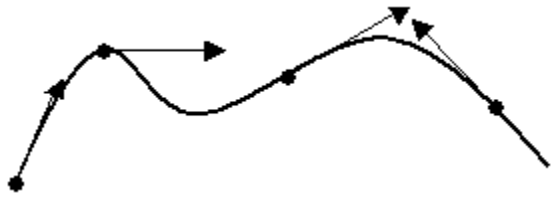


**Figure 1 - Tangents to a Curve**

*On Surface Normal*

**[On] | [Normal] [At] Surface <id> [Location {options}]**

The surface option simply finds a normal vector on a surface. Note that the "on", "normal" and "at" keywords are optional, as well as the location on the surface. If no location is specified, the normal vector at the center of the surface is found. If a location is specified, the location is projected to the surface, then the normal vector is found.

**draw direction on surface 1**
**draw direction on surface 1 location 1 2 0**

*From Location*

**[From] {Location {options} | Node|Vertex <id>} [Project] {Location {options} | [Entity]**
**{Node|Vertex|Curve|Surface} <id>}**

The from location option finds a direction that is from one location to another or from a location to an entity. If the second specification is an entity, the first location is projected to the entity to find the direction.

**draw direction from vertex 1 vertex 2**
**draw direction from location on curve 1 fraction .5 surface 3**

Note that when using an entity for the second specification, the Project and Entity keywords are generally optional. However, it is sometimes necessary to remove ambiguity from the previous location specification. For example, the following will not parse correctly:

**draw direction location on curve 1 distance 2 surface 3**

In this case, the location on the curve is parsed as a distance 2.0 from surface 3. Instead, the desired behavior is to find the location on curve 1 as a distance of 2.0 along the curve from the start of the curve, and project it to surface 3 to find the direction. The following commands (all equivalent) achieve this behavior:

**draw direction location on curve 1 distance 2 project surface 3**
**draw direction location on curve 1 distance 2 entity surface 3**
**draw direction location on curve 1 distance 2 project entity surface 3**

*Rotate*

**[Rotate {options}]**

The rotate option allows you to rotate the direction about another vector. You can string together as many rotations as necessary. For example:

**draw direction 1 0 0 rotate about z 135 rotate about curve 1 angle 50**

Options that can be used with rotate are as follows:

**{Ax|X|Ay|Y|Az|Z [Angle] <angle>} | { {[About] | Towards} Direction {options} Angle <val> } [Rotate (options)] [Origin (location)]**

Ax, Ay, Az (or X,Y,Z) angles can be entered in any order. The optional specification of another rotate keyword in the options indicated that multiple nested rotations are permitted.

*Cross*

**[Cross [With] Direction {options}]**

The cross option allows you to find the vector cross product of the direction with another direction.

*Reverse*

**[Reverse]**

This keyword simply reverses the direction specification.

*Previewing a Direction*

Sometimes it is helpful to preview a direction before using it in a command. A direction may be previewed using the Draw command. The direction options are described above. See Specifying a Location for a list of location options.

**Draw Direction {direction_options} [Location (location_options)]**

## Specifying a Location

Some commands require a specified location or point (such as create curve spline) for the command. A location is basically an x-y-z position in the model. The following options determine a location specification:

- [Position] <xval yval zval>
- Last
- [At] {Node|Vertex} <id_list>
- [On] Curve <id_list> [location on curve options]
- [On] Surface <id_list> [Close_To | At Location {options} | CENTER]
- [On] Plane <options> [Close_To | At Location {options}]
- Center Curve <id_list>
- Extrema {Curve|Surface|Volume|Body|Group} <range> [Direction] {options} [Direction {options}] [Direction {options}]
- Fire Ray Location {options} Direction {options} At {Body|Volume|Surface|Curve|Vertex} <ids> [Maximum Hits <val>] [Ray Radius <val>]
- Between { Location <options> Location <options>} | { Location <options> Project {Curve|Surface} <id> } [Stop] [Fraction <val>] }
- [Move [all] {<xval yval zval> | {Dx|X|Dy|Y|Dz|Z} <val> | Direction {options} Distance <val>} ]
- [Swing [all] [About] Axis {options} Angle <ang>]
- Multiple Location Specification

*Position (XYZ values)*

**[Position] <xval yval zval>**

The most basic way to specify a location is to just give the xyz values of the location. In this case the following two commands both draw a location at the coordinates (1, 2, 3), as the Position keyword is optional:

**draw location position 1 2 3**
**draw location 1 2 3**

*Last Location Used in a Command*

**Last**

The last option recalls the last location used in a command. For example, if the following command is entered after the above position commands a location would be drawn at the position (1, 2, 3).

**draw location last**

Last locations do not carry over from CUBIT session to CUBIT session. The last location defaults to (0, 0, 0) if no location has been used during the session.

*Node or Vertex*

**[At] {Node|Vertex} <id_list>**

Referring to a node or vertex simply returns the coordinates of that node or vertex. The command can also handle multiple locations where multiple locations are needed to complete the command string. The following draws a location at the coordinates of Vertex 5:

**draw location vertex 5**

*On a Curve*

Various options are available to specify a location on a curve. See the section Specifying a Location On a Curve for details.

*On a Surface*

**[On] Surface <id_list> [Close_To | At Location {options} | CENTER]**

If a surface is used to specify a location without other options, the geometrical center of the surface is found (the center keyword is optional - the default). Otherwise, you can specify another general location and that location is projected to the surface. For example,

the following command will draw the location that is position (5,0,0) projected to surface 1:

```
draw location on surface 1 location 5 0 0
```

Any valid location options listed on this page can be used to specify the location that is projected to the surface.

*On a Plane*

```
[On] Plane <options> [Close_To | At Location {options}]
```

A location can be defined at the closest point on a plane to a location. See Specifying a Plane for plane options.

*Center*

```
Center Curve <id_list>
```

Finds the center of an arc - an error is returned if the curve is not an arc.

*Extrema*

```
Extrema {Curve|Surface|Volume|Body|Group} <range> [Direction] {options}
[Direction {options}] [Direction {options}]
```

The extrema option returns the location of the maximum value, on the specified entity or group, in the specified direction. For example, the following places a vertex on a surface at the point of maximum y-axis value.

```
create vertex location extrema surf 1 direction y
```

*Fire Ray*

The **fire ray** command allows a user to identify a location, or set of locations, on an object by firing a ray at the object and determining the intersections. A ray can be fired at a list of bodies, volumes, surfaces, curves, or vertices. The fire ray command is:

```
Fire Ray Location {options} Direction {options} At
{Body|Volume|Surface|Curve|Vertex} <ids> [Maximum Hits <val>] [Ray Radius
<val>]
```

The location options are described on this page. The direction options are described under Specifying a Direction. The user can specify the maximum number of hits that he wishes to receive back from the command. If this value is omitted, the command will return all intersections found. When firing a ray at a curve, a ray radius must be used. The

ray radius is the distance from the curve the ray must be to be considered a "hit." If no ray radius is used, the geometry engine default is used.

*Between*

**Between {Location <options> Location <options> } | {Location <options> Project {Curve|Surface} <range>} [Stop] [Fraction <val>]}**

The between option finds a location that is between two locations or a location and an entity. An optional fraction can be given to specify the fractional distance from the first location to the second location or entity. For example, the following will draw a location at (5, 0, 0):

```
draw location between location 0 0 0 location 10 0 0
```

The following will draw a location at (2.5, 0, 0) - 25% of the distance from (0, 0, 0) to (10, 0, 0):

```
draw location between location 0 0 0 location 10 0 0 fraction .25
```

The second item can be an entity:

```
draw location between location 0 0 0 vertex 2
draw location between location 0 0 0 surface 1
```

In the second case, location (0, 0, 0) is projected to surface 1, then the location that is between (0, 0, 0) and the projected location is found.
Of course, any valid location can be used in the command. In the following example a location at the top center of the brick is found:

```
brick x 10
draw location between location bet vert 3 vert 2
location bet vert 8 vert 5
```

The first location is between vertices 3 and 2, and the second location is between vertices 8 and 5.
**Note:** you can "swing" a location about an axis, "rotate" a direction about another direction, "revolve" an axis about another axis and "spin" a plane about an axis. The only reason Cubit needs to use different keywords for each entity type is because the Cubit command language does not support expressions (as in using parentheses). The keyword **stop** is also used in the location/direction/axis/plane parsing as a partial workaround to this limitation. Using this stop keyword will aid in parsing out extended location specifications. Insert a stop after the first location to let the parser know that where the specifications begin and end.

*Move*

**Move [All] { <xval yval zval> | {Dx|X|Dy|Y|Dz|Z} <val> | Direction {options} Distance <val> }**

Any location can be optionally moved either a xyz distance or a certain distance in a given direction. As many moves as desired can be strung together. For example, the following will return a location at (5, 0, 0):

**draw location 0 0 0 move 5 0 0**

These examples add another move that basically moves the location (5, 0, 0) in a direction 45 degrees up and to the right a distance of 10 (all three commands are equivalent - see sections on directions and rotations):

**draw location 0 0 0 move 5 0 0 move {10*sind(45)} {10*sind(45)} 0**
**draw location 0 0 0 move 5 0 0 move direction 1 1 0 distance 10**
**draw location 0 0 0 move 5 0 0 move direction 1 0 0 rotate about 0 0 1 angle 45 dist 10**

*Swing*

**Swing [All] [About] Axis {options} Angle <ang>**

Any location can be "swung" (rotated) about an axis by a certain angle. (See the section on specifying an axis for the axis syntax). As with moves, multiple swings can be strung together. The following example rotates the location (2.5, 5, 5) thirty degrees about an axis defined by Curve 11. Note that the right-hand rule is used to determine the direction of the swing about the axis.

**draw location 2.5 5 5 swing about axis curve 11 angle 30**

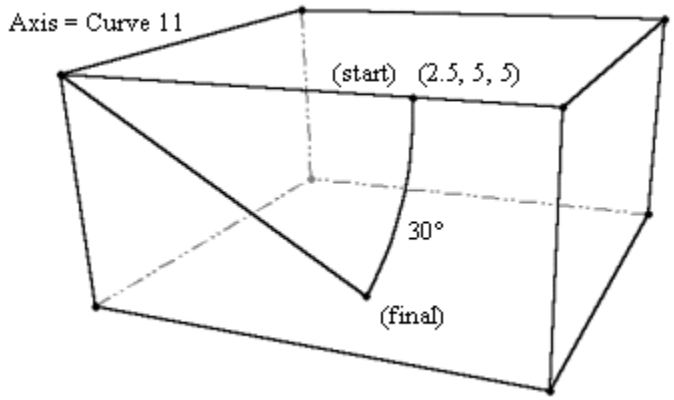



**Figure 1 - Swinging a Location**

*Multiple Location Specification*

**Location {options} Location {options}...**

Multiple location specifications can be used in a single command. For example, the following command uses several locations to create a spline curve at points (0,0,0), (1,2,3), (4,5,6), and (7,8,9).

**create curve spline location 0 0 0 location 1 2 3 location 4 5 6 location 7 8 9**

*Previewing a Location*

Sometimes it is advantageous to preview a location before using it in a command. A location can be previewed with the Draw command. All of the options that can be used to specify locations in a command can be used to preview locations as well. See above for a description of these options. The command syntax is:

**Draw Location {options}**

## Specifying a Location on a Curve

Some commands require you to specify a location on a curve (i.e., webcutting with a plane normal to a curve). The following are the options for specifying a location (or locations in the case of the segment option) on a curve:

- {MIDPOINT|Start|End}
- Center
- Fraction <val 0.0 to 1.0> [From Vertex <id> | Start|End]
- Distance <val> [From {Vertex|Curve|Surface} <id> | Start | End ]
- {{Close_To|At} Location {options} | Position <xval><yval><zval> | {Node|Vertex} <id>}
- Extrema [Direction] {options} [Direction {options}] [Direction {options}]
- Segment <num_segs>
- Crossing {Curve|Surface} <id_list> [Bounded|Near]}
- Previewing a Location

*Center*

**center**

The center option helps in identifying the location at the center of a given arc. Example: create vertex center curve 3

*Start, Midpoint, or End*

**{ MIDPOINT | Start | End |**

These options simply specify the location that is the midpoint, start or end point of a curve. By default, the midpoint is the understood location unless another location is specified.

*Fraction*

**Fraction <val 0.0 to 1.0> [From Vertex <id> | Start|End] |**

The fraction option simply finds the location that is a fractional distance along the curve. By default, the fraction references the start of the curve; however, you can optionally specify which vertex to reference from.

*Distance*

**Distance <d> [From {Vertex|Curve|Surface} <id> | Start | End ] |**

The distance option not only can find a location that is a certain distance along the curve from the start or end of the curve, but can also find a location (or locations if there is more than one solution) on a curve that is a specified distance from another curve or a surface. If the From Curve option is used both curves must lie in the same plane.

**draw location on curve 13 distance 7 from curve 2**

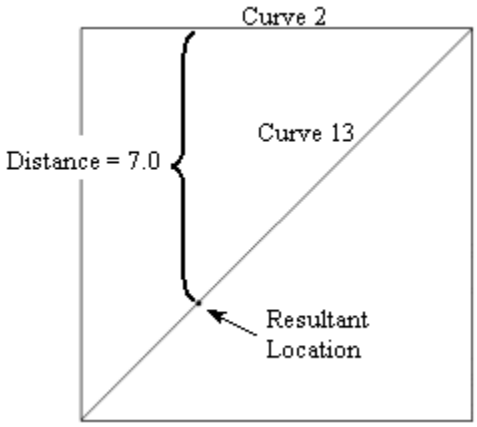



**Figure 1 - Location on a Curve a Distance from Another Curve**

*{Close_To|At} Location*

**{{Close_To|At} Location {options} | Position <xval><yval><zval>**
**|{Node|Vertex} <id>} |**

These options take a location closest to the location on the curve.

*Extrema*

**Extrema [Direction] {options} [Direction {options}] [Direction {options}]**

The extrema option finds the maximum value location along a curve in a specified direction. For example:

**create vertex location on curve 1 extrema ny**

Creates a vertex on curve 1 at the location where the y axis value of the curve is at a minimum.

*Segment*

**Segment <num_segs>**

The segment option finds locations spaced evenly along the curve such as to break the curve into equal length "segments" (of course the curve is not modified). You must specify a minimum of two segments (if two segments were specified a location would be found at the center of the curve). The following example results in 4 locations:

**draw location on curve 1 segment 5**

**create vertex on curve 1 segment 5**



**Figure 2 - Five Segments on a Curve**

*Crossing*

**Crossing {Curve|Surface} <id_list> [Bounded|Near]}**

The crossing option finds locations at the intersection of the curve and another curve or surface. By default, the curve(s) and surface are extended to infinity and the intersections are calculated; if the bounded option is specified only intersections that lie on the bounded entities will be returned. The near option is valid only for two linear curves. If near is specified the nearest location between the two linear curves will be returned.

*Previewing a Location on a Curve*

A location on a curve can be previewed with the Draw command. All of the options that can be used for specifying a location on a curve can be used to preview a location on a curve. See above for a description of these options. The command syntax is:

**Draw Location On Curve <curve id> {options}**

## Specifying a Plane

Some commands require a specified plane (such as sweep curve target) for the command. The following options determine a plane specification:

- {Location|Vertex|Node} <origin> Direction <normal>
- {Location|Vertex|Node} <origin> Direction <vec on plane> Direction <vec on plane>
- {Location|Vertex|Node} <2 locations> Direction <vector on the plane>
- {Location|Vertex|Node} <3 locations>
- Surface <id> [at location <loc>]
- [Normal To] Curve <id> [loc on curve options]
- Direction <Normal> Coefficient <val>
- Arc Curve <id>
- Linear Curve <id> <id>
- X|Xplane|Yz|Zy|Y|Yplane|Zx|Xz|Z|Zplane|Xy|Yx
- Last

The following options apply to all of the plane specifications listed above:

- [Offset <val>]
- [Move { <xval yval zval> | {Dx|X|Dy|Y|Dz|Z} <val> | Direction {options} [Distance <val>]]
- [[To] Location {options}]
- [Spin [About] Axis {options} Angle <ang>]]

*Location and Normal Vector*

**{Location|Vertex|Node} <origin> Direction <normal>**

The first way to specify a plane is to specify a starting point and a direction vector:

```
draw plane location 1 2 3 direction 0 1 1
draw plane vertex 1 direction tangent at curve 1
```

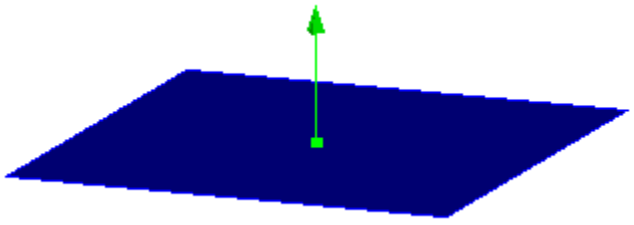


**Figure 1. Specifying a plane with a location and surface normal**

To see the options for location specification, see Specifying a Location. Direction options can be found at Specifying a Direction.

*Location and Two Vectors on the Plane*

**{Location|Vertex|Node} <origin> Direction <vec on plane> Direction <vec on plane>**

It is also possible to select an origin point and 2 direction vectors on the plane.

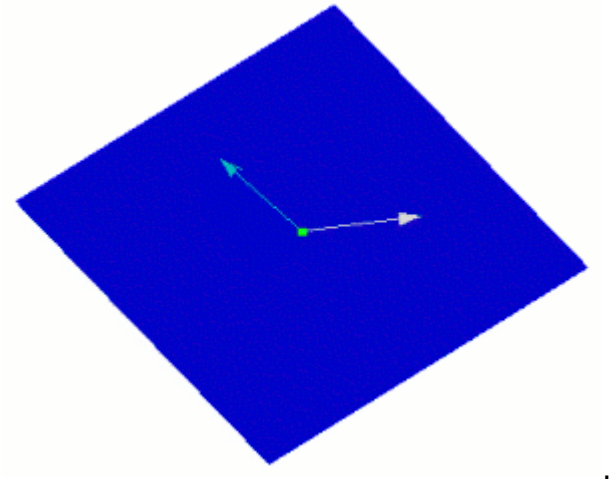


**Figure 2. Specifying a plane with a point and 2 in-plane vectors**

*Two Locations and Vector on the Plane*

**{Location|Vertex|Node} <2 locations> Direction <vector on the plane>**

You can also specify 2 locations and 1 direction on the plane to define the plane.

**draw plane vertex 1 2 direction 0 1 1**

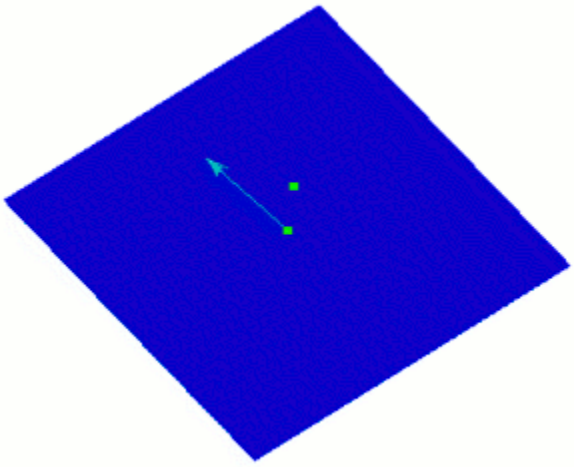


**Figure 3. Specifying 2 locations and 1 direction on the plane**

*Three Points on the Plane*

**{Location|Vertex|Node} <3 locations>**

A plane can be defined by three locations, vertices, or nodes. The locations are specified using Location Specification.

**draw plane vertex 1 2 3**
**draw plane vertex 1 2 location 3 4 5**

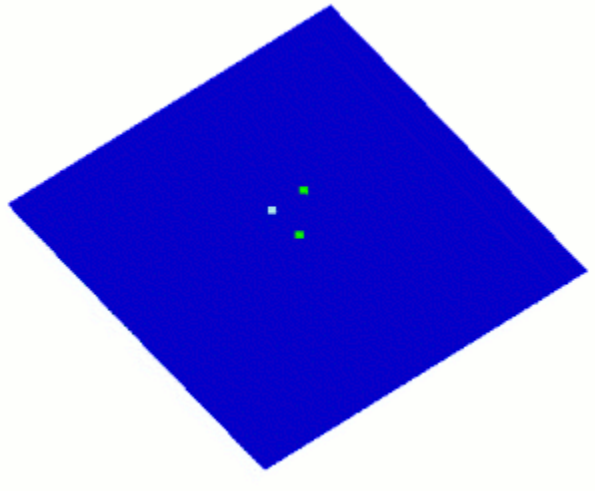


**Figure 4. A plane specified by three points**

*Plane defined by a Surface*

**Surface <id> [At Location <loc>]**

The surface option uses and existing surface to define the plane. If it is not a planar surface, the optional location specifier can be used to find the tangent plane of a specific point on the surface.

**draw plane surface 1 at location 4 0 0**

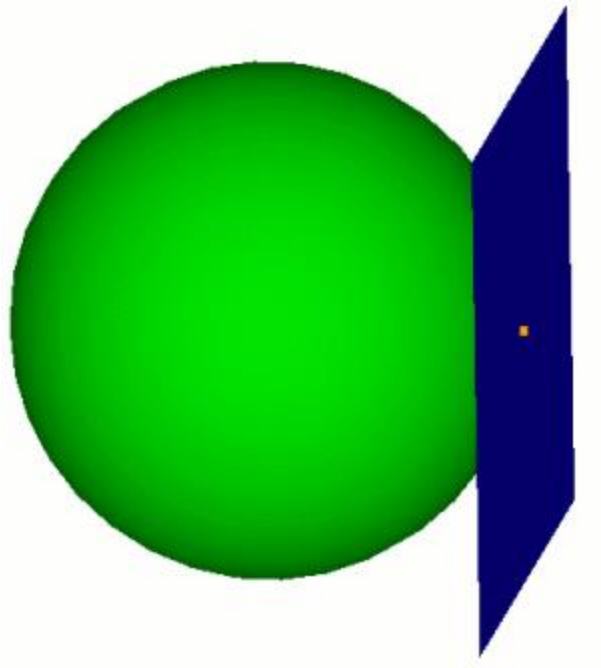


**Figure 5. Specifying a Tangent plane to a Surface**

*Plane Normal to a Curve*

**[Normal To] Curve <id> [loc on curve options]**

The Normal to Curve option allows you to define a plane by using an existing curve. The direction of the curve will define the surface normal of the new plane. The optional location argument specifies which point to use on the curve if it is not a straight curve. If no location is specified the plane will originate at the midpoint of the curve. See Specifying a Location on a Curve for more information on location options.

```
brick x 10
cylinder radius 3 z 12
subtract body 2 from 1
webcut body 1 xplane
draw plane normal to curve 30
```

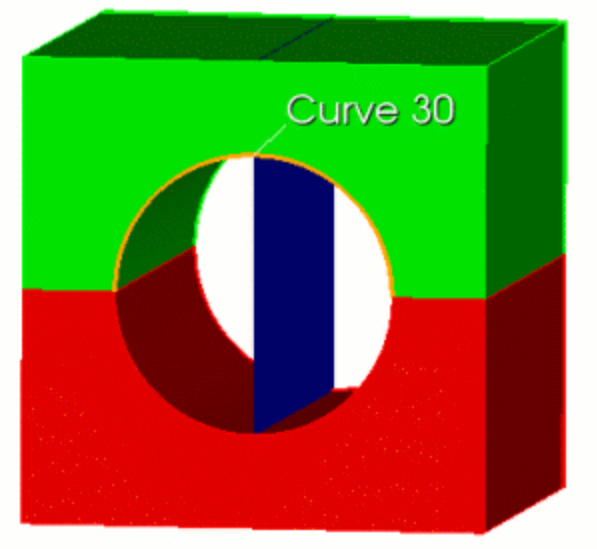



**Figure 6. Draw Plane Normal to Curve**

*Plane Defined by a Non-linear curve*

**Arc Curve <id>**

A plane can be defined by a single curve, provided that curve is not linear.

```
cylinder height 12 radius 3
draw plane arc curve 2
```

*Plane Defined by a two linear curves*

**Linear Curve <id> <id>**

A plane can be defined by a two linear curves, provided that the curves are not co-linear.

```
brick x 10
draw plane linear curve 2 3
```

*Normal Vector and Coefficient*

**Direction <Normal> Coefficient <val>**

The direction and coefficient option allows you to specify a plane based on a vector and an offset from the origin. The Coefficient argument specifies how far to offset the plane from the origin

```
draw plane direction 1 2 3 coefficient 3
```

*Coordinate Plane*

**X|Xplane|Yz|Zy|Y|Yplane|Zx|Xz|Z|Zplane|Xy|Yx**

A plane can be defined from any coordinate plane or combination thereof. The coordinate planes will pass through the origin unless optional specifiers are included.

**draw plane xplane**
**webcut volume 1 plane xz**

*Last Location Used*

**Last**

The last option will return the plane most recently used in a command. Last locations do not carry over from CUBIT session to CUBIT session. The last location defaults to (0, 0, 0) if no location has been used during the session.
The following options apply to all of the plane specification methods described above.

- **[Offset <val>]**
- **[Move {<xval yval zval>| {Dx|X|Dy|Y|Dz|Z} <val> | Direction {options} [Distance <val>]]**
- **[[To] Location {options}]**
- **[Spin [About] Axis {options} Angle <ang>]]**

A offset value will offset the plane in the direction of the surface normal.
The move option will displace the plane in the specified directions by the specified distance. The direction options are outlined on Specifying a Direction.
The location option will move the plane to a specified location without rotating it. See Specifying a Location for location options.
The spin option will rotate the plane around an axis. See Specifying an Axis for axis options.

*Previewing a Plane*

The ability to preview a plane prior to creating the plane or using it in a command is possible with the following commands:

**Draw Plane (options) [Graphics | {[Intersecting] {Body|Volume} <id_range>] [ [Extended] {Percentage|Absolute} <val>]}] [Color 'color_name']**

The options for specifying a plane are described above in the section on Plane Specification. By default, the commands draw the plane just large enough to intersect the bounding box of the entire model with minimum surface area. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the size of the surface by either a percentage distance or an absolute distance of the minimum area size. The default color is blue, but you can specify a different one. See the Appendix of the CUBIT Users Guide for available colors in CUBIT.

*Preview a Cylindrical Plane*

The ability to preview a cylindrical plane is possible with the following command:

**Draw Cylinder Radius <val> Axis {x|y|z|Vertex <id_1> Vertex <id_2> | <xyz values>} [Center <x_val> <y_val> <z_val>] [[Intersecting] Body <id_range>] [Extended Percentage|Absolute <val>] [Color 'color_name']**

The cylinder is defined by a radius and the cylinder axis. The axis is specified as a line corresponding to a coordinate axis, the normal to a specified surface, two arbitrary points, or an arbitrary point and the origin. The center point through which the cylinder axis passes can also be specified.
By default, the commands draw the cylinder just large enough to just intersect the bounding box of the entire model. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the length of the cylinder by either a percentage distance or an absolute distance of the cylinder length. The default color is blue, but you can specify a different one. See the Appendix of the CUBIT Users Guide for available colors in CUBIT.

# listing information

## Listing Information

The **List** commands print information about the current model and session. There are five general areas: *Model Summary, Geometry, Mesh, Special Entities,* and *CUBIT Environment*. The descriptions of these areas includes example output based on the model generated by a journal file listed below. The model consists of a 1x2x3 brick meshed with element size 0.1.

- List Model Summary
- List Geometry
- List Mesh
- List Special Entities
- List CUBIT Environment

**Journal File Used for List Examples**

```
brick x 1 y 2 z 3
body 1 size 0.1
mesh volume 1
block 1 volume 1
nodeset 1 surface 1
sideset 1 surface 2
group "my_surfaces" add surface 1 to 3
surface 2 name "BackSurface"
surface 3 name "BottomSurface"
surface 1 name "FrontSurface"
surface 4 name "LeftSurface"
surface 5 name "RightSurface"
surface 6 name "TopSurface"
```

## List Cubit Environment

The user may list information about the current CUBIT environment such as message output settings, memory usage, and graphics settings.

*Message Output Settings*

There are several major categories of CUBIT messages.

- **Info** (Information) messages tell the user about normal events, such as the id of a newly created body, or the completion of a meshing algorithm.
- **Warning** messages signal unusual events that are potential problems.
- **Error** messages signal either user error, such as syntax errors, or the failure of some operation, such as the failure to mesh a surface.
- **Echo** messages tell the user what was journaled.
- **Debug** messages tell developers about algorithm progress. There are many types of Debug messages, each one concentrating on a different aspect of CUBIT.

By default, Info, Warning, Error, and Echo messages are printed, and Debug messages are not printed. Information, Warning, Debug, and Echo message printing can be turned on or off (or toggled) with a set command; error messages are always printed. Debugging output can also be redirected to a file. Current message printing settings can be listed.

**List {Echo|Info|Errors|Warning|Debug}**

**Set Echo [On|Off]**

**Set {Info|Warning} [On|Off] [logging]**

**[Set] Debug <index> [On|Off]**

**[Set] Debug <index> File <'filename'>**

**[Set] Debug <index> Terminal**

Message flags can also be set using command line options:

**-warning {on|off}**

**-information {on|off}**

Debug flags can be enabled from the command line with

**-debug <setting>**

where **<setting>** is a comma-separated list of integers or ranges of integers denoting which flags to turn on. E.g., to set debug flags 1, 3, and 8 to 10 on, the syntax is -**debug 1,3,8-10**.

*Logging Output to a File*

Output from CUBIT can be redirected to a log file, and the current state of logging can be listed.

**[Set] Logging {Off|On File <'filename'> [Resume]}**

**List Logging**

If logging is enabled, by default any output to the console or command window will also go into the logging file. The **resume** option will append to the logfile, if it exists, instead of emptying the file. If the logfile doesn't already exist, it will be created.
Output of information and warning messages to the logging file can be controlled independent of console output settings by adding the **logging** option to the **set {info|warning} [on|off] logging** command.

*Default Block Creation*

**Set Default Block {ON|off|Volume|Surface|Curve]}**

**List Default Block**

The **set Default Block** command will toggle whether or not default blocks are written during the export operation if no other blocks have been specified. The **List Default Block** command lists the geometric entity types for which blocks will automatically be generated at export.

*Journaling Settings*

**List Journal**

The **List Journal** command lists which types of CUBIT commands will be journaled and the file to which the journaled commands are being written.

*Exodus Export Title*

**List Title**

**Title "<title_string>"**

The **List Title** command will list the title to be written to an Exodus file on export. To assign a title to an Exodus file, use the **Title** command.

*Listing Current Settings*

### List Settings

The **List Settings** command lists the value of all the message flags, journal file and echo settings, as well as additional information. The first section lists a short description of each debug flag and its current setting. Other message settings are listed next, followed by some flags affecting algorithm behavior.

**Sample output**

```
CUBIT> list settings
Debug Flag Settings (flag number, setting, output to, description):
 1  OFF  terminal           Debug Graphics toggle for some debug options.
 2  OFF  terminal           Whisker weaving information
 3  OFF  terminal           Timing information for 3D Meshing routines.
 4  OFF  terminal           Graphics Debugging (DrawingTool)
 5  OFF  terminal           FastQ debugging
 6  OFF  terminal           Submapping graphics debugging
 7  OFF  terminal           Knife progress whisker weaving information
 8  OFF  terminal           Mapping Face debug / Linear Programming debug
 9  OFF  terminal           Paver Debugging
.
.
.
echo               = On
info               = On
journal            = On
journal graphics   = Off
journal names      = On
journal aprepro    = On
journal file       = 'cubit11.jou'
warning            = On
logging            = Off
recording          = Off
keep invalid mesh = Off
default names      = Off
default block     = Volumes
catch interrupt    = On
name replacement character = '_', suffix character = '@'
Matching Intervals is fast, TRUE;
multiple curves will be fixed per iteration.
Note in rare cases 'slow', FALSE, may produce better meshes.
Match Intervals rounding is FALSE;
intervals will be rounded towards the user-specified intervals.
```

*Graphical Display Information*

### List View

**List view** prints the current graphics view and mode parameters; See Graphics Window .

*Memory Usage Information*

Users are encouraged to use Unix commands such as `top' to check total CUBIT memory use. Developers may check internal memory usage with the following command:

**List Memory [`<object type>']**

Without an object type, the command prints memory use for all types of objects.

## List Geometry

The following commands list information about the geometry of the model.

**List Names [Group|Body|Volume|Surface|Curve|Vertex|All]**
**List {Group|Body|Volume|Surface|Curve|Vertex} <range> [Ids]**
**List {geom_list} [Geometry|Mesh [Detail]]**
**List {Group|Body|Volume|Surface|Curve|Vertex} <range> {X|Y|Z}**

The first command lists the names in use, and the entity type and id corresponding to each name. Specifying **all** lists names for all types; other options list names for a specific entity type. The names for an individual entity can be obtained by listing just that entity. Sample output from the list names surface command is shown below. This output shows that, for example, Surface 2 has the name ` BackSurface '.

| Name | Type | Id | Propagated |
|---|---|---|---|
| BackSurface | Surface | 2 | No |
| BottomSurface | Surface | 3 | No |
| FrontSurface | Surface | 1 | No |
| LeftSurface | Surface | 4 | No |
| RightSurface | Surface | 5 | No |
| TopSurface | Surface | 6 | No |

**List Names Example**

The second command provides information on the number of entities in the model and their identification numbers. If a range is given then detailed information is given on each entity in that range, unless the **ids** option is also given. If the **ids** option is used, just a list of ids is printed. This list can be very useful for large models in which several geometry decomposition operations have performed. Sample output from the list surface command is shown below.

CUBIT> list surface ids
The 6 surface ids are 1 to 6.
CUBIT> list surf ids
The 108 surface ids are 192 to 266, 268 to 271, 273 to 301.

**List Surface [range] Ids' Examples**

The **<range>** can be very general using the general entity parsing syntax. Using a **<range>** gives a brief synopsis of the local connectivity of the model, e.g. one can list the ids of the surfaces containing vertex 2; as shown in the listing below.. An intermediately detailed synopsis can be obtained by placing the range of entities in a group, then listing the group.

CUBIT> list surface in vertex 2 ids
The 3 entity ids are 1, 5, 6.

```
CUBIT> group "v2_surfs" equals surface in vertex 2
CUBIT> list v2_surfs Group Entity 'v2_surfs' (Id = 3)
It owns/encloses 3 entities: 3 surfaces.
Owned Entities: Mesh Scheme Interval: Edge
_____Name____ Type_____Id +is meshed Count Size
FrontSurface Surface 1 map+ 1 H 0.1
 TopSurface Surface 6 map+ 1 H 0.1
RightSurface Surface 5 map+ 1 H 0.1
```

**Using 'List' for Querying Connectivity.**

The third command provides detailed information for each of the specific entities. This information includes the entity's name and id, its meshing scheme and how that scheme was selected, whether it is meshed and other meshing parameters such as smooth scheme, interval size and count. The entity's connectivity is summarized by a table of the entity's subentities and a list of the entity's superentities. Also, the nodesets, sidesets, blocks, and groups containing the entity are listed.
Specifying **geometry** will additionally list the extent of the entity's geometric bounding box, the geometric size of the entity, and depending on entity type, other information such as surface normal. See also the **list {entities} x** command below. If multiple volumes, surfaces, or curves are selected, it will list the total volume, area, or length of all entities, and the total geometric bounding box. If multiple volumes are selected, the centroid listed will be the composite centroid of the all of the volumes.
If a volume is recognized as a primitive (cylinder, brick, etc.), specifying **geometry** will list data for defining the primitive.

- Cylnder data is RCC: followed by 7 doubles: x,y,z coordinates of bottom center, x,y,z components for vector from bottom center to top center, and radius.
- Brick data is BOX: followed by 12 doubles: x,y,z coordinates of first corner, and 3 sets of x,y,z components for vectors from the first corner to each adjacent corner of the brick.

Specifying **mesh** will additionally list the number of mesh entities of each type interior to the entity and on bounding subentities. **Mesh detail** will list the ids of the mesh entities as well, following the format of the **list ids** command above.
The fourth command lists the entities sorted by either the x, y, or z coordinate of their geometric center. For example, in a large, basically cylindrical model centered around z-axis, it is useful to list the surfaces of a volume sorted by z to identify the source and target sweeping surfaces.

## List Mesh

The following commands list mesh entity information.

**List {Hex|Face|Edge|Node} <id_range>**

**List {Hex|Face|Edge|Node} <id_range> IDs**

For both of these commands, the range can be very general, following the general entity parsing syntax. The first command provides detailed information. For an entity, the information includes its id, owning geometry, subentities and superentities. For a hex, the Exodus Id is also listed. For a node, its coordinates are listed. The second command just lists the entity ids, and is usually used in conjunction with complex ranges.

## List Model Summary

The following commands print identical summaries of the model: the number of entities of each geometric, mesh, and special type

**List Model**

**List Totals**

The following output is generated from the **list model** command.

```
CUBIT> list model
Model                                   Entity                    Totals:
    Geometric                                                     Entities:
        0                                                         assemblies
        0                                                         parts
        2                                                         groups
        1                                                         bodies
        1                                                         volumes
        6                                                         surfaces
       12                                                         curves
        8                                                         vertices
    Mesh                                                          Entities:
      6000                                                        hexes
        0                                                         pyramids
        0                                                         tets
      7876                                                        faces
        0                                                         tris
      9854                                                        edges
      7161                                                        nodes
    Special                                                       Entities:
        1                                element                   blocks
        1                                                         sidesets
        1 nodesets
Journaled Command: list model
```

## List Special Entities

**List {special_type} <range>**

Special entities include (element) blocks, sidesets and nodesets (representing boundary conditions). Like the **list geometry** and **list mesh** commands, if no range is specified

then the number of entities of the given type is summarized. Otherwise, listing a special entity prints the mesh and geometry it contains.
(Some special entities are of interest mainly to developers and are not described here, e.g. whisker sheets, and whisker hexes.)

# gui

Graphical User Interface

- CUBIT Application Window
- Control Panel
- Graphics Window
- Model Tree
- Power Tools
- Property Editor
- Command Line Workspace
- Journal File Editor
- Toolbars
- Drop-Down Menus

The graphical user interface (GUI) can improve user productivity. It provides an easy way to control CUBIT without learning command syntax. Many geometry commands are faster and easier with the GUI. The underlying GUI components are constructed using a cross-platform development environment. As such, the GUI will behave similarly across all platforms supported by Cubit, yet each GUI will make use of platform specific widgets.
The GUI is built on top of the CUBIT command line. This means that GUI actions are translated to a CUBIT command-line string and journaled. Users familiar with command-line syntax can enter the same text in the GUI command-line window. Journal files can be created and played back in both environments with the same results. Although many things are faster and easier in the GUI, experienced users often use a combination of command line text and GUI button operations.
The discussion of the Graphical User Interface and its features is based on the basic windows contained within the CUBIT GUI Application Window. These are outlined in the subtopics listed above.
A full graphical user interface (GUI) with the standard look and feel consistent with major platforms is available on all supported Cubit platforms. The GUI version can improve productivity, making new users aware of the wide range of CUBIT capabilities, and freeing new and experienced users from having to remember esoteric syntax. The GUI and non-GUI versions create and play back identical journal files, making it easier to switch from one environment to the other.

tree view

# *Model Tree*

The model tree provides a complete graphical hierarchical representation of the parent child relationship of all geometric entities. The tree is populated as the model is constructed by Cubit. In addition to showing a hierarchy of geometric entities, the tree also shows active Groups, Boundary Layers, and active Boundary Condition entities.
The tree works directly with the graphics window and picking. Selecting an entity in the tree will select the same entity in the graphics window. Selecting an entity in the graphics window will highlight the tree entry if that entry is currently visible. If an entity's visibility is turned off, the icon next to that entity in the geometry tree will disappear.
If the tree entry is not visible the user may press the Find button located at the bottom of the tree. The first occurrence of the selected entity will be shown on the tree.
Virtual entities have a small (v) after the name to indicate that they are virtual entities.

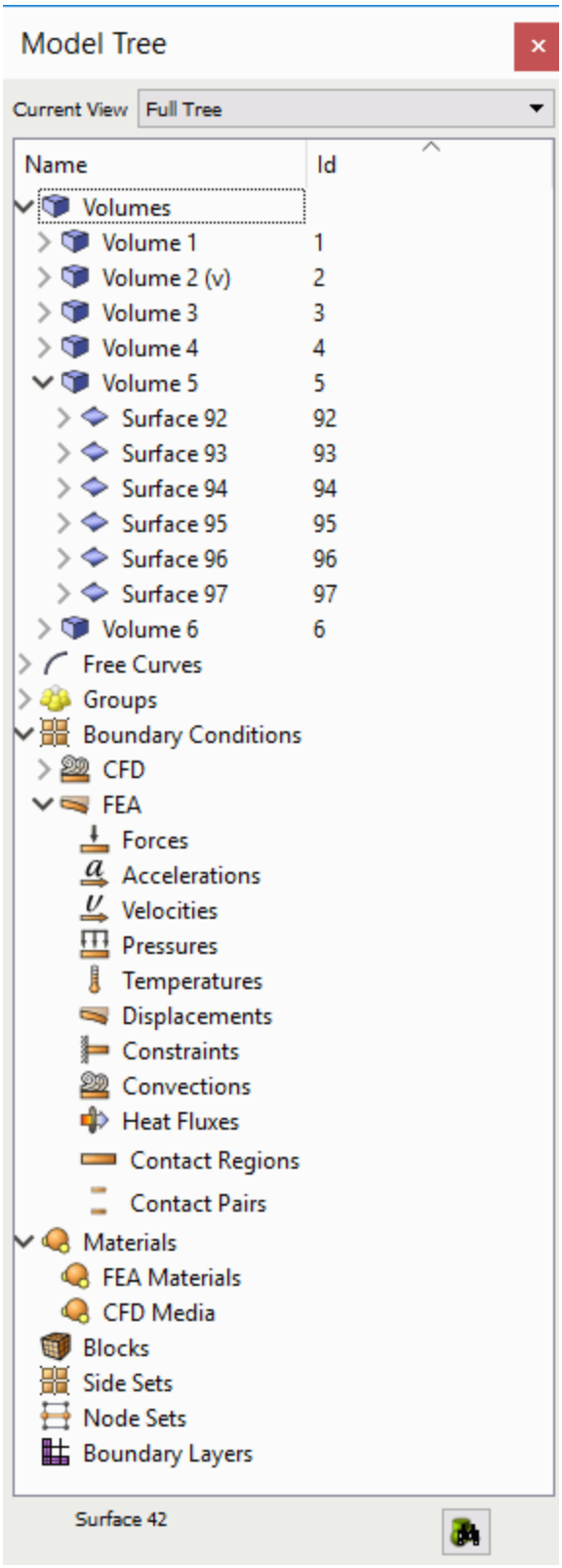



**Figure 1. Geometry Tree Window**

#### Drag and Drop

The Tree View window supports drag and drop of geometric entities into existing boundary condition sets. To create boundary conditions, see the Materials and Properties menu on the main control panel, or right-click on one of the boundary condition labels and select the "Create New" option from the context menu. Geometric entities or groups can be added to blocks, nodesets, or sidesets by dragging and dropping inside the tree view window.

#### Picked Group

The current selections in the graphics window can be added to a "picked group" by selecting the "Add to Picked Group" from the Right click menu. Selections can also be added to the picked group by dragging and dropping onto the group from the geometry tree window. The picked group can be substituted into any commands that use groups. To remove an item from the picked group, use the "Remove from Group" option in the right click menu in the geometry tree or from the graphics window.

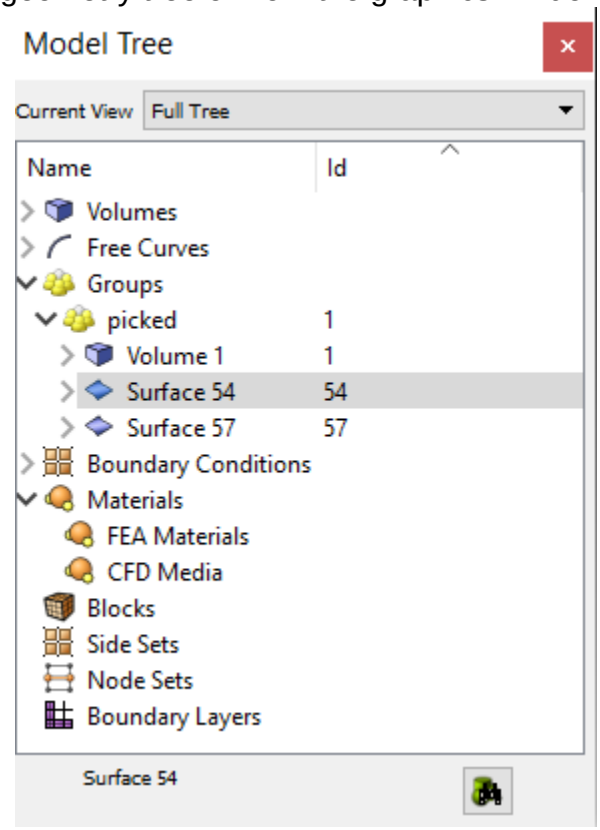



**Figure 2. Picked Group**

#### Right-Click Menu Functions

The geometry tree's context menu is sensitive to the type of item and the number of items selected. Functions that apply to the item type and number of selected items are available from the context menu. These include the following:

- **Zoom To** - Available for all geometric entities
- **Rotate About** - Change the center of rotation to the centroid of the entity without zooming
- **Fly-In** - Animated zoom feature
- **Locate** - Labels the selected entity in the graphics window
- **Draw** - Draw this entity by itself.
- **Isolate** - Similar to Draw command, but the display will not be refreshed with a graphics reset. To redisplay the model, select All Visible from the graphics window right-click menu.
- **Transparency On/Off** - Toggles transparency mode
- **Visibility On/Off** - Toggles visibility
- **Rename** - Allows you to rename entities from the tree. Clicking on a highlighted entity in the tree will do the same thing. This will also work for boundary condition entities (blocks, nodesets and sidesets)
- **Mesh** - Mesh selected entity at current settings.
- **Delete Mesh** - Available for meshed entities
- **Reset Entity** - Deletes mesh, and returns all settings to default values.
- **Delete** - Available when Volumes and Groups are selected.
- **Measure** - Available when two entities are selected or 1 curve is selected
- **Refresh Full Tree** - Used to return to main tree
- **Collapse Tree** - Available when entities are selected.
- **View Descendants/Ancestors** - Show this entity's individual hierarchy. Use the Refresh Full Tree option to return to main tree view.
- **View Neighbors** View adjacent entities. Use the Refresh Full Tree option to return to the main tree view.
- **Create New Volume** - Available when the user right-clicks over the Volumes (parent) label. Opens the geometry-volume-create panel
- **Import Geometry** - Available when the user right-clicks over the Volumes (parent) label. Opens import dialog.
- **Create New Group** - Available when the user right-clicks over the Groups (parent) label.
- **Clean Out Group** - Available when groups are selected. Removes all entities from group.
- **Remove from Group** - Available when groups are selected. Removes selected entity from the group.
- **Add Selected to Block/Nodeset/Sideset** - Add the selected entity in the graphics window to the chosen block, nodeset, or sideset in the geometry tree.
- **Delete Selected from Block/Nodeset/Sideset** - Delete the selected entity in the graphics window from the chosen block, nodeset, or sideset in the geometry tree.
- **Create New Block/Sideset/Nodeset** - Available when the user right-clicks over the respective Boundary Conditions (parent) label.
- **Create New <boundary condition>** - Available when highlighting desired boundary condition in the tree including CFD and FEA boundary conditions.
- **Draw Block/Sideset/Nodeset** - Draws the selected block/nodeset/sideset on top of existing entities
- **Draw Sideset/Nodeset Only** - Draws the selected nodeset/sideset independent of other entities

- **Delete Selected Boundary Condition** - Deletes any selected boundary conditions
- **Draw Selected Boundary Condition** - Draws selected boundary condition by itself
- **Draw Selected Boundary Condition (Add)** - Draws multiple boundary conditions
- **List Selected Boundary Condition** - Lists information about selected boundary conditions in the command line window
- **Remove from Block/Sideset/Nodeset** - Removes selected entity from the specified block, sideset or nodeset
- **Cleanup (Tets)** - Issues cleanup command for selected block. Only applicable for blocks composed of tet elements
- **Remesh (Tets)** - Issues remesh command for selected block. Only applicable for blocks composed of tet elements
- **List Info** - List information about selected entity in the output window.

# *Power Tools*

The power tools contain useful tools to help users through the mesh generation process. The Immersive Topology Environment for Meshing, also known as ITEM. This panel contains a wizard-like environment which guides the user through the mesh generation process through a series of panels and diagnostics. The geometry repair and analysis tools contains diagnostics and tools for analyzing and repairing geometry, although many of these can now be found in the ITEM environment as well. The mesh quality and meshing power tools aid in mesh generation and verification. The geometry and mesh comparison tool identifies correlation between existing geometry and mesh. The defeaturing tool assists users with defeaturing geometry in a more automated fashion. The assemblies tool help users manage assemblies, parts and related metadata.

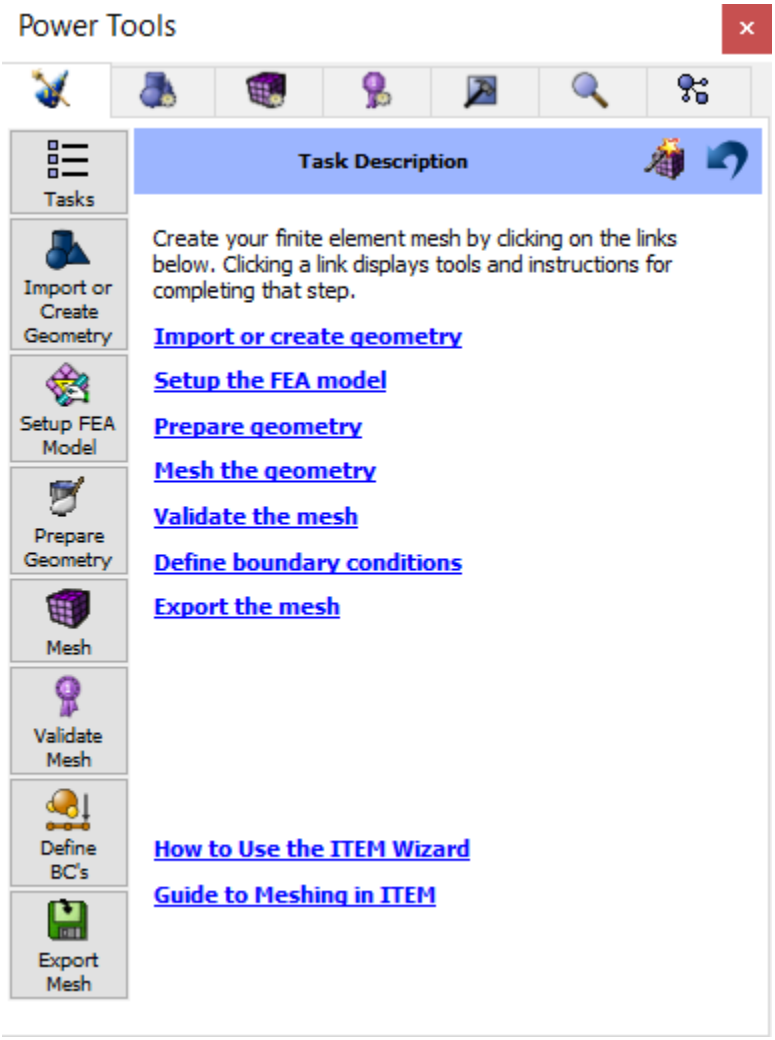



**Figure 1. Power Tools Window**

- Immersive Topology Environment for Meshing (ITEM)
- Geometry Analysis and Repair Tools
  - Machine Learning Tools
- Meshing Tools
- Mesh Quality Tools
- Defeaturing Tool
- Geometry/Mesh Comparison Tool
- Assemblies Tool

To familiarize yourself with the power tools environment (excluding ITEM), we recommend that you try the power tools tutorial.
To familiarize yourself with ITEM wizard, we recommend that you try the ITEM tutorial.

# *Geometry Power Tools*

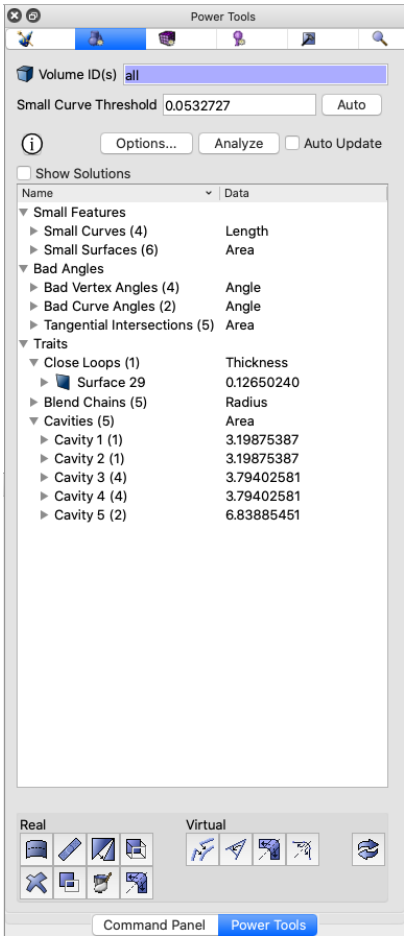



**Figure 1. Geometry power tools panel**

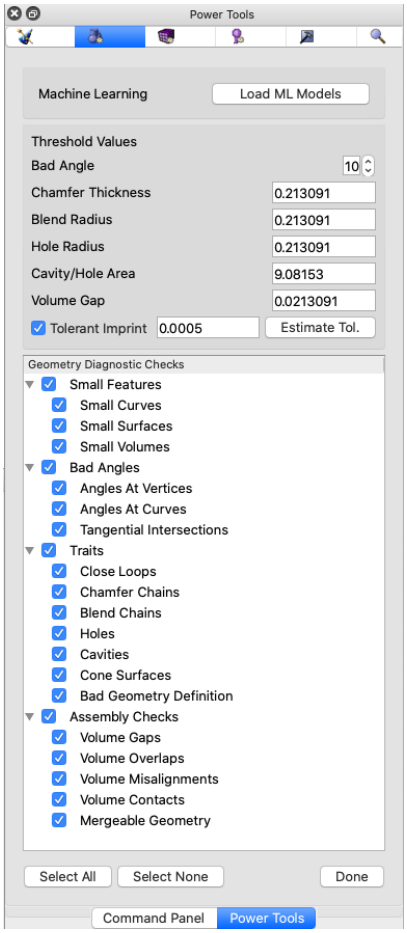



**Figure 2. Geometry power tools options panel**

The geometry power tools, shown in Figure 1. are located on the Tree View window under the blue geometry tab. The Geometry Power Tool provides several diagnostic tests to identify and repair problems in your CAD model prior to meshing including machine learning-based diagnostics and solutions.
Diagnostic tests include:

- Small Features: Small curves and surfaces and volumes
- Bad Angles: Near tangent angles at curves and vertices
- Geometric Traits: Identifies common geometric traits such as holes, blends, chamfers, etc.
- Assembly Checks: Gaps, overlaps and misalignments between multiple parts
- Beams and Shells: Tools for beam and shell modeling
- Tetmesh Poor Quality Predictions: Machine learning tool to predict poor quality tet mesh

- Part Classification: Machine learning tool to classify parts into common mechanisms

This tool analyzes geometry for various characteristics that may affect meshing outcomes and aid in simplification and defeaturing. It also contains a powerful toolkit of geometry modification methods to fix these problems. Many of the common geometry clean-up tools are available from this tool without the need to search through the command panels for relevant operations.
The geometry power tool includes a window that lists results from geometry analysis in a tree format. In addition, a solution window can be displayed that will display specific suggested geometry solutions for the currently selected entity.

#### Suggested Usage

The following is a suggested workflow for using the geometry power tool:

1. **Enter volumes to analyze:** Enter or pick the volume IDs you wish to analyze in the field labeled **Volume ID(s)**. By default, all volumes will be analyzed. For large or complex assemblies, consider selecting only a few volumes at a time to avoid long analysis times.
2. **Enter a small curve threshold:** The value entered in the field labeled **Small Curve Threshold** defines the basis for what is considered "small" for most geometry tests. If Cubit already has more than one volume defined, a default value for small curve threshold will be computed as **0.25*mesh_size**. To update the default **small curve threshold** for the current volumes, select the **Auto** button. If no mesh size is currently defined, an autosize factor of 2.5 will be used to compute a mesh size. (Equivalent to **vol all size auto factor 2.5**)
3. **Select diagnostics to perform:** Selecting the **Options...** button will display a list of available diagnostics grouped by category, as shown in Figure 2. By default all diagnostics are selected. Some diagnostics may not apply to specific geometry, or may only need to be run once per geometry. To avoid long analysis times, select only diagnostics that are relevant for your current problem scope. Clicking on the box by each test will select or deselect it. Categories of diagnostics may also be selected or deselected in a similar manner. All diagnostics may be selected or deseleted using the **Select All** and **Select None** buttons at the bottom of the panel. Threshold values used for some of the diagnostics can also be entered, including bad angle, chamfer thickness, blend or hole radius, cavity area and volume gap thresholds. Details on each of the diagnostics are described below. Select the **Done** button to return to the main Geometry power tool panel.
4. **Analyze the geometry:** Click the **Analyze** button to initiate an analysis of the selected diagnostics. The time taken for analysis will vary based on the number and complexity of volumes and the diagnostics selected.
5. **Select an entity to examine:** Once analysis is complete, the results will appear in the main window of the geometry power tool panel in the form of an expandable lists categorized by the selected diagnostics. Items in the list correspond to the selected tests. Expanding a list will display an ordered sub-list

of geometry entities that have been identified by the test. Selecting one or more entities in one of the lists will also highlight the entities in the graphics window. Use shift-click or command/ctrl-click to select multiple entities in the list. Use the context menu (right click) to zoom or fly in, locate, draw or other methods to graphically examine the selected entities.

6. **Choose a geometry repair solution:** Multiple methods are provided for choosing and selecting a relevant geometry repair solution:
    - **Context Menu:** Right clicking on an entity in the list will reveal a list of options that are normally relevant for the selected entity type. (See Figure 3.) For example, selecting the **Remove Surface...** menu item will bring up the Remove Surface command panel pre-populated with the relevant entity. To execute the same operation on many entities at once, first select all relevant entities in the list.
    - **Show Solutions:** Selecting the **Show Solutions** check box at the top of the results window will display an additional window, (See Figure 4.) populated with relevant operations for the currently selected entity. Selecting a solution will display a preview of the operation in the graphics window. Double clicking the solution will execute the solution. A right click on the solution will show a context menu revealing the following options:
        - **Execute:** Execute the selected solution (same as double click).
        - **Show More Solutions:** Add additional solutions computed for attached entities if they exist. For example, if a small curve is selected, this option will include additional solutions in the window for its attached surfaces and vertices.
        - **Open Command Panel Operation:** Depending on the type of solution selected, the relevant command panel will appear pre-populated with the options called for in the solution. This provides the option to further customize the solution if the precise desired command is not displayed.
    - **Command Panel Buttons:** The buttons at the bottom of the geometry power tool will display a specific geometry command panel. This can be useful if many similar operations are to be performed on different entities. A description of each is provided below.

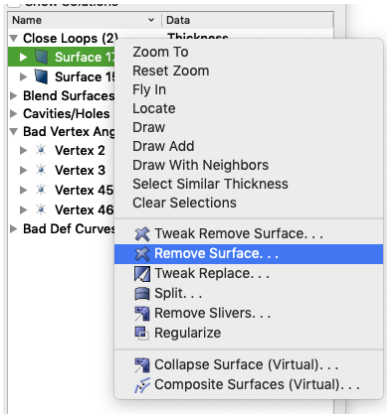



**Figure 3. Geometry entity context menu in power tool.**

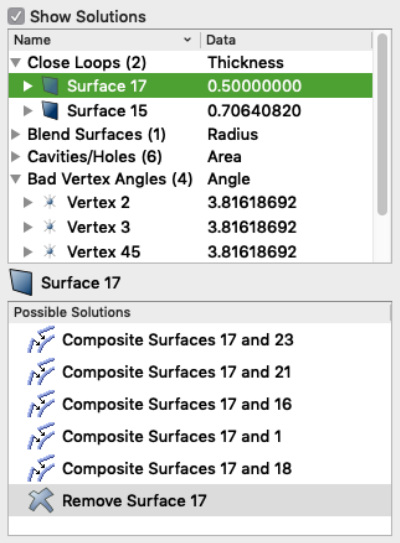



**Figure 4. Entitiy-specific solutions displayed in geometry power tool.**

#### Geometry Analysis Tools

The geometry power tools, contain various diagnostic tests that can be run on geometry to diagnose potential problems for mesh generation and defeaturing. To display a list of tests, click on the **Options...** button. The panel shown in Figure 2. will appear. Select or deselect the desired options from the window before performing an analysis. To avoid long analysis times, select only tests that are relevant for your current problem scope. Cubit will also save the current test selections between runs. The geometry analysis tests are summarized below:

##### Small Features

Small features may be necessary and desirable in a model, but many times they are the result of poor geometry construction, or they may just not be important to the analysis. The small features tests look for small curves, small surfaces, and small volumes. These tests rely on the user-defined **small curve threshold** value defined at the top of the Geometry power tool.

- **Small Curves** - Small curves, including zero-length curves such as hardpoints, are compared directly against the **small curve threshold** value, and identified if they are less than or equal to the given value.
- **Small Surfaces** - Small surfaces are identified based on area and hydraulic radius. Surface areas that are less than the square of the current **mesh_size** are identified as small. For surfaces where the hydraulic radius, defined as **4*surface_area/perimeter**, is less than the **small curve threshold** are also identified as small.

- **Small Volumes** - Small volumes are identified by their hydraulic radius, defined as **6*volume/surface_area**.

#### Bad Angles

Small geometric angles at vertices and curves can sometimes over-constrain the resulting mesh resulting in poor element quality. These tests are controlled by the **Bad Angle** threshold value defined at the top of the Geometry power tool Options panel.

- **At Vertices** - For vertices, the angle formed by two attached curves is measured. Vertices where angles less than the **Bad Angle** threshold. Figure 5 shows an example of a bad angle where the resulting tet or hex mesh may result in poor mesh quality. Note that depending on how the angle is measured at the vertex, the small angle may be also be identified if **360.0-angle** is less than the **Bad Angle** threshold. The **blunt tangency** or **collapse angle** operations are useful for removing bad angles at curves.

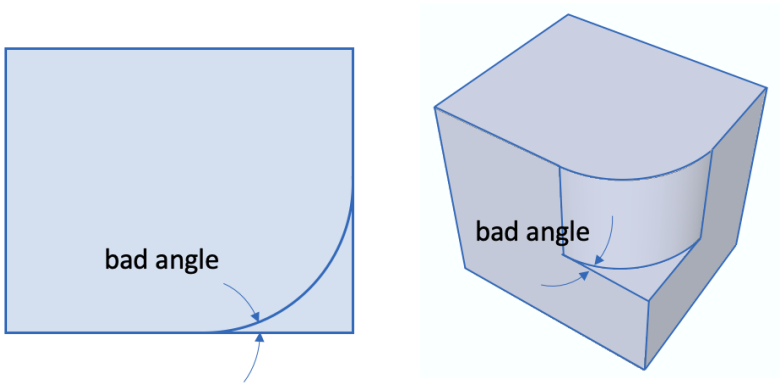



**Figure 5. Bad Angle at Vertex Example**

- **At Curves** - For curves, the angle formed by two attached surfaces is measured. Similar to vertices, curves where angles less than the **Bad Angle** threshold are identified.
- **Tangential Intersections** - A tangential intersection is formed when two parallel surfaces share an edge and have a 180 degree angle between them. The tangential intersection test is looking for the condition where two surfaces that meet tangentially share a common edge, and each of the surfaces has another edge which resides on a third face and forms a small angle. In the example shown in Figure 6., Surface 1 and Surface 2 are tangential to each other and share a common edge. Both Surface 1 and 2 have another edge which resides on Surface 3 and forms a small angle at the vertex common to all three surfaces.

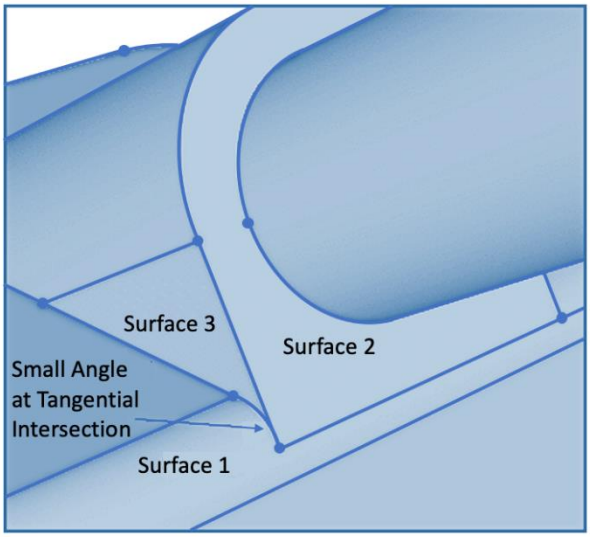



**Figure 6. Tangential Intersection Example**

#### Traits

The tests in the **Traits** category, group entities according to a specific characteristic of the geometry such as its thickness or radius. Use the threshold values at the top of the Geometry power tools Options panel to set limits on values used to control entities returned from these tests. Geometry Traits include the following:

- **Close Loops** - Close loops are identified by two curves on a single surface for which the shortest distance between them is less than the current **mesh_size**. Surfaces identified as close loops are ordered based on the minimum thickness of the surface between the loops. These surfaces and their immediate neighbors are often candidates for the **remove surface** command.

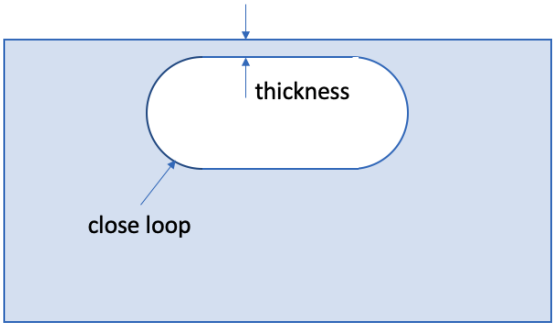



**Figure 7. Close Loop Example**

- **Chamfer Chains** - A chamfer surface can be identified as a narrow strip where its angle to neighboring surfaces is about 45 degrees as shown in Figure 7. Chamfers often occur as a chain or connected set of surfaces and are grouped

together in the power tool as a collection of surfaces that can be expanded and examined individually. Chamfer chains are ordered based on the narrow thickness of the surfaces illustrated in Figure 8. Setting the **Chamfer Thickness** threshold in the Options panel will control which chamfer chains will be identified. The default value for **Chamfer Thickness** threshold is the current **mesh_size** Since chamfers with small thickness can effect the resulting size of the elements the **remove surface** option is often used to eliminate them.

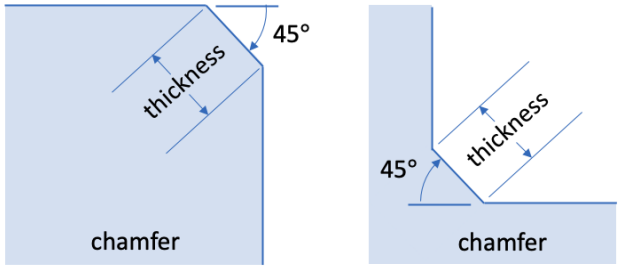



**Figure 8. Chamfer Examples**

- **Blend Chains** - A blend surface serves as a smooth transition between two neighboring surfaces, such as a fillet as shown in Figure 8. Blends are identified as surfaces having a constant radius along one of its parametric directions. Blends often occur as a chain or connected set of surfaces and are displayed as a collection of surfaces in the power tool that can be expanded and examined individually. Enter a **Blend Radius** threshold value at the top of Geometry power tools options panel to control the maximum radius of curvature for surfaces returned from this test. The default value for **Blend Radius** threshold is the current **mesh_size**. Resulting blend surfaces are ordered based upon their minimum radius of curvature. Blend chains can be candidates for the **remove surface blend_chain** or **split surface** commands.

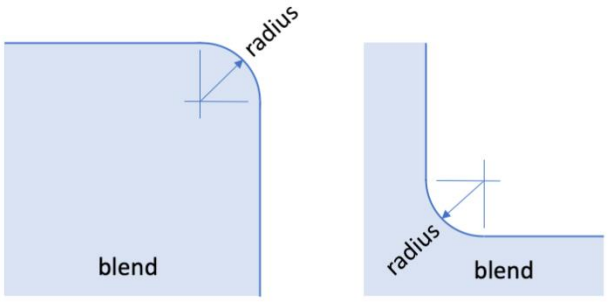



**Figure 9. Blend Examples**

- **Holes** - Holes are a special category of **Cavity** (see below). They are collections of surfaces that are bounded by curves where the exterior angle is greater than 180 degrees and at least one of the surfaces have a radius of curvature less than

the **Hole Radius** threshold. Figure 10 illustrates a hole that is comprised of a cylindrical surface and a planar circular surface. Resulting hole collections of surfaces are ordered based upon their cylindrical radius. Holes can be candidates for the **remove surface cavity** command.

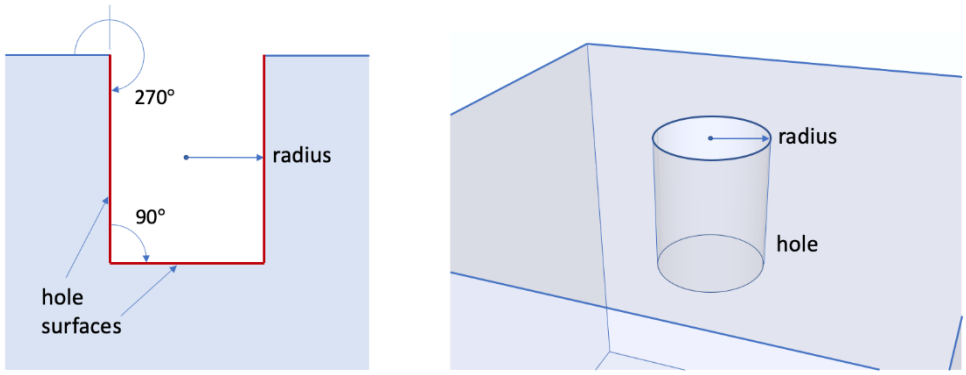



**Figure 10. Hole Example**

- **Cavities** - Small cavities in a volume may be candidates for removal from the geometry. A cavity is defined as a collection of surfaces bounded by curves with an external angle greater than 180 degrees. Enter the **Cavity Area** threshold value at the top of the Geometry power tools Options panel. This value controls the maximum total surface area for a cavity identified from this diagnostic test. Since cavities may consist of many individual surfaces, the resulting ordered list displayed in the power tool includes sub-lists of surfaces that can be expanded and examined individually. Surfaces contained with cavities or holes can be candidates for the **remove surface cavity** command which will remove all surfaces in the cavity sumiluatneously.

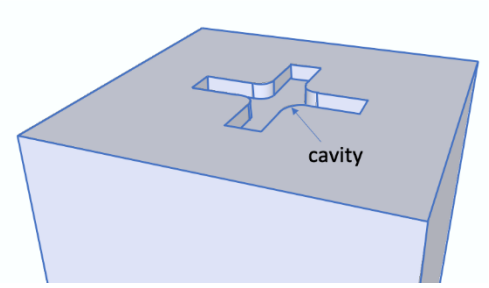



**Figure 11. Cavity Example**

- **Cone Surfaces** - Cones are defined as any surface comprising exactly two curves where one of the curves is of zero length. Cone surfaces can often cause difficulty for surface meshing, and should be removed when possible. Surfaces identified as cones are ordered based on their surface area. Cone surfaces are good candidates for the **tweak surface cone** command.

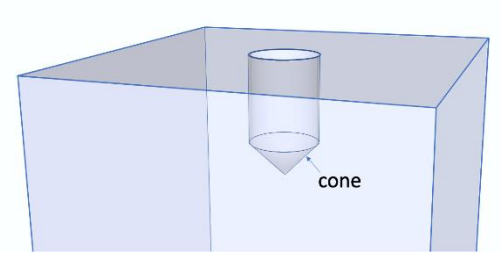



**Figure 12. Cone Example**

- **Bad Geometry Definition** - Cubit uses third party libraries, such as ACIS from Spatial, Inc. for much of its geometric modeling capabilities. The bad geometry definition check calls internal validation routines in these libraries, when available, to check for errors in geometry definition. Entities indetified as "bad geometry" are usually candidates for the **heal volume** command. If the third party library does not provide validation capabilities, this check will not return anything. Note: ACIS is a trademark of Spatial.

#### Assembly Checks

Check the interactions between multiple volumes. Here we check for overlaps, gaps and misalignments between nearby volumes. It will also identify volumes that are in contact as well as entities that are ready for merging.
For assemblies of volumes, it is important to identify if volumes will be connected (imprinted and merged) are in contact, or separated by some distance. The **Assembly Checks** provide diagnostics and solutions to validate and resolve these interactions.
The **Gaps**, **Overlaps** and **Misalignments** diagnostics normally identify undesirable conditions that must be resolved prior to **imprint** and **merge**. Once resolved, the **Volume Contacts** and **Mergable Geometry** can be used to validate connections before and after imprinting and merging.
The Options panel also provides a way to estimate or manually set an imprint tolerance. Entities closer than this tolerance will be considered mergable when used with the tolerant imprint command. When the **Tolerant Imprint** checkbox is selected in the Options panel, the diagnostic tests that identify gaps, overlaps and misalignments will also use the specified tolerance when computing issues.

- **Volume Gaps** - Lists volume pairs that are separated by a distance smaller than the **Volume Gap** tolerance specified in the Options panel, but are not in contact or overlapping. Gaps can result in parts that are not correctly merged and will not share nodes between volumes when meshed. Expanding a volume pair in the list will display individual surface pairs where gaps exist between the volumes. Figure 13 illustrates a gap between two volumes. Gaps can be visualized using the **Draw Volume Gap** context menu, also shown in Figure 13, where the surfaces that are within the gap tolerance are displayed in red. The **tweak surface replace** command can sometimes be used to correct overlaps.

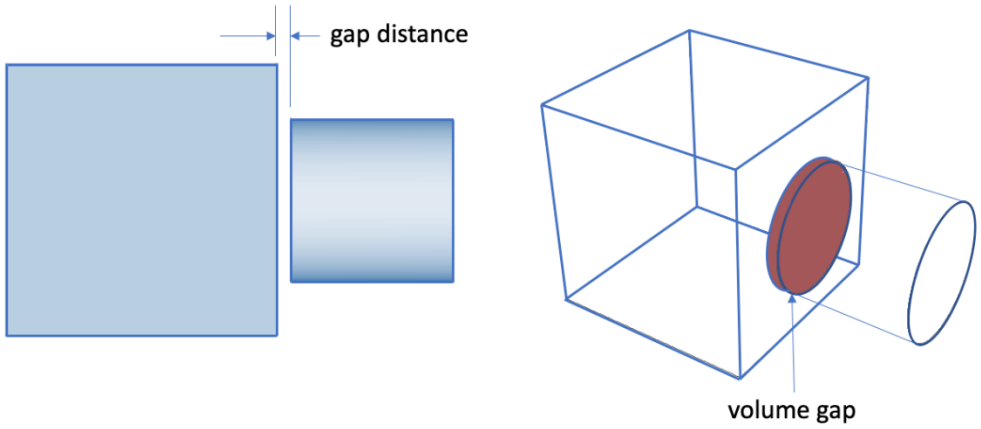



**Figure 13. Volume Gap Example**

- **Volume Overlaps** - Lists volume pairs that are overlapping. Figure 14. shows an example of a volume overlap. Overlapping volumes can result in sliver surfaces and bad element quality if they are not resolved prior to imprinting and merging. Overlaps can be displayed with the context menu item, **Draw Volume Overlap** which displays the overlap region in red. The **remove overlap** command or **tweak surface replace** commands can often be used to correct overlaps.

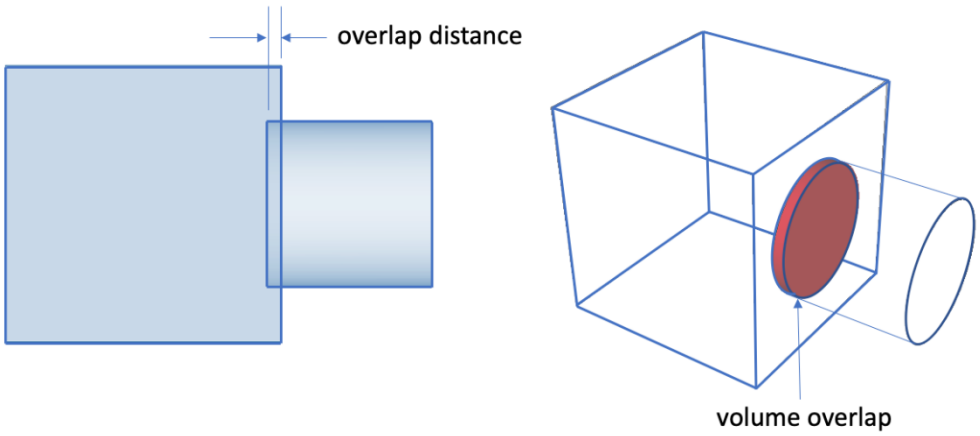



**Figure 14. Volume Overlap Example**

- **Volume Misalignments** - Misalignments are caused when neighboring volumes touch without overlap, but a small distance between neighboring vertices, curves or surfaces is identified. Figure 15 shows an example of a misalignment. Misalignments can result in sliver surfaces and bad element quality if not resolved prior to imprinting and merging. The **Volume Misalignments** diagnostic test will list pairs of volumes that are misaligned. Expanding a volume pair will reveal entity to entity misalignments that were detected between the pair. Three categories of misalignments will be displayed, namely: vertex-vertex, vertex-curve and vertex-surface ordered by their misalignment distance. These indicate

entity pairs that are closer than the **Volume Gap** tolerance that is set in the Options panel. The **tweak surface replace** command can often be used to correct misalignments.

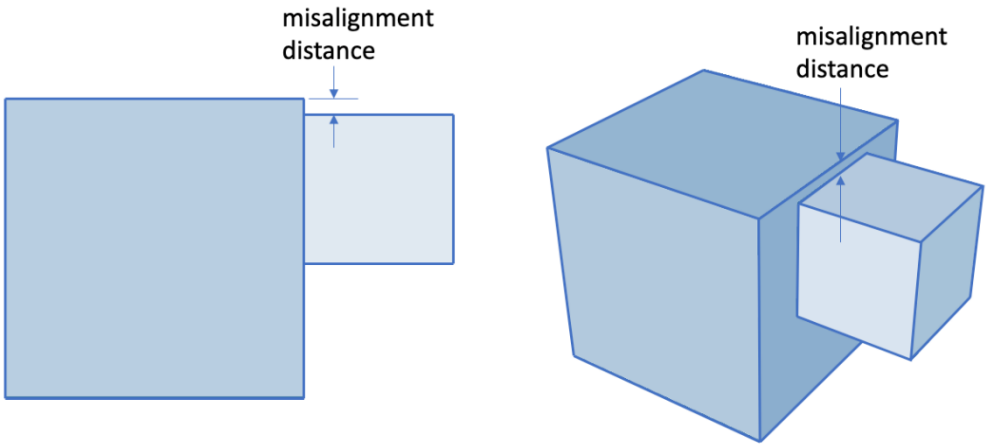



**Figure 15. Volume Misalignment Example**

- **Volume Contacts** - Volumes that have surfaces in contact but not merged are displayed with this diagnostic test. This provides a way to distinguish volumes that are merged from those that are not and validate whether a contact state should exist between neighboring volumes. This list will include volume pairs that are touching including those that have been identified by the **Volume Misalignment** diagnostic test. Expanding a volume pair will reveal pairs of surfaces on different volumes that are in contact. If the contact state is not correct, normally an **imprint** and **merge** operation should be performed
- **Mergable Geometry** - Pairs of entities on neighboring volumes that are co-located are identified by this diagnostic test. This is normally used to verify that the expected set of surfaces are coincident prior to merging. Mergeable geometry pairs of surfaces, curves and vertices are displayed in this list. Lower order entities (ie. curves and vertices) will not be displayed if its parent geometry (ie. surface) is identified as mergeable. In most cases, lower order entities identified by this diagnostic indicate the existence of overlaps or misalignments and should be resolved before imprint and merging. Note that the default merge tolerance of 1e-6 is used to determine if entities are mergeable unless the **Tolerant Imprint** checkbox is selected in the Options panel and a user defined tolerance is set.

#### Geometry Repair Tools

The geometry repair tool buttons appear at the bottom of the Geometry Power Tool. Selecting one of these buttons will bring up the relevant command panel. Tools included in this panel have proven useful for geometry repair and defeaturing.

 **Split Surface Button**

The split surface tool is used to split a surface into two surfaces. This is useful for blend surfaces, for example, where splitting a surface may facilitate sweeping. To select a surface for splitting, click on the surface in the tree view. To select multiple surfaces in the window, hold the CTRL key* while selecting surfaces (surfaces must be attached to each other). Then press the split surface button to bring up the Control Panel window with the ids of selected surfaces in the text input window. The split surface menu is located on the Control Panel under Geometry-Surface-Modify. You must press the Apply button for the command to be executed. You can also bring up the Split Surface menu by selecting surfaces in the tree view and selecting **Split** from the right click menu.
***Note:** For Mac computers, use the command key (or apple key) to select multiple entities

 **Heal Button**

The healing function in Cubit is used to improve ACIS geometry that has been corrupted during file import due to differences in tolerances, or inherent limitations in the parent system. These errors may include: geometric errors in entities, gaps between entities, and the absence of connectivity information (topology). To heal a volume, select the volume in the geometry repair tree view. Then press the heal button. You may also press the heal button without a geometry selected in the window, and enter it later. The Control Panel window will come up under the Geometry-Volume-Modify option with the selected volume id highlighted. If no entity is selected, or if another entity type is selected, the input window will be blank. You can also open the healing control panel by selecting **Heal** from the right click menu in the geometry power tools window.

 **Tweak Button**

The tweak command is used to eliminate gaps between entities or simplify geometry. The tweaking commands modify geometry by offsetting, replacing, or removing surfaces, and extending attached surfaces to fill in the gaps. Tweaking can be applied to surfaces, and it can be applied to curves with a valence no more than 2 at each vertex. It can also be applied to some vertices. To tweak a surface, select the surface in the tree view. The Geometry-Surface-Modify control panel will appear with the selected surface id in the input window.
Tweaking is available for curves. Tweaking a curve creates a blended or chamfered edge between two orthogonal surfaces. The curve option is located on the Geometry-Curve-Modify panel under the Blend/Chamfer pull-down option.
Tweaking is also available for some vertices. Tweaking a vertex creates a chamfered or filleted corner between three orthogonal surfaces. The vertex option is located on the Geometry-Vertex-Modify panel under the Tweak pull-down menu.
**Note: Only curves with valence 2 or less at each vertex are candidates for tweaking. Any other curve will cause the Geometry-Surface-Modify menu to appear.**

 **Merge Button**

The merge command is used to merge coincident surfaces, curves, and vertices into a single entity to ensure that mesh topology is identical at intersections. Unlike other buttons on the geometry repair panel, the merge button acts as an "Apply" button itself. All geometry that is listed under "mergeable entities" will be merged.

 **Remove Button**

The remove button is used to simplify geometry by removing unnecessary features. To use the remove feature, click on the surface(s) in the Tree View. Right click and select the Remove Option, or click the Remove icon on the toolbar. The Control Geometry-Surface-Modify control panel will appear, with the surface ids in the input window. The Remove control panel can also be accessed from the right-click menu in the Geometry Power Tools window. Select options and press apply.

 **Regularize Entity Button**

The regularize button is used to remove unnecessary topology. Regularizing an entity will essentially undo an imprint command.

 **Remove Slivers**

The remove slivers button is used to remove surfaces with less than a specified surface area. When ACIS removes a surface it extends the adjoining surfaces to fill the gap. If it is not possible to extend the surfaces or if the geometry is bad the command will fail.

 **Auto Clean Geometry**

The auto clean button is used to perform automatic cleanup operations on selected geometry. These automatic cleanup operations include forcing sweepable configurations, automatically removing small curves, automatically removing small surfaces, and automatically splitting surfaces.

 **Composite Button**

The composite button is used to combine adjacent surfaces or curves together using virtual geometry . Virtual geometry is a geometry module built on top of the ACIS representation. Surfaces may be composited to simplify geometry in order to facilitate sweeping and mapping algorithms by removing constraints on node placement. It is important to note that solid model operations such as webcut, imprint, or booleans, cannot be applied to models that have virtual geometry. Both curves and surfaces may be composited.

 **Collapse Angle Button**

The collapse angle button uses virtual geometry to collapse small angles. This is accomplished by partitioning and compositing surfaces in a way so that the small angle gets merged into a larger angle. Pressing the collapse button on the geometry power tools will open the collapse menu under Geometry-Vertex-Modify control panel. This panel can also be opened by selecting **Collapse** from the right click menu in the Geometry Tools window.

 **Collapse Surface Button**

Pressing this button will open the collapse surface panel on the main control panel. The collapse surface function uses virtual geometry to eliminate small surfaces on the model to improve mesh quality. It is most useful for blend surfaces.

 **Collapse Curve Button**

Pressing this button will open the collapse curve panel on the main control panel. The collapse curve command is used to eliminate small curves using virtual geometry.

 **Reset Graphics Button**

The reset graphics button will refresh the graphics window display.

**Note:** Pressing most of the geometry tool buttons on the panel will only bring up applicable command panels on the Control Panel. You must press the Apply button on the Control Panel to execute the command.

#### Context (Right Click) Menu

The following right click menu options are available from the geometry power tool's main window when a geometry entity or category is selected. Figure 3. shows an example of a context menu. Specific options depend on the type of entity or category.

##### Test Categories

- **Select All** - Selects all entities in the category
- **Draw All** - Draw all entities in the category
- **Draw All Add** - Draw all entities in the category without first clearing the display
- **Locate All** - Labels all entities in the category in the graphics window. Refresh screen to hide.
- **Expand All** - Expand all categories to show sub-lists of entities
- **Collapse All** - Collapse all categories to hide sub-lists of entities

##### Entity Visualization Options

- **Zoom To**- Zoom to selected entity in the graphics window
- **Reset Zoom** - Reset graphics window zoom
- **Fly-in** - Animated zoom
- **Locate** - Labels the selected entities in the graphics window. Refresh screen to hide.
- **Draw** - Displays only selected entities by themselves.
- **Draw Add** - Adds the selected entity to the display without clearing.
- **Draw with Neighbors** - Displays only selected entities with all attached neighbors
- **Select Similar ...** - Selects other entities in the same category that have the same geometry characteristic. For example, area, loop thickness, blend radius, angle at vertex, etc.
- **Clear Blend Chain** - Available in Blend category. Selects surfaces in the same blend chain as the selected surface.
- **Clear Cavity/Hole** - Available in Cavity/Hole category. Selects surfaces in the same cavity or hole collection as the selected surface.
- **Clear Selections** - Clears all highlighted entities and reset graphics

##### Cubit Solution Options

Each of the following menu options are available based on the category and entity type selected. In each case they will open the relevant command panel pre-populated with the

entity selected. Select multiple entities prior to selecting the context menu item below to execute the command on multiple entities simulaneously.

- **Tweak Remove Surface...**
- **Remove Surface...**
- **Tweak Replace...**
- **Split...**
- **Remove Slivers...**
- **Remove Blend...**
- **Remove Cavity...**
- **Blunt Tangency...**
- **Heal Owning Body...**
- **Merge Selected**
- **Merge All**
- **Unite Volumes**
- **Regularize**
- **Collapse Curve (Virtual)...**
- **Collapse Surface (Virtual)...**
- **Composite Curves (Virtual)...**
- **Composite Surfaces (Virtual)...**
- **Collapse Angle at Vertex (Virtual)...**

*Meshing Tools*

The meshing power tool provides a tool for determining whether a geometry can be meshed using autoscheme, or if it requires its scheme to be set explicitly. This tool is designed to help guide users through geometry decomposition process by providing a convenient way to see which geometries need further modification or decomposition prior to meshing.

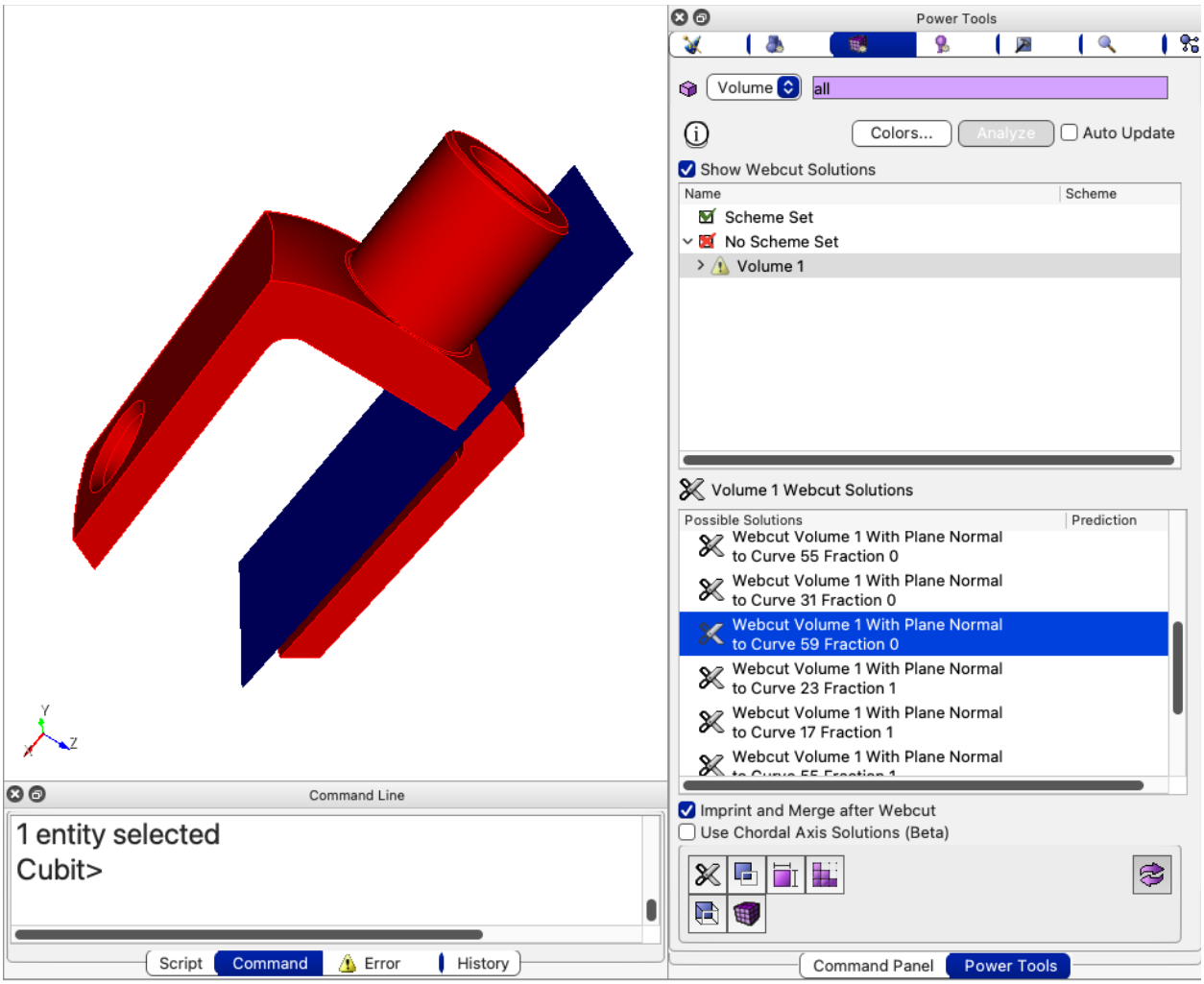



**Figure 1. Meshing Power Tool showing the solution window with a preview of a webcut solution.**

**Entity Specification**- The meshing power tool works for volumes or surfaces.
**Colors Button** - Opens the **Tools>Options** dialog to change the visualization colors of surface schemes for the meshing tool
**Show Webcut Solutions** - When selected the webcut solutions window will be displayed and populated with custom webcut solutions based on the selected volume.
**Analyze Button** - The Analyze button issues the autoscheme command for all selected volumes and surfaces and populates a two lists: **Scheme Set** and **No Scheme Set**.
**Auto Update Checkbox** - Used with the webcut solutions, when selected, after executing a webcut, the model will automatically update to determine meshability of the active volumes.
**Output Tree** - The output from the meshing tool is displayed in tree format. Geometry is divided into "Scheme Set" and "Scheme Not Set" divisions. The geometry is listed under these nodes. If autoscheme was successful, its assigned scheme is also displayed.
**Solutions Window** - A solutions window that displays potential webcuts for a selected volume is also available. Display the solutions window by selecting the **Show Webcut**

**Solutions** checkbox at the top of the panel. When a volume is selected, a list of potential webcuts will be displayed. The solutions can be previewed by selecting them and double-clicking will execute the webcut. To further customize the webcut, a right-click on the solution will provide access to the relevant webcut command panel, pre-populated with necessary parameters.
**Imprint and Merge After Webcut Checkbox** - Available when the Solutions window is displayed, when selected, the resulting webcut operations will also perform an imprint and merge operation.
**Toggle Visibility Button** - The meshing tool displays entities as red or green in the graphics window. Green means that they are currently meshable using the autoscheme. Red means that they require their scheme to be set explicitly. Turning this capability off will return the volumes and surfaces to their original colors.
**Meshing Tools Buttons** - Several meshing tools are available to the user from this window. Depending on the entity selected, these are also available from the right-click context menu, and they are described below.

#### Right Click Context Menu

- **Zoom To** - Zoom in on this element in the graphics window
- **Draw** - Draw this entity by itself in the graphics window
- **Locate** - Locates and labels entity in the graphics window
- **Rotate About** - Issues Rotate about command for selected entity
- **Visibility On/Off** - Toggle visibility
- **Reset Graphics**- Reset graphics display
- **Set Size** - Opens the Mesh/Entity/Interval panel on the control panel where you can set interval sizes for the selected geometry
- **Set Scheme** - Opens the Mesh/Entity/Mesh panel on the control panel where you can set a scheme for the selected entities
- **Set Vertex Type** - Available when surfaces are selected. Opens the Mesh/Surface/Mesh panel to set vertex types.
- **Imprint/Merge**- Opens the Geometry/Entity/Merge panel on the control panel. If you have entities selected in the tree window it will input them to the imprint/merge command.
- **Webcut** - Opens the Geometry/Volume/Webcut panel on the control panel. If a volume is selected in the meshing tool window it will input it in the webcut panel.
- **Color Surfaces** - Color surfaces based on their schemes. You can change the default colors by selecting the Options button.
- **Restore Colors** - Restores colors on selected entity or entity type
- **Mesh** - Meshes the selected entities (bypassing control panel)
- **Delete Mesh** - Deletes the mesh on selected entities
- **Unmerge** - Unmerges selected entities
- **View Descendants** - Opens a list of child entities and their meshing schemes. Press Analyze to return.
- **View Ancestors**- Opens a list of parent entities and their meshing schemes. Press Analyze to return.
- **View Neighbors**- Opens a list of bordering entities and their meshing schemes. Press Analyze to return.

# *Mesh Quality Tools*

The mesh quality tool is located in the entity tree window under the quality tab. The Mesh Quality Tool works on meshed entities to analyze mesh quality based on selected metrics. Output from the mesh quality analysis can be visualized using color-coded scales. The mesh quality tool also contains tools to improve mesh quality including smoothing, refinement, node merging, mesh validation, deleting mesh elements, and repositioning nodes.

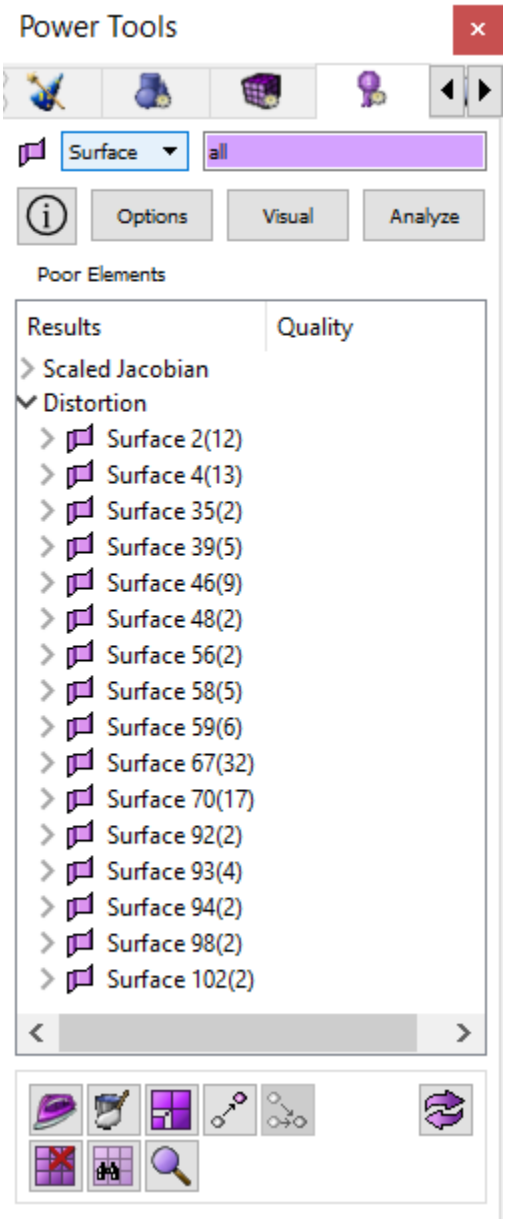



**Figure 1. Mesh Quality Tools**

**Entity Type** - The mesh quality tools can only be applied to mesh entities including volumes, surfaces, hexahedra, quadrilaterals, triangles, or tetrahedra.
**Help Button** - Opens context specific help for this topic.
**Options Button** - Clicking on this button will show the Tools>Option menu dialog that allows users to manually enter metric range settings. The settings are persistent between sessions. For a description of quality metrics and default ranges click on one of the following links:

- Metrics for Hexahedral Elements
- Metrics for Quadrilateral Elements
- Metrics for Tetrahedral Elements
- Metrics for Triangular Elements

**Visual Button** - Clicking on this button will open the Mesh/Entity/Quality command panel specific to the entity selected. To visualize elements in the graphics window based on a color-coded quality scale, you must select the entities to visualize and check the "Display Graphical Summary" check box. Once that box is selected, you must also make sure the "Draw Mesh Elements" option is selected. Then press the Apply button
**Analyze Button** - This button starts the quality processing based on the metrics/filters selected.
**Output Window/Tree** - The failed elements are shown in the tree under the heading "Poor Elements". For each metric/filter the output will be listed in a tree format with the following nodes.

1. The top node on the tree is the name of the metric.
2. The next node under is the owning volume or surface when volumes or surfaces are analyzed.
3. The next node will be categories or groups of elements. Possible categories are:
    - All Above Threshold - represents all mesh elements above the quality threshold upper range
    - All Below Threshold - represents all mesh elements below the quality threshold lower range
    - Top "n" - This will expand into a list, up to 50 elements long, of the worst offending elements above the upper threshold range.
    - Bottom "n" - This will expand into a list, up to 50 elements long, of the worst offending elements below the lower threshold range.
4. At the lowest level of the tree are mesh elements.

The mesh elements can be sorted by quality or by numeric order. To change the way items are sorted, click on the headings. The right-click or context menu will show various remedies depending on what is selected. Performing an operation on a parent node will perform the same operation on all of the child nodes.

#### Mesh Quality Tool Buttons

The buttons on the bottom of the mesh quality tool window are some of the tools you may use to improve mesh quality and include.

- **Smooth Button** - Opens the Mesh>Entity>Smooth panel
- **Refine Button** - Opens the Mesh>Entity>Refine panel
- **Move Node** - Opens the Mesh>Node>Move Node panel
- **Merge Node** - Opens the Mesh>Node>Merge Node panel
- **Delete Mesh Element** - Deletes selected mesh entity
- **Validate Mesh** - Issues the validate mesh command
- **Check Coincident Nodes** - Issues the check coincident nodes command.
- **Refresh Graphics**

#### Right-Click Context Menu Items

- **Draw** - issues a draw command for any tree node below the metric name.
- **Color Code** - Issues a 'quality .... draw mesh' command for any tree node below the metric name
- **Locate** - Issues Locate for volume/surface/hex/quad/tet/tri. The locate command will draw and label selected entities in the graphics window.
- **Fly-In** - Issues Fly-in for volume/surface/hex/quad/tet/tri. The fly-in command is an animated zoom feature.
- **Zoom to** - Issues Zoom command for volume/surface/hex/quad/tet/tri
- **Rotate About** - Issues Rotate About command for volume/surface/hex/quad/tet/tri
- **Vis on/off** - Issues visibility on/off for volume/surface
- **Smooth** - Issues generic smooth command for volume/surface/hex/tet
- **Smooth Surface Parent** - issues a smooth surface command for the surface parents of selected quads and tris.
- **Delete Mesh** - issues delete mesh propagate command for vol/surf
- **Delete Elements** - issues delete element command for mesh entities in all categories except 'all'
- **Validate mesh** - validates selected volume or surface
- **Check Coincident Nodes** - checks for coincident nodes on volume or surface
- **Smooth Panel** - brings up the correct smooth panel depending on what's selected
- **Smooth Surface Panel** - bring up the smooth surface panel with correct surface ids for selected quads and tris
- **Merge Node Panel** - brings up the panel to merge nodes
- **Move Node Panel** - brings up the panel to move nodes
- **Reset Graphics** - resets the display

*Assembly Tool*

In Cubit versions prior to 15.4 assembly data was managed on the model tree. Beginning with Cubit version 15.4 a power tool for assemblies is available.
A detailed, command-based discussion of assemblies and metadata can be found here.

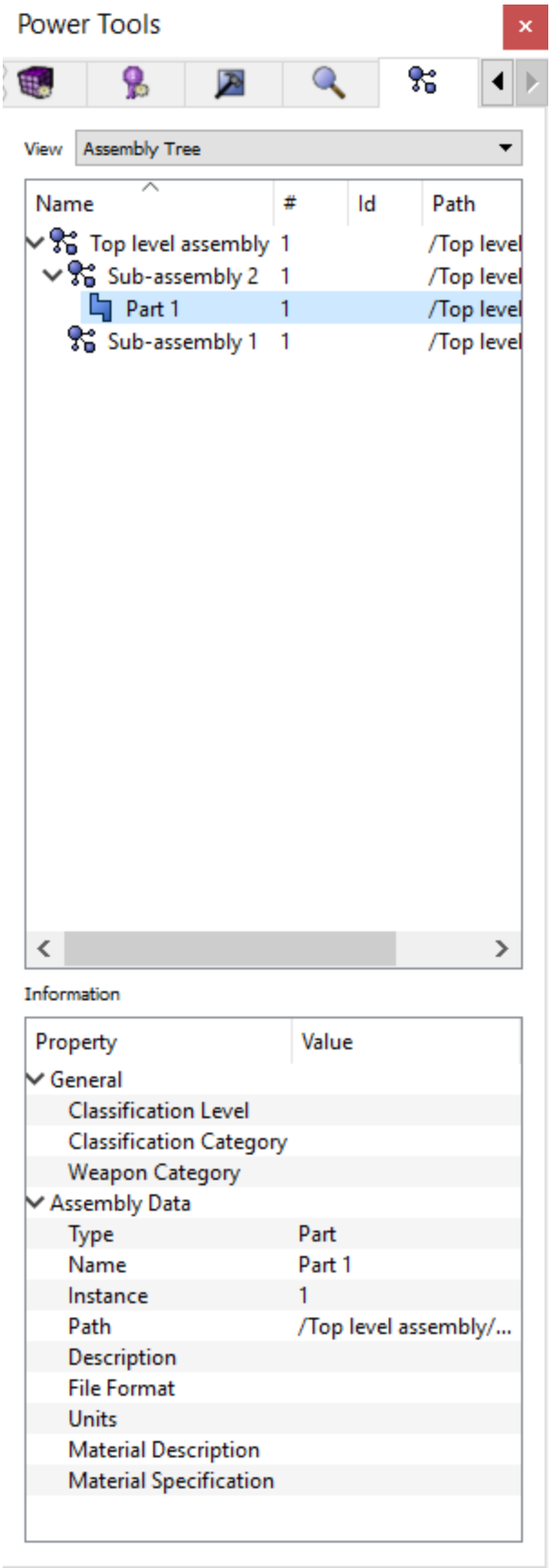

Power Tools
View
Assembly Tree
Name
#
Id
Path
Top level assembly 1 /Top level
Sub-assembly 2 1 /Top level
Part 1 1 /Top level
Sub-assembly 1 1 /Top level
Information
Property
Value
General
Classification Level
Classification Category
Weapon Category
Assembly Data
Type Part
Name Part 1
Instance 1
Path /Top level assembly/...
Description
File Format
Units
Material Description
Material Specification

**Figure 1 - Assembly Power Tool**

Like all other power tools in Cubit, the user is encouraged to experiment with the various options found in context menus. The context menus are specific to the entity type selected.

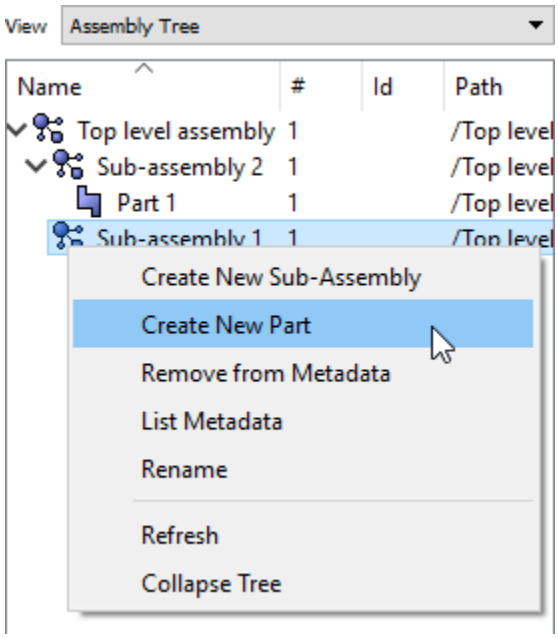



**Figure 2 - Assembly Context Menu**

#### Metadata Attributes

Users have access to the following attributes:

- **Type** - The metadata type: Assembly, Sub-Assembly or Part
- **Name** - The name for the assembly or part. This can be edited from the property window
- **Instance** - The numeric value associated with the part or assembly
- **Path** - The absolute path of the part or assembly
- **Description** - The description of the part or assembly
- **Material Description** - The name or description of the material of which this part is composed
- **Material Specification** - The formal specification number of the material of which this part is composed
- **File Format** - The name of the file system containing the original version of this entity
- **Units** - The unit system of this part or assembly

#### Another View

At the top of the tool is a pull down menu. This allows the user to change from the full assembly view to a parts view.

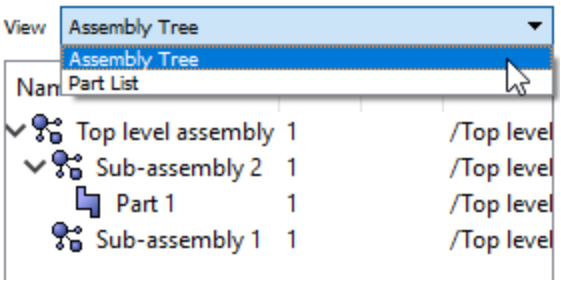
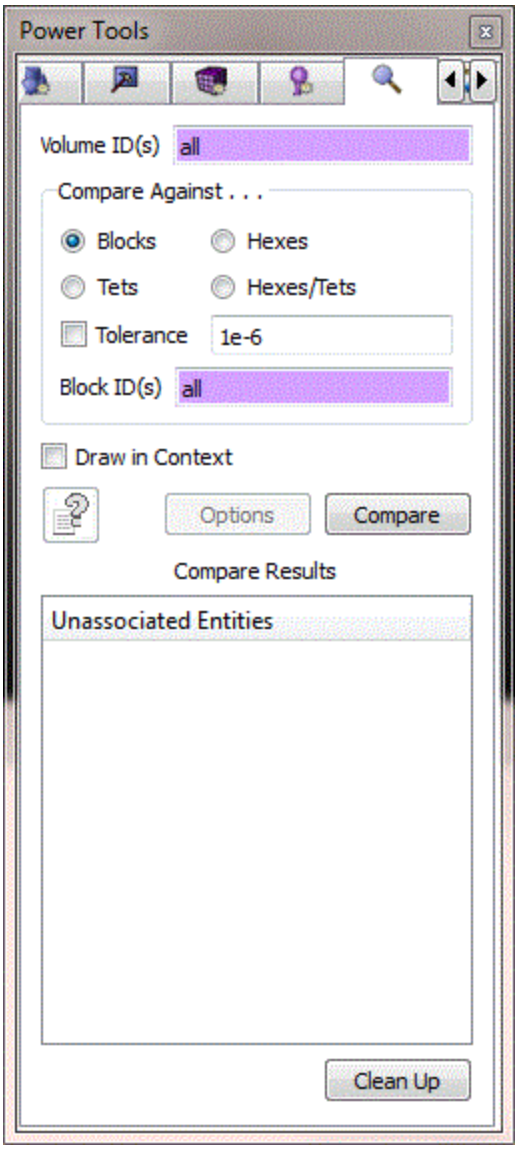



**Figure 3 - Pull down Menu**

*Geometry/Mesh Comparison Tool*

The Geometry/Mesh Comparison Tool tries to find geometry and mesh that do not correspond. The typical use is to import a geometry file and then import a mesh file that is associated with the geometry. The comparison tool will locate mesh that does not correspond to the geometry. The tool will also show geometry that does map to any mesh.




The user selects the volumes for the comparison, then selects the mesh entities for the comparison. A default comparison **tolerance** value of 1e-6 will be used unless otherwise specified. No additional setup is required. Select the "**Compare**" button to generate results.

Unassociated entities will be displayed in one of two categories:

1) Mesh elements not associated with any volume
2) Partially meshed volumes

Clicking on the labels in the tree will cause the entities to be drawn in the graphics window. If "**Draw Without Refreshing**" is selected, the draws will be additive. If "**Draw Without Refreshing**" is not selected, the previous draw will be removed when the current drawn entities are shown.

The underlying Cubit command for the tool is the following:

> **Compare volume <id range> {block <id range> | hex <id range> | tet <id range> [tolerance <value>]**

The command will create three types of groups that contain non-corresponding mesh and/or geometry. The group named "**mesh_with_no_volume**" contains hexes or tets that cannot be associated with any volume. The groups named "**No_meshed_Volume_***" contain the curves of a volume (for display purposes) that is completely void of any hexes or tets. Lastly, the groups named "**Partially_meshed_Volume_***" contain hexes or tets, faces or tris, and curves of volumes that could only be partially associated with mesh. The group is created with these entities so that the user can see the partially meshed regions of the volume.

*Defeature Tool*

The Defeature Tool is capable of removing small irrelevant curves and surfaces. These small curves and surfaces are one of the main sources of low quality elements and meshing failures. Sliver surfaces and curves generally exist at fillets, chamfers, and sliver surfaces at misalignments in imprinted assembly models.

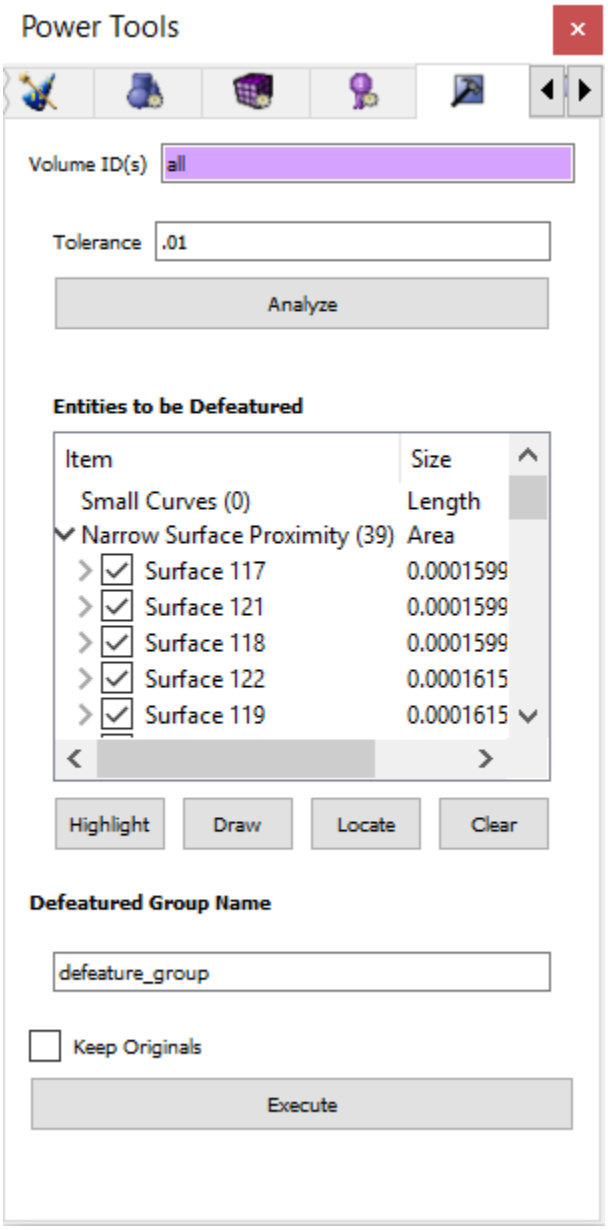



**Figure 1 - Defeature Power Tool**

Defeaturing small curves and surfaces involves three main steps:

1. Analyze the model to automatically detect small curves and surfaces.
2. Manually deselect, if needed, detected small curves and surfaces.
3. Execute the defeature tool to remove small curves and surfaces.

Step 1 requires specifying volume ids (e.g. all) and a tolerance (e.g. 0.6) as shown in Figure 1. Clicking “Analyze” button will automatically find small curves and surfaces in the volumes specified. Figure 2 shows the highlighted small curves and surfaces with the label information. Figure 3 shows a zoom view of a small surface.

In Step 2 the user is allowed to deselect entities by unchecking entities from the list “Entities to be Defeatured”. Users can also use “Highlight”, “Draw”, and “Locate” buttons to examine the automatically detected entities (see Figure 2).
In Step 3 actual defeaturing is performed by clicking the “Execute” button (see Figure 5). Figure 4 shows the zoom view of a defeatured volume. Defeatured volumes are created in a new user specified group (by default in “defeature_group”) as shown in Figure 6. Only the volumes that have small curves and surfaces will be defeatured. Also, by default old original volumes are deleted and new defeatured volumes (child entities) will use the corresponding old ids. Please use the option “Keep Originals” if you want to have both old original and new defeatured volumes.
NOTE:

1. The new defeatured volumes are in MBG format. That is defeatured volumes are facet based instead of NURBS based ACIS volumes. Therefore, it is highly recommended to perform NURBS based operations such as webcut and imprint before calling defeature.

#### Command Syntax:

**Set tolerant mesh mbg only**

This command forces the mesh to associate with new defeatured volume. Currently, this command must be called before calling the defeature command below.

**Defeature curve_length <value> [Curve <ids>] [Curve <ids>] surface_prox2d <value> [Surface <ids> ] [group <id>] [keep]**

**curve_length <value>**: Curves with length less than or equal to <value> are automatically detected as candidate for defeaturing if auto_identify is specified. Otherwise, [Curve <ids>] must be specified.
**surface_prox2d <value>**: Surfaces with narrow region between opposing bounding curves are automatically detected as candidate for defeaturing if auto_identify is specified. The 2d proximity <value> specified in detecting surfaces containing narrow regions. If auto_identify is not specified, then [Surface <ids>] must be specified.
**group <id>**: Defeatured volumes are added to the group id specified.
**keep**: If keep argument is specified original entities are kept along with new defeatured volumes. If keep argument is not specified, then original entities are deleted and new defeatured volumes and its subentities (surfaces, curves, and vertices) will use the ids of original volumes.

#### Preserving Critical Geometric Entities

Before defeaturing the geometry, the user may wish to specify geometry that will be preserved during defeaturing. The below given "Fix" keyword is used to preserve any entity. The user may specify a volume, surface, curve, or vertex to fix.

**Mesh Tolerant Fix [Volume|Surface|Curve|Vertex] <range>**

To reverse the effects of fixing a geometric entity, the user may "free" an entity using the following syntax

**Mesh Tolerant Free [Volume|Surface|Curve|Vertex] <range>**

Example for fixing geometric entities:
reset
brick x 10
brick x .1
move vol 2 x 5
unite all
mesh tolerant fix surf all
mesh tolerant fix curve all
Defeature curve_length .2 curve 31 29 27 26 24 32 13 30 17 28 22 25
surface_prox2d .2 surface 13 14 15 16 12

**Sample Journal File:**

Even though the defeature tool is mainly intended to driven by the GUI, it can be used via command line. Without the GUI, it will be harder to provide the list of small curves and surfaces to the defeature command. Here is a sample journal file:
# import simple assembly
import acis 'assembly11a.sat'
# perform any ACIS based operations such as webcutting and imprinting first
imprint all
merge all
# enable the developer only command
set developer on
# force the mesh to associate with defeatured MBG volumes
set tolerant mesh mbg only
# create a new group to store defeatured volumes
group 'defeatured_vols' add volume all
# perform actual defeaturing by specifying the volume ids, tolerance, and small curve/surf ids.
# defeatured volumes will be placed in the user specified group id and original entities can be
# kept along with new defetured volume using “keep” option.
defeature volume all curve_length 0.3 curve 107 103 102 100 88 85 82 80 9 6 4 2 214 212 211 210 203 200 199 197 188 187 185 183 170 167 164 162 234 232 227 225 254 253 252 251 249 248 243 242 272 271 270 269 265 264 259 258 288 287 286 285 281 280 275 274 304 303 302 301 297 296 291 290 312 311 307 306 surface_prox2d 0.3 surface 47 48 50 51 41 43 40 42 2 4 1 3 111 112 118 120 121 122 124 126 128 129 130 132 134 135 136 138 140 141 142 144 81 82 83 84 88 89 90 91 94 95 96 97 100 101 102 103 group 2 keep
# del any old original volumes if you don’t want it anymore
delete vol 1 to 11
# enable visibility of only defeatured vols

```
vol all vis off
vol all in group 2 vis on
#  set scheme to tetmesh
vol all in group 2 scheme tetmesh
# set mesh size
vol all in group 2 size 1
# mesh defeatured vols
mesh vol all in group 2
# disable developer only command
set dev off
```

**Figures**

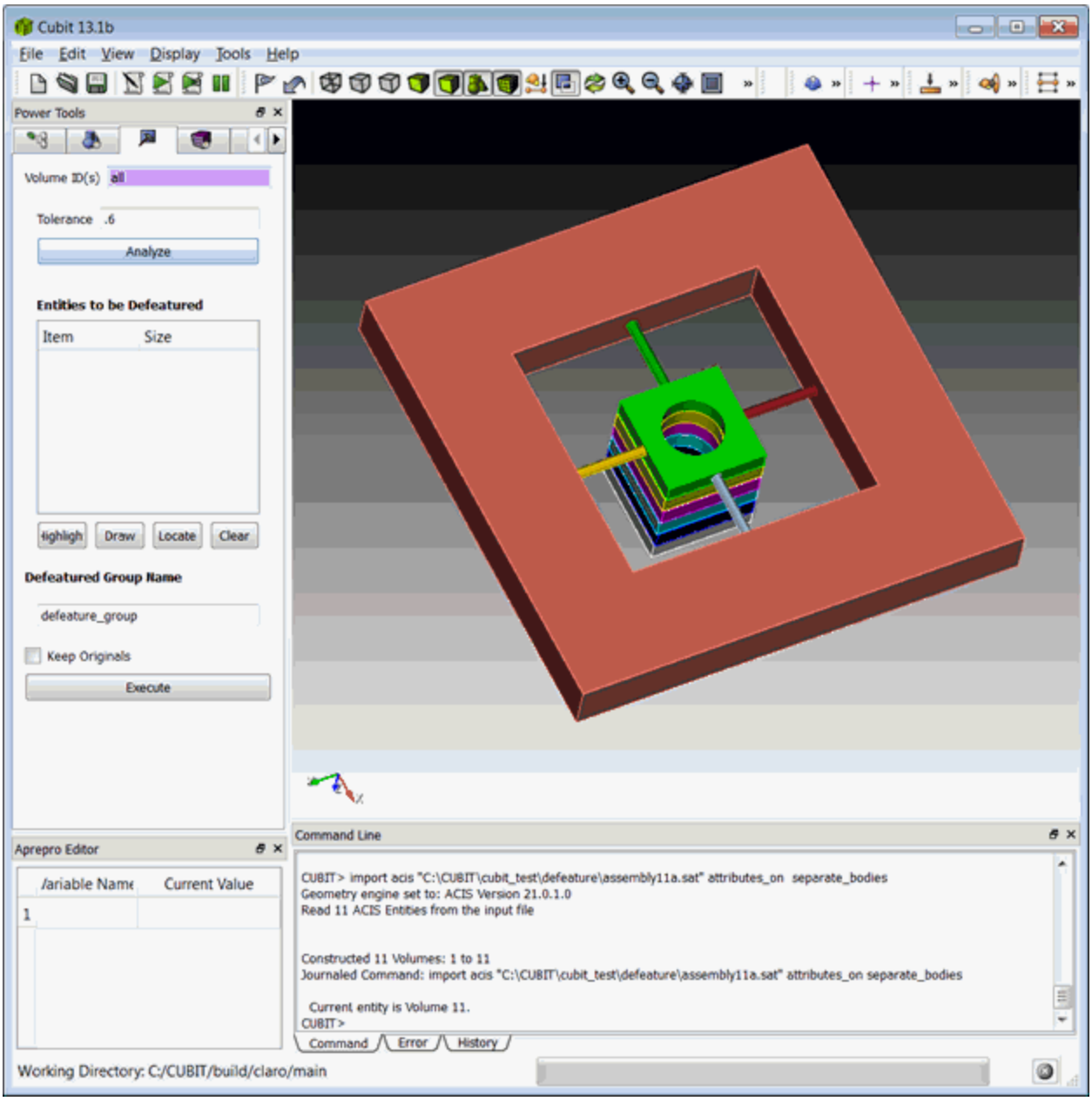



**Figure 1: Specify Volume ID and Tolerance before clicking "Analyze"**

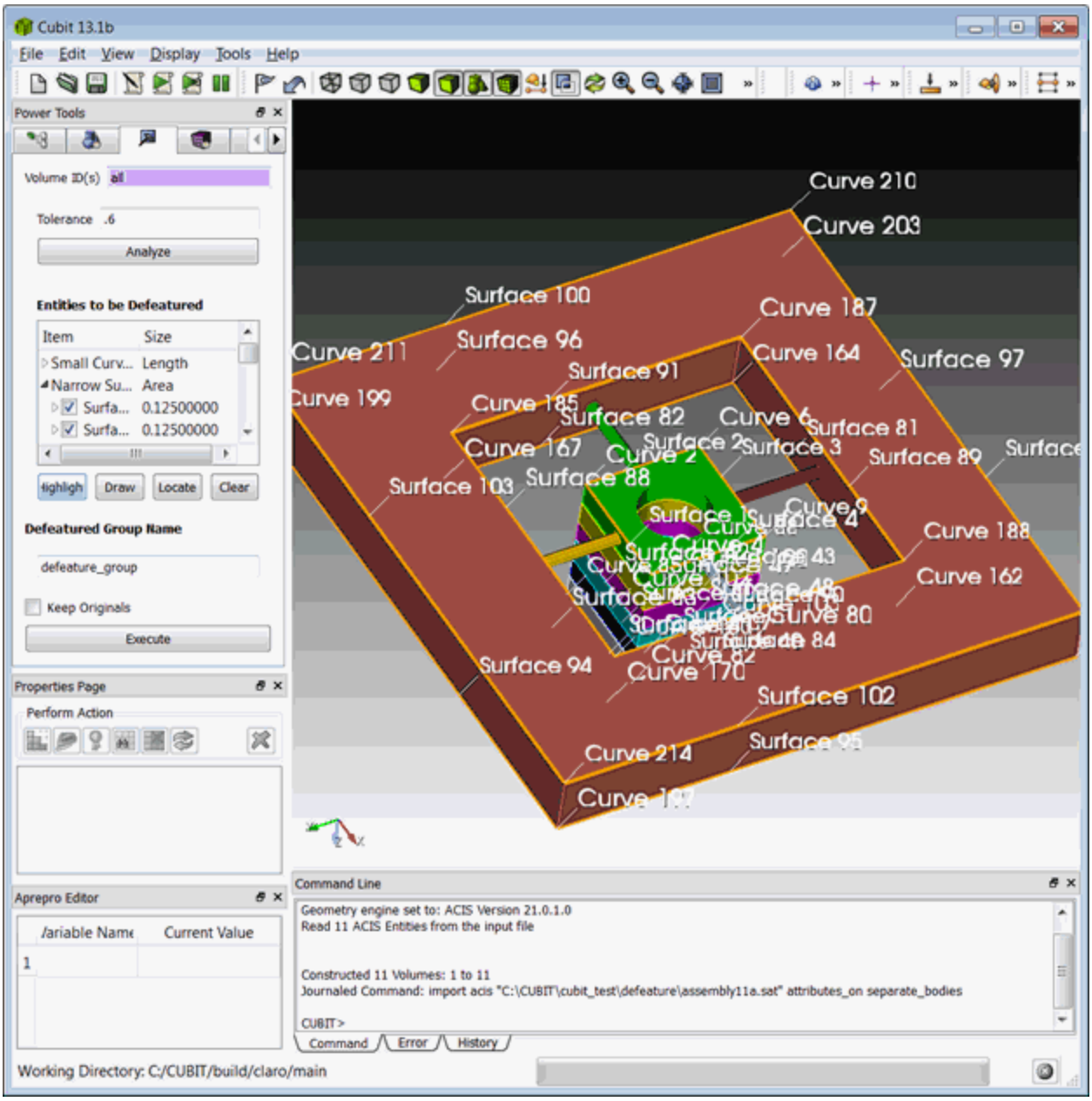



**Figure 2: Use “Highlight”, “Draw”, and “Locate” to visualize small curves and surfaces**

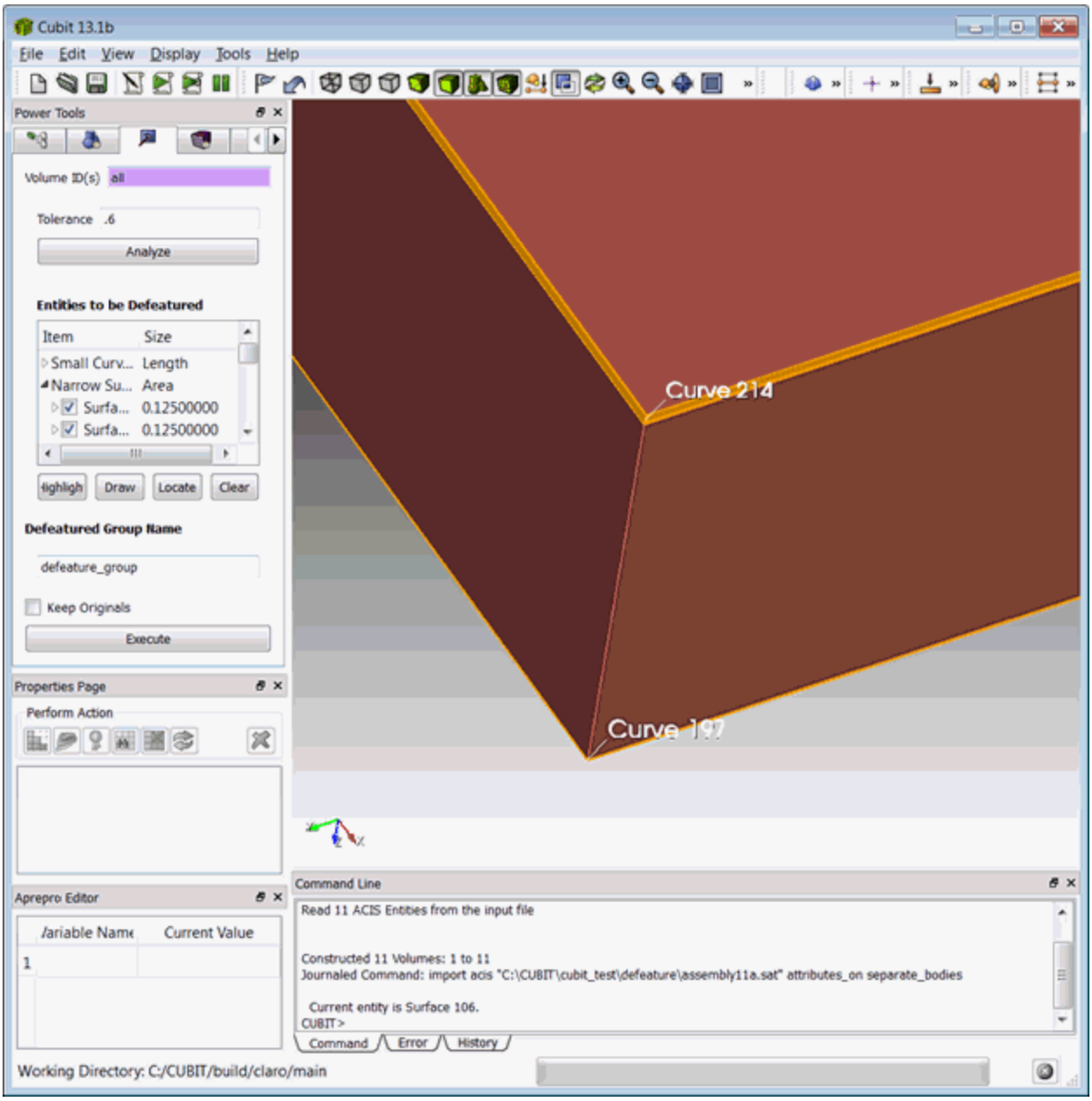



**Figure 3: Zoom view of a small curve and surface**

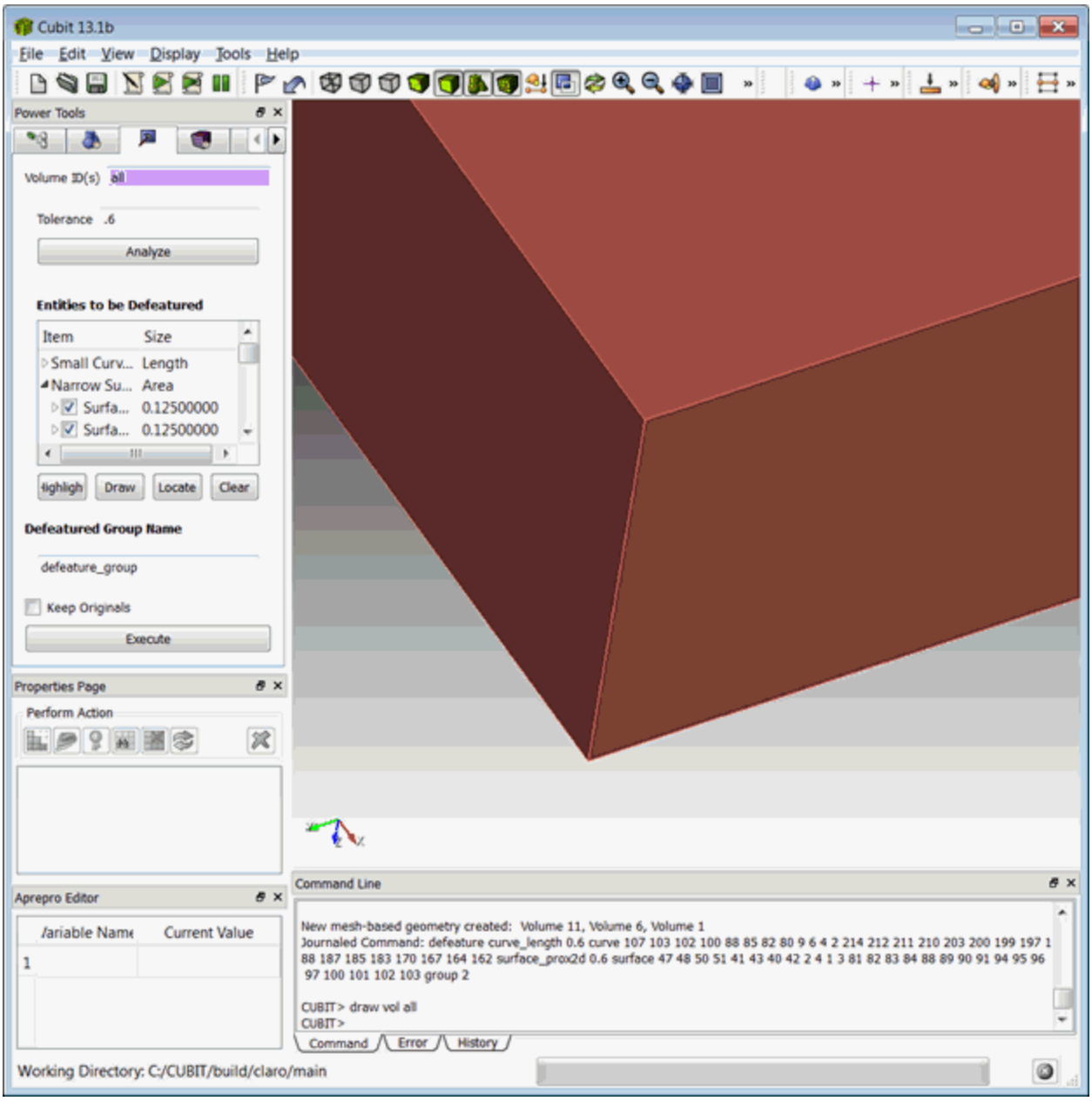



**Figure 4: Zoom view of defeatured volume**

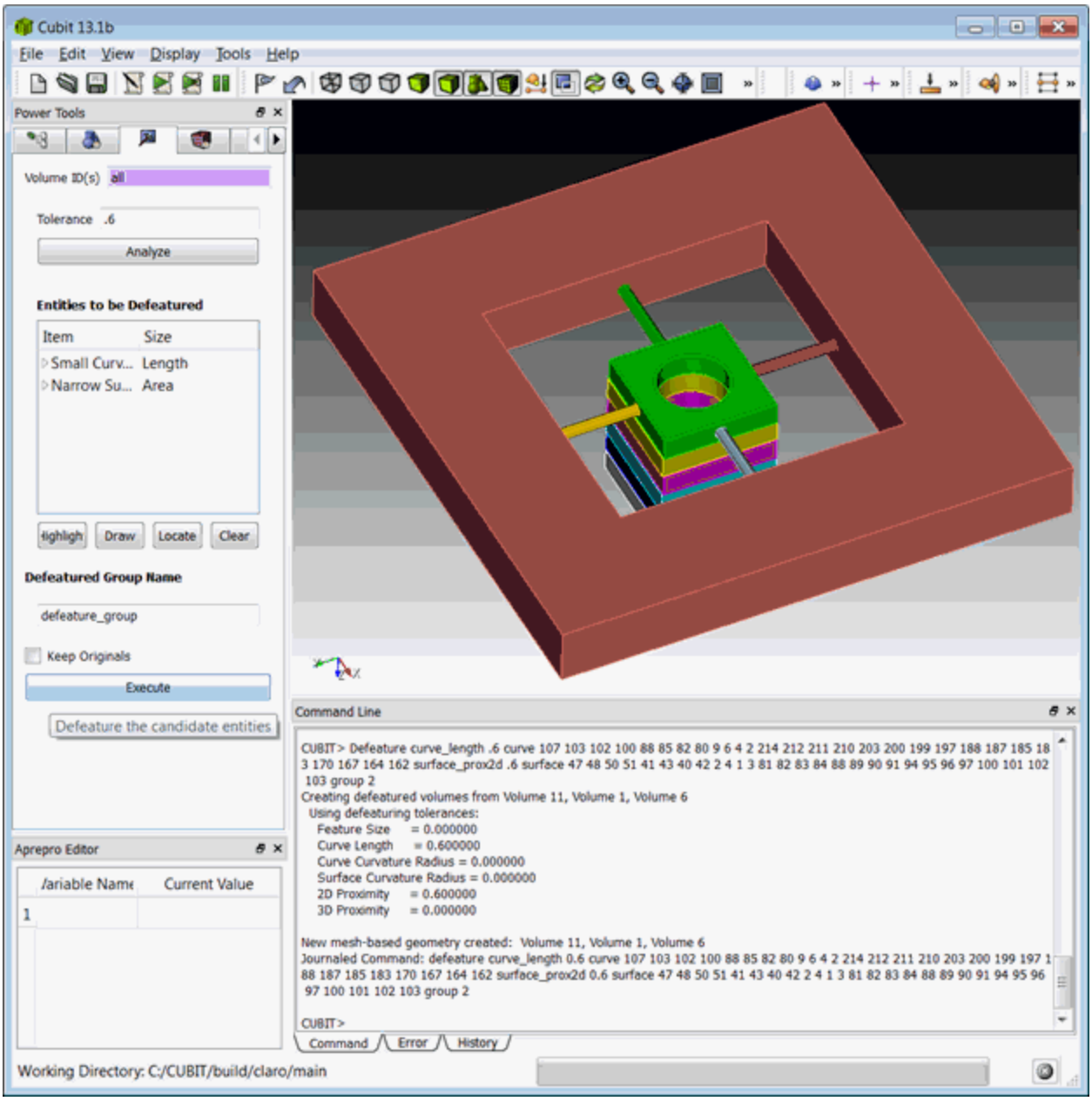



**Figure 5: Click Execute button to Defeature automatically/manually selected entities**

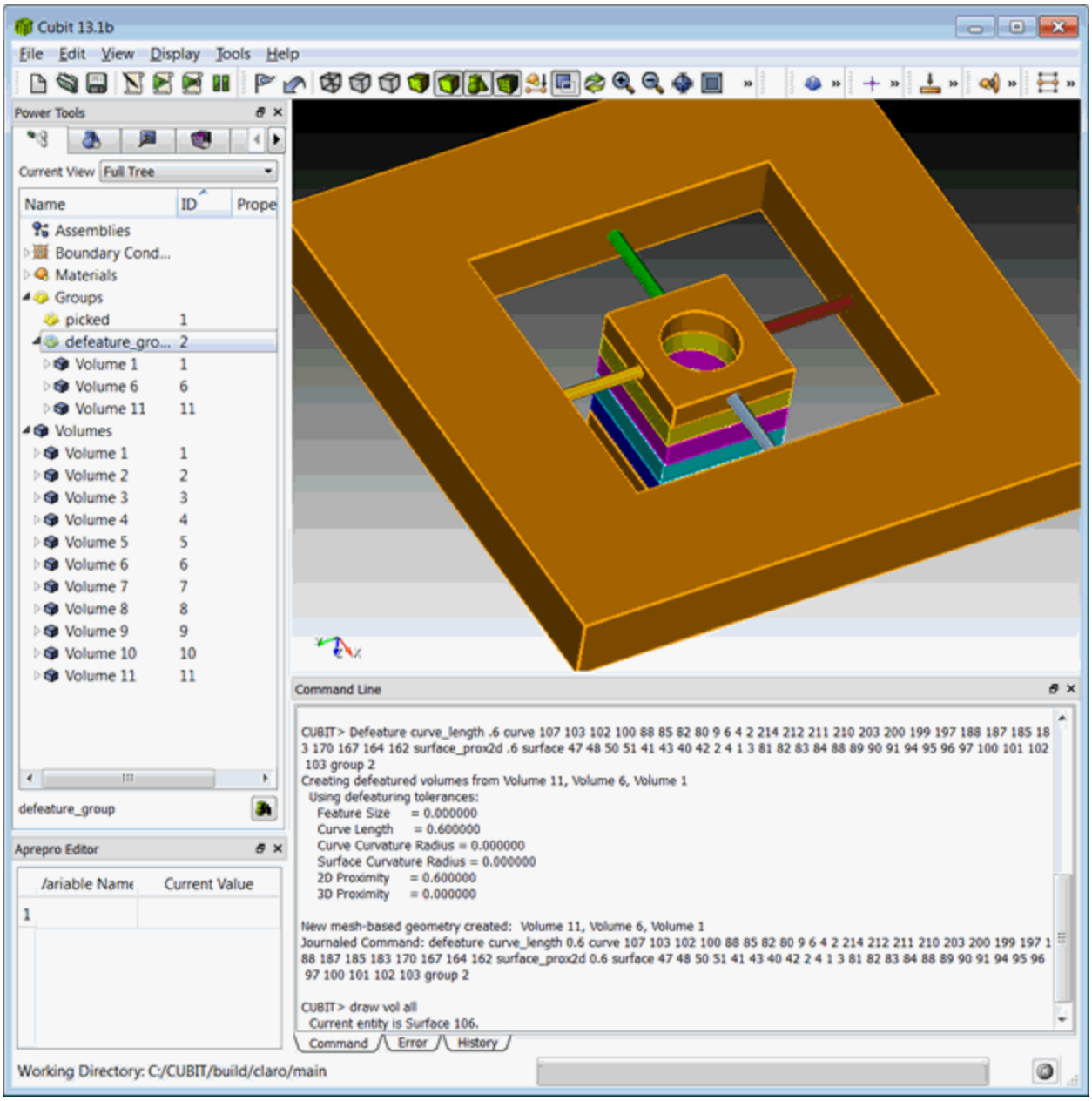



**Figure 6: New defeature_group contains defeatured volumes in MBG format**

# *Beams and Shells with the Geometry Power Tool*

This page describes the beam and shell modeling tools that are part of Cubit's Geometry Power Tool. Geometry simplification is often required when modeling assemblies comprised primarily of thin volumes. Shell finite elements are often used rather than full 3D hex or tet elements. The task for analysts in this case is to reduce the 3D set of thin

volumes to a set of connected sheet bodies where a mesh of triangles or quadrilaterals can be applied. This procedure can be managed with the Geometry Power Tool if the Beam and Shell diagnostic is selected.

**Background**

The Geometry Power Tool in Cubit provides a series of diagnostic checks on your model used to defeature or simplify a CAD model prior to meshing. Clicking the **Analyze** button will perform the selected diagnostic tests and display an expanding tree listing geometric entities that are identified by each test. Once identified, suggested solutions can be easily previewed and executed.
The Beam and Shell diagnostics work best in conjunction with the machine learning models. As such, it is recommended that if using these tools, that the **Load ML Models** button first be selected using the procedure described in the page Machine Learning with the Geometry Power Tool.

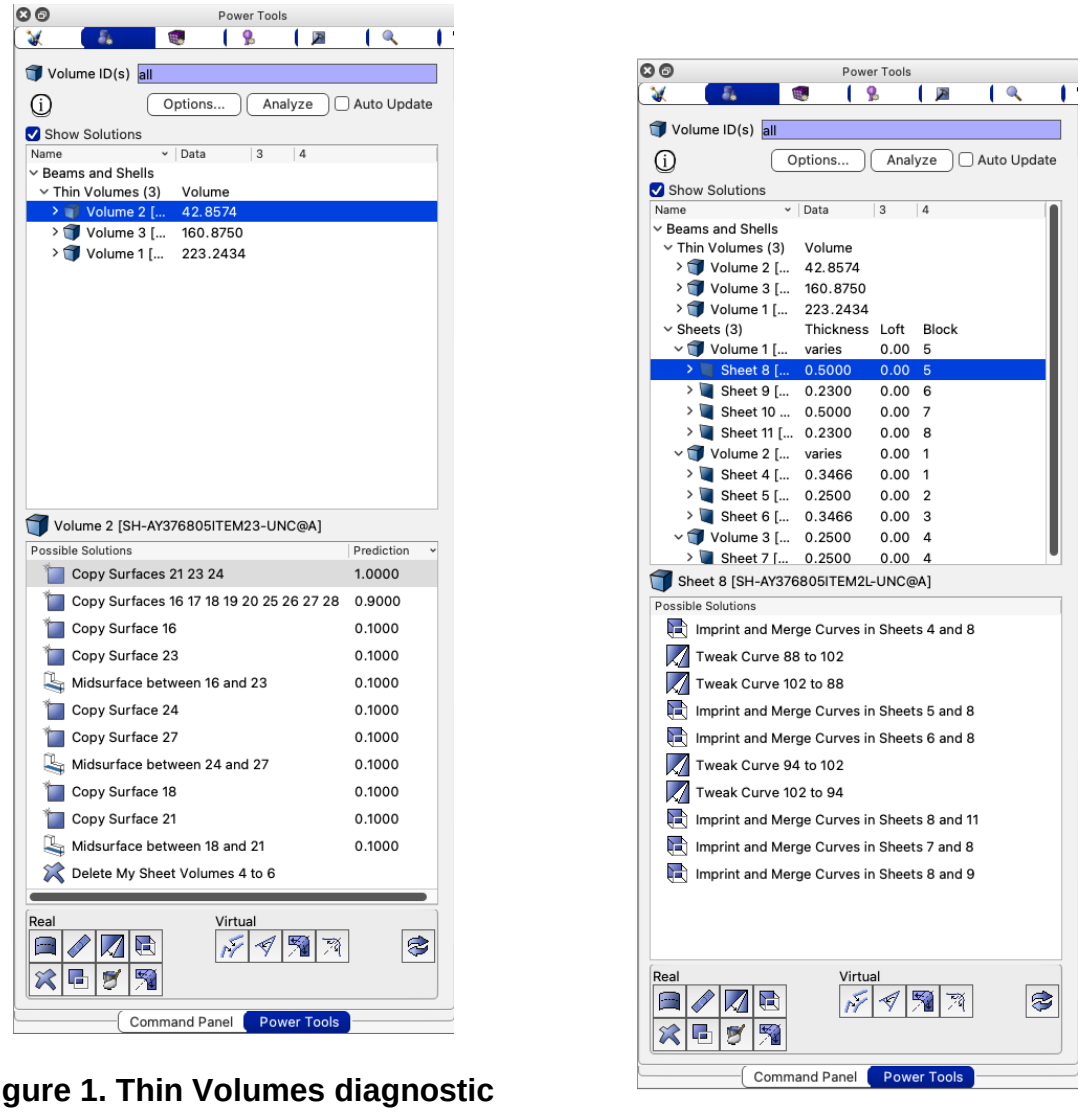



**Figure 1. Thin Volumes diagnostic displayed with Solution window.**

**Reduce solutions show for volume in figure 3.**

**Figure 2. Sheets diagnostic displayed with Solution window. Solutions for connections at one of the sheets are displayed.**

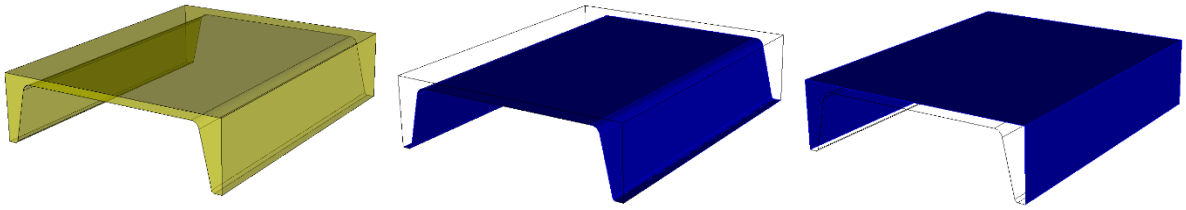


**Figure 3. Two reduce options for the same volume**

#### Thin Volumes

This diagnostic manages the reduction of thin volumes to their 2D or sheet body equivalent. Its primary use is to preview and execute custom reduction operations on selected thin volumes when using the Show Solutions checkbox. The default mode for this diagnostic is to simply list all active volumes. If the machine learning models have been loaded, this diagnostic will be limited to volumes identified as "thin" for its classification category. Figure 1 shows the power tool with an example of a simple 3 volume assembly, where volumes are listed in ascending order based on their geometric volume.

##### Thin Volume Solutions

If **Show Solutions** is checked, selecting a volume in the list will reveal custom solutions for reducing the selected volume to one or more sheet bodies. Figure 3 shows an example of a C-shape volume with its **reduce** solutions displayed in figure 1 in the **Solutions** window. Selecting a solution, will display a preview of the reduction operation. In this example, the first two solutions show the reduced representation previewed in blue. Double clicking on any of the solutions will execute the solution using the corresponding reduce thin command. Note that reducing a 3D thin volume will automatically set up an association from the volume to its child sheet body(s).

##### Deleting Reduced Solutions

If a reduce solution (sheet bodies originating from the volume) is already defined, a **delete** operation will also be available in the custom solution. Double-clicking the **delete** solution will delete the sheet body(s) and remove it as one of its sheet body children.

##### Machine Learning Confidence

If the machine learning models are loaded, a *Confidence* value will be displayed with each custom reduce option. This is a predicted suitability score based on existing training data. The solutions will be listed with the highest confidence listed first. Predictions are based

based on existing training data and can be updated using the Reinforcement Learning tool. They can also be modified to reflect user preferences using the right-click menu option when the solution is selected using either the **Maximize Confidence** or **Minimize Confidence** options.

#### Sheets

This diagnostic will list all volumes defined as sheet bodies. It is most useful when used with the **reduce thin** operations, as sheet bodies will automatically be grouped based on their 3D thin volume parent. If a sheet body was not created with a **reduce thin** command, it will be grouped under a generic "orphan" category.

##### Sheet Attributes

Additionally, when using the **sheets** diagnostic, three new columns for data entry will appear which are used for displaying and updating the **thickness**, **loft** and **block** for the selected sheet body. When using a **reduce thin** operation, the thickness and loft values for the new sheet body will be computed and displayed. Blocks will also automatically be created containing the new sheet body with the thickness and loft assigned as attributes. Clicking on the entries for thickness, loft or block for a given sheet or volume will permit editing of these values. Once edited, the appropriate commands will be issued to update the block attributes. Changing the block ID will also update or create a new block if necessary.
Sheet attributes can also be exported in the form of a spread sheet by right-clicking on the sheet bodies or volumes that should be exported and selecting "Export Sheet Data..."

##### Sheet Solutions

Custom solutions displayed will be **tweak** or **imprint/merge** operations that will attempt to connect nearby sheet bodies. For example, when selecting a solution for **tweak**, a preview similar to figure 4 may be displayed. Double-clicking will execute the **tweak**, extending the surfaces to touch as shown in figure 5.

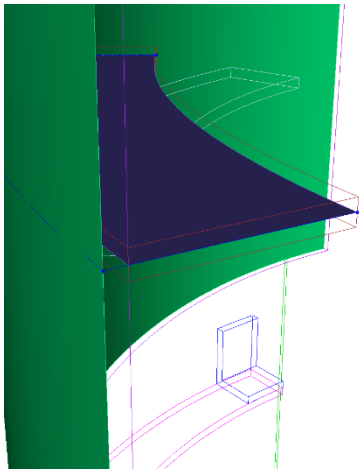


**Figure 4.**

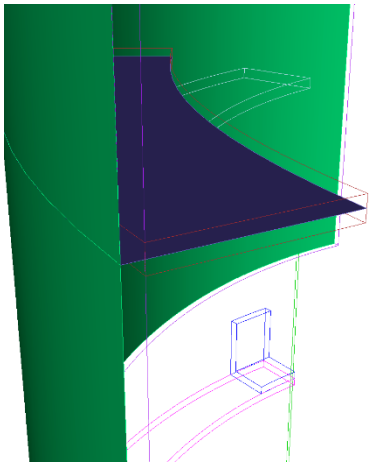


**Figure 5.**

**Right-Click Menu Options**

The beam and shell diagnostics provide many of the same visualization options available with the other diagnostics. A few additional tools, useful for shell modeling, are also provided and described here:

**Draw Sheets [Add]**

Draws the selected sheets in the current graphics display mode (usually smooth shade) and its parent volumes in wire frame. If right-clicking the "sheet" or "thin volume" categories in the diagnostic window, all sheets and their parents will be displayed.

**Draw Sheets with Thickness [Add]**

Draws the sheets similar to above, but also displays the sheet as a 3D transparent volume extrusion, where the thickness of the extrusion is defined by the current assigned thickness attribute. The transparent extruded volume will also reflect the current loft attribute for the assigned block. Note that this is similar this is similar to using the command line **draw shell volume**. This tool is useful for visually validating the currently assigned loft and thickness values.

**Select/Locate Merged Curves**

Selecting this tool will highlight or locate all curves from the selected sheets that are merged with a neighbor. This is useful for checking that the intended connections have been made. Note that this option will only select merged curves on sheet bodies that are children of neighboring 3D thin volumes. (It is assumed that sheet bodies on the same parent volume will be merged)

**Select/Locate Unmerged Curves**

Similar to the previous option, but finds connections that have not yet been made and highlights the closest curve. This is useful for identifying sheet bodies that are not yet connected but should be.

**Delete My Sheets**

Deletes child sheet bodies from a parent 3D thin volume and removes its association.

**Export Sheet Data...**

Brings up a file-picker dialog and exports the current sheet attribute data to a .csv file. If right-clicking the "Sheets" or "Thin Volumes" category headers, all sheet data will be exported. Otherwise, only the selected sheets will be exported.

**Maximize/Minimize Confidence**

Available when right-clicking on a reduce command in the solution window. If the machine learning models are loaded, a confidence value will be displayed next to each thin volume

reduce command in the custom solution window. This indicates a *suitability* score (0 to 1), with the solutions ordered highest confidence to lowest. In many cases choosing the highest confidence predicted by the ML supervised learning tool is sufficient, however since the predictions are based only on existing training data, or models it has seen before, it is possible that the highest confidence solution is not desirable. If this occurs, the **maximize** or **minimize confidence** right-click menu options can be used.
Selecting one of these options will effectively add or replace existing training data to augment the machine learning model. Executing one of these commands will trigger *retraining*, the ML model will be reloaded and a new predicted confidence displayed. Subsequent operations that use the same or similar reduction will now be updated to reflect the new confidence prediction.
The primary use case for **maximize** and **minimize confidence** is following reinforcement learning. If the RL methods are unable to identify the best solution for a given situation, the user can interveen and provide their own preferences to the learning model.

#### Auto Reduce Thin Volumes...

Available when right-clicking on the "Thin Volumes" diagnostic category. Brings up the command panel for the **reduce thin auto** command populated for all active thin volumes. This command provides an automatic method for reducing multiple thin volumes simultaneously while maintaining connections.

#### Reinforcement Learning...

Available when right-clicking on the "Thin Volumes" diagnostic category. Brings up the command panel for the **reduce thin RL** command populated for all active thin volumes. This command uses machine learning with reinforcement learning methods to build a knowledge base of reduction solutions with their suitability. Use RL in conjunction with the **maximize** and **minimize confidence** options to inject customized user preferences into the RL solutions. It can also be used for predicting an ordered sequence of Cubit commands (journal file) for reducing and connecting all active thin volumes.

# *Machine Learning with the Geometry Power Tool*

This page describes the machine learning tools that are part of Cubit's Geometry Power Tool for tet mesh quality prediction and part classification.
**Background**:
The Geometry Power Tool in Cubit provides a series of diagnostic checks on your model used to defeature or simplify a CAD model prior to meshing. Clicking the **Analyze** button will perform the selected diagnostic tests and display an expanding tree listing geometric entities that are identified by each test. Once identified, suggested solutions can be easily previewed and executed.

This new capability enhances the Geometry Power Tool using state-of-the-art machine learning (ML) methods to predict meshing outcomes, suggest solutions as well as classify certain common part types.

**Enable Machine Learning**:

Use the following procedure to access the new ML capabilities:

1. Navigate to the Geometry Power Tool by clicking on the Geometry icon in the power tools window as shown in Figure 1.

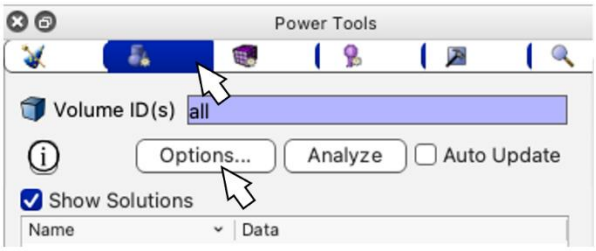



**Figure 1. Navigation to Power Tool and Options panel.**

2. Select the **Options...** button shown in Figure 1.
3. The panel shown in Figure 2. should appear. Click the button labelled **Load ML Models**
4. After the machine learning models are loaded, a variety of diagnostic options become accessible, as illustrated in Figure 3. Available diagnostics encompass "Reduce for Simulation," "Tetmesh Poor Quality Predictions," and classifications for "Parts" and "Surfaces." Furthermore, settings specific to ML-based diagnostics will also be made available for fine-tuning.

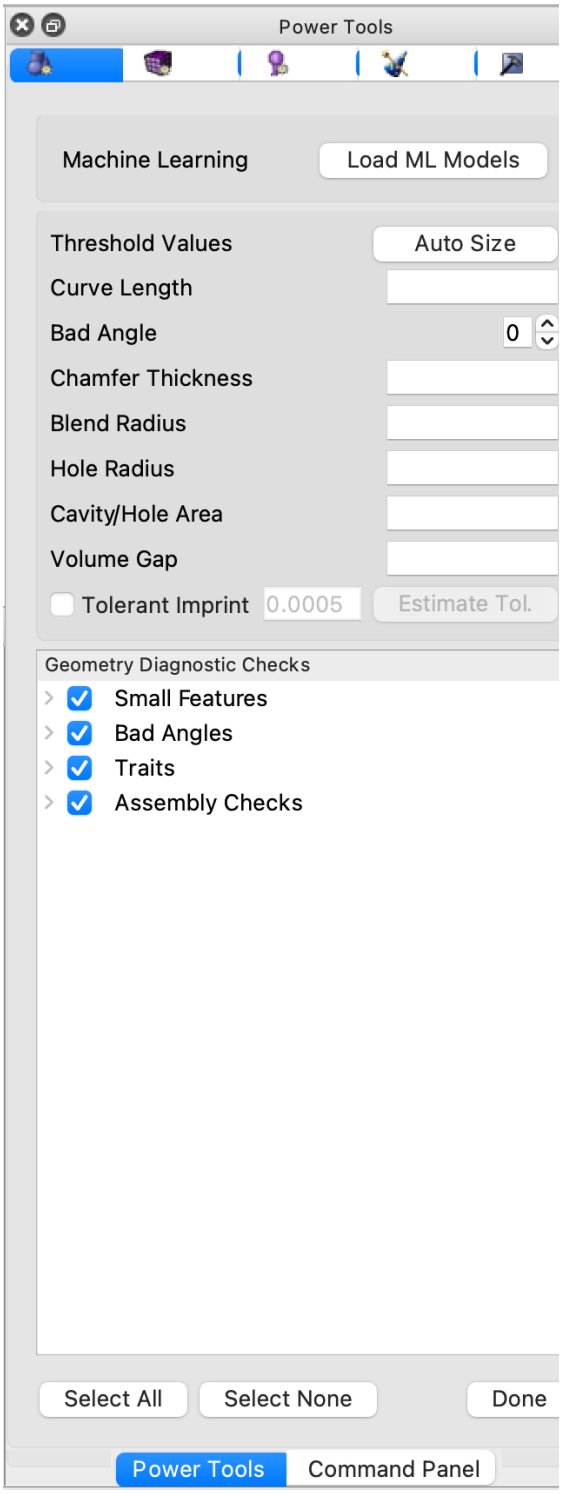



**Figure 2. Options panel without ML models loaded**

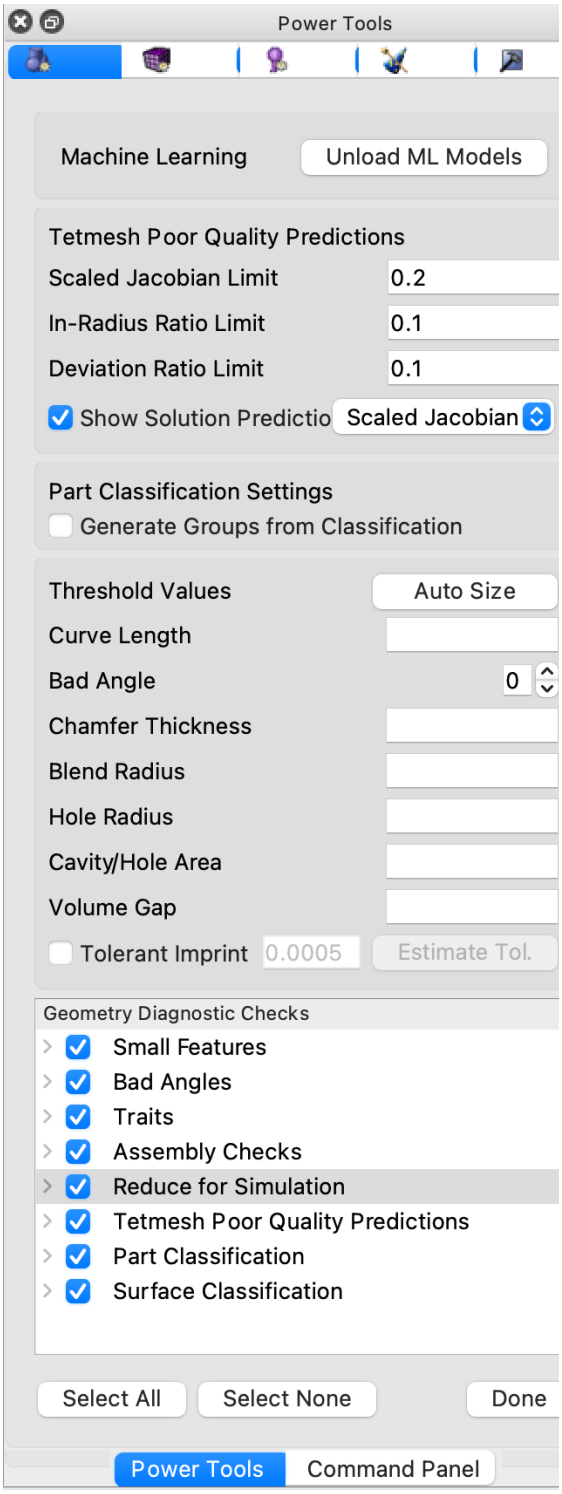



**Figure 3. Options panel after loading ML models.**

**Geometry Analysis Tools**

The following describe the additional diagnostics available when ML is enabled in the Geometry Power Tools.

1. Reduce for Simulation
2. Tetmesh Poor Quality Predictions
3. Part Classification
4. Surface Classification

#### Reduce for Simulation

This diagnostic category leverages machine learning to pinpoint geometric entities crucial for targeted physics simulations. It facilitates the examination, visualization, and contextual simplification into proxy geometry or boundary conditions suitable for simulation. This is achieved through the reduce command suite within Cubit. Supported categories include:

- Fasteners
- Beams and Shells
- Slot Surfaces

#### Fasteners

The "Fasteners" category builds upon the "bolt" section found in the Part Classification tool, streamlining the management of connections within assemblies. This diagnostic is especially useful given the tedious nature of managing fastener connections and the varying proxy requirements for simulation across different physics.
Procedure for Using the Fasteners Diagnostic Tool

1. **Open the Options Panel:** Start by opening the options panel in the Geometry Power Tool.
2. **Enable Reduce for Simulation:** Locate and check the "Reduce for Simulation" checkbox within the options panel.
3. **Select Categories:**
    - Expand the categories under "Reduce for Simulation".
    - Select the categories relevant to your project (Clamped Members, All Bolts, Pilot Holes). Avoid selecting categories that are not required to reduce unnecessary computation time.
4. **Close the Options Panel:** After making your selections, click the "Done" button to proceed.
5. **Optimize Analysis (Optional):**
    - If analyzing only a portion of the model, you can reduce computation time by entering the volume IDs of the portions of the assembly you are interested in.
6. **Initiate Analysis:**
    - Click the "Analyze" button to start the diagnostic process.
    - Note: The analysis might take a significant amount of time for larger models due to the machine learning classification process involved.

7. **Review Results:**
    - Upon completion, the tool will display up to three categories based on the findings, such as identified fasteners or pilot holes, if any.

Bolt Categorization Methods

1. **Clamped Members:**
    - Groups bolts by the members they fasten, naming clamped members according to their ID (e.g., "Volume 1 to 2").
    - A context menu provides options for visualizing, zooming, fly-in, drawing, and locating the clamped members, as well as selecting and visualizing all bolts clamping the two members.
    - Right-click menu options to open command panel for reduction operations, allowing operation on all bolts in clamped members simultaneously
    - Expanding the list under the clamping members' name shows individual bolts, with right-click options available for visualization.
    - Selecting an individual bolt populates the solutions window (see **Show Solutions** checkbox) with custom solutions. Choose to preview or execute directly, or access a command panel for additional options.
    - Right-click option to switch the display to show clamped member names based on their volume or block name (e.g., "PartA to PartB") rather than ID (e.g., "Volume 1 to 2"").
2. **All Bolts:**
    - Categorizes bolts by name, derived from step or ACIS file definitions, facilitating operations on bolts with similar characteristics.
    - Selection and visualization of all bolts with a common name are supported, with listing all bolts without further categorization if names are not provided.
3. **Pilot Holes:**
    - Lists concentric hole pairs for potential fastener application, useful in cases where CAD models do not provide fasteners or they have been removed.
    - Groups pilot holes according to their clamped members, with options for visualizing and operating on all pilot holes simultaneously.
    - Selecting an individual pilot hole populates the solutions window (see **Show Solutions** checkbox) with custom solutions. Choose to preview or execute directly, or access a command panel for additional options.

Below is an illustration of the Geometry Power Tool, showcasing the three categories and sample geometry with highlighted clamped members and associated bolts.

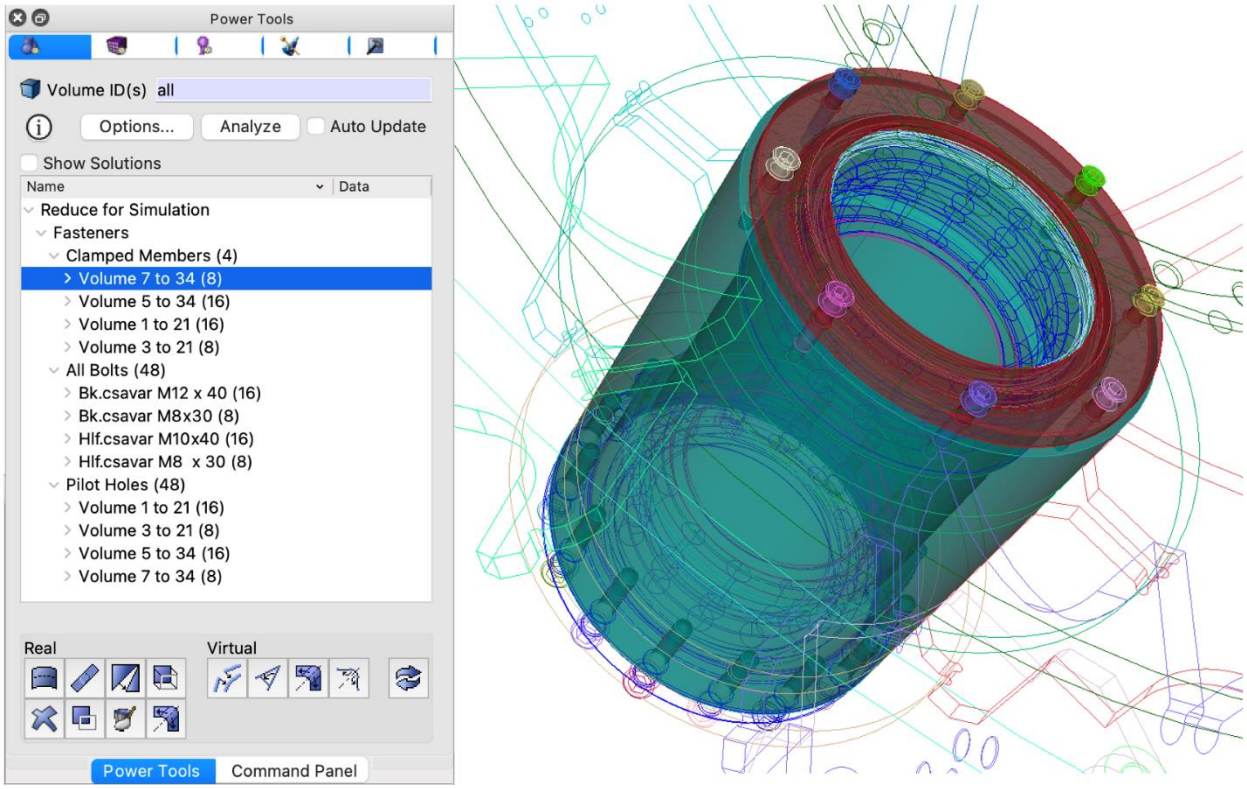



**Figure 2. Fasteners diagnostic shows clamped members with their associated bolts.**

#### Beams and Shells

Geometry simplification is often required when modeling assemblies comprised primarily of thin volumes. Shell finite elements are often used rather than full 3D hex or tet elements. The task for analysts in this case is to reduce the 3D set of thin volumes to a set of connected sheet bodies where a mesh of triangles or quadrilaterals can be applied. This procedure can be managed with the Geometry Power Tool if the Beam and Shell diagnostic is selected. See Beams and Shells with the Geometry Power Tool for details.

#### Slot Surfaces

The Slot Surfaces diagnostic is a new addition to the geometry power tool for preparing slot surfaces in Electromagnetic (EM) modeling. A slot surface is used to identify potential pathways where EM radiation can potentially exit. This diagnostic will help manage slot surfaces and prepare them for the application of boundary conditions and analysis. This diagnostic utilizes Machine Learning (ML) to predict the most likely slot surfaces. See Slot Surface Preparation with the Geometry Power Tool for details.

#### Tetmesh Poor Quality Predictions

The solid model used as the basis for a tetrahedral mesh may contain small features or angles that can lead to poor mesh quality or very small elements. This diagnostic will provide a list of entities (vertices, curves, surfaces) that are predicted to result in poor

quality tet elements sorted by their predicted mesh quality metric. This allows the user to quickly focus on regions of the model that will result in the worst mesh quality and apply geometry operations to improve the meshing outcome without having to mesh.
Three different mesh quality metrics are used for the basis of the predictions: Scaled Jacobian, In-radius and Deviation. Entities identified by these diagnostics will be sorted according to their ML-predicted metric starting with the worst quality entity. The three edit fields at the top of the Options panel control the quality limit. For example, a **Scaled Jacobian Limit** of 0.2 will identify all geometry entities predicted to have nearby tet elements whose Scaled Jacobian is less than 0.2. For our purposes, "nearby tets" are defined as those within two edge lengths of the geometric entity.

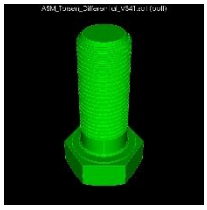
- **Poor Scaled Jacobian** - Select this option if you would like to view mesh quality predictions based on Scaled Jacobian. Scaled Jacobian is a value between -1.0 and 1.0, where 1.0 is ideal and -1.0 is completely inverted. This value is a function of the angles at the vertices of the tet so that values less than or close to zero should be avoided for analysis.
- **Poor In-Radius** - Select this option if you would like to view mesh quality predictions based on In-Radius. The in-radius is defined as the radius of the largest inscribed sphere in a tetrahedron. We are interested in a measure of how closely the tet elements meet the user-prescribed mesh size. Since in-radius is an absolute distance, we use the in-radius ratio defined as **in-radius/ideal-inradius**. The ideal in-radius is defined as the radius of an ideal tet (Scaled Jacobian = 1.0) whose edge lengths are precisely the user-specified mesh size. Therefore, a value of 1.0 meets exactly the mesh quality requirement, while a value less than 1.0 is smaller than prescribed. To determine the ideal in-radius current mesh size defined on the local volume will be used. if not set, then a target of **4*small curve threshold** will be used.
- **Poor Deviation** - Select this option if you would like to view mesh quality predictions based on Deviation. Deviation is defined as the distance from the midpoint of a triangular face at the boundary to its closest point on the geometry. A large value for deviation indicates that the mesh does not conform well or deviates significantly from the geometry. As deviation is also an absolute distance, we define the deviation ratio as deviation/mesh size. A value close to 0.0 for deviation ratio is better as it indicates the triangular faces very nearly match the geometry. A larger value closer to 1.0 indicates the local deviation from the geometry is on the order of the mesh size. To determine the deviation ratio, the current mesh size defined on the local volume will be used. If not set, then a target mesh size of **4*small curve threshold** will be used.

**Context Menus:** To display all entities in a specific category, select the diagnostic category title and use the right click context menu to choose **Draw**. Expanding the list will display individual entities that can be visualized using standard draw tools such as **Zoom To**, **Fly In**, **Draw Owning Volume**, **Draw with Neighbors**, etc. When one or more entities are selected in the list, a context menu also provides access to selected command panel options. When multiple entities are selected, the context menu can be used to invoke command panels to operate on multiple entities simultaneously.

**Part Classification**

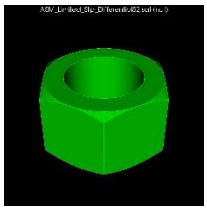
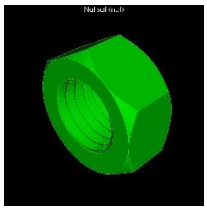
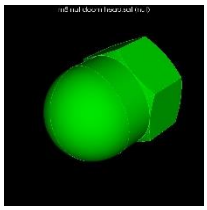
This diagnostic provides the ability to classify volumes according to several common part types. When selected, characteristic geometric features of each volume will be computed and ML methods will be used to determine the part classification. Each volume will be placed into a list based on its most probable categorization. Volumes in each classification will be ordered based upon a **Confidence** metric. A confidence of 1.0 indicates a 100% confidence of categorization. Confidence values closer to 0.5 are less likely to be categrized correctly, but ML indicated a higher probablility than other part types. The current part categories are illustrated below showing a few examples from each category. In addition to these Cubit's classify command can manage custom creation and editing of part categories.

Bolts

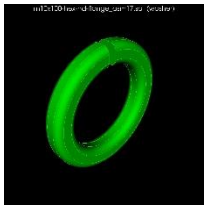
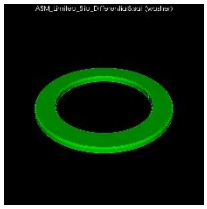
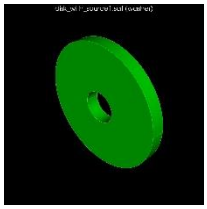




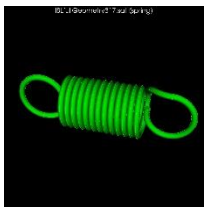
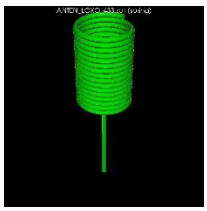

Nuts

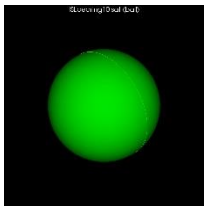
Washers

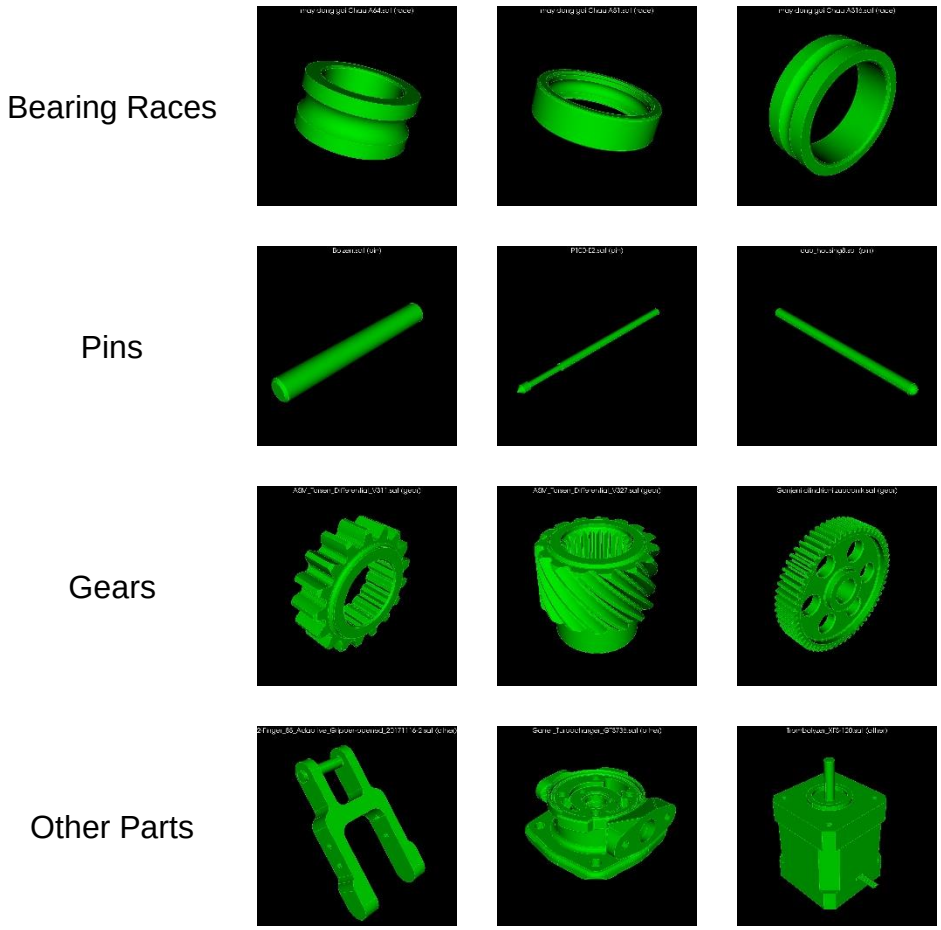




Springs


Balls






Note that the **Other Parts** category simply includes parts that cannot be categorized with sufficient confidence in any of the other categories.

**Context Menus:** To display all volumes in a specific category, select the part category title and use the right click context menu and choose **Draw**. Expanding the list will display individual volumes that can be visualized using standard draw tools such as **Zoom To**, **Fly In**, **Draw Overlapping Volumes**, **Draw Nearby Volumes**, etc. The context menu also provides access to command panel options such as **Delete** or **Reduce**. When multiple volumes are selected, the context menu can be used to invoke command panels to operate on multiple volumes simultaneously.

To view the volume features or the confidence values computed by ML, use the right click context menu when a part/volume is selected and choose either **List ML Features** or **List ML Predictions** menu option.

#### Surface Classification

This diagnostic provides the ability to classify surfaces. Surface categories provided with Cubit are currently limited to two categories: **slots** and **non_slots**. The limited categories are intended to be exemplars for additional user customized categires that can be defined using Cubit classify command. Similar to part classification, when selected, characteristic geometric features of each surface will be computed and ML methods will be used to determine its categorization. Each surface will be placed into a list based on its most

probable categorization. Surfaces in each classification will be ordered based upon a Confidence metric. A confidence of 1.0 indicates a 100% confidence of categorization. Confidence values closer to 0.5 are less likely to be categrized correctly, but ML indicated a higher probablility than other part types.

### Solution Predictions

#### Tetmesh Poor Quality Predictions

Once entities have been identified, custom solutions may be displayed by checking the **Show Solutions** check box above the list of entities. When checked, the Solution window will appear as shown in Figure 4. When an entity is selected in the upper window, an ordered list of solutions will appear in the Solution window. Each solution has an associated metric prediction so that the solution predicted by ML to provide the best possible meshing result will appear at the top of the list.

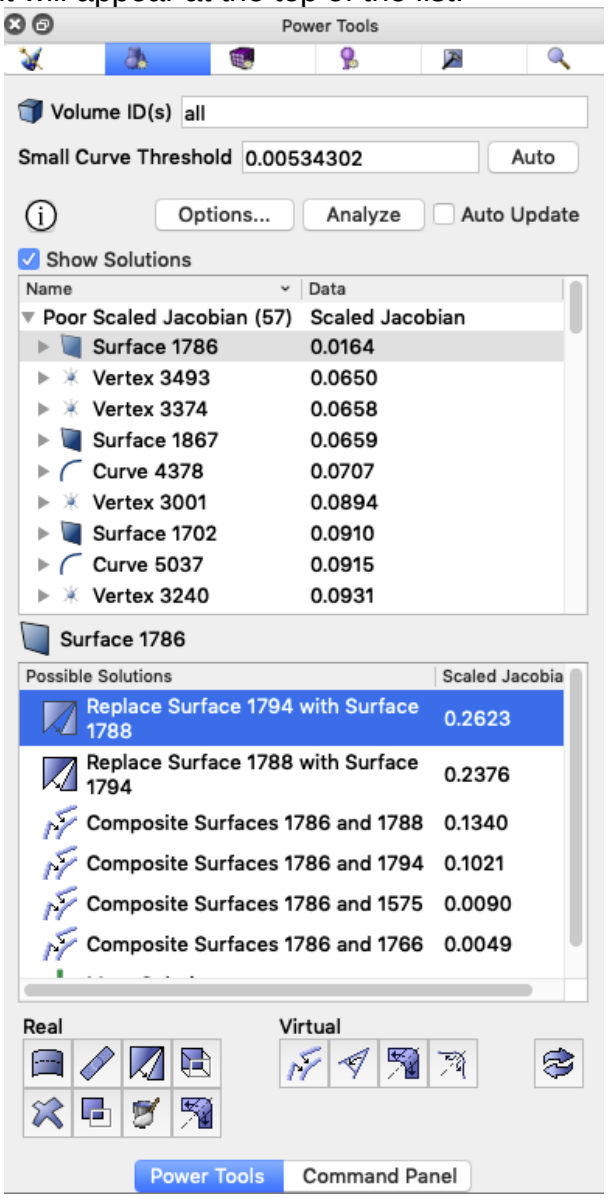


**Figure 4. Geometry Power Tool showing Scaled Jacobian quality predictions as well as an ordered list of possible solutions.**

For example, in Figure 4. Surface 1786 is selected which has a predicted minimum Scaled Jacobian metric at the surface of 0.0164. In most cases this metric would be unacceptable for analysis. To correct for this, the first solution in the list is a **Replace Surface** command that is predicted to result in a Scaled Jacobian of 0.2623. Selecting the solution will display a preview of the operation and double clicking will execute the solution. Other solutions may also be previewed and considered, however those with lower predictions would most likely be avoided.

**Context Menus:** A context menu is available when selecting a solution. This also provides an option for execution of the solution as well as additional visualization options. It also provides direct access to the appropriate command panel if further customization of the command is desired.

**Controlling Solution Predictions:** When selecting an entity from one of the metric categories (Scaled Jacobian, In-Radius, Deviation), the predicted value and ordering of the solutions will be based on the corresponding category. Note that solutions may also be generated for other diagnostics (ie. small curves, close loops, blend surfaces, etc.). When ML has been enabled, solutions will also be ordered these categories based upon the predicted tetmesh outcome. The metric used for ordering solutions can be customized using the **Options** panel shown in Figure 3. Use the **Show Solution Predictions** check box to toggle the use of ML-predictions to order solutions. When solution predictions are enabled, a drop-down selector will allow selection of one of the three metrics to prioritize solutions.

#### Part Classification

When selecting a volume from a classification category, selected solutions will also be displayed. Custom solutions based upon a specific part type are still in development and will expand in future versions of the geometry power tool. In particular, the reduce family of commands are available when one or more volumes classified as a "bolt" or "spring" are selected.

# *Slot Surface Preparation with the Geometry Power Tool*

This page describes the slot surfaces diagnostic tool that is part of Cubit's Geometry Power Tool. The Slot Surfaces diagnostic is a new addition to the geometry power tool for preparing slot surfaces in Electromagnetic (EM) modeling. A slot surface is used to identify potential pathways where EM radiation can potentially exit. This diagnostic will help manage slot surfaces and prepare them for the application of boundary conditions and analysis. This diagnostic utilizes Machine Learning (ML) to predict the most likely slot surfaces.

**Background**

The Geometry Power Tool in Cubit provides a series of diagnostic checks on your model used to defeature or simplify a CAD model prior to meshing. Clicking the **Analyze** button will perform the selected diagnostic tests and display an expanding tree listing geometric entities that are identified by each test. Once identified, suggested solutions can be easily previewed and executed.
The Slot Surfaces diagnostics work best in conjunction with the machine learning models. As such, to use this diagnostic the **Load ML Models** button must first be selected using the procedure described in the page Machine Learning with the Geometry Power Tool.

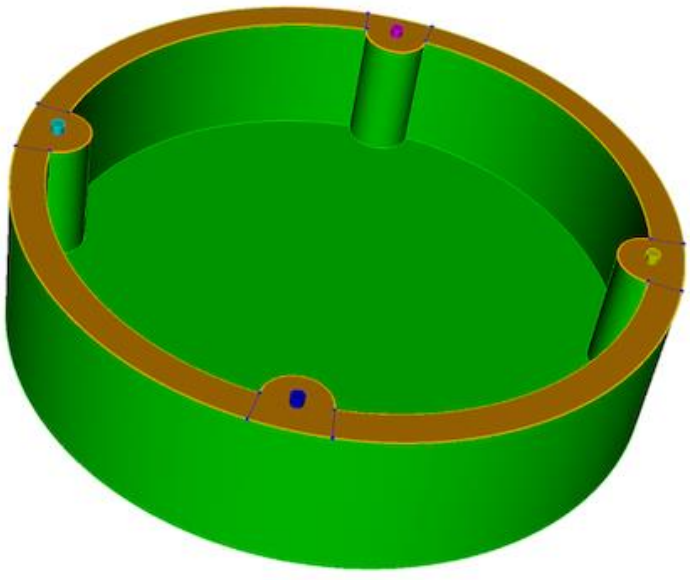


**Figure 1. Example volume showing slot surface highlighted.**

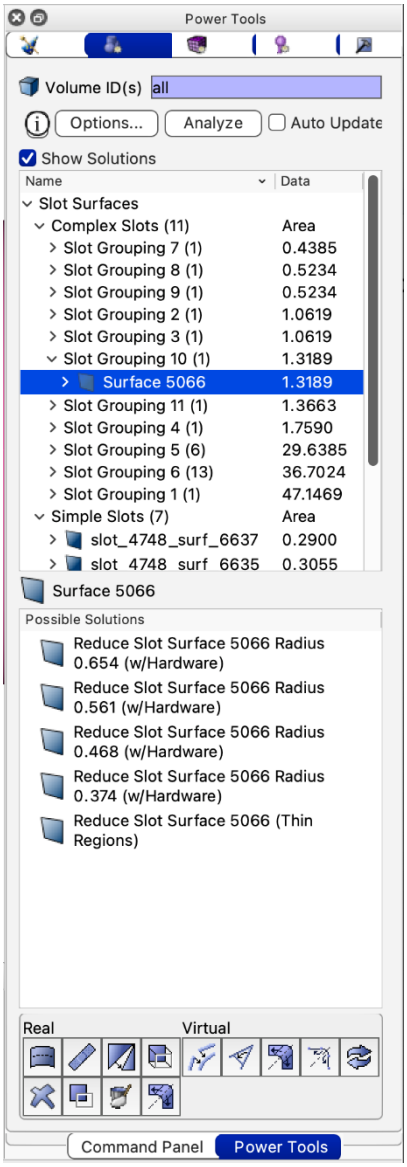



**Figure 2. Slot Surfaces diagnostic displayed with Solution window.**

#### Slot Surfaces

The Slot Surfaces diagnostic is only available when the ML Models have been previously loaded in the Options panel. Once the ML models are loaded, select the Slot Surfaces diagnostic from the list of diagnostic tools. When this diagnostic is selected, the analyze button will identify all surfaces that are identified as slots. The surfaces will be separated into two groups: simple and complex.

1. **Simple Slots:** Simple slots have four sides and can be used directly for representing slots without further decomposition.
2. **Complex Slots:** Complex slots can be comprised of groups of adjacent surfaces or can be a single surface that may require additional decomposition. For example, a gasket-type surface between two volumes that may be held together with bolts or another type of fastener. Each grouping of surfaces in the complex category can be expanded and individual surfaces selected. Selecting a surface will display potential reduce operations in the solution window.

#### Solutions Window

If the **Show Solutions** checkbox is selected, a list of potential decomposition strategies will be listed in the solutions window. The decomposition strategies will be variations of the **reduce surface slot** command. Selecting a solution will preview where cuts will occur and double-clicking will execute the cuts. If fasteners are present, the ML tools should identify them and use them in the **reduce surface slot** commands presented in the solutions window.

#### Right-Click Menu Options

The slot surfaces diagnostics provide many of the same visualization options available with the other diagnostics. A few additional tools, useful for shell modeling, are also provided and described here:

##### Export Slot Data

This option supports the setup of a Morph input deck and will execute the command **export slot data**. Morph is an external meshing tool developed at Sandia. Morph supports slot surfaces for modeling EM, but requires named surfaces and curves to identify the correct topology. When selected, a file browser will appear where a filename can be specified. A text file will then be written with the necessary information needed for Morph input for slot surfaces.

##### List Slot Data

Executes the command **export slot data preview**. Similar to the **Export Slot Data...** menu option, rather than writing a text file, the information will be echoed to the Cubit output window.

##### Reduce Slot Surface

Available when selecting a solution in the solutions window. The command panel for reducing surface slots will appear. This panel provides options to input the hardware and radius information, automatically populating the necessary fields based on the slot geometry. Additionally, users have the ability to specify grouping and naming options for the decomposed slot surfaces.

graphics window

# *Graphics Window*

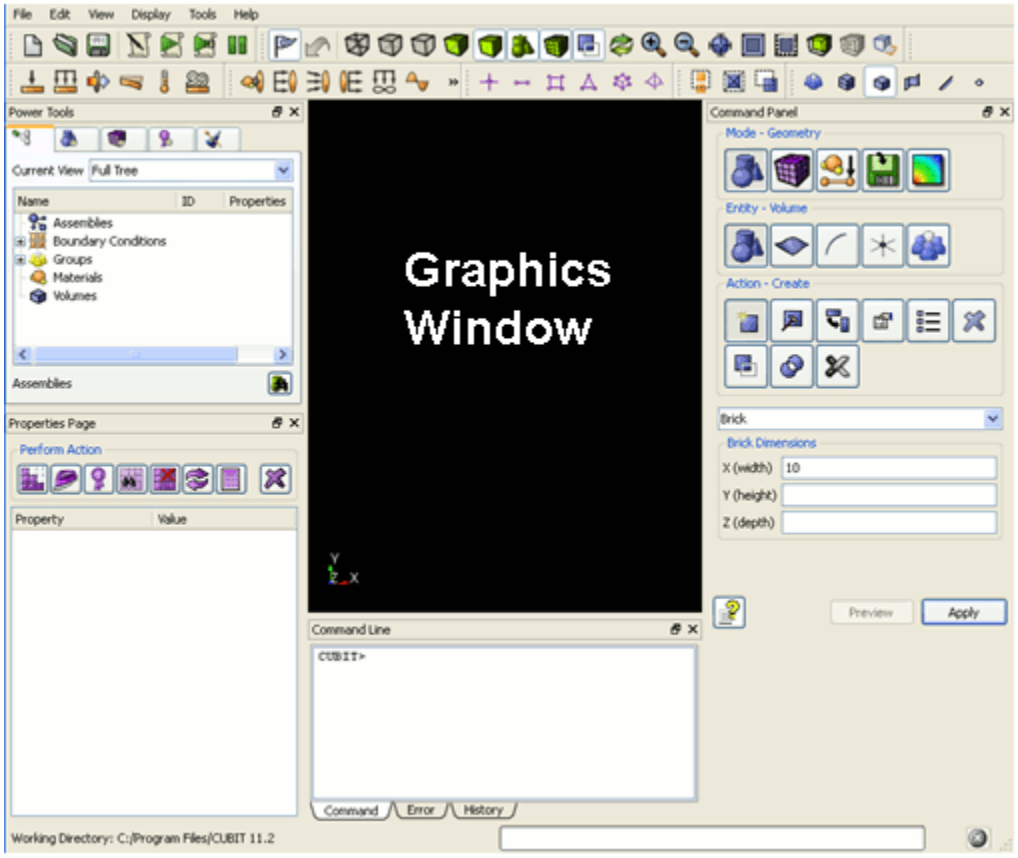



**Figure 1. Graphics Window**

The graphics window is used to view and select entities. Select one of the options below:

- View Navigation
- Selecting Entities
- Key Press Commands
- Right Click Commands for the GUI Graphics Window
- Viewing Curve Valence

# *Viewing Curve Valence*

To view your model based on a color-coded curve valence scale, click on the curve valence button on the Display Toolbar. Curve valence refers to the number of surfaces attached to each curve. Curves with exactly two surfaces attached are shown in blue. Curves with exactly one surface are shown in red. Curves with more than two attached surfaces are shown in white.
This tool is useful for quickly visualizing merged/unmerged topology. Merged curves will usually have a valence > 2, while unmerged curves typically have a valence of 2. Curves with a valence of 1 may indicate a floating surface.

*Key Press Commands for the GUI*

Several commands have a key press shortcut. These commands will be executed with respect to the currently selected entities; see the following table:

| Shortcut Key | Command |
|---|---|
| **l** | List information about the current entity to the output window. |
| **i** | Toggle the visibility of the selected entity (make invisible or visible). |
| **e** | Echo entity id to command line. |
| Tab | Select the next entity. |
| Shift Tab | Select the previous entity. |
| **0** | Toggle picking of vertices. |
| **1** | Toggle picking of curves. |
| **2** | Toggle picking of surfaces. |
| **3** | Toggle picking of volumes. |
| **4** | Toggle picking of groups. |
| Shift **0** | Toggle picking of mesh nodes |
| Shift **1** | Toggle picking of mesh edges. |
| Shift **2** | Toggle picking of mesh faces. |
| Shift **3** | Toggle picking of mesh hexes. |
| **F5** | Refresh graphics window |
| Shift **S** | Activate/inactivate graphics clipping plane |

*Right Click Commands for the GUI Graphics Window*

Clicking the Right mouse button in the graphics window will bring up a menu. One of two menus will appear, depending on whether an entity is currently selected.

**With Entity Selected**

When an entity is selected, the options available will depend upon the type of entity selected. The following describes the menu options and when they are available.

Entity Selections

| Menu Option | Description | Entity Type(s) |
|---|---|---|
| **Select Other...** | Brings up a dialog with alternate entity selections | All |
| **Select Similar Volume** | Selects other volumes with the same geometric volume and number of surfaces | Volumes |
| **Select Similar Surface** | Selects other surfaces with the same area and number of curves | Surfaces |
| **Select Blend Chain** | Selects other surfaces in the same blend chain that have the same radius of curvature | Blend surfaces |
| **Select Cavity** | Selects other surfaces in the same cavity collection. Surfaces bounded by curves where the external angle is greater than 180 degrees. | Cavity surfaces |
| **Select Similar Curves** | Selects other curves with the same length | Curves |
| **Select Enclosure** | Selects all connected surfaces to the current selection(s) on the same volume | Surfaces |
| **Select Between** | If at least two surfaces (or collections of surfaces) are selected, all connected surfaces situated between the two surfaces will be selected. | Surfaces |
| **Select Similar Curves** | Selects other curves with the same length | Curves |
| **Select by Name** | Selects other entities with the same name. Includes entities with the same name prefix prior to the "@" character if it exists. | Geometry entities |
| **Pick Extended...** | Brings up dialog for extended selection. Related entites can be selected using custom criteria defined from a python script. | Surfaces |

Entity Visualization

| Menu Option | Description | Entity Type(s) |
|---|---|---|
| **Zoom To** | Zoom to the selected entity | All |

| | | |
|---|---|---|
| **Rotate About** | Center of rotation is changed to the centroid of the selected entity(s) | All |
| **Locate** | Text label with line pointing to entity is displayed | All |
| **Draw** | Clears the graphics window and draws the selected entity(s) | All |
| **Draw Elements** | Brings up a secondary menu with options to Draw specific 3D mesh entity types (without geometry). | Meshed Volumes |
| **Draw Normal** | Displays an arrow from the center of the selected surface in the direction of its normal. Also colors surface indicating orientation. | Geometry Entities |
| **Draw With Neighbors** | Clears the graphics window and displays the selected entity(s) along with entities sharing a common vertex | Geometry Entities |
| **Draw With Nearby Volumes** | Draws the selected volume(s) along with volumes that are within a small distance | Volumes |
| **Draw Volume Overlap** | Draws the curves in the selected volume(s) as well as any volume it overlaps. The region shared between the overlapping volumes will be displayed in red | Volumes |
| **Isolate** | Turn off visibility for all entities except the selected entity(s) | Geometry Entities |
| **Visibility Off** | Turn off the visibility of the selected entity(s) | Geometry Entities |

Entity Operations

| Menu Option | Description | Entity Type(s) |
|---|---|---|
| **Add to Group/BC** | Opens a dialog box for adding the selected entity to an existing boundary condition or group | All |
| **Remove from Group/BC** | Opens a dialog box for removing the selected entity from an existing boundary condition or group | All |
| **Mesh** | Mesh the selected entities using the current meshing scheme and size settings | Unmeshed Geometry Entities |
| **Delete Mesh** | Deletes the mesh from the selected entities using the "propagate" option. (Deletes mesh entities on lower order entities.) | Meshed Geometry Entities |
| **Show Quality** | Displays the mesh quality of the selected entities to the output window. | Meshed Geometry Entities |
| **Measure** | When a single entity is selected, displays to the output window the geometric length, area or volume. When two entities are selected, displays the closest distance between the entities. | Geometry Entities |

| | | |
|---|---|---|
| List Information | Show a menu with additional options for listing information about the selected entities. **List Basic Info:** Lists connectivity and associated lower order entities. **Geometry:** Basic info plus additional geometric information such as centroid, bounding box, surface area etc. **List Mesh Info:** Basic info plus additional information about the associated mesh. | All |
| Save Selection As... | Brings up a File Browser to save the currently selected entities as a cub file. | Geometry Entities |
| Delete | Deletes the selected geometry entities | Free Geometry Entities |
| Remove | Removes a surface using the **remove surface** command. | Surfaces in Volumes |

**Without Entity Selected**

| Menu Option | Description | Entity Type(s) |
|---|---|---|
| Refresh Display | Clears temporary graphics a refreshes display of current geometry and mesh. | All |
| Draw Mesh | Displays any existing mesh entities without geometry. | Meshed Entities and Free Mesh |
| Reset Zoom | Reset graphics viewing options to original settings. | All |
| All Visible | Reset visibility status to ON for all entities and display. | All |
| Display Options... | Opens the Display Options Dialog to set graphics window characteristics such as background color, lighting, triad options etc. | All |

# *Selecting Entities in the GUI*

Geometry, mesh entities, and boundary conditions can be selected with the left mouse button directly in the graphics window. Before selecting any entity, however, the correct selection mode must be chosen. This dictates which entity types will be available for selection in the graphics window. The *Select* Toolbars, which are located above the graphics window by default, are used to change the entity selection modes.



**Figure 1. The Selection Toolbars for Geometry and Mesh Entities**

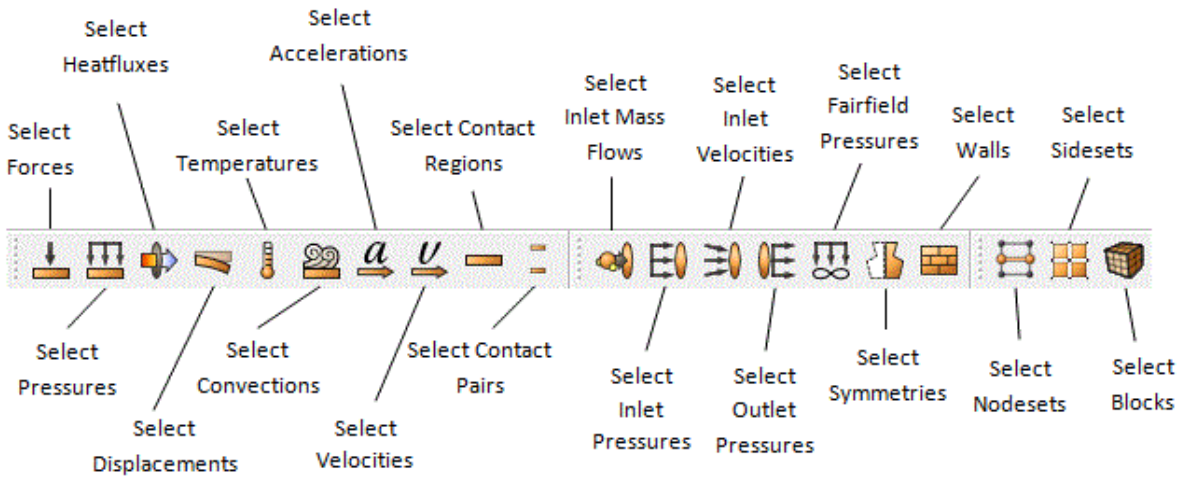



**Figure 2. The Selection Toolbar for Boundary Conditions**

Figures 1 and 2 shows the selection toolbars. Selecting one of the entity selection modes will only permit selection of that particular entity type within the graphics window. These selections will not override a Pick Widget in the command panel.
If both volume and surface entities are picked on the select toolbar, a single click will select the surface while a double click will select the volume. More detailed information on selecting and specifying entities can be found in Entity Selection and Filtering .

#### Pre-Selection

When the mouse cursor is over an entity type that has been selected from the Pick toolbar, that entity will become highlighted. This is called **pre-selection** and is used as a graphical guide to show which entity will be picked when the mouse button is clicked.
Graphics pre-selection may slow down your graphics speed for large models. You can disable pre-selection from the Tools->Options dialog box.

#### Polygon, Circle and Box Select

The polygon/circle/box selection feature allows you to select entities by drawing a box, circle or polygon on the screen. To create a box or circle selection, press and hold the <CTRL> button***** while clicking and dragging the left mouse button. Release the left mouse to complete the box or circle select. To create a polygon selection, press and hold the <CTRL>***** button while clicking and dragging the left mouse button. Click the left mouse button to create another side for the polygon. Press either of the other buttons to close the polygon and complete the selection. Only entities that are in active selection mode will be selected. To change between the polygon, circle or box method, press the *Toggle Between Polygon/Box/Circle Select* button on the Select Toolbar. Clicking the *Toggle Selected Enclosed/Extended* button will toggle between Enclosed Selection and Extended Selection. Enclosed selection will only select entities that are fully enclosed within the bounding box, circle or polygon. Extended selection will select entities that are either fully OR partially enclosed within the bounding box. Toggling the the *Select X-Ray*

will select entities that are hidden behind other entities. X-ray selection will only apply to smoothshade and hiddenline graphics modes.
***Note:** For Mac computers use the command (or apple) button for polygon or box select.

# *View Navigation in the GUI*

There are two different default paradigms for view navigation: Cubit command line and Cubit GUI. The user is allowed to customize the mouse settings as desired. Mouse settings in the GUI are modified by accessing the **Tools** pull-down menu, then select **Options**. The Mouse Settings dialog is shown below (See Mouse-Based Navigation for the command line version).

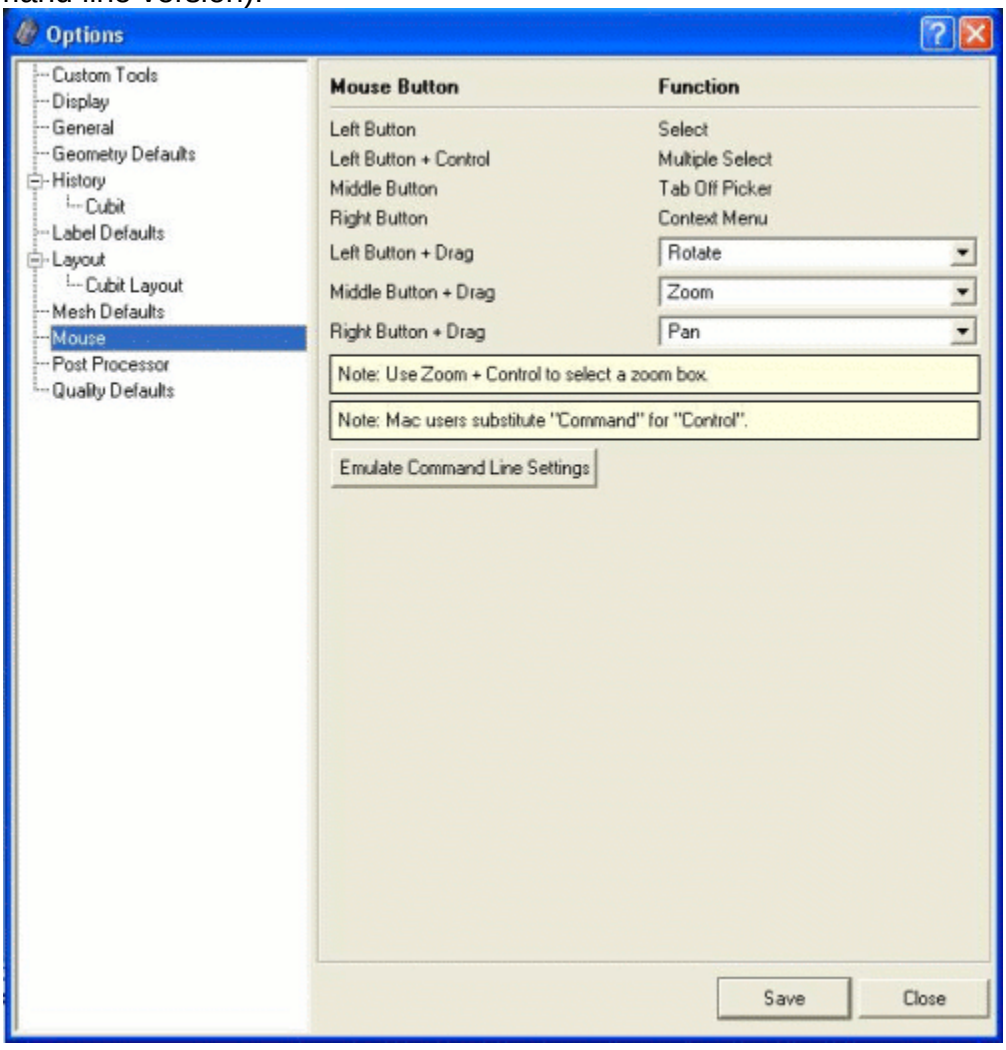



**Figure 1. Mouse Settings Dialog**

**Rotations**

Where the cursor is in the graphics window will dictate how the view will be rotated. If the cursor is outside of an imaginary circle, shown in Figure 2, the view will be rotated in 2d, around an axis normal to the screen. If it is inside the circle, as in Figure 3, the rotations will be in 3d, about the current view spin center. The spin center can be changed to any x-y-z location. The most common way is by zooming to an entity, which changes the spin center to the centroid of that entity. The "view at" command will change the spin center without zooming:

**View at vertex 3**

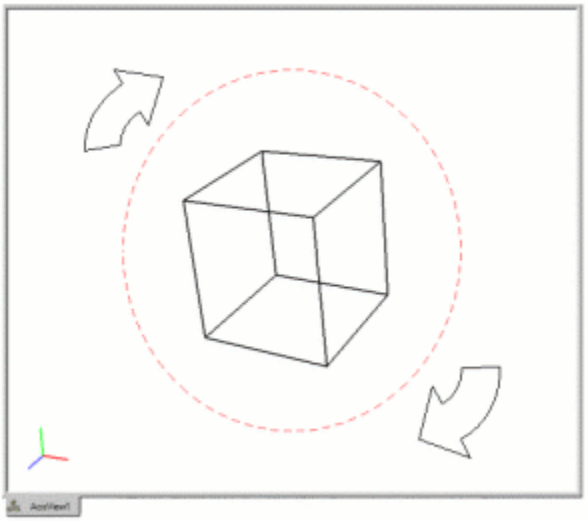


**Figure 2. With the mouse pointer outside the circle the view is rotated about an axis normal to the screen**

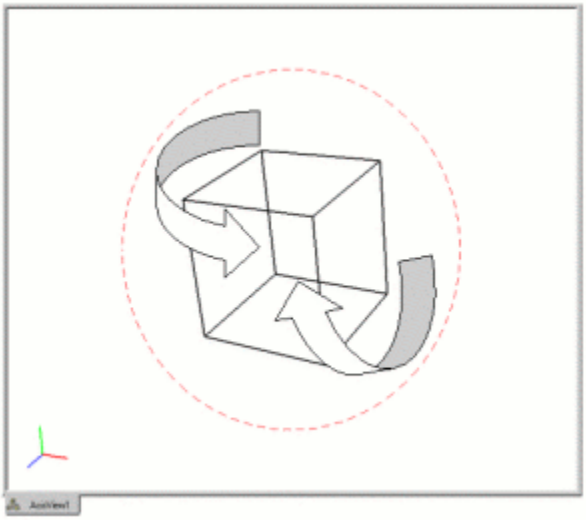


**Figure 3. With the mouse pointer inside the circle the view is rotated about the current spin center**

#### Zooming

To zoom, press the appropriate buttons or keys and move the cursor vertically, as shown in Figure 4. The wheel on a wheel mouse will also zoom.

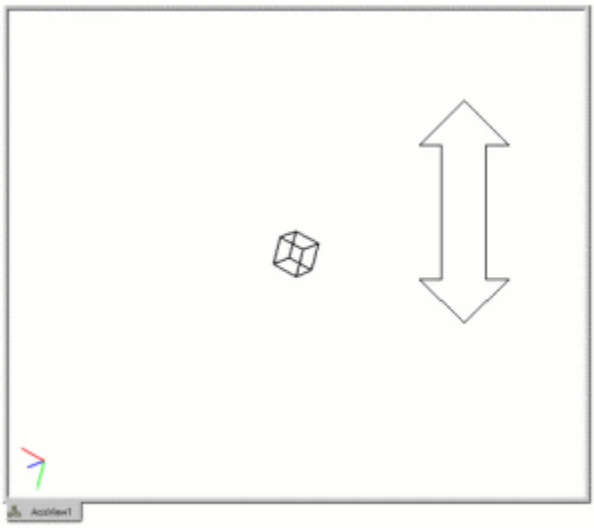


**Figure 4. Move the mouse pointer vertically to zoom in and out**

#### Panning

To pan, press the appropriate buttons or keys and move the cursor horizontally or vertically, as shown in Figure 5.

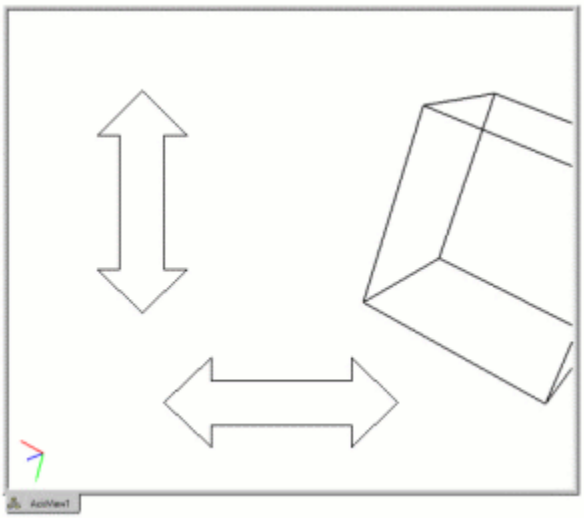


**Figure 5. Move the mouse pointer horizontally or vertically to pan the view**

drop down menus

# *Drop Down Menus*

The Cubit Drop-Down Menus, located at the top of the Cubit Application Window provide access to capabilities such as file management, checkpoints, display manipulation, journaling, system setup, component management, window management, and help.

#### Cubit (Mac Only)

This menu contains the Preferences dialog box, also called the Options dialog box on other platforms. It also contains the About Cubit menu and the Quit Cubit option. It is only available on Mac computers.

#### File

This menu provides common file operations, including importing and exporting of geometry and meshimport and export. A list of recently saved or imported files is also provided, allowing a quick way to import current or recent work. Non-Mac users can also exit and reset the program from this menu (These options are found under the Cubit tab for Mac Users).

#### Edit

This menu only provides a way to enable the Undo feature of the system. If Undo is enabled, one level of Undo is available to the user.

#### View

The View Menu lists all available toolbars and windows in the current CUBIT session. Selecting a toolbar or window will make it visible. Deselecting a toolbar or window will hide it. You can also hide an undocked window or toolbar by clicking on the small "x" in the upper right corner. For more information on docking and undocking toolbars, see CUBIT Application Window.

#### Display

The Display Menu controls display options for the graphics window. These options are explained below:

- **View Point** - Controls the camera view point. Choices are front, back, top, bottom, right, left and isometric views.
- **Render Mode** - Controls visibility modes, including: wireframe, true hidden, hidden line, transparent, and shaded.
- **Geometry** - Controls geometry visibility
- **Mesh** - Controls mesh visibility
- **Graphics Composite** - Controls the visibility of composited entities in the graphics window.
- **Refresh** - Updates the graphics display
- **Background** - Changes the background color
- **Zoom In** - Enlarges the model in graphics window

- **Zoom Out** -Shrinks the model in graphics window
- **Zoom To Fit** - Enlarges or shrinks model in the graphics window so it fills the whole screen
- **Toggle Perspective** - When this option is selected, the entities in the graphics display window are drawn in perspective mode.
- **Toggle Scale** - Turns on or off a graphical scale that can be drawn in the graphics window to obtain a bearing on model or part sizes.
- **Toggle Clipping Plane** - Turns on or off the graphics clipping plane
- **Toggle Clipping Plane Manipulation** - Turns on or off manipulation of the graphics clipping plane
- **Show Curve Valence** - Turns on or off the curve valence highlighting

#### Tools

The Tools Menu contains access to GUI-specific tools and options. These options are explained below.

- **Journal Editor** - Opens journal file editor. The Journal Editor is used to write, edit, play, and save journal files. It can also be used to create and edit Python scripts. A built-in translator will convert between the two files types.
- **Play Journal File** - Plays a specified journal file. You can browse through files and folders on your computer to select the journal file to play.
- **Options** - Opens the Option dialog box. This dialog box controls all of the preferences for the GUI including display colors and widths, mouse settings, journal file options, mesh and geometry defaults, and general layout preferences. MAC users can find this menu under the Cubit tab.
- **Components** - Opens the Components dialog box. This window is used to load and unload external and internal components.

#### Help

- **Tip of the Day** - Open the tip of the day box.
- **Cubit Tutorials** - Opens a menu of step-by-step tutorials for Cubit.
- **Cubit Manual** - Menu to bring up on-line searchable documentation (this document).
- **About** - Menu to show the current version number and trademark information. Mac users can find the version number under the About Cubit menu in the Cubit drop-down.

*Options Panel*

To change program preferences in the Graphical User Interface select: **Tools > Options**. On a Mac OS, the equivalent panel is called **Preferences**, and is available from **Cubit > Preferences** The options panel includes a series of Tabs, containing categories of global settings to customize the Cubit environment. The following categories are available:

- Command Panels
- Display
- General
- Geometry Defaults
- History and Cubit Journalling
- Label Defaults
- Layout
- Mesh Defaults
- Mouse Settings
- Post Processor
- Quality Defaults


#### Command Panels

This menu controls how command panels are displayed and managed, including which style of button hierarchy is displayed.

- **Command Panel Behavior**
  - Automatic Input Field Focus -
  - Automatic Panel Reset
- **Panel Warnings**
  - Hide Mesh Warning - Hide the warning dialog that appears when trying to mesh entities that already contain a mesh.
- **Navigation Hierarchy**
  - Classic (Restart Required) - Toggle between entity-first hierarchy (Classic) and action-first hierarchy (Non-classic). This changes the arrangement and navigation of panels in the geometry mode of the command panel. The classic navigation arranges the icons so that entities are first selected, followed by an action. For the non-Classic mode, the action is first selected followed by the entity type. A restart is required to see changes in the command panel navigation hierarchy.
- **Boundary Condition Modules** - Disable/Enable the display of GUI tools used for managing commercial FEA and CFD model preparation. These options are available only in non-classic navigation hierarchy. Uncheck the **Classic** checkbox to enable these options.
  - FEA (Restart Required) - Enable or Disable the FEA GUI tools
  - CFD (Restart Required) - Enable or Disable the CFD GUI tools

Display Preferences

This menu controls entity display features for the graphics window which include the following:

- Display Triad in Graphics Window
- Enable Pre-Selection
- Background Color
- Perspective Angle

- Line Width
- Highlight Line Width
- Text Size
- Ambient Intensity
- Ambient Color
- Light Intensity
- Light Color

General Preferences

This menu controls general program options including the following:

- **Prompt for Unsaved Application Data** - When this is checked and the user opens a new .cub file or exits the application with unsaved changes, a dialog box will pop up asking if they want to save changes first. The user can uncheck this option to prevent that dialog box from appearing. This is checked by default.
- **Prompt for Unsaved Journal Data** - When this button is checked and the user closes the journal file editor with unsaved changes the program will prompt to save the changes. The user can uncheck this button to prevent the dialog box from appearing. It is checked by default.
- **Change to Script Directory for Playback** - When this option is checked, Claro will change the working directory to the directory the script is in when the script/journal file is run. When the script is finished, Claro will change the directory back to the previous one. This is useful when using relative paths in a journal file. When the option is unchecked, Claro won't change the directory when a journal file is run in which case the user may have to manually change the working directory when their journal file has relative paths.
- **Prompt When Translating from Python** - When checked, if the user translates a python script to a cubit journal file, the journal editor will warn them that commands may be lost. When unchecked, the journal editor will not issue the warning. There is a checkbox on the warning dialog that sets this option as well.
- **Default Syntax** - Sets the default syntax to use when creating a new journal file in the editor. The Cubit option is only available when the cubit component is loaded.
- **Show Startup Splash Screen** - Option to hide the startup splash screen on opening Claro.

Geometry Defaults

This menu controls the geometry defaults.

- Vertex Size
- Use Silhouette on Geometry
- Silhouette pattern

The user can also change the default geometry engine to one of the following:

- ACIS
- Facets

The faceting tolerance can also be controlled from this menu to change the way facets are drawn in the graphics window.

History Preferences

This menu controls the input window history and journal file options. These include:

- **Maximum Number of Commands** - The max number of commands kept in the current command history.
- **Maximum Number of Lines** - Maximum number of lines in input window.
- **Journal Command History** - Whether to use a journal file to save command history. Default is to use a journal file.
- **Output Log** - When this option is checked, you can save error log to a separate output file.

**Label Defaults**

This menu controls the geometry and mesh entity labels in the graphics window.

- Text Size
- Label Geometry and Mesh Entities Toggles- Choose label visibility for each type of geometry or mesh entity

Layout Preferences

This menu option controls input window formatting and control panel docking options.

- **Font for command line workspace**
- **Font size for command line workspace**
- **Reset Window Layout Button** - Used to reset GUI windows to their default positions

Also included in the layout preferences is a list of available windows with a checkbox to show/hide each window.

Cubit Layout Settings

This menu controls the layout of Cubit specific buttons and tabs on the GUI.

- **Show script tab** - Shows the script tab on the command line window
- **Use Labels on Buttons**- Option to apply a label to each button on the control panel
- Preferred Location (currently under construction)

Mesh Defaults

- **Node Size**

- **Element Shrink**
- **Mesh Line Color** - The same as "Color Lines" command.
- **Default Element Type** - Tet/Tri or Hex/Quad
- **Surface Scheme Coloring** (used in Meshing Power Tool) - This option allows you to select different colors for surface schemes when visualized using the meshing power tools.

Mouse Settings

This menu controls mouse button controls. Pressing the **Emulate Command Line Settings** button will cause all of the settings to simulate mouse controls in the command line version of CUBIT. For a detailed description of mouse settings see the **View Navigation-GUI** page.

**Post Processor Settings**

**Post Processor Executable Directory** - Option to browse for post processor executable directory.

**Quality Defaults**

This menu controls quality defaults for different quality metrics. For a description of the different quality metrics see the respective pages:

- Hexahedral metrics
- Quadrilateral metrics
- Tetrahedral metrics
- Triangular metrics

 *Undo Button*

Cubit has an undo capability. To enable the Undo feature click on the "Enable Undo" button on the Toolbar.

 **Enable Undo Button**

Alternatively to turn undo on and off, the following command may be used in the command line:

**undo {on|off}**

The Undo capability is implemented for geometry commands including webcutting, geometry creation, transformations, and booleans. Multiple undos are also allowed. The commands will be undone in reverse order of their execution.

**Limitations**

- The undo button is not currently enabled for most meshing commands

control panel

# Command Panels

The Command Panels provide a graphical means of accessing almost all of the CUBIT functionality. The main CUBIT Command Panel is divided into seven modes. Each of these modes pertains to a major component of the CUBIT application. To view information about each of the tools in the Control Panel select the help icon on each panel to access context specific help.

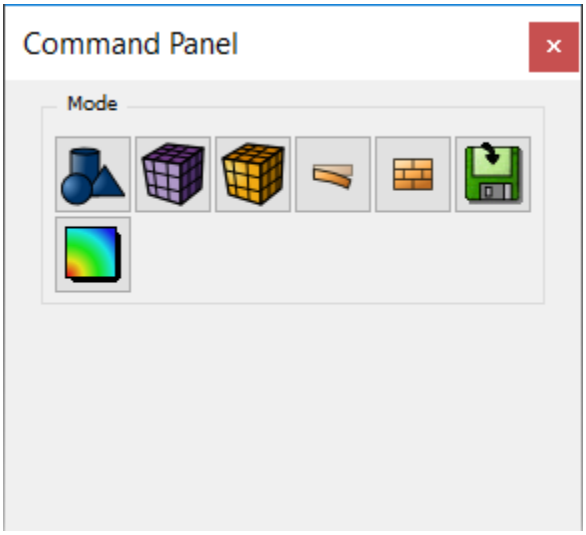
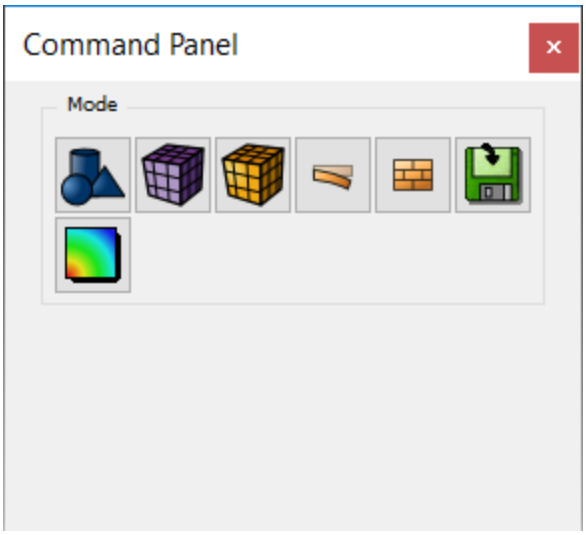



**Figure 1. The CUBIT Control Panel**

A brief description of the functionality of the Control Panel window follows.
Control Panel Functionality

# Command Panel Functionality

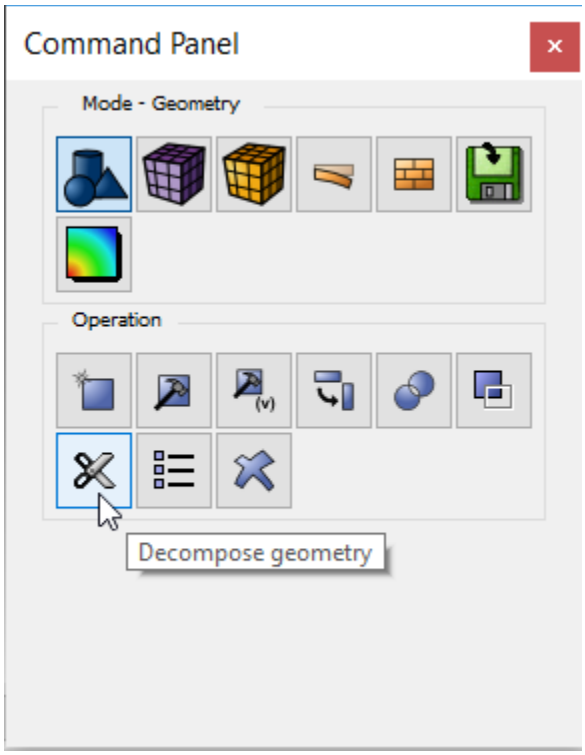
The Command Panel is arranged first by mode on the top row of buttons. Modes are arranged by task.



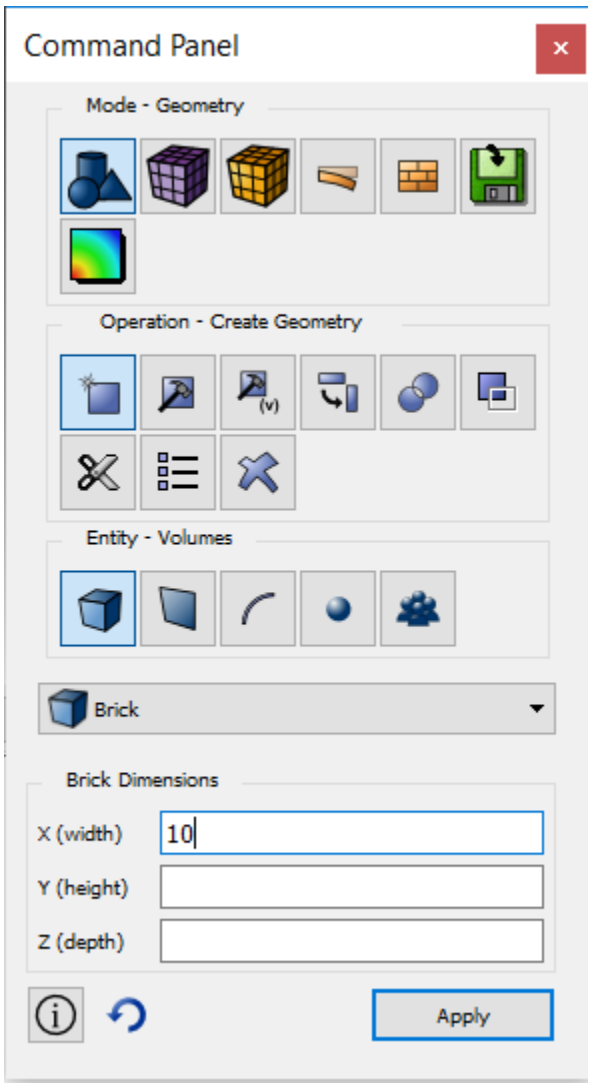

The modes are:

- Geometry
- Mesh
- Analysis Groups and Materials
- FEA Boundary Conditions
- CFD Boundary Conditions
- Analysis Setup (Export)
- Post-meshing Tool Launch

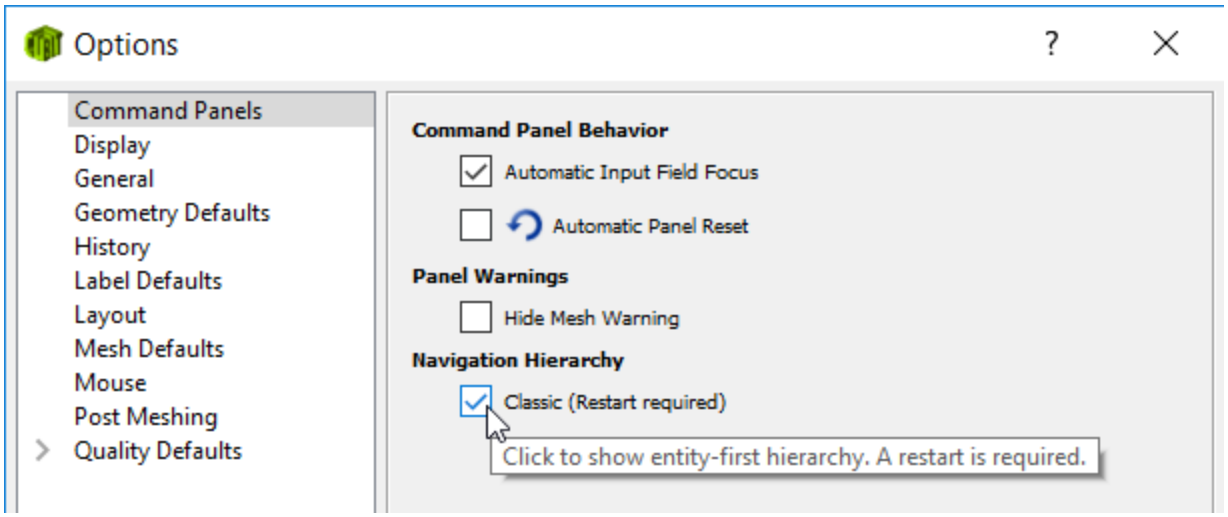
All of the geometry related tasks, for instance, can be found under the Geometry mode. When a mode is selected, a second row of buttons becomes available. The second row of buttons shown depends on the selected mode. For example, if Geometry is selected, the second row of buttons will contain operations that can be performed on geometry, such as creation, modification, decomposition, boolean, merging, deleting, and so forth.



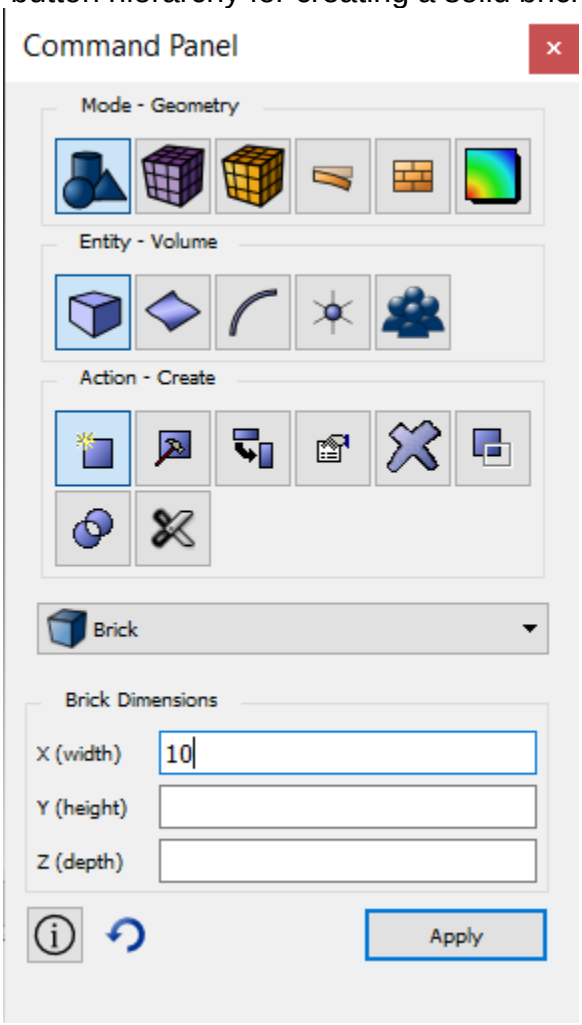

Selecting an operation will cause a row of geometry entity buttons to be displayed. Specific, entity-based operations will be shown after selecting the entity type, as shown below.



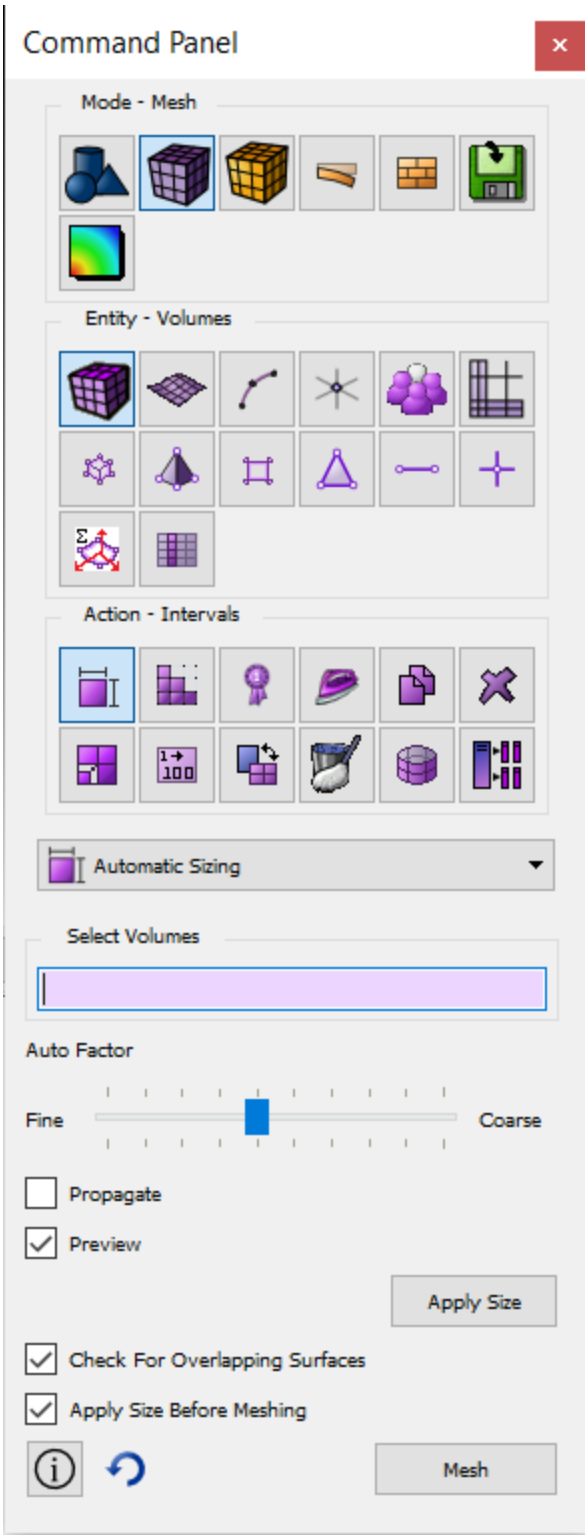

**Note: This hierarchy is different than previous versions of Cubit**. In previous versions the second row of buttons was geometry entity types, such as volume, surface, curve, vertex, and group. If a user wishes to use the 'classic' hierarchy for geometry, an option can be selected in Tools/Options/Command Panels.




The 'classic' hierarchy's button hierarchy for creating a solid brick is the following:




For all other modes, such as mesh, analysis groups, and so forth, the 'classic' button hierarchy remains.




All command panels are constructed similarly. Each abstracts a set of Cubit commands. Options are selected using check boxes, radio buttons, combo boxes, edit fields, and other standard GUI widgets. Each command panel includes an Apply button. Pressing the Apply button will generate a command to Cubit. Nothing happens until and unless the Apply button is pressed.

**Note: The edit fields are free form, which means the user may enter any valid string into the fields. Any string that is valid for the command line is valid for the command panel edit fields.**

Where possible, default values are placed into edit fields. At any time, with the cursor placed over a blank portion of the command panel, the user may right-click to select Reset Data which will clear all fields and replace default values.

#### ID Input Entry Methods

The *ID Input Fields* provide a location where Geometric IDs, required for the current command, can be entered. IDs can be entered in several ways:

**Simple Keyboard entry**

ID numbers can be entered directly in the field. Each ID must be separated with a space. Select the field first before typing.

**Graphical selection**

IDs can be entered automatically by selecting entities directly in the Graphics Window. The current entity available for selection is based on the current entity selection mode. In some cases, not all entities of the current entity selection mode will be available for picking. The program may automatically filter the applicable entities based on the context of the current command

**Geometry Tree selection**

IDs may be entered by selecting the corresponding geometric entity from the geometry tree. To select multiple entities use the **<ctrl>** key.

**Ranges**

A range of IDs may be typed into the field. For example:

**1 to 5**

will automatically enter all IDs from 1 to 5 inclusive in the field. Keywords such as **all** and **except** can also be used. Any range that can be entered directly on a CUBIT command line can also be used in the ID input field. See Command Line Entity Specification for a description of the syntax.

**As Part of Other Entities**

Syntax can be entered in the *ID Input field* that will specify an entity based upon its topological relationship to other entities For example, if a **Vertex** Selection Type Button was highlighted, entering

**in surf 1**

will automatically enter all vertices in surface 1 into the *Input Field*. CUBIT has a rich set of syntax rules for specifying entities based upon topology relationships. See Command Line Entity Specification for a description.

**In Groups**

Entities that are part of groups may be specified in the ID Input Field. For example, if the Vertex Selection Type Button is highlighted, entering:

**in picked**

will automatically enter all vertices in the picked group into the active *ID Input Field*.

**Dragged and Dropped**

Entities can be dragged and dropped into the *ID Input Field* from the Tree View window.

#### Right-Click Context Menu for ID Input Fields

When the right mouse button is selected while in an *ID Input Field*, the following menu options will appear:

- **Done Selecting** - Enters current selection and removes cursor from selection window
- **Select Other** - Displays selection dialog
- **Select All** - Selects all available entities and puts "select all" in input window
- **Highlight -** Highlight the current selection
- **Zoom To** - Zooms to current entity in the selection field within the graphics window
- **Rotate About** - Change center of rotation to the center of selected entity
- **Draw** - Draws the entities listed in the input field within the graphics window
- **Isolate** - Turns visibility off for all entities other than the selected entities. Similar to draw command, but entities remain hidden with a graphics refresh. Select **All Visible** in the graphics window to turn visibility back on.
- **Visibility Off** - Removes the current entity from the input window and hides it on the graphics screen
- **Mesh** - Mesh the listed entities using either an assigned scheme or a default scheme where none is assigned
- **Delete Mesh** - Deletes mesh on all entities listed in the input window
- **Reset Entity** - rehighlights the entities listed in the input field within the graphics window
- **List Info** - Displays a sub menu of choices including basic, geometry, and mesh. Selecting the basic option will list schemes, visibility, and interval assignments. The geometry option will add information about the geometry and geometry engine. The mesh option will list information about mesh entities.
- **Delete** - Deletes the current geometric object in the input window.

#### Value Fields

Integer and real values pertinent to the command are entered in this window. Input placed in parenthesis { } will be evaluated when the command is executed. For example:

```
{10*0.02}
```

is valid input. Additionally, any APREPRO syntax is valid in the *Value Field*, including mathematical functions and boolean operations. See the section, APREPRO for a description of syntax.

#### Advancing Pickwidgets

Some command panels have several id input fields such as the Mesh>Hex>Create panel. A convenience feature implemented for such panels is an advancing pickwidget feature. Pressing the middle mouse button after selecting an entity will advance to the next id input field.

## CUBIT Application Window

The default CUBIT Application Window is shown in the following image.

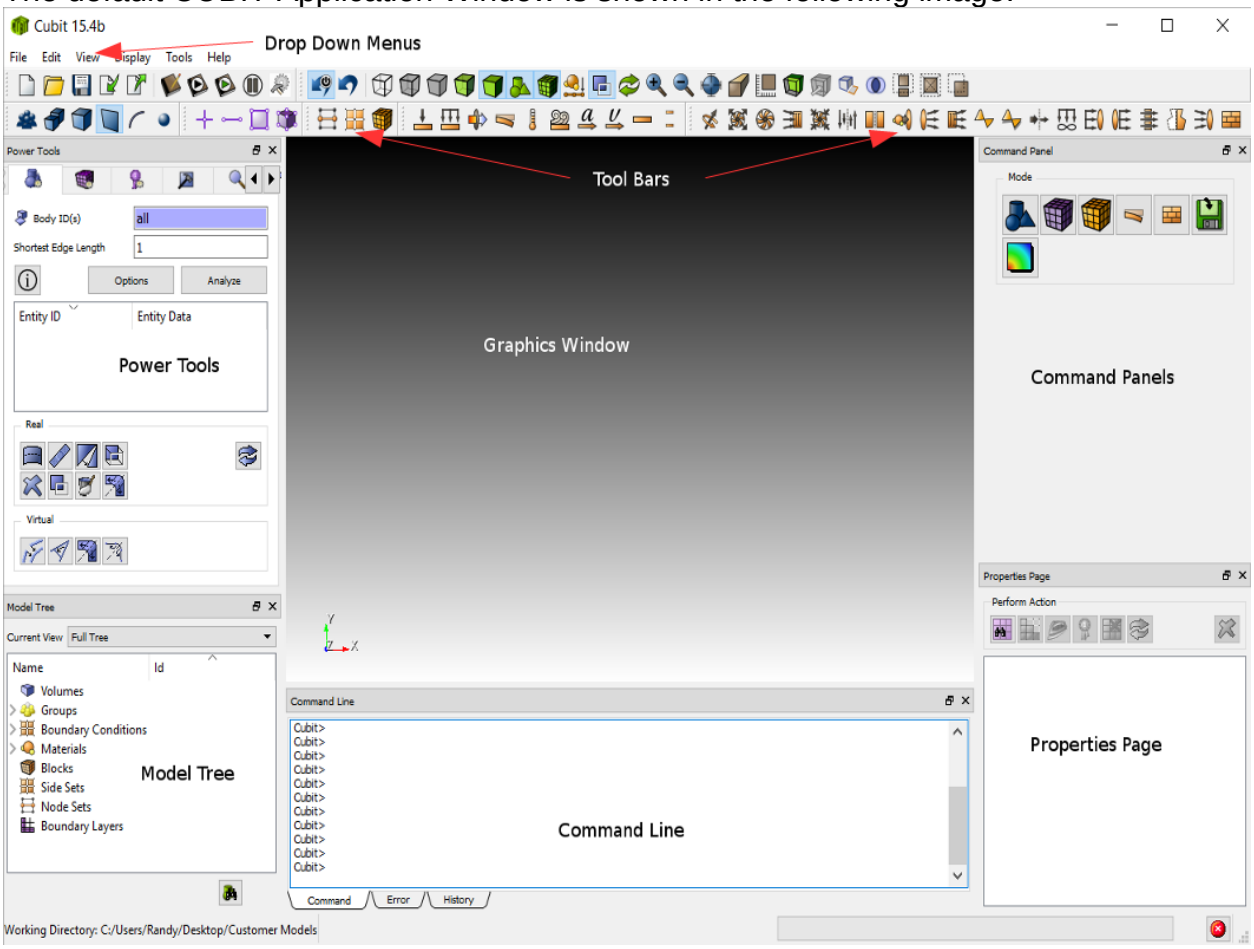



**Figure 1. The CUBIT Application Window**

**Graphics Window**- The current model will be displayed here. Graphical picking and view transformations are done here.
**Power Tools** - Geometry tree hierarchy view, geometry analysis and repair tool, meshing tool, meshing quality tool, and ITEM Wizard.
**Property Editor** - The Property Editor lists attributes of the current entity selection. Some of these properties can be edited from the window.
**Command Panel** - Most Cubit commands are available through the command panels. The panels are arranged topologically, by mode.
**Command Line Workspace** - The command line workspace contains both the cubit command and error windows. The command window is used to enter cubit commands and view the output. The error window is used to view cubit errors.
**Drop Down Menus** - Standard file operations, Cubit setup and defaults, display modes, and other functionality is available in the pull-down menus.
**Toolbars** - The most commonly used features are available by clicking toolbar icons.

*Context Sensitive Help in the GUI*

The Graphical User Interface has a context-sensitive help system. To obtain help using a specific window or control panel, press F1 when the focus is in the desired window. It may be necessary to click inside a text box to switch focus to a particular window. If no context specific help is available, it will open the cubit help documentation where you can search for a particular topic.

*Customizing the Application Window*

All windows in the CUBIT Application can be *Floated* or *Docked*. In the default configuration, all windows are docked. When a window is *docked* the user can click on the area indicated below.

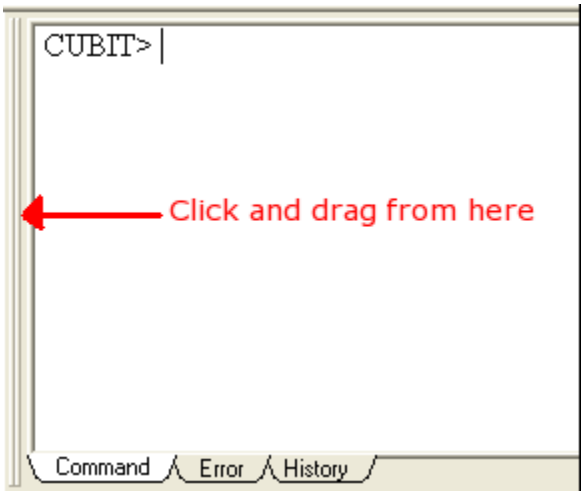



**Figure 2. A docked window. Click and drag to float.**

By dragging with the left mouse button held down, the window will be un-docked from the Application Window. Dragging the window to another location on the Application Window and releasing the mouse button will cause it to dock again in a new location. The bounding box of the window will automatically change to fit the dimensions of the window as it is dragged. Releasing the mouse button while the window is not near an edge will cause the window to Float. To stop the window from automatically docking, hold the CONTROL key down while dragging.

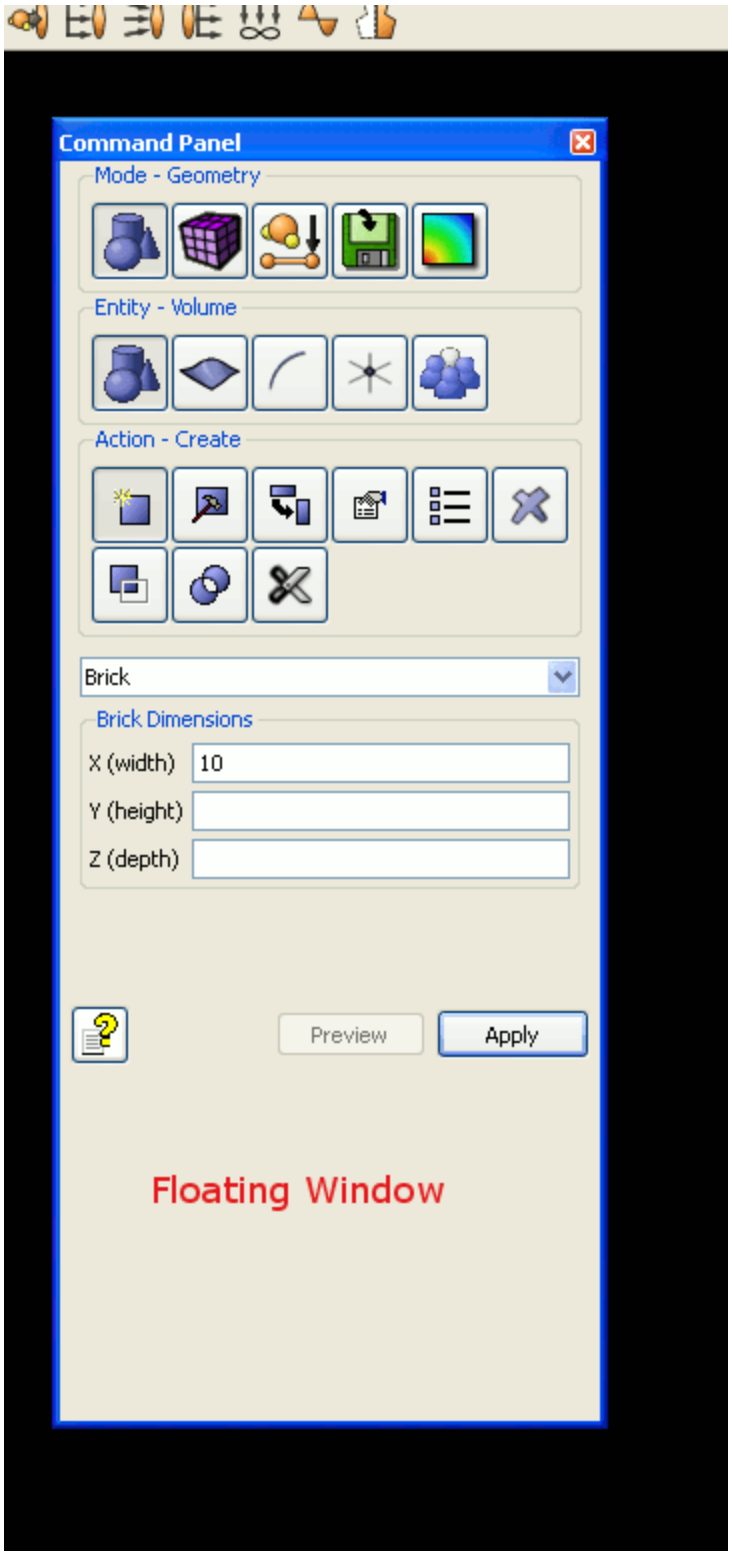
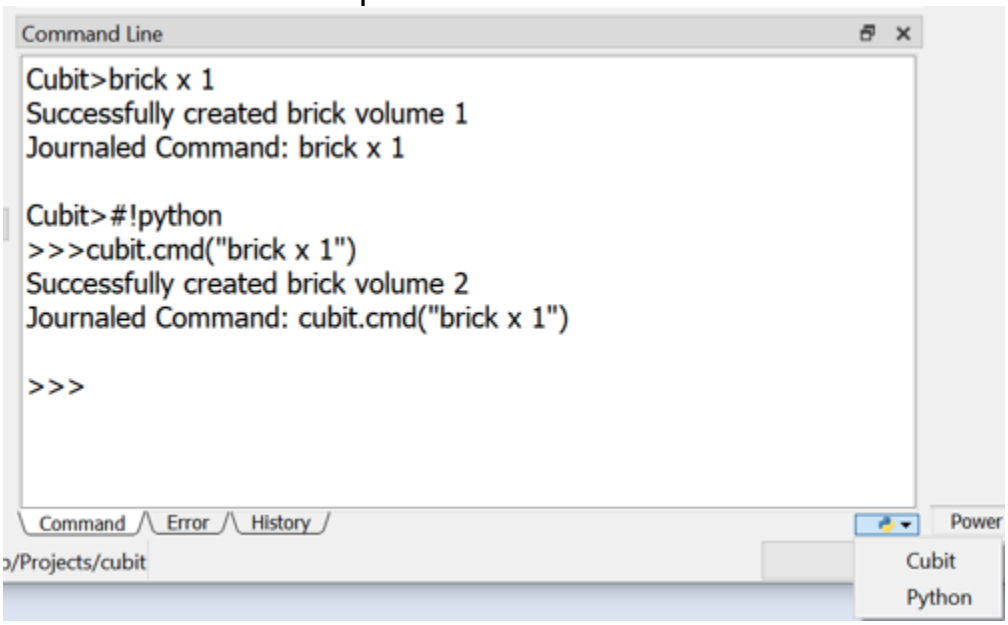



**Figure 3. A Floating Window**

When a window is *floating*, as shown in Figure 3, it is possible to dock it by clicking the title bar of the window and dragging it to its new docked location.
Note: Double clicking on the title bar of an floating window will cause the window to *redock* in its last docked position.

## Command Line Workspace




The Command Line Workspace is the interface for command interaction between the user and the CUBIT application. The user can enter commands into this window as if they were using the command line version of CUBIT. Journaled commands will be echoed to this screen, even if they were not typed in manually. Thus, if the user wants to know what the command sequence for a particular action on the GUI is, they can watch for the "Journaled Command:" line to appear. In addition, this screen will contain important informational and error messages. The command window has the following three tabs:

1. Command
2. Error
3. History

*Command Window*

The command line workspace emulates the environment in the command line version of Cubit. Commands can be entered directly by typing at the **CUBIT>** or **>>>** prompt. This window also prints out error messages, informational messages, and journaled commands.

#### Entering Commands

To enter commands in the command line workspace, the command window must be active. Activate the command window by clicking anywhere inside the window. Commands are typed in at the **CUBIT>** or **>>>** prompt. Cubit commands are accepted with the **CUBIT>** prompt and Python statements are accepted at the **>>>** prompt. If you

do not remember the specific Cubit command sequence you can type **help** and the name of the command phrase. The input window will show all of the commands that contain that word or phrase. Alternatively, if you know how a command starts, but do not remember all of the options, you can type **?** at the end of the command to show all possible command completions. See Command Syntax for an explanation of command syntax rules.

#### Command Mode

The command window can be in either a Cubit command mode or a Python statement mode. Switching modes can be done with **#!cubit** to switch to the Cubit command mode or with a **#!python** to switch to the Python statement mode. A convenience button is also available in the lower right corner of the command window to switch modes with a click of the mouse button.

#### Python Statements

The Python interpreter in Cubit works as though you were entering lines at the Python command prompt. This means a blank line is interpreted as the end of a block. If you want to add whitespace for clarity, you could add a # mark for a comment on any white line that is in a loop or a class.
A second python script can be included and executed by either using the **play** Cubit command or by using a Python **import** statement.
The interface between cubit and Python is the 'cubit' object. Among many methods, this object has a method called **cmd** which takes as an argument a Cubit command string. Thus the following command can be issued in the Command window at the **>>>** prompt, which will create a cube with sides 10 units long.

```
cubit.cmd("create brick x 10")
```

The following script is a simple example that illustrates using loops, strings, and integers in Python.

```
%>for i in range(4):
. . x=i*3
. . for j in range(4):
. .    y=j*3
. .    for k in range(4):
. .      z=k*3
. .      mystr="create vertex x "+str(x)+" y "+str(y)+" z "+str(z)
. .      cubit.cmd(mystr)
```

This simple script will create a grid of vertices four wide. Scripts can be more advanced, even creating customized windows and toolbars. For a complete list of python/cubit interface commands see the Appendix.

#### Repeating Commands

Use the **Up** and **Down** arrow keys on the keyboard to recall previously executed commands.
Commands can be repeated in other ways as well.

- Hitting the enter key while the cursor is on a previous command line will copy that command to the current prompt.
- The command window supports copy and paste for repeating commands.

#### Focus Follows Cursor

Beginning with version 13.0, Cubit includes a 'focus follows cursor' option for the command window. The option can be enabled and disabled from the Tools/Options/General options panel. The setting is persistent between sessions and is disabled by default.
Please note, the **focus follows cursor** behavior is available only in the command window. All other windows or widgets require the user to *click* the mouse in order to grab focus.

*Error Window*

The error window is located in the Command Line Workspace under the Error tab. If there are errors, a warning icon will appear on the tab. The icon will disappear when you open the window to view errors. The error window only displays the error output, which can make it easier to find and read the error output. The command that caused the error will be printed along with the error information. If the command was from a journal file, the file name and number will be printed next to the command.

*History Window*

The history window lists the last 100 commands. The number of commands listed can be configured in the options dialog on the History page. You can re-run the commands in the history window using the context menu. You can also clear the history using the context menu.

*Docking and Undocking the Input Window*

The command window can be undocked by clicking and dragging the left edge. If it is floating it can be redocked by double-clicking the solid blue bar. By default, it will always be redocked in the bottom of the application window. To change the size of the floating window, click and drag the edge of the window. To change the height of the docked window, click and drag the top edge or right edge.

## Journal File Editor

The Journal File Editor is a built-in, multi-document text editor that can read, edit, play, and translate CUBIT journal files and Python Scripts. To open the journal file editor, select the  icon on the File Tools toolbar, or from the Tools Menu.

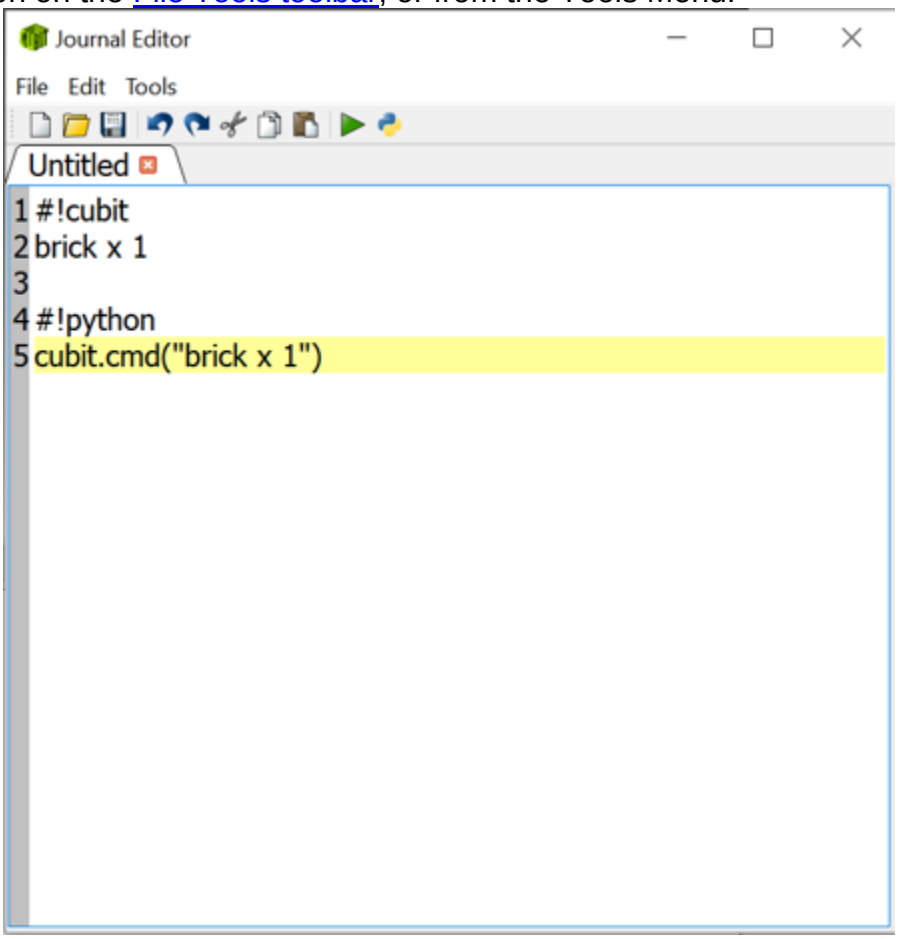



**Figure 1. The Journal File Editor**

The Journal File Editor can be used to create a new Python or Cubit command script. By default, a new journal file will be in Cubit command syntax. Enter the commands in the order you want them executed. You can play the commands all at once using the play button on the toolbar. You can also play a few commands at a time. Select the commands you want to play. Then, right click and select the "Play Selected" menu item. If you have a Cubit script, you can convert it to a Python script by hitting the Python toolbar button. If you have a few lines you want to convert to Python, select them, and right click to get the popup menu and choose "Translate Selected to Python"

The Journal File Editor can also be used to edit an existing journal file. Use the File > Open menu item to open the file you want to edit. You still have all the command play options with an existing journal file.

You can import commands entered in the Command Line Workspace. The File > Import menu gives the option to import commands from the history tab. Only the current commands shown in the history tab will be imported. Some of the commands you

previously entered might not show up if you have the recommended text trimming turned on. Text trimming improves the application's performance for speed and memory. It will trim off the oldest text in the window when a size limit is reached. To get all the command from your current session, make sure that command journaling is turned on.
The Journal File editor can be used to edit multiple files at the same time. Each document is displayed in its own tab. The tab shows the journal file's syntax and name. If you close the Journal File Editor with unsaved data, it will prompt you to save changes for each of the modified journal files you have open.

*Journal Editor Toolbar*

The Journal Editor's Toolbar provides quick access to several important functions.



- **New** - Creates a new journal file. The new journal file is placed in a new tab.
- **Open** - Used to select a journal file to open.
- **Save** - Saves the current journal file.
- **Undo** - Undo the last text change.
- **Redo** - Redo the last text change, after Undo.
- **Cut** - Standard text cut operation
- **Copy** - Standard text copy operation
- **Paste** - Standard text paste operation
- **Play Journal File** - Plays the entire journal file
- **Translate to Python** - Translates the current Cubit commands in the journal file to Python scripts.

*Other Functionality Available in the Journal Editor*

The context ('right-click') menu in the journal editor includes several additional functions, including:

- **Comment Selected Lines** - Highlight any text, select 'comment selected lines', and the highlighted lines will be commented.
- **Uncomment Selected Lines** - Highlight any text, select 'uncomment selected lines', and the highlighted lines will be uncommented.
- **Clear** - select this menu item to clear the contents of the journal file.
- **Find** - Selecting 'find' from the context menu, or from the edit menu, will bring up a dialog enabling the user to find text in the journal file. Options are available to do case-sensitive searches, change search direction, and so forth.

## Property Editor

The Property Editor is a window that lists properties about the current entity selection. Some of the properties, like CUBIT ID, entity type, or geometry engine, are listed for reference only. Other attributes, like name, or mesh intervals, color, mesh scheme, or smooth scheme can be edited from the window. The Property Editor is located on the left panel in the GUI. The highlighted entity/entities in the graphics window are listed in the

property editor window. The Property Editor also lists information about selected mesh entities, boundary conditions, and assemblies. Selecting an object from the Tree View will also open the object in the property editor.

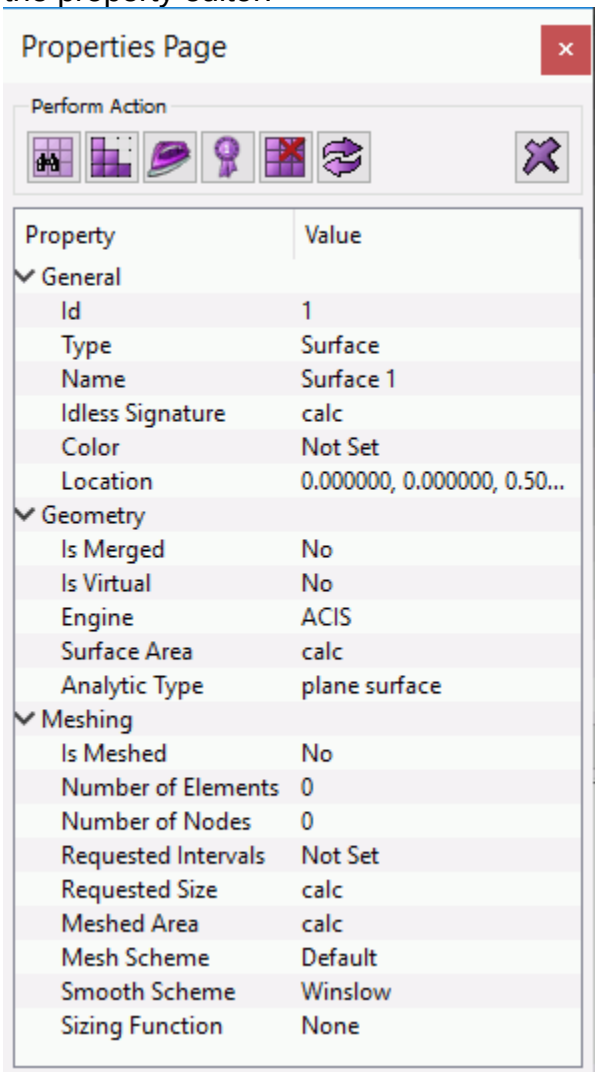



**Figure 1. Property Editor Window**

The row of buttons on the top of the editor are shortcuts to common commands. These include:

>

 Meshes the selected entity/entities at their current interval and scheme settings

 Smooth selected entity using the current smoothing scheme

 Preview mesh intervals on selected entity

 Delete mesh on specified entity (do not propagate to lower order entities)

 Reset entity to default settings and delete mesh

 Calculates volumes and surface areas

 Delete current entity

*Editing Entity Attributes from the Property Editor*

The Property Editor provides a convenient way to change attributes on entities. . Some of the fields cannot be changed, some can be edited from an input field, and others are edited by selecting from a list, or by opening the corresponding window from the Control Panel.
If multiple entities are selected, the attributes that are similar to both entities will be shown. Changing an attribute from the property editor will change that attribute on both entities. If multiple entities are selected the total volume, surface area, and length of all entities will be shown.
Below is a summary of properties listed for each attribute type.

#### General Attributes

- **Entity ID** - CUBIT ID for geometry or boundary condition element
- **Entity Type** - Geometric type such as Volume, Surface, Curve, Vertex
- **Name** - Name by which the entity can be referred to from within CUBIT instead of using its ID. The entity name can be edited from this window.
- **Color** - Opens a dialog box with available colors. A color name can also be input directly into the text field. See Appendix for a list of available colors.

#### Geometry Attributes

- **Is Merged** - Returns "Yes" if this entity is merged
- **Is Virtual** - Returns "Yes" if this entity is a virtual entity
- **Location** - Returns the location of specified vertex.
- **Geometry Engine** - ACIS or Mesh-Based Geometry
- **Volume** - The volume of the specified body
- **Surface Area** - Surface area of selected surface
- **Analytic Type** - Returns the analytic type of entity (such as cone, sphere, etc)
- **Length** - Length of selected curve

#### Meshing Attributes

- **Is Meshed** - Returns "Yes" if the entity is already meshed
- **Number of Elements** - Similar to "List Totals" command
- **Requested Intervals** - Number of requested mesh intervals on element. This can be edited from this window. The number must be an integer
- **Requested Size** - Requestd interval size for element. Clicking on box will open the interval specification panel on the control panel. The interval size can also be entered manually in the text box.

- **Meshed Volume** - The meshed volume may be slightly different than the actual element volume due to the mesh approximation on curved surfaces.
- **Meshed Area** - The meshed area may be slightly different than the actual surface area due to mesh approximation on curved edges.
- Length of Meshed Edges - Combined total of mesh edge lengths on curve
- **Mesh Scheme** - The mesh scheme for this entity. This can be changed from the property editor by selecting from the drop-down list.
- **Smooth Scheme** - The smooth scheme for this entity. This can be changed from the property editor by selecting from the drop-down list.

**Boundary Condition Attributes**

- **ID** - Boundary condition ID. This is an arbitrary user-defined ID that is exported with the finite element model. This value can be edited from the property editor
- **Name** - A user-defined name that is included in the metadata for that object. This value can be edited from the property editor.
- **Description** - A user-defined description that is included in the metadata for that object. This value can be edited from the property editor.
- **Color** - Opens a dialog box with available colors. A color name can also be input directly into the text field. See Appendix for a list of available colors.
- **Element Type** - The finite element type for this block, nodeset, or sideset.
- **Element Count** - The total number of elements for this block or sideset
- **Node Count** - Total number of nodes (available for nodesets only)
- **Attribute Count and Attributes**- The attributes represent material specification data that is associated with the element block. These values can be changed in the property editor. You can specify up to 10 attributes per block.

## Toolbars

The CUBIT toolbars provide an effective way for accessing frequently used commands. Below is a brief description of each of the available toolbars. To view a description of the function of each tool, hold the mouse over the tool in the CUBIT Application to display tool tips.
Users may customize and share toolbars.

*File*

Provides CUBIT (*.cub) file operations. This toolbar also includes Journal File operations.



**Figure 1. File Toolbar**

From left to right, the tool buttons are as follows:

- **New** - Issues a Reset command to begin a new session
- **Open** - Displays file open dialog

- **Save** - Displays file save dialog
- **Import** - Displays import dialog
- **Export** - Displays export dialog
- **Journal Editor** - Opens the journal editor
- **Play Journal File** - Plays a journal file
- **Play ID-Less Journal File** - Plays a journal file in Id-less mode
- **Pause** - Pause the playback of a journal file
- **Custom Toolbar Editor** - Opens the custom toolbar editor

*Display*

Controls the display mode, checkpoint undo, zoom, perspective clipping plane, and curve valence display options in the Graphics Window.



**Figure 2. Display Toolbar**

From left to right, the tool buttons are as follows:

- **Undo On/Off** - Toggle undo functionality on/off
- **Undo** - Undo last operation
- **Wireframe** - View in wireframe mode
- **True Hidden Line** - View in true hidden line mode
- **Hidden Line** - View in hidden line mode
- **Transparent** - View in transparent mode
- **Shaded** - View in shaded mode
- **View Geometry** - Toggle geometry display on/off
- **View Mesh** - Toggle mesh display on/off
- **View BC** - Toggle boundary condition display on/off
- **Graphics Composite** - Toggle graphics composite line display on/off
- **Refresh** - Refresh the graphics display
- **Zoom In**
- **Zoom Out**
- **Zoom to Fit**
- **Toggle Perspective**
- **Scale** - Toggle display of the scale
- **Clipping Plane** - Toggle display of the clipping tool
- **Clipping Manipulation** - When clipping plane is active, toggle clipping tools
- **Curve Valence** - Display curve valence in the graphics window
- **Overlapping Surface** - Locate (point to) overlapping surfaces in the model
- **Enclosed/Extended** - Toggle between enclosed and extended selection mode
- **X-Ray** - Toggle x-ray selection mode on/off
- **Selection Shape** - Toggle between box, sphere, polygon selection box

*Select*

Controls the Entity Selection Mode for picking or selecting entities.

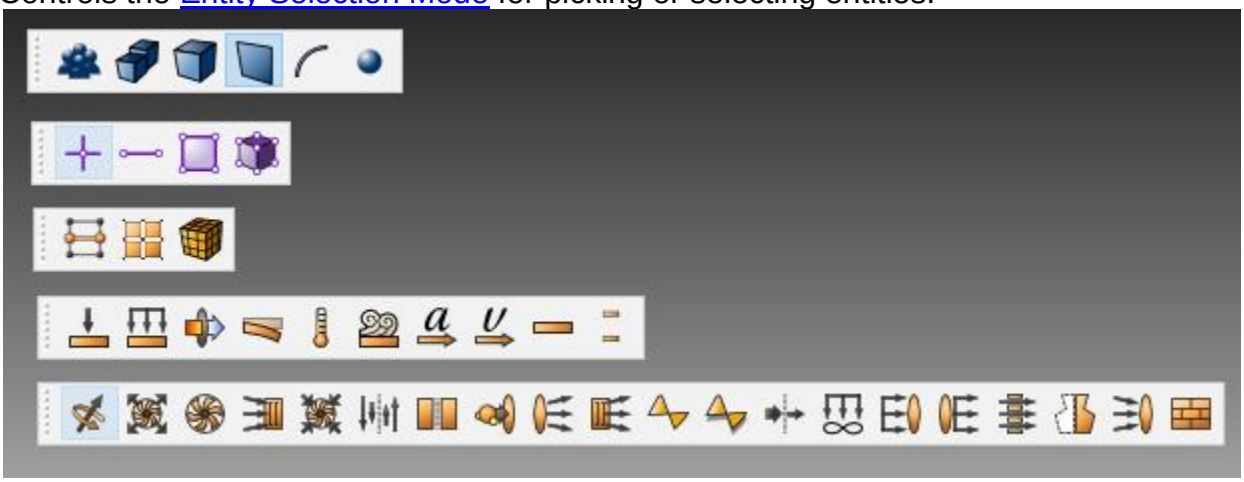


**Figure 3. Select Toolbars**

From top to bottom:

- Geometry Selection Filters
  - Groups
  - Bodies
  - Volumes
  - Surfaces
  - Curves
  - Vertices
- Mesh Selection Filters
  - Node
  - Edge
  - 2d (Triangle, Quadrilateral)
  - 3d (Hexahedral, Tetrahedral, Pyramid, Wedge)
- Boundary Condition and Material Containers
  - Nodeset
  - Sideset
  - Block
- FEA Boundary Conditions
  - Force
  - Pressure
  - Heatflux
  - Displacement
  - Temperature
  - Convection
  - Acceleration
  - Velocity
  - Contact Regions
  - Contact Pairs
- CFD Boundary Conditions

- Axis
- Exhaust Fan
- Fan
- Inlet Vent
- Intake Fan
- Interface
- Interiors
- Inlet Mass Flow
- Outflows
- Outlet Vents
- Periodics
- Periodic Shadows
- Porous Jump
- Farfield Pressure
- Inlet Pressure
- Outlet Pressure
- Radiator
- Symmetries
- Inlet Velocities
- Walls

## Toolbar Customization

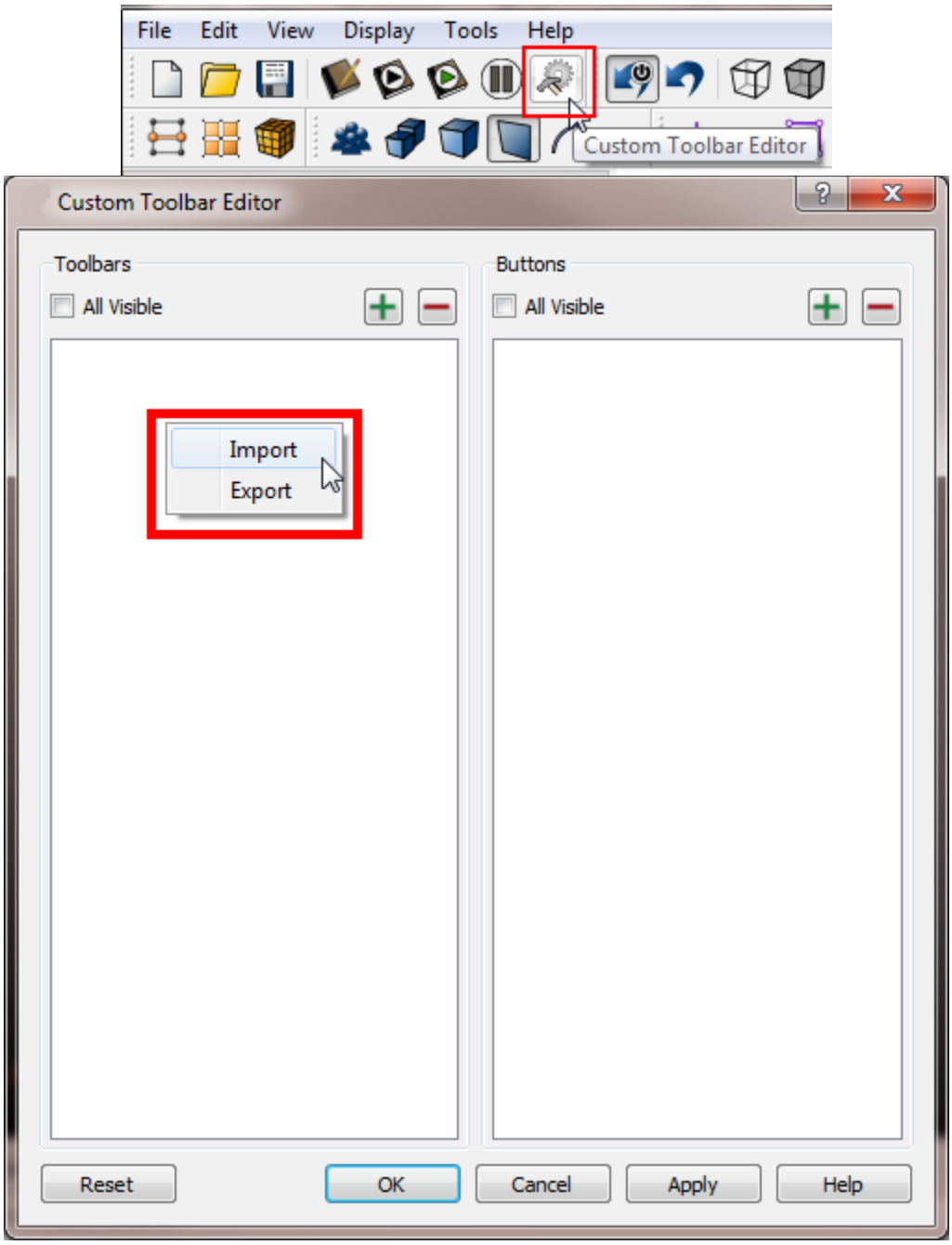
For many years Cubit has provided users with the ability to create custom tool buttons. These custom buttons launch pre-defined journal or Python scripts. With the release of Cubit 15.4 this capability has been expanded.

*Menu*

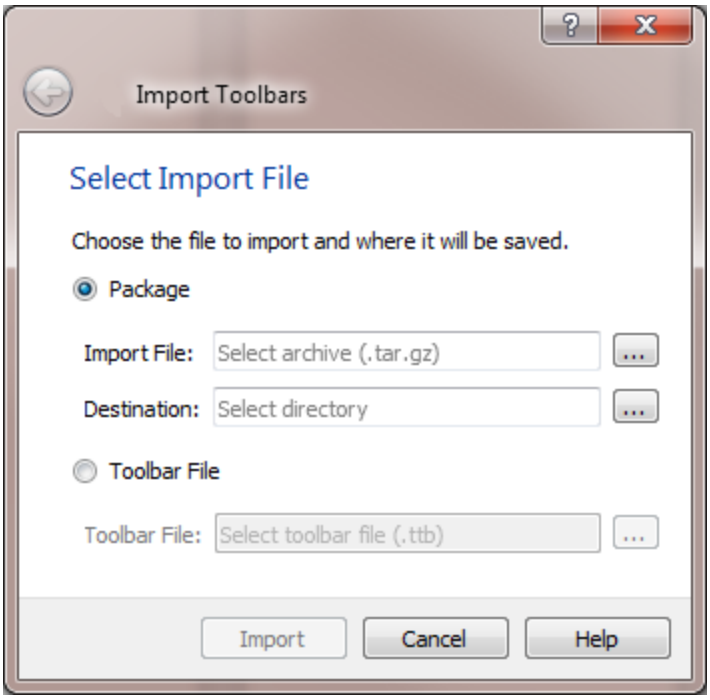
- Importing an Existing Toolbar
- Exporting a Toolbar
- Creating a new Toolbar
- Creating a Command Panel Button
- Creating a Journal File Button
- Creating a Python Script Button
- Creating a Basic Tool Button
- Modifying an Existing Toolbar

*Importing an Existing Toolbar*

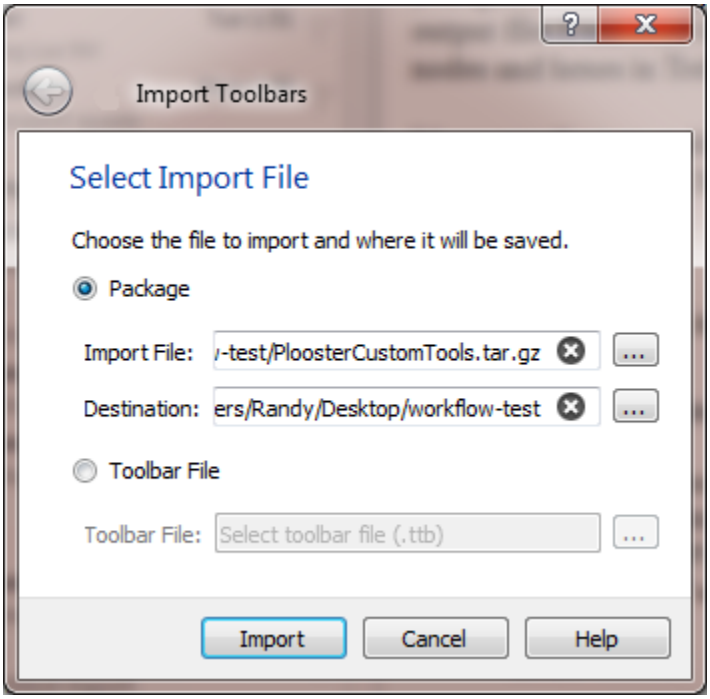
- Locate and press the Custom Toolbar Editor button located on the File Tools button bar. This will launch the Custom Toolbar Editor.



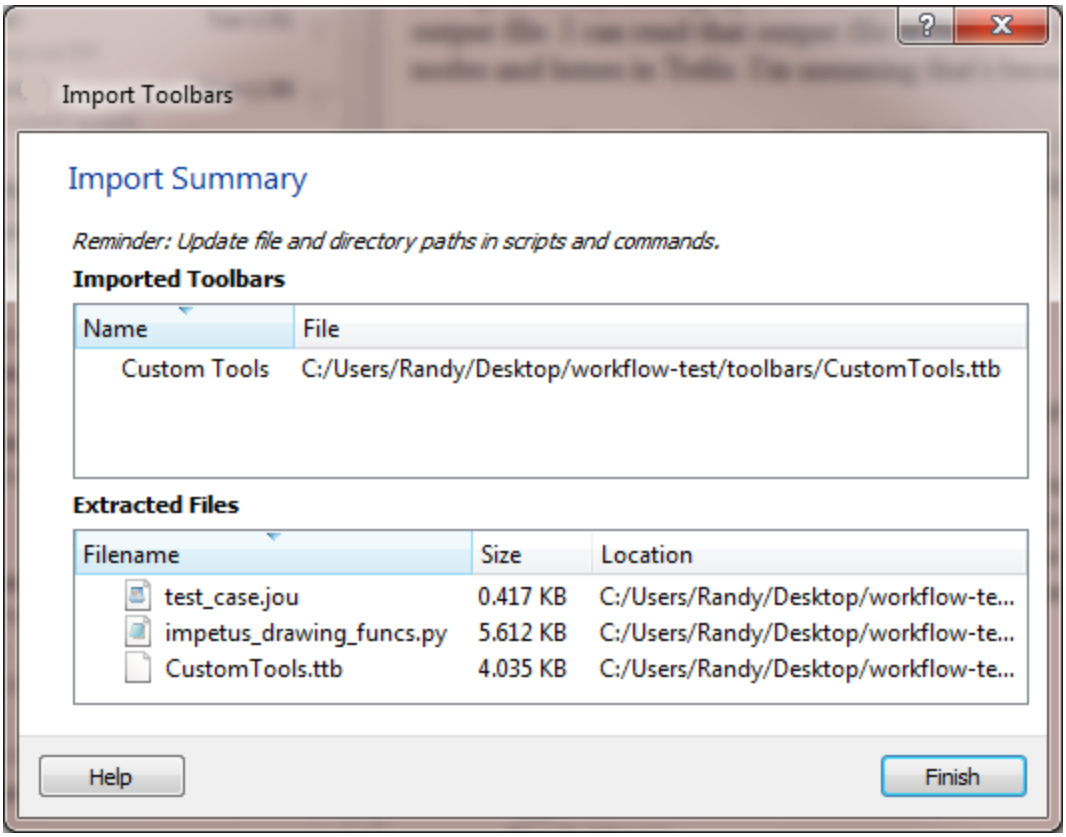

- Select Import from the context menu



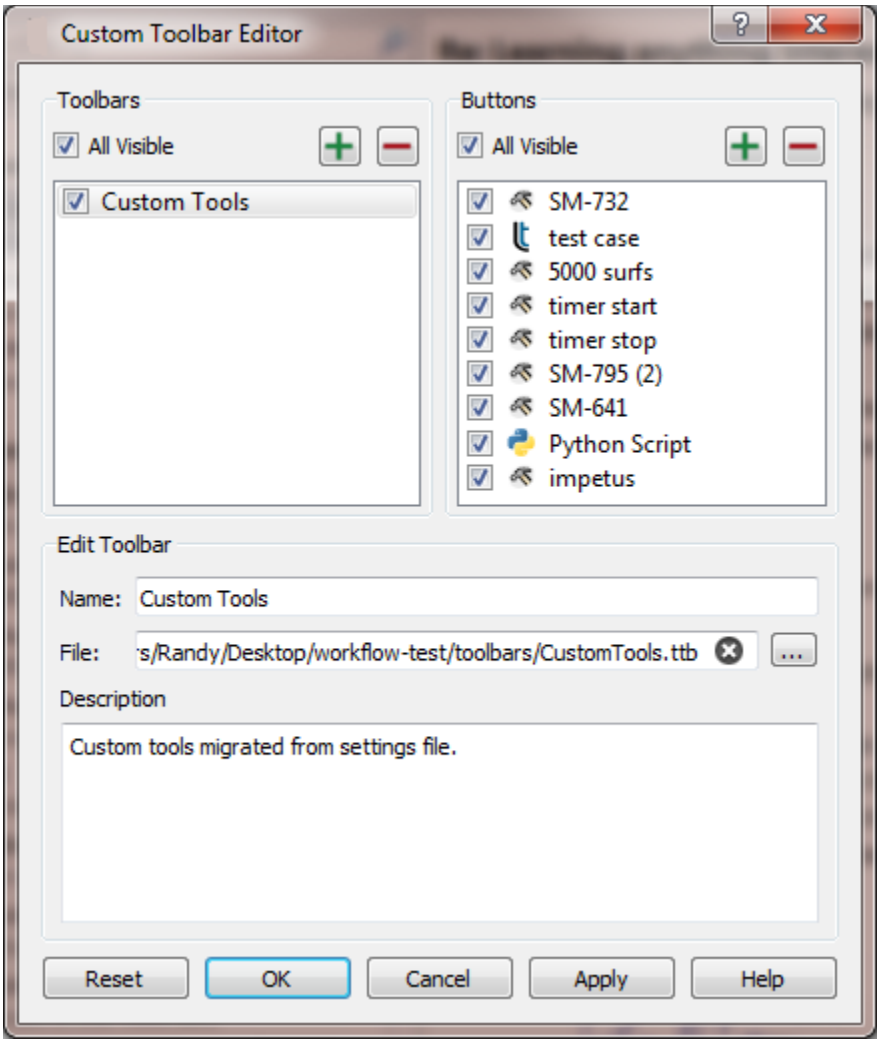

From this dialog a user may import an **entire package** containing multiple toolbars or a **single** toolbar. In this example we will import an entire package containing multiple toolbars.




- After selecting **import**, an import summary is shown.




- Select **Finish**

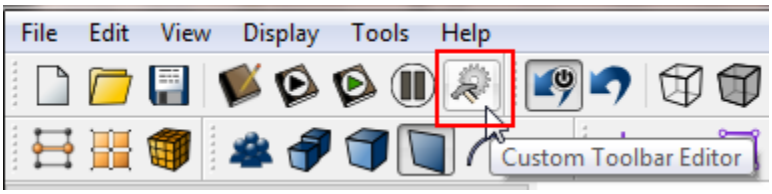



- Select **OK** to finish the import

The new toolbar and buttons will be displayed as the last toolbar on the GUI. It is a docking window so it can be moved and placed anywhere on the GUI.

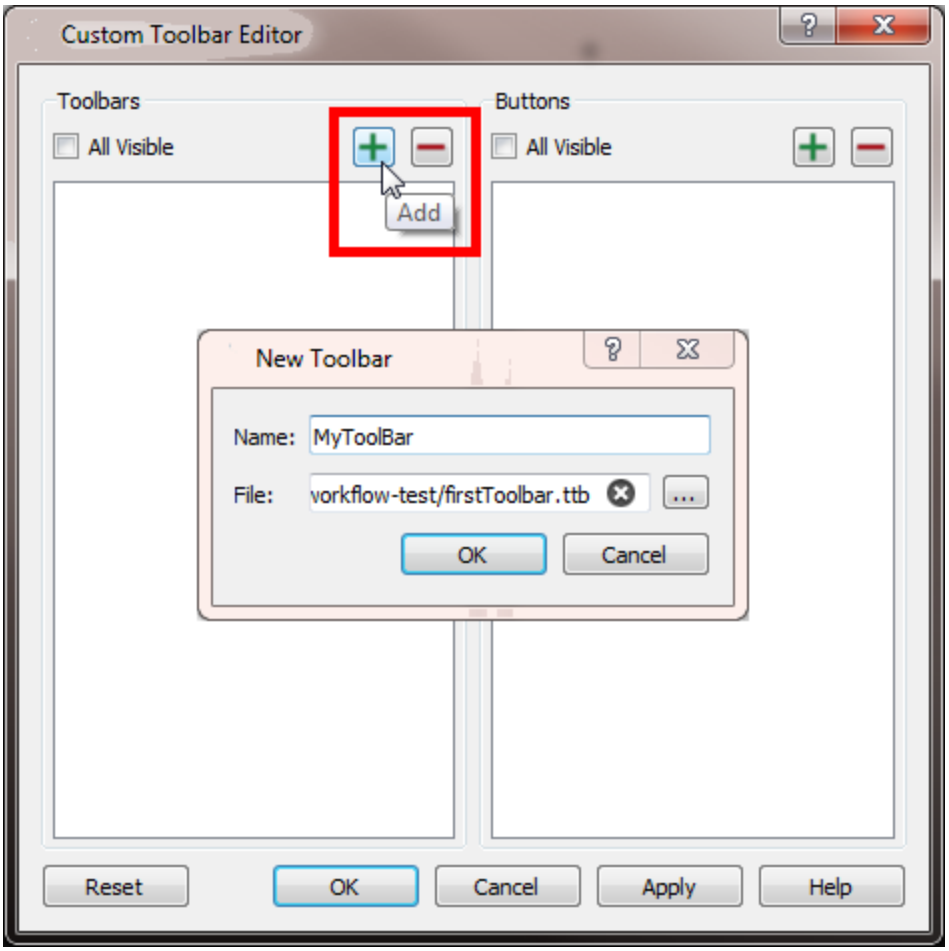
*Creating a New Toolbar*

- Locate and press the Custom Toolbar Editor button on the File Tools button bar. This will launch the Custom Toolbar Editor.

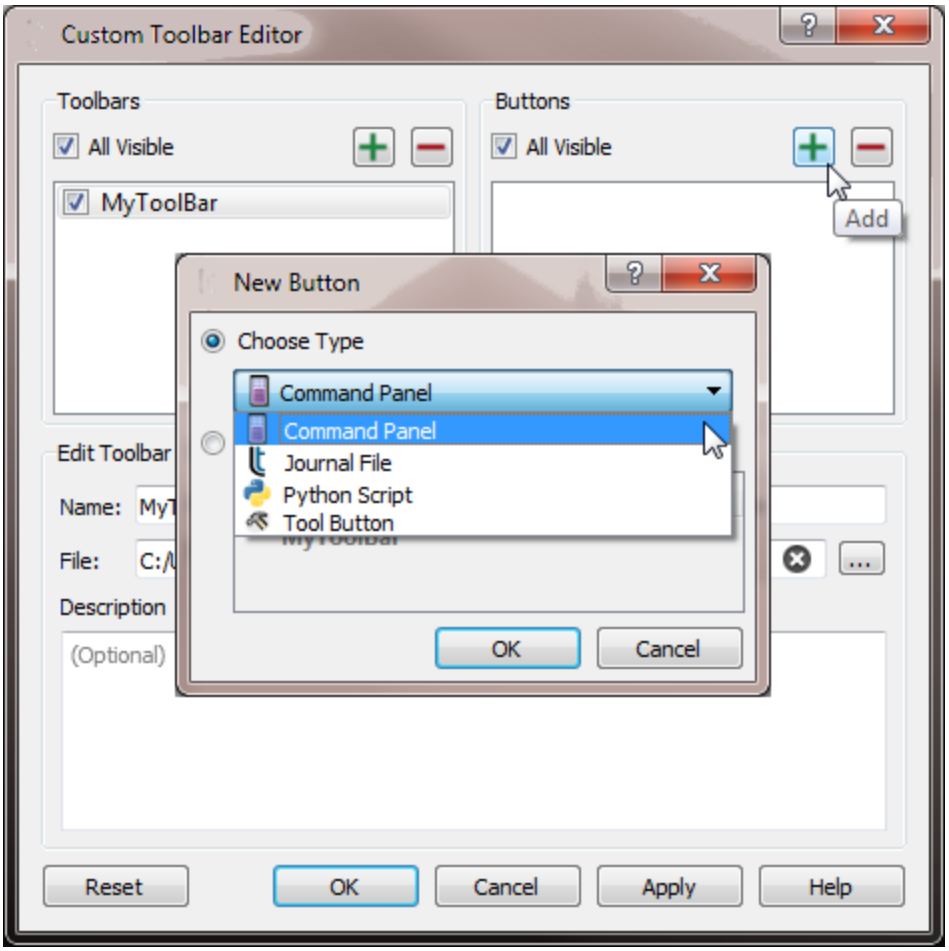



- Press the **Add** button

- Name the new toolbar and press **OK**




- Press the **Add** button in the Buttons area




A user may define 4 different types of toolbar buttons

- Command Panel
- Journal File
- Python Script
- Tool Button

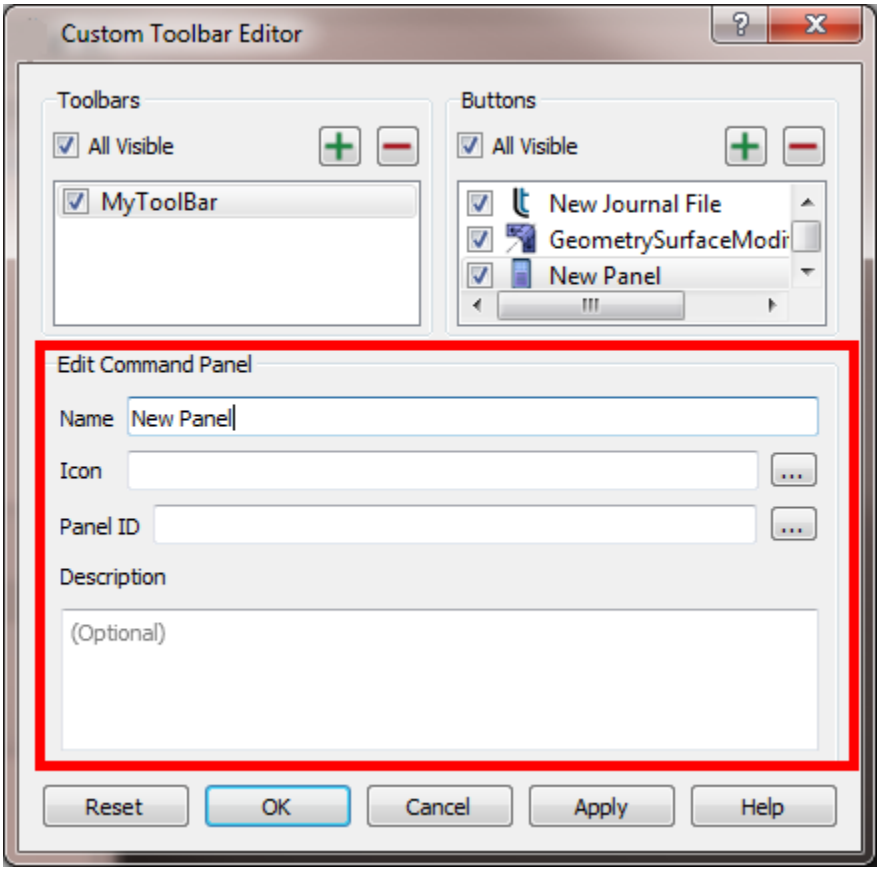
*Creating a Command Panel Button*

A **Command Panel Button** enables users to launch a command panel with the push of a button. A command panel button can be defined one of three ways:

**Use the definition dialog**

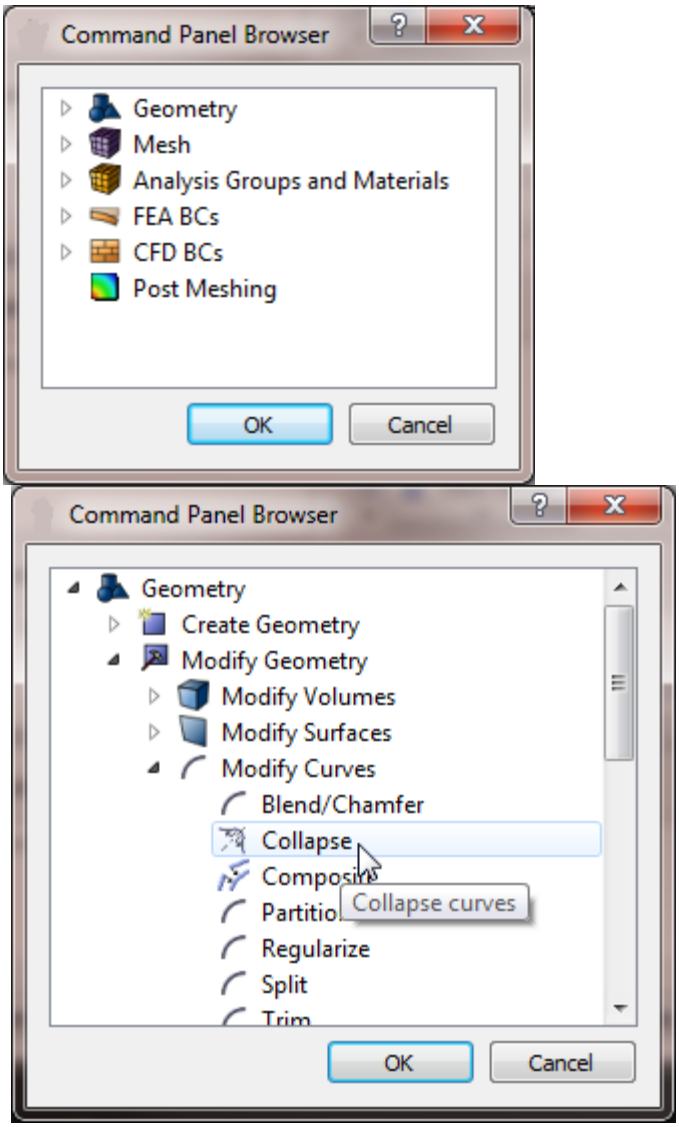
- Select **Command Panel** from the New Button type pulldown menu
- Press **OK**
- Complete the dialog indicating
  - the name of the button
  - the icon to use
  - the panel ID of the command panel to show -- **see instructions for find the panel ID (below)**

    - an optional description of the command panel
- Press **OK** to save the definition and exit the dialog
- Or, press **Apply** to save the definition



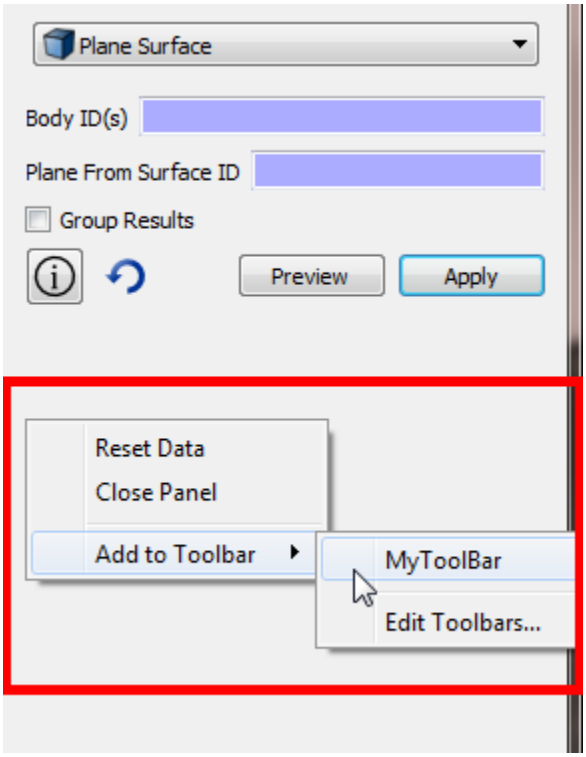

To find the Command Panel ID:

- Press the browse button next to the Panel ID edit field to launch the **Command Panel Browser**
- Navigate the browser to locate the desired command panel
- Select the desired command panel
- Press **OK** to make the selection
- The Panel ID will be shown in the Panel ID edit field



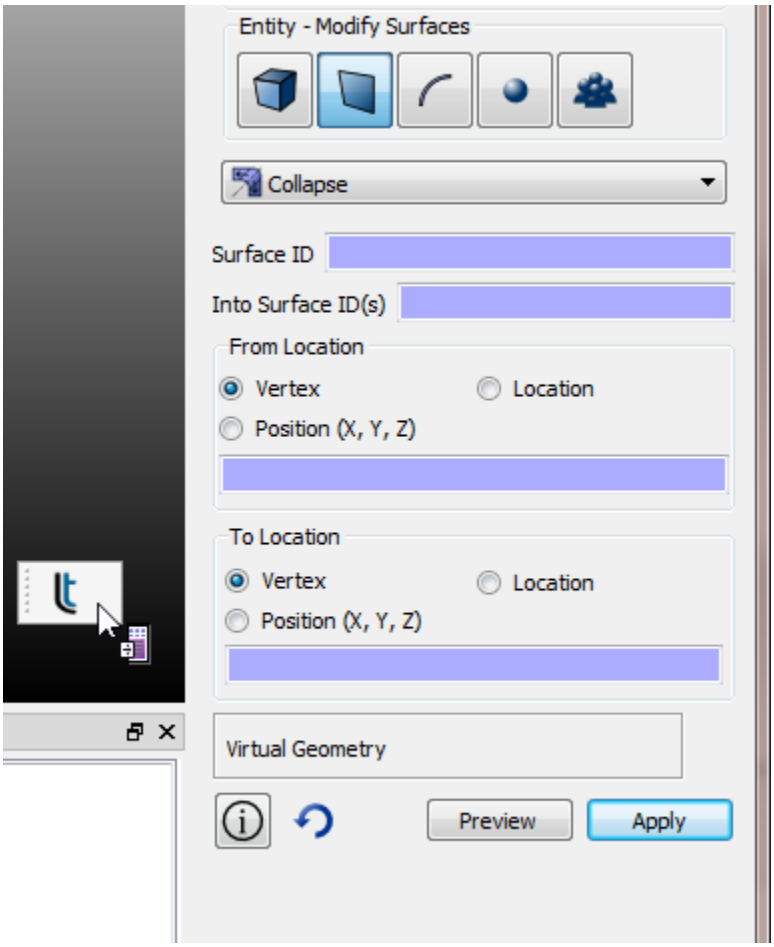

**Use the context menu on a command panel**

- Show the context menu on a command panel
- Select **Add to Toolbar**
- Select the toolbar to which this command panel will be added
- an icon representing the command panel will be added to the selected toolbar




**Drag a command panel onto the toolbar**

- Using the mouse, "drag" the command panel onto the desired toolbar
- an icon representing the command panel will be added to the selected toolbar
- In the image below, the Surface Collapse command panel is being dragged onto a toolbar




- The resulting toolbar looks like the following



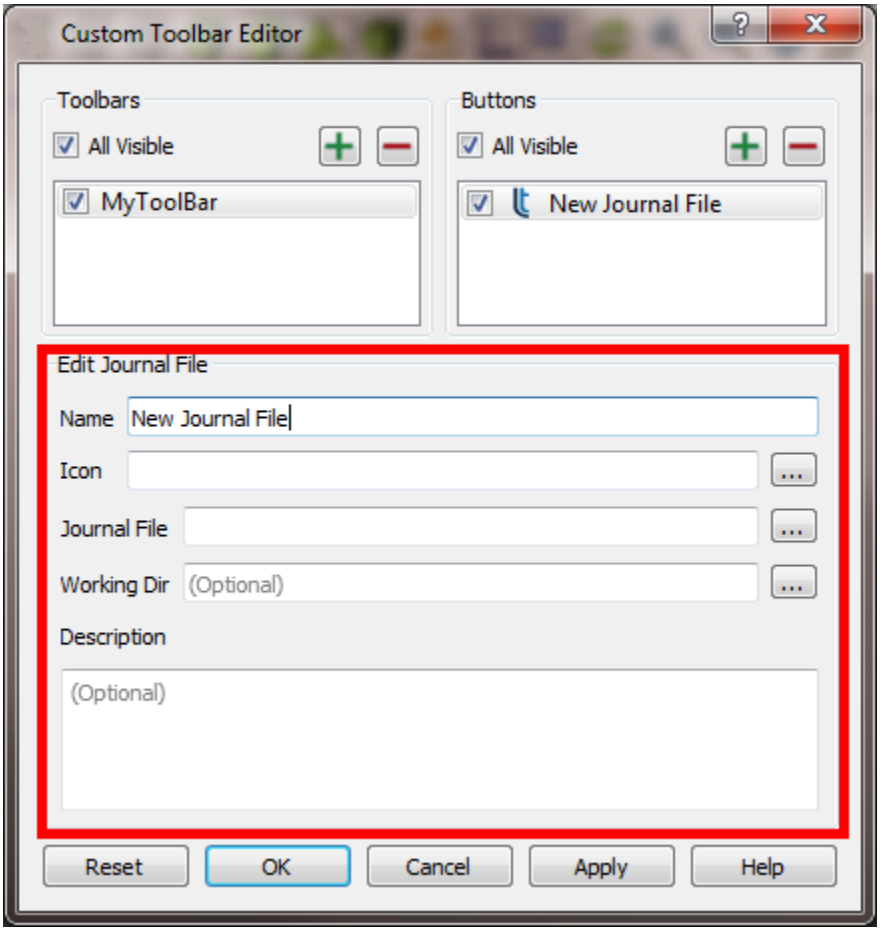
All command panels include a context menu which can be accessed by clicking on an empty place in the command panel and using the mouse to show the menu.

*Creating a Journal File Button*

A Journal File Button will launch a journal file when pressed. The journal file may reside anywhere on the file system. A journal file button is defined by:

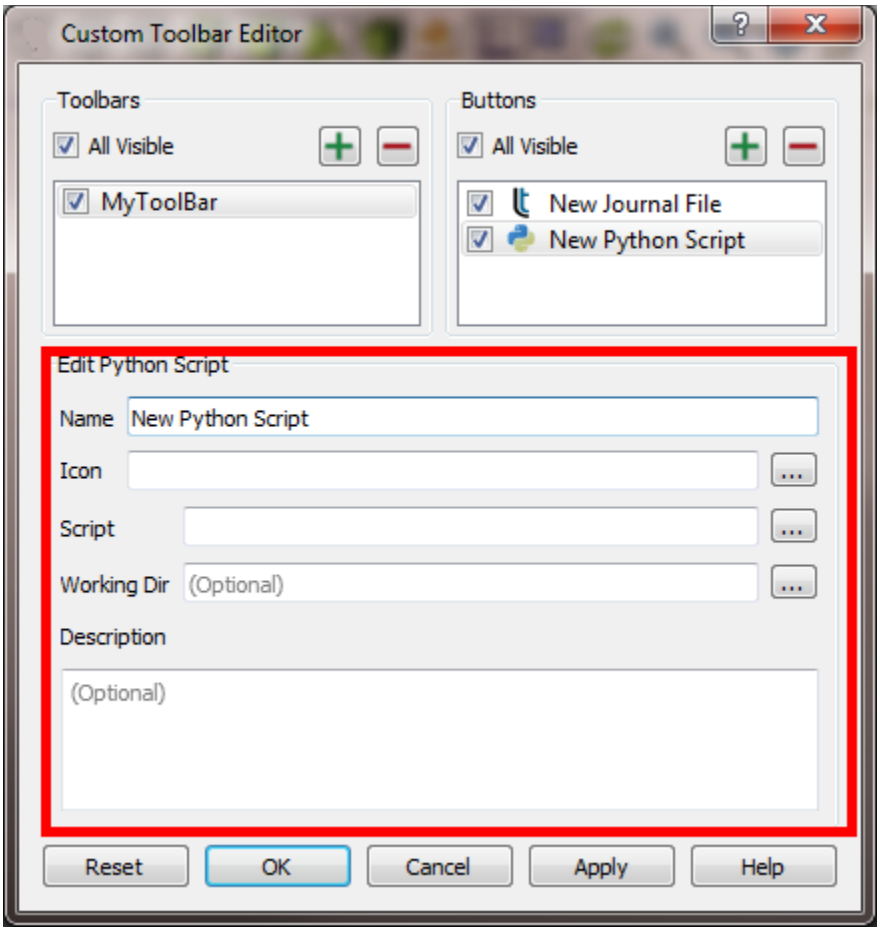
- Select **Journal File** from the New Button type pulldown menu
- Press **OK**
- Complete the dialog indicating
  - the name of the button
  - the icon to use
  - the name of the journal file to play
  - an optional working directory

    - an optional description of the journal file
- Press **OK** to save the definition and exit the dialog
- Or, press **Apply** to save the definition




*Creating a Python Script Button*

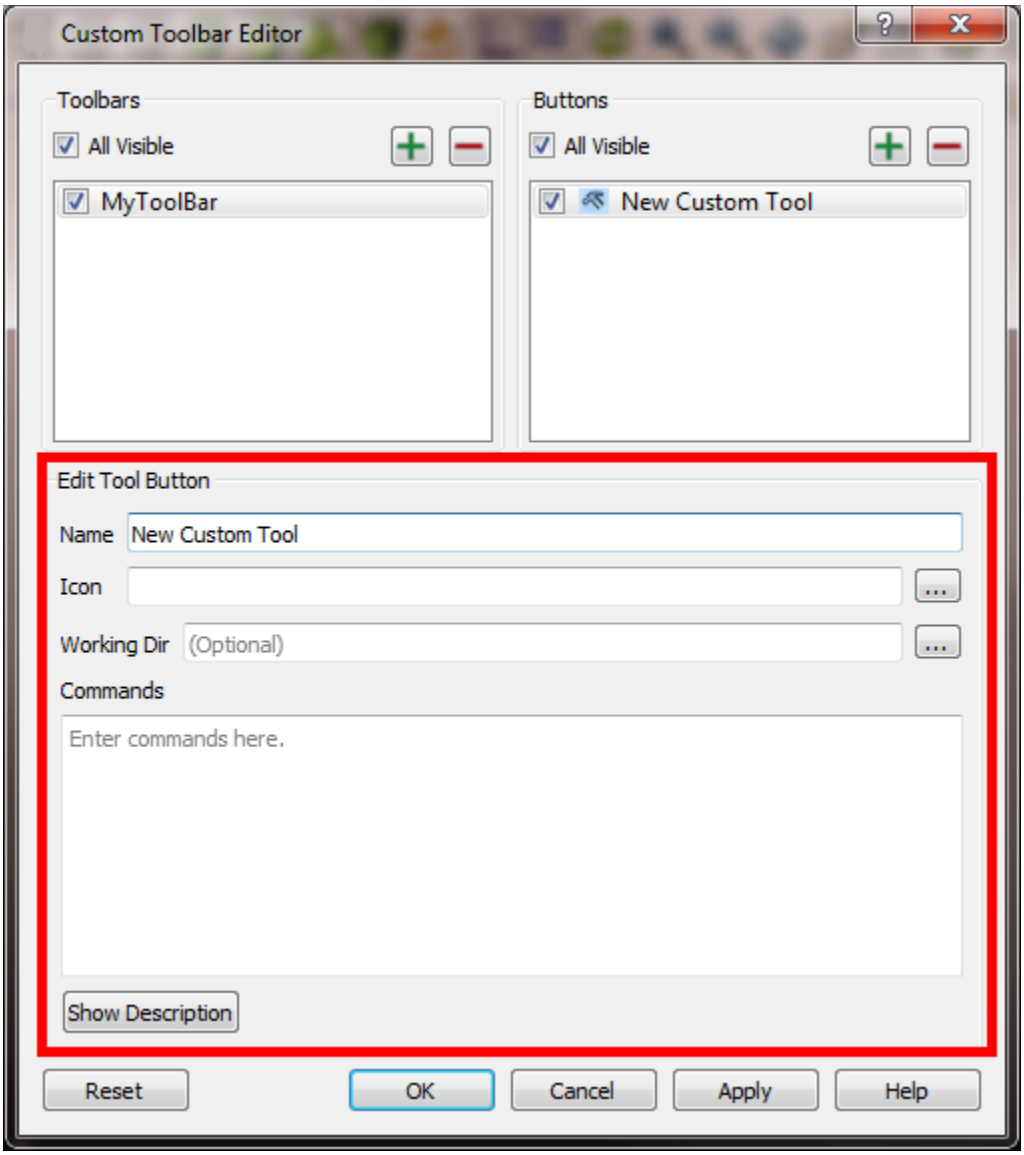
- Select **Python Script** from the New Button type pulldown menu
- Press **OK**
- Complete the dialog indicating
    - the name of the button
    - the icon to use
    - the name of the Python script to execute
    - an optional working directory
    - an optional description of the Python script
- Press **OK** to save the definition and exit the dialog
- Or, press **Apply** to save the definition




*Creating a Basic Tool Button*

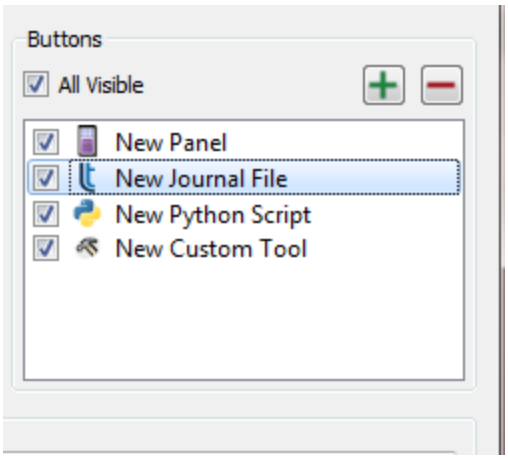
The "Basic" too button has been available to users for many years. It contains a set of commands that execute when the user presses the button.

- Select **Tool Button** from the New Button type pulldown menu
- Press **OK**
- Complete the dialog indicating
  - the name of the button
  - the icon to use
  - an optional working directory
  - the commands to execute (these are the same commands used in any journal file)
  - an optional description of the commands
- Press **OK** to save the definition and exit the dialog
- Or, press **Apply** to save the definition




*Modifying an Existing Toolbar*

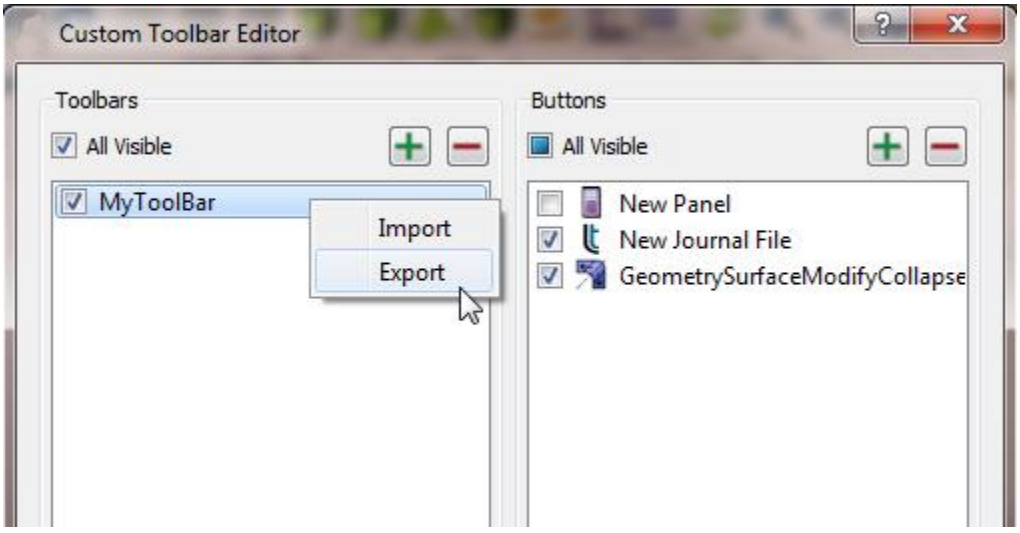
- In the Custom Toolbar Editor select the toolbar to modify
  - Press the Add (green plus-sign) button to add a new button
  - Press the Delete (red minus-sign) to remove a button
  - Select the check box to hide or show the button
  - **Change the button order** by selecting a button in the Buttons dialog and dragging to a new position
  - Any other parameter may be modified using the Edit Tool Button dialog
  - Press **OK** to save the definition and exit the dialog
  - Or, press **Apply** to save the definition



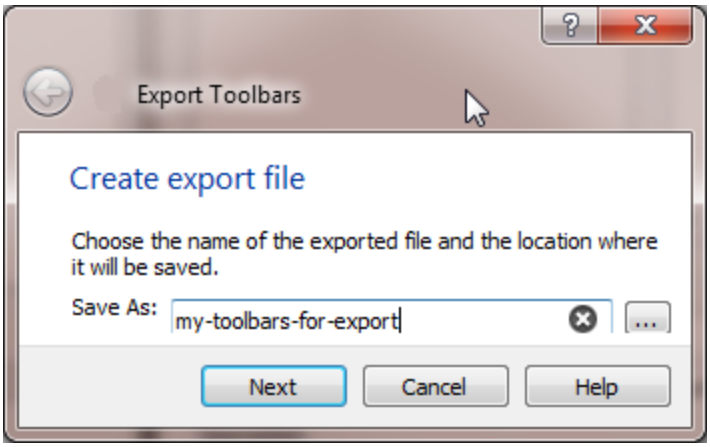

*Exporting a Toolbar*

A user may want to share a toolbar, or a set of toolbars, with another user. This is easily accomplished.

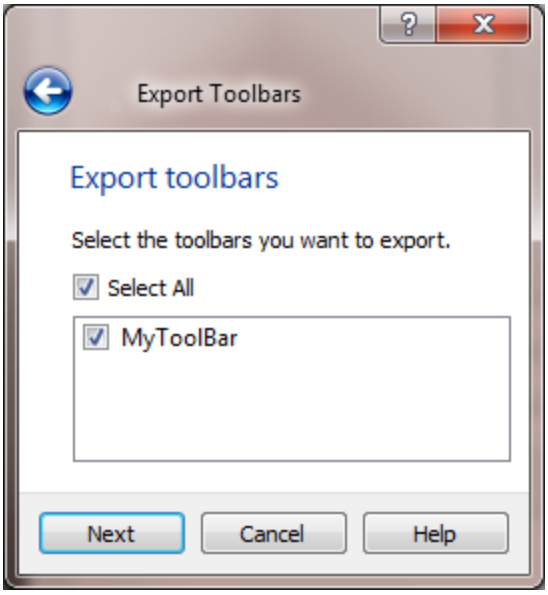
- Launch the Custom Toolbar Editor dialog by selecting the  icon.
- Or, select the **Edit** item from the toolbar's context menu
- Select **Export** from the context menu



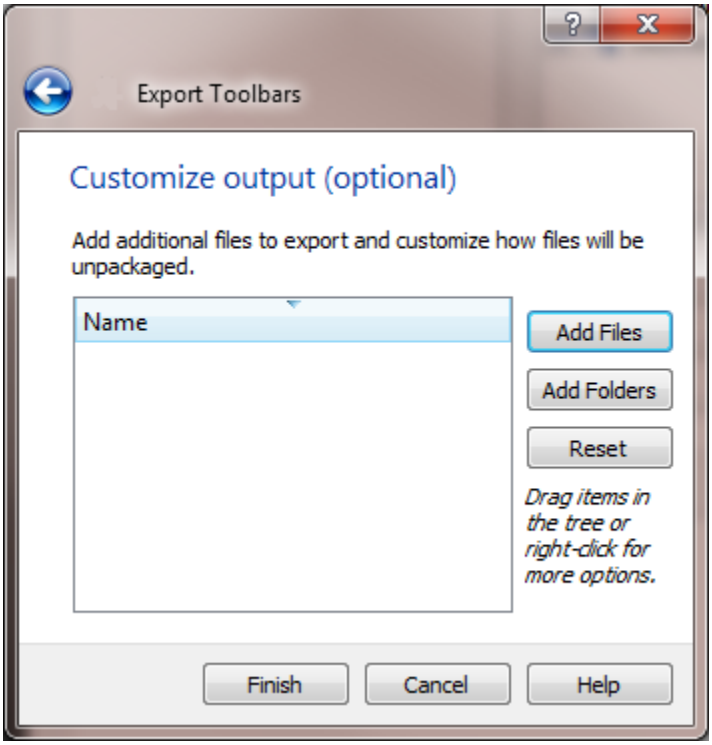

- Provide a file name to the Export Toolbars dialog. The file extension will be appended automatically
- The file type will be **.tar.gz**
- Click **Next** on the dialog




- In the next dialog select the toolbars to be included in the export
- Click **Next** on the dialog




- Optionally add files or folders that contain journal files or Python scripts referenced by tool buttons
- Click **Finish** in the dialog
- Look for the **.tar.gz** file in the designated folder




# graphics window control

## Graphics Window Control

The graphics display windows present a graphical representation of the geometry and/or the mesh. The quality and speed of rendering the graphics, the visibility, location and orientation of objects in the window, and the labeling of entities, among other things, can all be controlled by the user.

Unless the **-nographics** option was entered on the command line, a graphics window with a black background and an axis triad will appear when CUBIT is first launched. The geometry and mesh will appear in this window, and can be viewed from various camera positions and drawn in various modes (wire frame, hidden line, smooth shade, etc.). This section will discuss methods for manipulating the graphics with the mouse and for controlling the appearance of entities drawn in the graphics window.

All geometry, mesh, and simulation objects created in CUBIT are put into the view automatically. Visibility, color and various other attributes of entities in the view can be controlled individually. In addition, CUBIT can also optionally show entities in a temporary view mode independent of their visibility. Drawing of items in temporary mode can be added to the regular view mode to customize the appearance. The overall view is controlled by various attributes like graphics mode, camera position, and lighting, to further enhance the graphics functionality.

The graphics view relies on OpenGL to render the scene which leverages the graphics hardware of the system. CUBIT requires the graphics hardware to support OpenGL version 3.2 or newer. If OpenGL 3.2 is not available, CUBIT will attempt to fall back to a software based implementation bundled with CUBIT. This software based approach may not perform as well as a hardware-based approach.

The following items discuss the various graphics capabilities available in CUBIT:


- Command Line View Navigation: Rotate Zoom and Pan
- Mouse Based View Navigation: Rotate Zoom and Pan
- Updating the Display
- Graphics Modes
- Drawing and Highlighting Entities
- Drawing Locations, Lines and Polygons
- Mesh Visualization
- Graphics Clipping Plane
- Entity Labels
- Colors
- Geometry and Mesh Entity Visibility
- Graphics Camera
- Graphics Lighting Model
- Graphics Window Size and Position
- Saving Graphics Views
- Hardcopy Output
- Miscellaneous Graphics Options


## Graphics Clipping Plane

The graphics clipping plane feature allows the user to temporarily cut parts of the model away to help visualize the interior of a geometry or mesh. The command syntax is:

**Graphics Clip {On|Off} [ Plane <plane> | [Location <location>] [Direction <direction>]]**

**Graphics Clip Manipulation {On|Off}**

The GUI tool bar buttons to enable and manipulate the Graphics Clipping Plane are shown below:



The first command activates the graphics clip manipulation tools in the graphics window. The keyboard shortcut "Shift-S" while the graphics window is active will also activate the clipping plane. The manipulation of the clipping plane is controlled as follows:

- **Red Line** - Clicking and dragging the left mouse on plane bounded by a red tube moves the plane along the arrow
- **Center Ball** - Clicking and dragging the left mouse on the center ball moves the origin of the rotation plane
- **Arrow** - Clicking and dragging the left mouse button on the arrow head or tail changes the direction on which the plane moves

- **Right Mouse Button** - Clicking and dragging the right mouse button on any part of the window resizes it
- **Middle Mouse Button** - Clicking and dragging the middle mouse button on the red plane moves both the center of rotation and the cutting plane
- **White Bounding Border** - Clicking and dragging the left mouse on the white bounding border moves the whole widget

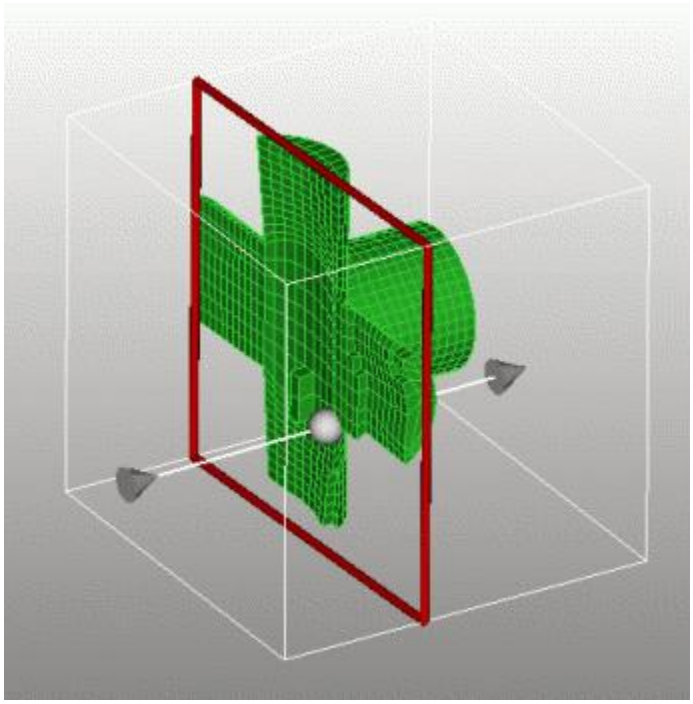


**Figure 1. Graphics Clipping Plane**

The second command turns on/off the visibility of manipulation widget in the graphics window. The clipping plane is still active, but the controls are hidden. The normal mouse-based view navigation controls apply.

*Examples*

```
brick x 10
sphere rad 1
graphics clip on location -2 0 0
rotate -45 about y
#shows the sphere inside the brick
brick x 10
cylinder rad 2 z 12
subtract 2 from 1
mesh vol 1
quality vol 1 draw mesh
```

**graphics clip on**
#shows the mesh quality on interior elements

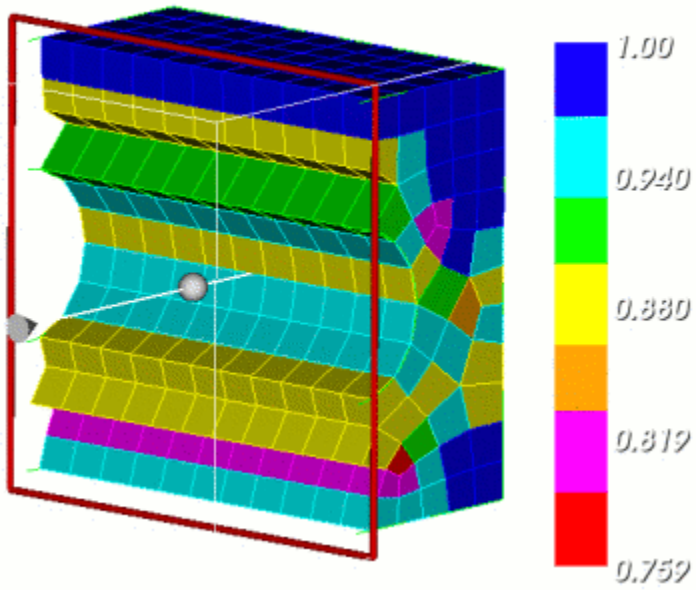



**Figure 2. Viewing mesh quality of interior elements**

## Colors

*Specifying Colors in Commands*

There are multiple ways to refer to a color in a command. They are

1. **<Color_Name>**
2. **User "name"**
3. **ID <id>**
4. **RGB**
5. **RGBA**
6. **Default**
7. **Highlight**

The first option uses the name of a pre-defined color as listed in the Available Colors Appendix. This option may not be used for user-defined colors. An example of a pre-defined color assignment is given below:

```
color volume 1 lightblue
```

The second option is used with user-defined colors only. Include the name of the user-defined color in quotes. Pre-defined colors will not work with this command.

```
color volume 1 user "mycolor"
```

The third option allows you to identify a pre-defined color by its ID. The color IDs are also listed in the Available Colors appendix. This option is rarely used.

**color volume 1 id 5**

The fourth option allows you to specify a color by R, G and B values ranging between 0.0 and 1.0. This option can be used to specify colors not found on the color table.

**color volume 1 rgb 0.7 0.4 0.35**

The fifth option allows you to specify a color by R, G, B, and A values ranging between 0.0 and 1.0. This option allows you to specify an Alpha component allowing for transparency. 1.0 is opaque and 0.0 is transparent. This option can be used to specify colors not found on the color table.

**color volume 1 rgb 0.7 0.4 0.35 0.4**

The default option is used to set an entity's color to its default value. The default color may also be specified in drawing commands, but the command's behavior will be the same as if the color option had not been included at all.

**color volume 1 default**

The seventh option refers to the current highlight color.

**draw curve 1 tangent color highlight**

*User-Defined Colors*

CUBIT has a palette of 92 pre-defined colors, listed in the Appendix under Available Colors. Users may also define their own colors in addition to those defined by CUBIT. Each color is defined by a name and by its RGB components, which range from 0 to 1. Adding user defined colors to the color table is an alternative to using RGB or RGBA color specifications.
To define an additional color, use either of the commands

**Color Define "<name>" RGB <r g b>**

**Color Define "<name>" R <r> G <g> B <b>.**

This is done with the command

**Color Release "<color_name>"**

Color names can be listed with the command

**Help Color**

They are also listed in the appendix of this manual, along with their RGB definitions. To view a chart of color names and IDs, including those for user-defined colors, use the command

**Draw Colortable**

*Assigning Colors*

Colors may be assigned to all geometric entities, and to some other objects as well. To assign a color to an entity or other object, use one of the following commands.

**Color Axis Labels {<color_name>| id <color_id> | rgb <r> <g> <b> }**

**Color Background {<color_name>| id <color_id> | rgb <r> <g> <b>} [<color_name2>|id <color_id2>| rgb <r> <g> <b>]**

**Color Block <block_id_range>{<color_name> | id <color_id> | rgb <r> <g> <b>}**

**Color Body <body_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | rgb <r> <g> <b> | Default}**

**Color Curve <curve_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | rgb <r> <g> <b> | Default}**

**Color Group <group_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | rgb <r> <g> <b> | Default}**

**Color Highlight {<color_name>| id <color_id>| rgb <r> <g> <b>}**

**Color Lines <color_spec>**

**Color NodeSet <id_range> { <color_name> | id <color_id> | rgb <r> <g> <b> | Default }**

**Color SideSet <id_range>{ <color_name> | id <color_id> | rgb <r> <g> <b> | Default }**

**Color Surface <surface_id_range> [Geometry|Mesh] {<color_name>|| rgb <r> <g> <b> Default}**

**Color Title {<color_name>|id <color_id>| rgb <r> <g> <b> }**

**Color Volume <volume_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | rgb <r> <g> <b> | Default}**

Including the Mesh keyword will change the color of the mesh belonging to the specified entity, without changing the color of the entity geometry itself. Conversely, including the Geometry keyword will change the geometry color without changing the mesh color. Including both keywords is identical to including neither keyword.
Colors are inherited by child entities. If you explicitly set the color for a volume, for example, all of its surfaces will also be drawn in that color. Once you assign a color to an entity, however, it will remain that color and will no longer follow color changes to parent

entities. To make an entity follow the color of its parent after having explicitly set another color, use Default as the color name in the color command.
Colors can also be assigned to nodesets, sidesets, and element blocks. These colors do not take effect, however, unless the nodeset, sideset, or element block is drawn with a Draw command.
The background color and the color used to draw highlighted entities can be changed to any color.
By default, the axes are labeled with a white X, Y, and Z, indicating the three primary coordinate directions. If the background is changed to white, these labels are impossible to read; the color used to draw axis labels can be changed to any color. Changing the axis label color will change the text color for both the model axis and the triad (corner axis).
When several entity types are labeled, it can become difficult to determine which labels apply to which entities. To help distinguish which entities are being referred to by the labels, you may want to change the color of labels for specific entity types.
When a meshed surface is drawn in a shaded graphics mode, the mesh edges are not drawn in the same color as the surface. This is to prevent confusion between mesh edges and geometric curves, and to make the mesh edges more visible. The color used to draw mesh edges in this situation is known as the line color, and is gray by default; this color can be changed to any color.

*Assigning Global Colors*

Colors may be assigned globally also. To assign a global color, use one of the following commands. Global color assignment is useful if one desires all entities to appear the same.

**Color Global {<color_name>| id <color_id> | rgb <r> <g> <b> | default}**

**Color Global Surface {<color_name>| id <color_id> | rgb <r> <g> <b> | default} Curve {<color_name>| id <color_id> | rgb <r> <g> <b> | default} Vertex {<color_name>| id <color_id> | rgb <r> <g> <b> | default}**

The first command assigns the desired color to all geometry entities. The color may be enter by color name or color id. The default option resets colors to the default value.
The second command assigns the desired colors to surfaces, curves and vertices. All three value must be entered. For example, users my select global colors for surface and vertex and specify that curves have default colors.

## Drawing, Locating, and Highlighting Entities

In order to effectively visualize the model, it is often necessary to draw an entity by itself, or several entities as a group. This is easily done with the command

**Draw {Entity specification} [Color <color_spec>] [Zoom] [Add]**

where Entity specification is an entity list as described in Command Line Entity Specification. This command clears the display before drawing the specified entity or entities. Specification of a color will draw those entities in that color. This will not permanently change the color of the entity. The zoom option will zoom in on the selected entities after drawing them in the graphics window. If the **add** option is specified, the display is not cleared, and the given entity is added to what is already drawn on the screen. The entities specified in this command are drawn regardless of their visibility setting (see Geometry and Mesh Entity Visibility for more details about visibility).
Entities may also be drawn by selecting them with the mouse and then typing Ctrl-D while the mouse is in the graphics window. This will clear the screen and then draw only those entities that are currently selected.
Entities can be highlighted using the command

**Highlight {Entity specification}**

This command highlights the specified entities in the current display with the current highlight color. Highlighting can be removed using the command

**Graphics Clear Highlight**

To return to the normal display of the entire model, type Display.
The **Locate** command will label and point to the specified entity or location in the graphics window. The command syntax is:

**Locate <entity_list> [<string>]**
**Locate location <options> [<string>]**

For example, suppose you have an idless reference to a curve of:

**Curve ( at 5 5 0 ordinal 1 )**

You can find the curve with the following command:

**locate location 5 5 0**

Supplying an optional string will draw the text as the label instead of the entity type and id.
Additionally, the visibility of individual entities, or sets of entities, can be controlled with the following visibility commands.

**{Vertex|Curve|Surface|Volume|Body|Group} <range> [Geometry|Mesh] Visibility {on|off}**

**Edge [Visibility] {on|off}**

**{Mesh|Geometry} [Visibility]{on|off}**

*Drawing Other Objects*

In addition to the common geometry, mesh and genesis entities, other objects may be drawn with variations of the Draw command. As with the other Draw commands, typing Display after drawing these objects will restore the scene to its normal display.

#### Displaying Entity Orientation

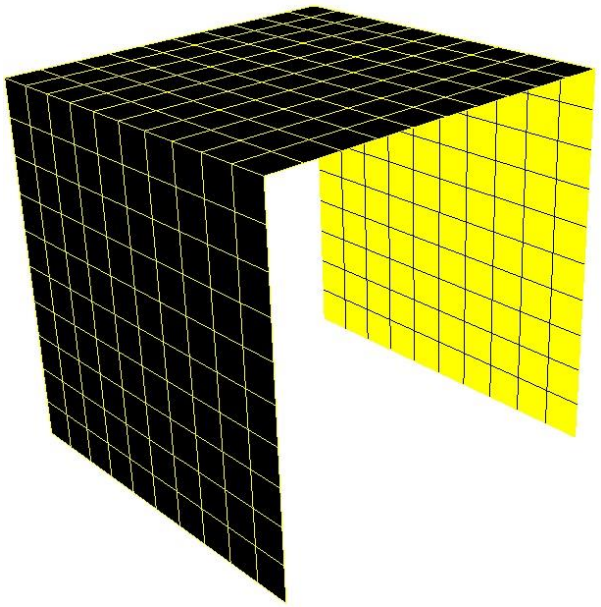
The normal to one or more surfaces, mesh faces, or mesh triangles may be drawn with the command

**Draw {Surface | Face | Tri} <id_range> Normal [Length <length>] [Face | Tri] Color <color> [Add]**

Surface normal command colors the surfaces using two different colors. The surface exposed to the positive half space (i.e, along the direction of normal), will always be colored black. The surace exposed to the negative half space will be colored using the specified <color>.

If the Face or Tri qualifier is included in the Draw Normal command, the normals for all faces or tris that belong to the specified surface are drawn.

Arrow representing the normal will be displayed if "Length" is specified



The forward, or tangent, direction of a curve can be drawn with the command:

The forward, or tangent, direction of a curve can be drawn with the command:

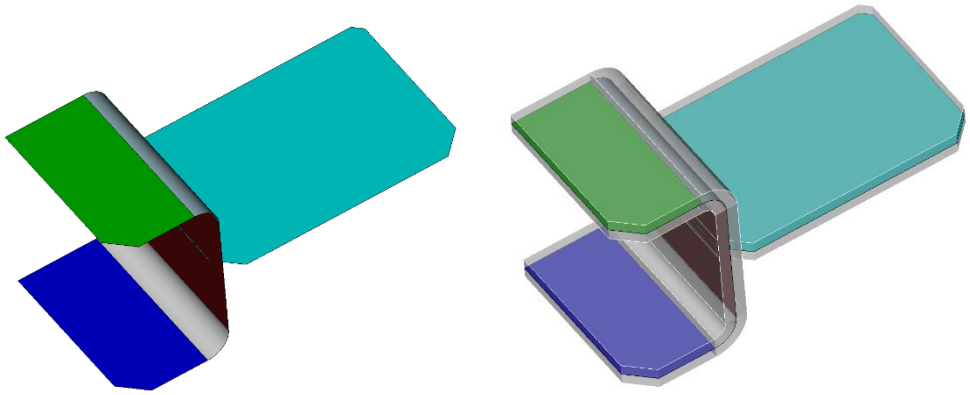
**Draw Curve <id_range> Tangent [Length <length>][Color <color_spec>]**

If a color is not specified, the tangent is drawn in the same color as the curve.

#### Displaying Sheet Volume Thickness

For volumes that can be identified as a **sheet**, an offset 3D volume can be displayed.

**Draw Volume <ids> Thickness <value> [Loft <value>] [Include_Normal] [Color <color>] [Transparent] [Add]**



Above figures show example sheet volumes (left) and the same sheet volumes displayed with their thickness (right) using the command:

**draw volume all thickness 4 loft 0.5 color white transparent add**

The **thickness** value is a positive floating number that indicates the thickness of the offset volume in a direction normal to the sheet volume. The **loft** option should be a value between 0 and 1 indicating where the preview volume will be displayed with respect to the sheet volume. A loft value of 0 will display the volume offset in the direction of the sheet volume surface normal. A value of 1 will reverse the direction of the offset. A value of 0.5 will display the volume centered on the sheet volume (as shown in the figure above). A **color** using one of Cubit's standard color keywords should also be specified. The **transparent** option can also be used to display the offset volume in transparent mode. Transparency can also be useful when using the **add** option so that both the sheet volumes and their offset volumes can be visualized together. The **include_normal** will display an arrow at the center of the sheet volume's surfaces indicating the normal direction of each surface

#### Volume Sources and Targets

Once the source and target surfaces have been set on a volume that will be meshed with the sweep algorithm, the source and target may be visually identified with the command

**Draw Volume <volume_id_range> [Source][Target] [Length <size>]**

If the Source keyword is included, the normal of the source surface or surfaces will be drawn in green into the specified volume. If the Target keyword is included, the normal of the target surface or surfaces will be drawn in red into the specified volume.

#### Model Axis

The model axis may be drawn with the command

**Draw Axis [Length <length>]**

The axis is drawn as three lines beginning at the model origin, one line in each of the three coordinate directions. The length of those lines is determined by the length parameter, which defaults to 1.

#### Surface Isoparameter Lines

Isoparameter lines may be drawn on surfaces in the model using the command

**Draw Surface <surface_id_range> Isoparametric [Number <number>| [u <number>] [v <number>]]**

If you specify the Number of lines, then the number of u- and v-parameter lines will be equal. You may specify instead a number of lines for each of the u and v parameters. The u-parameter lines will be drawn in red and the v-parameter lines will be drawn in blue.

#### Surface Overlap

The overlapping regions between two surfaces may be drawn with the command

**Draw Surface <id> <id>Overlap [Add]**

This command will draw the curves of each of the surfaces in green, and the portion of the surfaces that overlap in red. The Add keyword will draw the overlapping surfaces on top of the current graphics display. Without the Add keyword, the display will only show the specified surfaces and their overlapping regions.

#### Volume Overlap

The overlapping region between two volumes may be drawn with the command

**Draw Volume <id> <id> Overlap [Add]**

This command will draw the input volumes in transparent mode and draw the volume(s) of intersection as red, shaded solids. The **Add** keyword will draw the results on top of the current graphics display. Without the **Add** keyword, the display will only show the specified volumes along with the intersection volume(s).

#### Geometry Preview

Several options are available for previewing geometry without actually generating it. This is typically used in conjunction with webcutting and surface creation. The following Draw commands can be used for previewing geometry:

Draw Location On Curve
Draw Location
Draw Direction
Draw Line
Draw Polygon

Draw Axis
Draw Plane
Draw Cylinder

## Drawing Locations, Lines and Polygons

In some cases it may be useful to simply draw a location, line or polygon to the screen to help visualize some aspect of the model. Locations, Lines and polygons are not geometry or mesh entities and are only visible until a refresh or display command is issued.

*Drawing Locations*

**Draw Location {options}... [color <color_name>][no_flush]**

A single point or series of points may be drawn to the graphics window using this command. Any number of locations may be specified that will be drawn to the graphics window as single points. Options for specifying a location are described in the section Specifying a Location. The optional **color** argument allows for a custom color to be used. The available color definitions are located in the appendix. Other options for drawing locations and directions are also available dscribed in the section Drawing a Location, Direction, or Axis.

*Drawing Lines*

**Draw Line Location {options} Location {options} ... [color <color_name>][no_flush]**

A straight line or series of segments may be drawn to the graphics window using this command. Any number of locations may be specified that will be connected with a line. Options for specifying a location are described in the section Specifying a Location. The optional **color** argument allows for a custom color to be used. The available color definitions are located in the appendix.

*Drawing Polygons*

**Draw Polygon Location {options} Location {options} Location {options} ... [color <color_name>][no_flush]**

A filled polygon may be drawn to the graphics window using this command. Any number of locations may be specified as vertices. At least three locations must be specified. Locations for vertices can be described using any of the standard location options described in Specifying a Location. The optional **color** argument allows for a custom color to be used for the fill. The available color definitions are located in the appendix.

*Buffered Drawing*

The optional **no_flush** argument for both the **draw location**, **draw line** and **draw polygon** commands may also be used when many simultaneous **draw** commands are being issued. This prevents the graphics from being drawn after each command is issued,

which can be very inefficient. Instead the draw commands are buffered and sent all at once to be drawn. The following command:

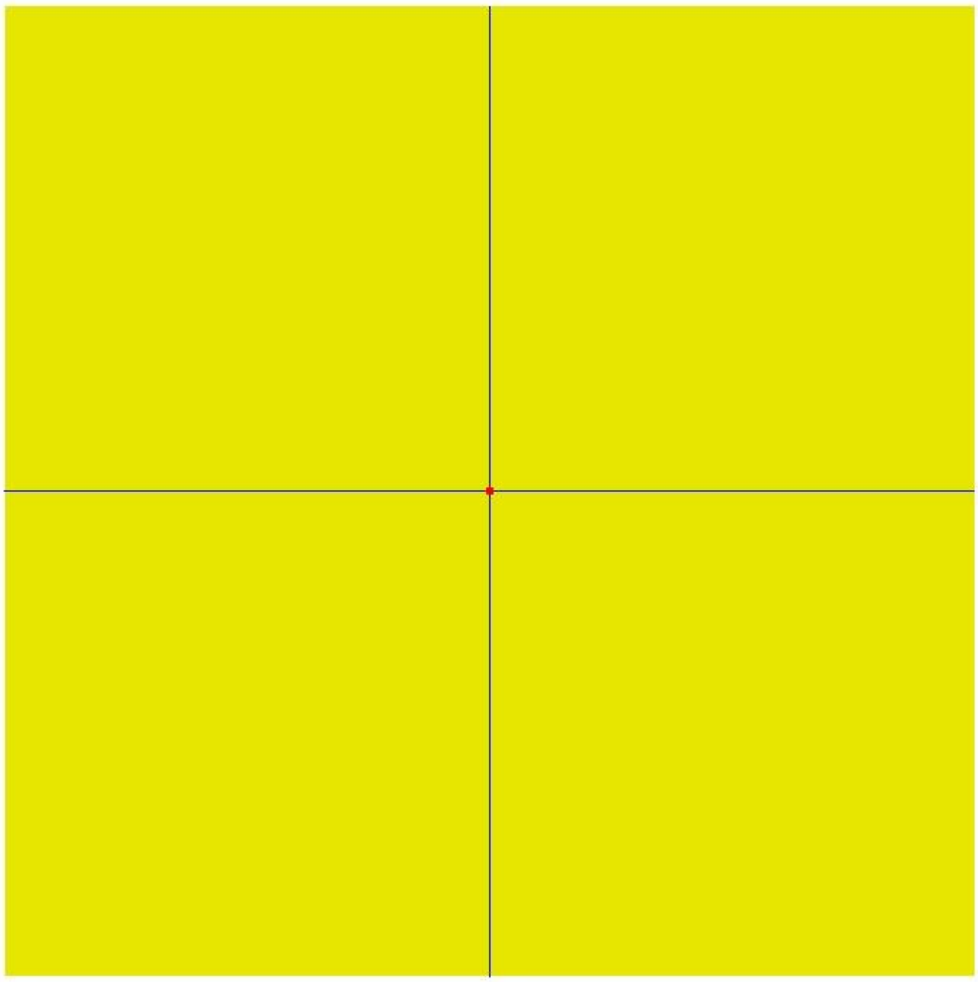
```
graphics flush
```

can be used to force a draw following a series of commands that use the **no_flush** option.

*Example*

The following is a simple example that will draw the figure below using cubit commands

```
draw polygon location pos -1 -1 0 location pos 1 -1 0 location pos 1 1 0 location
pos -1 1 0 color yellow no_flush
draw line location pos -1 0 0 location pos 1 0 0 color blue no_flush
draw line location pos 0 -1 0 location pos 0 1 0 color blue no_flush
draw location pos 0 0 0 color red no_flush
graphics flush
```



## Entity Labels

Most entities may be labeled with text that is drawn at the centroid of the entity.
Mesh entities can be labeled with their ID number or their Element ID. Element ID labels are only valid after putting the mesh entities into a block.

Geometric entities can be labeled with their ID number or with other information.
Labels for groups of entity types can be turned on or off.
The following commands will accomplish this.

**Label [On|Off|Name [Only|ID]|ID|Interval|Size|Merge|Firmness]**

**Label All [On|Off|Name [Only|ID]|ID|Interval|Size|Merge|Firmness]**

**Label Body [On|Off| Name [Only|ID] |ID|Interval|Size| Merge |Firmness]**

**Label Curve [On|Off|Name [Only|ID] |ID| Interval| Size| Merge| Firmness]**

**Label {Hex|Tet|Face|Tri|Edge} [On|Off|ElementId]**

**Label Element [On|Off]**

**Label Geometry [On|Off|Name [Only|ID] |ID| Interval| Size| Merge| Firmness]**

**Label Mesh [On|Off]**

**Label Node [On|Off|ElementId|SphereId]**

**Label Surface [On|Off|Name [Only|ID] |ID| Interval| Scheme| Size| Merge| Firmness]**

**Label Vertex [On|Off|Name [Only|ID] |ID|Interval| Size| Merge| Firmness]**

**Label Volume [On|Off|Name [Only|ID] |ID |Interval| Size |Scheme |Merge |Firmness]**

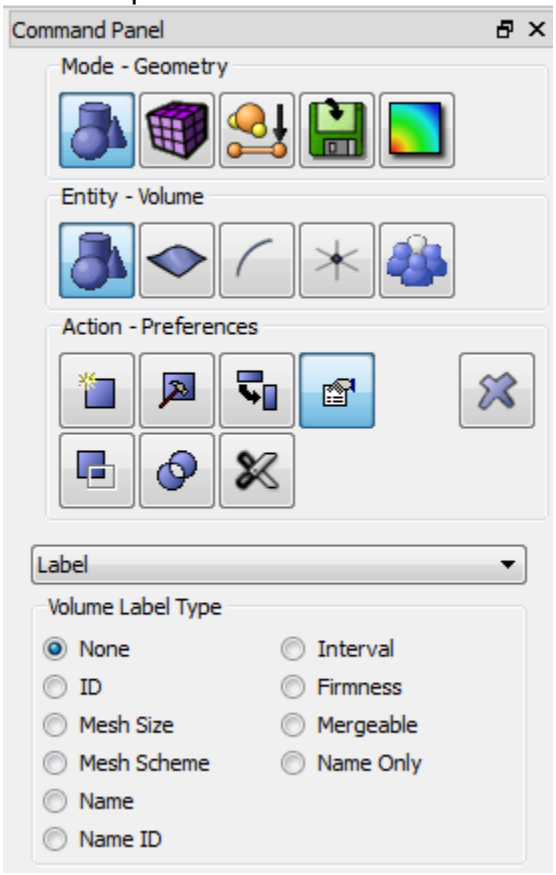
The meaning of each of each label type is listed below. Note that some label types don't make sense for every entity type.

**On** - The same as IDs.
**Name** - Name of the entity, if the entity has been named. Default name otherwise.
**Name Only** - If the entity has been named, use the name as the label. Otherwise, don't use a label.
**Name IDs** - If the entity has been named, use the name as the label. Otherwise, use the ID as the label.
**Interval** - The number of intervals set on the entity.
**Firmness** - Same as interval, but followed by a letter indicating the firmness of the interval setting (see the Mesh Generation chapter for description of firmness settings.)
**Merge** - Whether or not the entity is mergeable. Note that this is sometimes not clear, because, for example, a curve may show that it isn't mergeable because one of its owning surfaces may be unmergeable, while another owning surface may be mergeable.
**Size** - The mesh size set on this entity.

**ElementId** - The Global Element Id of each element. Will only be labeled for hexes, tets, tris, etc. which are in a block.
**SphereId** - The id of the sphere element associated with this node, if there is one. A sphere element is only associated with a node if the node (or it's geometry owner) is put into a block.

**Note:** Three dimensional entity types such as body will have their labels displayed in the center of the entity. Thus, in the **smooth shade** and **hidden line** graphics modes the labels will be hidden

The GUI includes command panels to manipulate the labels settings for any given entity type. The command panel for the Volumes labels settings is shown below as an example:




## Graphics Camera

One way to change what is visible in the graphics window is to manipulate the camera used to generate the scene. A scene camera has attributes described below, and depicted graphically in Figure 1. The values of these camera attributes determine how the scene appears in the graphics window.

These view settings may be accessed in the GUI via the **Display/View Point** menu.

**Position (From)** - The location of the camera in model coordinates.
**View Direction (At)** - The focal point of the camera in model coordinates.
**Up Direction (Up)** - The point indicating the direction to which the top of the camera is pointing. The Up point determines how the camera is rotated about its line of sight.
**Projection** - Determines how the three-dimensional model is mapped to the two-dimensional graphics window.

**Perspective Angle** - Twice the angle between the line of sight and the edge of the visible portion of the scene.

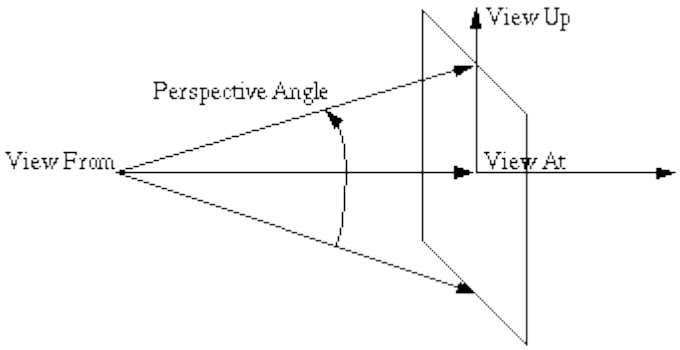



**Figure 1: Schematic of From, At, Up, and Perspective Angle**

The camera can be moved to one of several predefined orientations using the command

**View {Front | Back | Top | Bottom | Right | Left | Iso}**

At any time, the camera can be moved back to its original position and view using the command

**View Reset**

To see the current settings of these attributes, use the command

**List View**

The current value of the view attributes will be printed to the terminal window, along with other useful view information such as the current graphics mode and the width of the current scene in model coordinates.
Camera Attributes can be changed using the Rotate, Zoom and Pan commands, or directly as follows.

*Changing Camera Attributes Directly*

Camera attributes are most easily modified using interactive mouse manipulation (see Mouse-Based View Navigation) or using the rotate, pan and zoom commands. However, the camera attributes can also be modified directly with the following commands:

**From <x y z>**

**At <x y z>**

**At {Body|Volume|Surface|Curve|Vertex|Hex|Tet|Wedge|Tri|Face|Node}<id_list>**

**Up <x y z>**

**Graphics Perspective <On|Off>**

**Graphics Perspective Angle <degrees>**

If **graphics perspective** is **on**, a perspective projection is used; if graphics perspective is off, an orthographic projection is used. With a perspective projection, the scene is drawn as it would look to a real camera. This gives a three-dimensional sense of depth, but causes most parallel lines to be drawn non-parallel to each other. If an orthographic projection is used, no sense of depth is given, but parallel lines are always drawn parallel to each other.
In a perspective view, changing the **perspective angle** changes the field of view by changing the angle from the line of sight to the edge of the visible scene. The effect is similar to a telephoto zoom with a camera. A smaller perspective angle results in a larger zoom. This command has no effect when graphics perspective is off.
The GUI tool bar button for changing the graphics perspective mode is as follows:



## Graphics Modes

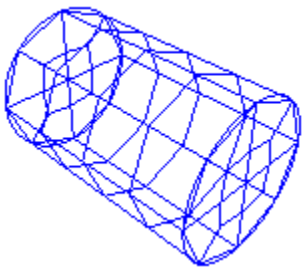
By default, the scene is viewed as a smoothshaded model. That is, only curves and edges are drawn, and surfaces are transparent. Surfaces can be drawn differently by changing the graphics mode:

**Graphics Mode {Wireframe | Hiddenline | Smoothshade | Transparent }**
**[Geometry | Mesh | Highlight]**

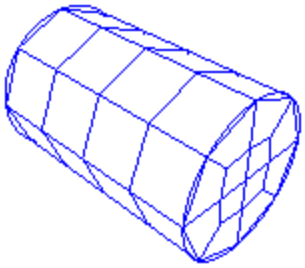
The GUI tool bar buttons for manipulating the graphics modes are as follows:



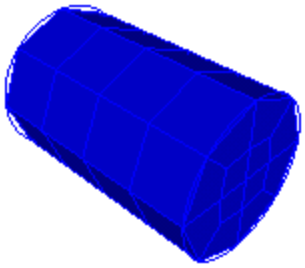
Examples and a brief description of each mode are shown below



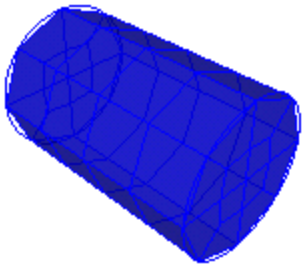
WireFrame - Surfaces are invisible. (This mode can also be accessed by typing **'wireframe'** at the command prompt.)



HiddenLine - Surfaces are not drawn, but they obscure what is behind them, giving a more realistic representation of the view. (This mode can also be accessed by typing **'hiddenline'** at the command prompt.)



SmoothShade - Surfaces are filled and shaded. Shaded colors are interpolated across the entire surface using the graphics lighting model. This produces the most realistic results. (This mode can also be accessed by typing **'shaded'** at the command prompt.)



Transparent - Renders surfaces as semi-transparent shaded images, allowing objects to shine-through from behind. Is not supported on all platforms, and generally requires advanced graphics hardware. (This mode can also be accessed by typing **'transparent'** at the command prompt.)

This determines what pattern is used to draw lines behind surfaces (e.g. dotted, dashed, etc.; click here for a list of valid line patterns).

*Displaying Using the Element Facets*

There is another option that is similar to a graphics mode, set with the command

**Graphics Use Facets [On|Off]**

This command determines how shaded and filled surfaces are drawn when they are meshed. If Graphics Use Facets is on, the mesh facets (element faces) are used to render the model. This is particularly helpful for curved surfaces which may cut through some of the mesh faces. A comparison of graphics facets on and off is shown below.

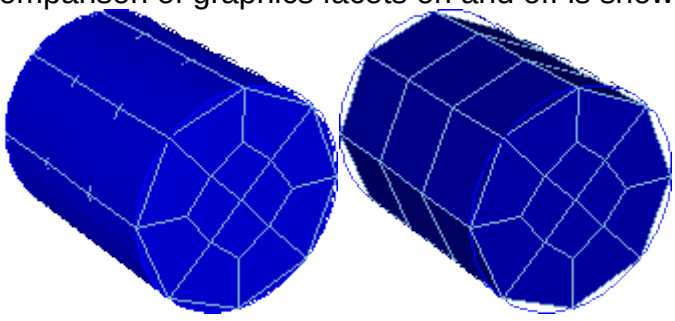


Figure 1. A meshed cylinder shown with graphics facets off (left) and graphics facets on (right); note how geometry facets on the curved surface obscure mesh edges when facets are off.

*Displaying Composite Surface Lines*

Composite surfaces are surfaces that have been joined together using virtual geometry. By default, the underlying surfaces are marked with dashed lines. To toggle this setting so that underlying surfaces are not shown, use the following command:

**Graphics Composite {On|Off}**

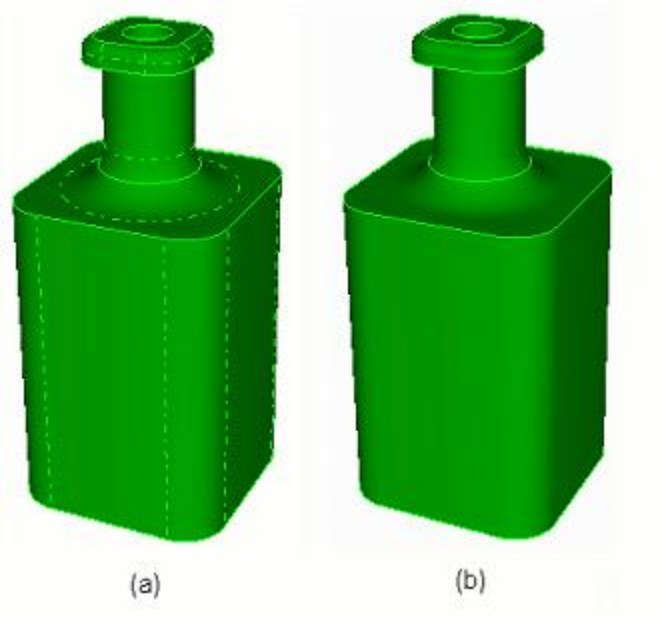



**Figure 2. A part shown with (a) composite surfaces displayed (b) composite surfaces not displayed**

The GUI tool bar button for toggling the display of graphics composites is as follows:



## Graphics Window Size and Position

By default in the command line version, CUBIT will create a single graphics window when it starts up (to run CUBIT without a graphics window, include -nographics on the command line when launching CUBIT.) The graphics window position and size is most easily adjusted using the mouse, like any other window on an X-windows screen. However, the size of the graphics window can also be controlled using the following commands:

**Graphics WindowSize <width_in_pixels> <height_in_pixels>**

**Graphics WindowSize Maximum**

**Graphics WindowSize Minimum**

After using the **Graphics WindowSize Maximum** and **Graphics WindowSize Minimum** commands, the previous window size can be restored by using the command

**Graphics WindowSize Restore**

The position of the graphics window can also be controlled using the **Graphics WindowLocation** command.

**Graphics WindowLocation <x> <y>**

The <x> and <y> coordinates refer to the distance in pixels from the upper left hand corner of the monitor.
In addition, on Unix workstations, the graphics window size and position can be controlled by placing the following line in the user's .Xdefaults file:

**cubit.graphics.geometry XxY+xpos+ypos**

where the **X** and **Y** are window width and height in pixels, respectively, and **xpos** and **ypos** are the offsets from the upper left hand corner.

*Using Multiple Windows*

You can use up to ten graphics windows simultaneously, each with its own camera and view. Each window has an ID, from 1 to 10, shown in the title bar of the window. Commands that control camera attributes apply to only one window at a time, the active window. Currently, the display lists of all windows are identical.
The following commands are used to create, delete, and make active additional graphics windows. These commands are also valid in the GUI (by typing at the command line prompt.)

**Graphics Window Create [ID]**

**Graphics Window Delete <ID>**

**Graphics Window Active <ID>**

## Hardcopy Output

CUBIT's Graphical User Interface provides the capability to print the contents of the graphics window directly to a printer. Use **File/Export/Screen Shot** to access this functionality.
In addition, a command line option is provided for dumping the contents of the graphics window to postscript or image files.
The command for generating hardcopy output files is:

**Hardcopy '<filename>' {jpg | gif | bmp | pnm | tiff | eps} [Window <window_id>]**

Each of these options saves the view in the specified window (or the current window), to the specified file, in the format indicated. The file can then be sent to a printer or inserted into another document.

*Screen Capture Programs*

It should also be noted that many commercial applications are available for capturing screen images. In many cases, these applications may be more convenient for interactively capturing and saving a portion of the screen than the **Hardcopy** command discussed above. On UNIX platforms, the XV utility written by John Bradley is a good choice. In some cases this utility or its equivalent may be included with your system software. For Windows users, the *Print Screen* button will send a copy of the screen to the clipboard which can then be pasted into a paint program.

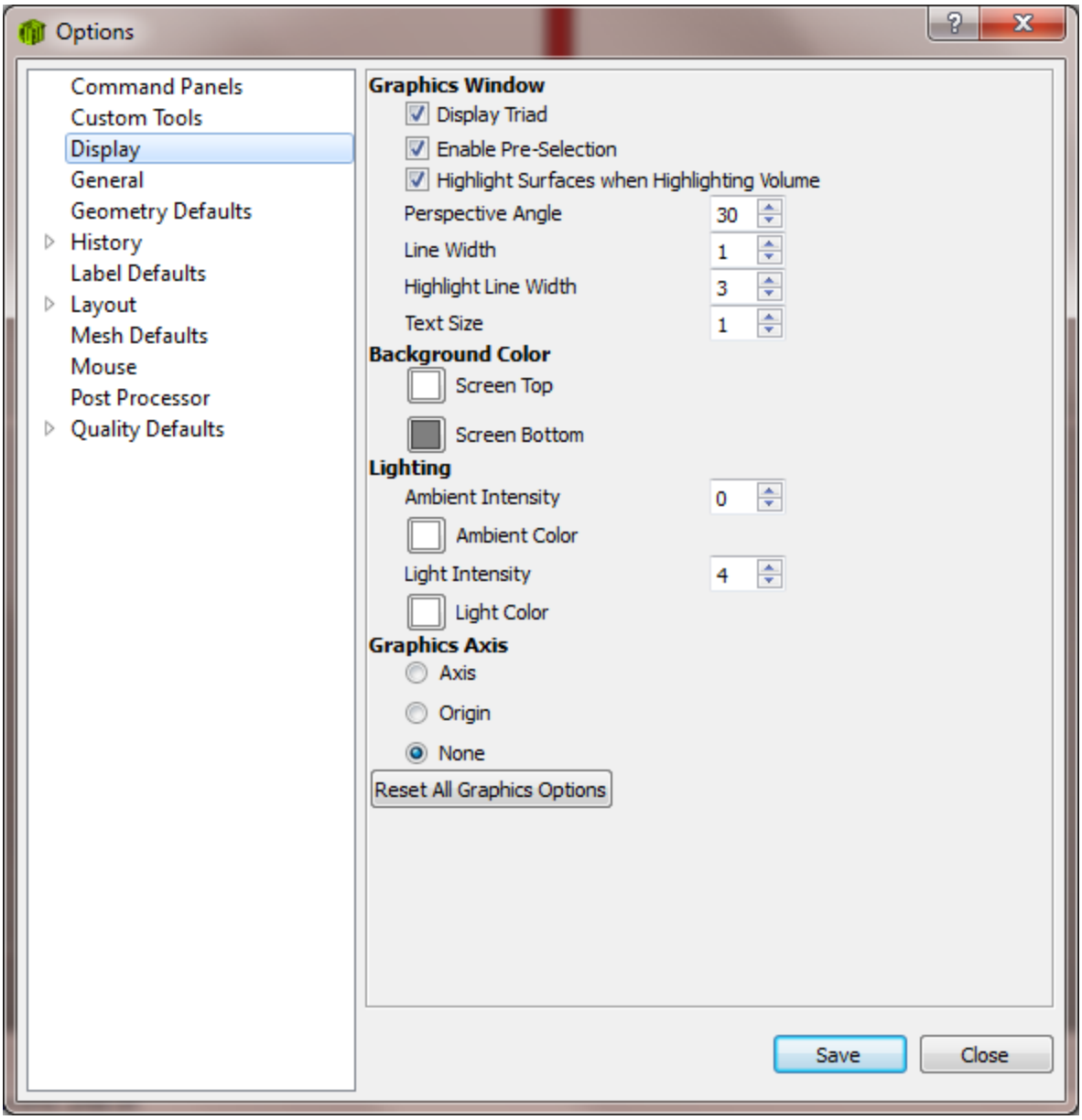
## Graphics Lighting Model

For shaded graphics display modes, the lighting model controls the intensity of the highlights and shadows for objects displayed in the graphics window. CUBIT offers two commands for controlling the lighting model.

**Graphics Ambient Intensity {<intensity> | <r g b>}**

**Graphics Light Intensity {<intensity> | <r g b>}**

The **ambient** intensity is the light available in the environment. There is no particular direction to the light source. In contrast, the **light** intensity is the effect of a simulated light source placed at the viewer's line of sight. The **light** intensity affects the intensity of the highlights and shadows, while the **ambient** intensity affects the brightness of the objects in the overall scene.
An **intensity** value from 0 to 1 can be used, where 0 represents no light and 1 represents maximum. Alternatively **r g b** color components can be used. This changes the color of the directional or ambient light source, affecting the resulting color of the objects in the model.
The GUI Options panel for manipulating these settings is found under Tools/Options and is shown below:




## Mesh Visualization

A volume mesh can be viewed one layer at a time using a visualization tool known as mesh slicing. This tool divides the elements of one or more volumes into axis-aligned layers, and then allows the mesh to be displayed one layer at a time. Mesh slicing is especially useful to view the quality of swept meshes that are axis aligned.

*Notes on Mesh Slicing*

Mesh slicing is only intended to be a rough visualization tool. Because the average mesh edge length is used to determine the thickness of each layer, a layer may be more than one element deep. Unstructured meshes, meshes with large variations in edge length, and non-axis-aligned meshes will be more difficult to visualize with this tool.

*Mesh Slicing Command*

Mesh slicing can be started either by entering a keypress in the graphics window, which slices the mesh of the entire model, or by entering the command

**Graphics Slice {Body | Volume} <id_range> Axis {X | Y | Z}**

which slices only the bodies or volumes indicated, with a plane along the axis specified. Key presses in the graphics window which control mesh slicing are summarized in the following table.

| Key | Action |
|---|---|
| **X,Y or Z** | Initiate mesh slicing using the X, Y or Z plane |
| **K** | Move the slicing plane in the positive coordinate direction |
| **J** | Move the slicing plane in the negative coordinate direction |
| **S** | Toggles drawing single or multiple slice layers in the view |
| **Q** | Exit from mesh slicing mode |

See Graphics Clipping Plane for instructions on clipping the graphics using the GUI clipping plane.

## Miscellaneous Graphics Options

In addition to the commands discussed above, there are several other graphics system options in Cubit that can be controlled by the user.
They include:

- Silhouette Lines
- Line Width
- Highlight Line Width
- Text Size
- Point Size
- Graphics Status
- Graphics Scale
- Model Axis
- Corner Axis
- Resetting the Graphics
- Shrink
- Facet Tolerance

*Silhouette Lines*

Some shapes, such as cylinders, are drawn with silhouette lines; these lines don't represent true geometric curves, but help visualize the shape of a surface. Silhouette lines can be turned on or off with the command

**Graphics Silhouette [On|Off]**

The pattern used to draw silhouette lines can be set using the command

**Graphics Silhouette Pattern [Solid | Dashdot | Dashed | Dotted | Dash_2dot | Dash_3dot | Long_dash | Phantom]**

*Line Width*

This option controls the width of the lines used in the **wireframe**, **shaded**, **transparent**, **hiddenline** and **truehiddenline** displays. The default is 1 pixel wide. The command to set the line width is

**Graphics LineWidth <width_in_pixels>**

*Highlight Line Width*

This option controls the width of the lines used when highlighting an entity. Setting this to a width greater than the global line width often makes it easier to locate highlighted entities. If this setting has not been changed, the line width set in the command above is used. After using this command, it is necessary to refresh the graphics by either typing "display" or clicking the Refresh Graphics button. The command to set the highlighting line width is

**Highlight LineWidth <width_in_pixels>**

*Text Size*

This option controls the size of text drawn in the graphics window. The size given in this command is the desired size relative to the default size. After using this command, it is necessary to refresh the graphics by either typing "display" or clicking the Refresh Graphics button. The command to set the text size is

**Graphics Text Size <size>**

*Point Size*

This option controls the size of points drawn in the graphics window, such as vertices or heads of vectors; alternatively, the size of points representing nodes or vertices can be set independently of the global point size. The commands to set the point sizes are

**Graphics Point Size <size>**

**Graphics [Node|Vertex] Point Size <size>**

*Graphics Status*

All graphics commands can be disabled or re-enabled with the command

**Graphics {On|Off}**

While graphics are off, changes in the model will not appear in the graphics window, and all graphics commands will be ignored. When graphics are again turned on, the scene will be updated to reflect the current state of the model.

*Graphics Scale*

A graphical scale can be drawn in the graphics window within the viewing area to obtain a bearing on model or part sizes. The command to turn the graphical scale on and off is:

**Graphics Scale [On|Off]**

*Model Axis*

The model axis may be drawn in the scene at the model origin. The axis is controlled with the command

**Graphics Axis [Type <AXIS | Origin>] [On|Off]**

The command is used to specify whether the model axis is visible, and to determine how the axis is drawn. If you include Type Axis , the axis will be drawn as three orthogonal lines; if you include Type Origin, the axis will be drawn as a circle at the model origin.

*Corner Axis (Triad)*

By default, an axis appears in the corner of the graphics window. This corner axis, also called the triad, can be disabled or re-enabled with the command

**Graphics Triad [On | Off]**

*Resetting the Graphics*

Many of the graphic options can be reset back to default values with the command:

**Graphics Reset**

The graphic options set to defaults are:

- ambient and spot light intensity
- background color
- text size
- graphics mode
- silhouetting

- point size
- view type (Perspective)

In addition, this command also:

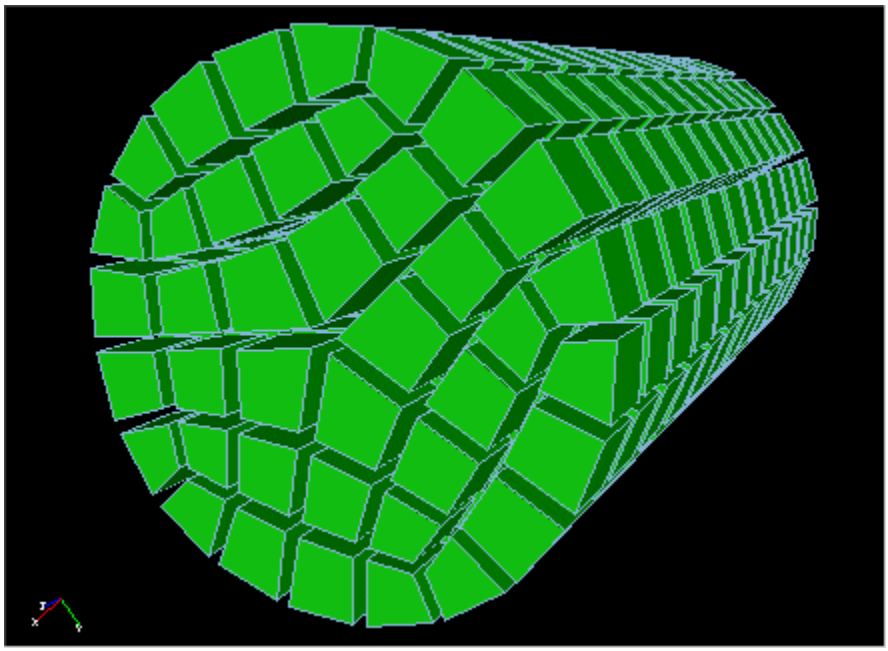
- centers the view on all visible entities (Zoom Reset)
- turns all labeling off
- turns vertex visibility off
- turns mesh and geometry visibility on
- moves the graphics camera back to its original position (View Reset)

*Shrink*

The shrink graphics attribute allows you to view the elements shrunken about their centroid. This is useful for viewing 3D meshes, permitting viewing of interior elements. It may also be useful for visually inspecting the mesh for missing elements. To use the shrink option use:

| | | |
|---|---|---|
| **graphics** | **shrink** | **<value>** |
| **draw** | **hex** | **<range>** |
| **draw** | **tet** | **<range>** |
| **etc...** | | |

where **value** is a number between 0 and 1. One (1) will shrink the elements to a point, while zero (0) will not shrink the elements. The following figures illustrate the effect of element shrink on a hex mesh.



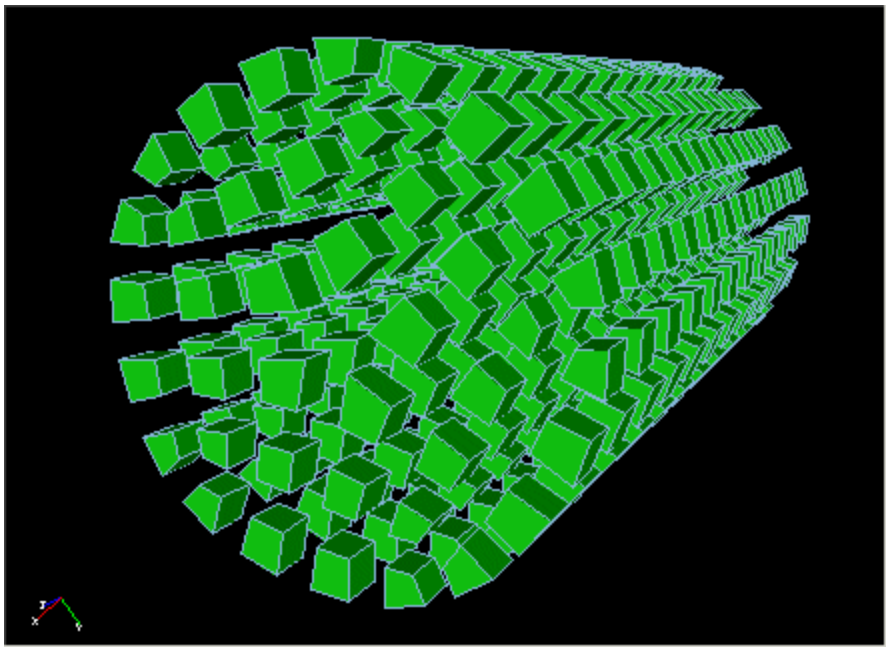


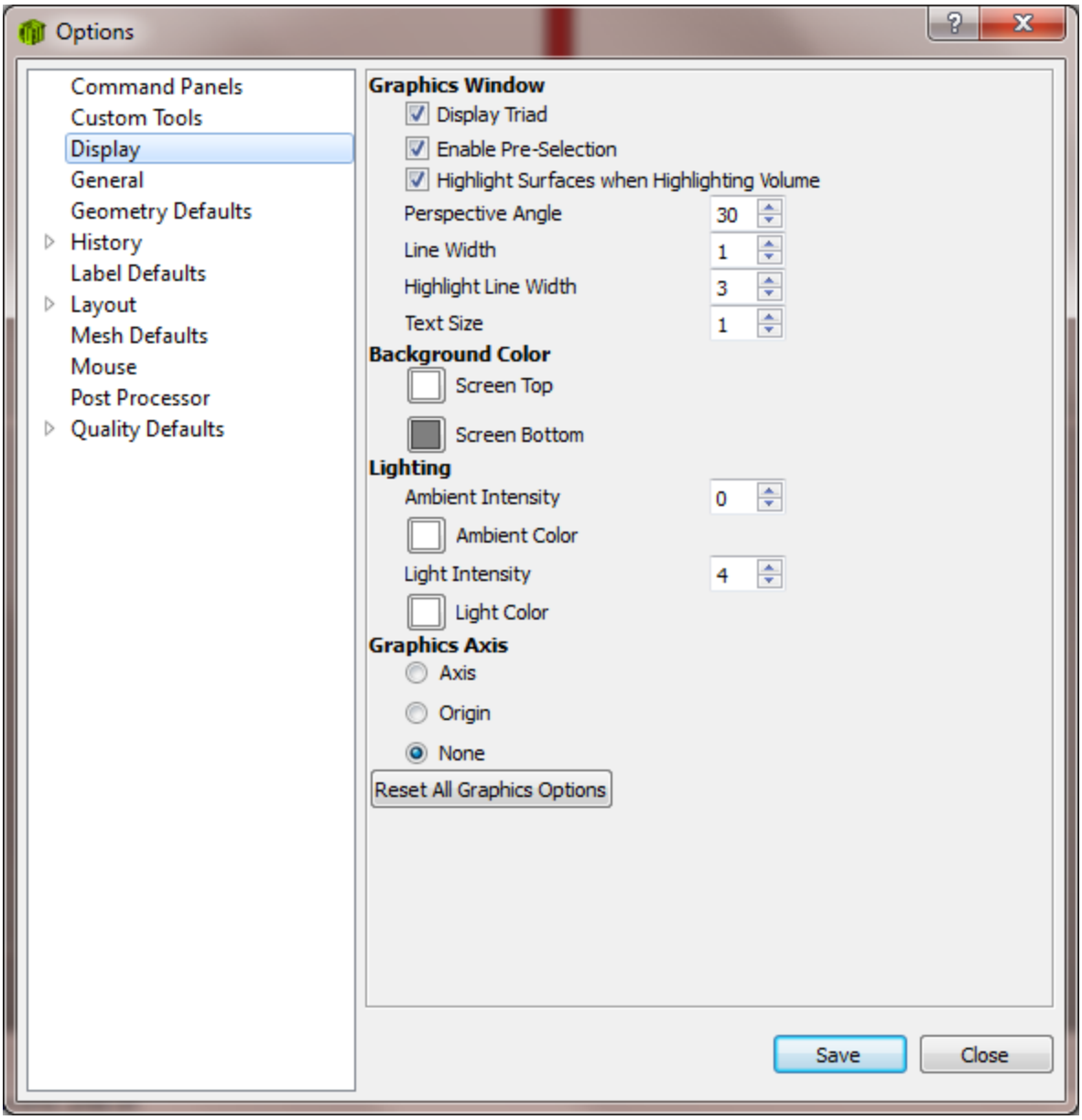
**Figure 1. Top: shrink=0.2, Bottom: shrink=0.5**

*Facet Tolerance*

The graphics tolerance commands change the way that facets are drawn in the graphics window. It does not affect the underlying geometry, just the graphics display. It can be useful to change the facet tolerance on large models if the refresh speed is slow.

**Graphics Tolerance [ [ANGLE|Distance] <val>|Default ]**

Specifying an **angle** will change the maximum allowable angle between neighboring facets. The **distance** option will set a maximum distance between adjacent facets. Increasing either of these numbers will result in coarser facets. The **default** option will return values to their default settings.
The GUI Options panel for manipulating these settings is found under Tools/Options and is shown below:




## Mouse Based View Navigation: Zoom, Pan and Rotate

The mouse can be used to navigate through the scene using various view transformations. These transformations are accomplished by clicking a mouse button in the graphics window and dragging, sometimes while holding a modifier key such as Shift or Control. When run with graphics on, CUBIT is always in mouse mode; that is, mouse-based transformations are always available, without needing to enter a CUBIT command. Mouse-based view transformations are accomplished by placing the pointer in the graphics window and then either holding down a mouse button and dragging, or by clicking on a location in the graphics window. Some functions also require one or more modifier keys to be held down; the modifier keys used in CUBIT are Shift Shift and Control Ctrl. Each of the available view transformations has a default binding to a mouse button-modifier key combination. This binding can be changed by the user if desired. Transformations and button mappings are summarized in the following table.

**Note:** These settings are applicable only to the UNIX command line version of CUBIT. For a description of the Graphical User Interface Mouse Operations see GUI View Navigation.
The bindings are based on the following mouse button definitions:

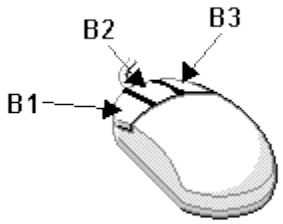



**Figure 1. Default Mouse Function Mappings for the Command Line**

**Table 1. Mouse Function Bindings for Zoom, Pan, and Rotate**

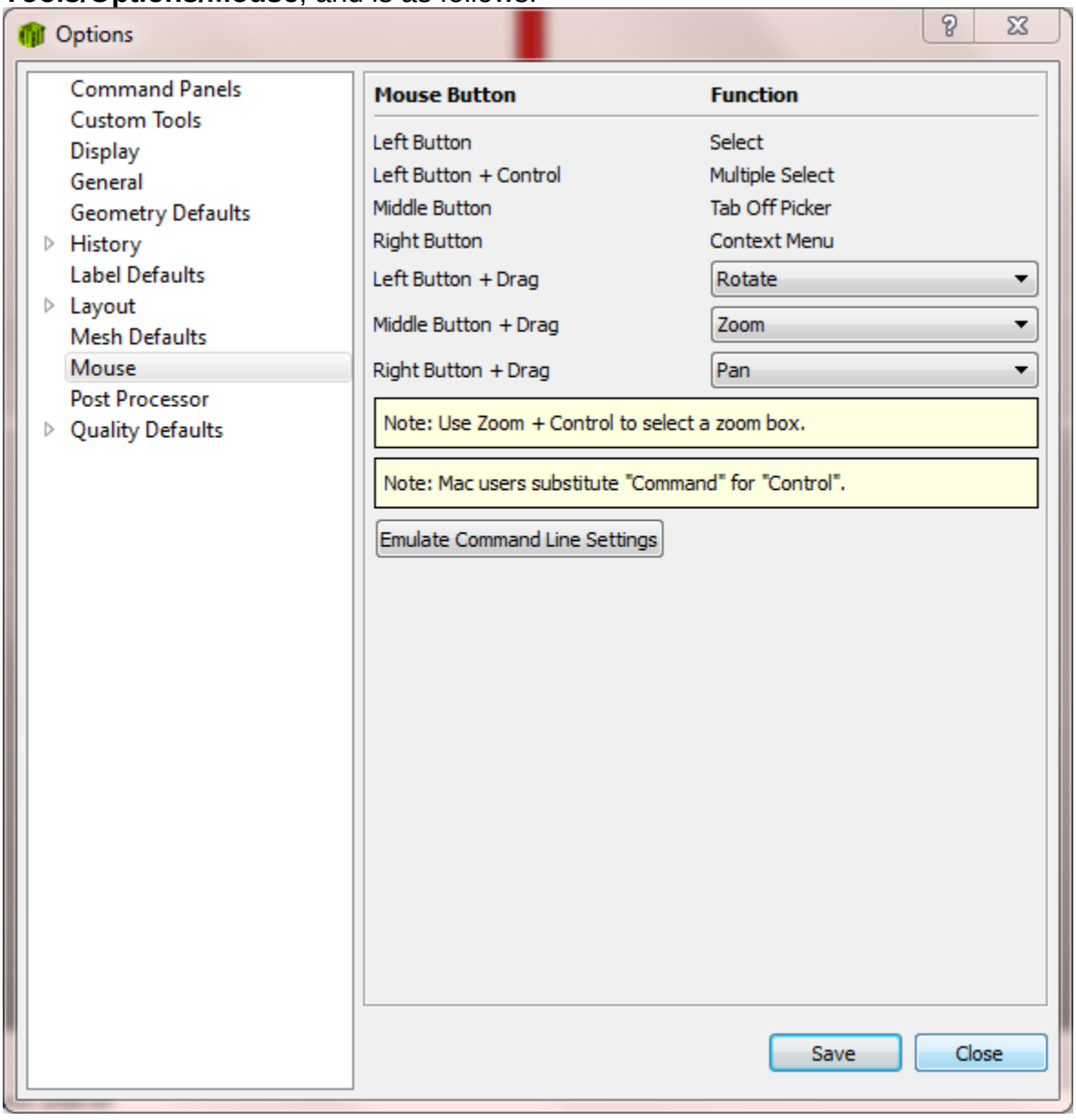
| Function | Description | Binding |
|---|---|---|
| **Rotate** | Rotates the scene about the camera axis. Dragging the mouse near the center of the graphics window will rotate the camera's X- or Y-axis; dragging near the edge of the window will rotate about the Z-axis (i.e. about the camera's line of sight). Type a u in the graphics window to see the dividing line between the two types of rotation. | **B1** |
| **Zoom** | Zooms the scene in or out by clicking the mouse in the graphics window and dragging up or down. If the mouse has a wheel, the wheel will also zoom. | **B2** |
| **Pan** | "Drags" the scene around with the mouse | **B3** |
| **Navigational Zoom** | Zooms the scene by moving both the camera and its focal point forward. | Shift **B2** |
| **Telephoto Zoom** | Zooms the scene by decreasing the field of view. | Shift Ctrl **B2** |
| **Pan Cursor** | Click on new center of view | Shift **B3** |

*Changing the View Transformation Button Bindings*

The default mapping of functions to mouse buttons, described in the **Default Mouse Function Mappings** table above, can be modified. There are two ways to assign a function to a button/modifier combination.
First, you can use the command

**Mouse Function <function_id> Button <1|2|3> [Shift][Control]**

Type **Help Mouse Function** to see a list of function IDs that may be used in this command.
Second, you can assign functions interactively. To do so, first put the pointer into a graphics window and then hit the F key. On-screen instructions will lead you through the rest of the process.
The GUI Options panel for managing the mouse bindings can be found at **Tools/Options/Mouse**, and is as follows:




*Saving and Restoring Views*

After performing view transformations, it may be useful to return to a previous view. A view is restored by setting the graphics camera attributes to a given set of values. The following keys, pressed while the pointer is in the graphics window, provide this capability:

**V** - Restores the view as it was the last time Display was entered.

**F1 to F12** - These function keys represent 12 saved views. To save a view, hold down the Control key while pressing the function key. To restore that view later, press the same function key without the Control key.

**Note:** In the Graphical User Interface version the **F1**, **F2** and **F3** keys are used as an alternate form of dynamic viewing, therefore the ability to save views is not currently supported in the GUI.

You can also save a view by entering the command

**View Save [Position <1-12>] [Window <window_id>]**

The current view parameters will be stored in the specified position. If no position is specified, the view can be restored by pressing V in the graphics window. If a position is specified, the view can be restored with the command

**View Restore Position <1-12> [Window <window_id>]**

These commands are useful in as entries in a .cubit startup file. For example, to always have **F1** refer to a front view of the model, the following commands could be entered into a .cubit file:

```
From 0 1
At 0
Up 0 1 0
Graphics Autocenter On
View Save Position 1
```

The first three commands set the orientation of the camera. The fourth command ensures that the model will be centered each time the view is restored. The final command saves the view parameters in position 1. The view can be restored by pressing **F1** while the cursor is in a graphics window.

Additionally, you can change the 'gain' on the mouse movements by changing the mouse gain setting, via the command:

**Mouse Gain <value>**

where a value of 3 would be 3X as sensitive to mouse movements, and a value of 0.5 would be half as sensitive.

**Set ReverseZoom {on|off}**

Another user preference, the direction of 'zooming' obtained by using the mouse can be 'flipped', by toggling the reversezoom setting.

## Saving Graphics Views

The current graphics view can be saved and restored using the following commands:

```
View Save Position <n>

View Restore Position <n>
```

When you save a view, you save the camera settings in effect at the time the command is issued. When you restore the view, the camera is returned to the saved position, orientation, and field of view.
If autocenter is on at the time you save the view, then restoring the view will automatically adjust the camera settings to center on the entire model and fit the entire model on the screen, a lot like "zoom reset." You turn autocenter on by typing "graphics autocenter on."
Example of how to **save a top view**:

```
at 0

from 0 1 0

up 1 0

graphics autocenter on

view save position 3
```

Use this command to restore that view:

```
view restore position 3
```

The view will then be looking down the y-axis, with the x-axis to the top and the z-axis to the right. The model will be centered in the view and zoomed so that everything just fits into the graphics window. This is true even if the model is not centered on the origin.
If autocenter is off when the "view save" command is issued, the camera is not adjusted to fit the scene into the graphics window. Instead, it is placed exactly where it was at the time the "save" command was issued.
Note that many graphics commands, such as "at", "from", and "up", do not change what appears in the graphics window until a "display" command is issued. They do, however, take immediate effect internally, and they do affect what is saved by the "view save" command.
In the command line version of CUBIT, you can save a view by holding down the shift key and pressing one of the function keys (F1-F12). Each function key corresponds to a different saved view. A total of 12 views can be saved. A view can be restored at a later time by pressing the appropriate function key WITHOUT holding down the shift key.
It may be useful to save views in your cubit file so that they are available every time you run CUBIT. Use CUBIT to save front, top, and side views in positions 1, 2, and 3. If views are saved in your cubit file, it is convenient to add a "view reset" command after the views

have been saved. Then the graphics will initially appear as they would if the view commands had not been included in your cubit file.

## Updating the Display

Among the most common graphics-related commands is:

**Display**

This command clears all highlighting and temporary drawing, and then redraws the model according to the current graphics settings. The GUI tool bar button for executing this command is:



Two related commands are:

**Graphics Flush**

**Graphics Clear**

**Graphics Flush** redraws the graphics without clearing highlighting or temporary drawing. **Graphics Flush** is useful when a previously executed command modified the graphics and didn't update the screen and the user wishes to update the display. The **Graphics Clear** command clears the graphics window without redrawing the scene, leaving the window blank.

**NOTE:** Although most changes to the model are immediately reflected in the graphics display, some are not (for graphics efficiency). Typing **Display** will update the display after such commands. **Ctrl-R** will also update the display as long as the mouse is in the graphics window.

*Prevent Graphics From Updating*

For especially large models, it may take excessively long to update the display after an action has been performed. To prevent the graphics from automatically updating, use the following command:

**Graphics Pause**

This command prevents the graphics window from being updated until the next time the **Display** command is issued**.**

**NOTE:** The **Plot** command is synonymous to the **Display** command, and either can be used with identical results**.**

## Geometry, Mesh, and BC Entity Visibility

The visibility of geometry, mesh, BC and Genesis entities can be turned on or off, either individually or globally. After visibility is turned off, the associated entities will remain invisible until visibility is turned on again.

The command to control global visibility is:

**{Mesh|Geometry|BC} { [Visibility] [on|off] }**

This command sets the global visibility on or off for *all* mesh, geometry, or BC entities, respectively. Turning off BC visibility also affects Genesis entities such as blocks, sidesets, and nodesets. Global visibility settings take precedence over the visibility set on individual entities. By default, Mesh and Geometry visibility is on, and BC visibility is off.
Global visibility of geometry, mesh, and BC entities can also be controlled from these tool bar buttons in the GUI (from left to right):



The command to control the individual visibility of geometry entities is:

**{ {Body|Curve|Surface|Volume|Vertex} <range> } [Mesh][Geometry] Visibility [On|Off]**

If the **Mesh** keyword is included, only the visibility of the *mesh* belonging to the specified geometric entity is affected. Similarly, if the **Geometry** keyword is included, only the visibility of the *geometry* is affected. If neither keyword is included, the command is identical to including both keywords.
Invisibility of geometry is inherited; visibility is not. For example, if a volume is invisible, its surfaces are also invisible unless they also belong to some other visible volume. As another case, if the volume is visible, but a surface is set to invisible, the surface will not follow its parent's visibility setting, but will remain invisible.
If vertex visibility is turned on, the vertices of the geometry become visible. The default for vertex visibility is off. The default for all other geometry entities is on.

The commands to control visibility of edges and nodes are:

**Edge [Visibility] [On|Off]**

**Node [Visibility] [On|Off]**

These commands set the global visibility on or off for *all* edges or nodes, respectively. If edge visibility is off, mesh edges will not be drawn when mesh faces are drawn. Edge visibility is on by default; node visibility is off by default. Face visibility is always on when mesh visibility is on.

The command to control the individual visibility of genesis entities is:

**{Block|Nodeset|Sideset} <range> visibility [{on|off}]**

Genesis entities and boundary conditions are best viewed with geometry and mesh visibility off and BC visibility on.

Entity visibility for individual geometry and Genesis entities can also be controlled via context (right-click) menus in the Tree and in the graphics window.
Entities that are not visible can still be drawn temporarily using the "draw" command to display one or more specific entities.

## Command Line View Navigation: Zoom, Pan and Rotate

Commands used to affect camera position or other functions are listed below. All rotation, panning, and zooming operations can include the **Animation Steps** qualifier, makes the image pass smoothly through the total transformation. Animation also allows the user to see how a transformation command arrives at its destination by showing the intermediate positions.

*Rotation*

**Rotate <degrees> About [Screen | Camera | World] {X | Y | Z} [Animation Steps <number_steps>]**

**Rotate <degrees> About Curve <curve> [Animation Steps <number_steps>]**

**Rotate <degrees> About Vertex <vertex_1> Vertex <vertex_2> [Animation Steps <number_steps>]**

Rotation of the view can be specified by an angle about an axis in model coordinates, about the camera's "**At**" point, or about the camera itself. Additionally rotations can be specified about any general axis by specifying start and end points to define the general vector. The right hand rule is used in all rotations.
Plain degree rotations are in the Screen coordinate system by default, which is centered on the camera's **At** point. The **Camera** keyword causes the camera to rotate about itself (the camera's From point). The **World** keyword causes the rotation to occur about the model's coordinate system. Rotations can also be performed about the line joining the two end vertices of a curve in the model, or a line connecting two vertices in the model.

*Panning*

**Pan [{Left|Right} <factor1>] [{Up|Down} <factor2>] [Screen | World ] [Animation Steps <number_steps>]**

Panning causes the camera to be moved up, down, left, or right. In terms of camera attributes, the **From** point and **At** point are translated equal distances and directions, while the perspective angle and up vector remain unchanged. The scene can also be panned by a factor of the graphics window size.
Screen and World indicate which coordinate system **<factor>** is in. If **Screen** is indicated (the default), **<factor>** is in screen coordinates, in which the width of the screen is one unit. If **World** is indicated, **<factor>** is expressed in the model units.

*Zooming*

**Zoom Screen <factor> [Animation Steps <number_steps>]**

**Zoom <x_min> <y_min> <x_max> <y_max> [Animation Steps <number_steps>]**

**Zoom {Group | Body | Volume | Surface | Curve | Vertex | Hex | Tet | Face | Tri | Edge | Node} <id_range> [Animation Steps <number_steps>] [Direction {options}]**

**Zoom cursor [click|drag][animation steps <number>]**

**Zoom Reset**

**Zoom Screen** will move the camera **<factor>** times closer to its focal point. The result is that objects on the focal plane will appear **<factor>** times larger.
Zooming on a specific portion of the screen is accomplished by specifying the zoom area in screen coordinates; for example, **Zoom 0 .25 .25** will zoom in on the bottom left quarter of the screen.
Zooming on a particular entity in the model is accomplished by specifying the entity type and ID after entering **Zoom**. The image will be adjusted to fit bounding box of the specified entity into the graphics window, and the specified entity will be highlighted. You can specify a final direction to look at when zooming by using the direction option.
To center the view on all visible entities, use the **Zoom Reset** command.
The GUI tool bar buttons for controlling zoom in, zoom out, and zoom reset are as follows:



# entity selection and filtering

## Entity Selection

- Command Line Entity Specification
- Extended Command Line Entity Specification
- Selecting Entities With the Mouse
- Extended Selection Dialog
- Part Classification with Machine Learning
- Training CAD Operations with Machine Learning

CUBIT Entity specification is a means of selecting objects or groups of objects. Entities can be selected from the command line using entity specification parameters, or directly in the graphics window using the mouse. This chapter describes these methods of entity selection.

### Extended Selection Dialog

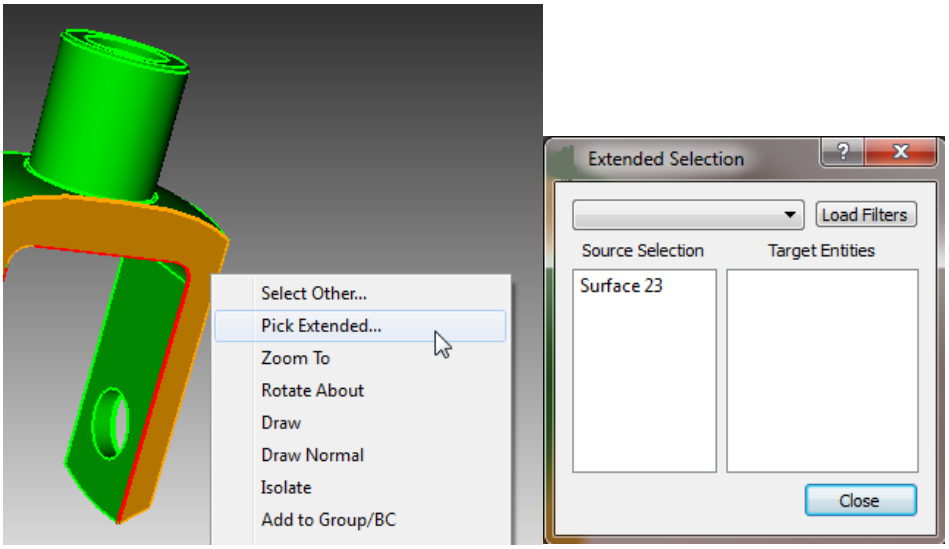
Selecting entities in the graphics window can sometimes be complicated. The Extended Selection Dialog leverages the combination of Python and the CubitInterface class to give users a very powerful mechanism for creating and managing custom selection filters.

*Accessing the Dialog*

The dialog is accessible any time a geometry entity is selected in the graphics window. Consider this workflow:

**Launch the Dialog**

- Select a geometric entity
- Get the context menu and select "Pick Extended...". The Extended Selection dialog will be shown with the selected entity (or entities) listed in the **Source Selection** window.

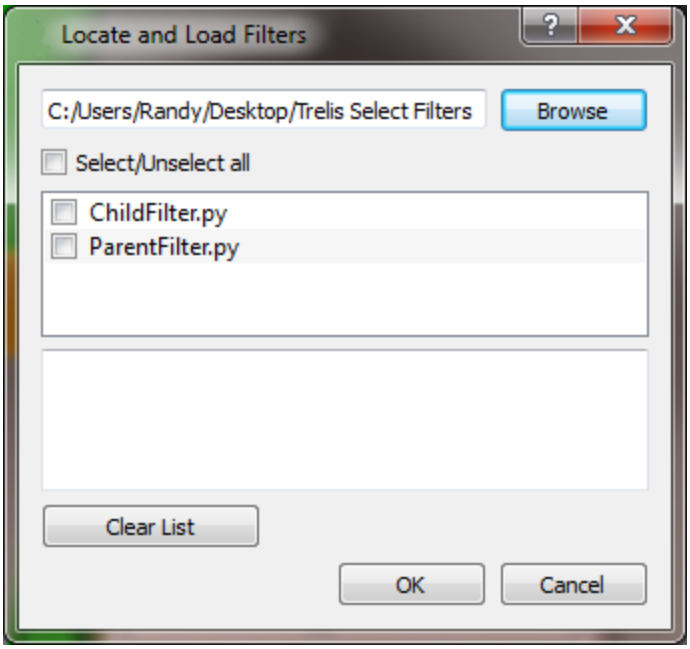



When the dialog is first shown, no filters are available.

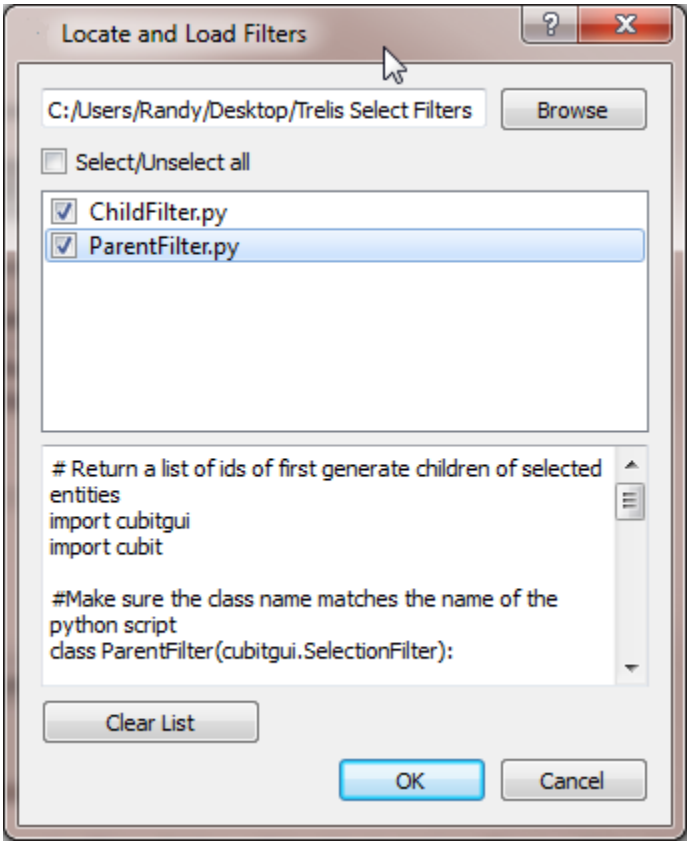
**Load Existing Filters**

Load existing filters by:

- Press the "**Load Filters**" button
- The **Locate and Load Filters** dialog will be shown




The very first time this dialog is shown, the path to the filters folder will be blank and no filters will be shown in the filter list. Press the "**Browse**" button and select the folder that contains the custom filters. This folder can be anywhere on the file system. Cubit will remember the location and use it during subsequent sessions. The folder may be changed at any time.
A list of custom filters, written in Python, will be shown. Select any given filter to examine its contents. Check all filters to be included in the menu for Extended Filters. Then press "**OK**".

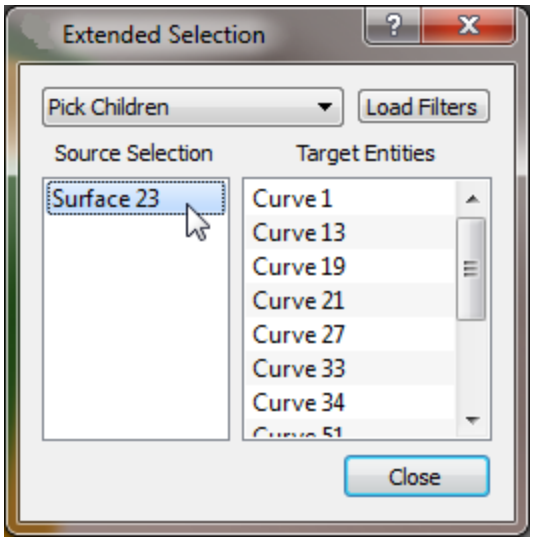



At this point, all of the filters selected in the Locate and Load Filters dialog will be available for use in the Extended Selection dialog.

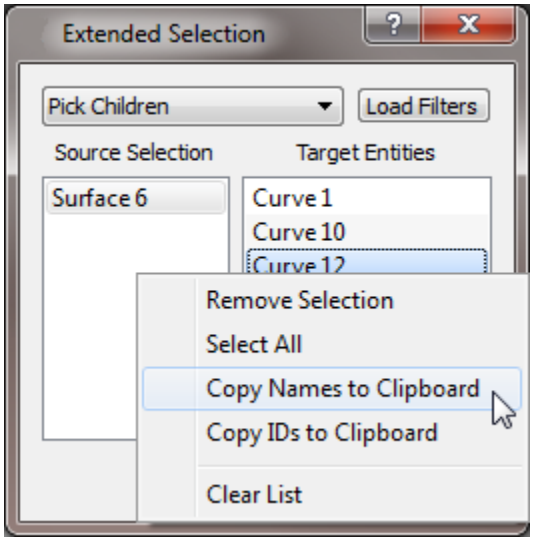



**Use a Filter**

Use the pull down menu to select a filter. Click on an entity in the **Source Selection** list. Geometric entities that fit the filter criterion will be shown in the **Target Entities** list.




- Make selections in the **Target Entities** list
- If a pickwidget is active, selected IDs will be copied into the pickwidget
- Access the context menu in the **Target Entities** list for other options, including copying entity names or IDs to the clipboard.



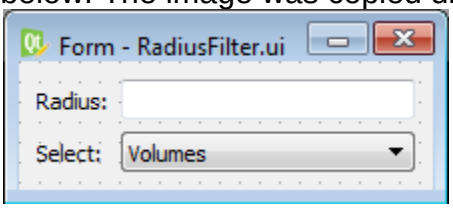

**Dragging from Target Entities to Source Selection**

Depending on the nature of the selection filter it may be useful to 're-seed' the **Source Selection** with an item from **Target Entities**. Simply drag an item or items from **Target Entities** into the **Source Selection** list.

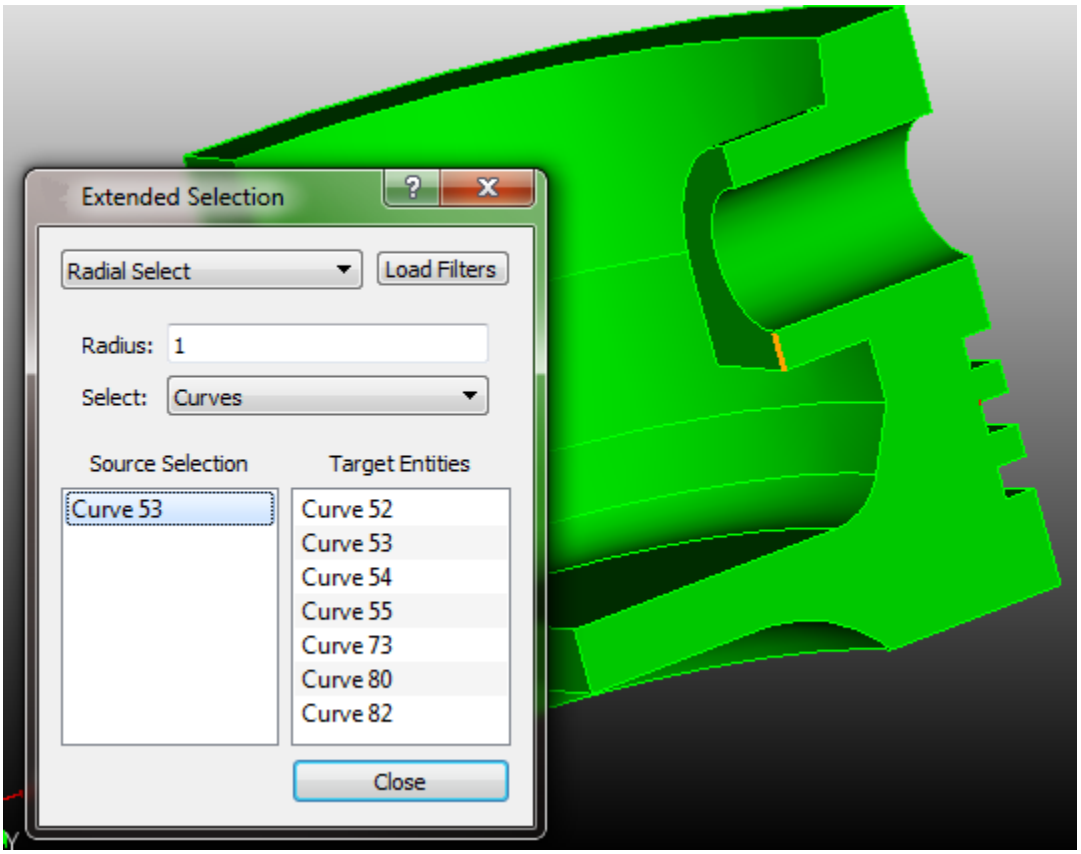
*Creating Parameterized Filters*

Parameterized filters will require a user interface which will be added into the extended selection filter dialog. The user interface can be made using Qt's Designer, which is a free

tool that ships with the Qt toolkit. The Qt Designer tool produces an XML file that will be read by Cubit and automatically included in the Extended Selection Filter dialog.
For example, if we wanted to create a selection filter that would select all entities of a certain type within a certain radius of a source entity, we would require a user interface that captures the desired radius and the desired entity type to be selected. An image of that user interface is shown below. The image was copied directly from Qt Designer.




Notice two input fields: 1) a Line Edit to capture the desired selection radius and 2) a Combo Box that contains "Volumes", "Surfaces", "Curves", "Vertices" to specify the target entity selection type. The extended selection filter that contains this custom interface is shown below. The example shows a selection of all curves within 1 unit of the source selection.




*Writing Custom Python Filters*

The class CubitInterface is used by the GUI to drive Cubit and access its database. You can read about the Python Interface used by Cubit for more details. Suffice to say, all of the functions and data included in CubitInterface are available to Python programmers.

Extended Selection custom filters are written in Python. Follow the instructions below, save the filters on the file system, then load the filters as explained above.

A Simple Example Filter

This first example shows a filter that will return a list of first generation children of the selected entities. No user input is required and no additional user interface is necessary.

ChildFilter.py

```
# Return a list of first generation children of selected entities
import cubitgui
import cubit

#Make sure the class name matches the name of the python script
class ChildFilter(cubitgui.SelectionFilter):

  # This name will be shown in the Extended Selection dialog filter menu
  def display_name(self):
    return "Pick Children"

  # This is the filter to be executed
  def run_filter(self):
    # Get whatever is in the Source Selection list
    source_types = self.get_source_types()
    source_ids = self.get_source_ids()
    self.clear_target_selections()
    for st, id in zip(source_types, source_ids):
      if st == "body":
        relativetype = "volume"
      else:
        if st == "volume":
          relativetype = "surface"
        else:
          if st == "surface":
            relativetype = "curve"
          else:
            if st == "curve":
              relativetype = "vertex"
            else:
              if st == "vertex":
                relativetype = "volume"
              else:
                return
      relative_ids = cubit.get_relatives(st, id, relativetype)
      for rid in relative_ids:
        self.add_target_selection(relativetype, rid)

```

- The filter's File Name and Class Name must match. In this example, the file name is 'ChildFilter.py' and the class name is "ChildFilter".

- LINE 2: import cubitgui -- this is the Python module that owns the base class (SelectionFilter) from which this filter class is derived.
- LINE 3: import cubit -- this is the Python module that grants access to all of the CubitInterface functions.
- LINE 6: When defining the class do the following:
    - Ensure the class name matches the file name
    - Derive the class from cubitgui.SelectionFilter
- LINE 9: Implement the 'virtual' function "display_name". Return a string that represents the name of the filter. It is this name that will be shown in the Extended Selection dialog's filter menu.
- LINE 13: Implement the 'virtual' function "run_filter. This is the actual filter.
- LINE 15: The function get_source_types() returns a list of the types selected in the "Source Selection" list of the Extended Selection dialog.
- LINE 16: The function get_source_ids() returns a list of the ids selected in the "Source Selection" list of the Extended Selection dialog.
- LINE 17: The function clear_target_selections() clears the Target Entities list of the Extended Selection dialog.
- LINE 18: The example filter begins manipulating data to suit its needs
- LINE 35: The function get_relatives(...) is a member of CubitInterface. It returns a list of ids of a specified type
- LINE 37: The function add_target_selection(<type> <id>) adds one instance of an entity-type/id to the Target Entities list.

#### The Selection Filter Class

As mentioned above, the custom Python filters must implement a class and that class must be derived from the SelectionFilter class. The SelectionFilter class is available in the Cubit SDK. The SDK is available to any user.

#### A Sample Filter that includes Additional User Interface

In order to include user input into an extended selection dialog, two additional things must happen:

1. The developer must create the user interface definition file (.ui file) using Qt Designer.
2. An additional function must be implemented in the class definition of the Python filter. The additional function is called **get_ui_file()**.

Qt UI objects supported by the extended selection filter include:

- **QLineEdit** - std::string get_line_edit_value ( object-name )
- **QComboBox** - std::string get_combo_box_value ( object-name )
- **QRadioButton** - bool get_radio_button_value ( object-name )
- **QCheckBox** - bool get_check_box_value ( object-name )
- **QSpinBox** - int get_spin_box_value ( object-name )
- **QSlider** - int get_slider_value ( object-name )

As the example Python code shows, the Qt objects are referenced by name. These code snippets below are not complete. Complete examples and a video tutorial are available from www.csimsoft.com.

```
class RadiusFilter(cubitgui.SelectionFilter):
    def display_name(self):
        return "Radial Select"

    def get_ui_file(self):
        script_dir = os.path.dirname(os.path.realpath(__file__))
        return script_dir + "/RadiusFilter.ui"

    def run_filter(self):
        # Get the radius and selection type from the ui
        try:
            radius = float(self.get_line_edit_value("leRadius"))
        except ValueError:
            print("Warning: Could not convert radius to decimal value")
            return

        combo_text = self.get_combo_box_value("cbSelectionType")
        if combo_text == "Volumes":
            selection_type = "volume"
        elif combo_text == "Surfaces":
            selection_type = "surface"
        elif combo_text == "Curves":
            selection_type = "curve"
        elif combo_text == "Vertices":
            selection_type = "vertex"
        else:
            return

        self.clear_target_selections()

        # Get the center of our current selection
        source_types = self.get_source_types()
        source_ids = self.get_source_ids()
```

```
center = [0.0, 0.0, 0.0]
for st, id in zip(source_types, source_ids):
    geom_center = cubit.get_center_point(st, id)
    center[0] += geom_center[0]
    center[1] += geom_center[1]
    center[2] += geom_center[2]

num_sources = len(source_ids)
center[0] /= num_sources
center[1] /= num_sources
center[2] /= num_sources

min_coord = [center[0] - radius, center[1] - radius, center[2] - radius]
max_coord = [center[0] + radius, center[1] + radius, center[2] + radius]
range_str = "with x_coord >= {0} and x_coord <= {1} ".format(min_coord[0], max_coord[0])
range_str += "and y_coord >= {0} and y_coord <= {1} ".format(min_coord[1], max_coord[1])
range_str += "and z_coord >= {0} and z_coord <= {1}".format(min_coord[2], max_coord[2])

potential_selection = cubit.parse_cubit_list(selection_type, range_str)
if len(potential_selection) == 0:
    return

# Go through the potential selections and verify that it is within the radius
for target_id in potential_selection:
    geom_center = cubit.get_center_point(selection_type, target_id)
    dist_squared = 0.0
    for index in range(0, 3):
        dist_squared += (geom_center[index] - center[index])**2

    if dist_squared <= radius**2:
        self.add_target_selection(selection_type, target_id)
```

## Command Line Entity Specification

CUBIT identifies objects in the geometry, mesh, and elsewhere using ID numbers and sometimes names. IDs and names are used in most commands to specify which objects on which the command is to operate.

These objects can be specified in CUBIT commands in a variety of ways, which are best introduced with the following examples (the portion of each command which specifies a list of entities is shown in blue):

**General ranges**: Surface 1 2 4 to 6 by 2 3 4 5 Scheme Pave
**Combined geometry, mesh, and genesis entities**: Draw Sideset 1 Curve 3 Hex 2 4 6
**Geometric topology traversal**: Vertex in Volume 2 Size 0.3
**Mesh topology traversal**: Draw Edge in Hex 32
**Genesis entity traversal**: Draw Block in Hex 32
***All* keyword**: ListBlock all
***Expand* keyword**: my_curve_group expand Scheme Bias Factor 1.5
***Except* keyword**: List Curve 1 to 50 except 2 4 6

In addition to the examples above, there is an extended parsing capability that allows entities to be specified by a general set of criteria. See Extended Entity Specification for details. The following is a simple example of an extended entity specification:

**By Criteria**: Draw Curve With Length > 3

*Types of Entity Range Input*

The types of entity range input available in CUBIT can be classified in 4 groups:

1. **General range parsing**

   Entity IDs can be entered individually (volume 1), in lists (volume 1 2 3), in ranges (volume 3 to 7), and in stepped ranges (volume 3 to 7 step 2). The word **all** may also be used to specify all entities of a given type.
   An ID range has the form **<start_id> to <end_id>**. It represents each ID between start_id and end_id, inclusive.
   A stepped ID range has the form **<start_id> To <end_id> {Step|By} <step>**. It represents the set of IDs between **start_id** and **end_id**, inclusive, which can be obtained by adding some integer multiple of step to start_id. For example, **3 to 8 step 2** is equivalent to **3 5 7**.
   The various methods of specifying IDs can be used together. For example:
   **draw surface 1 2 4 to 6 vertex all**

2. **Topological traversal**

   Topological traversal is indicated using the "in" and "common_to" identifiers, can span multiple levels in a hierarchy, and can go either up or down the topology tree. For example, the following entity lists are all valid:
   **vertex in volume 3**
   **volume in vertex 2 4 6**
   **surface common_to volume 2 3**
   **curve common_to surface 2 3**
   **curve 1 to 3 in body 4 to 8 by 2**
   If ranges of entities are given on both sides of the "in" identifier, the intersection of the two sets results. For example, in the last command above, the curves that have ids of 1, 2 or 3 and are also in bodies 4, 6 and 8 are used in the command.
   Topology traversal is also valid between entity types. Therefore, the following commands would also be valid:
   **draw node in surface 3**
   **draw surface in edge 362**
   **draw hex in face in surface 2**
   **draw node in hex in face in surface 2**
   **draw edge in node in surface 2**
   **draw face common_to volume 1 2**

3. **Exclusion**

Entity lists can be entered then filtered using the "except" identifier. This identifier and the ids following it apply only to the immediately preceding entity list, and are taken to be the same entity type. For example, the following entity lists are valid:
**curve all except 2 4 6**
**curve 1 2 5 to 50 except 2 3 4**
**curve all except 2 3 4 in surface 2 to 10**
**curve in surface 3 except 2 (produces empty entity list!)**
Entity names can also be used to specify the exclusion list. For example:
**curve all except pivot_1**
When using mulitple names to specify the exclusion list it is necessary to use the "in" keyword with parentheses. For example:
**curve all except curve in (pivot_1 top_left)**
In the above example, all curves are in the entity list except the curve named "pivot_1" and the curve named "top_left".

4. **Group expansion**

Groups in CUBIT can consist of any number of geometry entities, and the entities can be of different type (vertex, curve, etc.). Operations on groups can be classified as operations on the group itself or operations on all entities in the group. If a group identifier in a command is followed immediately by the `expand' qualifier, the contents of the group(s) are substituted in place of the group identifier(s); otherwise the command is interpreted as an operation on the group as a whole. If a group preceding the `expand' qualifier includes other groups, all groups are expanded in a recursive fashion.
For example, consider group 1, which consists of surfaces 1, 2 and curve 1. Surfaces 1 and 2 are bounded by curves 2, 3, 4 and 5. The commands in Table 1, illustrate the behavior of the `expand' qualifier.

**Table 1. Parsing of group commands; Group 1 consists of Surfaces 1-2 and Curve 1; Surfaces 1 and 2 are bounded by Curves 2-5.**

| **Command** | **Entity list produced** |
|---|---|
| Curve in Group 1 | Curve 1 |
| Curve in group 1 expand | Curves 1, 2, 3, 4, 5 |

The `expand' qualifier can be used anywhere a group command is used in an entity list; of course, commands which apply only to groups will be meaningless if the group id is followed by the `expand' qualifier.

*Precedence of "Except" and "In"*

Several keywords take precedence over others, much the same as some operators have greater precedence in coding languages. In the current implementation, the keyword "Except" takes precedence over other keywords, and serves to separate the identifier list into two sections. Any identifiers following the "Except" keyword apply to the list of entities excluded from the entities preceding the "Except". Table 2 shows the entity lists resulting from selected commands.

**Table 2. Precedence of "Except" and "In" keywords; Group 1 consists of Surfaces 1-2 and Curve 1.**

| Command | Entity list produced |
|---|---|
| Curve all except 1 in Group 1 | (All curves except curve 1) |
| Curve all except 2 3 4 in Surf 2 to 10 | (All curves except 2, 3, 4) |

In the first command, the entities to be excluded are the contents of the list "[Curve] 1 in Group 1", that is the intersection of the lists "Curve 1" and "Curve in Group 1"; since the only curve in Group 1 is Curve 1, the excluded list consists of only Curve 1. The remaining list, after removing the excluded list, is all curves except Curve 1.
In the second command, the excluded list consists of the intersection of the lists "Curve 2 3 4" and "Curve in Surf 2 to 10"; this intersection turns out to be just Curves 2, 3 and 4. The remaining list is all curves except those in the excluded list.

*Placement in CUBIT Commands*

In general, anywhere a range of entities is allowed, the new parsing capability can be used. However, there can be exceptions to this general rule, because of ambiguities this syntax would produce. Currently, the only exception to this rule is the command used to define a sideset for a surface with respect to an owning volume.

## Extended Command Line Entity Specification

In addition to basic entity specification, entities may be specified using an extended expression. An extended expression identifies one or more entities using a set of entity criteria. These criteria describe properties of the entities one wishes to operate upon.

*Extended Parsing Syntax*

The most common type of extended parsing expression is in the following format:

```
{Entity_Type} With {Criteria}
```

Entity_Type is the name of any type of entity that can be used in a command, such as Curve, Hex, or SideSet. Criteria is a combination of entity properties (such as Length), operators (such as >=), keywords (such as Not), and values (such as 5.3) that can be evaluated to true or false for a given entity. Here are some examples:

```
curve with length <1
surface with is_meshed = false
node with x_coord > 10 And y_coord > 0
```

*Keywords*

These are the keyword defined by extended parsing

| Keyword | Description |
|---|---|

| | |
|---|---|
| **All, To, Step, By, Except, In, Common_To, Expand, Include** | These keywords are used the same way as in basic entity specification. For example:<br>**draw surface all**<br>**draw surface 1 to 5 step 2 curve 1 to 3 in body 4 to 8 by 2**<br>**draw hex in face in surface 2**<br>**draw face common_to volume 1 2**<br>**draw node in hex in face in surface 2 curve 1 2 5 to 50 except 2 3 4**<br>**draw volume 10 include similar**<br>**list block in hex 10** |
| **Not** | Not flips the logical sense of an expression - it changes true to false and false to true. For example:<br>**draw surface with not is_meshed** |
| **Of** | The "of" operator is used to get an attribute value for a single entity, such as "length of curve 5". Only attributes that return a single numeric value may be used in an "of" expression. There must be only one entity specified after the "of" operator, but it can be identified using any valid entity expression. An example of a complete command which includes the "of" operator is:<br>**list curve with length < length of curve 5 ids** |
| **And, Or** | These logic operators determine how multiple criteria are combined.<br>**draw surface with length > 3 or with is_meshed = false** |
| **< > <= >= = <>** | These relational operators compare two expressions. You may use = or == for "equals". <> means "not equal". For example:<br>**draw surface with x_max <= 3**<br>**draw volume with z_max <>12.3** |
| **+ - * /** | These arithmetic operators work in the traditional manner.<br>**draw surface with length * 3 + 1.2 > 10** |
| **( )** | Parentheses are used to group expressions and to override precedence. When in doubt about precedence, use parentheses.<br>**draw surface with length > 3 and ( with is_meshed = false or x_min > 1 )** |

*Functions*

The following functions are defined. Not all functions apply to all entities. If a function does not apply to a given entity, the function returns 0 or false.

| Keyword | Description |
|---|---|
| **ID** | the ID of an entity |
| **Length** | The length of a curve or edge |
| **Area** | The area of a surface. |
| **Volume** | The volume of a volume. |
| **Exterior_Angle** | Works for curves with an exterior angle greater than (>), less than (<), or equal to (=) a given angle in degrees. This is used if you want to do some operation, such as refinement, on all the reentrant curves or curves with surfaces that form a certain angle. |
| **Radius** | Radius of an arc or blend surface. |
| **Normal** | Compares the normal of a surface to a vector.<br>**draw surface with normal 1 0 0**<br>The given vector is normalized to a unit vector for comparison. The two vectors are compared using the magnitude of the difference between them. A default tolerance of 1e-6 is used unless specified.<br>**draw surface with normal 1 0 0 tolerance 0.001** |
| **Type** | Compares the element type of a block.<br>**select block with type "tetra10"**<br>The element type needs to be in quotes. The generic element type can be used to select any order of that element type. For example:<br>hex => hex8, hex20, hex27 |
| **Is_Duplicate** | Whether a Volume has a duplicate (exact copy of itself at the same location) |
| **Is_Meshed** | Whether a geometric entity has been meshed or not |
| **Is_Spline** | Whether a geometric entity is defined using a NURBS representation. Otherwise the entity has an analytic representation. |
| **Is_Blend** | Whether a geometric surface is a blend. Blends have a constant principal radius of curvature and meet two or more adjoining surfaces at an angle of approximately 180 degrees. "Fillets" are examples of blend surfaces. |
| **Is_Chamfer** | Whether a geometric surface is a chamfer. Chamfers are thin surfaces bounded by surfaces where the exterior angle is approximately 45 or 225 degrees. |
| **Is_Plane** | Whether a geometric surface is planar. |

| | |
|---|---|
| **Is_Periodic** | Whether a geometric surface is periodic, such as a sphere or torus. |
| **Is_Sheetbody** | A geometric entity is a sheetbody if it is a collection of surfaces that do not form a solid. |
| **Element_Count** | The number of elements owned by this geometric or exodus entity. Only elements of the same dimension as the geometric entity are counted (number of hexes in a volume, number of faces on a surface, etc.). For an exodus entity, all contained mesh entities are counted. |
| **Dimension** | The topological dimension of an entity (3 for volumes, 2 for surfaces, etc.). |
| **X_Coord, Y_Coord, Z_Coord** | The x, y, or z coordinate of the entity's bounding box center point projected onto the entity. |
| **X_Min, Y_Min, Z_Min** | The x, y, or z coordinate of the minimum extent of the entity's bounding box |
| **X_Max, Y_Max, Z_Max** | The x, y, or z coordinate of the maximum extent of the entity's bounding box |
| **Is_Merged** | Whether a geometry entity has a merge flag on. All geometric entities have one set by default. |
| **Is_Virtual** | A flag that specifies whether an entity is virtual geometry. An entity is virtual if it has at least one virtual (partition/composite) topology bridge. |
| **Has_Virtual** | An entity "has_virtual" if it is virtual itself, or has at least one child virtual entity |
| **Is_Real** | An entity "is_real" if it has at least one real (non-virtual) topology bridge. |
| **Num_Parents** | Used to specify geometry entities with a specified number of parent entities. May be used to find "free curves" where num_parents=0 or non-manifold curves where num_parents>2. |
| **Is_Free** | Used to specify geometry entities without parent entities. May be used to find free curves and vertices where num_parents=0 or free surfaces that do not form a solid (sheet bodies). |
| **Block_Assigned** | Used to specify elements which have been assigned to a block. This is also useful to find elements NOT assigned to a block by using "not block_assigned". |
| **Has_Scheme** | Used to specify geometry entities which have been assigned a specified scheme. The scheme name is specified with the keyword string used when setting the |

| | |
|---|---|
| | scheme. Wildcards can also be used when specifying the scheme name. For example, **draw surface with has_scheme '*map'** will draw surfaces with scheme map or submap. |
| **Similar** | Used with the **include** keyword. Compares a list of geometry entities and adds additional entities that are classified as similar. Implemented for **curves**, **surfaces** and **volumes**. Similar is defined as the same geometric length, area or volume using a tolerance of 0.1 percent, as well as the same number of child entities. For example, **"draw volume 10 include similar"** will draw all volumes with the same geometric volume and number child surfaces as volume 10. |
| **Cavity** | Used with the **include** keyword. Compares a list of surface entities and adds additional adjacent surfaces that are part of the same cavity. Implemented only for **surfaces**. A cavity is defined as the collection of surfaces bounded by curves where the exterior angle is greater than 180 degrees. For example, **"draw surface 10 include cavity"** will draw surface 10 along with the cavity to which it belongs. |
| **Hole** | Used with the **include** keyword. Compares a list of surface entities and adds additional adjacent surfaces that are part of the same hole. Implemented only for **surfaces**. A hole is a special case of a cavity that includes at least one cylindrical surface. For example, **"draw surface 10 include hole"** will draw surface 10 along with the hole to which it belongs. |
| **Blend_Chain** | Used with the **include** keyword. Compares a list of surface entities and adds additional adjacent surfaces that are part of the blend_chain. Implemented only for **surfaces**. A blend_chain is defined as the collection of attached surfaces that have the same minimum radius of curvature. For example, **"draw surface 10 include blend_chain"** will draw surface 10 along with the blend_chain to which it belongs. |
| **Chamfer_Chain** | Used with the **include** keyword. Compares a list of surface entities and adds additional adjacent surfaces that are part of the chamfer_chain. Implemented only for **surfaces**. A chamfer_chain is defined as the collection of attached surfaces that have the same width and are bounded by exterior angles of 135 and/or 225 degrees. For example, **"draw surface 10 include** |

| | |
|---|---|
| | **chamfer_chain"** will draw surface 10 along with the chamfer_chain to which it belongs. |
| **Continuous** | Used with the **include** keyword. Compares a list of curve or surface entities and adds additional adjacent entities that are continuous. Implemented for **curves** and **surfaces**. For surfaces, **continuous** is defined as the collection of attached surfaces that are bounded by curves where the exterior angle is approximately 180 degress (tolerance = 15 degrees). For curves, **continuous** is defined as the collection of attached curves that are bounded by vertices where the angle is approximately 180 degrees (tolerance = 15 degrees). For example, **"draw surface 10 include continuous"** will draw surface 10 along with other surfaces within the same continuous collection. |
| **Nearby** | Used with the **include** keyword. It will include additional geometry entities of the same dimension that are within a small distance of the specified entities. Implemented for **curves**, **surfaces** and **volumes**. The distance tolerance is automatically defined relative to the size of the entity(s). An example use case would be, **"draw volume 10 include nearby"** which will draw volume 10 along with volumes that are within a small distance of volume 10. |

*Precedence*

For complicated expressions, which entities are referred to is influenced by the order in which portions of the expression are evaluated. This order is determined by precedence. Operators with high precedence are evaluated before operators with low precedence. You may always include parentheses to determine which sub-expressions are evaluated first. Here all operators and keywords listed from high to low precedence. Items listed together have the same precedence and are evaluated from left to right.

**(, ) Expand Not *, / +, - <, >, <=, >=, <>, = And, Or Except In Of With**

Because of precedence, the following two expressions are identical:

```
curve with length + 2 * 2 > 10 and length <= 20 in my_group

expand(curve with (((length + (2*2)) > 10 )and( length <= 20 ))) in ( my_group expand )
```

## Selecting Entities with the Mouse

The following discussion is applicable only to the command line version of CUBIT. See GUI Entity Selection for a description of interactive entity selection with the Graphical User Interface. Also refer to Extended Selection Dialog to learn how to use Python scripts to create extensive selection capabilities.

Many of the commands in CUBIT require the specification of an entity on which the command operates. These entities are usually specified using an object type and ID (see Entity Specification) or a name. The ID of a particular entity can be found by turning labels on in the graphics and redisplaying; however, this can be cumbersome for complicated models. CUBIT provides the capability to select with the mouse individual geometry or mesh entities. After being selected, the ID of the entity is reported and the entity is highlighted in the scene. After selecting the entities, other actions can be performed on the selection. The various options for selecting entities in CUBIT are described below, and are summarized in Table 1:

**Table 1. Picking and key press operations on the picked entities**

| Key | Action |
|---|---|
| **ctrl + B1** | Pick entity of the current picking type. |
| **shift + ctrl + B1** | Add picked entity of the current picking type to current picked entity list. |
| **tab** | Query-pick; pick entity of current picking type that is below the last-picked entity. |
| **n** | Lists what entities are currently selected. |
| **l** | Lists basic information about each selected entity. This is similar to entering a List command for each selected entity. |
| **g** | Lists geometric information about the selection. As if the List Geometry command were issued for each entity. If there are multiple entities selected, a geometric summary of all selected entities is printed at the end, including information such as the total bounding box of the selection. |
| **i** | Makes the current selection invisible. This only affects entities that can be made invisible from the command line (i.e. geometric and genesis entities.) |
| **s** | Draws a graphical scale showing model size in the three coordinate axes. This is a toggle action, so pressing the 's' key again in the graphics window will turn the scale off. |
| **ctrl + z** | Zoom in on the current selection. |
| **e** | Echo the ID of the selection to the command line. |

| | |
|---|---|
| **a** | Add the current selection to the picked group. Only geometry will be added to the group (not mesh entities). If a selected entity is already in the picked group, it will not be added a second time. |
| **r** | Remove the current selection from the picked group. If a selected entity was not found in the picked group, this command will have no effect. |
| **ctrl + r** | Redisplays the model. |
| **c** | Clear the picked group. The picked group will be empty after this command. |
| **m** | Lists what entities are currently in the picked group. |
| **d** | Display and select the entities in the picked group. |
| **ctrl + d** | Draws the entity that is selected. |

Details of selecting entities with a mouse are outlined in the following items:

- Entity Selection
- Query Selection
- Multiple Selected Entities
- Information about the Selection
- Picked Group
- Substituting the Selection into Commands
- Select Commands

*Entity Selection*

Selecting entities typically involves two steps:
**1. Specifying the type of entity to select**
Clicking on the scene can be interpreted in more than one way. For example, clicking on a curve could be intended to select the curve or a mesh edge owned by that curve. The type of entity the user intends to select is called the picking type. In order for CUBIT to correctly interpret mouse clicks, the picking type must be indicated. This can be done in one of two ways. The easiest way to change the picking type is to place the pointer in the graphics window and enter the dimension of the desired picking type and an optional modifier key. The dimension usually corresponds to the dimension of the objects being picked:

**Table 2. Picking Modes in Graphics Window**

| Number | Default pick | Number +shift pick |
|---|---|---|
| 0 | vertices | nodes |
| 1 | curves | edges |

| | | |
|---|---|---|
| 2 | surfaces | all 2D elements |
| 3 | volumes | all 3D elements |
| 4 | bodies | |

If a Shift modifier key is held while typing the dimension, the picking type is set to the mesh entity of corresponding dimension, otherwise the geometry entity of that dimension is set as the picking type. For example, typing 2 while the pointer is in the graphics window sets the picking type so that geometric surfaces are picked; typing Shift-1 sets the picking type so that mesh edges are picked. To differentiate between picking "tris" or "quads" use "**pick face**" or "**pick tri**"
The picking type can also be set using the command

**Pick <entity_type>**

where entity_type is one of the following: Body , Volume , Surface , Curve , Vertex , Hex , Tet , Face , Tri , Edge , Node , or DicerSheet .
2. **Selecting the entities**
To select an object, click on the entity (this command can be mapped to a different button and modifiers, as described in the section on Mouse-Based View Navigation). Clicking on an entity in this manner will first de-select any previously selected entities, and will then select the entity of the correct type closest to the point clicked. The new selection will be highlighted and its name will be printed in the command window.

*Query Selection*

If the highlighted entity is not the object you intended to selected, press the Tab key to move to the next closest entity. You can continue to press tab to loop through all possible selections that are reasonably close to the point where you clicked. Shift-Tab will loop backwards through the same entities.

*Multiple Selected Entities*

To select an additional entity, without first clearing the current selection, hold down the control key while clicking on an object. You can select as many objects as you would like. By changing the picking type between selections, more than one type of entity may be selected at a time. When picking multiple entities, each pick action acts as a toggle; if the entity is already picked, it is "unpicked", or taken out of the picked entities list.
To select entities using rubberband, hold the control key, click and drag to enclose the entities to select. Different rubberband shapes are available to use: box, circle and polygon. A toolbar button is provided to toggle between the different shapes.

*Information About the Selection*

When an entity is selected, its name, entity type, and ID are printed in the command window. There are several other actions which can then be performed on the picked entity

list. These actions are initiated by pressing a key while the pointer is in the graphics window. Table 1 summarizes the actions which operate on the selected entities.

*Picked Group*

There is a special group whose contents can be altered using picking. This group is named picked , and is automatically created by CUBIT. Other than its relationship to interactive picking, it is identical to other groups and can be operated on from the command line. Like other groups, both geometric and mesh entities can be held in the picked group. Table 1 lists the graphics window key presses used with the picked group.

**Note: It is important to distinguish between the current selection and the picked group contents. Clicking on a new entity will select that entity, but will not add it to the picked group. De-selecting an entity will not remove an entity from the picked group.**

*Substituting Selection into Other Commands*

There are three ways to use mouse-based selection to specify entities in commands.

**1. The Selection Keyword**

You may refer to all currently selected entities by using the word selection in a command; the picked type and ID numbers of all selected entities will be substituted directly for selection . For example, if Volume 1 and Curve 5 are currently selected, typing

```
Color selection Blue
```

is identical to typing

```
Color Volume 1 Curve 5 Blue
```

Note that the selection keyword is case sensitive, and must be entered as all lowercase letters.

**2. Echoing the ID of the Selection**

Typing an e into a graphics window will cause the ID of each selected entity to be added to the command line at the current insertion point. This is a convenient way to use entities of which you don't already know the name or ID.

As an added convenience, the picking type can be set based on the last word on the command line using the ` key. Note that this is not the apostrophe key, but rather the left tick mark, usually found at the upper-left corner of the keyboard on the same key as the tilde (~). For example, a convenient way to set the meshing scheme of a cylinder to sweep would be as follows:

```
Volume (hit `, select cylinder, hit e) Scheme Sweep Source Surface (hit `,
select endcap, hit e) Target (select other endcap, hit e)
```

The result will be something similar to

**Volume 1 Scheme Sweep Source Surface 1 Target 2**

Notice that you must use the word Surface in the command, or ` will not select the correct picking type.

**3. Using the Picked Group in Commands**

Like other groups, the picked group may be used in commands by referring to it by name. The name of the picked group is picked. For example, if the contents of the picked group are Volume 1 and Volume 2, the command

**Draw picked**

is identical to

**Draw Volume 1 Volume 2**

Note that picked is case sensitive, and must be entered as all lowercase letters.

*Select Commands*

**Creating and Modifying Selections**

The following commands may be used to create a new selection or modify the current seelction.

**Select <entity_list> [add|remove]**

This command selects the specified entities. If the **add** option is specified, the entities are added to the current selection. If the **remove** option is specified, the entities are removed from the current selection. If neither is specified, the current selection is replaced with the specified entities.

**Select None**

This command clears the selection.

**Select Seed {face <ids>|tri <ids>} feature_angle <angle>**

This command creates a new selection based on a seed face and feature angle. It finds all the neighboring faces with a feature angle that is less than the specified angle and adds them to the selection.

**Rubberband Selection Control**

The following commands control the behavior of rubberband selection.

**Select Occluded {on|off}**

When turned on, the selection will include entities that are occluded, or hidden behind other entities, in the current graphics view.

**Select Partial {on|off}**

When turned on, the selection will include all entities that touch the rubberband. When turned off, the selection will include only entities that lie completely within the rubberband region.

**Select Rubberband Shape {box|polygon|circle}**

Choose the rubberband shape to be box, polygon, or circle. If **polygon** is selected, the shape of the polygon is defined by the left mouse button clicks in the graphics windows. To end defining the polygon shape and make the selection, click the right mouse button.

## Part Classification with Machine Learning

Predicting Part Classification
Categorizing Parts into New Categories
Training
Listing Categories
Resetting Categories
User Training Data
Reclassification
Volume Selection by Category

In a complex assembly containing tens or hundreds of volumes and surfaces, it is sometimes useful to classify or identify entities according to a predefined category. Cubit uses machine learning methods to categorize volumes or surfaces into one of a predefined set of categories. Custom categories, defined by the user, can also be set up used for classification. The **classify** command is primarily used to identify and control part classification using the Cubit command line. In addition, the machine learning tools encapsulated in the geometry power tool in Cubit's graphical user interface provides an extensive set of tools for classifying and applying geometric solutions.

**Syntax:**

**classify {volume|surface <ids>} [confidence] [features [importance]]**
**classify {volume|surface <ids>} "<string>" [export_acis]**
**classify train**
**classify list**
**classify reset ["<string>"]**

# classify user path ["<string>"]
# classify aggregate [model "<string>" category "<string>"]
# reclassify {volume <ids>} "<string>"

#### Predefined Fixed Part Classification Categories

Cubit currently provides the following set of predefined categories for classification:
**Volumes**

1. bolt
2. spring
3. washer
4. nut
5. insert
6. pin
7. ball
8. race
9. gear
10. thin
11. other

**Surfaces**

1. slot
2. non_slot

*Discussion*

Part and Surface classification uses machine learning methods to identify entities based on characteristic geometric features of the volume or surface. These methods utilize a fixed set of training data which has been provided with your Cubit installation. This training data may be augmented by the user to refine the existing categories or add new categories. When using any of the above **classify** commands for the first time, a short pause may occur while the fixed training data from the Cubit installation and the user training data are loaded into the program.

#### Predicting Part and Surface Classification

The command **classify {volume|surface <ids>}** is used to identify a volume or surface based upon the set of categories from both fixed and user training data. When executed, various geometric features of the specified entity(s) are computed and a prediction made as to the most likely categorization based upon the existing training data. Depending upon

the number of entities to classify and the complexity of the surfaces or volumes, this command may take a few seconds to complete. The result will be a string printed to the output window of the predicted category, such as:

```
Volume 1 (Solid) is "spring"
```

or

```
Surface 1 is "slot"
```

The optional **confidence** argument may be used to list the confidence values for each of the existing categories for both fixed and user defined. The confidence is a numerical value from 0 to 1 computed by the machine learning method indicating the confidence level of the classification for each category. The highest numerical value of confidence is selected as the predicted category.

The **features** and **importance** arguments are primarily used for diagnostics to examine the resulting geometric features and their relative significance to the prediction. The features are a list of about 50 scalar values that are computed for each surface or volume that are used in the machine learning methods and the importance is the relative weight given to each feature when making the prediction.

#### Categorizing Entities into New Categories

If the existing set of fixed categories is insufficient, users can create new categories to augment the fixed categories. In addition, if a predicted category for a given surface or volume is incorrect or insufficient, training data for the fixed categories may also be added. To add training data to a category, use the **classify {volume|surface <ids>} "<string>"** command, where <string> is either an existing category name or a new category. Use quotation marks to distinguish the category name. For example:

```
classify volume 1 "spring"
```

When executed, the features of the given surface or volume will be computed and written to disk as a new set of training data. User training data defined in this way is written to an application directory which is persistent so it can be used in subsequent runs of Cubit.

In most cases the more examples of a category that can be provided, the more accurate will be subsequent predictions. For example, if a single volume of new category **"widget"** is added, all volumes that are identical to the initial example will most likely be predicted as **"widget"**, however small variations of the volume may not. For this reason, providing as many varying examples of a **"widget"** as possible is advantageous when setting up a new category. Providing identical examples of the same volume to the **classify** command is also not detrimental, as duplicates are automatically identified and filtered out. Note that when categorizing a new surface or volume, the output window will indicate whether new training data was added or whether the volume represents duplicate data.

The **export_acis** option can be used to automatically write an acis **.sat** file of the specified volume(s). If used, the file(s) will be written to the user training directory titled with the name of the category. Each acis file will contain a single volume and named using the category name and a unique incremented integer ID. While primarily used for debugging,

they can also be used as a visual representation of the current user categories. (Not currently supported for surfaces)

#### Training

Whenever new training data is added, the classification models must be retrained. Although retraining happens automatically any time training data is added or deleted when ussing Cubit commands, the **train** command is useful for forcing a retrain should additional training data be added manually to the training directories on disk. In addition to rerunning the training process, it will print to the output window the number of supporting volumes used for each category when computing classification predictions.

#### Listing Categories

All existing classification categories may be printed to the output window using the **classify list** command. For user defined categories or fixed categories that have been augmented, a **(U)** will follow the name of the category.
In addition to the category names, the **classify list** command will also print the path to both the fixed training data and the user training data on disk.

#### Resetting Categories

To remove an existing user defined category use the command **classify reset ["<string>"]** where **<string>** is a a category name in quotations. This will remove all user training data for the given category from disk and retrain the classification models without the category. If a fixed category is specified, the user-defined training data defined in the category will be removed, keeping only the fixed data. If this command is used without a category, all user training data will be removed.

#### User Training Data

When a new category is created or a fixed category is added to, the resulting training data is written to a default application directory that is specific to a platform. For example, for Mac and Linux OS the directory will be located at:

**/Users/<user_name>/Library/Application Support/Cubit/ml**

To display both the current user training data directory and the fixed training data directory, use the **classify list** command. While the fixed training data directory canot be changed, it may be worthwhile to change the user training directory. To change the user training directory, use the command **classify user path "<path string>"** where **<path string>** is the full path to a writable directory on disk in quotes. Changing the user training directory may be useful to temporarily use training data from another source or to ignore all user training data without removing it.

Using the command, **classify user path** without a path specification will set the user training data directory back to its default for the platform.

#### Aggregate Classification Data

To add (aggregate) user-defined classification training data to Cubit's existing data, use the **Classify Aggregate** command. If a specific model and category are not specified, all user-defined data will be aggregated.

#### Reclassification

At times, it may be necessary to remove a surface or volume from its current category classification, or move the surface or volume from one category to another. Use the **reclassify {volume|surface <ids>} "<string>"** command to update the training data for one or more volumes where **<string>** is the name of a category. If the training data for the given volume is currently present in another category, it will be removed from its current category and added to the new specified category. If no category is specified, it will be removed from its current category without adding it to another category. Training data from the fixed training data cannot be removed or modified.

#### Surface or Volume Selection by Category

It may be useful to identify all surfaces or volumes based on their predicted classification category. To do so, use the syntax **with category "<string>"** in conjunction with other Cubit commands that accept IDs. A few example uses this syntax might include:

```
group "mysprings" add volume with category "spring"
draw volume with category "bolt"
delete volume with category "insert"
```

In the first example, a group named **mysprings** is created and all volumes with the predicted category of **spring** added to it. The second example would draw all volumes that are classified as **bolt**, and the third example would delete all volumes that are classified as **insert**.

Note that in order to identify surfaces or volumes based on their predicted category, features for all surfaces or volumes in the model will be computed. For assemblies with hundreds or thousands of parts, this may be time consuming. Also note that a similar capability is available in the geometry power tool machine learning tools which will list the predicted category for each volume in the model in separate drop-down lists and build cubit groups from each category if requested.

## Training CAD Operations with Machine Learning

Regression Models
Predicting CAD Operation Outcomes
Assigning new CAD Operation Training Data
Training CAD Operations
Listing Regression Models
Resetting Regression Models
User Training Data

**Syntax:**

```
predict copy surface {volume <ids> surface <ids>} [features [importance]]
predict midsurface {volume <ids> surface <ids>} [features [importance]]
learn copy surface {volume <ids> surface <ids>} label <value> [export_acis]
learn midsurface {volume <ids> surface <ids>} label <ids> [export_acis]
learn train [<string>]
learn reset ["<string>"]
learn user path ["<string>"]
```

*Regression Models*

Some of the ML models used in Cubit are used for recommending CAD operations that can be used to resolve a particular geometry or topology issue, known as *regression* models. They differ from classification ML models described, in that they predict a specific numerical outcome for a given operation. Cubit currently supports two types of regression models based on the expected output or ground truth label: *Mesh Quality* and *Suitability*.

**Mesh Quality**

These models predict a mesh quality outcome, including scaled Jacobian, scaled In-radius and scaled deviation. The intent of the mesh quality models is to recommend a solution from a range of different CAD operations based on the predicted best mesh quality outcome. Each of the mesh quality regression models have a corresponding Cubit command that they are used for. The currently supported mesh quality ML models include the following:

1. vertex_no_op
2. curve_no_op
3. surface_no_op
4. remove_surface
5. tweak_replace_surface
6. composite_surfaces
7. collapse_curve
8. remove_topology_curve
9. virtual_collapse_curve
10. remove_topology_surface
11. blunt_tangency
12. remove_cone
13. collapse_angle
14. remove_blend
15. remove_cavity

Mesh quality models are mostly used for driving defeaturing of CAD models. They are currently used in Cubit's Geometry Power Tool diagnostics to predict where potential mesh quality issues will appear on the FE mesh before meshing and to provide a recommendation for CAD operations that can be used to resolve the issue based on a predicted mesh quality outcome.

#### Suitability

The *suitability* models will predict a heuristic value from 0 to 1 based on how well the outcome is expected to be successful within the context of a given assembly. Suitability models include the following:

1. copy_surface
2. midsurface

The *suitability* models are currently limited to making recommendations for the corresponding **reduce thin volume** commands. They are also used as the primary user knowledge base for Cubit's reinforcement learning procedure for reducing thin volumes. The following sections outline how the regression models can be managed using Cubit's command line tools. At present, only the suitability models are supported for predicting and training user training data.

*Predicting CAD Operation Outcomes*

A suitability score will be displayed at the command line with the following commands:

# Learn copy_surface volume <id> surface <ids>

# Learn midsurface volume <id> surface <ids>

For these commands, the **copy_surface** and **midsurface** ML regression models will be used to predict a suitability score. These reflect the suitability of using the respective reduce volume thin commands in the current context. The volume ids and surface ids used in the **Learn** commands should be the same as those used in the equivalent reduce thin commands.

*Adding New CAD Operation Training Data*

Adding additional user training data is the same as prediction (above) except that a **label** value is provided, where a value of zero represents low or infeasible condition and 1.0 is high or most preferred solution. In this case, features and labels are appended to the existing user training data so that subsequent ML predictions will use this data to determine suitability outcomes.

# Learn copy_surface volume <id> surface <ids> label <value>

# Learn midsurface volume <id> surface <ids> label <value>

*Training CAD Operations*

A training operation is automatically triggered anytime new training data is added. Training can also be invoked from a command by using the following command.

# Learn Train ["<string>"]

The **<string>** value should be either one of **copy_surface** or **midsurface**. Mesh Quality models are currently not supported. If the **<string>** is absent, then all available regression models will be trained.

*Listing Regression Models*

To list all available regression models use the following command:

# Learn List ["<string>"]

If the **<string>** is absent, then all available regression models will be listed.

*Resetting Regression Models*

To remove all use training data for a given regression model, the following command can be used:

# Learn Reset ["<string>"]

If the **<string>** is absent, then all available regression models will be reset.

*User Training Data*

# Learn User Path ["<path_string>"]

When a new training datum is written, it will write to a default application directory that is specific to a platform. For example, for Mac and Linux OS the directory will be located at:

```
/Users/<user_name>/Library/Application Support/Cubit/ml
```

To display both the current user training data directory and the fixed training data directory, use the **Learn List** command. While the fixed training data directory cannot be changed, it may be worthwhile to change the user training directory. To change the user training directory, use the command **Learn user path "<path_string>"** where **<path_string>**> is the full path to a writable directory on disk in quotes. Changing the user training directory may be useful to temporarily use training data from another source or to ignore all user training data without removing it.

Using the command, **Learn User Path** without a path specification will set the user training data directory back to its default for the platform. Changing the user path using the **Learn** command will also change the path for the **Classify** command.

# Geometry

## Geometry

- CUBIT Geometry Formats
- Geometry Creation
- Geometry Transforms
- Geometry Booleans
- Geometry Decomposition
- Geometry Cleanup and Defeaturing
- Geometry Imprinting and Merging
- Virtual Geometry
- Geometry Orientation
- Geometry Groups
- Geometry Attributes
- Entity Measurement
- Parts, Assemblies, and Metadata
- Geometry Deletion

CUBIT usually relies on the ACIS solid modeling kernel for geometry representation; there is also mesh-based geometry. Other solid model kernels are planned. Geometry is imported or created within CUBIT. Geometry is created bottom-up or through primitives. CUBIT imports ACIS SAT files. CUBIT can also read STEP, IGES, and FASTQ files and convert them to the ACIS kernel. SolidWorks, AutoCAD, and some other commercial CAD systems can write SAT files directly.
Once in CUBIT, an ACIS model is modified through booleans. Without changing the geometric definition of the model, the topology of the model may be changed using virtual geometry. For example, virtual geometry can be used to composite two surfaces together, erasing the curve dividing them.
Sometimes, an ACIS model is poorly defined. This often happens with translated models. The model can be healed inside CUBIT.

## model definitions

### CUBIT Geometry Formats

- ACIS
- Mesh-Based Geometry

*Setting the Geometry Kernel*

The geometry kernel can be switched between ACIS and Mesh-Based Geometry from the command line using the following command:

**Set Geometry Engine {Acis|Facet}**

The geometry engine will automatically be set when importing a model.

*Terms*

Before describing the functionality in CUBIT for viewing and modifying solid geometry, it is useful to give a precise definition of terms used to describe geometry in CUBIT. In this manual, the terms topology and geometry are both used to describe parts of the geometric model. The definitions of these terms are:
**Topology**: the manner in which geometric entities are connected within a solid model; topological entities in CUBIT include vertices, curves, surfaces, volumes and bodies.
**Geometry**: the definition of where a topological entity lies in space. For example, a curve may be represented by a straight line, a quadratic curve, or a b-spline. Thus, an element of topology (vertex, curve, etc.) can have one of several different geometric representations.

*Topology*

Within CUBIT, the topological entities consist of vertices, curves, surfaces, volumes, and bodies. Each topological entity has a corresponding dimension, representing the number of free parameters required to define that piece of topology. Each topological entity is bounded by one or more topological entities of lower dimension. For example, a surface is bounded by one or more curves, each of which is bounded by one or two vertices.

#### Bodies and Volumes

A CUBIT Body is defined as a collection of other pieces of topology, including curves, surfaces and volumes. The use of Body is not required, and is in fact deprecated in favor of using Volume. Bodies may still be used for grouping volumes, but it is suggested to use Groups instead.
Although a Body may contain groups of Surfaces or Volumes, for most practical purposes within the CUBIT environment, a single Volume or Surface will belong to a single Body. For typical three-dimensional models, this means that there should be one Body for every Volume in the model, where the default Body ID is the same as the Volume ID. For this reason, in many instances the term Volume and Body are used interchangeably, although it is more consistent to always refer to Volumes and Volume IDs, and only use Bodies when absolutely necessary.

#### Non-Manifold Topology

In many applications, the geometry consists of an assembly of individual parts, which together represent a functioning component. These parts often have mating surfaces, and for typical analyses these surfaces should be joined into a single surface. This results in a mesh on that surface which is shared by the volume meshes on either side of the shared surface. This configuration of geometry is loosely referred to as **non-manifold topology**.

*Bounding Box Calculations*

Bounding box calculations are used for many routines and subroutines in Cubit. These calculations are done using a faceted representation by default. To use the default modeling engine for more accurate (and longer) calculations change the **Facet Bbox** setting.

**Set Facet BBox [ON|Off]**

There are also various settings to control the accuracy of bounding box calculations based on point lists.

**Set Tight [[Bounding] [Box] [{Surface|Curve|Vertex} {on|off}]]**

If surfaces are used, surface facet points will be included in the point list used to calculate the tight bounding box. This will include vertices and points on the curves. This is the default implementation.
If curves are used, curve tesselation points will be included in the point list used to calculate the tight bounding box. This includes the vertices on the ends of the curves. One use for this is to find a more accurate tight bounding box, since curve tessellations are typically more fine than surface tessellations. However, in practice, it is recommended to just use surface tessellations. One special case is if the user sends in a list of curves as the criteria for the tight bounding box, the curve tessellations are always used, even if this parameter is false.
If vertices are used, vertex points will be included in the point list used to calculate the tight bounding box. In extremely large models, it could be advantageous to just use vertices. So the user would turn off both the surface and curve flags. One special case is if the user sends in a list of curves as the criteria for the tight bounding box, the curve tessellations are always used, even if the curve parameter is false and this parameter is true.

## ACIS Geometry Kernel

ACIS is a proprietary format developed by Spatial Technologies. CUBIT incorporates the ACIS third party libraries directly within the program. The ACIS third party libraries are used extensively within CUBIT to import, export and maintain the underlying geometric representations of the solid model for geometry decomposition and meshing. There are many ways to get geometry into the ACIS format. ACIS files can be exported directly from several commercial CAD packages, including SolidWorks, AutoCAD, and HP PE/SolidDesigner. Third party ACIS translators are also available for converting from native formats such as Pro Engineer. CUBIT also uses the ACIS libraries for importing IGES and STEP format files.
Importing and creating geometry using the ACIS geometric modeling kernel currently provides the widest set of capabilities within CUBIT. All geometry creation and modification tools have been designed to work directly on the ACIS representation of the model.

## Mesh-Based Geometry

In contrast to the ACIS format, Mesh-Based Geometry (MBG) is not a third party library and has been developed specifically for use with CUBIT. Most of CUBIT's mesh generation tools require an underlying geometric representation. In many cases, only the finite element model is available. If this is the case, CUBIT provides the capability to import the finite element mesh and build a complete boundary representation solid model from the mesh. The solid model can then be used to make further enhancement to the mesh. While the underlying ACIS geometry representation is typically non-uniform rational b-splines (NURBS), Mesh-Based Geometry uses a *facetted* representation. Mesh-Based Geometry can be generated by importing either an Exodus II format file or a facet file.

- Creating Mesh-Based Geometry Models
- Improving Mesh-Based Geometry Models for Meshing
- Meshing Mesh-Based Models
- Exporting Mesh-Based Geometry

Many of the same operations that can be done with traditional CAD geometry can also be done with mesh-based geometry. While all mesh generation operations are available, only some of the geometry operations can be used. For example, the following can be done with geometric entities that are mesh-based:

- Geometry Transformations
- Merging
- Virtual Geometry Operations

Some operations that are not yet available with mesh-based geometry include:

- Booleans
- Geometry Decomposition
- Geometry Clean-Up

*Creating Mesh-Based Geometry Models*

Mesh based geometry models can be created in one of two ways

- Importing Exodus II files
- Importing facet files

While both of these methods create geometry suitable for meshing, there are some significant differences:

**Exodus II files**

Exodus II contains a mesh representation that may include 3D elements, 2D elements, 1D elements and even 0D elements. It may also contain deformation information as well as boundary condition information. The import mesh geometry command is designed to decipher this information and create a complete solid model, using the mesh faces as the basis for the surface representations. Exodus II is most often used when a solid model

that has previously been meshed requires modification or remeshing. Importing an Exodus II file will generate both geometry and mesh entities, assigning appropriate ownership of the mesh entities to their geometry owners. Deleting the mesh and remeshing, refining or smoothing are common operations performed with an Exodus II model.

**Facet files**

The facet file formats supported by CUBIT are most often generated from processes such as medical imaging, geotechnical data, graphics facets, or any process that might generate discrete data. Importing a facet file will generate a surface representation only defined by triangles. If the triangles in the facet file form a complete closed volume, then a volume suitable for meshing may be generated. In cases where the volume may not completely close or may not be of sufficient quality, a limited set of tools has been provided. In addition to the standard meshing tools provided in CUBIT, it is also possible to use the triangle facets themselves as the basis for an FEA mesh.

*Improving Mesh-Based Geometry Models for Meshing*

In many cases, the triangulated representations that are provided from typical imaging processes are not of sufficient quality to use as geometry representations for mesh generation. As a result, CUBIT provides a limited number of tools to assist in cleaning up or repairing triangulated representations.

**1. Using tolerance on STL files**

Stereolithography (STL) files, in particular, can be problematic. The import mechanism for STL provides a **tolerance** option to merge near-coincident vertices.

**2. Using the stitch option on AVS and facet files**

The stitch option on the import facets|avs command provides a way to join triangles that otherwise share near-coincident vertices and edges. This is useful for combining facet-based surfaces to generate a water-tight model.

**3. Using the improve option on facet files.**

The improve option on the **import facets** command will collapse short edges on the boundary of the triangulation. This option improves the quality of the boundary triangles.

**4. Smoothing faceted surfaces.**

Individual triangles in a faceted surface representation may be poorly shaped. Just like mesh elements may be smoothed, facets may also be smoothed in CUBIT using the following command

**Smooth <surface_list> Facets [Iterations <value>] [Free] [Swap]**

To use this command, the surface cannot be meshed. Facet smoothing consists of a simple Laplacian smoothing algorithm which has additional logic to make sure it does not turn any of the triangles in-side out. It also determines a local surface tangent plane and projects the triangle vertices to this plane to ensure the volume will not "shrink". The **iterations** option can be used to specify the number of Laplacian smoothing operations to perform on each facet vertex (The default is 1).

The **free** option can be used to ignore the tangent plane projection. Used too much, the **free** option can collapse the model to a point. One of two iterations of this option may be enough to clean up the triangles enough to be used for a finite element mesh.
The **swap** option can be used to perform local edge swap operations on the triangulation. The quality of each triangle is assessed and edges are swapped if the minimum quality of the triangles will improve.

**5. Creating a thin offset volume**

Offset surfaces may be generated from an existing facet-based surface. This would be used in cases where a thin membrane-like volume might be required where only a single surface of triangles is provided. This command may be accomplished by using the standard **create body offset** command
The result of this command is a single body with an inside and outside surface separated by a small distance which is generally suitable for tet meshing. This command is currently only useful for small offsets where self-intersections of the resulting surface would be minimal. It is most useful for bodies that may be initially composed of a single water-tight surface.

**6. Creating volumes from surfaces**

A mesh-based geometry volume can be created from a set of closed surfaces. This can be accomplished in the same manner as the standard create body surface command

**Create Body Surface <surface_id_range>**

This command is limited to surfaces that match triangles edges and vertices at their boundary. The command will internally merge the triangles to create a water-tight model that would generally be suitable for tet meshing.

*Meshing Mesh-Based Models*

Mesh-Based models may be meshed just like any other geometry in CUBIT by first setting a scheme, defining a size and using the mesh command. This standard method of mesh generation can be somewhat time consuming and error prone for complex facet models with thousands of triangles. CUBIT also provides the option of using the facets themselves as a surface triangle mesh, or as the input to a tetrahedral mesher. This may be accomplished with one of two options:

**Mesh <entity_list> From Facets**

This command will generate triangular finite elements for each facet on the surface. If the **entity_list** is composed of one or more volumes, then the tetrahedral mesh will automatically fill the interior. This method is useful when further cleanup and smoothing operations are needed on the triangles after import.

**Import Facets <filename> Make_elements**

The make_elements on the import facets command will generate the triangular finite elements on the surface at the time the facets are read and created. This option is useful if no further modifications to the facets are necessary.
Creating triangular finite elements in this manner can greatly speed up the mesh generation process, however it is limited to non-manifold topology. If the triangular elements are to be used for tetrahedral meshing (i.e. all edges of the triangulation should be connected to no more than two triangles)

*Exporting Mesh-Based Geometry*

Mesh-Based geometry models and their mesh may be exported by one of the following methods:

- Exporting to an Exodus II File
- Exporting to a facet file

**Exodus II**
Exporting to an Exodus II file saves the finite element mesh along with any boundary conditions placed on the model. It will not save the individual facets that comprise the mesh-based geometry surface representation. Importing an Exodus II file saved in this manner will regenerate the surfaces only to the resolution of the saved mesh.
**Facet files**
CUBIT also provides the option to save just the surface representation to a facet or STL file. The following commands can be used for saving facet or STL files:

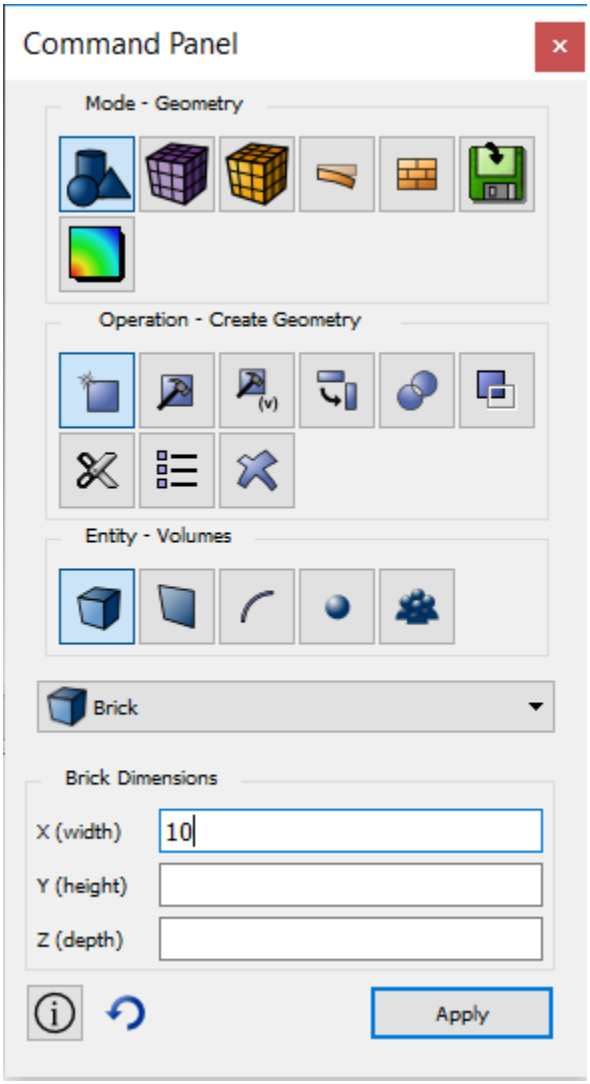
```
Export Facets 'filename' <entity_list> [Overwrite]

Export STL [ASCII|Binary] 'filename' <entity_list> [Overwrite]
```

These commands provide the option of saving specific surfaces or volumes to the facet file. If no entities are provided in the command, then all surfaces in the model will be exported to the file. The **overwrite** option forces a file to overwrite any file of the same name in the current working directory.

# geom creation

## Geometry Creation

There are three primary ways of creating geometry for meshing in CUBIT. First, CUBIT provides many geometry primitives for creating common shapes (spheres, bricks, etc.) which can then be modified and combined to build complex models. Secondly, geometry can be imported into CUBIT. Finally, geometry can be defined by building it from the "bottom up", creating vertices, then curves from those vertices, etc. Two of these three methods for creating geometry in CUBIT will be described in detail in this section.
All of these geometry creation commands have been expressed in the GUI's command panels. To navigate to the volume creation command panels, for example, select "Mode-Geometry", then "Entity-Volume", then "Action-Create", as shown below. Other geometry creation command panels are available for each geometry type.




- Bottom-Up Geometry Creation
- Geometric Primitives

primitive geometry

# *Creating Bricks*

The brick is a rectangular parallelepiped.
**Command**

```
[Create] Brick {Width|X} <width> [{Depth|Y} <depth>] [{Height|Z} <height>]
[Bounding Box {entity_type} <id_range>] [Tight] [[Extended] {Percentage|
Absolute} <val>]]
```

**Notes**

- A cubical brick is created by specifying only the width or x dimension.
- A brick can be specified to occupy the bounding box of one or more entities, specified on the command line.
- If the **Tight** option is specified with **Bounding Box**, the result is the smallest brick that can contain the entities specified, which is the default behavior of the Bounding Box option.
- If the **Extended** option is specified with **Bounding Box**, the result is a brick that is extended from a "tight" brick by the input percentage or absolute value.
- If a bounding box specification is used in conjunction with any of the other parameters (X, Y or Z), the parameters specified override the bounding box results for that or those dimensions.

# *Creating Pyramids*

A pyramid is a general n-sided prism.
**Command**

**[Create] Pyramid [Height|Z] <z-height> Sides <nsides> Radius <radius> [Top <top-x-radius>]**

**[Create] Pyramid [Height|Z] <z-height> Sides <nsides> [Major [Radius] <x-radius> Minor [Radius] <y-radius> ] [Top <top-x-radius>]**

# *Creating Frustums*

A frustum is a general elliptical right frustum, which can also be thought of as a portion of a right elliptical cone.
**Command**

**[Create] Frustum [Height|Z] <z-height> Radius <x-radius> [Top <top_radius>]**

**[Create] Frustum [Height|Z] <z-height> Major Radius <radius> Minor Radius <radius> [Top <top_radius>]**

**[Create] Frustum Volume <id> [Axis <options>]**

**Notes**

- If used, Major Radius defines the x-radius and Minor Radius the y-radius.
- If used, Top Radius defines the x-radius at the top of the frustum; the top y radius is calculated based on the ratio of the major and minor radii.
- A frustum can also be created to bound a volume. An optional axis for the frustum can be specified. If not specified, the axis will be determined from a tight bounding box.

# *Creating Toruses*

The torus command generates a simple torus
**Command**

**[Create] Torus Major [Radius] <major-radius> Minor [Radius] <minor-radius>**

**Notes**

- **Minor Radius** is the radius of the cross-section of the torus; **Major Radius** is the radius of the spine of the torus.
- The **minor radius** must be less than the **major radius**.

# *Creating Cylinders*

The cylinder is a constant radius tube with right circular ends.
**Command**

**[Create] Cylinder [Height|Z] <val> Radius <val>**

**[Create] Cylinder [Height|Z] <val> Major Radius <val> Minor Radius <val>**

**[Create] Cylinder Volume <id> [Axis <options>]**

**Notes**

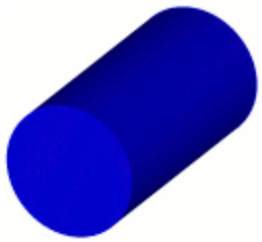
- A cylinder may also be created using the frustum command with all radii set to the same value.
- Specifying major and minor radii can produce a cylinder with an oval cross section.

- A cylinder that tighty bounds a volume can also be created. An optional axis for the cylinder can be specified. If not specified, the axis will be determined from a tight bounding box.

# *Geometric Primitives*

The geometric primitives supported within CUBIT are pre-defined templates of three-dimensional geometric shapes. Users can create specific instances of these shapes by providing values to the parameters associated with the chosen primitive. Primitives available in CUBIT include the brick, cylinder, torus, prism, frustum, pyramid, and sphere. Each primitive, along with the command used to generate it and the parameters associated with it, are described next. For some primitives, several options can be used to generate them, and are described as well.
The following Primitives can be generated with CUBIT:

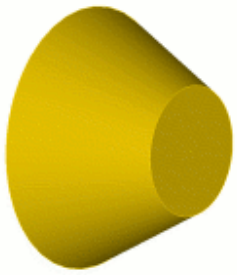
Brick



Cylinder



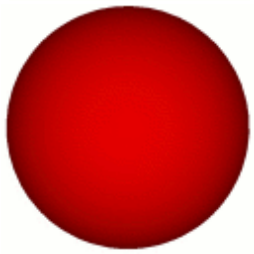
Prism



Frustum



Pyramid



Sphere



Torus

**General Notes**

- Primitives are created and given an ID equal to one plus the current highest body ID in the model.
- Primitive solids are created with their centroid at the origin or the world coordinate system.
- For primitives with a Height or Z parameter, the axis going through these primitives will be aligned with the Z axis.
- For primitives with a Major Radius and a Minor Radius, the Major Radius will be along the X axis, the Minor Radius along the Y axis.
- For primitives with a Top Radius, this radius will be that along the X axis; the Y axis radius will be computed using the Major, Minor and Top Radii given.

# *Creating Prisms*

The prism is an n-sided, constant radius tube with n-sided planar faces on the ends of the tube.

**Command**

**[Create] Prism [Height|Z] <z-val> Sides <nsides> Radius <radius>**

**Notes**

- The radius defines the circumradius of the n-sided polygon on the end caps.
- If a major and minor radius are used, the end caps are bounded by a circum-ellipse instead of a circumcircle.
- The number of sides of a prism must be greater than or equal to three. A prism may also be created using the pyramid command with all radii set to the same value.
- If the **Extended** option is specified with **Bounding Box**, the result is a brick that is extended from a "tight" brick by the input percentage or absolute value.
- If a bounding box specification is used in conjunction with any of the other parameters (X, Y or Z), the parameters specified override the bounding box results for that or those dimensions.

# *Creating Spheres*

The sphere command generates a simple sphere, or, optionally, a portion of a sphere or an annular sphere.
**Command**

**[Create] Sphere Radius <radius> [Xpositive]|[Xnegative] [Ypositive]|[Ynegative] [Zpositive]|[Znegative] [Delete] [Inner [Radius] <radius>]**

**Notes**

- If Xpositive/Xnegative, Ypositive/Ynegative, and/or Zpositive/Znegative are used, a sphere which occupies that side of the coordinate plane only is generated, or, if the delete keyword is used, the sphere will occupy the other side of the coordinate plane(s) specified. These options are used to generate hemisphere, quarter sphere or a sphere octant (eighth sphere).
- If the inner radius is specified, a hollow sphere will be created with a void whose radius is the specified inner radius.

## bottom up creation

*Bottom-Up Geometry Creation*

CUBIT supports the ability to create geometry from a collection of lower order entities. This is accomplished by first creating vertices, connecting vertices with curves and connecting curves into surfaces. Currently only ACIS bodies or volumes may not be constructed by stitching a set of surfaces together, and only in a certain number of cases; however surfaces may also be swept or rotated to create bodies or volumes. Existing geometry may be combined with new geometry to create higher order entities. For example, a new surface can be created using a combination of new curves and curves

already extant in the model. Commands and details for creating each type of geometry entity are given below.
The following describes each of the basic entities that can be generated with CUBIT using the bottom-up approach

- Creating Vertices
- Creating Curves
- Creating Surfaces
- Creating Bodies

*Creating Volumes*

Currently, CUBIT can create volumes:

1. from surfaces by sweeping a single surface into a 3D solid,
2. by offsetting an existing volume,
3. by extending one or more surfaces or sheet bodies
4. by sweeping a curve around an axis,
5. by stitching together surfaces that can form a closed volume,
6. by lofting from one surface to another surface, or
7. by thickening a surface body.

Sweeping of planar surfaces, belonging either to two- or three-dimensional bodies, is allowed, and some non-planar faces can be swept successfully, although not all are supported at this time. The following methods for generating volumes are described:

- Sweep Surface Along Vector
- Sweep Surface About Axis
- Sweep Surface Along Curve
- Sweep Surface Perpendicular
- Sweep Surface to a Volume
- Offset
- Sheet extended from surface
- Sweep Curve About Axis
- Stitch Surfaces Together
- Loft Surfaces Together
- Thicken Surfaces
- Sweep Surface
- Sweep Surface along Direction
- Sweep Surface along Helix

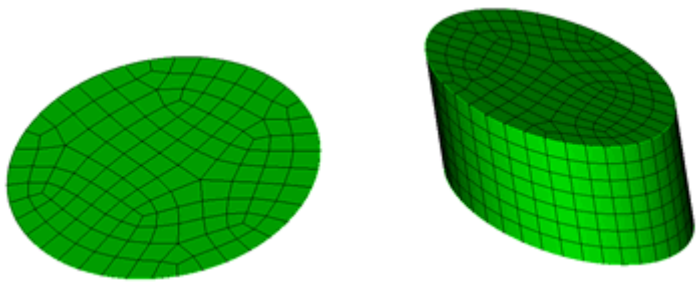
There are five forms of the sweep command; the syntax and details for each are given below. Common options for first four forms are:

**draft_angle**: This parameter specifies the angle at which the lateral faces of the swept solid will be inclined to the sweep direction. It can also be described as the angle at which the profile expands or contracts as it is swept. The default value is 0.0.
**draft_type**: This parameter is an ACIS-related parameter and specifies what should be done to the corners of the swept solid when a non-zero draft angle is specified. A value of 0 is the default value and implies an extended treatment of the corners. A value of 1 is also valid and implies a rounded (blended) treatment of the corners.
**anchor_entity**: The default behavior for the sweep command is to move the source surface along a path to create a new 3D solid. The anchor_entity option instructs the sweep to leave the source surface in its original location.
**include_mesh**: This option will sweep the source surface and existing mesh into a meshed 3D solid. The mesh size is automatically computed using the Default auto interval specification.

The sweep operations have been designed to produce valid solids of positive volume, even though the underlying solid modeling kernel library that actually executes the operation, ACIS, allows the generation of solids of negative volume (i.e., voids) using a sweep.

**1. Sweep Surface Along Vector:** Sweeps a surface a specified distance along a specified vector. Specifying the distance of the sweep is optional; if this parameter is not provided, the face is swept a distance equal to the length of the specified vector. The include_mesh option will create a volumetric mesh if the surface is already meshed as shown below. The keep option will keep the original surface while creating the volume.

**Sweep Surface {<surface_id_range>} Vector <x_vector y_vector z_vector> [Distance <distance_value>] [switchside] [Draft_angle <degrees>] [Draft_type <0|1>][rigid][anchor_entity][include_mesh] [keep] [merge]**

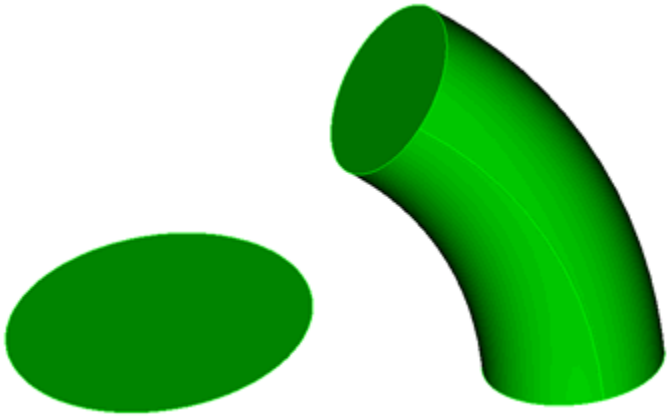


**Surface mesh swept along a vector**

**2. Sweep Surface About Axis:** Sweeps a surface about a specified vector or axis through a specified angle. The axis of revolution is specified using either a starting point and a vector, or by a coordinate axis. This axis must lie in the plane of the surfaces being swept. The steps parameter defaults to a value of 0 which creates a circular sweep path. If a positive, non-zero value (say, n) is specified, then the sweep path consists of a series of n linear segments, each subtending an angle of [( sweep_angle ) / ( steps-1 )] at the axis of revolution. The include_mesh option will create a volumetric mesh if the surface

is already meshed as shown below. The keep option will keep the original surface while creating the volume.

**Sweep Surface {<surface_id_range>} Axis {<xpoint ypoint zpoint xvector yvector zvector>|Xaxis|Yaxis|Zaxis} Angle <degrees> [switchside] [Steps <number_of_sweep_steps>] [Draft_angle <degrees>] [Draft_type <0|1>][rigid][anchor_entity][include_mesh] [keep] [merge]**



**Surface swept around an axis of 50 degree angle**



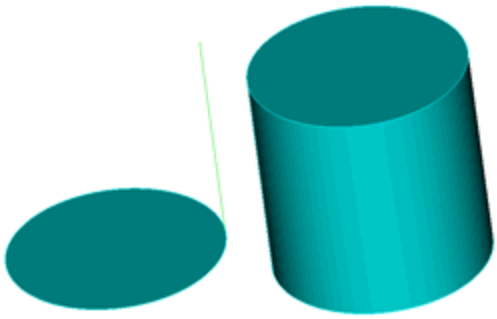
Specifying multiple surfaces that belong to the same body will not work as expected, as ACIS performs the sweep operation in place. Hence, if a range of surfaces is provided, they ought to each belong to different bodies.

**3. Sweep Surface Along Curve:** This command allows the user to sweep a planar surface along a curve:

**Sweep Surface <surface_id_range> Along Curve <curve_id> [Draft_angle <degrees>] [Draft_type <0 | 1 | 2>][rigid][anchor_entity][include_mesh] [keep] [individual] [merge]**

One of the ends of the curve must fall in the plane of the surface and the curve cannot be tangential to the surface. The relationship between the surface orientation and the guide curve is maintained through out the sweep. If the "rigid" option is specified the orientation of the surface is kept static throughout the sweep. Sweep along curve also supports an additional draft type "2" which implies a "natural" extension of the corners from their curves.

The include_mesh option will create a volumetric mesh if the surface is already meshed as shown below. The keep option will keep the original surface while creating the volume. If multiple curves to sweep along are specified, the individual option creates an individual or separate volume along each curve.

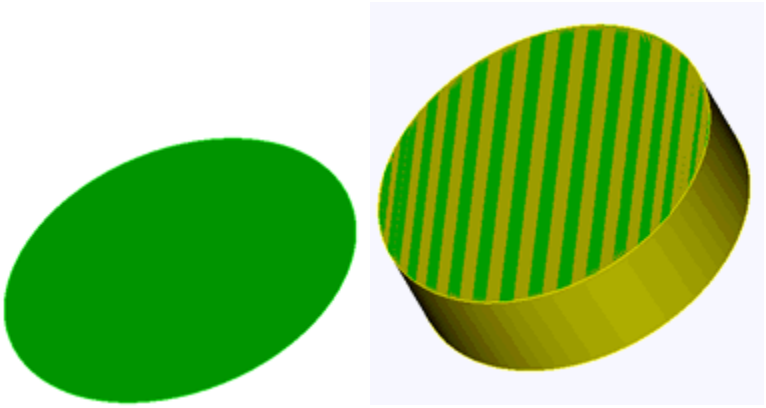


**Volume generated by sweeping a surface along a reference curve**

**4. Sweep Surface Perpendicular:** This command allows the user to sweep a planar surface perpendicular to the surface:

**Sweep Surface <surface_id_range> Perpendicular Distance <distance> [Switchside] [Draft_angle <degrees>] [Draft_type <integer>][anchor_entity][include_mesh] [keep] [merge]**

The sweeping plane must be planar in order to determine the sweep direction. The switchside option will reverse the direction of the sweep.



**The original surface is retained with the 'keep' option. A new volume is created by sweeping the surface along the surface normal.**

The include_mesh option will create a volumetric mesh if the surface is already meshed as shown below. The keep option will keep the original surface while creating the volume.

**5. Sweep Surface to a Volume:** This command allows users to sweep a surface to a volume.

**Sweep Surface <surface_id_range> Target {Volume|Body} <id> [Direction {options}] [Plane {options}]**

The direction keyword can be used to control the direction of sweep. Without it, Cubit will determine the sweep direction (usually normal to the sweeping surface). The plane option can be used to define a stopping plane.

**6. Offset:** The following command creates a body offset from another body or set of surfaces at the specified distance. The new surfaces are extended or trimmed appropriately. A positive distance results in a larger body; a negative distance in a smaller body.

**Create Body Offset [From] Body <id_range> Distance <value>**

**Create Sheet Offset From Surface <id_list> Offset <val> [Surface <id_list> Offset <val>] [Surface <id_list> Offset <val> ...] [Preview]**

Using the second form of the command, the sheet body can be created from a list of surfaces, and the surfaces may offset by different distances. This command currently requires the original surfaces to be on solid bodies.

This option is also available for limited cases for facet-based surfaces.

**7. Sheet Extended from Surface:** The following command creates a body offset from another body or set of surfaces at the specified distance. The new surfaces are extended or trimmed appropriately. A positive distance results in a larger body; a negative distance in a smaller body.

**Create Sheet Extended From Surface <id_list> [Intersecting <entity_list>] [Extended {Percentage|Absolute} <val>] [Preview]**

This command allows multiple surfaces to be extended at the same time. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the size of the surface by either a percentage distance or an absolute distance of the minimum area size. The plane can be previewed with the preview option. Figure 1 shows a set of surfaces being created using the extended absolute option.

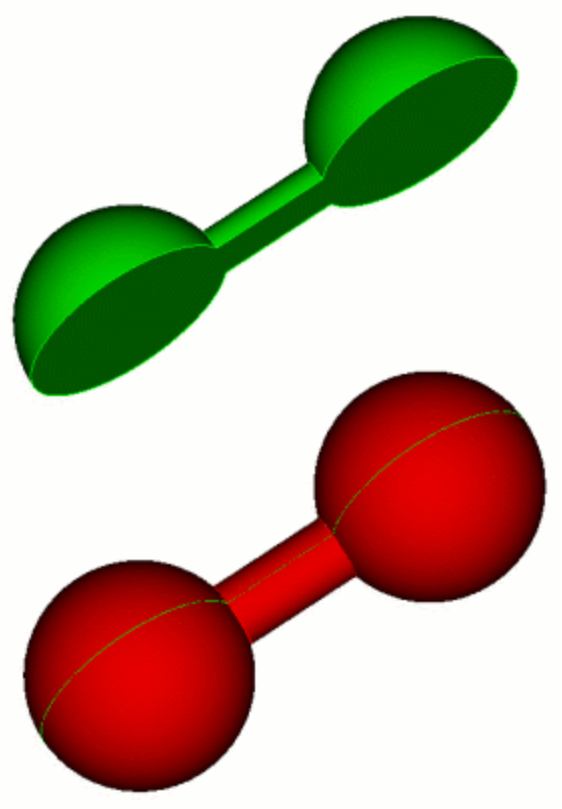


**Figure 1. Sheet created from extending multiple surfaces**

**8. Sweep Curve About Axis:** Sweeps a curve or set of curves about a given axis through a specified angle. The axis is specified the same as in the Sweep Surface About Axis command. The steps, draft_angle, and draft_type options are the same as are described above. To create the solid, the make_solid option must be specified, otherwise a surface will be created, rather than a solid. If the rigid option is specified, then the curve or set of curves will remain oriented as originally oriented, rather than rotating about the axis.

**Sweep Curve <curve_id_range> {Axis <xpoint ypoint zpoint xvector yvector zvector>|Xaxis|Yaxis|Zaxis} Angle <degrees> [Steps <Number_of_sweep_steps>] [Draft_angle <degrees>] [Draft_type <integer>] [Make_solid] [Rigid]**

**9. Stitch Surfaces Together:** A body can be created from various surfaces that form a closed volume with command below. The geometry must be ACIS-type geometry (i.e. imported from IGES, STEP or fastq files) This option is also available for limited cases for facet-based surfaces.

**Create {Body|Volume} Surface <surface_id_range> [HEAL|Noheal] [Keep] [Sheet]**

The **heal** option will attempt to close small gaps in the surface; the **noheal** option disables this behavior. The **keep** option preserves the original surfaces.
All of the surfaces must form a closed water-tight volume for this command to succeed unless the **sheet** option is specified.
The **sheet** option allows for the creation of an open body. If the set of surfaces form a closed volume a sheet body is created instead of a volume.
In situations where the boundaries are not exactly within tolerance, the following command may be more effective:

```
Stitch {Body|Volume} <id_range>

 [tolerance <value>] [no_tighten_gaps]
```

**10. Loft Surfaces Together:** A body can be "lofted" between two surfaces to form a new body. Surfaces from solid bodies and sheet bodies may be used to create a loft body. In order to create the loft body, two surfaces coincident to the input surfaces are created. The loft body is extruded along the shortest path between the corresponding vertices that define the shapes of the two surfaces. If guide curves are used the loft body is extruded along the guide curves. This new body is solid. The surfaces used to create the loft body are unchanged.

**Create {Body|Volume} Loft Surface <ids> [guide curve <id_list> [global_guides]] [Takeoff_factors <one value per surface in order>=.001] [Takeoff_vector Surface <id> {direction options}] [match vertex <ids>] [closed] [preview] [show_matching_curves]**

**Note:**Source surface ids must be specified in lofting order.
Go to Location, Direction, and Axis Specification to see the direction command description.
The following options are available for lofting:

- **Guide curve:** Multiple curves may be specified to guide the loft. The curves must touch the curves of each source surface. If the global_guides option is specified the guides curves are applied in a global nature.
- **Takeoff_factors:** Takeoff factors control how strongly the loft follows the takeoff vectors. When specifying takeoff factors one value must be specified for each source surface.
- **Takeoff_vector:** The takeoff vector controls the tangency or direction of the loft for each surface. The default takeoff vector for each surface is the normal at the surface centroid. If manually specified, one takeoff vector must be specified for each surface.
- **Match vertex:** This option guides the loft in how to match the vertices of the source surfaces. Multiple match vertex sets may be specified. When specifying

match vertices, one vertex id from each source surface must be specified. The match vertices must be specified in loft order.

- **Closed:** This option atempts to create a toroidal solid. The last source surface is lofted to the first source surface.
- **Preview:** This option will preview the linking curves of the final solid.
- **Show_matching_curves:** This option will preview how the vertices of the source surfaces will be matched.

Lofting can be used to split a body in order to create a more structured mesh. Figure 2 below shows a single volume swept from a large paved surface. Figure 3 shows this same volume after surfaces defined on the source and target surfaces have been used to create a loft body. This original body was chopped with the loft body. The resulting two bodies were merged. The yellow volume was swept as the volume in Figure 2 was but the purple volume was submapped, producing a much more structured mesh overall.

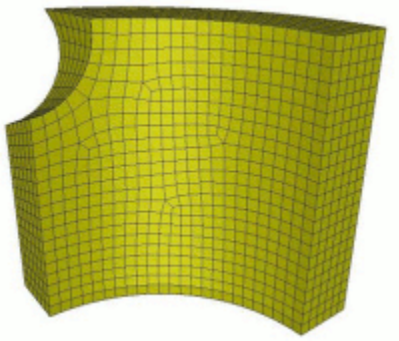


**Figure 2. Mesh before loft. Single swept volume with a large paved face.**

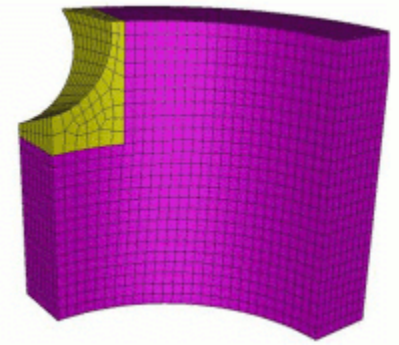


**Figure 3. Mesh after loft. The yellow volume is paved and the purple volume is submapped.**

**11. Thicken Surfaces:** A surface body can be thickened to create a volume body. The surface can be thickened in both directions using the "both" keyword, thickened in the direction of surface normal using a positive depth, or thickened in the opposite direction using a negative depth. To thicken multiple surfaces, all surface normals must be consistent.

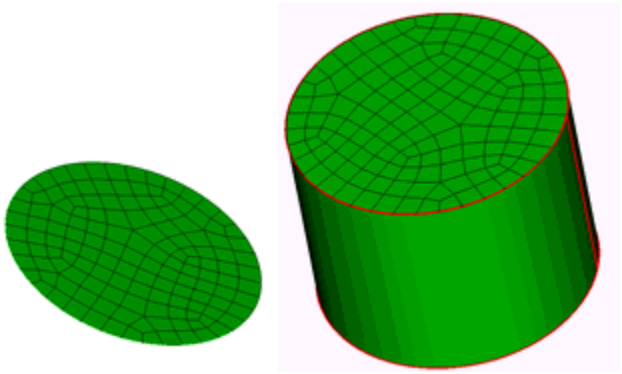
**Thicken [Volume|BODY] <id> Depth <depth> [Both]**

**12. Sweeping a Surface to a Plane:** Sweeps a surface normal to a plane and towards the plane until the swept surface reaches the plane. See plane options for ways to describe a plane.

**Sweep surface <id> target plane <options>**

**13. Sweep Surface along a Direction**: Sweep a surface along a direction to create a volume. See direction options for ways to specify a direction.

**Sweep Surface <surface_id_range> Direction (options) [switchside] [draft_angle <degrees>] [draft_type <integer>] [rigid] [anchor_entity] [include_mesh] [keep] [merge]**



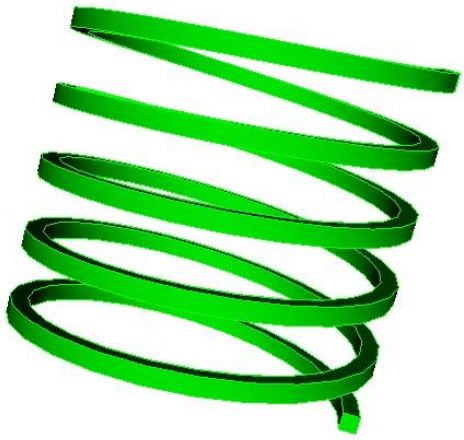
**Surface extruded along -X direction without 'include_mesh' option**

**14. Sweep Surface along Helix**: Sweep a surface along a helix, where the helix is defined by an axis, **thread_distance** (distance between turns in axis direction), axis, and handedness (**right_handed** or **left_handed**.

**Sweep {Surface|Curve} <id_range> Helix {axis <xpoint ypoint zpoint xvector yvector zvector> | xaxis | yaxis | zaxis} thread_distance <val> angle <val> [RIGHT_HANDED|left_handed] [anchor_entity] [include_mesh] [keep] [merge]**

***** Specifying multiple Surfaces that belong to the same Body can cause the creation of invalid Bodies and is discouraged. *****

**axis** = axis about which to create the sweep
**thread_distance** = distance between each 360 degree segment of the helix
**angle** = number of degrees in rotation of the helix
**handedness** = right-handed or left- handed threads



**Helical Sweep**

*Creating Curves*

Curves are created by specifying the bounding lower-order topology (i.e. the vertices) and the geometry (shape) of the curve (along with any parameters necessary for that geometry). There are several forms of this command:

- Straight
- Parabolic, Circular, Ellipse
- Spline
- Copy
- Combine Existing Curves
- Arc Three
- Arc End Vertices and Radius
- Arc Center Vertex
- Arc Center Angle
- From Vertex Onto Curve
- Offset
- From Mesh Edges
- Close_To
- Surface Intersection
- By Projection
- Helix
- Tangents

**1. Straight:** The first form of the command creates a straight line or a line lying on the specified surface. If a surface is used, the curve will lie on that surface but will not be associated with the surface's topology.

**Create Curve [Vertex] <vertex_id> [Vertex] <vertex_id> [On Surface <surface_id>]**

Straight curves can be created using an axis. The syntax is as follows:

**Create Curve Axis {options}**

The length of the axis must be specified. Go to Location, Direction, and Axis Specification to see the axis command description.
Additionally, several connected straight curves can be created with a single command. The syntax for the polyline command is as follows:

**Create Curve Polyline Location {options} Location {options} ...**

Notice that two or more locations are used to define a polyline. See Location, Direction, and Axis Specification for the location command description.
**2. Parabolic, Circular, Ellipse:** The parabolic option creates a parabolic arc which goes through the three vertices. The circular and ellipse options create circular and elliptical curves respectively that go through the first and last vertices.

> **Create Curve [Vertex <vertex_id> [Vertex] <vertex_id> [[Vertex] <vertex_id> [Parabolic|Circular|ELLIPSE [start angle <val=0>] [stop angle <val=90>]]]**

If **'ellipse'** is specified, Cubit will create an ellipse assuming the vectors between vertices (1 and 3) and (2 and 3) are orthogonal. v1-v3 and v2-v3 define the major and minor axes of the ellipse and v3 defines the center point. These vectors should be at 90 degrees. If not, Cubit will issue a warning indicating the vertices are not sufficient to create an ellipse and will then default to creating a spiral.
The angle options will specify what portion of the ellipse to create. If none are specified, **start angle** will default to 0 and **stop angle** to 90 and the ellipse will go from vertex 1 to vertex 2; if the vertices are free vertices they will be consumed in the ellipse creation. **Start angle** tells Cubit where to start the ellipse -- the angle from the first axis (v1 - v3) specified. **Stop angle** tells Cubit where to end the ellipse -- the angle from the first axis. The angle follows the right-hand rule about the normal defined by (v1 - v3) X (v2 - v3).

**3. Spline:** The spline form of the command creates a spline curve that goes through all the input vertices or locations. To create a curve from a list of vertices use the syntax shown below. The **delete** option will remove all of the intermediate vertices used to create the spline leaving only the end vertices.

**Create Curve [Vertex] <vertex_id_list> [Spline] [Delete]**

Additionally, spline curves can be created by inputting a list of locations. Where the spline will pass through all of the specified locations. The syntax is shown below:

**Create Curve Spline {List of locations}**

See Location, Direction, and Axis Specification to view the location specification syntax.
**4. Copy:** This command actually copies the geometric definition in the specified curve to the newly created curve. The new curve is free floating.

**Create Curve From Curve <curve_id>**

**5. Combine Existing Curves:** This command creates a new curve from a connected chain of existing ACIS curves.

**Create Curve combine curve <id_list> [delete]**

**6. Arc Three:** The following command creates an arc either through 3 vertices or tangent to 3 curves. The *Full* qualifier will cause a complete circle to be created.

**Create Curve Arc Three {Vertex|Curve} <id_list> [Full]**

**7. Arc End Vertices and Radius:** The following command creates an arc using two vertices, the radius and a normal direction. The *Full* qualifier will cause a complete circle to be created.

**Create Curve Arc Vertex <id_list>**
**Radius <value> Normal {<x> <y> <z> | {direction options} [Full]**

Go to Location, Direction, and Axis Specification to see the direction command description.
**8. Arc Center Vertex:** The next form of the command creates an arc using the center of the arc and 2 points on the arc. The arc will always have a radius at a distance from the center to the first point, unless the *Radius* value is given. Again, the *Full* qualifier will cause a complete circle to be created.

**Create Curve Arc Center Vertex <center_id> <end1_id> <end2_id>**
**[Radius <value>] [Full]**
**[Normal {<x> <y> <z> | {direction options}]**

Go to Location, Direction, and Axis Specification to see the direction command description.
**Note:** Requires 3 Vertices - first is the center, the other two are the end points of the arc. A normal direction is required when the three points are colinear. Otherwise a normal direction is optional.
**9. Arc Center Angle:** This form of the command creates an arc using the center position of the arc, the radius, the normal direction and the sweep angle.

**Create Curve Arc Center {<x=0> <y=0> <z=0> | {location options}**
**Radius <value>**
**Normal {<x> <y> <z> | {direction options}**
**Start Angle <value=0> Stop Angle <value=360>**

Go to Location, Direction, and Axis Specification to see the location and direction command descriptions.

**10. From Vertex Onto Curve:** The following command will create a curve from a vertex onto a specified position along a curve. If none of the optional parameters are given, the location on the curve is calculated as using the shortest distance from the start vertex to the curve (i.e., the new curve will be normal to the existing curve).

**Create Curve From Vertex <vertex_id> Onto Curve <curve_id> [Fraction <f> | Distance <d> | Position <xval><yval><zval> | Close_To Vertex <vertex_id> [[From] Vertex <vertex_id> (optional for 'Fraction' & 'Distance')]] [On Surface <surface_id>]**

**Note:** Default = Normal to the Curve

**11. Offset:** The next command creates curves offset at a specified distance from a planar chain of curves. The direction vector is only needed if a single straight curve is given. The offset curves are trimmed or extended so that no overlaps or gaps exist between them. If the curves need to be extended the extension type can be *Rounded* like arcs, *Extended* tangentially (the default -straight lines are extended as straight lines and arcs are extended as arcs), or extended *naturally*.

**Create Curve Offset Curve <id_list> Distance <val> [Direction <x> <y> <z>] [Rounded|EXTENDED|Natural]**

**Note:** Direction is optional for offsets of individual straight curves only
In all cases, the specified vertices are not used directly but rather their positions are used to create new vertices.

**12. From Mesh Edges**: This commands creates a curve from an existing mesh given a starting node and an adjacent edge.

**Create Curve From Mesh Node <id> Edge <id> [Length <val>]**

The adjacent edge indicates which direction to propagate the curve.
The curve will be composed of mesh edges up to the specified length.
If no length is specified the curve will propagate as far as the boundary of the mesh.
Figure 1 shows a example of a curve generated from the mesh.

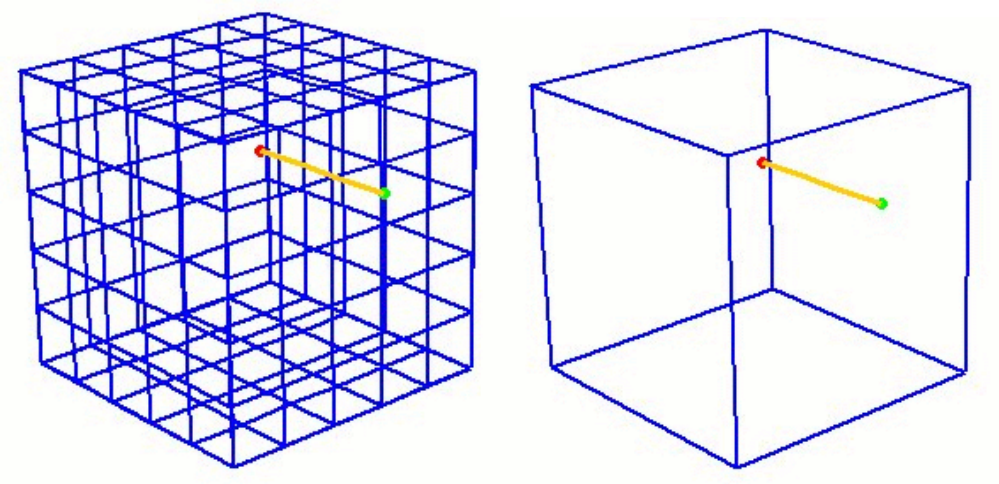


**Figure 1. Example of curve created from mesh**

The underlying geometry kernel used for this command is Mesh-Based geometry. The new curve will also be meshed with the edges it was propagated through. A related command for assigning mesh edges directly to a mesh block is the Rebar command. See Element Block Specification for more details.

Note: Full hexes or full tets must be used to propagate the curves through the interior of volume.

**13. Close_To** This option takes two geometric entities and creates the shortest possible curve between the two entities at the location where the two entities are the closest. The two entities may NOT intersect. If two vertices are given, the command will create a straight line between the two vertices.

**Create Curve Close_To {Vertex|Curve|Surface|Volume|Body} <id_1> {Vertex|Curve|Surface|Volume|Body} <id_2>**

**14. Surface Intersection** The following command creates curves at surface intersections. Multiple curves can be created from a single command.

**Create Curve Intersecting Surface <id_list>**

**15. By Projection** The project command projects curves, or the curves of a surface another a single surface or multiple surfaces of a volume or body. The command syntax is as follows:

**Project { Curve <id_list> | Surface <id_list> } Onto Surface <surface_id> [Imprint [Keepcurve] [Keepbody]] [Trim]**

**Project { Curve <id_list> | Surface <id_list> } Onto {Body <id> | Volume <id>} [Target_surface <id_list>] [Imprint [Keepcurve] [Keepbody]]**

The first form of the command takes a list of curves or surfaces, and a projection surface. If a list of curves is given, the result will be the creation of a set of free curves on top of

the projection surface. If a list of surfaces is given, the result will be the same as selecting the curves of the surface (i.e. a group of free curves on the projecting surface).
The second form will imprint the list of curves (or curves of the surface(s)) onto the surfaces of the specified bodies or volumes. The **Target_surface** option helps when the projection is ambiguous, for example projecting curves onto a thin-walled volume where the projection could be to either side, as shown in Figure 2.

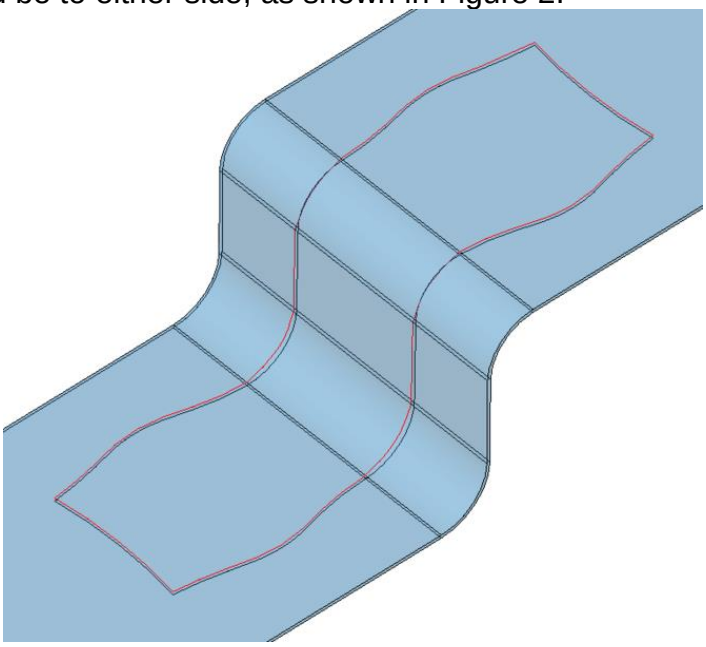


**Figure 2. Example of projecting curves to specific side of thin-walled volume.**

The imprint option will imprint the resulting projected curves onto the projection surface. If this option is NOT given, the new curves will lie coincident to the surface, but will not be part of the surface. Imprinting changes the topology of the projection surface. Keepcurve option retains the new curves as both free curves, and curves in the projection surface. The keepbody option retains the original body under the new imprinted body. When projecting curves, the trim option will cause the curve to be trimmed to the target surface.
**16. Creating a Helix**: This command will create a helical curve. The command syntax is as follows:

> **Create Curve Helix { axis <xpoint ypoint zpoint xvector yvector zvector> | xaxis | yaxis | zaxis } location (options) thread_distance <value> angle <value> [RIGHT_HANDED | left_handed]**

**axis** = axis about which to create the helix
**location** (options) = starting point of the helix
**thread_distance** = distance between each 360 degree segment of the helix
**angle** = number of degrees in rotation of the helix
**handedness** = right-handed or left- handed threads

**17. Tangents**: This command will create a spline curve by specifying the end points and the tangents at those points. The command syntax is as follows:

**create curve tangent vertex <id> vertex <id> [start direction (options)] [end direction (options)]**

**create curve tangent start location (options) end location (options) start direction (options) end direction (options)**

Both forms of the command can be broken into two parts: the points and the tangent directions. The first point is associated with the start direction.
The first form of the command takes an existing vertex and an optional tangent direction. If the direction is not specified, it will be taken from the tangent at the endpoint of the connected curve. If the vertex is not connected to a curve, or it is connected to multiple curves, the direction must be specified. If the vertex is not connected to a curve, it will be incorporated into the new curve. Otherwise, a new vertex will be created for the new curve.
The second form of the command takes a location and a tangent direction. The directions must be specified and vertices will be created for the locations.
The two command forms can also be mixed. A curve can be created from an existing vertex and a specified location.
**Examples:**

```
create curve tangent start location 0 0 0 end location 1 1 0 start direction 1 0 0 end direction 1 0 0
```

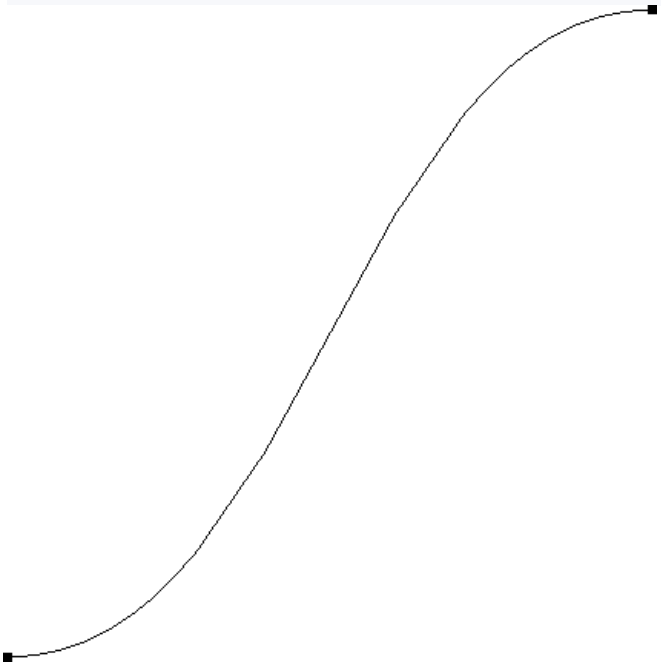


**Figure 3. Create tangent curve with locations and tangent directions**

```
create curve tangent vertex 1 vertex 3
merge vertex all
```

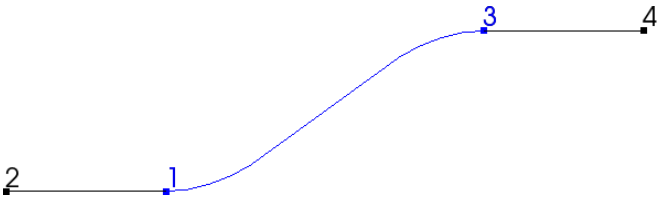



**Figure 4. Create tangent curve by specifying vertices on curves. Tangents are extracted from the curve at the vertex location.**

Go to Location, Direction, and Axis Specification to see the location and direction command description.

*Creating Surfaces*

There are two major ways to create surfaces in CUBIT. First, surfaces can be created in CUBIT by fitting an analytic or spline surface over a set of bounding curves. In this case, the curves must form a closed loop, and only one loop of curves may be supplied. The second method, is by sweeping a curve about an axis, along a vector, or along another curve. The result of these surface creation commands is a "sheet body" or a body that has zero measurable volume (it does however have a volume entity). This body may be decomposed with booleans and special webcutting commands or it may be used as a tool to decompose other bodies. Booleans can be used to cut holes out of these surfaces. The following options may be used for creating a surface in CUBIT.

- Bounding Curves
- Bounding Vertices or Nodes
- Copy
- Extended Surface
- Planar Surface
- Net Surface
- Offset
- Skinning
- Sweeping of Curves
- Midsurface
- Weld Profile
- Meshed Entities
- Circular Surface
- Parallelogram
- Ellipse
- Rectangle

**1. Bounding Curves:** The first form of this command produces an analytic or spline surface fit to cover the bounding curves.

**Create Surface Curve <curve_id_1> <curve_id_2> <curve_id_3>...**

Another version of this command creates a surface from a set of bounding curves that all lie on one surface. If the curves are selected they must lie on the surface, and they must create a closed loop. The **On Surface** option forces the surface to match the geometry of the underlying surface exactly.

**Create Surface Curve <id_list> On Surface <surface_id>**

**2. Bounding Vertices or Nodes:** The second form of this command uses vertices to fit an analytic spline surface. The **On Surface** option creates the surface from a set of nodes and vertices that all lie on one surface and restrains the surface to match the geometry of the underlying surface. The project option will project the nodes or vertices to the specified surface.

**Create Surface [Node|Vertex| <id_list> [On Surface <surface_id> {Project} ]**

**3. Copy:** The next form creates a surface using the same geometric description of the specified surface. The new surface will be a stand-alone sheet body that is geometrically identical to the user supplied surface.

**Create Surface From Surface <surface_id>**

**4. Extended Surface:** The fourth form of the command creates a surface that is extended from a given surface or list of surfaces. The specified surface's geometry is examined and extended out "infinitely" relative to the current model in CUBIT (i.e. extended to just beyond the bounding box of the entire model). The given surfaces are extended as shown in the table.

**Create Surface Extended From Surface <surface_id>**

**Table 1. Surface Extension Results**

| **Surface Type** | **Resulting Extended Surface** |
|---|---|
| Spherical | Shell of Full Sphere |
| Planar | Plane of infinite size relative to model |
| Toroidal | Shell of Full Torus |
| Conical, cone, cylinder... | Shell of outside conic axially aligned with given conic of infinite height relative to model |
| Spline | Surface is extended to extents of the spline definition. This may not be any further than the surface itself, so caution should be used here. |

Multiple surfaces can be offset at the same time to form a sheet body, by using the Create Sheet Extended from Surface command.

**5. Planar Surface:** The following commands create ***planar*** surfaces. The first passes a plane through 3 vertices, the second uses an existing plane, the third creates a plane normal to one of the global axes, and the fourth creates a plane normal to the tangent of a curve at a location along the curve. By default, the commands create the surface just large enough to intersect the bounding box of the entire model with minimum surface area. Optionally, you can give a list of bodies to intersect for this calculation. You can also extend the size of the surface by either a percentage distance or an absolute distance of the minimum area size. The plane can be previewed with the command Draw Plane [with]... (where the rest of the command is the same as that to create the surface).

**Create Planar Surface [With] Plane Vertex <v1_id> [Vertex] <v2_id> [Vertex] <v3_id> [Intersecting] Body <id_range>] [Extended Percentage|Absolute <val>]**

**Create Planar Surface [With] Plane Surface <surface_id> [Intersecting] Body <id_range>] [Extended Percentage|Absolute <val>]**

**Create Planar Surface [With] Plane {Xplane|Yplane|Zplane} [Offset <val>] [Intersecting] Body <id_range>] [Extended Percentage|Absolute <val>]**

**Create Planar Surface [With] Plane Normal To Curve <curve_id>{Fraction <f>| Distance <d> | Position <xval><yval><zval> | Close_to vertex <vertex_id>} [[From] Vertex <vertex_id> (optional for 'fraction' & 'distance')] [Intersecting] Body <id_range>] [Extended Percentage|Absolute <val>]**

**6. Net Surface:** *Net surfaces* can be created with two different commands. A net surface passes through a set of curves in the u-direction and a set of curves in the v-direction (these u and v curves would looked like a mapped mesh). The first form of the command uses curves to create the net surface. The curves must pass within tolerance of each other to work. The second form uses a mapped mesh to create the surface. The mapped mesh can be of a single surface or a collection of mapped or submapped surfaces that form a logical rectangle. By default net surfaces are healed to take advantage of any possible internal simplification.

**Create Surface Net U Curve <id_list> V Curve <id_list> [Tolerance <value>] [HEAL|Noheal]**

**Create Surface Net [From] [Mapped] Surface <id_list> [Tolerance <value>] [HEAL|Noheal]**

A suggested geometry cleanup method is to use a virtual composite surface to map mesh a set of complicated surfaces then create a net surface from this mesh. Then the original surfaces can be removed with the *noextend* option and the new net surface combined back onto the body.

**7. Offset**: The following command creates surfaces ***offset*** from existing surfaces at the specified distances.

**Create Surface Offset [From] Surface <id_list> Distance <val>**

The surface offset command will only translate the existing surfaces, without extending or trimming them. An alternate form of the command for sheet bodies will maintain connections between surface by extending or trimming as they are offset, shown in Figure 1. On the left, the surfaces are offset using the surface offset command. On the left, the surface is created by using the "sheet" version of the command.

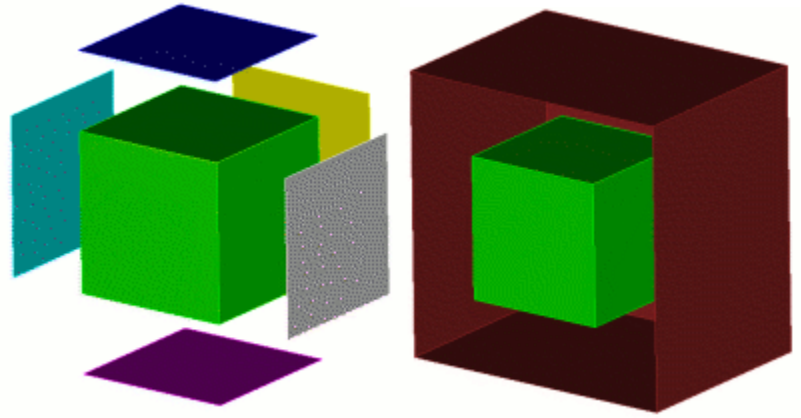


**Figure 1. Offsetting surfaces to form individual surfaces or sheet bodies**

**8. Skinning:** The following command creates a ***skin*** surface from a list of curves. An example of a skin surface is to create a surface through a set of parallel lines.

**Create Surface Skin Curve <id_list>**

**9. Sweeping of Curves:** A curve or a set of curves can be swept along a path to create new surfaces. The path may be specified as an axis and angle, a vector and distance, by indicating another curve or set of contiguous curves, or by specifying a target plane. The following commands show the options available:

**Sweep Curve <curve_id_range> { Axis <xpoint ypoint zpoint xvector yvector zvector> | Xaxis | Yaxis | Zaxis } Angle <degrees> [Steps <Number_of_sweep_steps>] [Draft_angle <degrees>] [Draft_type <integer>] [Make_solid] [Include_mesh] [Keep][Rigid]**

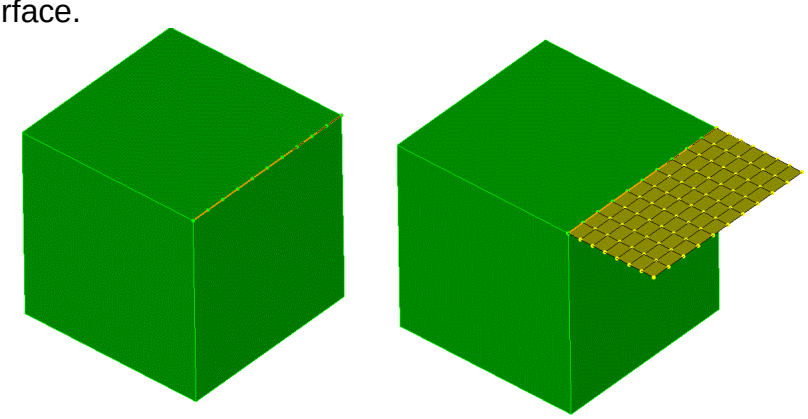
**Sweep Curve <curve_id_range> Vector <xvector yvector zvector> [Distance <distance>] [Draft_angle <degrees>] [Draft_type <integer>] [Include_mesh] [Keep] [Rigid]**

**Sweep Curve <curve_id_range> Along Curve <refcurve_id_range> [Draft_angle <degrees>] [Draft_type <integer>] [Include_mesh] [Keep] [Rigid]**

**Sweep Curve <curve_id_range> Target Plane <options>**

**Sweep Curve <curve_id_range> Target {Volume|Body} <id> Direction {options} [Plane <options>] [Unite]**

In the first command, the steps options provides a way of faceting the sweep, so instead of a smooth round sweep, there are facets to the surface. The **make_solid** option closes the newly-created surface to the axis, so that a solid is created instead of a surface.
In the above commands, the include_mesh option will create a surface mesh if the curve is already meshed (see figure below). The keep option will keep the original curve while creating the surface.



The **sweep curve target plane** command sweeps a curve until it hits a target plane. The options for the target plane are described under Specifying a Plane.
The last command sweeps a curve to a target volume or body and can only be used on sheet bodies. Use the direction keyword to specify the sweep direction and the plane keyword to specify a stopping plane. The **unite** keyword will unite the sheet bodies after sweeping
The other options are as follows:
draft_angle: determines how much drafting in of the surface is desired
draft_type:

0 => extended (draws two straight tangent lines from the ends of each segment until they intersect)

1 => rounded (create rounded corner between segments)

2 => natural (extends the shapes along their natural curve) ***
rigid: normally the curve will rotate to maintain its original orientation to the sweep path. The rigid option disallows this rotation.
**10. Midsurface:** Multisurfaces may be created midway between pairs of surfaces using the following command:

**Create Midsurface {Body|Volume} <id> Surface <id11> <id12> ... <idN1> <idN2>**

where N denotes the number of pairs of surfaces. An even number of surfaces must be specified, and the command will group them by pairs in the order in which they are provided. The resulting surface will be trimmed by the specified body or volume <id>. This replaces the *Create Midplane* command in previous versions of CUBIT.

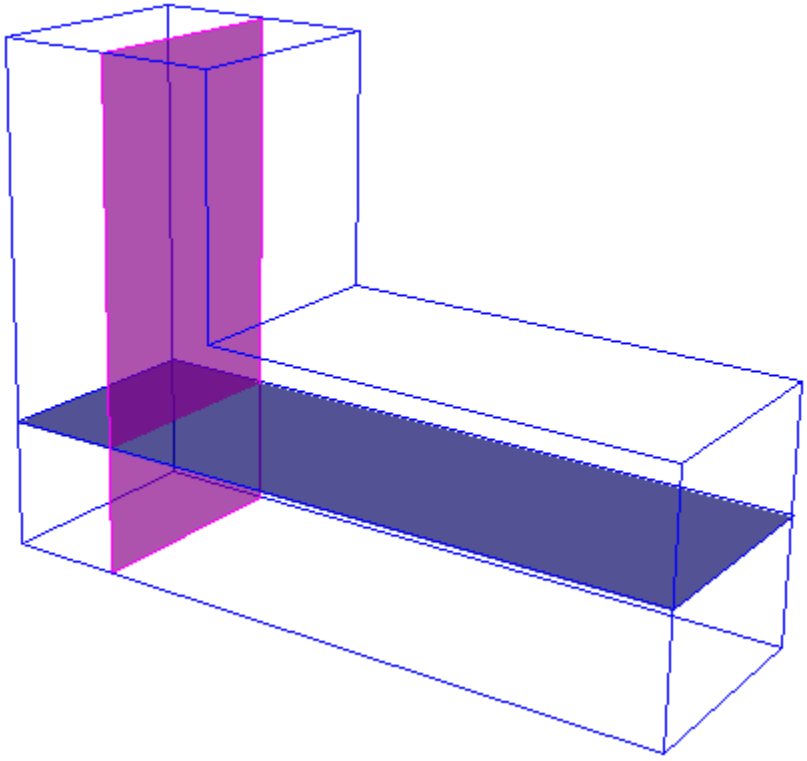


**Figure 2. Multisurface created with the Create Midsurface command**

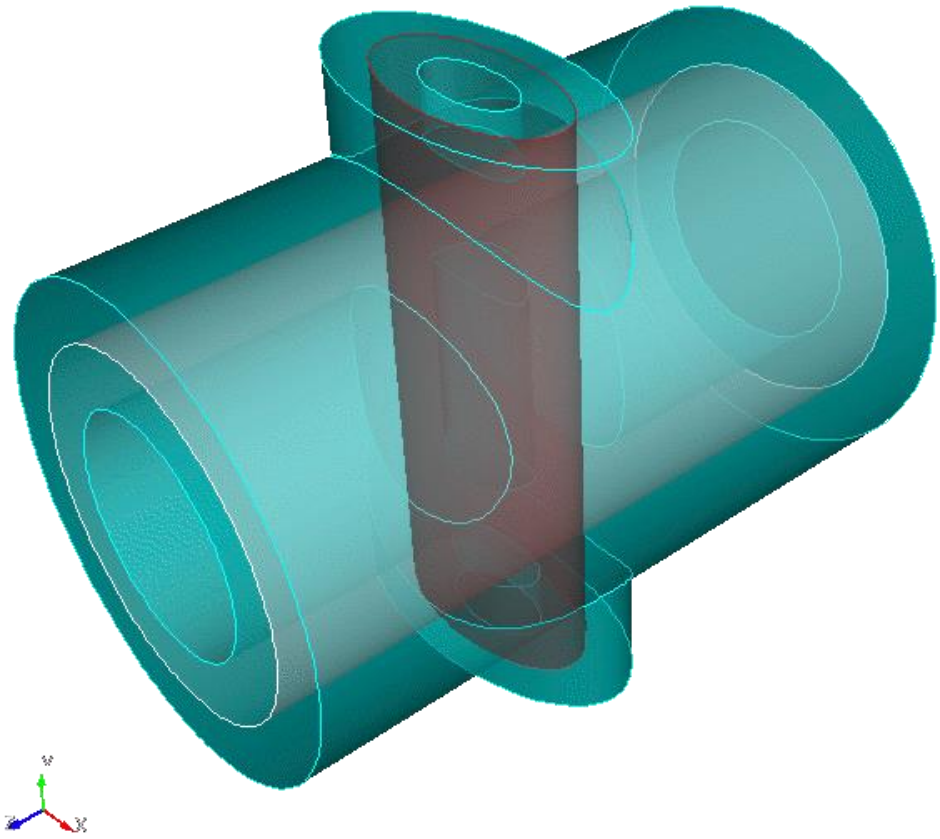


**Figure 3. Midsurface created from 2 pairs of cylindrical surfaces**

Midsufaces can also be extracted without surface pair specification if the resulting surface is a single sheet of surfaces (no T intersections). The following is the command syntax for automatic midsurface extraction:

**Create Midsurface {Body|Volume} <id_range> Auto [Delete] [Transparent] [Thickness] [Limit <lower_bound> <upper_bound>] [Preview]**

Figure 4 shows a simple auto midsurface example. The command for the example is:
**create midsurface volume 1 auto delete**

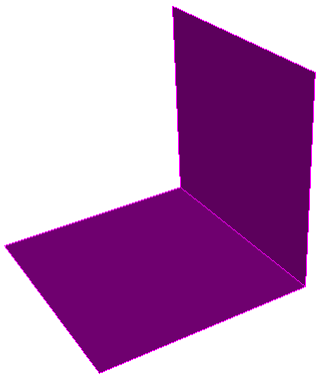


**Figure 4. Midsurface created from a volume**

The command option descriptions are listed below.
**Auto** enables the automatic mid-surface algorithm. Turning Auto off requires the user to specify a single surface pair to create a mid-surface.
**Transparent** shows the successfully midsurfaced volumes as transparent in the graphics display
**Thickness** applies a 2D property to the created mid-surface geometry.
**Limit** search range gives the algorithm a range to find surface pairs within.
**11. Weld Profile:** Surfaces may be created by specifying a weld profile using the following command:

```
Create Surface Weld [Root] Location {options} Weld Surface <id_list> Length <val> [<val2>]
```

Weld surfaces can be used to create a simulated welded joint by sweeping the surface along the root curve and uniting the new body to the model. An example of the command is illustrated below. For a detailed description of the location specifier see Location Direction, and Axis Specification.

```
create surface weld root location vertex 25 weld surface 13 14 length 2
```

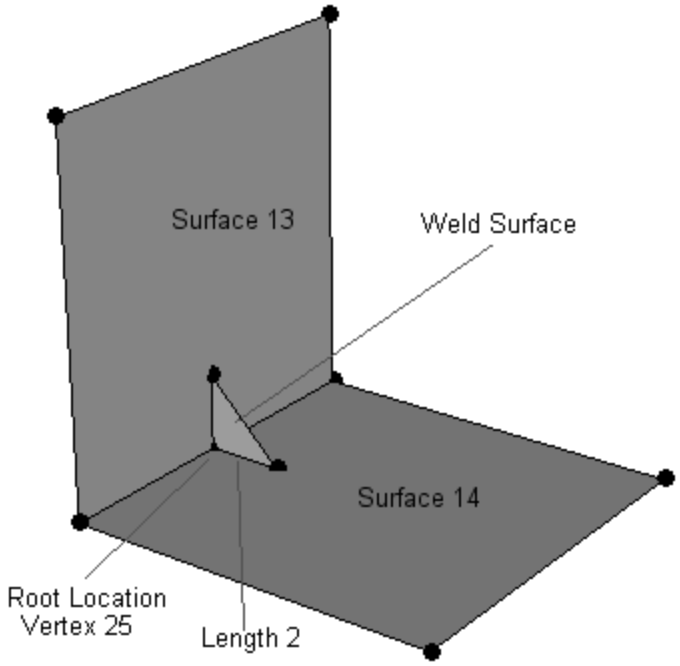



**Figure 5. Weld Profile surface with length and root specifications**

**12. Creating A Surface From Mesh Entities:** Surfaces may be created from the boundaries of meshed volumes, surfaces, and/or from individual quadrilateral mesh elements. The individual option makes it so you can enter multiple surfaces at once, and not have them merged together into a larger surface, but instead retain their own original boundaries. The optional tolerance value allows the user to specify a tolerance to which the resulting surface should be fit. The default value is 0.001. If surface creation fails, increasing the tolerance value can help.

**Create Acis [From] {Surface <id_range> | Volume <id_range> | Face < id_range> [Individual]} [Tolerance <value>]**

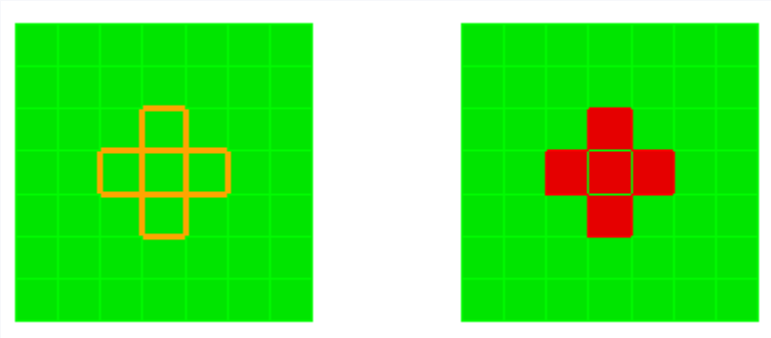


**Figure 6. Acis Surface created from a Set of Quadrilaterals**

**13. Creating a Circular Surface**: This command creates a 2D circular surface. The surface will be centered at the origin and on the z-plane if a plane option is not specified.

`create surface circle radius <value> {xplane|yplane|ZPLANE}`
This command creates a 2D circular surface by specifying three vertices; the first vertex will be the center of the surface, the second vertex will be used to define the radius of the surface, and the third vertex will assist in defining the plane that the surface will lie in.

`create surface circle center vertex <v1_id> <v2_id> <v3_id>`
This command creates a 2D circular surface by forming a circular curve through three points.

`create surface circle vertex <v1_id> <v2_id> <v3_id>`
**14. Creating a Parallelogram**: This command creates a 2D parallelogram surface, centered at the origin, by specifying three corner vertices. These vertices will form three consecutive corners of the parallelogram surface.

`create surface parallelogram vertex <v1_id> v2_<id> <v3_id>`
**15. Creating an Ellipse**: This command creates a 2D elliptical surface, centered at the origin, by specifying at least a major radius. On an x-y plane this radius will be the radius along the x-direction. The minor radius will be the radius along the y-direction. By default, the surface will lie in the z-plane.

**Create Surface Ellipse major radius <value> [minor radius <value>] [xplane|yplane|ZPLANE]**

This command creates a 2D elliptical surface using three vertices. The first two vertices define the major and minor radii of the ellipse surface. The third point defines the center of the ellipse. It is important to note that a line from v1_id to v3_id must be orthogonal to a line from v2_id to v3_id, otherwise the command will fail.

**Create Surface Ellipse vertex <v1_id> <v2_id> <v3_id>**

**16. Creating a Rectangle**: This command creates a rectangular surface centered at the origin. If only a width value is specified, the surface will be a square. On an x-y plane, the width value is the x-direction and the height is the y-direction. By default, the surface will lie in the z-plane.

**Create Surface rectangle width <value> [height <value>] [xplane|yplane|ZPLANE]**

*Creating Vertices*

The basic commands available for creating new vertices directly in CUBIT are:

- XYZ location
- On Curve - Fraction
- On Curve - General
- From Vertex
- At Arc
- At Intersection

**1. XYZ location:** The simplest form of this command is to specify the XYZ location of the vertex. It can also be created lying on a curve or surface in the geometric model by specifying the curve or surface id; the position of the vertex will be the point on the specified entity which is closest to the position specified on the command. With all of these commands, the user is able to specify the color of the vertex.

**Create Vertex <x><y><z> [On [Curve | Surface] <id>] [Color <color_name>]**

**2. On Curve - Fraction:** A vertex can be positioned a certain fraction of the arc length along a curve using the second form of the command.

**Create Vertex On Curve <id> Fraction <0.0 to 1.0> [Color <color_name>]**

Vertex 3 in the following example was created with this command:
**create vertex on curve 1 fraction 0.25 from vertex 1**

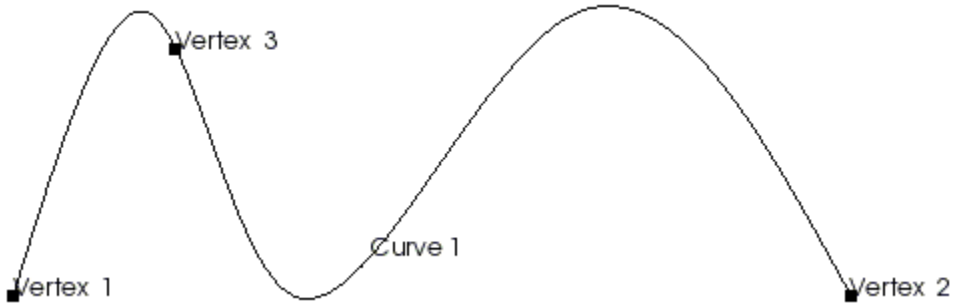



**Figure 1. Create Vertex a Fraction of the length of a Curve**

**3. On Curve - General:** A more general purpose form of the command is also available for creating vertices on curves:

**Create Vertex On Curve <id_list> { MIDPOINT | Start | End | Fraction <val 0.0 to 1.0> [From Vertex <id> | Start|End] | Distance <val> [From {Vertex|Curve|Surface} <id> | Start|End] | {{Close_To|At} Location {options} | Position <xval><yval><zval>|{Node|Vertex} <id>} | Extrema [Direction] {options} [Direction {options}] [Direction {options}] | Segment <num_segs> | Crossing {Curve|Surface} <id_list> [Bounded|Near] } [Color <color_name>]**

It allows the vertex to be created at a fractional distance along the curve, at an actual distance from one of the curves ends, at the closest location to an xyz position or another vertex, or at a specified distance from a vertex, curve or surface. You can also preview the location first with the command Draw Location On Curve (where the rest of the command is identical to the Create Vertex form).
**4. From Vertex:** Create a vertex from an existing vertex.

**Create Vertex from Vertex <id_list> [ On {Curve|Surface} <id> ] [Color <color_name>]**

If 'on curve|surface' option is used, the vertex is positioned on that curve or surface. When the 'on curve|surface' is not used, the new vertex is positioned on the existing vertex.
**5. At Arc:** Another form simply creates vertices at arc or circle centers.

**Create Vertex Center Curve <id_list> [Color <color_name>]**

**6: At Intersection:** The last form creates vertices at the intersection of two curves. If the *bounded* qualifier is used, the vertices are limited to lie on the curves, otherwise the extensions of the curves are also used to calculate the intersections. The *near* option is only valid for straight lines, where the closest point on each curve is created if they do not actually intersect (resulting in two new vertices).

**Create Vertex AtIntersection Curve <id1> <id2> [Bounded] [Near] [Color <color_name>]**

# transforms

## Geometry Transforms

- Align
- Copy
- Move
- Scale
- Rotate
- Reflect

Bodies can be modified in CUBIT using transform operations, which include align, copy, move, reflect, restore, rotate, and scale. With the exception of the copy operation, transform operations in CUBIT do not create new topology, rather they modify the geometry of the specified bodies. ACIS, Mesh Based Geometry and Virtual Geometry representations may be transformed. If the geometric entity has been meshed, the nodes of the mesh will be transformed along with the geometry. To transform the nodes of a mesh as they are written to the Exodus II mesh file without modifying their location within CUBIT, see Transforming Mesh Coordinates.

## Align Command

The align command is a combination of the rotate and move commands. This transformation is useful for aligning surfaces in preparation for geometry decomposition and aligning models for axis-symmetric analysis. If the **[include_merged]** option is used, all entities that are merged with the specified volume will be included in the align transformation also.

The first align command will transform the specified volumes by computing a transformation that aligns the source axis or surface with the target axis or surface such that the source axis centroid or surface centroid is coincident with the target axis origin or surface centroid and the surface normal(s) and/or axis(s) are pointing either in the same or opposite direction (depending on their initial alignment). The source surface need not be in the specified volumes. If the **[reverse]** option is specified, the resulting alignment is flipped 180 degrees.

An optional second axis or surface source-target pair can be specified. This will result in an additional rotational alignment: The specified volumes will be rotated about the first target axis or surface normal such that the second target axis or surface is at the same angle about the first target axis or surface normal.

**Align Volume <id_range>**
**Source {Axis {options}|Surface <surface_id>}**
**Target {Axis {options}|Surface <surface_id>}**
**[Source {Axis {options}|Surface <surface_id>}**
**Target {Axis {options}|Surface <surface_id>}]**
**[reverse] [include_merged] [preview]**

The image below shows the two alignments that occur when the additional source-target pair is specified.

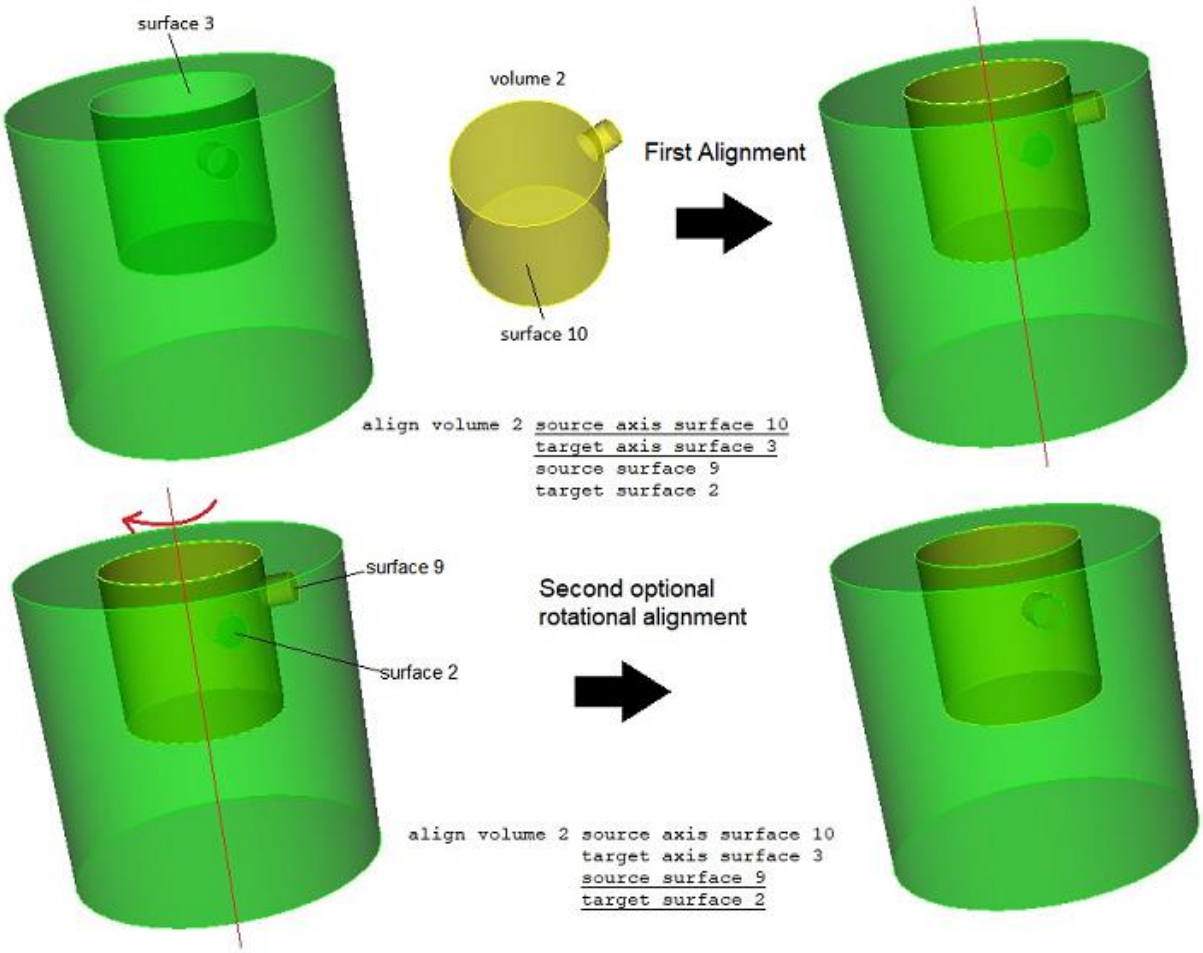



**Figure 1 - Aligning with optional rotation**

The second form of the align command either aligns a face of a volume or two vertices (forming a direction) with the xy, yz, and xz planes or the x, y, and z axes. If the **[reverse]** option is specified, the resulting alignment is flipped 180 degrees.

**Align Volume <id_range> {Surface <surface_id>| Vertex <vertex_id>} {{X|Y|Z Axis}|{XY|XZ|YZ plane}} [reverse] [include_merged] [preview]**

The third form of the command is a rotational alignment, where the specified entities are rotated about the specified axis, where the angle of rotation is the angle between the first and second locations with respect to the axis.

**Align Volume <id_range> Location {options} with Location {options} about Axis {options} [include_merged] [preview]**

The fourth form of the command uses vertex pairs to define the transformation. The first pair of vertices define a translation so that the source and target vertices are coincident (i.e.,the source is moved to the target). The optional second pair of vertices define a

rotation such that the source and target vertices are all collinear after the transformation. The optional third pair of vertices define a rotation such that all the source and target vertices will be coplanar.

**{Body|Volume|Surface|Curve|Vertex|Group} <id_range> Align Using Vertex <src_id> <tgt_id>**
**[Collinear Vertex <src_id> <tgt_id> [Coplanar Vertex <src_id> <tgt_id>] ] [include_merged]**

## Copy Command

The copy command copies an existing entity to a new entity without modifying the existing entity. A copy can be made of several entities at once, and the resulting new entities can be translated or rotated at the same time. The commands for copying entities are:

**Vertex <range> Copy [Move [X <dx>] [Y <dy>] [Z <dz>]] [Preview]**

**Vertex <range> Copy [Move <direction_options> [Distance <val>]] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <range> Copy Move [X <dx>] [Y <dy>] [Z <dz>] [Nomesh] [Repeat <value>] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <range> Copy Move <direction_options> [Distance <val>] [Nomesh] [Repeat <value>] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <range> Copy Reflect {X|Y|Z} [Nomesh] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <range> Copy Reflect [Vertex <v1_id> [Vertex] <v2_id] [Nomesh] [Preview]**

**{Body|Volume|Surface|Curve} <range> Copy Reflect <x> <y> <z> [Nomesh] [Preview]**

**{Body|Volume|Surface|Curve} <range> Copy Rotate <angle> About {X|Y|Z} [Repeat <value>] [Nomesh] [Preview]**

**{Body|Volume|Surface|Curve} <range> Copy Rotate <angle> About <x> <y> <z> [Nomesh] [Repeat <value>] [Repeat <value>] [Preview]**

**{Body|Volume|Surface|Curve} <range> Copy Scale <scale> | X <val> Y <val> Z <val> [About Vertex <id>] [Nomesh] [Repeat <value>] [Preview]**

If the copy command is used to generate new entities, a copy of the original mesh generated in the original entity will also be copied directly onto the new entity unless the **nomesh** option is used.

Several of the commands include the **Repeat** token. If that token is used the command will repeat itself **value** times.
This is currently limited to copies that do not interact with adjacent geometry through non-manifold topology. For details on mesh copies, see the Mesh Duplication documentation.

## Move Command

The move command moves a body, volume, free surface, free curve or free vertex by a specified offset. The command syntax is:

**Vertex <id_range> [Move [X <dx>] [Y <dy>] [Z <dz>]] [Copy] [Preview]**

**Vertex <id_range> Move <direction_options< [Distance <val>] [Copy] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <id_range> [Move [X <dx>] [Y <dy>] [Z <dz>]] [Copy [Nomesh]] [Preview]**

**{Body|Volume|Surface|Curve|Vertex|Group} <id_range> Move <direction_options> [Distance <val>] [Copy [Nomesh]] [Preview]**

where <dx> <dy> <dz> and <distance> represent relative offsets in the major axis directions. If the copy option is specified, a copy is made and the copy is moved by the specified offset. The nomesh option will copy and move only the geometry.
These forms of the Move command will only work on free surfaces and free curves. To move a curve or surface that is part of a higher-order entity, the **Move {entity} ...** command is used.

*Moving Other Geometric Entities*

It is also possible to move bodies by specifying one of its child entities. For example, a body can by moved by specifying one of its curves. However, if a lower-order entity is moved, the parent body and all related entities will also be moved. The commands for moving bodies using a child entity are given below. Alternatively, the tweak command can be used to move curves and surfaces without moving the parent body.

**Move {Vertex|Curve|Surface|Volume|Body|Group} <id_range> [Midpoint] Location <x> [<y> [<z>]] [Include_Merged] [Preview]**

**Move {Vertex|Curve|Surface|Volume|Body|Group} <id_range> Location [Midpoint] [X <val>] [Y <val>] [Z <val>] [Except [X] [Y] [Z]] [Include_Merged] [Preview]**

**Move {Vertex|Curve|Surface|Volume|Body|Group} <id_range> Normal to Surface <id> Distance <val> [Include_Merged] [Preview]**

**Move {Vertex|Curve|Surface|Volume|Body|Group} <id_range> [Midpoint] General Location <location_options> [Except [X] [Y] [Z]] [Include_Merged] [Preview]**

The first form of the command will move the entity to an absolute location. If moving a group, the centroid of the group is moved to that location. The second form will move the entity by a relative distance in any of the xyz axis directions. "Except" is used to preserve the x, y, or z plane in which the center of the entity lies. The third form of the command will move the body along an axis defined by the outward-facing surface normal of another surface. The fourth form of the command uses general location parsing to move the entity.

*Moving Bodies Relative to Other Geometric Entities*

It is also possible to move bodies relative to other geometric entities in the model. The following command takes as arguments two geometric entities. The first entity is the one to move. The second entity is where it will be moved. In both cases, the midpoints of the specified entity are used to determine the distance and direction of the move. In the case of groups, centroids are used. "Except" is used to preserve the x, y, or z plane in which the center of the entity lies.

**Move {Vertex|Curve|Surface|Volume|Body|Group} <id_range> [Midpoint] Location {Vertex|Curve|Surface|Volume|Body|Group} <id> [Midpoint] [Except [X] [Y] [Z]] [Include_Merged] [Preview]**

*Moving Merged Entities*

The easiest way to move merged entities is by adding the *include_merged* keyword to the command. All entities that are merged with the specified entities will move together. The only other way that merged entities can be moved is by including each of the merged entities in the entity list.

*Move Undo*

The Undo option allows a user to reverse the most recent move. This command will only work for the **Move {entity}** commands, and not the **{Entity} Move** commands. The syntax is:

**Move Undo**

## Reflect Command

The reflect command mirrors the body about a plane normal to the vector supplied. The reflect command will destroy the existing body and replace it with the new reflected body, unless the copy option is used.

**{Body|Volume|Surface|Curve|Vertex|Group} <range> [Copy] Reflect <x-comp> <y-comp> <z-comp>**

**{Body|Volume|Surface|Curve|Vertex|Group} <range> [Copy] Reflect {X|Y|Z}**

## Rotate Command

The rotate command rotates a body about a given axis without adding any new geometry. If the Angle or any Components are not specified they are defaulted to be zero. The commands to rotate a body or bodies are:

**Body <range> [Copy] Rotate <angle> About {X|Y|Z} [Preview]**

**Body <range> [Copy] Rotate <angle> About <x-comp> <y-comp> <z-comp> [Preview]**

**Rotate {Body|Volume|Surface|Curve|Vertex|Group} <id_range> about {X|Y|Z|<xval> <yval> <zval>} Angle <val> [Include_Merged] [Preview]**

**Rotate {Body|Volume|Surface|Curve|Vertex|Group} <id_range> About Vertex <id> Vertex <id> Angle <val> [Include_Merged] [Preview]**

**Rotate {Body|Volume|Surface|Curve|Vertex|Group} <id_range> About Normal of Surface <id> Angle <val> [Include_Merged] [Preview]**

**Rotate {Body|Volume|Surface|Curve|Vertex|Group} <id_range> About Origin <xval> <yval> <zval> Direction <xval> <yval> <zval> Angle <val> [Include_Merged] [Preview]**

If the copy option is specified, a copy is made and rotated the specified amount.

*Rotating Merged Entities*

The easiest way to rotate merged entities is by adding the *include_merged* keyword to the command. All entities that are merged with the specified entities will rotate together. The only other way that merged entities can be rotated is by including each of the merged entities in the entity list.

## Scale Command

The **scale** command resizes an entity (body, volume, surface, or curve) by a scaling factor. The scaling factor may be a constant, or may differ in the x, y, and z directions. The entity chosen will be scaled about the point or vertex indicated. If no point or vertex is entered, it will be scaled about the origin. Any mesh on the object will be scaled too, unless the **nomesh** keyword is used.
The command to scale entities is:

**{Body|Volume|Surface|Curve} <id_range> Scale {<scale> | x <val> y <val> z <val>} [About {<x> <y> <z> | Vertex <id>}] [Nomesh] [Copy [Repeat <value>] [Group_Results]] [Preview]**

If the **copy** option is specified, a copy of the entity is made and scaled the specified amount. Use the **repeat** option to create multiple copies.

# booleans

## Geometry Booleans

- Intersect
- Subtract
- Unite

CUBIT supports boolean operations of intersect, subtract, and unite for bodies.
An automatic function associated with webcutting operations is regularizing geometry which can be turned off or back on with the following command:

**Set Boolean Regularize [ON | off]**

### Intersect

The intersect command generates one or more new volumes composed of the space that is shared by the volumes being intersected. There are two forms of the command. The first form is:

**Intersect {Volume|[Body]} <range> [Keep] [Preview]**

In this form, all volumes in the range will be intersected, and the resulting intersection volume(s) will contain all intersections of all input volumes. The original volumes will be deleted. The second form is:

**Intersect {Volume|[Body]} <id> [With {Volume|[Body]} <range>] [Keep] [Preview]**

In this form, the first volume is intersected with all volumes in the range specified using the **with** keyword. Intersections between volumes in the range are not included in the output. The original volume will be deleted and the other volumes updated with the intersection results.
The **keep** option results in the original volumes used in the intersect being kept.
If the **Preview** option is included in the command, the input volumes will not be modified. The computed intersection volume will be drawn as a red, shaded solid. For best results change the graphics mode to transparent or hidden line so the intersection is visible. Otherwise the intersection volume will be hidden by the volumes being intersected.

### Subtract

The subtract operation subtracts one body or set of bodies from another body or set of bodies. The order of subtraction is significant - the body or bodies specified before the **From** keyword is/are subtracted from bodies specified after **From**. The new body retains the original body's id. If any additional bodies are created, they will be given the next highest available ids. The **keep** option simply retains all of the original bodies. The command is:

**Subtract [Volume|BODY] <range> From [Volume|BODY] <range> [Imprint] [Keep]**

The imprint option imprints the subtracted bodies onto the resultant body.

*Remove Overlap*

The **Remove Overlap** command is a simplified form of the **subtract** operation above. This command takes exactly two volumes as arguments and uses the **modify** argument to define the volume where material is to be removed.

**Remove Overlap Volume <id1> <id2> modify [Volume <id> | Smaller | Larger]**

The **Smaller** and **Larger** options are an alternate method of specifying which volume from **<id1>** or **<id2>** where material will be removed. When these argument are used, the geometric volume is measured for both volumes and the smaller or larger volume repectively is used as the **modify** entity. When both volumes have exactly the same volume, the smaller or larger entity ID is used.

## Unite

The unite operation combines two or more bodies into a single body. The original bodies are deleted and the new body is given the next highest body ID available, unless the **keep** option is used. The commands are:

**Unite [Volume|BODY] <range> [With [Volume|BODY] <range>] [Keep]**

**Unite Body {<range> | All} [Keep]**

**Unite Body {<range> | All} [Include_mesh]**

The second form of the command unites multiple bodies in a single operation. If the all option is used, all bodies in the model are united into a single body. If the bodies that are united do not overlap or touch, the two bodies are combined into a single body with multiple volumes.
The unite command allows sheet bodies to be united with solid bodies. To disable this capability you can turn the following setting off:

**Set Unite Mixed {ON|Off}**

# decomposition

## Geometry Decomposition

Geometry decomposition is often required to generate an all-hexahedral mesh for three-dimensional solids, as fully automatic all-hex mesh generation of arbitrary solids is not yet possible in CUBIT. While geometry booleans can be used for decomposition (and are the basis of the underlying implementation of advanced decomposition tools described here), CUBIT has a webcut capability specially tuned for decomposition. It is also useful to split periodic surfaces to facilitate quad and hex meshing.

- Web Cutting
- Splitting Geometry
- Section Command
- Separating Multi-Volume Bodies
- Separating Surfaces From Bodies

web cutting

# *Web Cutting with an Arbitrary Surface*

An arbitrary "sheet" surface can also be used to web cut a body. This sheet need not be planar, and can be bounded or infinite. The following commands are used:

**Webcut {blank} with sheet {body|surface} <id> [webcut_options]**

**Webcut {blank} with sheet extended [from] surface <id> [webcut_options]**

In its first form, the command uses a sheet body, either one that is pre-existing or one formed from a specified surface. Note that in this latter case the (bounded) surface should completely cut the body into two pieces. Sheet bodies can be formed from a single surface, but can also be the combination of many surfaces; this form of web cut can be used with quite complicated cutting surfaces.
Extended sheet surfaces can also be used; in this case, the specified surface will be extended in all directions possible. Note that some spline surfaces are limited in extent, and so these surfaces may or may not completely cut the blank.

*Chop Command*

The **chop** command works similarly to a web cut command, but is faster. Given two bodies, the command will find the intersection of the two bodies, and divide the main body into a body that lies outside the intersection, and a body that lies inside the intersection.

The tool body will be deleted, unless the **keep** option is specified. The syntax of the command is:

**Chop [Volume|BODY] <id> with [Volume|BODY] <id> [keep] [nonreg]**

The **nonreg** option results in the bodies being non-regularized.

*Web Cutting with a Planar or Cylindrical Surface*

The commands used to **web cut** with a planar or cylindrical surface in CUBIT are:

- Coordinate Plane
- Planar Surface
- Plane from 3 Points
- Plane Normal to Curve
- General Plane Specification
- Cylindrical Surface
- Cone Surface

#### Coordinate Plane

In the command's simplest form, a coordinate plane can be used to cut the model, and can optionally be offset a positive or negative distance from its position at the origin.

**Webcut {Volume|Body|Group} <id_range> [With] Plane {xplane|yplane|zplane} [Offset <val>] [rotate <theta> about x|y|z <xval> <yval> <zval> [center <xval> <yval> <zval>]] webcut_options**

The cutting plane can be rotated about a user-specified axis using the **rotate** option. The center of rotation can be moved by using the **center** option.

#### Planar Surface

An existing planar surface can also be used to cut the model; in this case, the surface is identified by its ID as the cutting tool.

**Webcut {Volume|Body|Group} <id_range> [With] Plane Surface <surface_id> webcut_options**

#### Plane from 3 Points

Any arbitrary planar surface can be used by specifying three vertices that define the plane, and can optionally be offset a positive or negative distance from this plane.

**Webcut {Volume|Body|Group} <id_range> [With] Plane Vertex <vertex_1> [Vertex] <vertex_2> [Vertex] <vertex_3> [Offset <value>] webcut_options**

The plane to be used for the web cut can be previewed with the preview option in the general webcut options.

#### Plane Normal to Curve

The next command allows a user to specify an infinite cutting plane by specifying a location on a curve. The cutting plane is created such that it is normal to the curve tangent at the specified location.

**Webcut {Volume|Body|Group} <id_range> [With] Plane Normal To Curve <curve_id>**
**{Position <xval><yval><zval> | Close_To Vertex <vertex_id>} webcut_options**

**Webcut {Volume|Body|Group} <id_range> [With] Plane Normal To Curve <curve_id>**
**{Fraction <f> | Distance <d>} [[From] Vertex <vertex_id>] webcut_options**

The position on the curve can be specified as:

1. A fraction along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
2. A distance along the curve from the start of the curve, or optionally, from a specified vertex on the curve.
3. An xyz position that is moved to the closest point on the given curve.
4. The position of a vertex that is moved to the closest point on the given curve.

The point on the curve can be previewed with the Draw Location On Curve command and the plane to be used for the web cut can be previewed with the preview option in the general webcut options.

#### General Plane Specification

A webcut plane can be defined using the general plane specification options in the Specifying a Plane section of the documentation.

**Webcut {Volume|Body|Group} <id_range> [With] General Plane {options}**
**webcut_options**

#### Cylindrical Surface

Finally, a semi-infinite cylindrical surface can be used by specifying the cylinder radius, and the cylinder axis. The axis is specified as a line corresponding to a coordinate axis, the normal to a specified surface, two arbitrary points, or an arbitrary point and the origin. The "center" point through which the cylinder axis passes can also be specified.

**Webcut {Volume|Body|Group} <range> [With] Cylinder Radius <val> Axis {x|y|z|normal of surface <id>| vertex <id_1> vertex <id_2>| <x_val> <y_val> <z_val>>} [center <x_val> <y_val> <z_val>] webcut_options**

#### Cone Surface

A semi-infinite cone surface can be used by specifying the cone outer radius, and the cone inner radius. The axis is specified as a location first of where the outer radius is applied and the second location of where the inner radius is applied.

> **Webcut {Volume|Body|Group} <ids> [With] cone radius <val> <val> location {options} location {options} [Imprint] [Merge] [group_results] [preview]**

*Web Cutting by Sweeping Curves or Surfaces*

Webcutting with sweeping creates a swept tool body in the same step as the web cut operation. There are 4 general ways to **web cut** with sweeping:

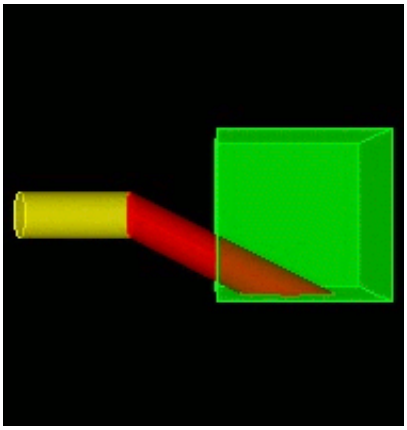
- Web Cutting by Sweeping a Surface Along a Trajectory
- Web Cutting by Sweeping a Surface About an Axis
- Web Cutting by Sweeping a Curve(s) Along a Trajectory
- Web Cutting by Sweeping a Curve(s) About an Axis

#### Web Cutting by Sweeping a Surface Along a Trajectory

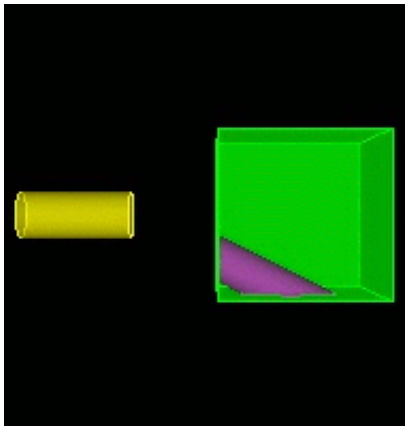
This command allows one or more surfaces to be swept, creating a volume that is used for the web cut. If more than one surface is specified, the surfaces must contain coincident curves. The surfaces are swept along a direction and some distance or perpendicular and some distance or along a curve. For best results the curve to sweep the surface along should intersect one of the surfaces. The **through_all** option will sweep the surfaces along the trajectory far enough so as to intersect all input bodies. The **stop surface <id>** option is used to identify a surface at which the sweep will stop. If using this option when sweeping along a curve, the sweep will stop at the first place possible. The **up_to_next** option indicates that the user wants to web cut with only the first water tight volume that forms as a result of the intersection between sweep and union of all blank bodies. The **[Outward|Inward]** options specify a sweeping direction that is either INTO the volume or OUT from the volume.

**Webcut {Volume|Body|Group} <range> Sweep Surface <id_range> {Vector <x> <y> <z> [Distance <distance>] | Along Curve <id>} [Through_all | Stop Surface <id> | Up_to_next ] [webcut_options]**

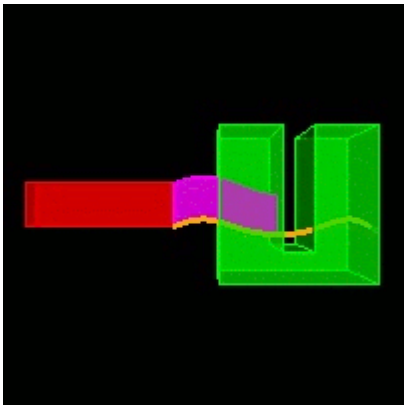
**Webcut {Volume|Body|Group} <id> Sweep Surface <id_range> Perpendicular {Distance <distance> | Through_all | Stop Surface <id>} [OUTWARD|Inward] [webcut_options]**



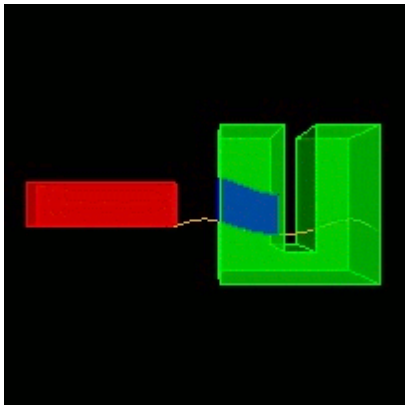
sweeping a surface in a direction



resultant web cut



along a curve to a stop surface



resultant web cut

**Figure 1. Examples of web cutting with swept surfaces**

#### Web Cutting by Sweeping a Surface About an Axis

This command allows a one or more surfaces to be swept, creating a volume that is used for the web cut. If more than one surface is specified, the surfaces must contain coincident curves. The surface is swept about a user-defined axis or about one of the x y z coordinate axes and a specified angle. The **stop surface <id>** option is used to identify a surface at which the sweep will stop. The **up_to_next** option indicates that the user wants to web cut with only the first water tight volume that forms as a result of the intersection between sweep and union of all blank bodies. For these 2 options to work correctly the user must specify an angle large enough for the rotation to traverse the **stop surface** or the **up_to_next** surface.

**Webcut {Volume|Body|Group} <id> Sweep Surface <id_range> {Axis <xpoint ypoint zpoint xvector yvector zvector> | Xaxis | Yaxis | Zaxis } Angle <degrees> [Stop Surface <id> | Up_to_next] [webcut_options]**

#### Web Cutting by Sweeping a Curve(s) Along a Trajectory

This command allows a curve(s) to be swept, creating a surface that is used for the web cut. If multiple curves are specified, they must share vertices and form a continuous path. The curve(s) is swept along a direction and some distance or along another curve. If sweeping a curve(s) along another curve, for best results the curve(s)-to-swept and the curve to sweep along should intersect at some point. The **stop surface <id>** option is used to identify a surface at which the sweep will stop. If using this option when sweeping along a curve, the sweep will stop at the first place possible. The **through_all** option will sweep the curve(s) along the trajectory far enough so as to intersect all input bodies. For the web cut to be successful, the swept curve(s) must completely traverse a portion of a blank body(s), cutting off a complete piece of the blank body(s). Option **through_all** should not be used when defining the web cut with a vector and a distance or along a curve.

**Webcut {Volume|Body|Group} <id> Sweep Curve <id_range> {Vector <x> <y> <z> [Distance <distance>| Along curve <id>] } [Through_all | Stop Surface <id>] [webcut_options]**

#### Web Cutting by Sweeping a Curve(s) About an Axis

This command allows a curve to be swept, creating a surface that is used for the web cut. If multiple curves are specified, they must share vertices and form a continuous path. The curve(s) is swept about a user-defined axis or about one of the x y z coordinate axes and a specified angle. For the web cut to be successful, the swept curve(s) must completely traverse a portion of a blank body(s), cutting off a complete piece of the blank body(s). The **stop surface <id>** option is used to identify a surface at which the sweep will stop. For this option to work correctly the user must specify an angle large enough for the rotation to traverse the **stop surface**.

**Webcut {Volume|Body|Group} <id> Sweep Curve <id_range> {Axis <xpoint ypoint zpoint xvector yvector zvector> | Xaxis | Yaxis | Zaxis } Angle <degrees> [Stop Surface <id>] [webcut_options]**

### *Web Cutting using a Tool or Sheet Body*

Any existing body in the geometric model can be used to cut other bodies; the command to do this is:

**Webcut {blank} tool [body] <id> [webcut_options]**

This simply uses the specified tool body in a set of boolean operations to split the blank into two or more pieces.
Another form of the command cuts the body list with a temporary sheet body formed from the curve loop. This is the same sheet as would be created from the command Create Surface Curve <id_list>.

**Webcut {Body|Group} <id_range> [With] Loop [Curve] <id_range> NOIMPRINT|Imprint] [NOMERGE|Merge] [group_results]**

**Webcut {Volume|Body|Group} <id_range> [With] Bounding Box {Body|Volume|Surface|Curve|Vertex <id_range>} [Tight] [[Extended] {Percentage|Absolute} <val>] [{X|Width} <val>] [{Y|Height} <val>] [{Z|Depth} <val>]] NOIMPRINT|Imprint] [NOMERGE|Merge] [group_results]**

The final form of this command cuts a body with the bounding box of another entity. This bounding box may be tight or extended.

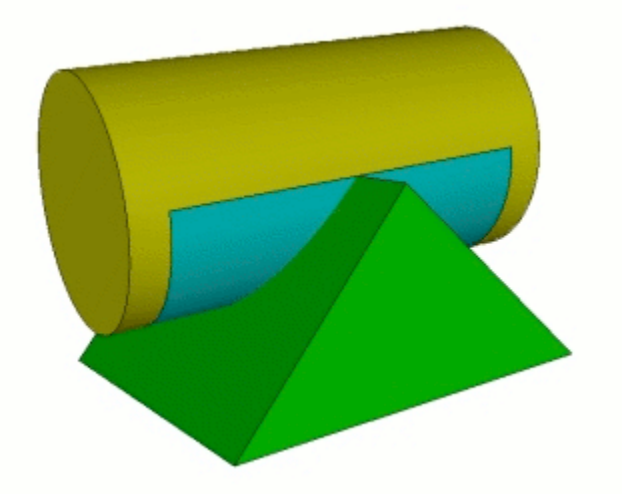


**Figure 1. Cylinder cut with bounding box of prism.**

*Web Cutting*

The term "web cutting" refers to the act of cutting an existing body or bodies, referred to as the "blank", into two or more pieces through the use of some form of cutting tool, or "tool". The two primary types of cutting tools available in CUBIT are surfaces (either pre-existing surfaces in the model or infinite or semi-infinite surfaces defined for web cutting), or pre-existing bodies.
The various forms of the web cut command can be classified by the type of tool used for cutting. These forms are described below, starting with the simplest type of tool and progressing to more complex types.

- Web Cutting Using the Chop Command

- Web Cutting Using Planar or Cylindrical Surface
- Web Cutting with Arbitrary Surface
- Web Cutting Using Tool or Sheet Body
- Web Cutting by Sweeping Curves or Surfaces
- Web Cutting Options

**General Notes**

The primary purpose of web cutting is to make an existing model meshable with the hex meshing algorithms available in CUBIT. While web cutting can also be used to build the initial geometric model, the implementation and command interface to web cutting have been designed to serve its primary purpose. Several important things to remember about web cutting are as follows:

- The geometric model should be checked for integrity (using imprinting and merging) before starting the decomposition process. This makes the checking process easier, since there are fewer bodies and surfaces to check. Once the model passes that initial integrity check, it is rare that decompositions using web cut will result in a model that does not also pass the same checks.
- The use of the Imprint option can in cases save execution time, since it limits the scope of the imprint operations and thereby works faster. The alternative is performing and Imprint All on the pieces of the model after all decompositions have been completed; this operation has been made much faster in more current releases of CUBIT, but will still take a noticeable amount of time for complicated models.
- While the web cut commands make it very simple to cut your model into very many pieces, we recommend that the user restrict the decomposition they perform to only that necessary for meshability or for obtaining an acceptable mesh. Having more volumes in the model may simplify individual volumes, but may not always result in a higher quality mesh; it will always increase the run time and complexity of the meshing task.
- When the web cut command is executed the associated geometry will be regularized. This behavior can be changed, see geometry booleans.
- Web cutting volumes will automatically separate parent bodies as well. This behavior can also be changed, see Separating Multi-Volume Bodies.
- If a geometric entity got split after the webcut operation, then the notesets/sidesets/blocks applied on that initial geometric entity will be carried over to the split entities.

The Decomposition Tutorials and the Power Tools Tutorial contain some examples that demonstrate the use of web cutting operations.

*Web Cutting Options*

The following options can be used with all web cut commands:

**[NOIMPRINT|Imprint [include_neighbors] ]:** In its default implementation, web cutting results in the pieces not being imprinted on one another; this option forces the code to imprint the pieces after web cutting. The include_neighbors option will also imprint adjacent bodies.
**[NOMERGE|Merge]:** By default, the pieces resulting from an imprint are manifold; specifying this option results in a merge check for all surfaces in the pieces resulting from the web cut.
**[Group_results]:** The various pieces resulting from the previous command are placed into a group named `webcut_group'.
**[Preview]:** This option will preview the web cutting plane without executing the command.

splitting geometry

# *Split Curve*

The Split Curve command will split a curve without the need for geometry creation (unlike imprinting). The syntax is shown below.

```
Split Curve <id> [location on curve options] [Merge] [Preview]
```

To split a curve, simply specify a location or a location on curve (see location specification). Using the **Preview** keyword will draw the splitting location on the curve. The **Merge** keyword will merge any topology that contains the newly created vertex.

# *Split Periodic Surfaces*

Solids which contain periodic surfaces include cylinders, torii and spheres. Splitting periodic surfaces can in some cases simplify meshing, and will result in curves and surfaces being added to the volume. The command used to split periodic surfaces is:

```
Split Periodic Body <id_range|all>
```

This command splits all periodic surfaces in a body or bodies.

# *Split Surface*

The Split Surface command divides one or more surfaces into multiple surfaces. The command results are similar to imprint with curve. However, curve creation is not necessary for splitting surfaces. Three primary forms of the command are available.

- Split Across
- Split Extend

- Split (Automatically)
- Split Skew

The first form splits a single surface using locations while the second splits by extending a surface hard-line until it hits a surface boundary. The split automatic splits either a single surface or a chain of surfaces in an automatic fashion.

#### Split Across

Two forms of Split Across are available:

**Split Surface <id> Across [Pair] Location <options multiple locs> [Preview [Create]]**

**Split Surface <id> Across Location <multiple locs> Onto Curve <id> [Preview] Create]]**

This command splits a surface with a spline projection through multiple locations on the surface. See Location, Direction, and Axis Specification for a detailed description of the location specifier. Figure 1 shows a simple example of splitting a single surface into two surfaces. A temporary spline was created through the three specified locations (Vertex 5 6 7), and this curve was used to split the surface.

**split surface 1 across location vertex 5 6 7**

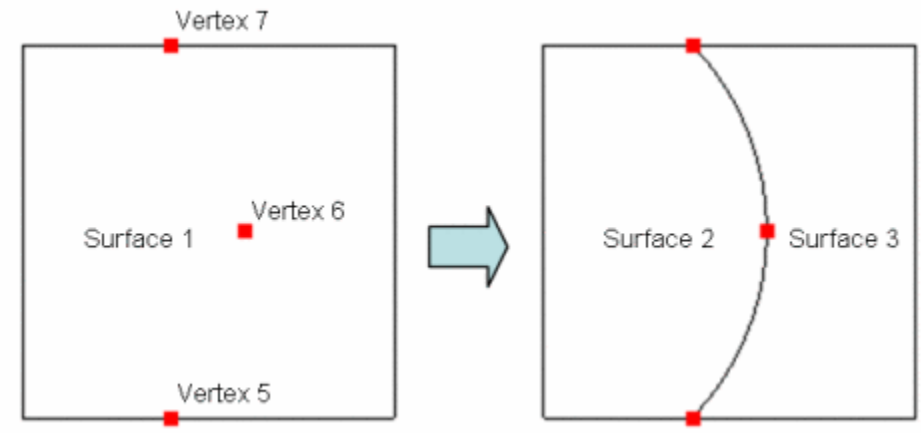



**Figure 1 - Splitting Across with Multiple Locations**

The **Pair** keyword will pair locations to create multiple surface splitting curves (each defined with two locations). An even number of input locations is required. Figure 2 shows an example:

**split surface 1 across pair vertex 5 7 6 8**

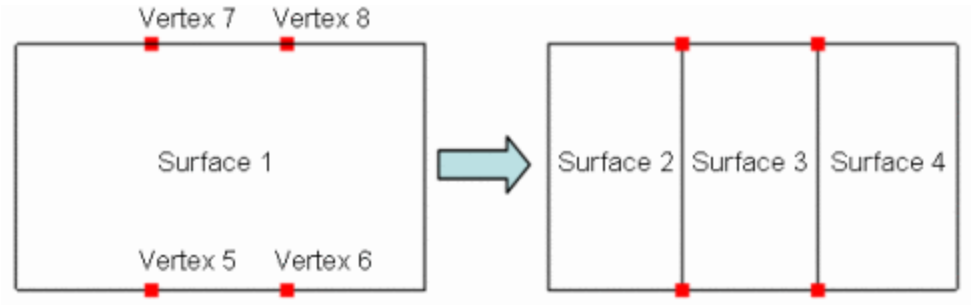



**Figure 2 - Splitting Across with Pair Option**

The **Preview** keyword will show a graphics preview of the splitting curve. If the **Create** keyword is also specified, a free curve (or curves) will be created - these are the internal curves that are used to imprint the surfaces.
The **Onto Curve** format of the command takes one or more locations on one side of the surface and projects them onto a single curve on the other side of the surface. Figure 3 shows an example:

**split surface 1 across vertex 5 6 onto curve 4**

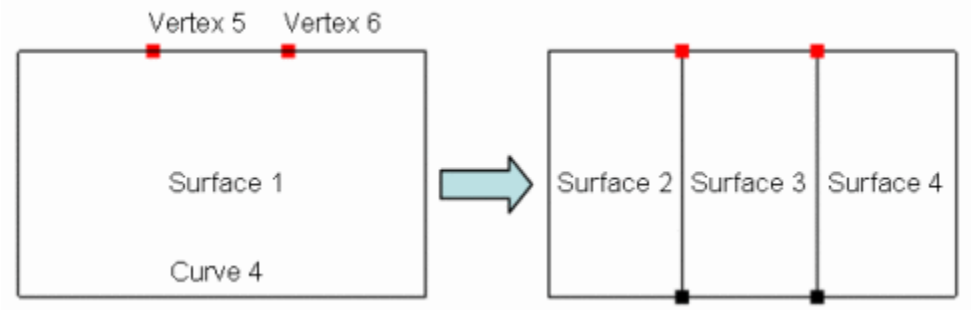



**Figure 3 - Splitting Across with Onto Curve**

#### Split Extend

The Split Extend function can be called with the following command:

**Split Surface <id_list> Extend [Vertex <id_list> | AUTO] [Preview [Create]]**

With the following settings:

**Set Split Surface Extend Normal {on|OFF}**

**Set Split Surface Extend Gap Threshold <val>**

**Set Split Surface Extend Tolerance<val>**

This command splits a surface by extending a surface hard-line until it hits a surface boundary. Figure 4 shows a simple example of extending a curve. The hard-line curve was extended from the specified vertex until it hit the surface boundary.

**split surface 1 extend vertex 2**

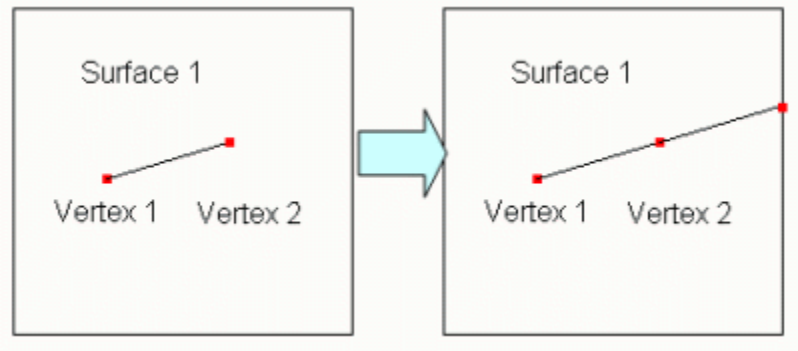



**Figure 4 - Splitting by Extending Hard-line**

The **auto** keyword will search for all hard-lines and extend them according to the Split Surface Extend settings. Figure 5 shows an example:

**split surface 1 extend auto**

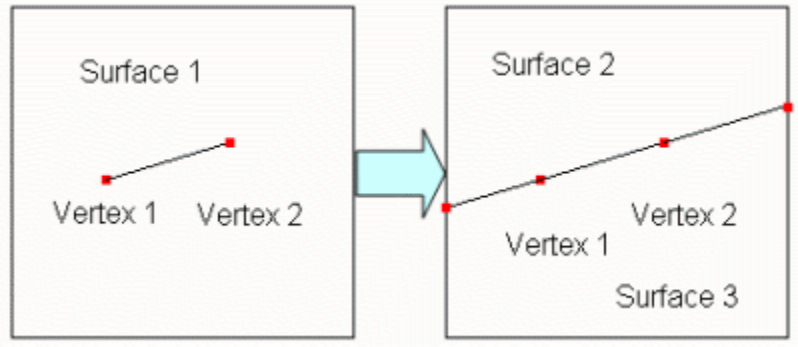



**Figure 5 - Splitting by Extending with Auto Option**

The **preview** keyword will show a graphics preview of the splitting curve. If the **create** keyword is also specified, a free curve (or curves) will be created - these are the internal curves that are used to imprint the surfaces.

The **normal** setting can be turned on or off. When it is on, Cubit will attempt to extend the hard-line so that it is normal to the curve it will intersect. An example of this is in Figure 6:

**set split surface normal on**
**split surface 1 extend vertex 2**

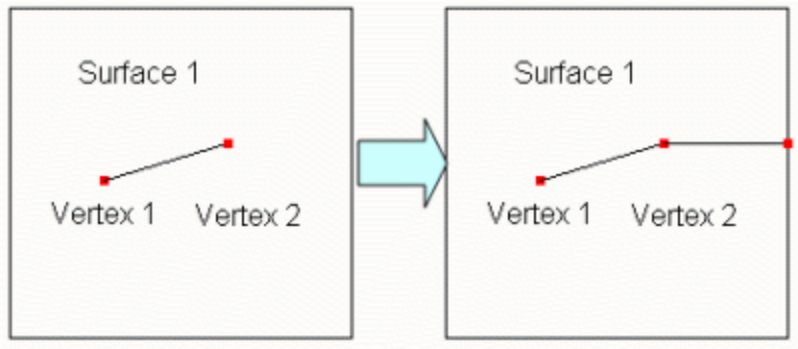



**Figure 6 - Splitting by Extending a Hard Line with Normal Setting ON**

Cubit uses the **gap threshold** to decide whether or not to extend a hard-line when the user specifies *auto*. If the distance between a vertex on a hard-line and the curve it will hit is greater than the gap threshold, then Cubit will not extend that hard-line. The default value is INFINITY, and can be set to any value. To reset the value back to INFINITY, set the gap threshold to -1.0. **Note: This setting only applies when using the keyword *auto***. An example of using the gap threshold is shown in Figure 7:

```
set split surface gap threshold 2.0
split surface 1 extend auto
```

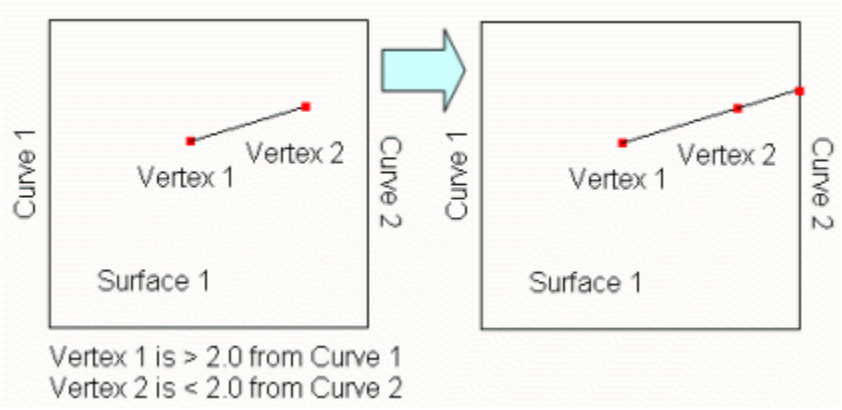



**Figure 7 - Extending Hard-lines with Gap Threshold = 2.0.**
**(Notice Vertex 1 was not extended because it exceeded the gap threshold)**

The **tolerance** setting can be used to avoid creating short curves on the surface boundary. If Cubit tries to extend a hard-line that comes within tolerance of a vertex, it will instead snap the extension to the existing vertex. An example of this is shown in Figure 8:

```
set split surface tolerance 1.0
split surface 1 extend vertex 2
```

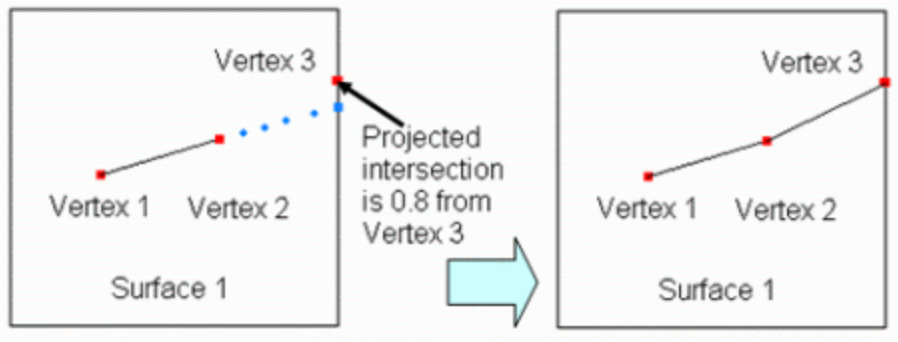



**Figure 8 - Extending Hard-lines with Tolerance**
**(Notice the extension snapped to Vertex 3)**

#### Split (Automatically)

This form of the command splits a single surface or a chain of surfaces in an automatic fashion. It is most convenient for splitting a fillet or set of fillets down the middle - oftentimes necessary to prepare for mesh sweeping. These surfaces cannot have multiple curve loops.

**Split Surface <id_list> [Corner Vertex <id_list>] [Direction Curve <id>] [Segment|Fraction|Distance <val> [From Curve <id>]] [Through Vertex <id_list>] [Parametric <on|OFF>] [Tolerance <val>] [Preview [Create]]**

- Logical Rectangle
- Split Orientation
- Corner Vertex <id_list>
- Direction Curve <id>
- Segment|Fraction|Distance <val> [From Curve <id>]
- Through Vertex <id_list>
- Parametric <on|OFF>
- Tolerance <val>
- Preview [Create]
- Settings (Tolerance, Parametric, Triangle)

The volume shown in Figure 9 was quickly prepared for sweeping by splitting the fillets and specifying sweep sources as shown (with the sweep target underneath the volume). The surface splits are shown in blue.

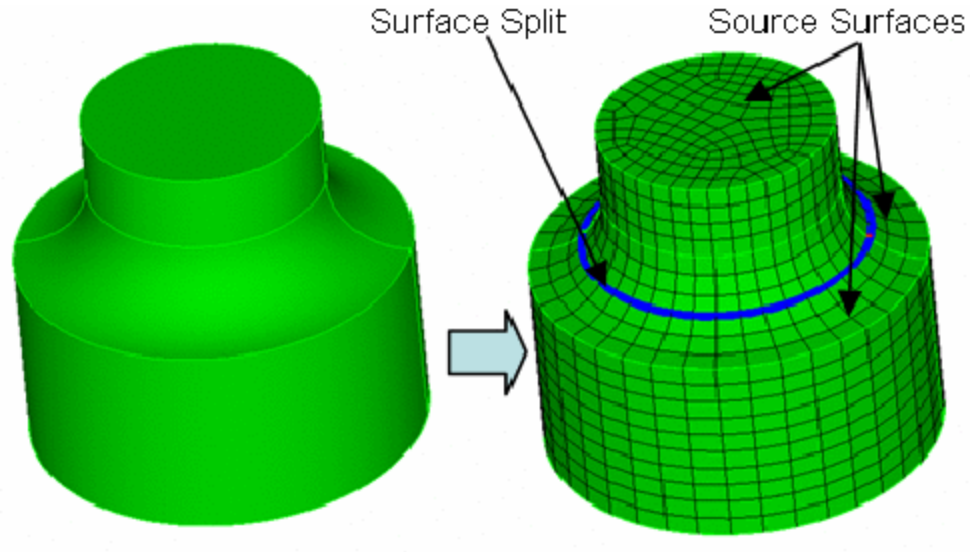



**Figure 9 - Splitting Fillets to Facilitate Sweeping**

Each surface is always split with a *single* curve along the length of the surface (or multiple single curves if the Segment option is used). The splitting curve will either be a spline, arc or straight line.

**Logical Rectangle**

The Split Surface command analyzes the selected surface or surface chain to find a *logical rectangle*, containing four logical sides and four logical corners; each side can be composed of zero, one or multiple curves. If a single surface is selected (with no options), the logical corners will be those closest to 90 and oriented such that the surface will be ***split parallel to the longest aspect ratio*** of the surface. If a chain of surfaces is selected, the logical corners will include the two corners closest to 90 on the starting surface of the chain and the two corners closest to 90 on the ending surface of the chain (***the split will always occur along the chain***).

In Figure 10, the logical corners selected by the algorithm are Vertices 1-2-5-6. Between these corner vertices the logical sides are defined; these sides are described in Table 1. The default split occurs from the center of Side 1 to the center of Side 3 (parallel to the longest aspect ratio of the surface), and is shown in blue.

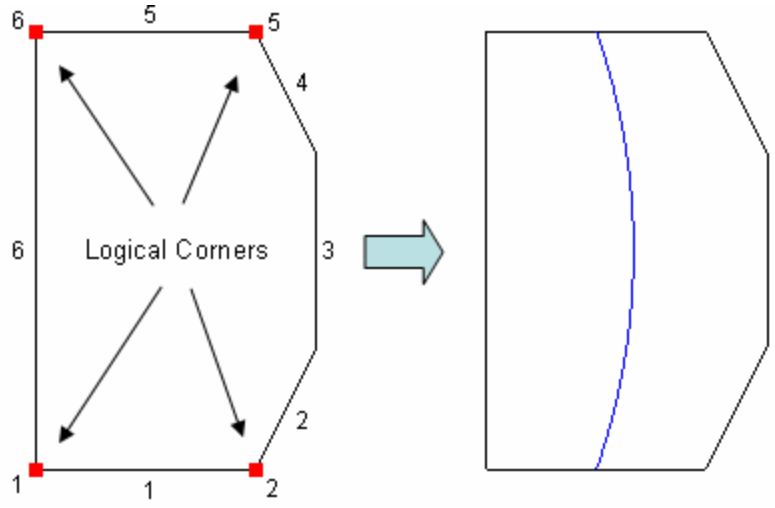



**Figure 10 - Split Surface Logical Properties**

**Table 1. Listing of Logical Sides for Figure 10**

| Logical Side | Corner Vertices | Curve Groups |
|---|---|---|
| 1 | 1-2 | 1 |
| 2 | 2-5 | 2,3,4 |
| 3 | 5-6 | 5 |
| 4 | 6-1 | 6 |

Figure 11 shows a surface along with 2 possibilities for its logical rectangle and the resultant splits.

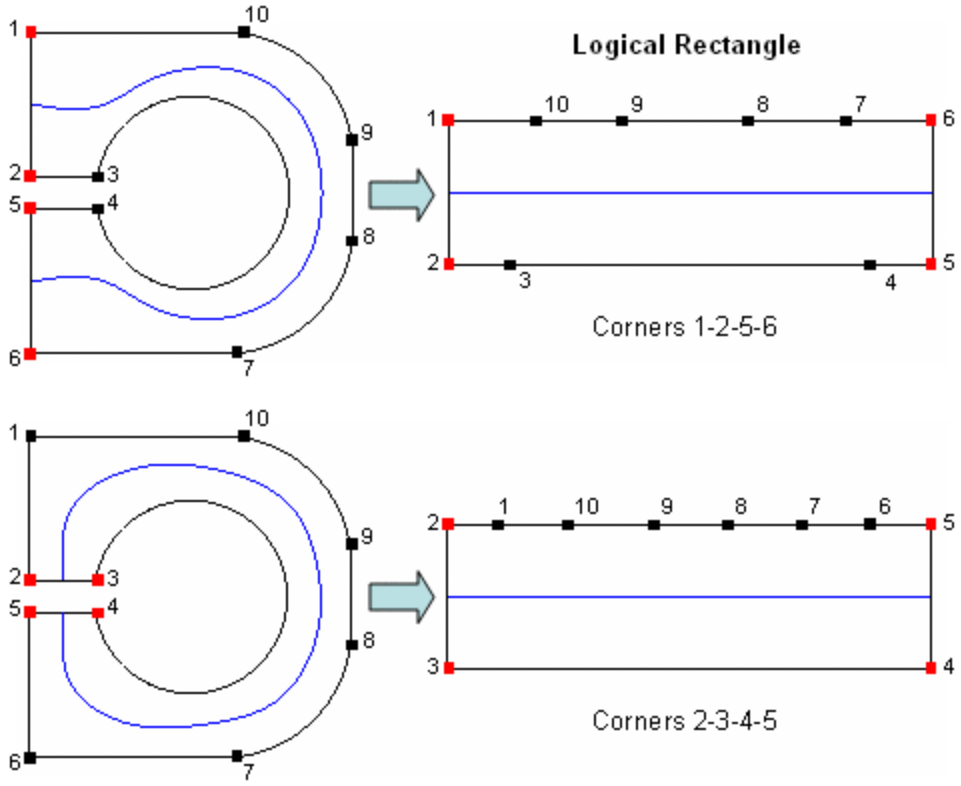
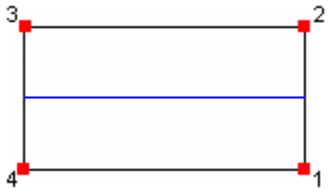
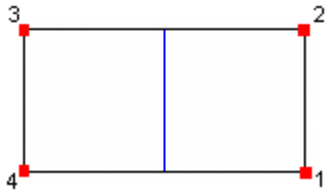
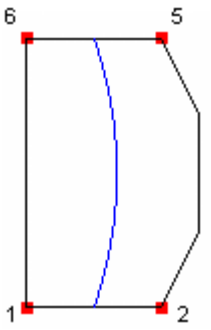
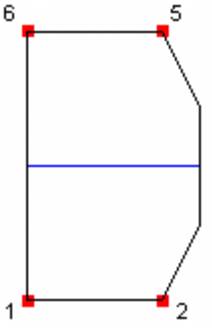
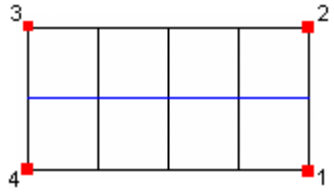
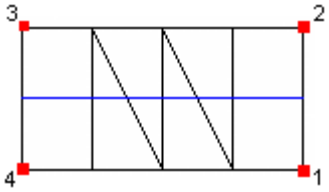
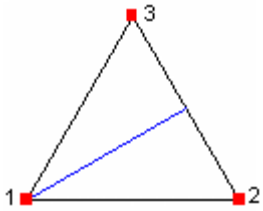
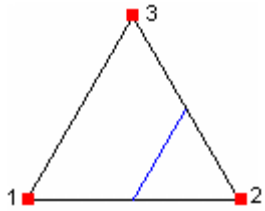


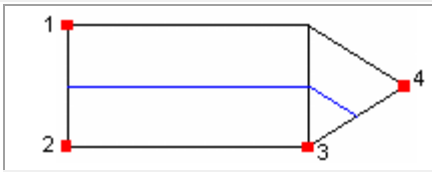
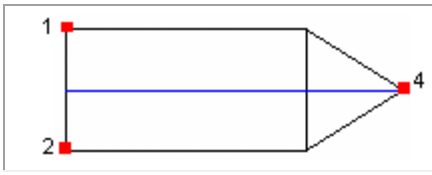
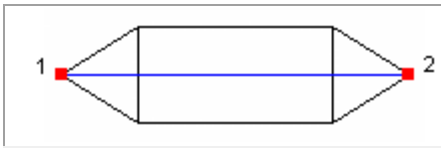
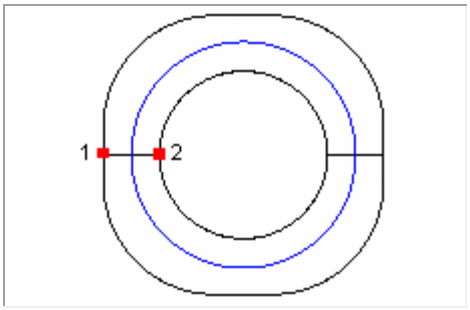

**Figure 11 - Different Possible Logical Rectangles for Same Surface**

Table 2 shows various surfaces and the resultant split based on the automatically detected or selected logical rectangle. Note that surfaces are always traversed in a counterclockwise direction.

**Table 2 - Sample Surfaces and Logical Rectangles**

| Surface(s) (Resultant Split in Blue) | Ordered Corners (to form the *Logical Rectangle*) |
|---|---|
|  | 1-2-3-4<br><br>(using aspect ratio) |
|  | 4-1-2-3<br><br>(user selected) |

| | |
|---|---|
|  | 1-2-5-6 |
|  | 2-5-6-1 |
|  | 1-2-3-4<br>(split is always along the chain) |
|  | 1-2-3-4<br>(notice triangular surfaces along the chain) |
|  | 1-1-2-3<br>(note side 1 of the logical rectangle is collapsed; side 3 is from vertex 2 to 3) |
|  | 1-2-2-3<br>(note side 2 of the logical rectangle is collapsed) |

| | |
|---|---|
|  | 1-2-3-4 |
|  | 1-2-4-4 |
|  | 1-1-2-2 |
|  | 1-1-2-2<br>(selected automatically) |

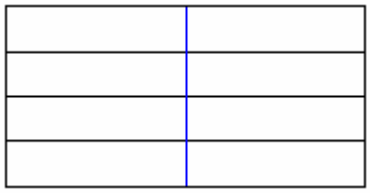
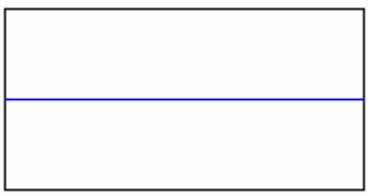
**Split Orientation**

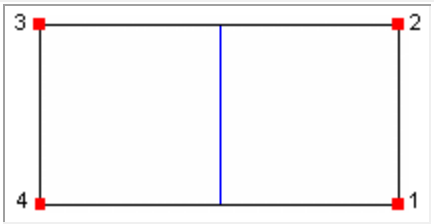
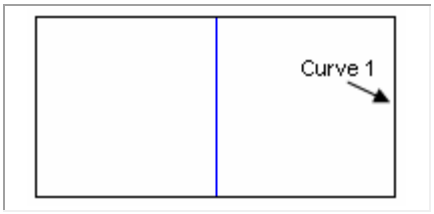
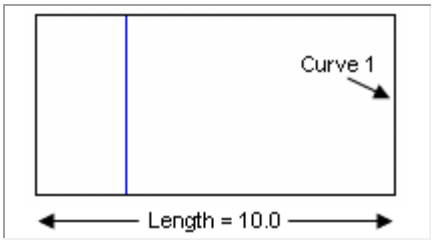
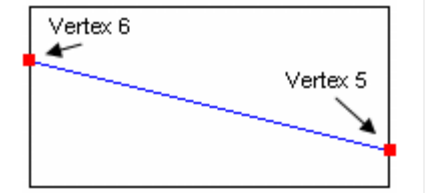
If a chain of surfaces are split, the surfaces will always be split along the chain. The command will not allow disconnected surfaces.

For a single surface, the split direction logic is a bit more complicated. If no options are specified, the surface aspect ratio determines the split direction - the surface will be split parallel to the longest aspect ratio side through the midpoint of each curve. This behavior can be overridden by the order the Corner vertices are selected (the split always starts on the side between the first two corners selected), the Direction option, the From Curve option, or the Through Vertex list.

Table 3 shows examples of the various split orientation methods. These options are explained in more detail in the sections below.

**Table 3 - Split Orientation Methods**

| Surface Example | Split Orientation Method |
|---|---|
|  | Multiple surfaces are *always* split along the chain |
|  | Parallel to longest surface aspect ratio (default) |

| | |
|---|---|
|  | Corner Vertex 4 1 2 3<br>(split always starts on side 1 of the logical rectangle) |
|  | Direction Curve 1 |
|  | From Curve 1 Fraction .75<br>or<br>From Curve 1 Distance 7.5 |
|  | Through Vertex 5 6 |

**Corner Specification**

The **Corner** option allows you to specify corners that form *logical rectangle* the algorithm uses to orient the split on the surface. When analyzing a surface to be split, the software automatically selects the corners that are closest to 90. The Preview option displays the automatically selected corners in red. Sometimes incorrect corners are chosen, so you must specify the desired corners yourself. The split always starts on the side between the first two corners selected and finishes on the side between the last two corners selected. Figure 12 shows a situation where the user had to select corners to get the desired split.

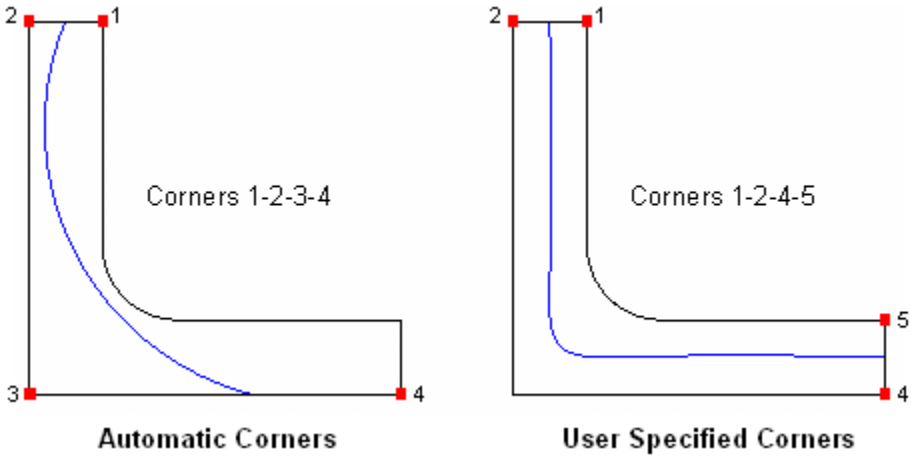
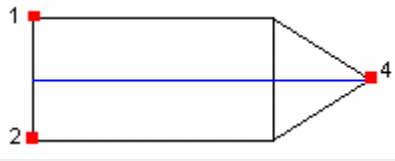
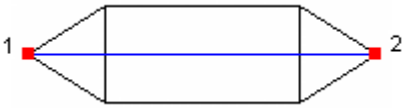



**Figure 12 - Selecting the Desired Corners**

The split can be directed to the tip of a triangular shaped surface by selecting that corner vertex twice (at the start or end of the corner list) when specifying corners, creating a zero-length side on the logical rectangle. A shortcut exists whereas if you specify only 3 corner vertices, the zero-length side will be directed to the first corner selected. If you specify only 2 corner vertices, a zero-length side will be directed to both the first and second corner you select. Table 4 shows these examples. Note the software will automatically detect triangle corners based on angle criteria - the corner selection methods for zero-length sides explained in this section need only be applied if the angles are outside of the thresholds specified in the Set Split Surface Auto Detect Triangle settings.

**Table 4 - Selecting Corners to Split to Triangle Tips**

| Surface | Corner Specification |
|---|---|
|  | 1-2-4-4- or 4-4-1-2<br>or<br>4-1-2 (shortcut method) |
|  | 1-1-2-2 or 2-2-1-1<br>or<br>1-2 or 2-1 (shortcut method) |

**Direction**

The **Direction** option allows you to conveniently override the default split direction on a single surface. Simply specify a curve from the logical rectangle that is parallel to the desired split direction. If Corners are also specified, the Direction option will override the split orientation that would result from the specified corner order. The Direction option is not valid on a chain of surfaces. Figure 13 shows an example.

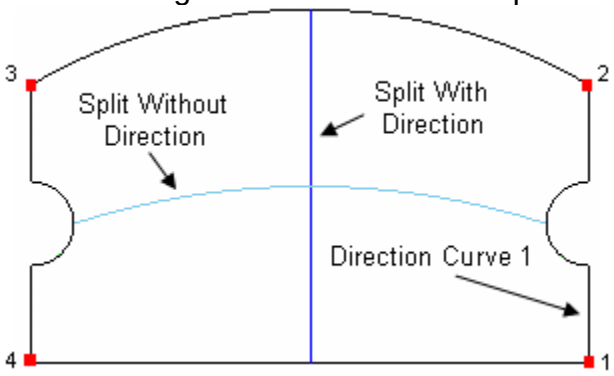


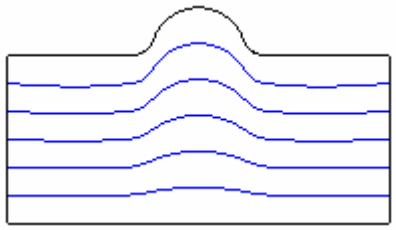
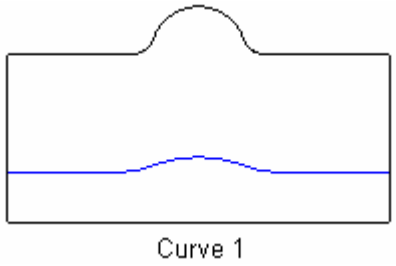
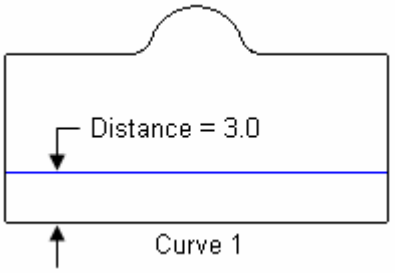

**Figure 13 - Direction Specification Overrides Corner Order**

**Segment|Fraction|Distance**

The **Segment** option allows you to split a surface into 2 or more segments that are equally spaced across the surface. The **Fraction** option allows you to override the default 0.5 fractional split location. The **Distance** option allows you to specify the split location as an absolute distance rather than a fraction. By specifying a **From Curve**, you can indicate

which side of the logical rectangle to base the segment, fraction or distance from (versus a random result). Table 5 gives examples of these options.

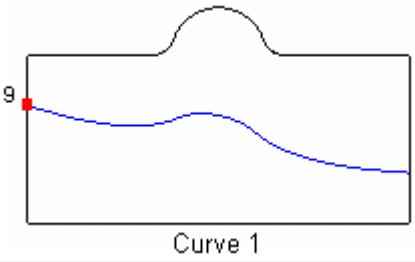
**Table 5 - Segment, Fraction, Distance Examples**

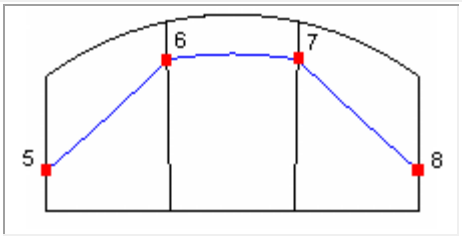
| Surface | Command Options |
|---|---|
|  | Segment 6 From Curve 1 |
|  | Fraction .3 From Curve 1 |
|  | Distance 3 From Curve 1 |

**Through Vertex**

The **Through Vertex** option forces the split through vertices on surface boundaries perpendicular to the split direction. Use this option if the desired fraction is not constant from one end of the surface to another or if a split would otherwise pass very close to an existing curve end resulting in a short curve. Through vertices can be used in conjunction with the Fraction option - the split will linearly adjust to pass exactly through the specified vertices. It is not valid with the Segment option. The maximum number of Through Vertices that can be specified is equal to the number of surfaces being split plus one. The selected vertices can be free, but must lie on the perpendicular curves. Table 6 gives several examples.

**Table 6 - Through Vertex Examples**

| Surface(s) | Command Options |
|---|---|
|  | Fraction .3 From Curve 1 Through Vertex 9 |

| | |
|---|---|
|  | Through Vertex 5 6 7 8 |

**Parametric**

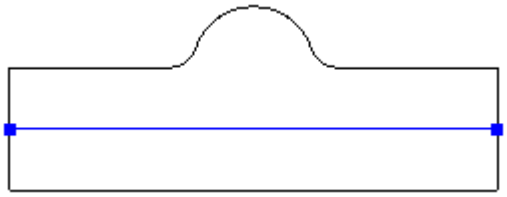
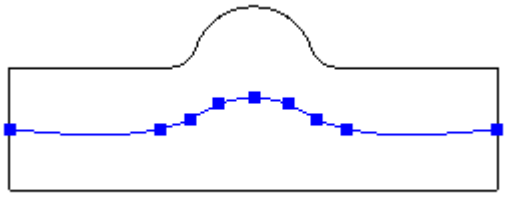
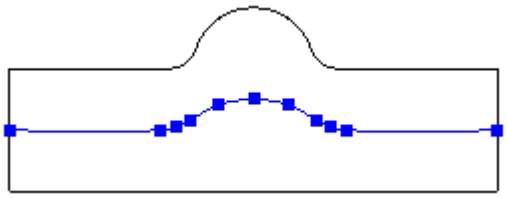
By default, split locations are calculated in 3D space and projected to the surface. As an alternative, split locations can be calculated directly in the surface parametric space. In rare instances, this can result in a smoother or more desirable split. The command option **Parametric {on|Off}** can be used to split the given surfaces in parametric space. Alternatively, the default can be overridden with the Set Split Surface Parametric {on|OFF} command.

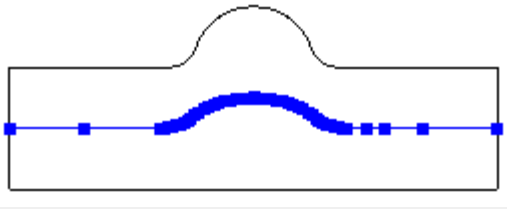
**Tolerance**

A single absolute tolerance value is used to determine the accuracy of the split curves. A smaller tolerance will force more points to be interpolated. The tolerance is also used when detecting an analytical curve (e.g., an arc or straight line) versus a spline. A looser tolerance will result in more analytical curves. The default tolerance is 1.0. The command option **Tolerance <val>** can be used to split the given surfaces using the given tolerance. Alternatively, the default tolerance can be overridden with the Set Split Surface Tolerance <val> command.

It is recommended to use the largest tolerance possible to increase the number of analytical curves and reduce the number of points on splines, resulting in better performance and smaller file sizes. The Preview option displays the interpolated curve points. Table 7 shows the effect of the tolerance for a simple example.

**Table 7 - Effect of Tolerance on Split Curve**

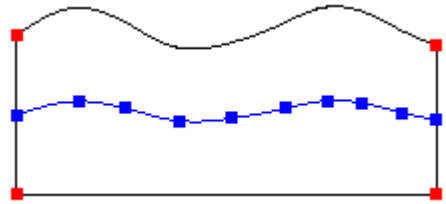
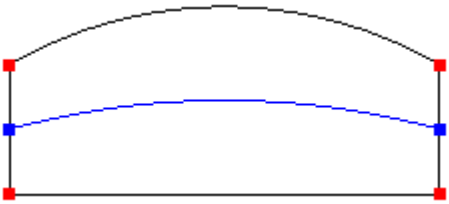
| Surface | Tolerance |
|---|---|
|  | 2.0 |
|  | 1.0 |
|  | 0.5 |

| | |
|---|---|
|  | 0.01 |

**Preview**

The **Preview** keyword will show a graphics preview (in blue) of the splitting curve (or curves) and the corner vertices (in red) selected for the logical rectangle. The curve preview includes the interpolated point locations that define spline curves. Note that if no points are shown on the interior of the curve, it means that the curve is an analytical curve (line or arc). If the **Create** keyword is also specified, a free curve (or curves) will be created - these are the internal curves that are used to imprint the surfaces. Table 8 shows some examples.

**Table 8 - Graphics Preview**

| Surface | Curve Type |
|---|---|
|  | Spline |
|  | Arc (no preview points shown on interior of curve) |

**Settings**

This section describes the settings that are available for the automatic split surface command. To see the current values, you can enter the command **Set Split Surface**, optionally followed by the setting of interest (without specifying a value).

**Set Split Surface Tolerance <val>**

This sets the default tolerance for the accuracy of the split curves. See the Tolerance section for more information.

**Set Split Surface Parametric {on|OFF}**

This sets the default for whether surfaces are split in 3D (default) or in parametric space. See the Parametric section for more information.

**Set Split Surface Auto Detect Triangle {ON|off}**

**Set Split Surface Point Angle Threshold <val>**

**Set Split Surface Side Angle Threshold <val>**

The split surface command automatically detects triangular shaped surfaces as explained in the section on Corners. This behavior can be turned off with the setting above. Two thresholds are used when detecting triangles - the **Point Angle** threshold and the **Side Angle** threshold, specified in degrees. Corners with an angle below the Point Angle threshold are considered for the tip of a triangle (or the collapsed side of the logical rectangle). Corners within the Side Angle threshold of 180 are considered for removal from the logical rectangle. In order for a triangle to actually be detected, corners for both the point and side criteria must be met. The default Point Angle threshold is 45, and the default Side Angle threshold is 27. Figure 14 provides an illustration.

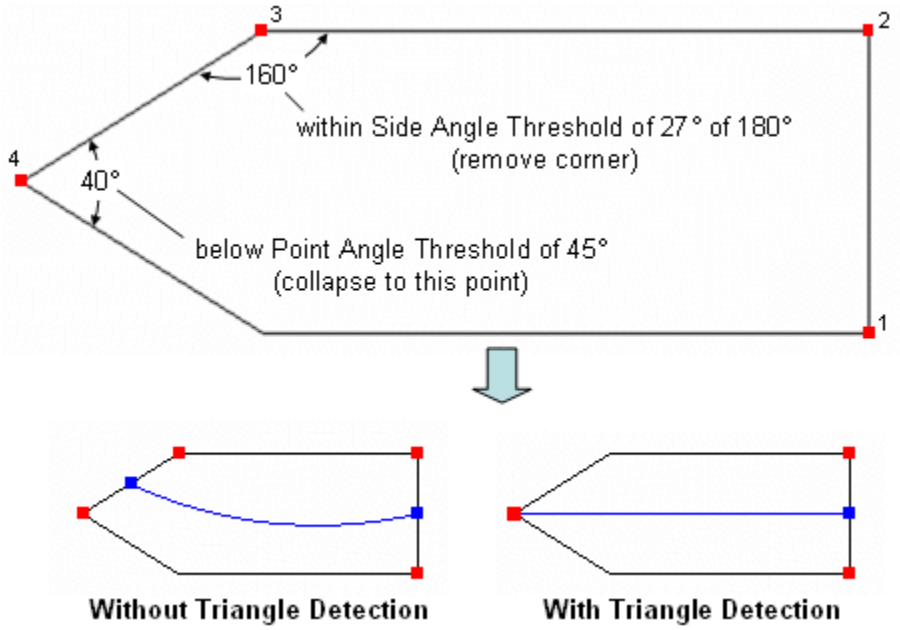



**Figure 14 - Triangle Detection Settings**

#### Split Skew

The Split Skew function can be called with the following command:

**Split Surface <id_list> Skew [Preview] [Create]**

This command will split a surface or list of surfaces in a logical way to reduce the amount of skew in a quadrilateral mesh. This function uses the control skew algorithm to determine where to make these logical splits. Users should note that Split Skew can only be utilized effectively on surfaces that lend themselves to a structured meshing scheme.

These surfaces cannot have multiple curve loops. Figure 15 shows a simple example of a surface being split.

**split surface 1 skew**

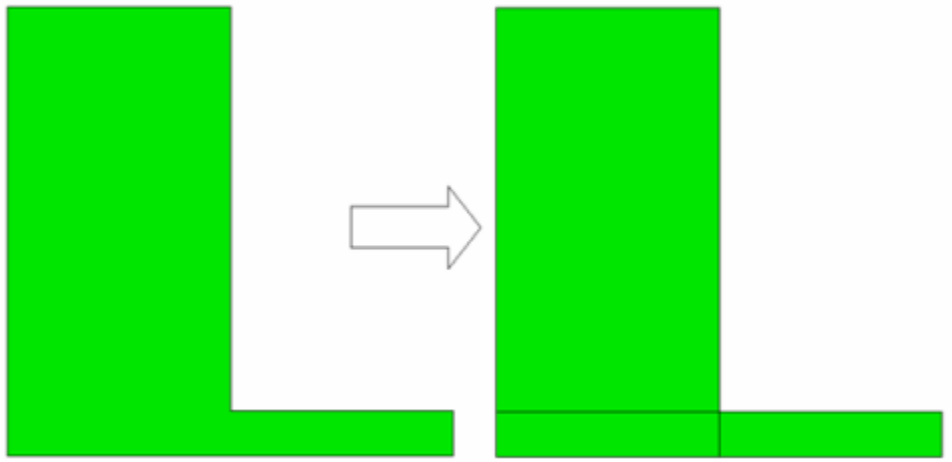


**Figure 15. Split Skew applied to an L-shaped surface**

The **Preview** keyword will show a graphics preview of the splitting curves. If the **Create** keyword is also specified, free curves will be created.

# *Splitting Geometry*

The Split command divides curves or surfaces into multiple entities. The command results are similar to imprinting. However, vertex and/or curve creation is not necessary for the split command.

- Split Curve
- Split Surface
- Split Periodic Surfaces

## Section Command

This command will cut a body or group of bodies with a plane, keeping geometry on one side of the plane and discarding the rest. The syntax for this command is:

**Section {Body|Group} <id_range> [With] {Xplane|Yplane|Zplane} [Offset <value>] [NORMAL|Reverse] [Keep]**

**Section {Body|Group} <id_range> With Surface <id> [NORMAL|Reverse] [Keep]**

In the first form, the specified coordinate plane is used to cut the specified bodies. The offset option is used to specify an offset from the coordinate plane. In the second form, an existing (planar) surface is used to section the model. In either case, the reverse keyword results in discarding the positive side of the specified plane or surface instead of the other side. The keep option results in keeping both sides; the section command used with this option is equivalent to webcutting with a plane.

### Separating Surfaces from Bodies

The separate surface command is used to separate a surface from a sheet body or a solid body. The command is:

**Separate Surface <range>**

Separating a surface from a solid body will create a "hole" in the solid body. Thus the solid body will become a sheet body. The newly separated surface will be also sheet body, but it will have a different id. Multiple surfaces can be separated from a body at the same time, but each separated surface will result in a distinct sheet body, as if the command had been performed on each surface individually.

### Separating Multi-Volume Bodies

The separate and split commands are used to separate a body with multiple volumes into a multiple bodies with single volumes. The commands are:

**Separate {Body|Volume} <id_range|all>**

and

**Split {Body|Volume} <id_range|all>**

Only very rarely will either of these commands be needed. They are provided for the occasional instance that a multi-volume body is found. These commands are interchangeable.

Another related command allows the user to control the separation of bodies after webcutting. In most instances the user will want to separate bodies after webcutting. One reason to possibly have this option turned off is to be able to keep track of all the volumes during a webcut. Setting this option to "off" keeps all volumes in the same body. But the more common approach is to name the original body and allow naming to keep track of volumes. This setting is on by default. The syntax is:

**Set Separate After Webcut [ON|Off]**

# cleanup and defeaturing

## Geometry Cleanup and Defeaturing

Frequently, models imported from various CAD platforms either provide too much detail for mesh generation and analysis, or the geometric representation is deficient. These deficiencies can often be overcome with small changes to the model. Several tools are provided in CUBIT for this purpose.
The following describes the features available in CUBIT for clean up and defeaturing


- Healing
- Tweaking Geometry
- Removing Geometric Features
- Automatic Geometry Clean-up
- Regularizing Geometry
- Finding Surface Overlap
- Validating Geometry
- Debugging Geometry
- Geometry Accuracy
- Trimming and Extending Curves
- Stitching Sheet Bodies
- Blunting Tangencies (Removing small angles)
- Defeaturing Tool
- Reducing Geometry


healing

# *Analyzing Geometry*

The following command analyzes the ACIS geometry and will indicate problems detected:

**Healer Analyze Body <id_range> [Logfile ['filename'] [Display]]**

The **logfile** option writes the analysis results to the filename specified, or to 'healanalysis.log' by default. In the GUI version of CUBIT, the **display** option will write the results in a dialog window.
The outputs include an estimate of the percentage of good geometry in each body. The optional logfile will include detailed information about the geometry analysis. By default CUBIT will also highlight the bad geometry in the graphics and give a printed summary indicating which entities are "bad". Sample output from this command is shown below:

```
Percentage good geometry in Body 9: 98%
HEALER ANALYSIS SUMMARY:
------------------------
Analyzed 1 Body: 9

Found 2 bad Vertices: 51, 52

Found 3 bad Curves: 76, 77, 80
```

```
Found 2 bad CoEdges. The Curves are: 76

Found 1 Bodies with problems: 9

Journaled Command: healer analyze body 9
```

Note that it is not necessary to analyze the geometry before healing; however, it can be useful to analyze first rather than healing unnecessarily. Also note that healer analysis can take a bit of time, depending on the complexity of the geometry and how bad the geometry is.

The validate geometry commands work independently of the healer and give more detailed information.

#### Healer Settings

You can control the outputs from the healer with the following commands:

**Healer Set OnShow {Highlight|Draw|None}**

**Healer Set OnShow {Badvertices|Badcurves|Badcoedges|Badbodies|All} {On|Off}**

**Healer Set OnShow Summary {On|Off}**

These settings allow you to highlight, draw or ignore the bad entities in the graphics. You can control which entity types to display, as well as whether or not to show the printed summary at the end of analysis.

After you have analyzed the geometry (which can take some time), you can show the bad geometry again with the "**show**" command. This command simply uses cached data (healing attributes - see the next section) from the previous analysis.

**Healer Show Body <id_list>**

### *Auto Healing*

Healing is an extremely complex process. The general steps to healing are:

- **Preprocess** - trim overhanging surfaces and clean topology (remove small curves and surfaces).
- **Simplify** - converts splines to analytic representations, if possible.
- **Stitch** - geometry cleanup and stitching loose surfaces together to form bodies.
- **Geometry Build** - repairing and building geometry to correct gaps in the model.
- **Post-Process** - calculating pcurves and further repairing bad geometry.
- **Make Tolerant Curves & Vertices** - a last optional step that allows special handling of unhealed entities for booleans - allowing inaccurate geometry to be tolerated.

*Autohealing* makes these steps automatic with the following command:

**Healer Autoheal Body <id_range> [Rebuild] [Keep] [Maketolerant] [Logfile ['logfilename'] [Display]]**

The **rebuild** option unhooks each surface, heals it individually, then stitches all the surfaces back together and heals again. In some cases this can more effectively fix up the body, although it is much more computationally intensive and is not recommended unless normal healing is unsuccessful.
The **keep** option will retain the original body, putting the resulting healed body in a new body.
The **maketolerant** option will make the edges *tolerant* if ACIS is unable to heal them. This can result in successful booleans even if the body cannot be fully healed - ACIS can then sometimes "tolerate" the bad geometry. Note that the healer analyze command will still show these curves as "bad", even though they are tolerant. The validate *geometry* commands however take this into consideration.
The output from the autoheal command can be written to a file using the **logfile** option; the default file name is autoheal.log. The **display** option works as before, displaying the results in a window in the GUI version of CUBIT.

# *Healing*

Healing is an optional module that detects and fixes ACIS models.
It is possible to create ACIS models that are not accurate enough for ACIS to process. This most often happens when geometry is created in some other modeling system and translated into an ACIS model. Such models may be imprecise due to the inherent numerical limitations of their parent systems, or due to limitations of data transfer through neutral file formats. This imprecision can also result when an ACIS model is created at a different tolerance from the current tolerance settings. This imprecision leads to problems such as geometric errors in entities, gaps between entities, and the absence of connectivity information (topology). Since ACIS is a high precision modeler, it expects all entities to satisfy stringent data integrity checks for the proper functioning of its algorithms. Therefore, if such imprecise models must be processed by an ACIS based system, "healing" of such models is necessary to establish the desired precision and accuracy.
The following sections describe how to use the Healing capability in ACIS and CUBIT to analyze and heal defective ACIS geometry.

- Analyzing Geometry
- Healing Attributes
- Auto Healing
- Spline Removal
- What if Healing is Unsuccessful?

*Healing Attributes*

Once the geometry is analyzed, the results are stored as attributes on the solid model - this allows you to use the "**show**" command to quickly display the bad geometry again. The results attributes are automatically removed when the geometry is exported or any boolean operations are performed. They can also be explicitly removed with the command

```
Healer CleanAtt Body <id_range>
```

You can force the results to be removed immediately after each analyze operation with the "**CleanAtt**" setting (this can save a little memory):

```
Healer Set CleanAtt {On|Off}
```

# *Spline Removal*

If healing fails to convert spline surfaces to analytic ones fails, the simplification tolerance can be modified and healing re-run:

```
healer default simplifytol .1

healer autoheal body 1
```

Spline surfaces can also be forced into an analytic form (use this command with caution):

```
Healer Force {Plane|Cylinder|Cone|Sphere|Torus} Surface <id_list> [Keep]
```

The Keep option will retain the original body and generate a new body containing analytic surfaces. Note: Spline curves can be found using entity filters:

```
Execute Filter Curve Geometry_type Spline
```

# *What if Healing is Unsuccessful?*

The ACIS healing module is under continued development and is improving with every release. However, there will often be situations where healing is unable to fully correct the geometry. This might be okay, as meshing is rarely affected by the small inaccuracies healing addresses. However, boolean operations on the geometry can fail if the bad

geometry must be processed by the operation (i.e., a webcut must cut through a bad curve or vertex).
Here are some possible methods to fix this bad geometry:

- Return to the source of the geometry (i.e., Pro/ENGINEER) and increase the accuracy. Re-export the geometry.
- Heal again using the **rebuild** option.
- Heal again using the **make tolerant** option.
- Remove the offending surface from the body (using the remove surface command), then construct new surfaces from existing curves and combine the body back together.
- Composite the surfaces over the bad area, mesh and create a net surface from the composite, remove the bad surfaces and combine.
- Export the geometry as IGES, import the IGES file into a new model and look for double surfaces or surfaces that show up at odd angles using the find overlap commands. Delete and recreate surfaces as needed and combine the surfaces back together into a body.

Contact the development team (cubit-dev@sandia.gov) if you need further help with fixing bad geometry.

tweaking geometry

# *Tweaking Surfaces*

The following options of the Tweak Surface command are available. Command syntax and examples follow below.


- Tweaking a Surface Using an Offset
- Tweaking a Surface by Moving
- Tweaking Surfaces to Target Surfaces
- Removing a Surface
- Tweaking a Conical Surface
- Tweaking Doublers to Target Surface
- Removing Holes and Slots from Sheet Bodies
- Removing Fillets from Sheet Bodies
- Changing the Taper of Surfaces


#### Tweaking a Surface Using an Offset

**Tweak Surface <id_list> Offset <val> [Surface <id_list> Offset <val>] [Surface <id_list> Offset <val> ...] [Keep] [Preview]**

The **Tweak Offset** form of the command offsets an existing set of surfaces and extends the attached surfaces to meet them. A positive offset value will offset the surface in the positive surface normal direction while a negative value will go the other way. Different offsets may be specified for each surface. Figure 1 shows a simple example of offsetting.

Note that you can also offset whole groups of surfaces at once. The keep option will retain the original surfaces and curves.

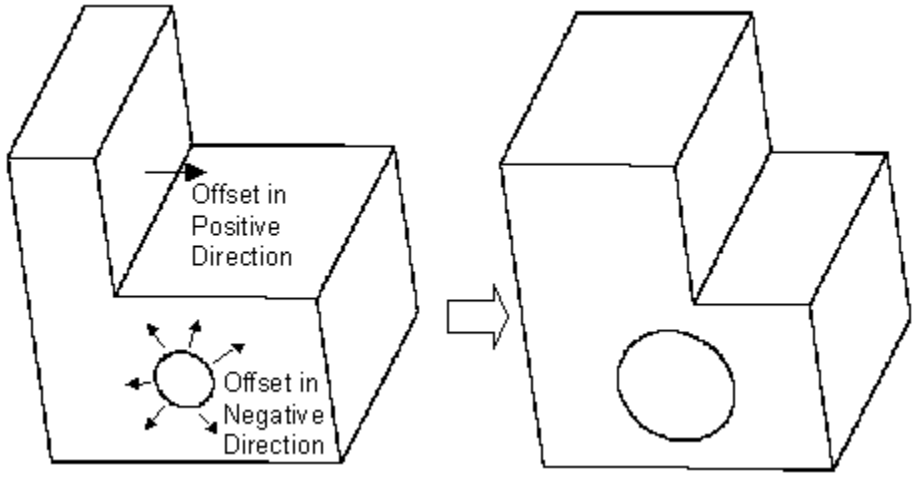



**Figure 1. Tweak Offset**

#### Tweaking a Surface by Moving

The **Tweak move** form of the command simply moves the given surfaces along a vector direction. The direction can be specified either absolutely or relative to other geometry entities in the model (from entity centroid to location). Note that when moving a surface for tweak, the surface is moved and the surface and the adjoining surfaces are extended or trimmed to match up again. So, for example, moving a vertically oriented planar surface in the vertical direction will have no effect. In this example, if you move the surface 10 in the x and 5 in the y the effect will be to move it simply 10 in the x. You can also use this form of the command to move a protrusion around - just be sure to specify all of the surfaces on the protrusion for moving. The last form of the command can be used to move a surface along another surface's normal.

**Tweak Surface <id_range> Move {Vertex|Curve|Surface|Volume|Body} <id> Location {Vertex|Curve|Surface|Volume|Body} <id> [Except [X][Y][Z]] [Keep] [Preview]**

**Tweak Surface <id_range> Move {Vertex|Curve|Surface|Volume|Body} <id> Location <x_val> <y_val> <z_val> [Except [X][Y][Z]] [Keep][Preview]**

**Tweak Surface <id_range> Move <dx_val> <dy_val> <dz_val> [Keep] [Preview]**

**Tweak Surface <id_range> Move Direction <options> Distance <val> [Keep] [Preview]**

**Tweak Surface <id_range> Move Normal To Surface <id> Distance <val> [Except [X][Y][Z]] [Keep][Preview]**

#### Tweaking Surfaces to Target Surfaces

The **Tweak target** form of the command actually replaces the given surfaces with a copy of the new surfaces, then extends and trims surfaces to match up. This can be useful for closing gaps between components or performing more complicated modifications to models. The command syntax is:

**Tweak {Curve|Surface} <id_list> Target {Surface <id_list> [Limit Plane (options)] [EXTEND|noextend] | Plane (options)} [keep] [preview]**

**Tweak Surface <id_list> Replace [With] Surface <id_list> [Keep] [Preview]**

The **plane** option allows a plane to be specified instead of target surface(s). If a target surface is supplied, the user can also use a **limit plane** if he wishes. A limit plane is a plane that the tweak will stop at if the tweaked surface does not completely intersect the target surface. The limit plane must be used with the extend option. See the help for Specifying a Plane for the options available to define a plane.
Single target surfaces are automatically extended so that the tweaked body will fully intersect the target. Unfortunately, extending multiple target surfaces can sometimes result in an invalid target, so the option is given to tweak to unextended targets with the **noextend** option. In this case, the tweaked body must fully intersect the existing targets for success. If you experience a failure when tweaking to multiple targets or the results are unexpected, it is recommended to try the **noextend** option (**NOTE:** Tweaking to multiple targets is only implemented in the ACIS geometry engine). It is recommended to always **preview** before using the tweak target commands.
Figure 2 shows a simple example.

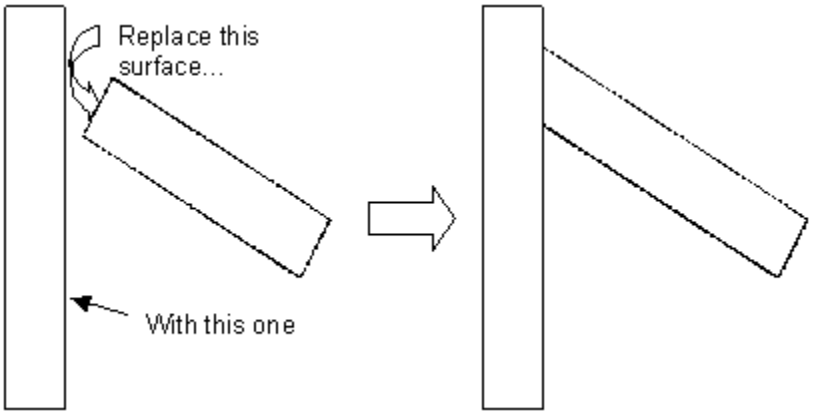



**Figure 2. Tweak Surface Target (Viewed directly from the side)**

#### Removing a Surface

The **Tweak remove** command allows you to remove surfaces from a model by extending the adjacent surfaces to fill in the resulting gaps. It is identical to the **Remove Surface** command. See Removing Surfaces for a description of the command options.

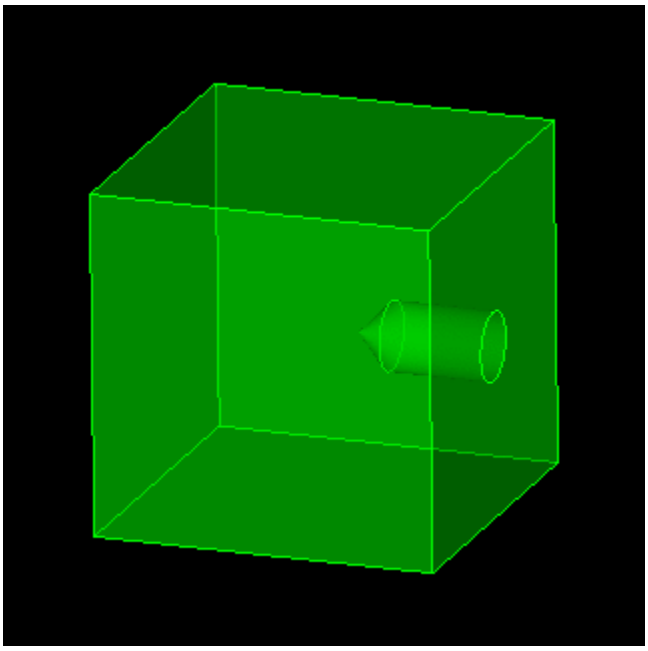
**Tweak Surface <id_list> Remove [Blend_Chain] [Cavity] [EXTEND|Noextend] [Keepsurface] [Keep] [Preview]**

#### Tweaking a Conical Surface

The **Tweak cone** form of the command is used to replace a conical projection with a flat circular surface. This command is useful for simplifying bolt holes. The command syntax is.

**Tweak Surface <id_range> Cone [Preview]**

The following is a simple example illustrating the use of the tweak surface cone command.



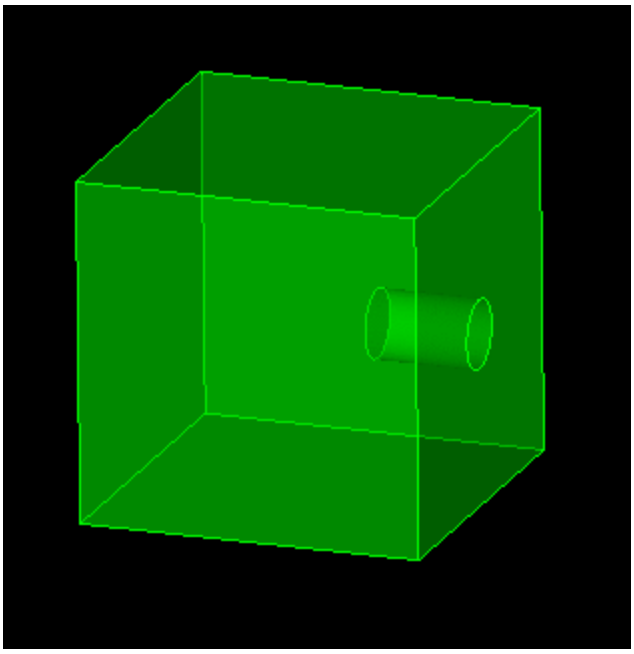


**Figure 3. Conical bolt hole before and after tweaking**

#### Tweaking Doublers to Target Surfaces

The Tweak Doubler form of the command takes a specified surface and creates drop-down surfaces either normal to the doubler surface or by a user specified vector to a target surface. This can be helpful in creating surfaces for weld elements between midsurfaced geometry. The resulting surfaces do not create a bounding volume, and do not imprint themselves onto the target surface. The command syntax is:

**Tweak Surface <id_list> Doubler Surface <id_list> {[Limit Plane (options)] [EXTEND|noextend]} [Internal] [Direction (options)] [Thickness] [Preview]**

The **plane** option allows a plane to be specified instead of target surface(s). If a target surface is supplied, the user can also use a **limit plane** if he wishes. A limit plane is a plane that the tweak will stop at if the tweaked surface does not completely intersect the target surface. The limit plane must be used with the extend option. See the help for Specifying a Plane for the options available to define a plane.
Single target surfaces are automatically extended so that the tweaked body will fully intersect the target. Unfortunately, extending multiple target surfaces can sometimes result in an invalid target, so the option is given to tweak to unextended targets with the **noextend** option. In this case, the tweaked body must fully intersect the existing targets for success. If you experience a failure when tweaking to multiple targets or the results are unexpected, trying the noextend option is recommended.
If the doubler surface has a thickness property value, you can propagate that thickness value to the newly created drop-down surfaces by using the **thickness** flag.

It is recommended to always preview before using the tweak doubler commands.
NOTE: This function only works for ACIS geometry.

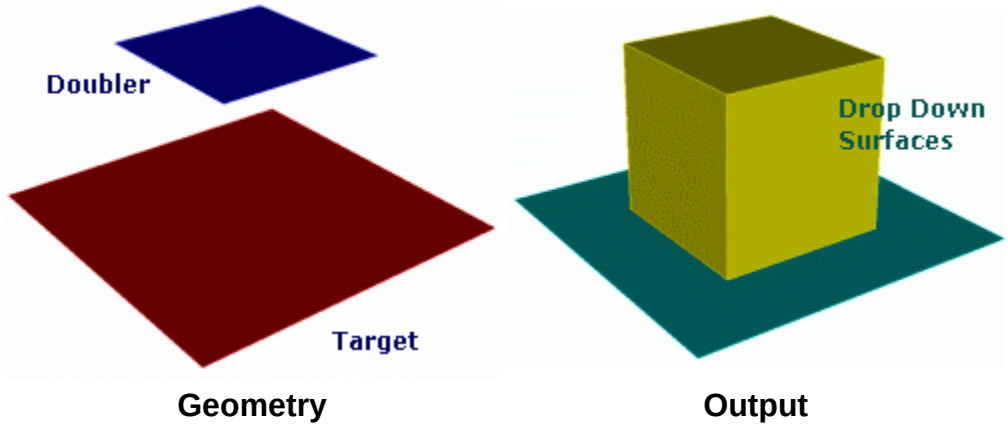



**Figure 3. Extending a doubler surface to target**

The **internal** option will also include internal curves when the surface is extended (see Figure 4c). The **direction** option will create a skewed surface along the given direction (see Figure 4d).

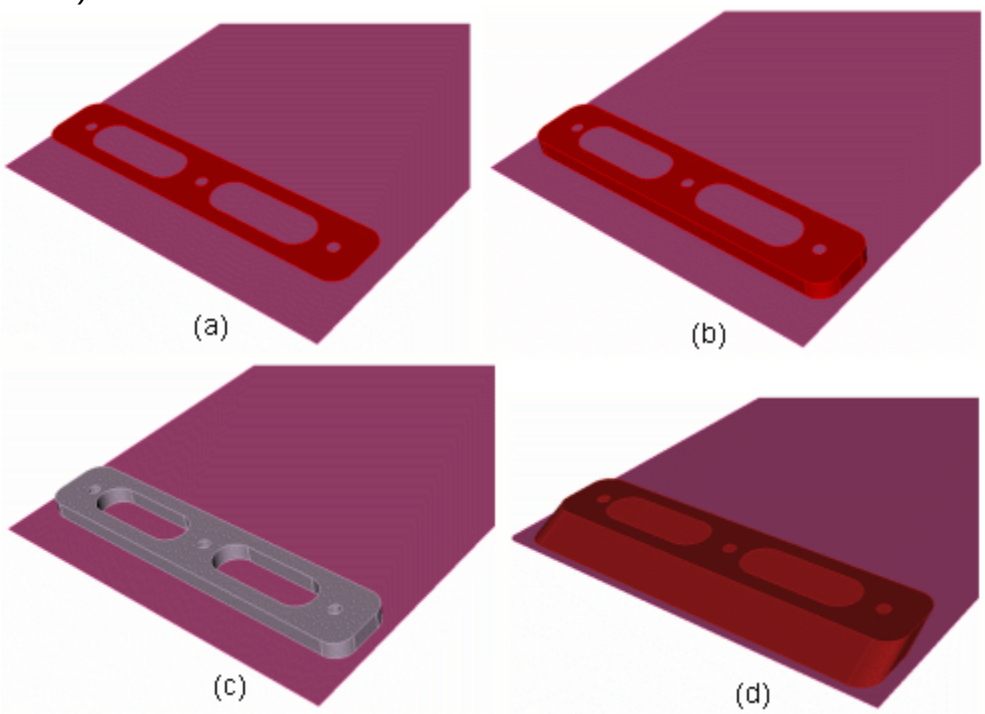



**Figure 4. Explanation of tweak doubler options (a) Original surfaces (b) No option flags used (c) Internal option used - notice internal curves dropped down (d) Direction flag - notice skew**

#### Removing Holes and Slots from Sheet Bodies

The **Tweak Hole/Slot Idealize** command takes a specified sheet body(s) and searches for either holes or slots (or both) which meet the user's input parameters. This can be helpful in removing small holes or slots quickly and efficiently from midsurfaced bodies where such level of detail isn't required. The command syntax is:

**Tweak Surface <id_list> Idealize {[Hole Radius <val>] [Slot Radius <val> Length <val>]} [Exclude Curve <id_list>] [Preview]**

Below is a diagram showing the different parameters available for input by the user.

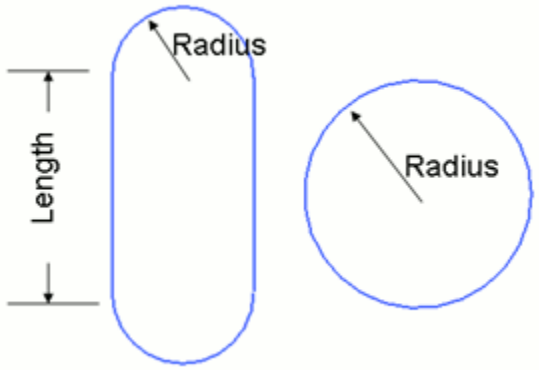



**Figure 5. Input parameters for tweak surface idealize command**

#Hole Removal Example
**tweak surface 13 idealize hole radius 6**

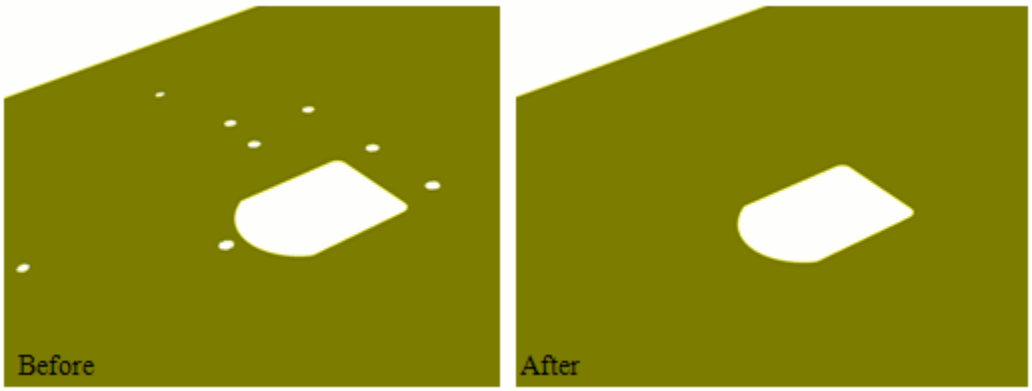



**Figure 6. Example of hole removal using tweak surface idealize command**

The **exclude** option allows the user to specify individual curves that should not be deleted, even if they meet the search criteria for removal. Figure 7 shows another hole removal example where several curves were excluded.

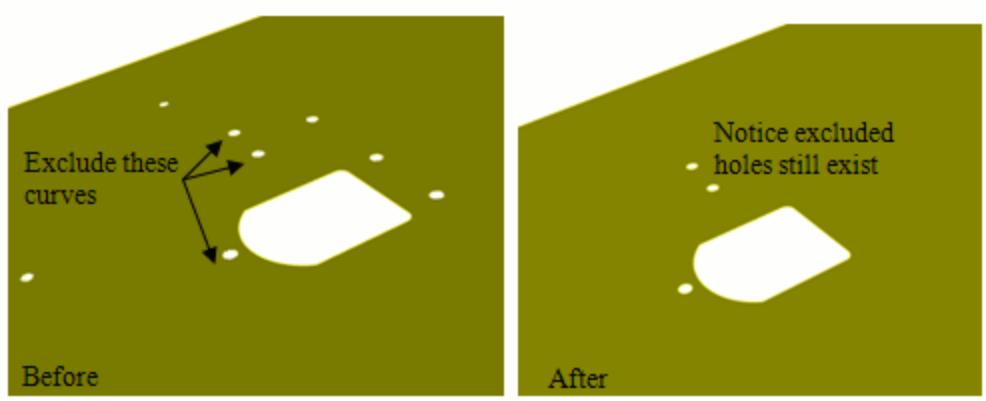



**Figure 7. Example of hole removal using exclude option**

**Note:** This feature is for ACIS geometry
It is recommended to always **preview** before using the tweak command. Preview will highlight all curves slated to be removed if the command is executed.

#### Removing Fillets from Sheet Bodies

The **Tweak Fillet Idealize** command takes a specified sheet body(s) and searches for either internal or external fillets (or both) which meet the users' radius parameter. This can be helpful in removing fillets quickly and efficiently from midsurfaced bodies where such level of detail isn't required. The command syntax is:

**Tweak Surface <id_list> Idealize Fillet Radius <val> {[Internal] [External]} [Exclude Curve <id_list>] [Preview]**

#Fillet Removal Example
**tweak surface 13 idealize fillet radius 6 internal**

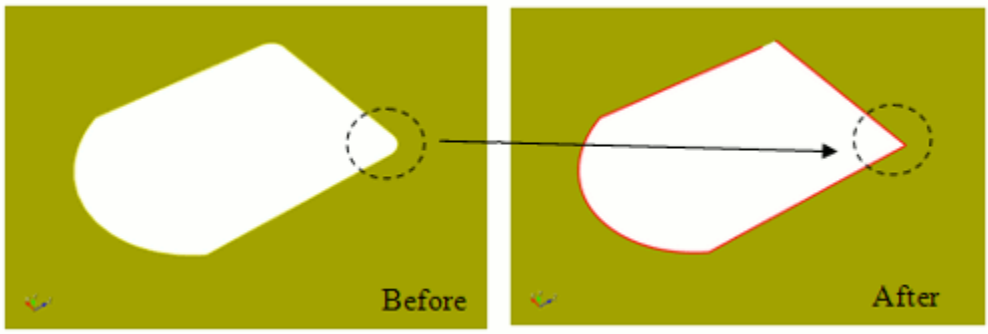



**Figure 8. Example of fillet removal using tweak surface idealize command**

**Note:** This feature is for ACIS geometry
It is recommended to always **preview** before using the tweak command. Preview will show the result if the command is executed.

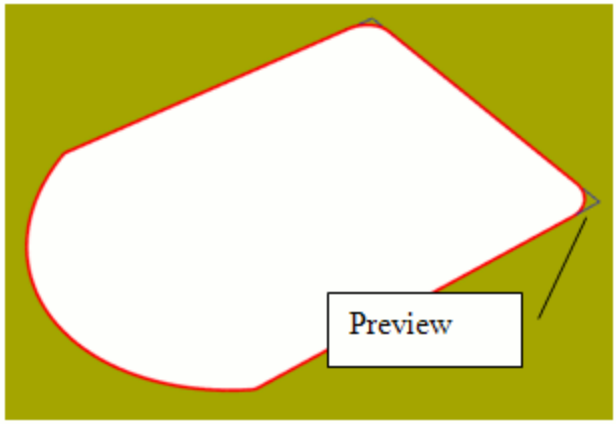



**Figure 9. Preview of the tweak surface idealize command**

#### Changing the Taper of Surfaces

The taper or angle of a surface can be made shallower or steeper using the taper surface command. Several surfaces can be tapered at once to get a smooth transition. The command syntax is:

**taper Surface <id_list> angle <val> {from plane <options> | about curve <id> | about vertex <id_1><id_2>} [keep] [preview]**

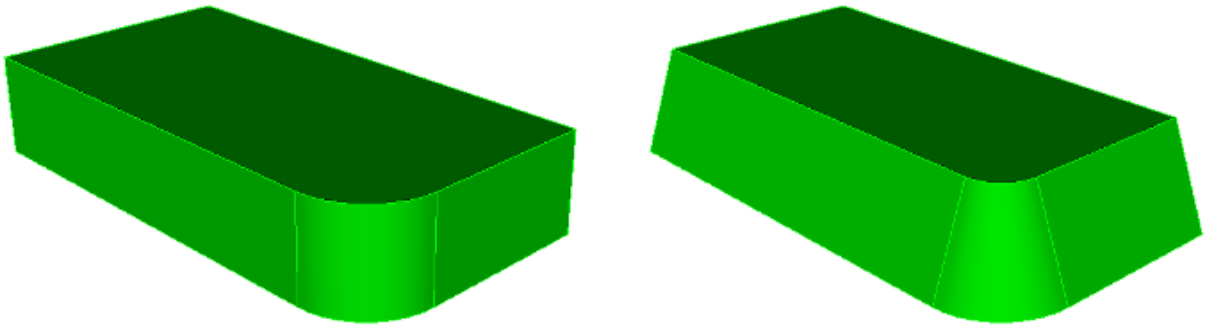


**Figure 10. taper surface 1 7 6 angle -20 from plane surface 3**

The **keep** and **preview** options are helpful when figuring out the correct command setup. The **from plane** form of the command is the most basic form of the command. The plane normal defines the direction used to set the angle for the surface. The origin of the plane determines how a surface is tapered. If the plane intersects the middle of a surface, the surface will be rotated in on the top and out on the bottom. If it intersects the bottom, the top of the surface will be rotated in and the bottom will stay fixed. If it intersects the top, the bottom of the surface will rotate out and the top will stay fixed.

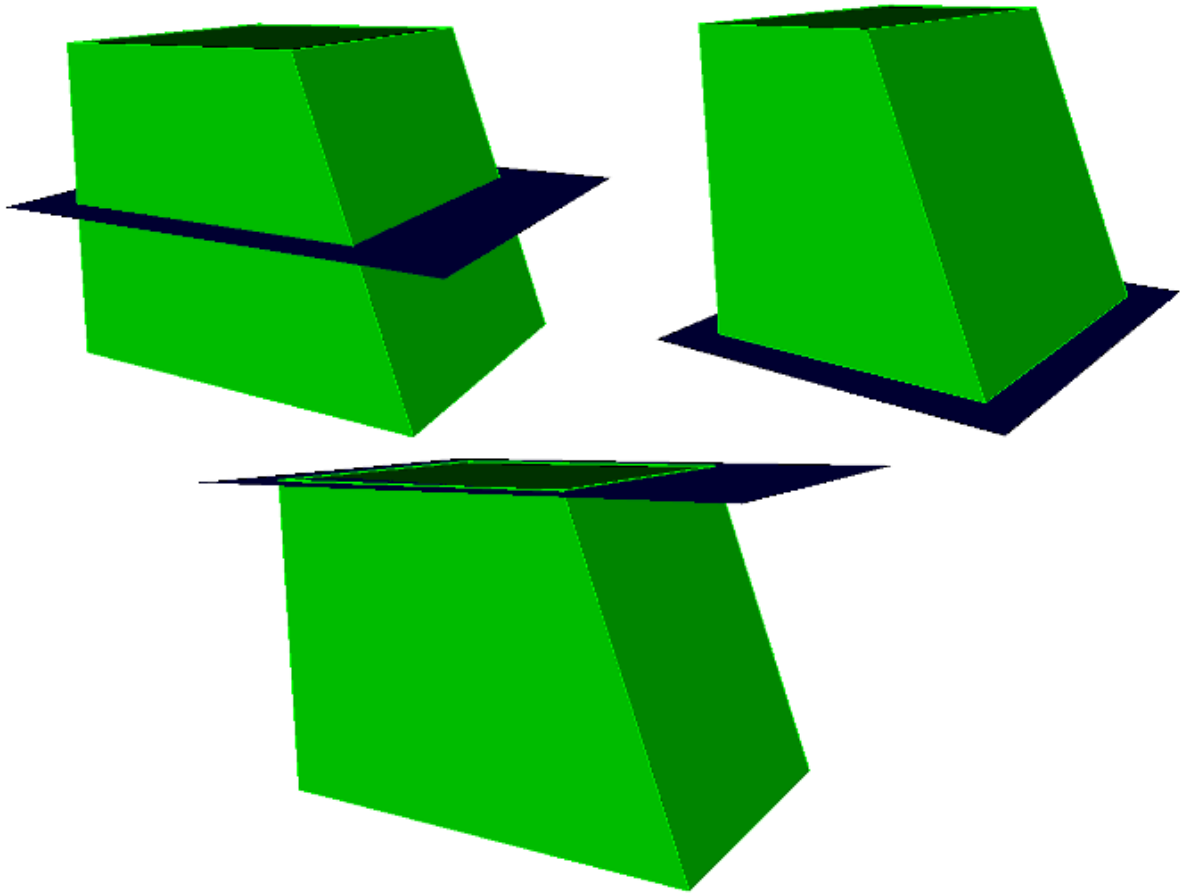


**Figure 11. A cube with one surface tapered 20 degrees with a plane in the middle, bottom, and top.**

The **about curve** form of the command can be used to rotate a surface about a curve. The axis of rotation is determined using the tangent direction of the curve at its starting point. That axis crossed with the normal of the surface at the starting point determine the direction used to set the surface angle. In order to work, the curve must be part of one of the surfaces being changed. Since the direction is based on the curve sense, the final direction is not always obvious. Running the command with the **preview** option first can help determine what angle to apply.
The **about vertex** form of the command is similar to the **about curve** form. The axis of rotation is a vector from the first vertex to the second. That axis crossed with the normal of the surface at the starting point determine the direction used to set the surface angle. In order to work, the vertices must be part of one of the surfaces being changed.

# *Tweaking Vertices*

The Tweak Vertex command can be used to do the following:

- Tweaking a Vertex With a Chamfer
- Tweaking a Vertex With a Non-Equal Chamfer
- Tweaking a Vertex With a Fillet Radius

#### Tweaking a Vertex With a Chamfer

**Tweak Vertex <id_range> Chamfer Radius <value>[Keep] [Preview]**

This form of the command creates a chamfered corner at the specified vertex. Can be use on volumes or free surfaces. The 'keep' option creates another volume on which the tweak is applied; the original volume remains unmodified.

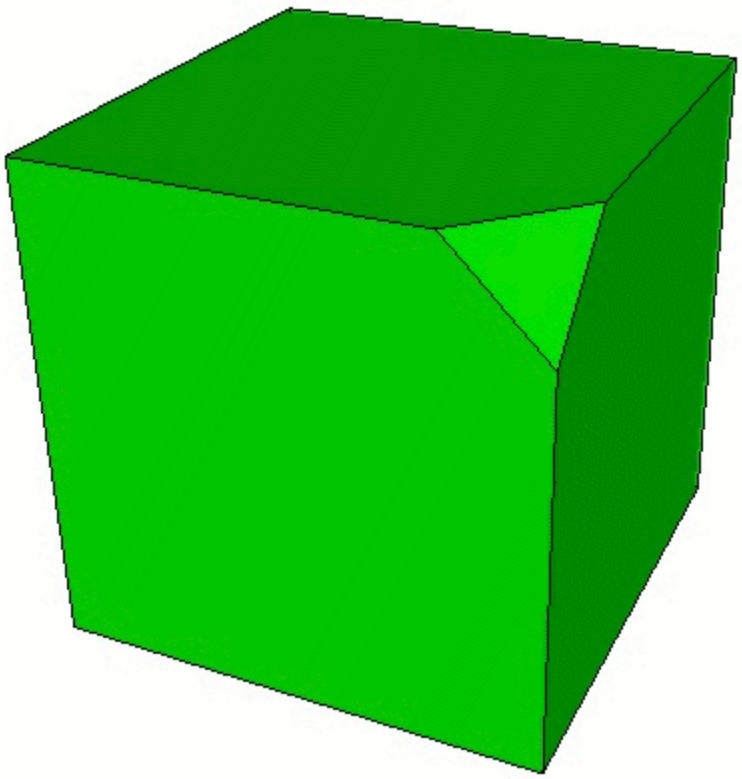


**Figure 1. Tweak Vertex Chamfer**

#### Tweaking a Vertex With a Non-Equal Chamfer

**Tweak Vertex <id_range> Chamfer Radius <value> [Curve <id> Radius <value> Curve <id> Radius <value> Curve <id>] [Keep] [Preview]**

This next form of the command creates a non-equal chamfered corner at the specified vertex. Can only be used on vertices of volumes. The 'keep' option creates another volume on which the tweak is applied; the original volume remains unmodified.

#### Tweaking a Vertex With a Fillet Radius

**Tweak Vertex <id_range> Fillet Radius <value> [Keep] [Preview]**

This command replaces a vertex with a filleted radius. The command can only be used on free surfaces. The 'keep' option creates another volume on which the tweak is applied; the original free surface remains unmodified.

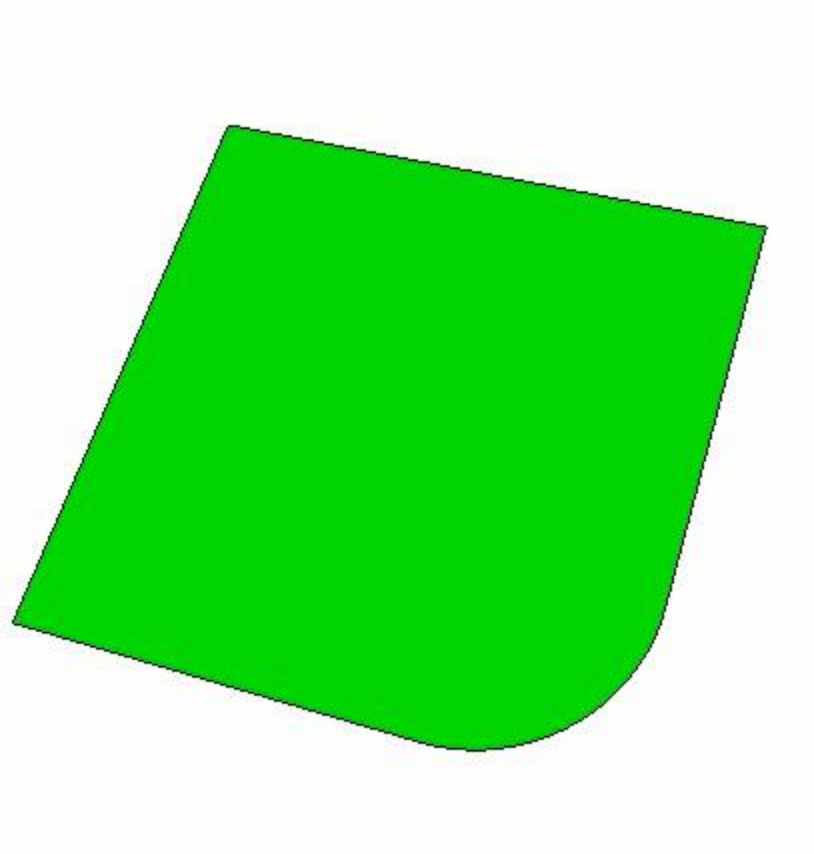


**Figure 2. Tweak Vertex Fillet**

# *Tweak Volume Bend*

Entity bending bends a solid model around a given axis. In any bending operation, some material is stretched while other material is compressed, but the topology of the model is maintained. The command syntax is:

**Tweak {Volume|Body} <id_list> Bend Root <location_options> Axis <direction_vector> Direction <direction_vector> Radius <val> angle <val> [Preview] [Keep] [Center_bend] [Location <options>]**

**Root** and **axis** determine location for the bend. **Direction** determines direction of the bend. **Radius** and **angle** determine how much to bend. **Center_bend** will bend both sides of the volume around the bend location instead of one side. **Location** can be used to select only specific parts of a volume to bend.

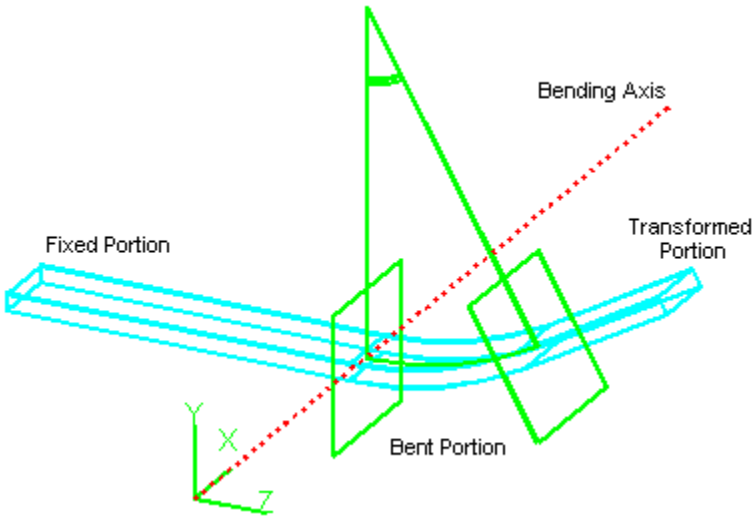



**Figure 1. Bending a volume**

#Ex: Bend parts of a body specified by the location option

```
create brick width 11 height 1
create brick width 1 depth 10 height 10
create brick width 1 depth 10 height 10
create brick width 1 depth 10 height 10
move body 2 general location position -3 5 0
move body 3 general location position 0 5 0
move body 4 general location position 3 5 0
subtract body 2 from body 1
subtract body 3 from body 1
subtract body 4 from body 1
tweak volume 1 bend root 0 0 0 axis 1 0 0 direction 0 0 -1 radius 1 angle 3.14 location vertex 39 47
```

# *Tweaking Geometry*

- Tweaking Vertices
- Tweaking Curves

- Tweaking Surfaces
- Tweak Remove Topology
- Tweak Volume Bend

The tweaking commands modify models by moving, offsetting or replacing surfaces, curves, or volumes while extending the adjoining surfaces to fill the resulting gaps. This is useful for eliminating gaps between components, simplifying geometry or changing the dimensions of an object.

*Tweaking Curves*

The following options of the Tweak Curve command are available. Command syntax and description follow below.

- Create a Chamfer or Fillet
- Tweaking a Curve Using an Offset Distance
- Removing a Curve
- Tweaking a Curve Using a Target Surface, Curve, or Plane
- Tweaking a Pair of Curves to a Corner

#### Create a Chamfer or Fillet

The Tweak Curve Chamfer or Fillet command is used to fillet or chamfer a curve. The radius value is the radius of the fillet arc or chamfer cut distance. The command syntax is:

**Tweak Curve <id_range> {Fillet|Chamfer} Radius <value> [Keep] [Preview]**

In addition to creating chamfers of a single cut distance, the chamfer can be specified be two values. The syntax is:

**Tweak Curve <id_list> Chamfer Radius <val1> [<val2>] [Keep] [Preview]**

Figure 1 shows a brick ('br x 10') chamfered with two different cut distances ('Tweak Curve 1 2 Chamfer Radius 2 4').

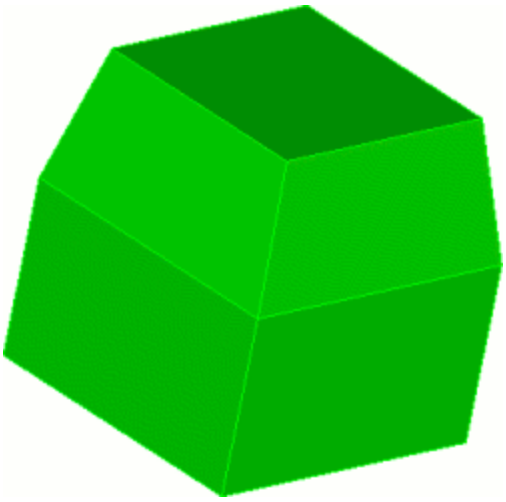


**Figure 1 Chamfer with two different distances**

Individual curves can also be filleted with different start and finish radius values. The syntax is:

**Tweak Curve <id> Fillet Radius <val1> [<val2>] [Keep] [Preview]**

Figure 2 shows a brick ('br x 10') filleted with different start and end radius values (‘Tweak Curve 1 2 Chamfer Radius 2 4’).

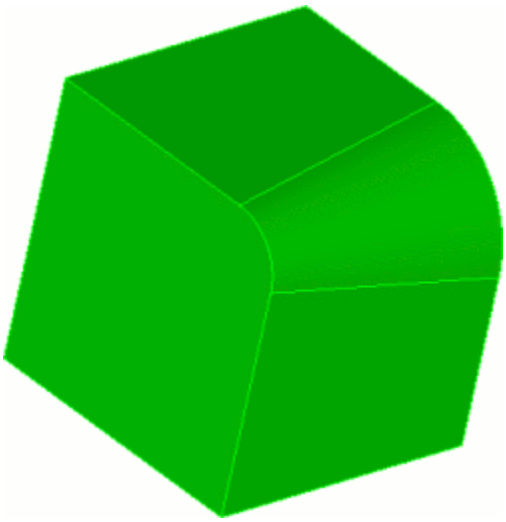


**Figure 2. Fillet with two different radii**

For all Tweak Fillet and Tweak Chamfer variations, the keep option prevents the destruction of the original geometry after the operation and the preview option temporarily displays the new geometry configuration without actually changing the geometry.

#### Tweaking a Curve Using an Offset Distance

**Tweak Curve <id_list> Offset <val> [Curve <id_list> Offset <val>] [Curve <id_list> Offset <val> ...] [Keep] [Preview]**

Tweaking curves a specified distance offsets the existing curves and extends the attached surfaces to meet them. A positive offset value will enlarge the surface while a negative value will decrease the area of the attached surface. Different offset values can be specified for each curve. The keep option prevents the destruction of the original geometry after the operation. The preview option temporarily displays the new geometry configuration without actually changing the geometry. Figure 3 shows an example of offsetting a curve a specified distance.

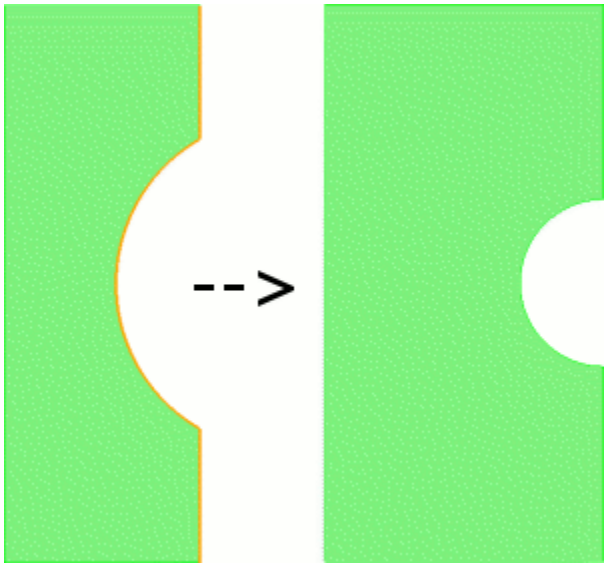



**Figure 3 Offsetting a set of curves a specified distance**

**Removing a Curve**

**Tweak Curve <id_list> Remove [Keep] [Preview]**

Similar to the Tweak Curve Remove command, the tweak curve remove function removes a specified curve from a sheet body. Figure 4 shows a simple example of removing a curve from a sheet body.

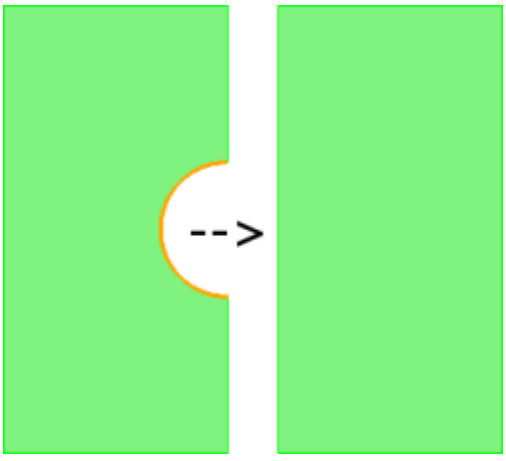


**Figure 4. Removing a curve from a sheet body**

The keep option prevents the destruction of the original geometry after the operation. The preview option temporarily displays the new geometry configuration without actually changing the geometry.

#### Tweaking a Curve Using Target Surfaces, Curves, or Plane

Use Tweak Curve Target to offset a curve to a specified surface, plane or curve. Figure 5 shows an example of tweaking a curve to several surfaces.

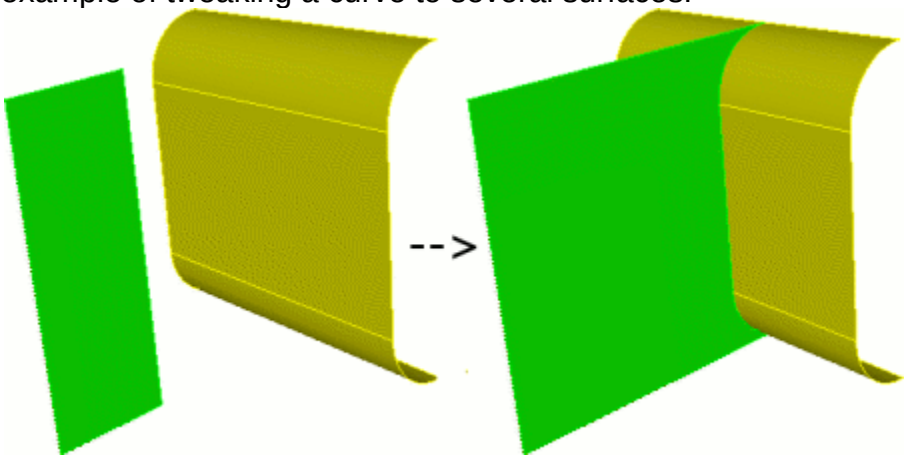



**Figure 5 Tweaking a curve to multiple target surfaces**

Similarly, a target plane can be specified using the Plane specification syntax. The Tweak Curve syntax is:

**Tweak Curve <id_list> Target {Surface >id_list> [Limit Plane (options)] [EXTEND|Noextend] | Plane (options)} [Max_area_increase <val>] [Keep] [Preview]**

**Tweak Curve <id_list> Target Curve <id_list > [EXTEND|Noextend] [Max_area_increase <val>] [Keep] [Preview]**

If a target surface is supplied, the user can also use a **limit plane** if he wishes. A limit plane is a plane that the tweak will stop at if the tweaked curve does not completely intersect the target surface. The limit plane must be used with the extend option. See the help for Specifying a Plane for the options available to define a plane.
It should be noted that if the source and target surfaces are from the same body the resulting geometry will be automatically stitched. Single target surfaces are automatically extended so that the tweaked body will fully intersect the target. Unfortunately, extending multiple target surfaces can sometimes result in an invalid target, so the option is given to tweak to non-extended targets with the **noextend** option. In this case, the tweaked body must fully intersect the existing targets for success. If you experience a failure when tweaking to multiple targets or the results are unexpected, it is recommended to try the

noextend option (**NOTE:** Tweaking to multiple targets is only implemented in the ACIS geometry engine). If a value for the **max_area_increase**keyword is given, Cubit will not perform the tweak if the resulting surface area increases by more than the specified amount. The keyword expects a percentage to be entered (i.e. '50' for 50%). It is recommended to always **preview** before using the tweak target commands.
For all tweak target variations, the **keep** option prevents the destruction of the original geometry after the operation and the preview option temporarily displays the new geometry configuration without actually changing the geometry.
Although it may not be intuitive curves can also serve as the target geometry. Figure 6 shows an example of extending a curve to another curve.

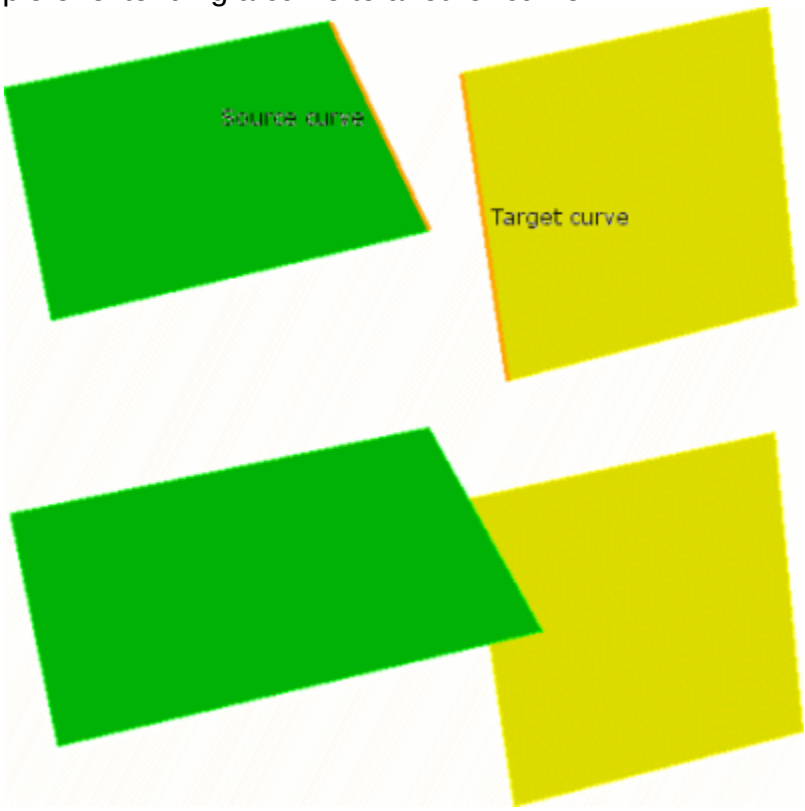



**Figure 6 Tweaking a curve to a target curve**

Notice that the source curve actually extends to the target curve as if the target were a surface.

#### Tweaking a Pair of Curves to a Corner

When creating mid-surface geometry it is often useful to extend surfaces to form a corner. To handle this specific but common case use the tweak corner command.

**Tweak Curve <id> <id> Corner [Preview]**

Figure 7 shows a typical tweak corner example. Notice that surfaces are extended/trimmed to intersect at a corner.

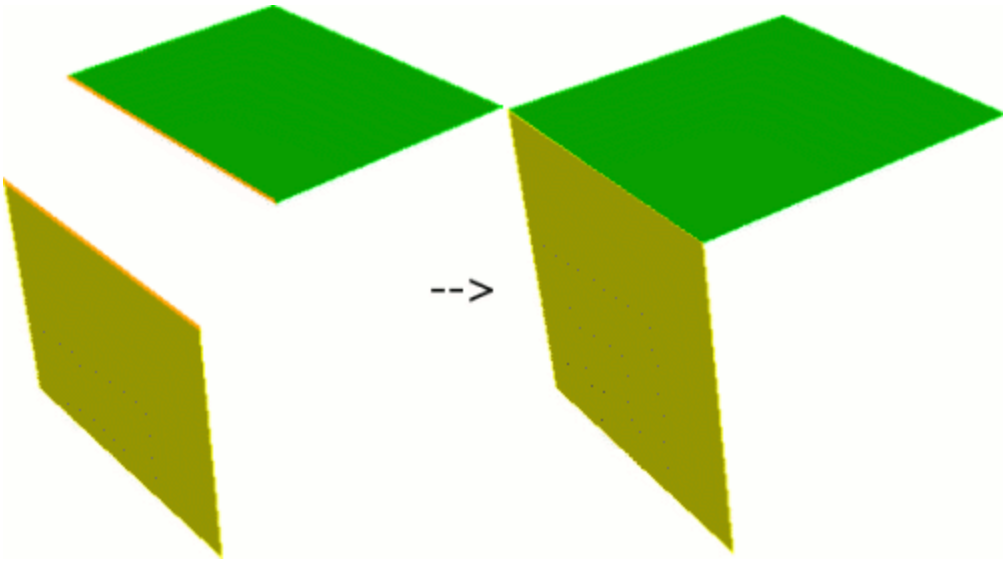



**Figure 7. Tweaking two curves to a corner**

The preview option temporarily displays the new geometry configuration without actually changing the geometry.

# *Tweak Remove Topology*

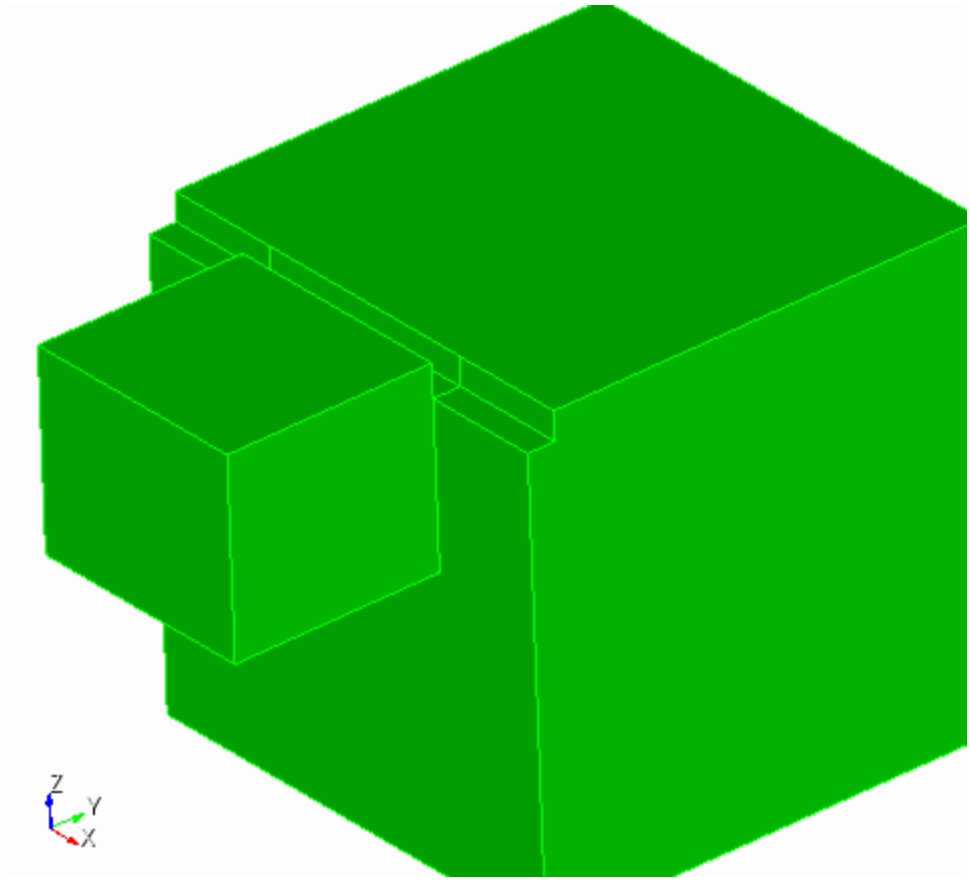
The **Tweak Remove Topology** command removes curves and surface from a model and replaces them with new topology. The reconstruction of the new topology and the stitching of it into the model is done using real solid modeling kernel operations. This command is intended to be used on small curves and surfaces in the model. The command tries to find small curves/surfaces neighboring the specified topology and includes these neighbors in the removal process. Thus, the command can often be used to remove networks of small features just by specifying a single curve or surface.

**Tweak Remove_Topology {Surface <id_range> | Curve <id_range> | Surface <id_range> Curve <id_range>} Small_curve_size <val> Backoff_distance <val>**

The **small_curve_size** is input by the user, and is used to calculate the small curves and surfaces. The **backoff_distance** value specifies how far away from the original topology cuts are made to cut out the old topology and stitch in the new topology. The removed topology is replaced by simplified topology where possible often resulting in a dimension reduction of the original topology. Extraneous curves that are introduced during the cutting and stitching process are regularized out if possible using the solid modeling kernel regularize functionality or are composited out using virtual geometry if the regularization is not possible.
**Note:** This command is currently only implemented for ACIS and Catia models.

**Example**

```
reset
set attribute on
import acis "test10.sat"
separate body all
set attribute off
Auto_clean Volume 1 Split_narrow_regions Narrow_size 2.2
tweak remove_topology curve 19 small_curve_size .21 backoff 1.5
```


Z
Y
X

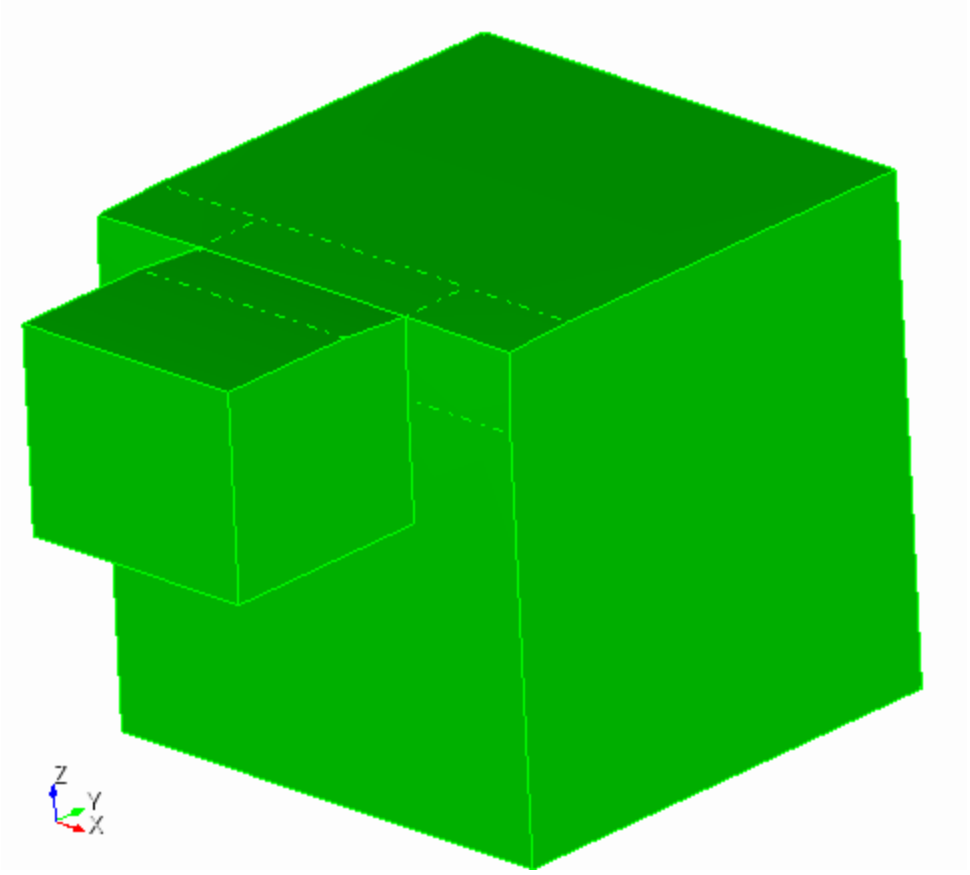



**Figure 1. Tweak Remove Topology command**

removing geometric features

# *Removing Surfaces*

- Remove Sliver Surfaces

The remove surface command removes surfaces from bodies. By default, it attempts to extend the adjoining surfaces to fill the resultant gap. This is a useful way to remove fillets and rounds and other features such as bosses not needed for analysis. See Figure 1 for an example of this process.  The syntax for this command is:

**Remove Surface <id_range> [Blend_Chain] [Cavity] [EXTEND|Noextend] [Keepsurface] [Keep] [TOGETHER|Individual|connected_sets]**

The **noextend** qualifier prevents the adjoining surfaces from being extended, leaving a gap in the body. This is sometimes useful for repairing bad geometry - the surface can be rebuilt with surface from curves or a net surface, etc.., then combined back onto the body.

The **keep** option will retain the original body and put the results of the remove surface in a new body. The **keepsurface** option will retain the surface which was removed.
By default, the **TOGETHER** option removes the surfaces in a single call to the solid modeler. The **individual** option will remove surfaces one-by-one instead of as a group. If one removal fails, the rest are still attempted. Without the **individual** option, no surface is removed unless they are all able to be removed together. The **connected_sets** option sorts the surfaces into sets of connected surfaces, operating on each set separately, allowing the remove operation to succeed on some sets if it fails on others.
The **blend_chain** option will not only remove the selected surface but will also remove any surfaces belonging to the same blend chain. A blend is a non-planar surface such as a fillet that has a constant radius of curvature in at least one of its principal parametric directions. The blend chain includes all connected surfaces that share a common radius of curvature.
The **cavity** option can be used to remove all surfaces defining a hole or cavity. A cavity is defined as the collection of surfaces bounded by curves where the exterior angle is greater than 180 degrees. Designating the cavity option will automatically include all surfaces that are part of the cavity to which the surface belongs. If the surface does not belong to a cavity, this option will be ignored.
This command is identical to the Tweak Surface Remove command.

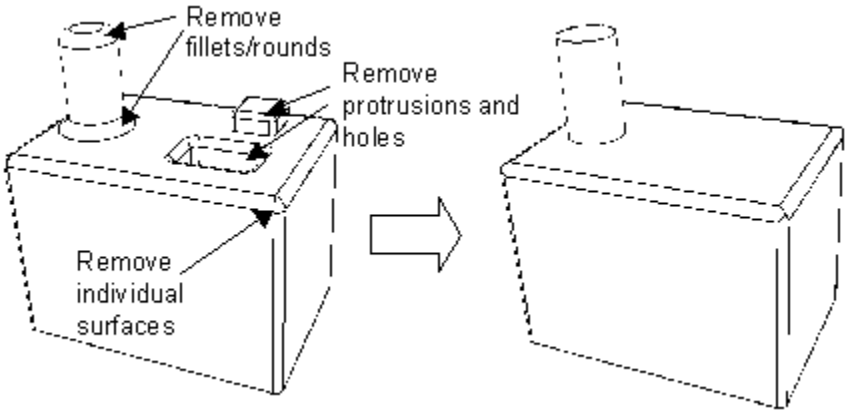



**Figure 1. Remove Surface Example**

#### Remove Sliver Surface

This command uses the ACIS remove surface capability on surfaces that have area less than a specified area limit. When ACIS removes a surface it extends the adjoining surfaces and intersects them to fill the gap. If it is not possible to extend the surfaces or if the geometry is bad the command will fail. The syntax for this command is:

**Remove Slivers Body <id_range> [EXTEND|Noextend] [Keepsurface] [Keep] [Arealimit [<double>]]**

**Default Arealimit = 0.1**

The **noextend**, **keepsurface** and **keep** options operate as for the remove surface command. The **arealimit** option allows the user to set the area below which surfaces will be removed.

# *Removing Vertices*

At times you may find that you have an extraneous vertex in your model. This would be a vertex connected to two and only two curves. This stray vertex can cause unwanted mesh artifacts, due to the fact that a mesh node MUST lie on this vertex, thereby disallowing the possibility of movement for better quality. Fortunately there is a relatively easy way of getting rid of this stray vertex using the tweak surface command.

**Tweak Surface <id> Replace With Surface <same_id>**

Note that you are replacing a surface with itself. In doing so, the geometry engine will do an intersection check on that surface, and should realize that the vertex doesn't need to be there.

# *Removing Geometric Features*

- Vertex Removal
- Curve Removal
- Surface Removal

Cubit has the ability to remove surfaces, curves, and vertices in an effort to simplify the model for meshing. For example, surface removal extends adjacent surfaces to those being removed to fill in the gap where the removed surfaces were. Curve removal replaces a short, sliver curves with a vertex. And vertex removal can be facilitated with tweaking if it is connect to two curves of the same geometric type.

auto clean

# *Automatic Geometry Clean-up*

The automated geometry clean-up commands are used to automatically clean up geometry in preparation for meshing. These commands are built in to the ITEM interface, but they can also be used on their own. They include:

- Automatic Forced Sweepability
- Automatic Small Curve Removal
- Automatic Small Surface Removal

- Automatic Surface Split

# *Automatic Forced Sweepability*

In some cases, a volume can be "forced" into a sweepable configuration by compositing surfaces on the linking surfaces. The automatic forced sweep command will attempt to automatically composite linking surfaces together to create a sweepable topology. This command can be useful in cases where there are many linking surfaces that prohibit sweepability and are not needed to define the mesh. It is assumed that the user has assigned the source and target surfaces for the sweep prior to calling this function. CUBIT will try to composite linking surfaces together to get rid of problems such as 1) non-submappable linking surfaces, 2) interior angles between curves of a surface that deviate far from multiples of 90 degrees, and 3) surfaces with curves smaller than the small curve size, if a small curve size is specified. This command is incorporated into the ITEM GUI, but is also available from the command line using the following command syntax.

```
Auto_clean Volume <id_range> Force_sweepability [Small_curve_size <val>]
```

The **small_curve_size** qualifier is an optional argument. If a curve size is specified, the command will try to remove surfaces with curves smaller than this size by compositing the surface with adjacent surfaces.

**Example**

The following cylinder has been webcut and had surface splits so that it is not sweepable. The split surface command has also introduced 3 small curves on the surfaces. After the source and target surfaces are set, the force sweepability command is issued to automatically composite neighboring surfaces to make the volume sweepable and remove the small curves. The results are shown in the image below.

```
auto_clean volume 1 force_sweepability small_curve_size .7
```

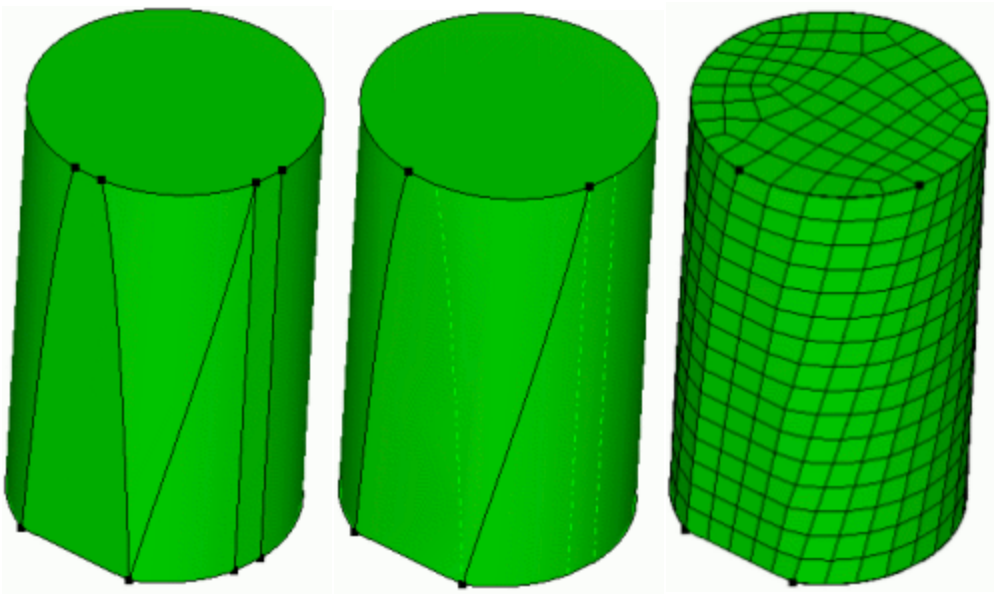


**Figure 1. Linking surfaces are composited to force a sweepable volume topology**

# *Automatic Surface Split*

This auto clean command will attempt to automatically split narrow regions of surfaces. In this context, any surface that contains a portion that narrows down to a small angle is considered a narrow region. The command will use the split command from the underlying solid modeling kernel. The user specifies a size that defines what it narrow. This command also propagates the splits to neighboring narrow surfaces. This command is usually used as a preprocessor to the "tweak remove_topology" command but can also be used on its own.

**Auto_clean Volume <id_range> Split_narrow_regions Narrow_size <val>**

**Example**
The model has a surface that necks down to a narrow region. This surface also has some neighboring narrow surfaces to which the splits are propagated.

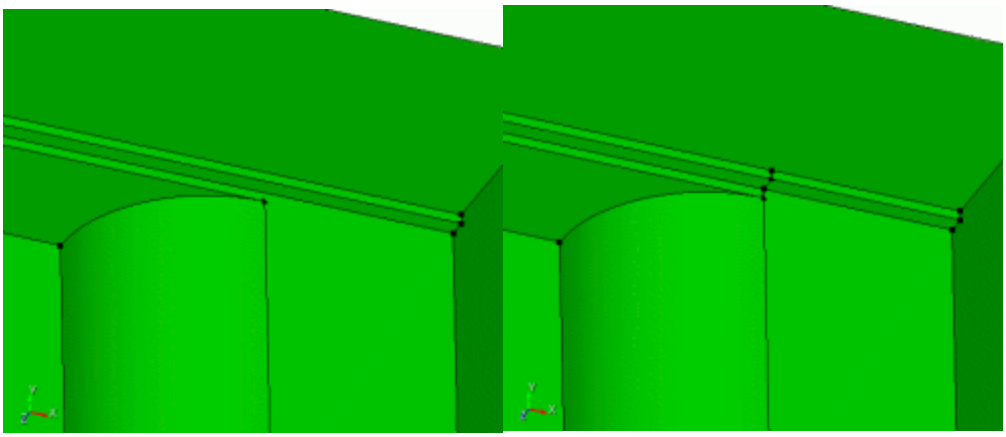


**Figure 1. Automatic small and narrow surface removal on a cylinder**

# *Automatic Small Curve Removal*

The automatic small curve removal command uses composites and collapse curves commands to automatically remove small curves from a volume. This is useful for removing small or unnecessary details from a model to facilitate meshing algorithms. The user enters a small curve size. Any curve smaller than this specified size will be removed. This command is issued from the ITEM toolbar. More information can be found by reading the section entitled Small Details in the Model in the ITEM documentation. This command can also be called from the command line. The syntax of this command is:

**Auto_clean Volume <id_range> Small_curves Small_curve_size <val>**

Note: The automatic curve removal should be used with caution, as the user has little control over how curves are removed.

**Example:**

The cylindrical model has 3 small curves just less than 0.7. The remove small curves command will remove two of the small curves by compositing two neighboring surfaces and the third using the collapse curve functionality.

**auto_clean volume 1 small_curves small_curve_size .7**

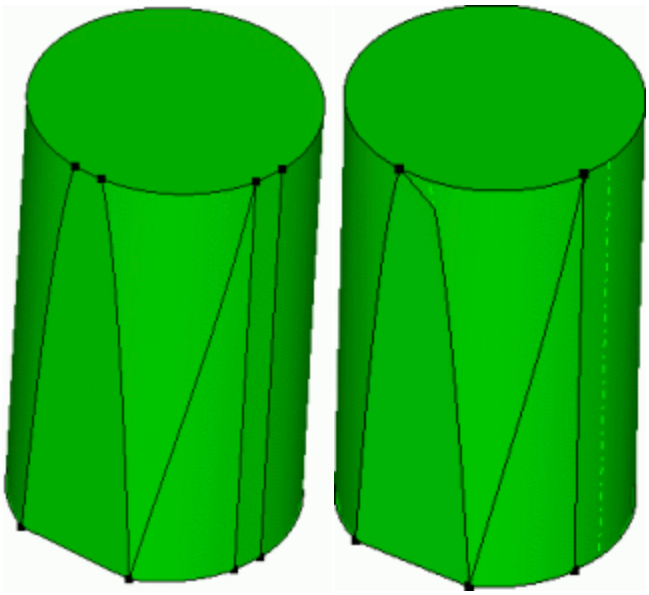


**Figure 1. Automatic small curve removal on a cylinder**

# *Automatic Small Surface Removal*

This auto clean command will attempt to remove small and narrow surfaces from the model by compositing them with neighboring surfaces. The user specifies a small curve size value. This value is used in two different ways. First, a small area is calculated as the small curve size squared. This value is used to compare against when looking for small surfaces. The small curve size is also used to identify surfaces that are narrower than the small curve size.

**Auto_clean Volume <id_range> Small_surfaces Small_curve_size <val>**

**Example**
The cylindrical model has 2 small surfaces and a few narrow surfaces. The surfaces are composited to remove these.

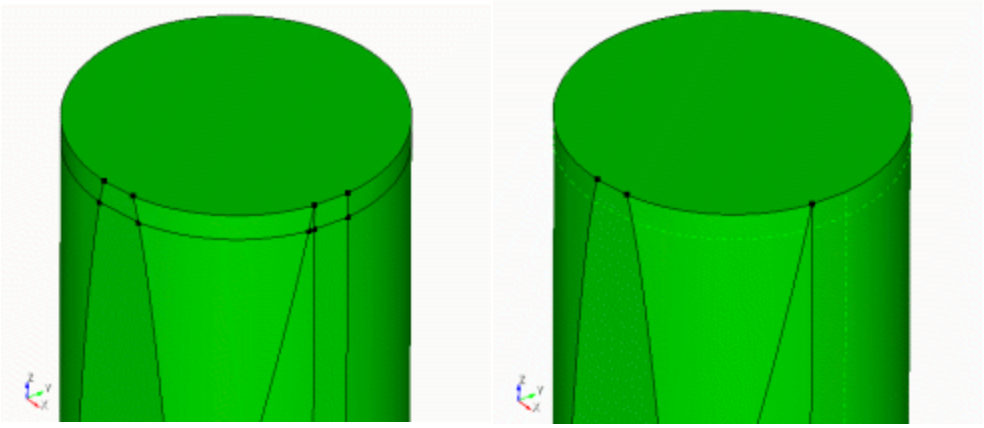


**Figure 1. Automatic small and narrow surface removal on a cylinder**

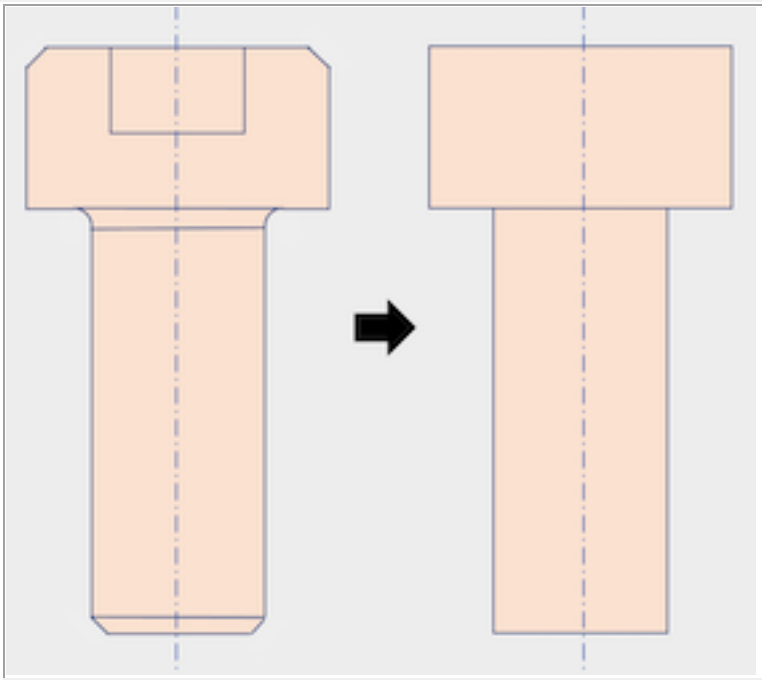
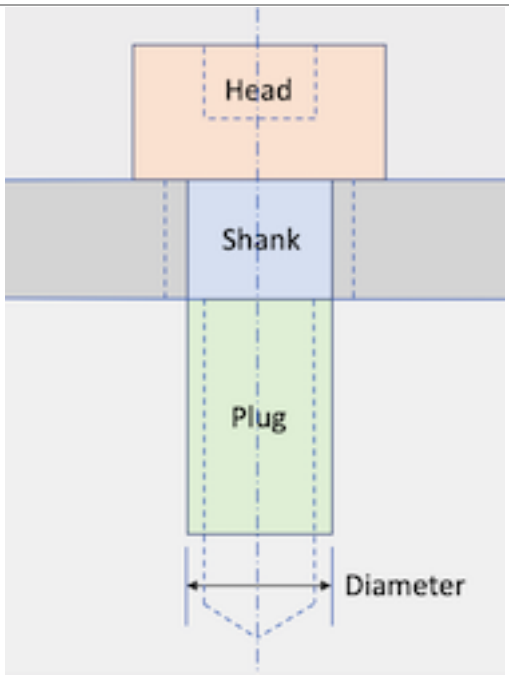
reduce

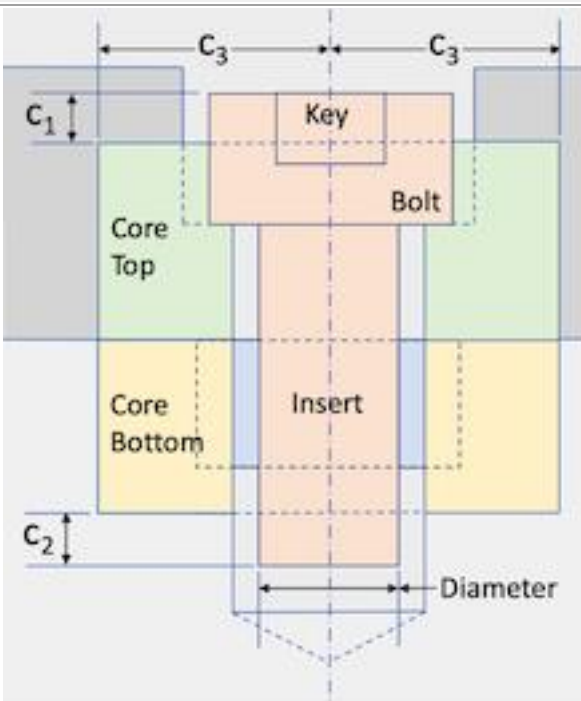
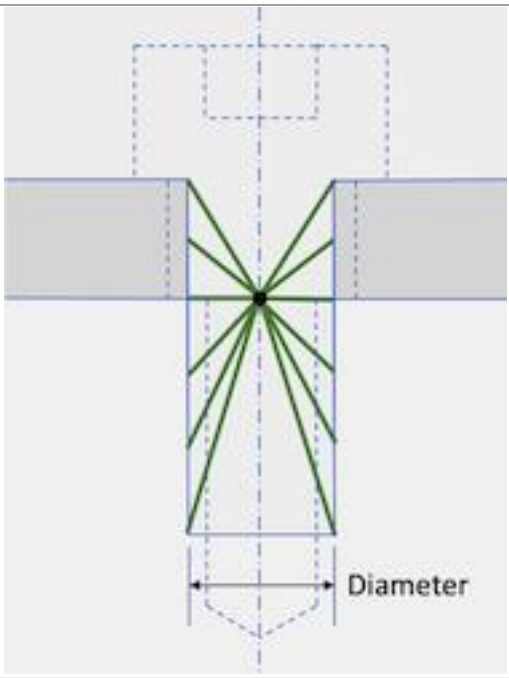
# *Reducing Geometry*

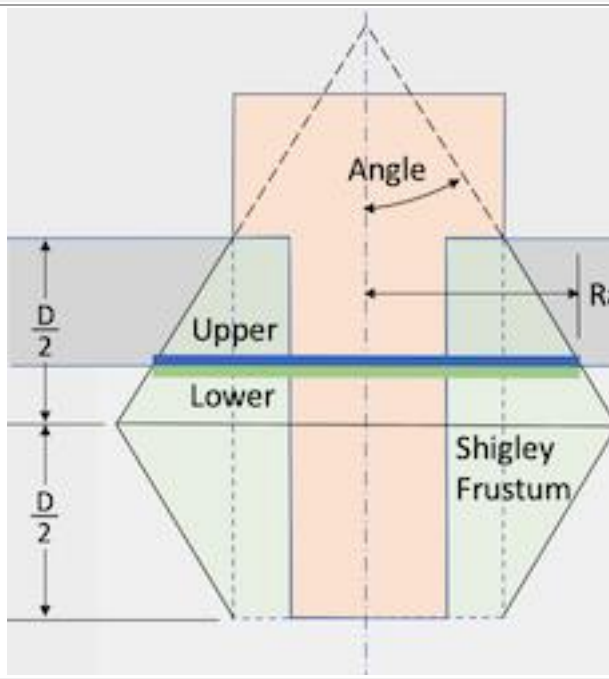
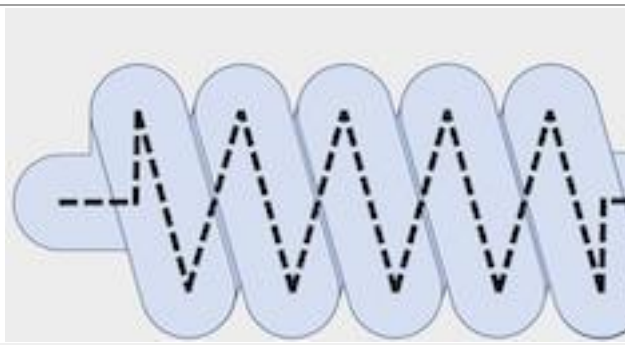
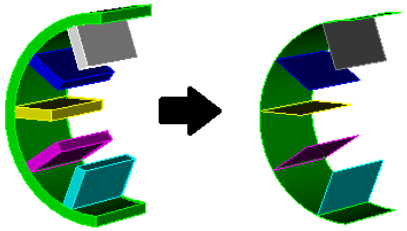
The **reduce** options provide automatic defeaturing and simplification solutions for specific classes of geometry. These options are most often used as methods for rapidly modifying geometry representing fasteners and springs to representations that can be readily used in analysis. For example, the CAD representation of a bolt may include threads, fillets, chamfers, cavities and may also overlap surrounding geometry. The **reduce** option can automatically defeature, webcut and modify surrounding geometry as well as mesh and apply boundary conditions according to a designated *recipe*. A limited number of reduce recipes are currently supported for bolts, including the following:

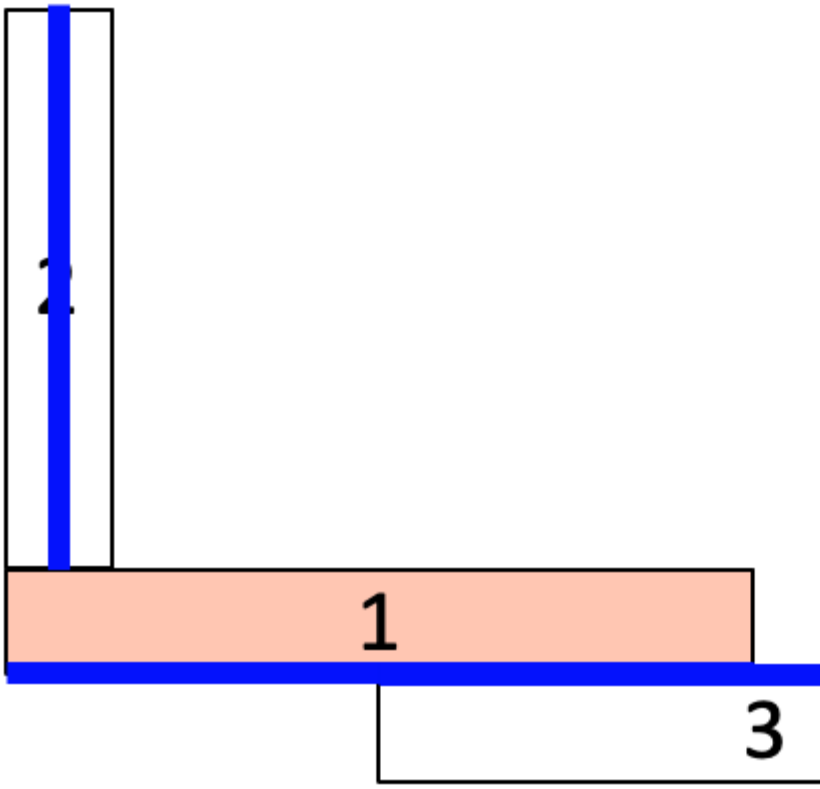
- Reduce Simplify
- Reduce Bolt Fit Volume
- Reduce Bolt Core
- Reduce Bolt Spider
- Reduce Spring
- Reduce Thin Volumes
- Reduce Thin Volumes with Reinforcement Learning
- Reduce Slot Surface

| **Reduce Recipe** | **Command Syntax** | Description | |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Reduce Bolt (Simplify)** | **reduce volume <id_list>** | Simplify/Defeature bolt geometry only |  |
| **Reduce Bolt Fit Volume** | **reduce volume <id_list> bolt fit_volume** | Simplify bolt and fit to surrounding geometry |  |

| | | | | |
|---|---|---|---|---|
| **Reduce Bolt Core** | **reduce volume <id_list> bolt core** | Simplify bolt and insert surrounding core geometry | |  |
| **Reduce Bolt Spider** | **reduce volume <id_list> bolt spider** | Remove bolt and replace with spider mesh | |  |

| | | | |
|---|---|---|---|
| **Reduce Bolt Patch** | **reduce volume <id_list> bolt patch** | Remove bolt and replace with a circular sideset patch |  |
| **Reduce Spring** | **reduce volume <id_list> spring** | Creates curve(s) following center of spring cross-section |  |
| **Reduce Thin Volume** | **reduce volume <id_list> thin auto** | Reduces the thin 3D volumes to a connected set of 2D shell surfaces |  |

| | | | |
|---|---|---|---|
| **Reduce Thin Volume RL** | **reduce volume <id_list> thin RL** | Uses machine learning and reinforcement learning to reduce thin 3D volumes to a connected set of 2D shell surfaces. See also Beam and Shell Modeling with the Geometry power Tool |  |
| **Reduce Slot Surface** | **reduce surface <id> slot** | Provides decomposition solutions for Electromagnetics simulation for decomposing and preparing slot surfaces for analysis. See also Slot Surface Preparation with the Geometry power Tool | |

The **reduce** command is often used in conjunction with the Geometry Power tool and the machine learning classification methods. The classification tool can group volumes according to commonly recognized shapes such as bolt, nut, washer, spring, etc.

# *Reduce (Simplify)*

The simplest form of the **Reduce** command. This option will simplify and defeature volumes without affecting the surrounding geometry.

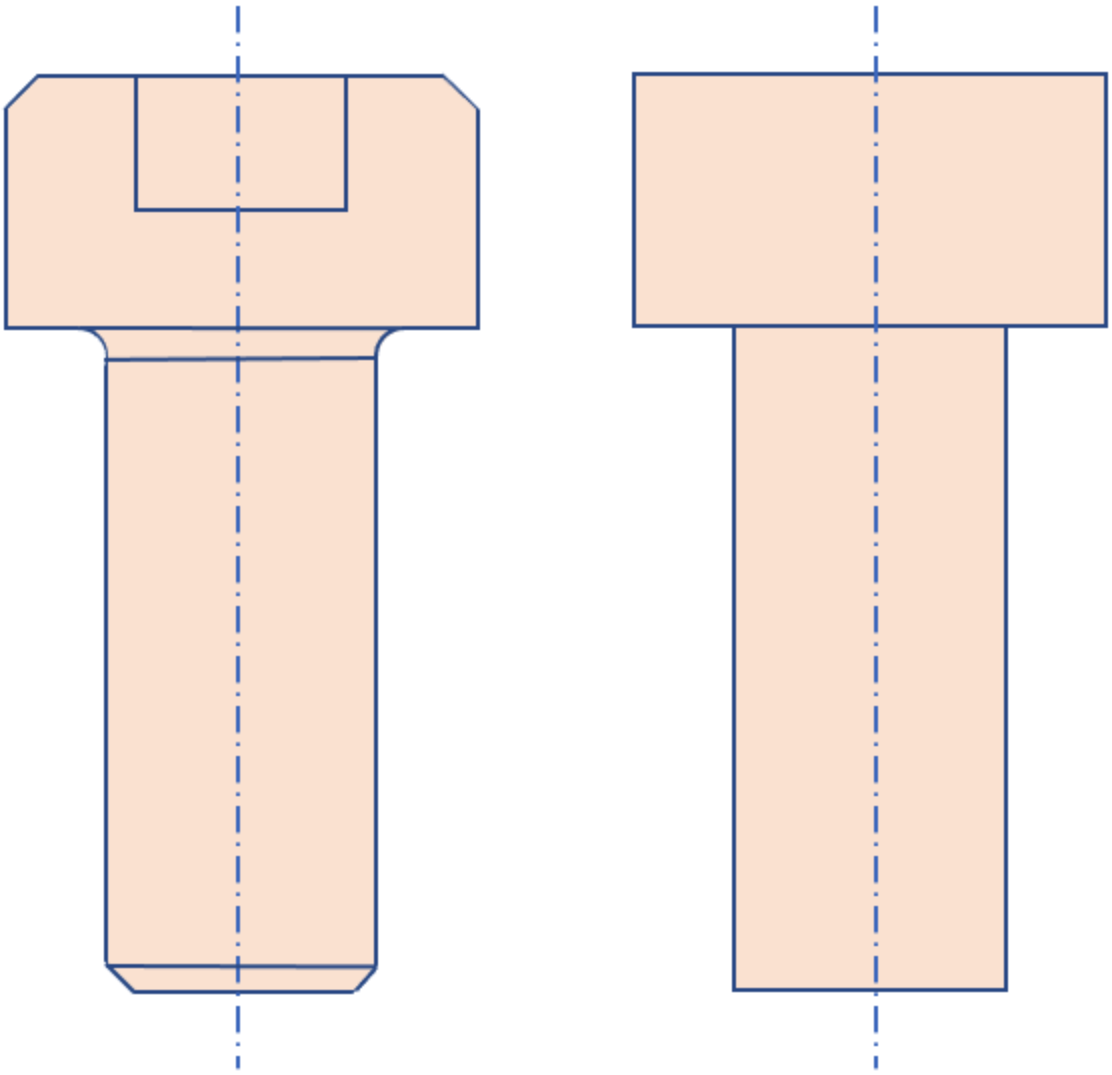


**Figure 1. Example before and after of the Reduce command**

# Syntax:

**Reduce volume {<ids>} [preview]**

# Discussion:

The **reduce** command without additional options will perform simplification and defeaturing operations on the given volumes. This includes automatic chamfer, blend, cavity and small surface removal. While intended primarily for the bolt and fastener use case, it can also be useful for other volume types, however results may vary depending on complexity of the geometry.

**volume ids**: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "bolts".
**preview**: optional argument to display a preview of the reduce operation without execution of the reduction. This option will display the reduced volume in blue with the surrounding volumes displayed in wireframe.

# *Reduce Bolt Fit_Volume*

The **Reduce Bolt** command is intended to prepare a volume identified as a **bolt** for analysis by simplifying its geometry, fitting to overlapping geometry, creating blocks and groups, etc. The **fit_volume** option can also modify the surrounding hole geometry or optional **insert** which may be at the bolt shaft. In addition to simplifying the bolt and insert geometry, this operation can remove any overlap and fit the shaft geometry to its surrounding volume or insert. See also the **reduce bolt core** option which adds the ability to generate a core region surrounding the bolt.

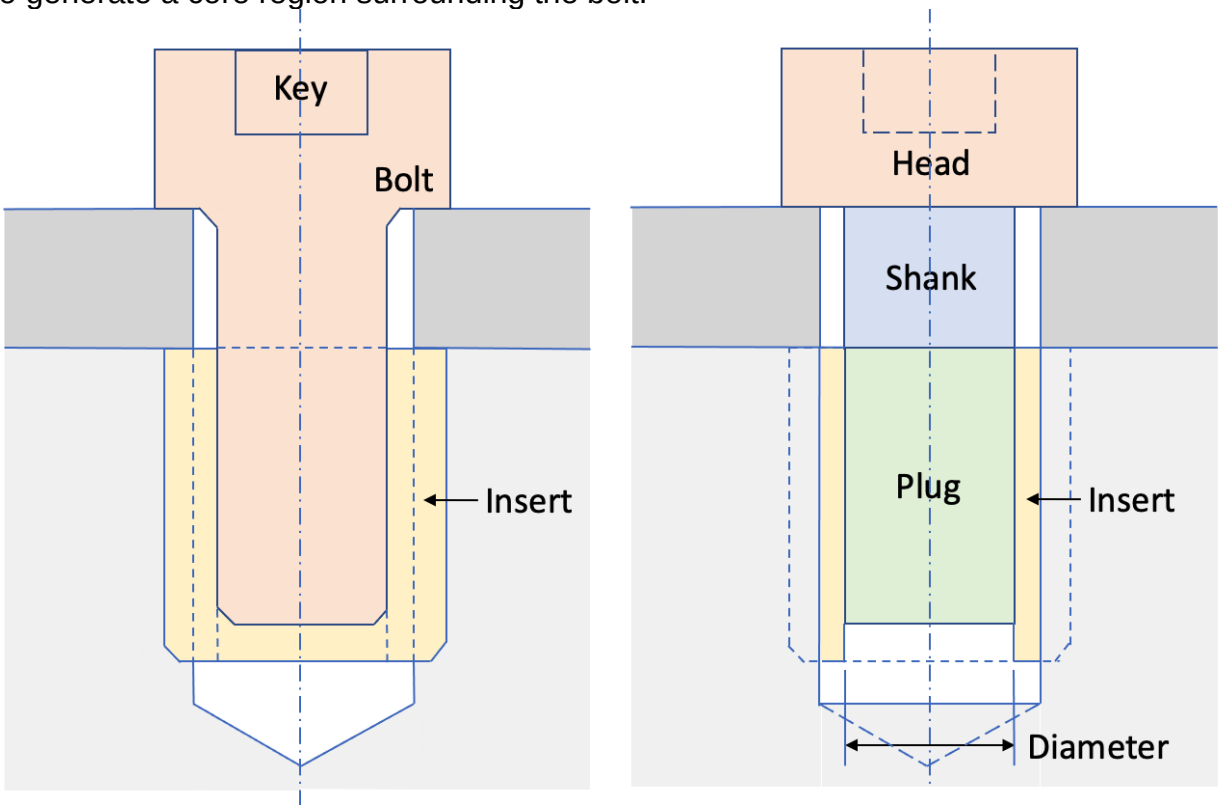



**Figure 1. Example before and after of the Reduce Bolt Fit_Volume command. Also**

**shows optional insert geometry at the bolt shaft.**

**Syntax:**

```
 Reduce {volume <ids>} bolt fit_volume
[insert {volume <ids>}] [diameter
{<value>|auto}] [align_axis] [tight_fit]
[adjust_hole_diameter] [simplify_hole]
[remove_key] [merge] [webcut
[{Head|Shank|BOTH}]] [mesh]
[mesh_size <value>] [bolt_block_id
{<value>|Default}]
[increment_bolt_block_id]
[bolt_block_name {<string>|Default}]
[head_block_id {<value>|Default}]
[increment_head_block_id]
[head_block_name {<string>|Default}]
[shank_block_id {<value>|Default}]
[increment_shank_block_id]
[shank_block_name {<string>|Default}]
[plug_block_id {<value>|Default}]
[increment_plug_block_id]
[plug_block_name {<string>|Default}]
```

**[insert_block_id {<value>|Default}]**
**[increment_insert_block_id]**
**[insert_block_name {<string>|Default}]**
**[preview]**

**Discussion:**

**CAD representations of a bolt assembly can often define overlapping volumetric regions or leave gaps between the bolt and its surrounding geometry. It may also include an optional insert, a cylindrical part surrounding the bolt shaft which can also overlap the bolt and surrounding geometry. The fit_volume option is intended to manage this overlap or gap by adjusting the hole or insert diameter and/or removing void space below the bolt.**

**In addition to removing overlap or gap, the bolt can be automatically webcut into three parts: head, shank and plug as shown in figure 1. The resulting webcut volumes can in turn be assigned to**

**blocks and the diameter of the shank can be altered. The plug can also be merged or contiguously meshed with the fastened geometry.**
**The following outlines the options for the reduce bolt fit_volume command.**
**volume ids: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "bolts".**

**Optional Arguments**

**insert {volume <ids>}: An insert is a cylindrical shaped volume which may be surrounding the shaft of the bolt. The top of the insert is usually flush with the lower volume (light grey volume in figure 1). When using the insert option with the ID of the insert volume, the reduce command will automatically simplify the insert geometry, removing any fillets, rounds or small features on the insert.**

**Any overlap or gap between the shaft and surrounding geometry will be removed so that the insert will fit flush with the shaft and bolt hole. The tight_fit and adjust_hole_diameter options cannot be used when including an insert. If the insert option is not used, any existing insert geometry that may be present at the bolt shaft will be ignored.**

**diameter {<value>|auto}: Use the diameter option to alter the diameter of the bolt shank to a specific diameter indicated by <value>. The auto option will change the diameter of the bolt to exactly match the diameter of its hole. If the diameter option is not used, the existing diameter of the bolt will be used.**

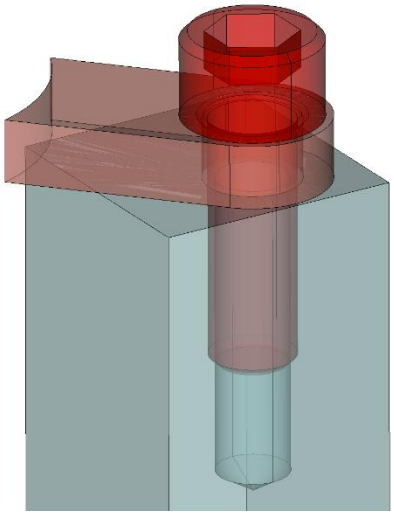


**Figure 2. Bolt geometry prior to reduce operation showing nearby volumes.**

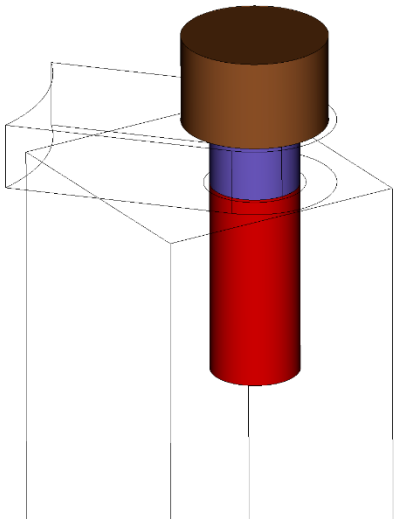


**Figure 3. Bolt after reduce operation using fit_volume, webcut and block assignment options.**

**align_axis: Use this option if the center line of the bolt does not exactly match its hole center line. This option will automatically transform and rotate the bolt volume so that the center line of the hole and bolt are aligned and the bolt head is in contact with the upper volume. tight_fit: This option is normally used when the bolt shaft is overlapping the lower geometry as shown in figure 2. The tight_fit option will perform a boolean subtract operation to ensure that the bolt exactly fits the shaft geometry to the**

**lower volume, removing any gaps and overlaps surrounding or below the bolt. Figure 3 shows the result of the tight_fit option. This option cannot be used if an insert volume is also specified. See figure 4. for an example illustrating the tight_fit option.**

**adjust_hole_diameter: This option is normally used to adjust the diameter of the surrounding hole to match that of the bolt shaft, removing any gap or overlap. In contrast to the tight_fit option, only the hole diameter is tweaked without removing any existing void space below the bolt. This option cannot be used if an insert volume is also specified. See figure 4 for an example illustrating the the adjust_hole_diameter option.**

**simplify_hole: Bolt holes can sometimes include a fillet at the lip of the hole and/or a conical shape indentation at its base. The simplify_hole option will automatically remove these features**

**from the hole leaving a flat base and sharp corner at the lip. Simplify_hole is usually used with the adjust_hole_diameter to ensure the shank fits flush with the hole surfaces and the hole is simplified. Simplify_hole has no effect on the lower volume if the tight_fit option is used. It can however simplify the upper volume if it contains any fillets or chamfers at its lip.**

**remove_key: The bolt geometry can often include a hexagonal shaped hole at the top of the bolt that fits an allen key. By default, this key hole will not be removed. Including the remove_key will ensure this hole is removed from the bolt geometry. If the mesh option is used, a hex mesh can be generated in most cases either with or without the key removed. Hex meshing will most likely be unsuccessful if the diameter of the key hole exceeds that of the bolt shaft. As a consequence,**

**remove_key option may be necessary to facilitate meshing.**

**webcut [{Head|Shank|BOTH}]: The bolt can be cut into up to 3 different volumes as shown in figure 1. Using the optional Head or Shank options a single webcut can be executed separating just the head from the shank or just the shank from the plug respectively. If no arguments are used to the webcut option, or the both option is used, both webcuts will be performed resulting in three volumes. The location of the webcut on the shank will be where the bolt exits the lower volume.**

**merge: When the merge option is used. the plug portion of the bolt will be imprinted and merged with the hole surface(s) in the lower volume. In addition, the plug, shaft and head will be merged together. If an insert volume is specified, the exterior insert surface will be imprinted and merged with the hole**

**surface(s) and the bolt plug will be imprinted and merged with the interior surface(s) of the insert. If the merge option is not used, all volumes created from the reduce command will not be imprinted or merged.**

**mesh: This option will attempt to generate a swept mesh on the bolt and insert geometries as part of the reduce command. If the allen key hole is present in the bolt head, a webcut will be performed on the bolt geometry to facilitate meshing. If meshing is unsuccessful, consider using the remove_key option.**

**mesh_size <value>: Specifies the mesh size to be used with the mesh option. If not specified, an automatic size will be determined.**

**Block ID and name assignment:When the webcut option is used, the resulting volumes may be assigned to a block. The**

**following options may be used to assign to blocks based on an ID or a block name:**

- **head_block_id {<value>|Default}, head_block_name {<string>|Default}:**
- **shaft_block_id {<value>|Default}, shaft_block_name {<string>|Default}:**
- **plug_block_id {<value>|Default}, plug_block_name {<string>|Default}:**

**The blocks may be defined by either a block_name or block_id. If the block does not yet exist, a new one will be created. The Default option will automatically select an id or name.**
**If webcut is not used, the resulting reduced bolt volume may still be assigned to a block id using either the bolt_block_id {<value>|Default} or bolt_block_name {<string>|Default} options.**
**Each of the block ID specifications for bolt, plug, shaft and head also have a corresponding optional increment**

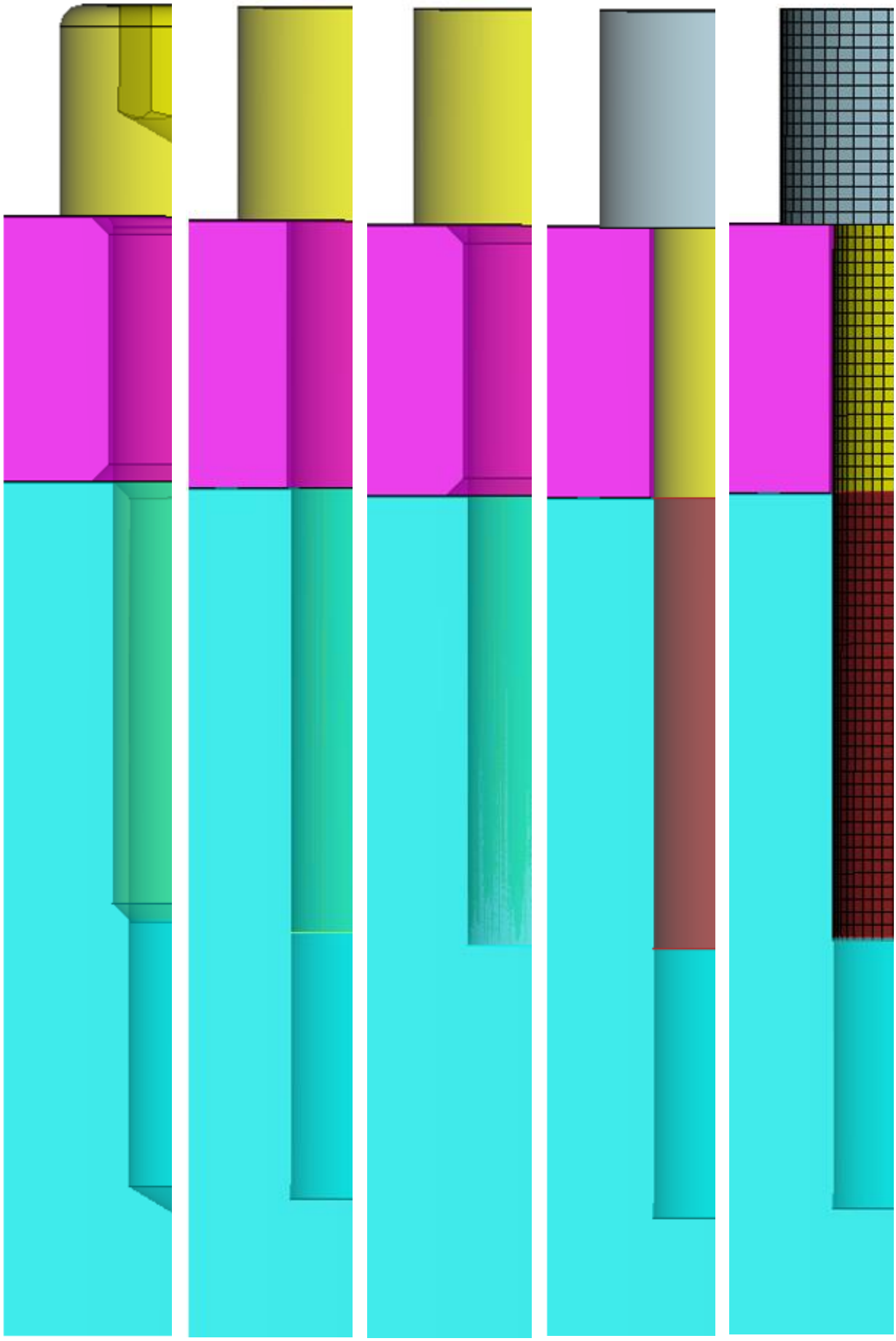
**argument. When assigning new webcut volumes to blocks, the block ID can be automatically generated by incrementing from a specified block_id. For example, if head_block_id is defined as 100, and the increment_head_block_ids option is used, each new head volume generated will be assigned to a new unique block id starting with 100, followed by 101, 102, 103, etc.**

If an **insert** volume is specified, **insert_block_id**, **insert_block_name** and **increment_insert_block_id** may also be used in a similar manner to set up block information on the insert volume.

**preview**: optional argument to display a preview of the **reduce bolt fit_volume** operation without execution of the reduction. This option will display the reduced bolt geometry in blue with the surrounding volumes displayed in wireframe.

# Examples

The following figures illustrate variations on the **fit_volume** option for the **reduce bolt** command. The original CAD geometry is pictured on the left with results on the right.



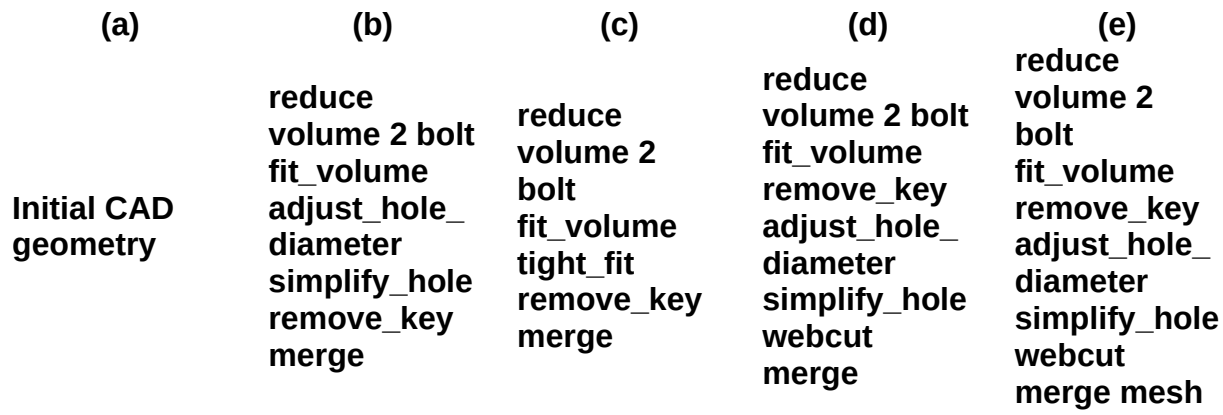
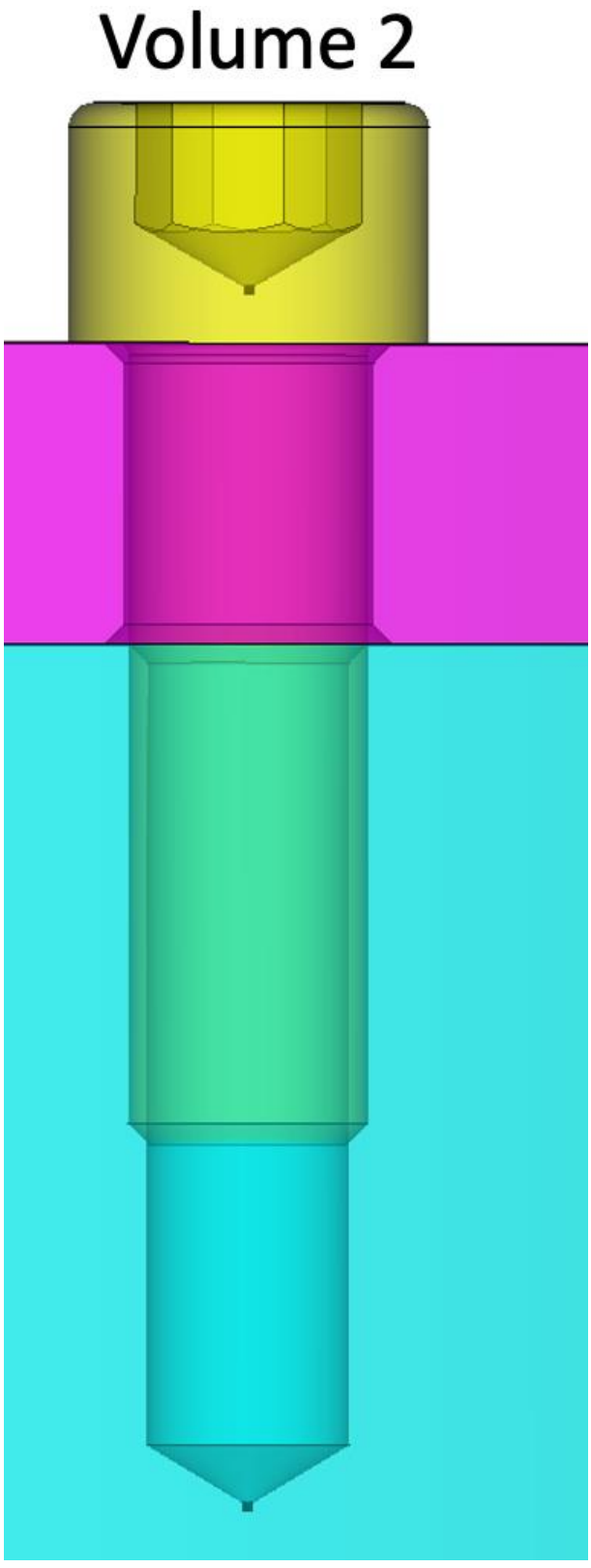
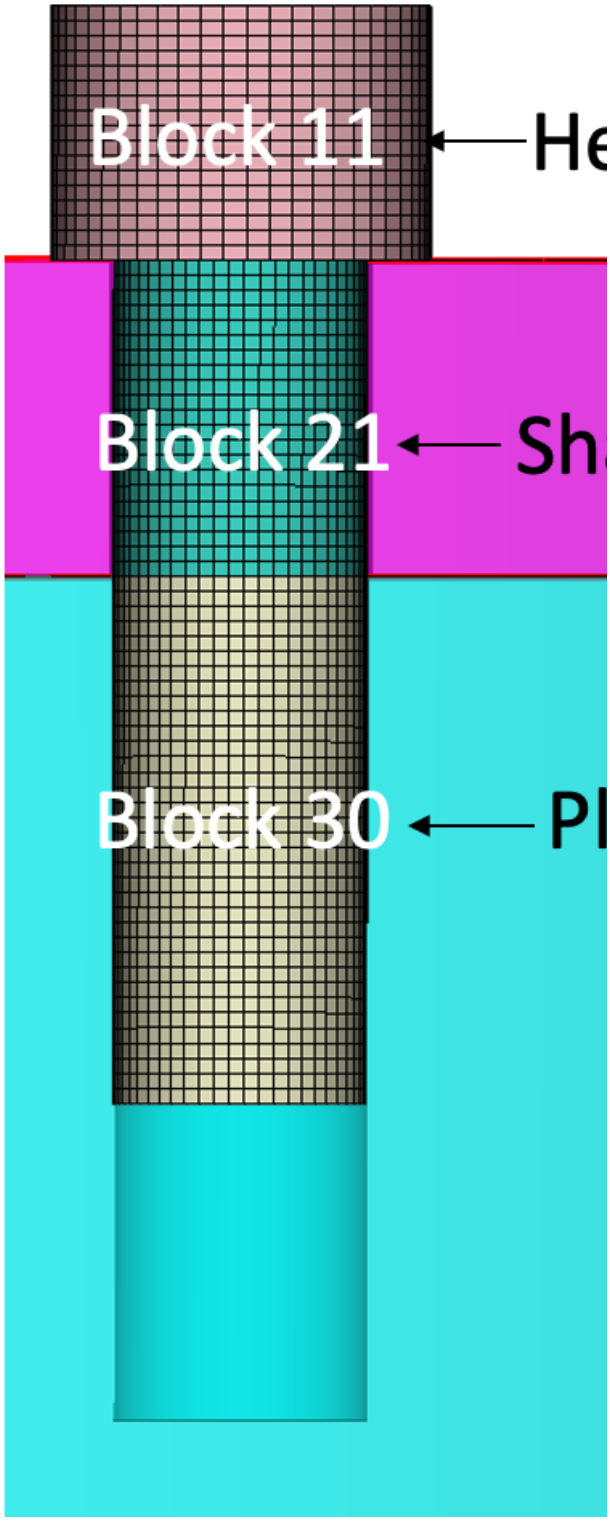



**Figure 4. Example of four different variations of syntax for the reduce bolt fit_volume command on a single bolt. In this case, the bolt (volume 2) overlaps the cyan colored lower volume. Note the difference between the adjust_hole in figures (b), (d) and (e) and tight_fit (c) options. In both cases, volume overlap is removed, however adjust_hole_diameter only adjusts the hole diameter to match the bolt shaft, but tight_fit also removes any void space below the bolt. Also note in figure (d), three separate volumes are created from the bolt geometry when the webcut option is used. In figure (e), the bolt has been meshed with default sizing using Cubit's sweep tool**




(a)




(b)

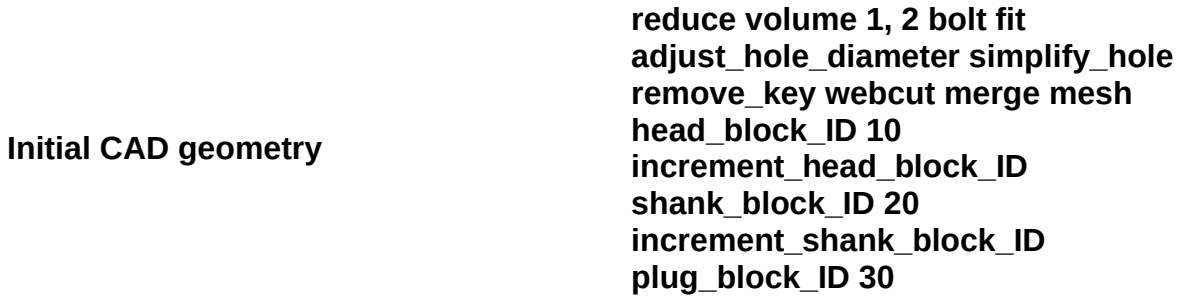
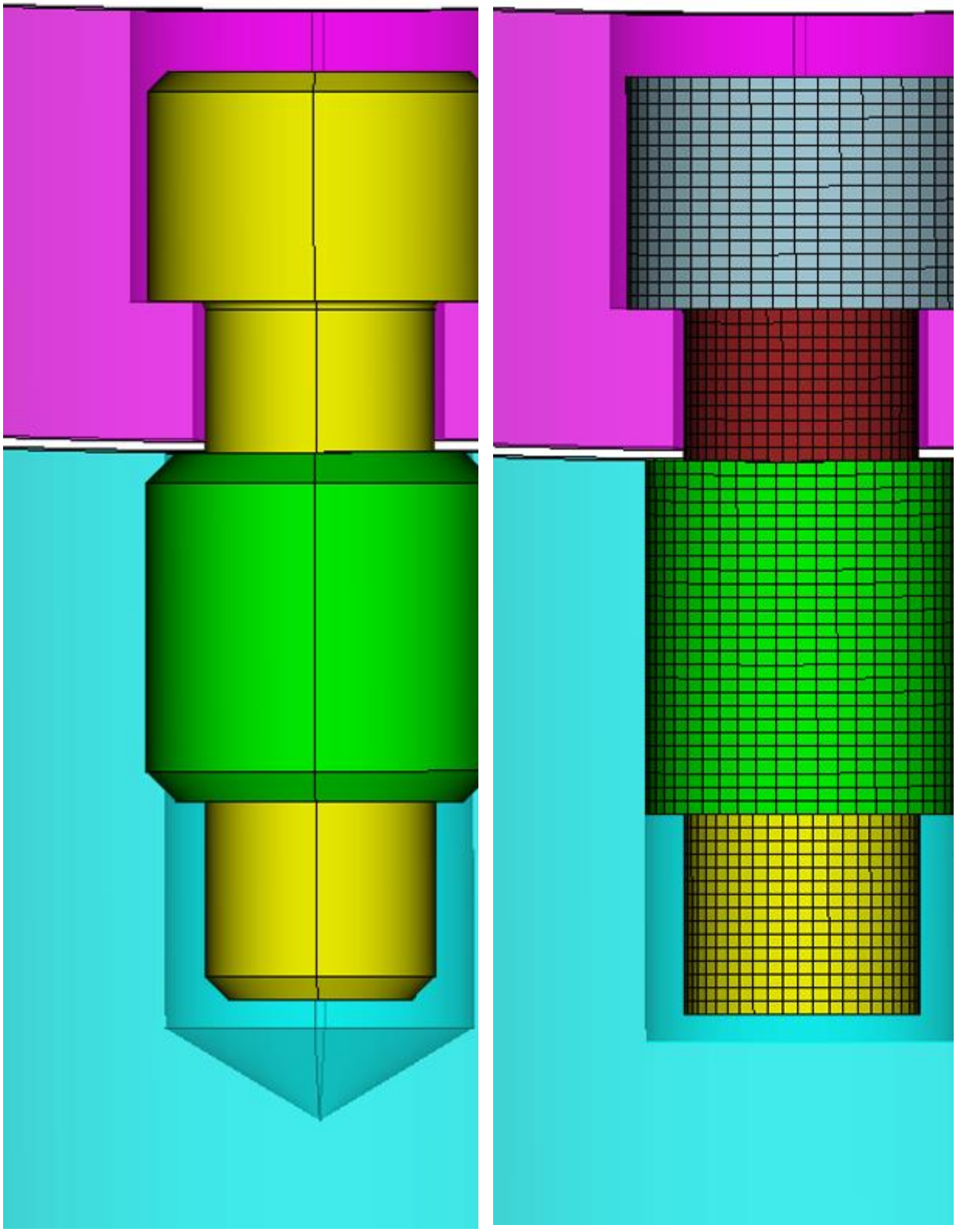



**Figure 5. Example of using the reduce bolt fit_volume command to reduce multiple bolts with a single command while specifying blocks. In this case the block IDs are specified for head, shank and plug volumes. Note that the increment_head_block_ID and increment_shank_block_ID options are used, but the increment_plug_block_ID option is not. The result is that the block IDs on the heads (block 10, 11) and shanks (blocks 20, 21) are incremented with each new bolt, while the block defined on all plugs will remain the same (block 30) without incrementing.**



**(a)** **(b)**

**Initial CAD geometry**

**reduce volume 2 bolt fit_volume insert volume 1 simplify_hole webcut merge mesh**

# Figure 6. Example of using the reduce bolt fit_volume command with the insert option. In this case, volume 1 is specified as the insert and volume 2 is the bolt. Note that both the bolt and insert volumes have been simplified and the insert volume imprinted and merged with both the hole surfaces and the bolt plug.

## *Reduce Bolt Core*

The **Reduce Bolt** command is intended to prepare a volume identified as a **bolt** for analysis by simplifying its geometry, fitting to overlapping geometry, creating blocks and groups, etc. For the **core** option of the **reduce** command, a cylindrical geometry surrounding the bolt will be webcut from the surrounding geometry. The core geometry is often used to define a higher resolution hex mesh than the surrounding geometry to better capture potential failure conditions. The **core** option can also manage corresponding "insert" volumes surrounding the shaft of the bolt. The **reduce bolt core** command is similar to the **reduce bolt fit_volume** command except that the **core** option adds the ability to define a core region surrounding the bolt.

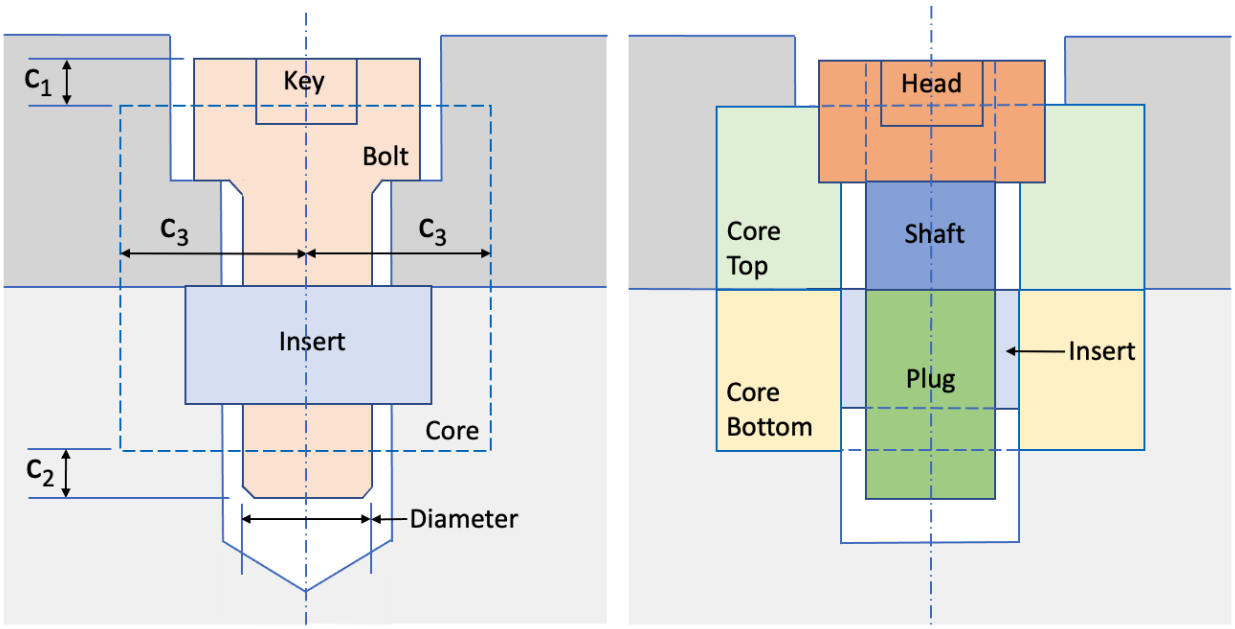



**Figure 1. Example before and after of the Reduce Bolt Core command**

## Syntax:

**Reduce volume {<ids>} bolt core [insert volume {<ids>}] [c1 <value>] [c2 <value>] [c3 <value>] [diameter {<value>|auto}] [align_axis] [tight_fit] [adjust_hole_diameter] [simplify_hole] [remove_key] [merge] [webcut [{Head|Shank|BOTH}]] [mesh] [mesh_size <value>] [bolt_block_id {<value>|Default}] [bolt_block_name {<string>|Default}]**

[increment_bolt_block_id]
[head_block_id {<value>|Default}]
[head_block_name {<string>|Default}]
[increment_head_block_id]
[shank_block_id {<value>|Default}]
[shank_block_name {<string>|Default}]
[increment_shank_block_id]
[plug_block_id {<value>|Default}]
[plug_block_name {<string>|Default}]
[increment_plug_block_id]
[insert_block_id {<value>|Default}]
[insert_block_name {<string>|Default}]
[increment_insert_block_id]
[core_top_block_id {<value>|Default}]
[core_top_block_name
{<string>|Default}]
[increment_bolt_core_top_id]
[core_bottom_block_id
{<value>|Default}]
[core_bottom_block_name
{<string>|Default}]
[increment_core_bottom_block_id]
[preview]

# Discussion:

The dimensions of the core geometry are defined relative to the bolt that it surrounds. Two core volumes are normally generated where the cylindrical core geometry is subtracted from the top and bottom volumes as illustrated in figure 1. In addition to generating the core volumes, the bolt can be automatically webcut into three parts: head, shank and plug. If an insert volume is present, it can be automatically modified to remove overlap and merged to the shank and hole surfaces. In addition the diameter of the bolt shank can be altered, the hole fit to the shank diameter and the the allen key cavity optionally removed. The resulting volumes can in turn be hex meshed and assigned to blocks.

An example of the **reduce bolt core** operation is illustrated in figures 2 and 3. In this example, an "insert" volume is present that is overlapping the hole geometry. The resulting geometry from the operation is shown in figure 3.

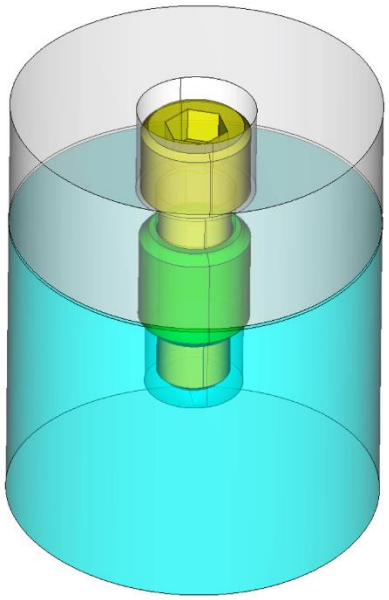


**Figure 2. Bolt and insert geometry prior to reduce bolt core operation showing nearby volumes.**

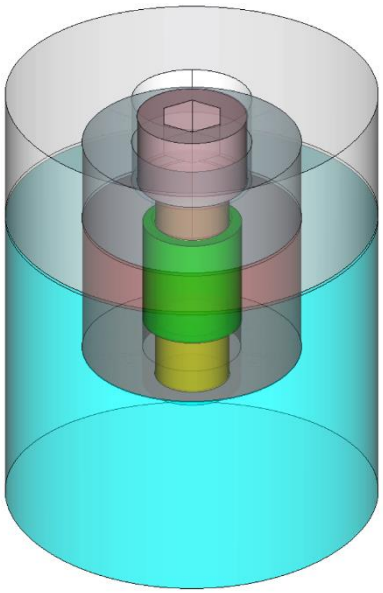


**Figure 3. Bolt, insert and core volumes after the reduce operation**

The **reduce bolt core** operation also provides the option to assign the resulting volumes to blocks and mesh the volumes at a specified size. Figure 4 illustrates the resulting default blocks and mesh denerated from the example shown in figure 3.

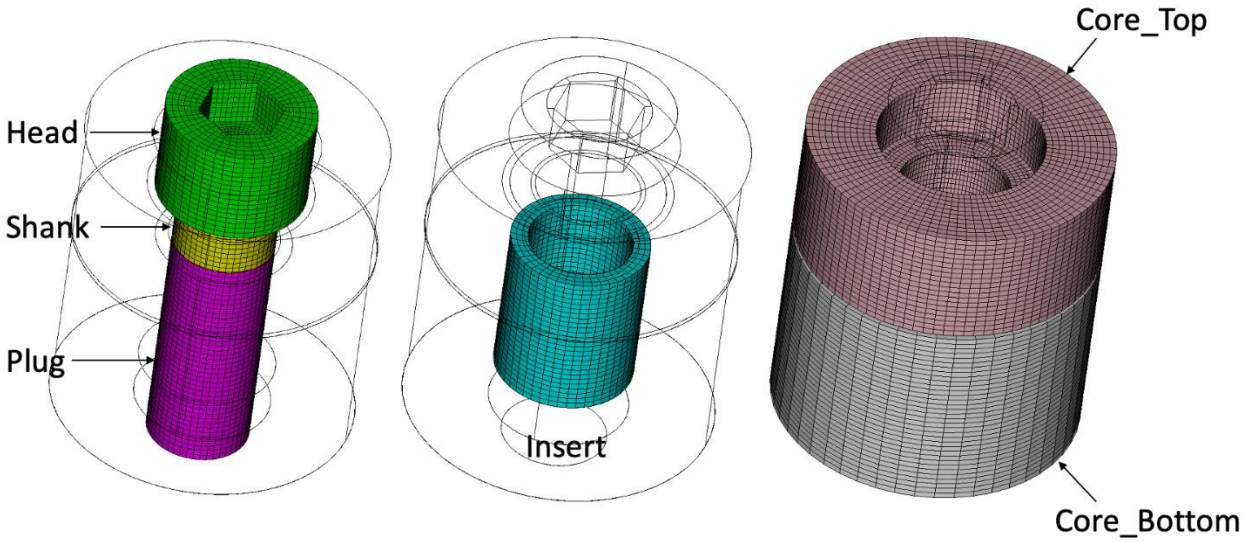



**Figure 4. Example blocks and mesh automatically generated from the Reduce Bolt Core command**

The following outlines the options for the **reduce bolt fit_volume** command.
**volume ids**: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "bolts".

# Optional Arguments

**insert {volume <ids>}**: An insert is a cylindrical shaped volume which may be surrounding the shaft of the bolt. The top of the insert is usually flush with the lower volume (light grey volume in figure 1). When using the **insert** option with the ID of the insert volume, the **reduce** command will automatically simplify the insert geometry, removing any fillets, rounds or small features on the insert. Any overlap or gap between the shaft and surrounding geometry will be removed so that the insert will fit flush with the shaft and bolt hole. The **tight_fit** and **adjust_hole_diameter** options cannot be used when including an **insert**. If the **insert** option is not used, any existing insert geometry that may be present at the bolt shaft will be ignored.
**c1 <value>, c2 <value>, c3 <value>**: Dimensions relative to the bolt defining the size of the core geometry as illustrated in Figure 1.

1. **c1 <value>**: Distance from the top of the bolt head to the top of the core geometry.
2. **c2 <value>**: Distance from the bottom of the bolt shaft to the bottom of the core geometry.
3. **c3 <value>**: Radius of core relative to the center line of the bolt.

**diameter {<value>|auto}: Use the diameter option to alter the diameter of the bolt shank to a specific diameter indicated by <value>. The auto option will change the diameter of the bolt to exactly match the diameter of its hole. If the diameter option is not used, the existing diameter of the bolt will be used.**

**align_axis: Use this option if the center line of the bolt does not exactly match its hole center line. This option will automatically transform and rotate the bolt volume so that the center line of the hole and bolt are aligned and the bolt head is in contact with the upper volume.**

**tight_fit**: This option is normally used when the bolt shaft is overlapping the lower geometry as shown in figure 2. The **tight_fit** option will perform a boolean subtract operation to ensure that the bolt exactly fits the shaft geometry to the lower volume, removing any gaps and overlaps surrounding or below the bolt. Figure 3 shows the result of the **tight_fit** option. This option cannot be used if an **insert** volume is also specified. See figure 4. for an example illustrating the **tight_fit** option.

**adjust_hole_diameter**: This option is normally used to adjust the diameter of the surrounding hole to match that of the bolt shaft, removing any gap or overlap. In contrast to the **tight_fit** option, only the hole diameter is tweaked without removing any existing void space below the bolt. This option cannot be used if an **insert** volume is also specified. See figure 4 for an example illustrating the the **adjust_hole_diameter** option.

**simplify_hole**: Bolt holes can sometimes include a fillet at the lip of the hole and/or a conical shape indentation at its base. The **simplify_hole** option will automatically remove these features from the hole leaving a flat base and sharp corner at the lip. **Simplify_hole** is usually used with the **adjust_hole_diameter** to ensure the shank fits flush with the hole surfaces and the hole is simplified. **Simplify_hole** has no effect on the lower volume if

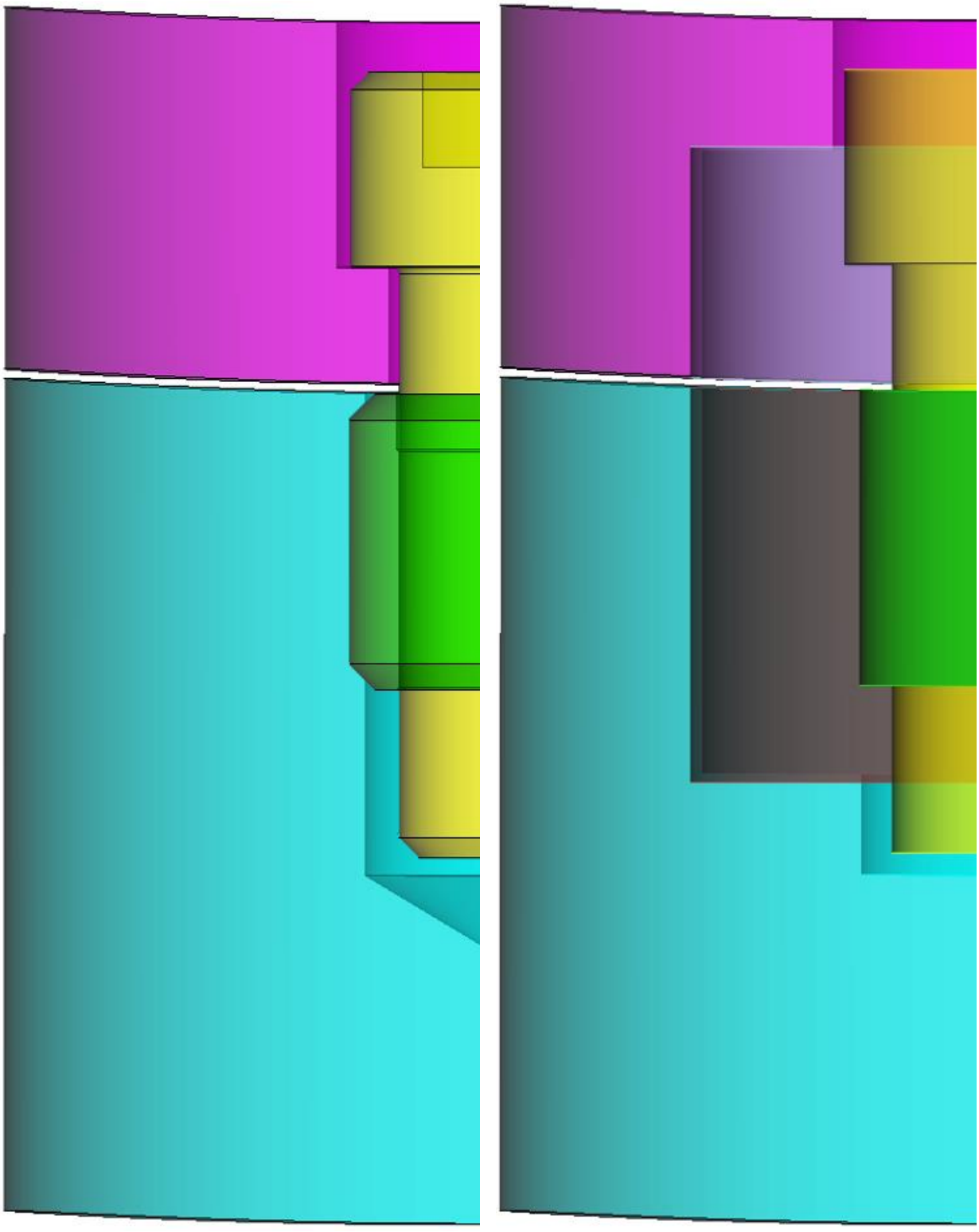
the **tight_fit** option is used. It can however simplify the upper volume if it contains any fillets or chamfers at its lip.
**remove_key**: The bolt geometry can often include a hexagonal shaped hole at the top of the bolt that fits an allen key. By default, this key hole will not be removed. Including the **remove_key** will ensure this hole is removed from the bolt geometry. If the **mesh** option is used, a hex mesh can be generated in most cases either with or without the key removed. Hex meshing will most likely be unsuccessful if the diameter of the key hole exceeds that of the bolt shaft. As a consequence, **remove_key** option may be necessary to facilitate meshing.
**webcut [{Head|Shank|BOTH}]**: The bolt can be cut into up to 3 different volumes as shown in figure 1. Using the optional **Head** or **Shank** options a single webcut can be executed separating just the head from the shank or just the shank from the plug respectively. If no arguments are used to the **webcut** option, or the **both** option is used, both webcuts will be performed resulting in three volumes. The location of the webcut on the shank will be where the bolt exits the lower volume.
**merge**: When the **merge** option is used. the plug portion of the bolt will be imprinted and merged with the hole surface(s) in the lower volume. In addition, the plug, shaft and head will be merged together. If an insert volume is specified, the exterior insert surface will be imprinted and merged with the hole surface(s) and the bolt plug will be imprinted and merged with the interior surface(s) of the insert. If the **merge** option is not used, all volumes created from the **reduce** command will not be imprinted or merged.
**mesh**: This option will attempt to generate a swept mesh on the bolt and insert geometries as part of the **reduce** command. If the allen key hole is present in the bolt head, a webcut will be performed on the bolt geometry to facilitate meshing. If meshing is unsuccessful, consider using the **remove_key** option.
**mesh_size <value>**: Specifies the mesh size to be used with the **mesh** option. If not specified, an automatic size will be determined.
**Block ID and name assignment**:When the **webcut** option is used, the resulting volumes may be assigned to a block. The following options may be used to assign to blocks based on an ID or a block name:

- **head_block_id {<value>|Default}, head_block_name {<string>|Default}:**
- **shaft_block_id {<value>|Default}, shaft_block_name {<string>|Default}:**
- **plug_block_id {<value>|Default}, plug_block_name {<string>|Default}:**

The blocks may be defined by either a **block_name** or **block_id**. If the block does not yet exist, a new one will be created. The **Default** option will automatically select an id or name.
If **webcut** is not used, the resulting reduced bolt volume may still be assigned to a block id using either the **bolt_block_id {<value>|Default}** or **bolt_block_name {<string>|Default}** options.
Each of the block ID specifications for bolt, plug, shaft and head also have a corresponding optional **increment** argument. When assigning new webcut volumes to blocks, the block ID can be automatically generated by incrementing from a specified block_id. For example, if **head_block_id** is defined as **100**, and the **increment_head_block_ids** option is used, each new **head** volume generated will be assigned to a new unique block id starting with **100**, followed by **101**, **102**, **103**, etc.
If an **insert** volume is specified, **insert_block_id**, **insert_block_name** and **increment_insert_block_id** may also be used in a similar manner to set up block information on the insert volume.
**preview**: optional argument to display a preview of the **reduce bolt fit_volume** operation without execution of the reduction. This option will display the reduced bolt geometry in blue with the surrounding volumes displayed in wireframe.

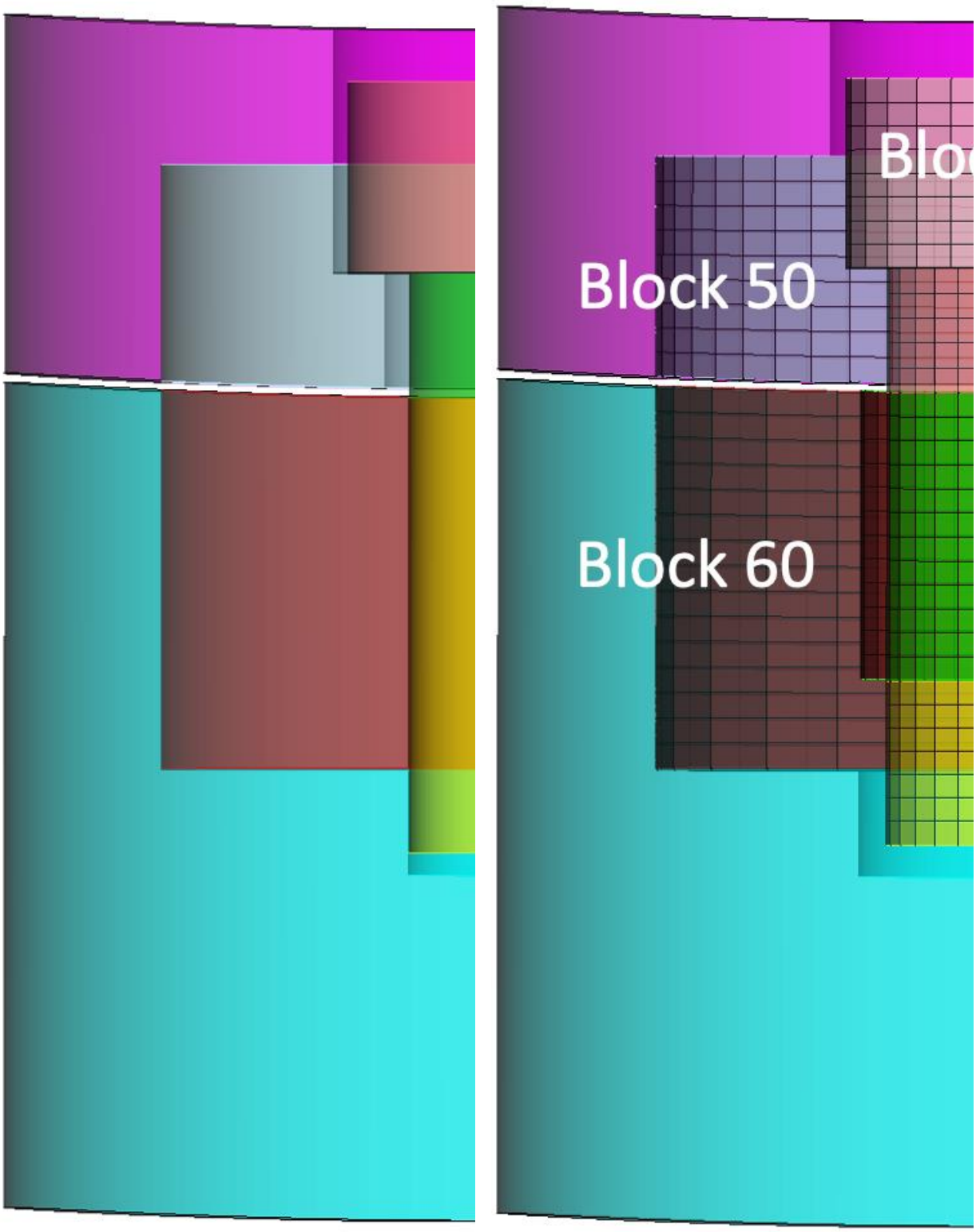
# Examples

The following figures illustrate variations on the **core** option for the **reduce bolt** command.



**(a)** **(b)**

| | |
|---|---|
| **Initial CAD geometry** | **reduce volume 2 bolt core insert volume 1 c1 0.936345 c2 0.936345 c3 3.4749 simplify_hole remove_key merge** |




**(c)** **(d)**

**reduce volume 2 bolt core c1 1 c2 1 c3 4 diameter 2 adjust_hole_diameter simplify_hole remove_key webcut merge**

**reduce volume 2 bolt core insert volume 1 c1 1 c2 1 c3 4 diameter 2 simplify_hole remove_key webcut merge mesh mesh_size 0.3 head_block_ID 10 shank_block_ID 20 plug_block_ID 30 insert_block_ID 40 core_top_block_ID 50 core_bottom_block_ID 60**

**Figure 5. Examples of using the reduce bolt core command where (a) is the original geometry. In (b), an insert volume is included and the dimensions of the core are specified. Figure (c) does not include the insert and modifies the dimensions of core using the c1, c2 and c3 options. In addition, the diameter of the bolt shaft has been modified from its original diameter and the bolt cut into head, shank and plug volumes. Figure (d) again includes the insert and also specifies a mesh and custom mesh size to be used. In addition, note that block IDs for each of the resulting volumes have been specified. The resulting block IDs are also illustrated in figure (d).**

## *Reduce Bolt Spider*

The **Reduce Bolt** command is intended to prepare a volume identified as a **bolt** for analysis by replacing the bolt geometry with a set of mesh edges or *beams* and generating blocks containing the mesh. There are currently three different options for the spider command as illustrated in figures 1 to 3.

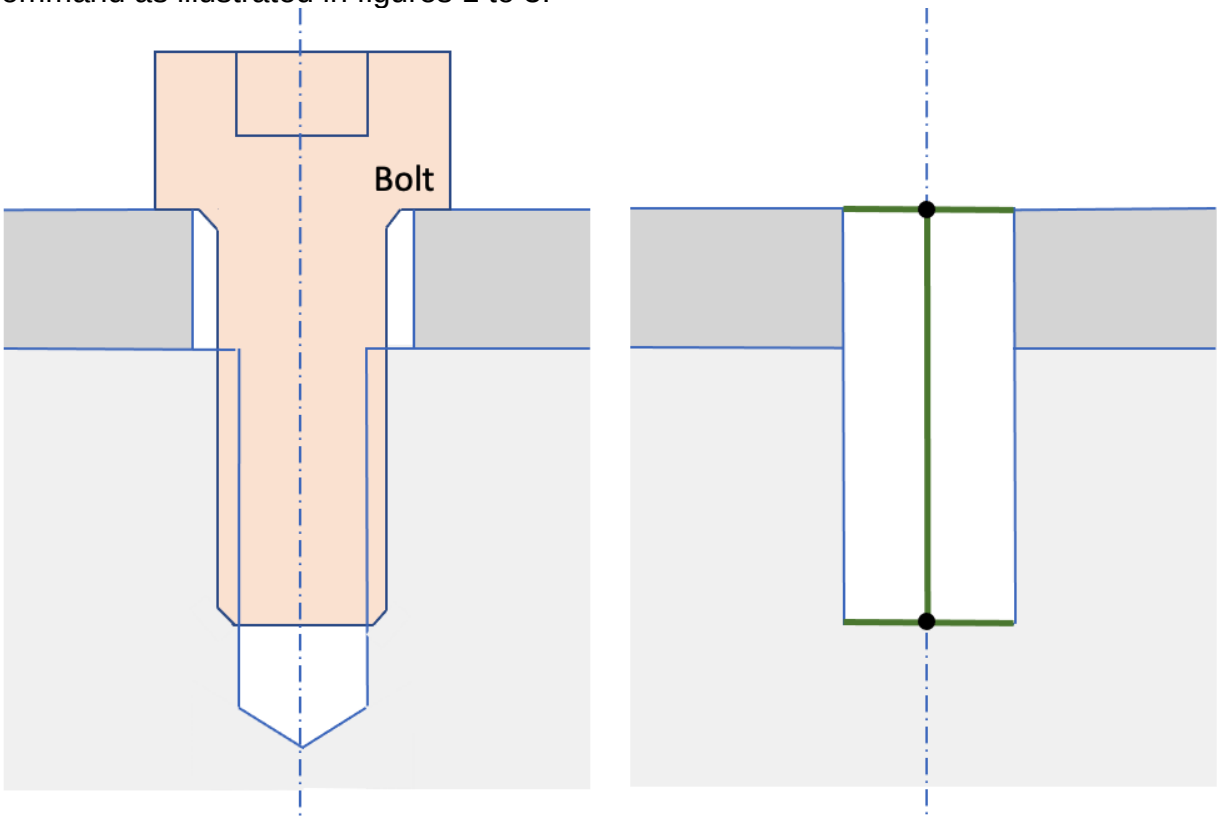



**Figure 1. Example before and after of the Reduce Bolt Spider Wagon_Wheel command**

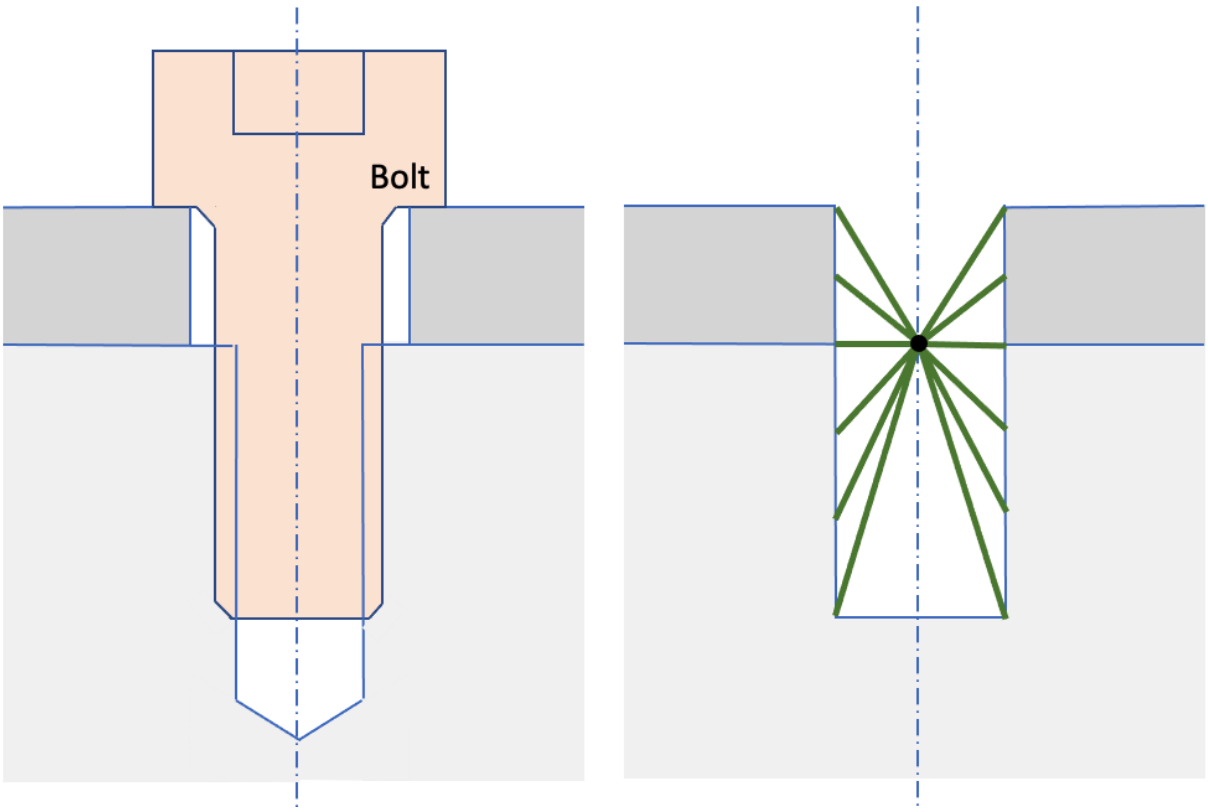



**Figure 2. Example before and after of the Reduce Bolt Spider J2G command**

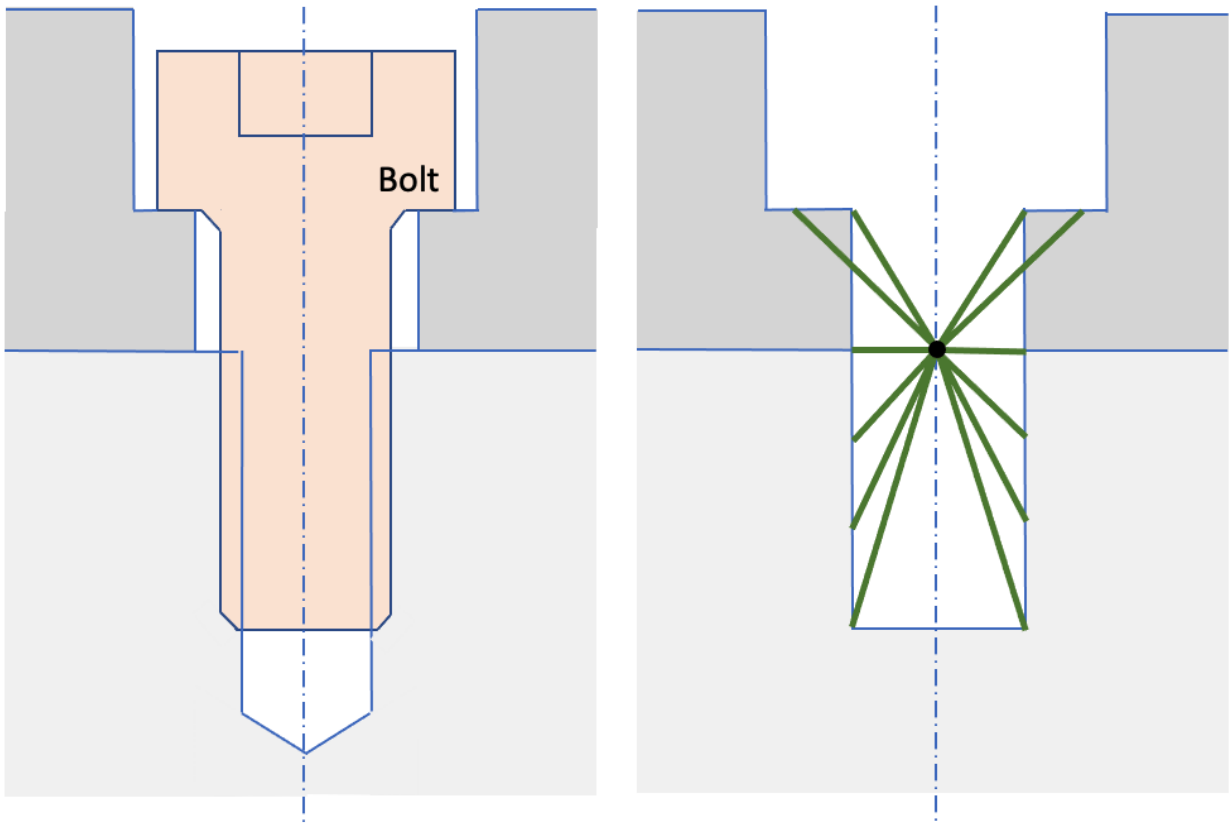



**Figure 3. Example before and after of the Reduce Bolt Spider Countersink command**

**Syntax:**

**Reduce {volume <ids>|{upper surface <ids>lower surface <ids>}} bolt spider {wagon wheel|j2g|countersink} [diameter <value>] [mesh] [mesh_size] [spider_block_id {<value>|Default}] [spider_block_name {<string>|Default}] [increment_spider_block_id] [rebar_block_id {<value>|Default}]**

**[rebar_block_name {<string>|Default}]**
**[increment_spider_block_id]**
**[upper_spider_block_id {|Default}]**
**[increment_upper_spider_block_id]**
**[upper_spider_block_name {|Default}]**
**[lower_spider_block_id {|Default}]**
**[increment_lower_spider_block_id]**
**[lower_spider_block_name {|Default}]**
**[preview]**
**[preview]**

# Discussion:

The **spider** options will also simplify the surrounding geometry at a bolt hole, including removing any blends or chamfers. It will also simplify the hole so in the lower volume to which the bolt is fastened will fit exactly to the bolt geometry or specified diamater.
The **spider** options will not, by default, generate the beam mesh as shown in figures 1 to 3, but instead define blocks to which the beam elements will be added when the mesh is generated or the **mesh** option is used. The following describes the options for the **reduce bolt spider** command.
Bolt holes can be defined by specifing the cylindrical bolt volume or by specifying the upper and lower surfaces of the bolt holes. The latter is helpful if bolt volumes don't exist in the model but the holes do.
**volume ids**: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "bolts".
**upper/lower bolt hole surface ids**: Specify the upper and lower surfaces of the bolt hole. Multiple upper/lower pairs can be specified.
**{wagon wheel|j2g|countersink}**: One of these three methods must be specified for the **spider** option.

1. **wagon wheel**: Shown in figure 1., a center beam element is generated and assigned to the **rebar** block. In addition, beams are generated extending from the ends of the rebar beam radially connecting the top and bottom circular curves of the hole. The radial beam elements are assigned to the **spider** block.
2. **j2g**: Shown in figure 2., a center node is generated and assigned to the **rebar** block. In addition, beams are generated extending from the the rebar node

radially connecting all nodes of the cylindrical surfaces of the hole. The radial beam elements are assigned to the **spider** block.

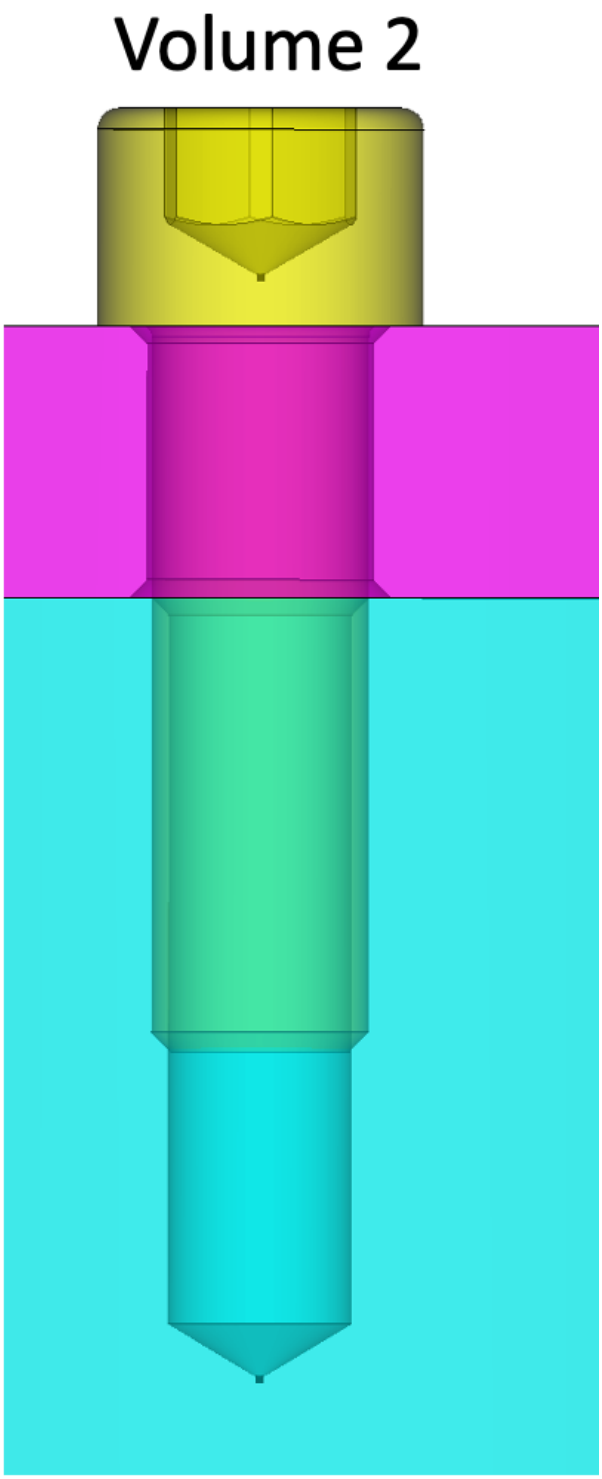
3. **countersink**: Shown in figure 3., similar to the **j2g** option, however beam elements are also generated connecting the outer radius of bolt with the center rebar node.

# Optional Arguments

**diameter <value>**: Use the **diameter** option to alter the diameter of the resulting hole. If no value is specified for **diameter**, the existing diameter of the bolt will be used.
**mesh**: This option can be used to generate the beam mesh as part of the **reduce** operation. Since the resolution of the beam mesh depends on the nodes defined on the hole geometry, the surfaces of the hole will also be meshed using a mapped meshing scheme. Use the **mesh_size** option to control the resolution of the beam mesh. When the hole surfaces are meshed, the beam elements will also be generated and assigned to the existing **spider** and **rebar** blocks. Note that if the **mesh** option is not used, an empty **spider** and **rebar** block will be generated.
**mesh_size**: Use the **mesh_size** option to control the resolution of the beam mesh. If no **mesh_size** has been defined, if a mesh size has been defined on the bolt volume, it will be used, otherwise an automatic default size will be computed and used for meshing.
The **spider** and **rebar** blocks may be defined by either a **block_name** or **block_id**. If the block does not yet exist, a new one will be created. The **Default** option will automatically select an id or name. The default names for the **spider** and **rebar** blocks are **Spider_Block** and **Rebar_Block** respectively.
When building new blocks for each bolt volume, the block IDs can be automatically generated by incrementing from a specified block_id. The **increment_spider_block_id** and **increment_rebar_block_id** options are used for this prupose. For example, if **rebar_block_id** is defined as **100**, and the **increment_rebar_block_ids** option is used, each new **rebar** generated will be assigned to a new unique block id starting with **100**, followed by **101**, **102**, **103**, etc.
Upper and lower portions of the resulting spider can be placed into separate blocks with the **upper_spider_*** and **lower_spider_*** options. (Default is to put the spider joint into a single block.) Names and ids of these upper and lower blocks can be controlled with ***_upper_*** and ***_lower_*** forms of the block name/id parameters detailed in the preceeding paragraph.
**preview**: optional argument to display a preview of the spider operation without execution of the reduction. This option will display the proposed beam mesh with the surrounding volumes displayed in wireframe.

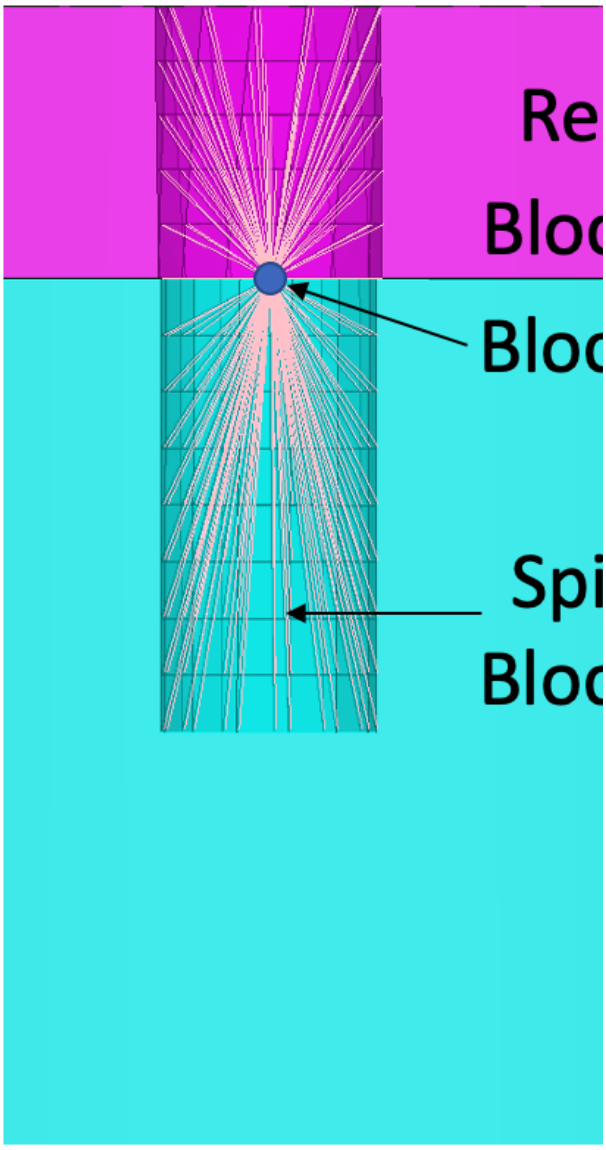
# Examples

The following figure illustrates the **spider** option for the **reduce bolt** command.




(a)

**Initial CAD geometry**




(b)

**reduce volume 1 2 bolt spider J2G mesh mesh_size 3.0 spider_block_ID**

**10 increment_Spider_Block_ID rebar_block_ID 20**



**Figure 4. Examples of using the reduce bolt spider j2g command where (a) is the original geometry of two bolts overlapping the lower volume (cyan color). In (b), the spider j2g option has been used which replaces the bolts with a beam mesh. Note that the bolt hole has been modified to fit the shaft of the original bolts. In this case, the rebar blocks are comprised of the central nodes from which the beam elements radiate. Since the increment_rebar_block_id option is used, block 10 is defined where volume 1 existed and block 11 at volume 2. For the spider block, since the increment_spider_block_id was not used, all beam elements were placed into block 20. Note that since the mesh option was used, the hole surfaces were meshed and the spider block beam elements were generated. Otherwise, a subsequent mesh operation performed on the lower (cyan) volume would also generate the beam**

**elements and assign them to their appropriate blocks.**

## *Reduce Spring*

The **Reduce Spring** command is intended to prepare a volume identified as a **spring** for analysis by creating one or more free curves that follow the spring's path at the middle of its cross-section. The new curve(s) can be meshed with beam elements and used in place of 3D elements to represent the spring.

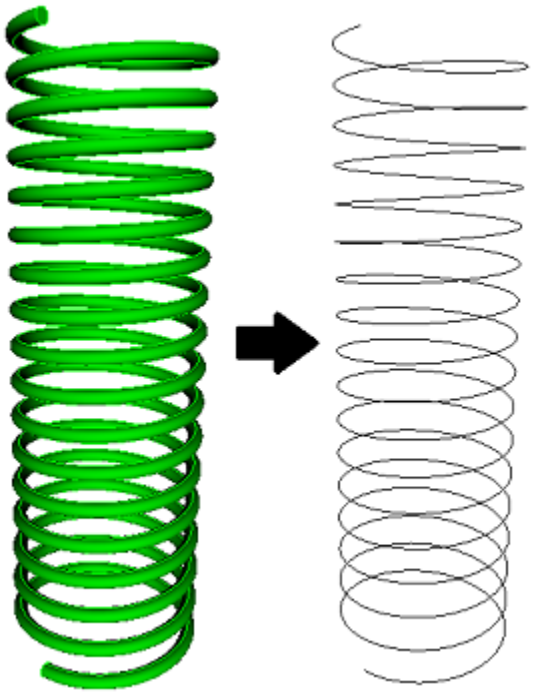


**Figure 1. Example before and after of the Reduce Spring command**

**Syntax:**

**Reduce Volume <ids> Spring [combine] [mesh [size {<value>}]] [keep]**

# [block_id {<value>|Default}] [increment_block_id] [block_name {<string>|Default}] [preview]

## Discussion:

**volume ids**: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "springs".

**combine**: By default, the curves resulting from the **reduce spring** command will be based upon the initial surface geometry of the spring. This may result in multiple curves of varying lengths connected by vertices. The **combine** option reduces the spring to a single free curve. This is helpful if the default operation would otherwise result in short curves that would produce unacceptable beam elements.

**mesh**: Use the **mesh** option to automatically mesh the resulting curve(s) with beam elements. The optional **size** argument can be used to specify a target length for the beam elements along the curve(s). If a **size** is not specified, an automatic size is assigned.

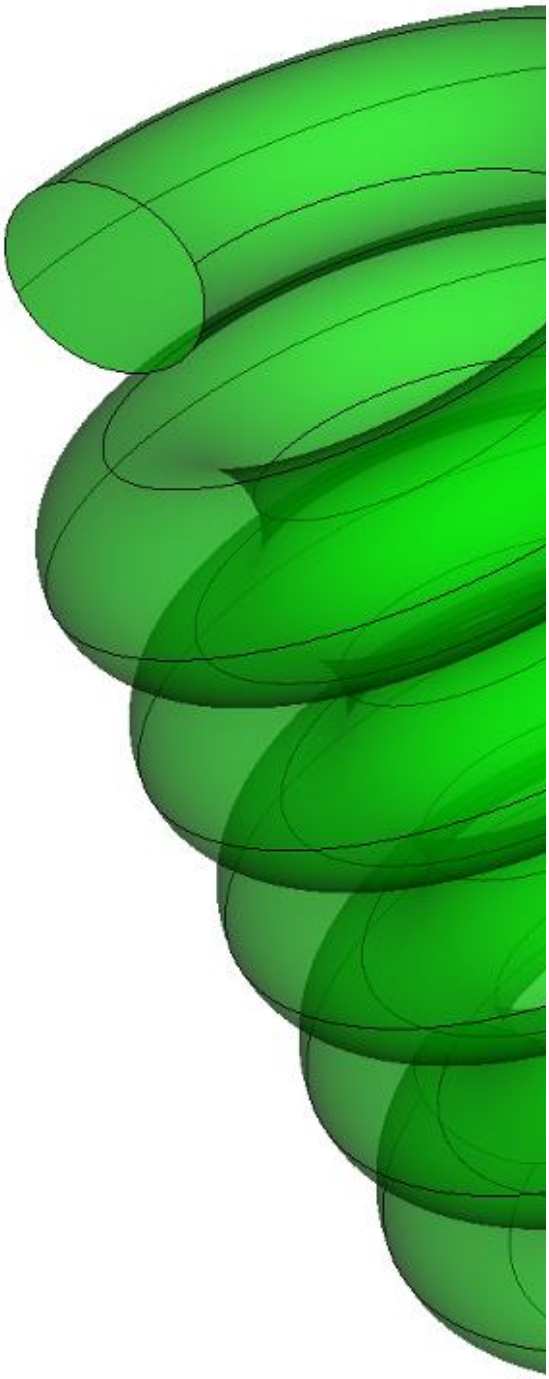
**keep**: Optionally retain the initial spring geometry after the reduce command is completed. If **keep** is not used, the spring volume will be deleted when a valid set of mid-curves has been produced.

**Block ID and name assignment**:When the **block_id** or **block_name** options are used, the resulting curves and mesh may be assigned to a block. The block may be defined by either a **block_name** or **block_id**. If the block does not yet exist, a new one will be created. The **Default** option will automatically select an id or name.

**increment_block_id**:When reducing multiple spring volumes in a single **reduce** command, it may be desirable to change the block ID assignment with each spring. When assigning new curves or beam elements to blocks, the block ID can be automatically generated by incrementing from a specified **block_id**. For example, if **block_id** is defined as **100**, and the **increment_block_ids** option is used, each new set of mid-curves and their associated beam elements generated for a unique spring volume will be assigned to a new block id starting with **100**. The next spring volume will be block **101**, followed by **102**, **103**, etc.

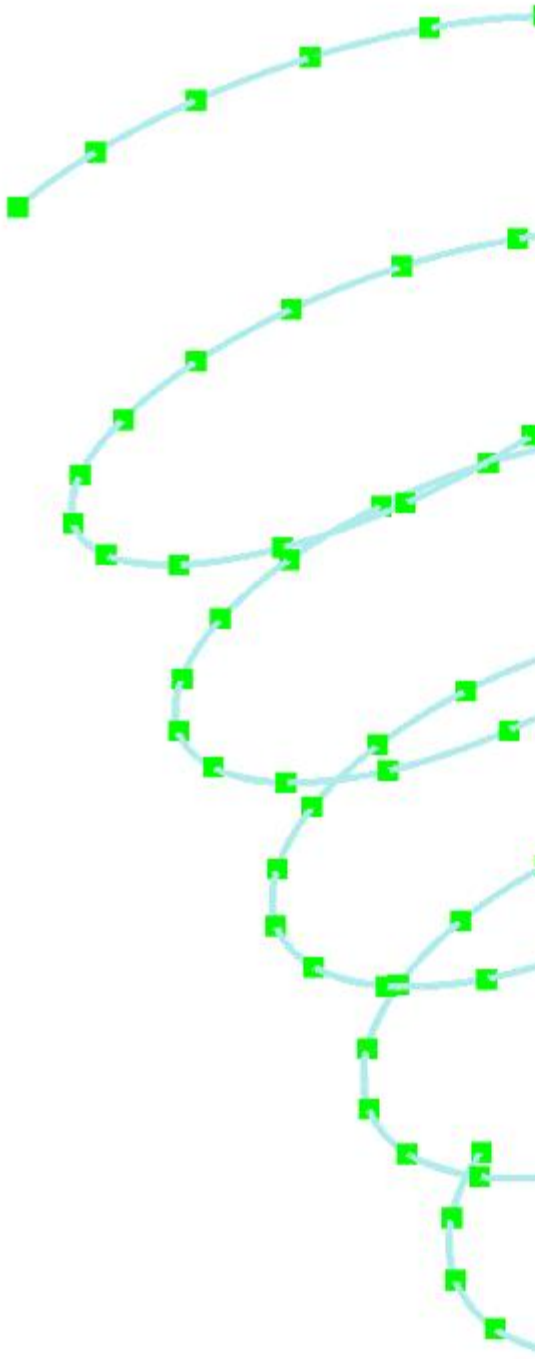
# Examples

The following figure illustrates the **spring** option for the **reduce** command.



**(a)**

**Initial CAD geometry**



**(b)**
**reduce volume 1 spring combine mesh size 0.001 block_ID Default**

**Figure 2. Example of using the reduce spring command where (a) is the original geometry of a spring. In (b), the spring option has been used which in this case replaces the spring volume with a single free curve meshed with beam elements. A new block is generated and assigned the resulting curve and beam elements and the original spring geometry is deleted leaving only the free curves in place of the volume.**

## *Reduce Thin Volumes*

The **Reduce Thin Volumes** commands simplifies a 3D thin volume into a connected set of sheet bodies. The **reduce thin** commands are frequently used with the intention of generating shell finite elements. Note that these commands will perform similar geometric operations to the **surface copy** and **midsurface** commands, but will also keep track of **thickness** and **loft**; attributes necessary for building a full representation for shell finite element analysis. The resulting sheet bodies will also automatically generate blocks with these attributes and will maintain their association to their original 3D parent geometry.

The **Thin Auto** version (illustrated in figure 1) automatically reduces a set of 3D thin volumes into a connected set of sheet bodies based on an internal geometric reasoning algorithm resulting in sheet bodies that are connected at shared (merged) curves. It will also automatically build thickness, loft and block information. The **copy** and **midsurface** versions of the command are more prescriptive, and offer more control so that specific required reductions can be performed. The **copy** option can be used to prescribe a few required reductions in an assembly prior to automatically reducing the remainder with the **reduce thin auto** command.

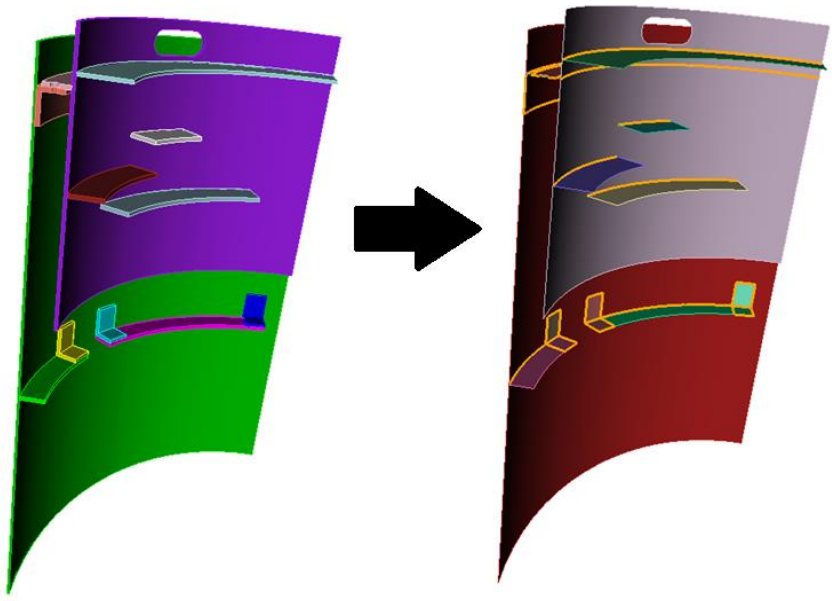


**Figure 1. Reducing a set of 3D thin volumes. Merged curves are highlighed.**

**Syntax:**

**Reduce Volume <volume ids> Thin Auto [Sort] [Preview]**

**Reduce Volume <volume ids> Thin Copy <surface ids> loft factor <values> thickness <values>...[Combine] [Delete] [Preview]**

**Reduce Volume <volume id> Thin Midsurface surface <id1, id2> loft factor <value> thickness <value> [Combine] [Delete] [Preview]**

**Thin Auto**

To use the **Thin Auto** command, the 3D thin volumes must be touching so that if the user imprinted and merged them, they would be connected at merged surfaces. This routine traverses the set of 3D volumes, finding the best 2D reductions that preserve connections between the 3D volumes. This routine also creates shell element blocks containing the resulting 2D shell surfaces. Each block also is attributed with appropriate thickness and loft factor values corresponding to the parent 3D volumes. These **thickness** and **loft** factor values are the first and second block attributes respectively. Finally the command creates an internal association between the original 3D parent volume and the 2D reduction so that the user is able to visually inspect and validate the reduction. See the **draw shell volume** command below for more details.

**volume ids:** Specify the ids of the 3D volumes to be reduced. These volumes should not already be imprinted and merged. However, users should be certain that the volumes do indeed imprint and merge successfully since the routine does this internally to ultimately connect the resulting 2D shell surfaces. If a user wants to prescribe a specific reduction solution, it can be done manually using the **Reduce Thin Copy** command. To include the manually reduced volumes in the **auto** operation, the parent 3D volumes (not the generated 2D reductions) should be specified along with the other 3D volumes being reduced.

**sort:** If the **sort** option is specified, the surfaces of all 2D reductions containing the same base name will be added to the same block. This is done so that copies of the same volume remain together. These volumes should be identical in nature, with the same thickness. For example, if the user has a repeated part in the model with the base name 'bracket', all 2D reductions with that base name, like 'bracket@A', 'bracket@B', 'bracket@C',... will have all their surfaces added to the same block. If the **sort** option is not used, the surfaces of each 2D reduction will be in their own block. In either case, the next available block id is used.

It is also important to note that only 3D thin volumes that have two logical sides can be automatically reduced. For example, a piece of sheet metal, bent or pressed into most shapes has two logical sides. 3D thin volumes with t-junctions do not have two logical sides. See figure 2 below for some examples.

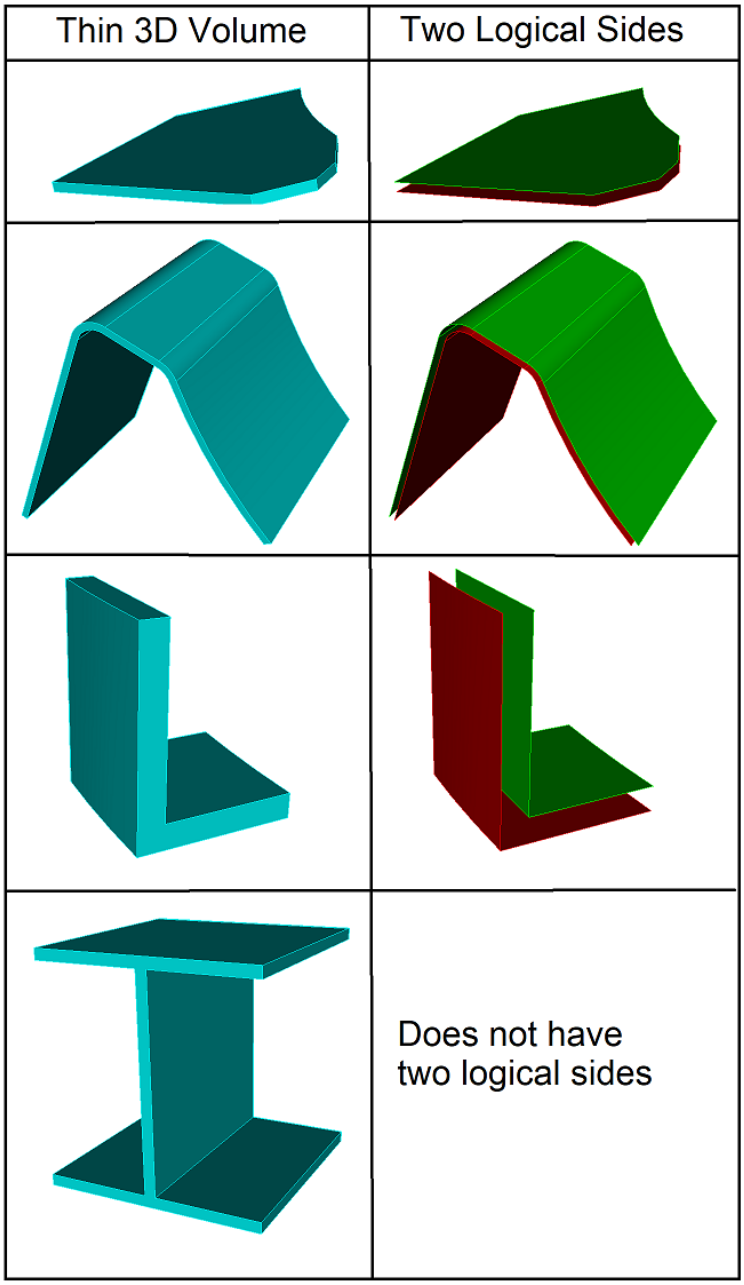



**Figure 2. Logical sides of thin volumes.**

# Thin Copy and Midsurface

Using this form of the command, the user defines the reduction by specifying the surfaces the thin 3D volume will be reduced to. Unlike the **auto** form, nothing is done to make adjacent reductions contiguous. Each surface must be specified with a corresponding **loft** factor and **thickness** value. The **combine** option reduces a volume into a multi-surface

sheet body. Otherwise, each surface will result in a single-surface sheet body. **delete** option deletes the original 3D thin volume(s).
Note that sheet bodies created using the \textbf{reduce thin midsurface} option, are currently not supported in the **auto** option. If the intention is to initially prescribe a few reductions prior to using the **auto** command, avoid using the **midsurface** command.

# Draw Shell Volume

To visually inspect that the 2D reductions correctly approximate the original 3D thin volumes, the command below can be used. The 3D volume is drawn in wireframe mode, 2D reduction in shaded mode, and the element block of the 2D reduction in shaded or optionally transparent mode.

# Draw Shell Volume <ids> [Color<color_spec>] [Transparent] [Add]

# Exporting Shell Data

To export information to a csv file detailing the reductions, the command below can be used:

# Export Shell Data <string> [Volume<ids>] [Brief] [Overwrite]

By default, the following columns are written for each 3D volume specified. If no volumes are specified, the data for all 3D thin volumes reduced are exported. For each 3D thin volume, the following columns of data are exported:

| | | | | | | |
|---|---|---|---|---|---|---|
| sheet body ID | sheet body name | parent volume ID | parent volume name | thickness | loft | block |

If the **brief** option is used, only the following columns of data are exported:

| | | | |
|---|---|---|---|
| block | sheet base name | thickness | loft |

# *Reduce Thin Volumes with Reinforcement Learning*

The **Reduce Thin Volumes RL** command, similar to the **Reduce Thin Volumes *Auto*** simplifies a 3D thin volume into a connected set of sheet bodies. The **reduce thin volume** commands are frequently used with the intention of generating shell finite elements. In contrast to the **auto** option, the **RL** option uses machine learning methods to build a persistent knowledge base to improve its choice of reduce operations. If an assembly has not yet been *learned*, the method may initially predict a relatively poor solution. As the RL method is run, it builds training data, effectively learning the most suitable reduce solutions for a given configuration of thin volumes. In general, the more diverse problems encountered, the better the outcomes of the RL predictions.

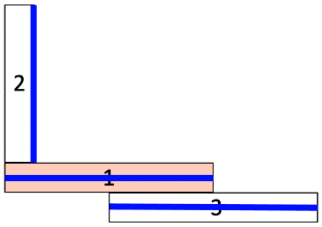



**Figure 1.**

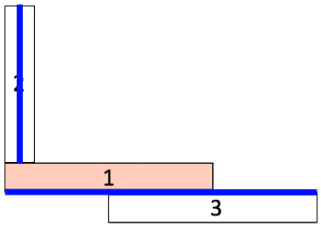



**Figure 2.**

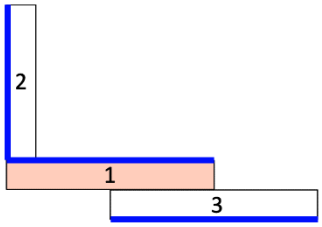



**Figure 3.**

When generating a shell model of a 3D set of thin volumes, the user often must navigate many tools to reduce the volumes and then establish connections. The sequence of commands to build such a model, for anything other than a trivial case, can become unwieldy. Figures 1-3 show one such example where multiple different solutions can be generated for this simple idealized model of the thin volumes. It should be apparent that some solutions may be acceptable, while others are not. The RL tool assists in developing a knowledge base of assembly states and how best to resolve them when encountered. The **RL** option can be used to generate a sequence of Cubit commands, which can be exported to a journal file. The user can then validate the result, edit and annotate the journal for archival purposes. The commands generated for reduction will be of the form **reduce thin copy** or **reduce thin midsurface**. Note that these commands will perform similar geometric operations to the **surface copy** and **midsurface** commands, but will also keep track of **thickness** and **loft**; attributes necessary for building a full representation for shell finite element analysis. The resulting sheet bodies will also automatically generate blocks with these attributes and will maintain their association to their original 3D parent geometry.

**Syntax:**

# Reduce Volume <volume ids> Thin RL [Number Iterations <value>] [Initialize Random [<value>]] [Stopping Criteria <value>] [Learning Interval <value>] [Journal Result <string>] [Delete] [Preview]

**Preparation**

Similar to the **Reduce Thin Auto**, to use the **Reduce Thin RL** command, for best results, the 3D thin volumes should be touching so that if the user imprinted and merged them, they would be connected at merged surfaces. This may involve using checking for gaps and overlaps and resolving prior to using this tool. Some cleanup and defeaturing of the

volumes may also be necessary, such as removing chamfers, rounds or other small features that may not be relevant to the final FEA model.

#### Strategies

The RL tool allows for learning and exploration of the state space so that a knowledge base (training data) can be established. It can also be used for faster prediction of reduction operations once a sufficient knowledge base has been established. In all cases the user is able to validate and experiment with the result using the journal file produced.

**Predict Mode:** This mode is normally used when sufficient training data has been established. By setting the argument **Number Iterations = 1**, the RL method will build the sequence of reduce commands based on predictions made from the existing training data.

**Learning Mode:** This is normally used when working with a new assembly that may be significantly different than previous models encountered. The user will typically run the RL methods for multiple iterations (**Number Iterations > 1**), allowing the method to gather information about the state space of the assembly. Depending on the complexity of the assembly, this may take anywhere from a few minutes to several hours to gather sufficient data. At each iteration, a single *average reward* value is computed and displayed at the command line that represents the completeness of the resulting solution. This value is a heuristic measurement of how well the resulting sheet bodies connect as well as other factors including potential introduction of small curves and narrow surfaces.

To operate in *learning mode*, RL will use various measures to determine when to terminate learning:

1. **Stopping Criteria:** A value between zero and one. If the average reward meets or exceed this value. Default value is 0.999.
2. **Number Iterations:** Maximum number of iterations to perform. Default value is 100.
3. **No Additional Unique Solutions:** In some cases, espcially for smaller models, the RL procedure will have attempted all possible combinations of reduce operations, and may terminate prior to reaching the stopping criteria or number of iterations.
4. **User Abort:** If the process appears to be taking too long, the use may hit the red (x) in the Cubit GUI to abort the command prior to reaching stopping criteria or number of iterations. If this occurs, whatever the RL method has learned to date will be retained.

**User Influenced Learning:** The RL tool provides an ideal method for efficiently establishing a lot of training data in a short amount of time. The *learning*, however is built on internal predefined heuristics that generally apply to most assembly states. The user can over-ride these preferences by adding their own training data. The most convenient way to do that is through the Cubit Geometry Power Tool, using the **Thin Volumes** diagnostic. This diagnostic will display the current predicted confidence value for a given **reduce** command in the solution window. This can be influenced by the user by using the **maximize** or **minimize confidence** right-click options. These options are also available from the Cubit command line using the **Learn** series of commands.

#### Arguments

**Number Iterations <value>:** Maximum number of RL iterations to perform before termination. Default is 100.
**Initialize Random {<value>}:** Used to initialize the state to random **reduce** solutions. Normally the state is initialized by predicting the best solution based on the current training data. Random initialization can be used for maximum exploration of the state space, and is most useful for models that have not been seen before. It can also be used to experiment with known solutions to see if alternative solutions can be derived. The options **<value>** is a *seed* that can be used to guide the internal random number generator.
For maximum exploration of the state space in learning mode, the **Initialize Random** may also used. Normally, the starting point for the RL method will be a prediction based on the current knowledge base (training data). The Initialize Random options, ignores any current
**Stopping Criteria <value>:** A value between zero and one. The RL method terminates if the average reward meets or exceed this value. Default value is 0.999.
**Learning Interval <value>:** Integer value that indicates how often to update the machine learning model. For example, a learning interval of 5 will update the training model with information it has learned at every fifth RL iteration. Following the update, the state rewards will be reinitialized based on predictions from the ML model to begin the next iteration. Default for **learning interval** is 100. If the default is used, normally the ML model is only updated after the final iteration.
**Journal Result "<string>":** If a filename is provided using this option, a journal file with that name will be written to the current working directory with the resulting sequence of commands from the *best* RL iteration. The resulting journal file will be annotated with comments indicating reward values and success or failure of connections.
**Delete:** Parent volumes will be deleted after the sheet bodies have been generated. This is generally not recommended as the association between 3D solid thin volumes and their resulting child sheet body(s) is maintained behind the scenes in Cubit. Visualization and management of sheet data, such as block attributes is maintained through this association, as well as the **Export Shell Data** command.
**Preview:** If the preview option is used, the final best solution will not be executed, but rather a graphical preview of the result will be displayed. The preview will also include visual cues for connections that have been maintained and those that do not. In addition, if the **journal result** is used, the journal file will also be produced. This option is often used to preview and validate the result, prior to running the resulting journal once the solution is established as acceptable.

# *Reduce Slot Surface*

The **Reduce Slot Surface** is used to decompose slot surfaces in Electromagnetic (EM) modeling for easier application of boundary conditions.

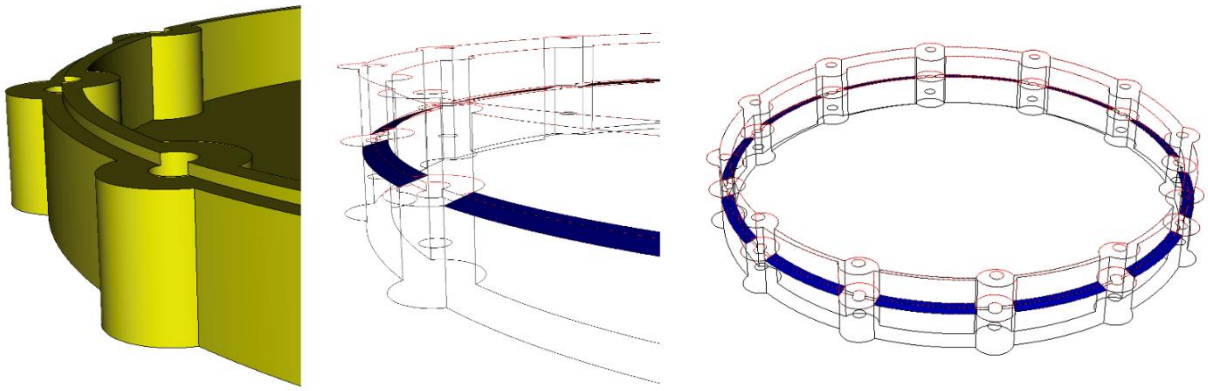


**Figure 1.Example preview of slot surfaces, shown in blue (right) constructed from the original geometry (left).**

**Syntax:**

**Reduce Surface <ids> slot [radius {<value>|Factor <value>|SHIGLEY [angle {<value>}]}] [inner {curve <ids>}] [outer {curve <ids>}] [hardware volume <ids>] [[create] skin] [group <string>] [group_edges <string>] [group_inner_edges <string>] [group_outer_edges <string>] [group_inner_surfaces <string>] [group_outer_surfaces <string>] [name <string>] [name_edges <string>] [name_inner_edges <string>]**

# [name_outer_edges <string>] [name_inner_surfaces <string>] [name_outer_surfaces <string>] [make_free_curves] [preview]

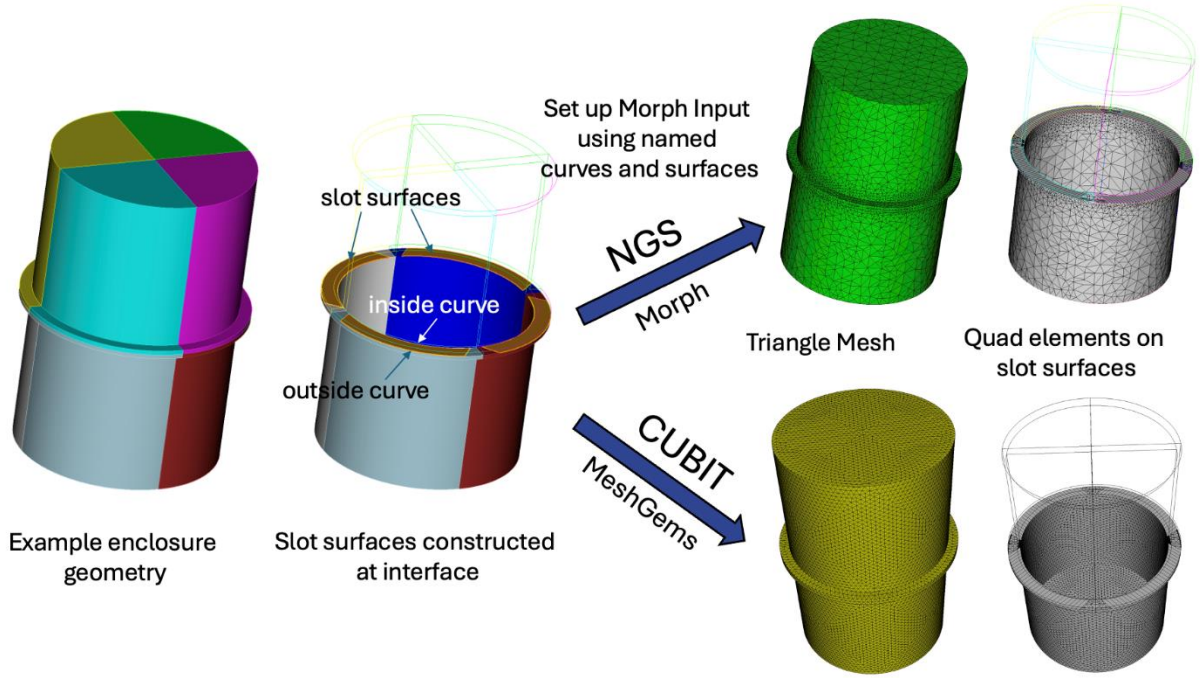
# Discussion:

The **Reduce Surface Slot** command is engineered for the preparation of models for electromagnetic (EM) simulations, facilitating the creation of slots—pathways allowing EM radiation to traverse. This command is versatile, designed for compatibility with both Cubit quad and tri meshers and the Morph mesher. It plays a vital role in generating surface meshes essential for EM simulations.

# Prerequisites and Best Practices:

Before invoking the **Reduce Surface Slot** command, adhere to these best practices to ensure optimal model preparation:

1. **Defeaturing and Simplification:** Simplify the model by defeaturing or removing unnecessary details that may complicate splitting procedure in the **reduce slot** command.
2. **Enclosure Integrity:** Verify that the enclosure region is fully closed. Use the **remove surface** commands as necessary to seal any gaps or holes.
3. **Imprint and Merge:** For mismatched mating surfaces above and below the slot pathways, perform an **imprint** and **merge** operation to ensure that the upper and lower slot surfaces align perfectly.



# Figure 2.Slot Preparation Workflow: Identification and division of slot surfaces via the reduce surface slot command. The images display the distinct meshing results with Cubit or Morph.

# Common Steps for Cubit and Morph Workflows:

1. **Adjust Options:** Customize the command settings according to the provided command syntax description.
2. **Invoke the Command:** Execute the Reduce Surface Slot command to begin the process of defining and splitting the slot surfaces. It's also worth noting that achieving the desired outcome may require some trial and error, with a **preview** option available to visualize the slot pathways before finalizing.

# Cubit Workflow:

For Cubit, the focus is on direct mesh generation:

1. **Mesh Preparation:** Unlike Morph, Cubit does not require specific naming of surfaces and curves for mesh generation.
2. **Quad Meshes on Slot Surfaces:** Explicitly generate quad meshes on the slot surfaces using Cubit commands for assigning intervals, setting schemes and meshing.
3. **Tri Meshes for Remaining Surfaces:** Apply tri meshers for the rest of the model, setting mesh size, scheme, and then meshing directly in Cubit.

# Morph Workflow:

For Morph, the emphasis is on preparing data for meshing in an external process:

1. **Surface and Curve Naming:** Important for Morph, where naming conventions are essential for the mesher to recognize and process the model correctly. Names can be explicitly identified in the command or the default naming convention may be used.
2. **Export Patch Data:** Utilize the **export slot data** command to provide Morph with the necessary information for quad and tri mesh generation.

# Command Syntax Description:

**surface ids**: Specifies the slot surfaces designated for decomposition. Multiple surfaces can be included in the selection, provided they form a coherent, continuous loop. These surfaces are allowed to incorporate disruptions like holes, which accommodate hardware

or fasteners, ensuring component cohesion. Importantly, these surfaces must delineate the region intended for the electromagnetic radiation pathway.
**hardware volume ids**: Identifies the volumes of fasteners that intersect with the slot surface. The centroid of each fastener, as it appears on the slot surface, guides the determination of cut locations. This parameter is optional; in its absence, the command automatically seeks out holes, inferring potential fastener centroids based on the centers of these holes.
**radius**: Specifies the distance from the centers of fasteners on the slot surface to where cuts will be made. The radius can be defined using one of three methods:

1. **Explicit Definition:** Directly sets the radius to a specific value.
2. **Factor:** Determines the radius as a multiplier of the bolt or hole dimensions, applicable when bolts or holes are present.
3. **Shigley Method:** Calculates the radius based on a specified angle, the bolt head radius, and the thickness of the clamped member. This method is applicable only in the presence of hardware fasteners.

The process aims to prevent the creation of sliver geometries. However, users should exercise caution to prevent cuts that produce overly small curves or surfaces. A tolerance of 10% of the specified radius is applied, allowing cuts to be adjusted to the nearest vertex to mitigate the risk of slivers or tangent conditions.
**group**: Used to create and name a group containing all resulting decomposed surfaces. The options **group_edges**, **group_inner_edges**, **group_outer_edges**, **group_inner_surfaces** and **group_outer_surfaces** are used to create and name groups of cutting curves, inner curves, and outer curves, respectively.
**name**: Similar to the group option, but individual names are assigned to the resulting decomposed surfaces and curves. The names of decomposed surfaces, side curves, inner curves, outer curves, inner surfaces and outer surfaces can be defined using the respective options. If not specified, a default naming convention will be used. Note that these names will be used in the Morph input deck and can be written using the **export slot data** command
**make_free_curves**: A specialized option to generate free curves at the boundaries of the decomposed slot surfaces. Grouping and naming options also apply to the free curves.
**create skin:** This function constructs a sheet body through a lofting process that spans from inner to outer curves. It is applicable to any slot surface, offering significant utility when dealing with multiple surfaces that do not form coplanar paths. This feature is particularly valuable for defining surfaces for quad meshing, effectively outlining the electromagnetic (EM) pathway. An alternative to having this command create a skin surface is for the user to create a skin surface and tweaking nearby surfaces to match the skin surface. If done this way, the tweak operation will preserve any bolt holes, and a **reduce surface slot** operation can be performed afterwards.
**preview**: Displays a blue preview of the slot paths without performing the actual cutting operations as well as a wire frame outline of the owning volume

# Export Slot Data:

The **Export Slot Data** command is used to list or export the information required for a Morph input deck. It can either provide a preview in the output window or export the data to a text file if a filename is specified. This command assumes the use of default naming conventions. The syntax is as follows:

# Export Slot Data [surface ] [full] [free_curves] [overwrite]
# Export Slot Data [surface ] [full] [free_curves] preview

There are two different forms of the command. The first form exports the data to a file, while the second form provides a preview in the output window.

**Optional arguments:**

- **surface :** This optional argument allows you to specify the surfaces for which you want to export Morph slot data. If not specified, data for all surfaces with the expected naming convention will be written or previewed.
- **full:** In addition to the slot data, this option writes out the full Morph input deck. Otherwise, it will only write the section of the input deck related to slots.
- **free_curves:** Use this option when the **make_free_curves** option is used on the reduce slot command. It ensures that the curves are named appropriately, taking into account the existence of free curves in the model.
- **overwrite:** When exporting to a file, this option allows automatic overwrite of any file with the same name.

# *Reduce Bolt Patch*

The **Reduce Bolt** command generates a proxy representation of a **bolt** for analysis. It replaces the bolt geometry with two concentric circular surfaces centered on the bolt axis, separating the connected volumes where sidesets are automatically applied.

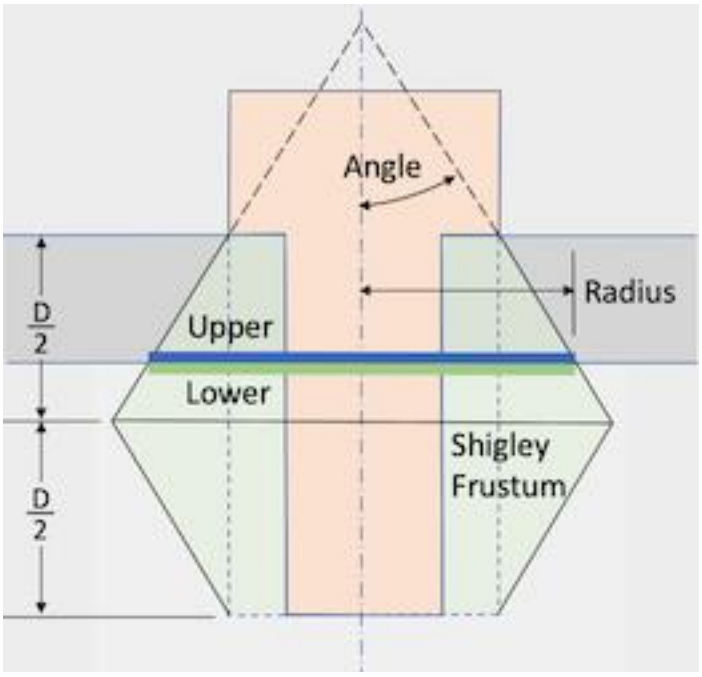



**Figure 1. Example of the Reduce Bolt Patch command.** **The Shigley frustum is used to calculate the diameter of the sidesets positioned between the volumes.**

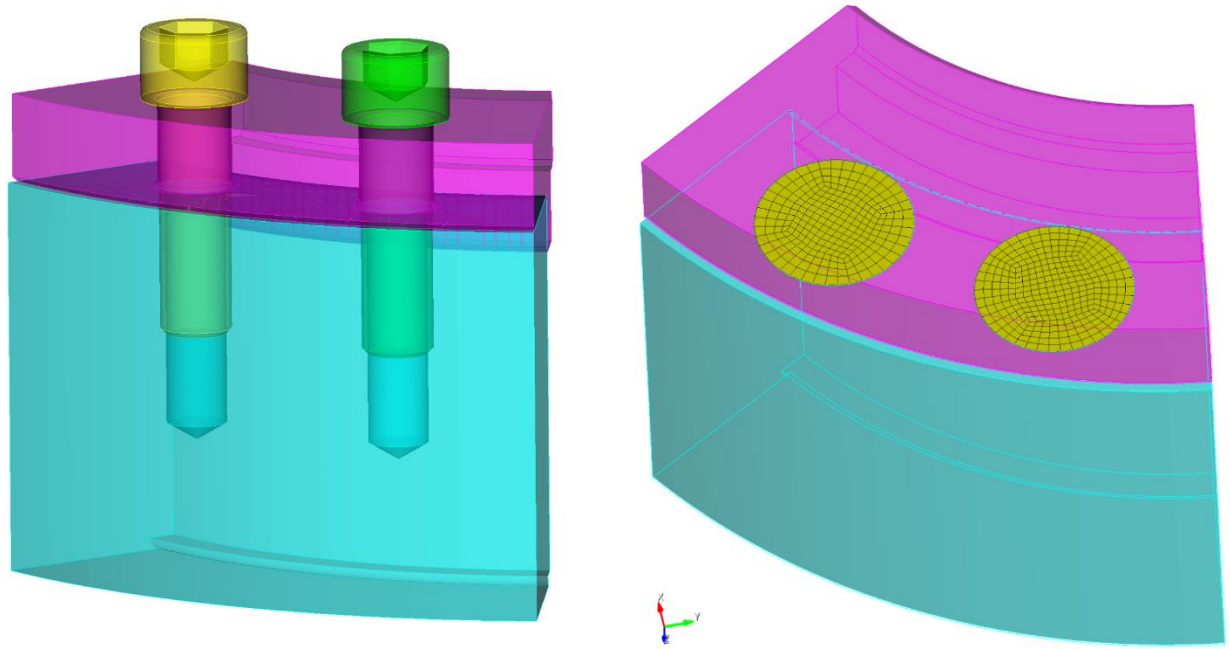
## Syntax:

**Reduce {volume <ids>|upper surface <ids> lower surface <ids>} {bolt|hole} patch [contact upper {surface <ids>}] [contact lower {surface <ids>}] [radius {<value>|Factor <value>|SHIGLEY [angle {<value>}]}] [mesh] [mesh_size <value>] [mesh_scheme <string>] [no_simplify_hole] [{FILL|no_fill}] [{DELETE|no_delete}] [array_name**

**<string>] [upper_patch_sideset_id {<value>|Default}] [increment_upper_patch_sideset_id] [start_upper_sideset_id {<value>|Default}] [upper_patch_sideset_name {<string>|Default}] [lower_patch_sideset_id {<value>|Default}] [increment_lower_patch_sideset_id] [start_lower_sideset_id {<value>|Default}] [lower_patch_sideset_name {<string>|Default}] [name_patch_surfaces] [preview]**

# Discussion:

The **Reduce Bolt Patch** command is often used in structural dynamic applications to replace bolts or bolt holes with concentric circular sidesets attached to the upper and lower volumes. If the specified radius exceeds surface bounds, clipping occurs. The resulting circular surfaces are automatically assigned to customizable sidesets. The patch radii are determined using the default Shigley frustum angle, as illustrated in Figure 1. Users can modify the Shigley angle or set the diameter explicitly as an absolute value or a factor of the shank radius. The command also supports automatic meshing of the sidesets with a user-defined scheme and size.



# Figure 2. Example of before and after where the reduce bolt patch command has been applied to two bolts. This example uses the mesh and default fill and delete options.

The following outlines the options for the **reduce bolt patch** command.
You can input either the ID of the bolt to be reduced or the IDs of holes intended for fasteners if no bolt volume is present. Choose between the **bolt** or **hole** options based on your specific needs.

**bolt**

**volume <ids>**: Specify the ids of the volumes to be reduced. The Geometry Power Tool classification diagnosic can be used for identifying volumes as "bolts".

**hole**

**upper surface <ids> lower surface <ids>:** Specify hole IDs for fastener reduction, including at least one surface from both upper and lower holes. Cubit checks axis alignment and matches multiple holes if specified. For more than two fastened volumes, Cubit supports up to three; designate the topmost with **>upper surface** and the rest with **lower surface**.

**Optional Arguments**

**contact upper {surface <ids>} contact lower {surface <ids>}:** In rare cases, the **reduce patch** command may not auto-identify upper and lower contact surfaces. If this

occurs, you can manually specify these surfaces to resolve the issue. Note: This option is only available for a single bolt or hole set.
**radius {<value>|Factor <ids>|SHIGLEY [angle {<ids>}]}**
**radius <value>:** The **Reduce Bolt Patch** allows you to directly set the radius by specifying a floating-point value. Use this option to set an absolute value for the sideset radius on the contact surfaces. This value will determine the radius of the resulting imprinted circle where the sideset is defined.
**radius factor <value>:** The **radius factor** option allows you to define the radius of the resulting imprinted circle as a multiple of the shank bolt's radius. If no bolt is present and only the hole is specified, the radius will be a multiple of the upper hole's radius.
**radius shigley angle <value>:** The **radius shigley angle** option is the default method for determining the radius. It employs a Shigley angle, as outlined in SAND2008-0371 (Figure 1). If no radius specification is provided, the Shigley angle defaults to 30 degrees to compute the radius. This radius is determined by the intersection of the frustum with the surface between the upper and lower volumes.
The circular patch size can vary based on several factors, such as the thickness of the upper volume, the radius of the bolt head, and the length of the bolt. For more comprehensive details, refer to the SANDIA REPORT SAND2008-0371, "Guideline for Bolted Joint Design and Analysis: Version 1.0."
**Note:** The **radius shigley angle** option is applicable only when a bolt is specified. If no bolt is present and only the holes are specified, and no radius is given, the **radius factor** option becomes the default. In such cases, a default factor of 2.5 will be used to compute the radius.
The **Reduce Bolt Patch** command imprints concentric circular patches onto the surfaces of the upper and lower volumes. If the surfaces on the upper and lower volume do not extend beyond the specified radius, the resulting surfaces will be clipped accordingly.
**mesh:** Option to specify whether to include meshing as part of the command.
**mesh_size <value>:** Optionally specify a target mesh size when using the **mesh**
**mesh_scheme <string>:** Specify a target meshing scheme. The following meshing schemes can be used for this operation:

- **circle** - Uses Cubit's circle scheme. If the **no_fill** option is used, the **hole** scheme will automatically be used. Meshes with quads.
- **pave** - Uses Cubit's paving scheme to mesh with quads at the specified size.
- **tri** - Uses triangle meshing at the specified size.

**FILL|no_fill:** The **FILL|no_fill** option controls hole treatment. By default setting, **FILL** removes the hole entirely, along with any features like fillets. The **no_fill** option retains the hole but simplifies it, removing fillets and creating an annulus centered on the hole's axis. The annulus diameter is set by one of the **radius** options.
**DELETE|no_delete:** The default behavior of this command is to delete the bolt as part of the operation. However, users have the flexibility to choose whether or not to delete the bolt.
**no_simplify_hole**: The **reduce surface patch** command, coupled with the **no_fill** option, is designed to streamline the meshing process by automatically eliminating chamfers, blends, and conical elements from hole geometries. However, the

**no_simplify_hole** switch overrides this behavior, preserving all existing features of the hole's geometry. This can be particularly useful in scenarios where the default simplification process might lead to command failure due to complex hole configurations. Activating this option allows for the continuation of the operation without the standard simplification step.
**array_name:** This option requires a string value that will be employed as a prefix for all resulting sidesets which are created by the command. When the **name_patch_surfaces** option is simultaneously activated, this prefix is also applied to the names of any generated surfaces. Therefore, the prefix ensures consistent naming conventions across sidesets and, conditionally, surfaces, streamlining subsequent identification and processing. Note that if the **array_name** is not explicityly specified, the name prefix "Bolt_Patch" will be prepended to the name of each of the sideset and surface names.
**Sideset IDs and name assignment**: The following options may be used to assign the resulting circular surfaces based on an ID or a sideset name:

- **upper_patch_sideset_id {<value>|Default} [increment_upper_patch_sideset_id][start_upper_sideset_id {<value>|Default}][upper_patch_sideset_name {<string>|Default}]**
- **lower_patch_sideset_id {<value>|Default} [increment_lower_patch_sideset_id][start_lower_sideset_id {<value>|Default}][lower_patch_sideset_name {<string>|Default}]**

Specify sidesets using either **sideset_name** and/or **sideset_id**. If not existing, a new one will be created. Using **Default** selects the next available ID(s) or names them as "Upper_Patch" and "Lower_Patch" if unspecified.
Optional **increment** and **start_id** arguments are available for both upper and lower sideset ID specifications. When used, these options automatically generate incrementing sideset IDs. For example, if **lower_patch_sideset_id** is set to **100** and **increment_lower_patch_sideset_id** is used, new bolt patches will have unique sideset IDs starting from 100 (e.g., 101, 102, 103). If both **upper_patch_sideset_id** and start_upper_sideset_id (or their lower counterparts) are used with **increment**, the new patches may be added to the existing sideset and a new, incrementing sideset, effectively adding the same patch to two different sidesets.
**name_patch_surfaces:** This option allows you to name the surfaces created by the **reduce patch** command. You can use either default or custom names. Default names are "Upper_Patch" and "Lower_Patch". If you specify names using **upper_patch_sideset_name** or **lower_patch_sideset_name**, those names will instead be applied to the resulting circular surfaces. This is helpful for referencing in future scripts or journal commands.

#### Sideset Naming Convention

**Default Naming:** Sidesets are automatically named to reflect the connections between components. For instance, for clamped volumes with names "PartA" and "PartB" generates sideset names such as "PartA_to_PartB_123" and "PartB_to_PartA_123." These names encapsulate the parts involved and a unique identifier for the fastener's volume, "_123" in this case. In this example, the name "PartA_to_PartB_123" signifies

the sideset on PartA that denotes its interface with PartB, and similarly, "PartB_to_PartA_123" is assigned to PartB to mark its connection with PartA. When blocks are designated and the clamped volumes are assigned, the naming convention defaults to using block names rather than volume names.
Default Naming Format:
**array_name** + "_" + [NameA] + "_" + [NameB] + "_" + [Volume ID of Bolt]
where **NameA** designates the name of the volume or block the sideset is attached to and **NameB** designates the volume or block the sideset is in contact with.
Naming Examples:
If **array_name** is set to "Connect", the volume ID of the bolt is 3, and the clamped volume names are "PartA" and "PartB" the resulting sideset names for the resulting sidesets would be:

- Connect_PartA_to_PartB_3
- Connect_PartB_to_PartA_3

**Alternative Default Naming:** Activating the option with **sideset_naming_convention** set to **1** changes the naming scheme. Sidesets begin with a "Bolt_Patch" prefix (assuming no **array_name** is given), then add "_Lower" or "_Upper" to indicate their position relative to the bolt's bearing surface, and conclude with a numeric identifier for the bolt volume. This convention offers an alternative to the default, which is applied automatically without needing explicit selection.
Alternative Naming Format:
**array_name** + "_" + [Volume ID of Bolt] + "_Upper/Lower"
Naming Examples:
If **array_name** is set to "Connect" and the volume ID of the bolt is 3, the resulting sideset names for the upper and lower portions would be:

- Connect_3_Upper
- Connect_3_Lower

This naming applies to surfaces as well when the **name_patch_surfaces** option is activated. In cases where a single bolt connects more than two volumes, the naming includes the individual connections. For example, a bolt fastening three volumes would yield names such as:

- Connect_1_3_Upper
- Connect_1_3_Lower
- Connect_2_3_Upper
- Connect_2_3_Lower

Here, the first number represents the index of the connection at the same bolt, and the second number is the volume ID of the bolt.
Overriding the Default:
Be aware that these default naming conventions can be superseded by the explicit naming options, **upper_patch_sideset_name** and **upper_patch_sideset_name**, provided in the previous section of this documentation.

#### Export Patch Data Command

This section outlines the use of the **export patch data** command, which allows users to save or preview information derived from the **reduce bolt patch** process. This command facilitates the generation of data necessary for Morph input decks or simulation purposes, including the creation of a .csv file. Below is the detailed syntax for utilizing this command.

**Export Patch Data <filename> [{csv|morph|BOTH}] [surface <ids>] [lower name <name>] [upper name <name>] [sideset_naming_convention <value>] [full] [overwrite]**

**Export Patch Data [surface <ids>] [lower name <name>] [upper name <name>] [sideset_naming_convention <value>] [full] preview**

The command is available in two variations: one for writing the output to a file and another for previewing the morph patch data in the output window. Both variations require the prior use of the **name_patch_surfaces** option within the **reduce bolt patch** command to identify and process the surfaces correctly.
Command Options and Arguments

- **csv|morph|BOTH**: Specifies the output format. By default, both CSV and Morph input data are written to a file. If both are selected, the base filename provided is used for the Morph input file, with a .csv extension appended for the CSV file. Selecting **csv** generates only a CSV file, including a header with named patch surfaces listed as pairs, the distance between these pairs, and the x-y-z coordinates of the contact pairs' center. Choosing **morph** outputs only the Morph input data, containing essential information for the Morph tet meshing tool.
- **surface <ids>**: Allows specification of one or more surfaces for output. If omitted, the command writes data for any surfaces meeting the sideset patch data criteria.
- **sideset_naming_convention <value>**: Defines the naming convention used in the reduce bolt patch command. It is crucial to use the same convention as previously defined. A value of zero or not using this option defaults to the convention outlined above. A value of **1** indicates an alternative naming convention.

- **[lower name <name>] [upper name <name>]**: Applicable when **sideset_naming_convention=1** is used. These options assume a naming convention for upper and lower patches based on the specified strings.
- **full**: When used, additional data necessary for a comprehensive definition of the Morph input deck is included. Without this option, only basic patch data is written.

## Debugging Geometry

The following command checks for inconsistencies in the CUBIT topological model, by checking the specified entities and all child topology and/or comparing to solid model topology:

**Geomdebug Validate [compare] <entity_list>**

This command checks for:

- Consistent CoFace senses
- Loops are closed/complete
- Consistent CoEdge senses
- Correct vertex order on curves w.r.t. parameterization
- Correct tangent direction of curves w.r.t. parameterization

**Related Commands:**

**Geomdebug Vertex <vertex_id>**

**Geomdebug Curve <curve_id>**

**Geomdebug Surface <surface_id>**

**Geomdebug body <body_id>**

**Geomdebug Containment {Curve | Surface} <id> {Location (options) | Node <id_list>}**

The following command prints info about GeometryEntities owned by specified entity:

**Geomdebug Geometry <entity_list> [interval <n>] [index <n>] [TEXT] [GRAPHIC] [attributes]**

The following command lists (TopologyBridge) topology for specified entity:

**Geomdebug solidmodel <entity_list> [index <n>] [depth<n>|up<n>|down<n>]**

The following command lists GroupingEntities.

**Geomdebug GPE <entity_list>**

## Finding Surface Overlap

The surface overlap capability finds surfaces that *overlap* each other, with the capability to specify a distance and angle range between them. This is useful for debugging geometry imprinting and merging problems, as well as for finding gaps in large assembly models. Finding overlapping geometry is done using the command:

**Find [Surface] Overlap [{Body|Surface|Volume} <id_list> [Filter_Sliver]**

If a list of entities is not specified, all bodies in the model are checked. By default the command does not check the surfaces within a given body against each other; rather, it only checks surfaces between bodies. This can be overridden by inputting a surface list (i.e. **find overlap surface all**), or with a setting (see below).
The **filter_sliver** option will remove false positives from the list by weeding out sliver surfaces that have a merged curve between them. The following pictures is an example of a sliver surface.

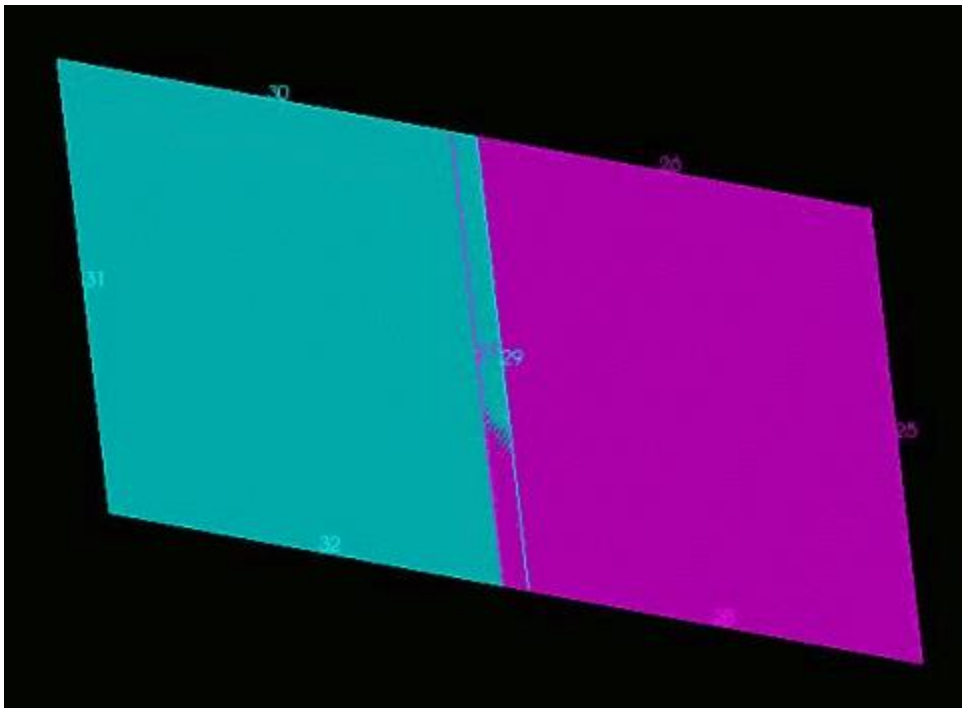



**Figure 1. Example of a sliver surface**

If curves 27 and 29 are merged before you run the find overlapping surface checkthe user will get the two surfaces in the picture as an overlapping surface pair. However, if the *filter_sliver* keyword is used, Cubit will not find the two surfaces to be overlapping.

*Facetted Representation*

This command works entirely off of the facetted surface representation of the model (the facetted representation is what you see in a shaded view in the graphics). There are inherent advantages and disadvantages with this method. The biggest advantage is avoidance of closest-point calculations with NURBS based geometry, which tends to be slow. This method also eliminates possible problems with unhealed ACIS geometry. The disadvantage is working with a less accurate (i.e., facetted) representation of the geometry. To circumvent problems with this facetted geometry, various settings can be used to control the algorithm. For example, you might consider using a more accurate facetted representation of the model - see below.

*Find Overlap Settings*

Various settings are used to control the precision and handling of overlaps during the find overlap process. A listing of the settings that find overlap uses is printed using the command:

**Find [Surface] Overlap Settings**

These settings, and the commands used to control them, are described below.
**Facet - Absolute/Angle** - The angular tolerance indicates the maximum angle between normals of adjacent surface facets. The default angular tolerance is 15 - consider using a value of 5 . This will generate a more accurate facetted representation of the geometry for overlap detection. This can be particularly useful if the overlap command is not finding surface pairs as you would expect, particularly in "curvy" regions. Note however that the algorithm will run slower with more facets. The distance tolerance means the maximum actual distance between the generated facets and the surface. This value is by default ignored by the facetter - consider specifying a reasonable value here for more accurate results.
**Set Overlap [Facet] {Angle|Absolute} <value>**
**Gap - Minimum/Maximum** - the algorithm will search for surfaces that are within a distance from the minimum to maximum specified. The default range is 0 to 0.01. Testing has shown this to be about right when searching for coincident surfaces. Gaps can be found by using a range such as 3.95 to 5.05.

**Set Overlap {Minimum|Maximum} Gap <value>**

**Angle - Minimum/Maximum** - the algorithm will search for surfaces that are within this angle range of each other. The default range is 0.0 to 5.0 degrees. Testing has shown that this range works well for most models. It is usually necessary to have a range up to 5.0 degrees even if you are looking for coincident surfaces because of the different types of faceting that can occur on curvy type surfaces. For example, for the case of a shaft in a hole, the facets of the shaft usually won't be coincident with the facets of the hole, but may be offset by a certain distance circumferentially with each other. The 5 degree max angle range will account for this. If you find that the algorithm is not finding coincident surfaces when it should, you can increase the upper range of this value. Note that this parameter is useful also for finding plates coming together at an angle.

**Set Overlap {Minimum|Maximum} Angle <value>**

**Normal** - this setting determines whether to search for surfaces whose normals point in the same direction as each other (**same**), away from each other (**opposite**) or either (**any**). The default is ANY, but it may be useful to limit this search to *opposite*, as this would be the usual case for most finds.

**Set Overlap Normal {ANY|opposite|same}**

**Tolerance** - two individual facets must overlap by more than this area for a match to be found. Consider the two cylindrical curves at the interface of the shaft and the block in Figure 2. Note that some of the facets actually overlap, even though the curves will analytically be coincident. You can filter out false matches by increasing the overlap tolerance area. The default value for this setting is 0.001.

**Set Overlap Tolerance <value>**

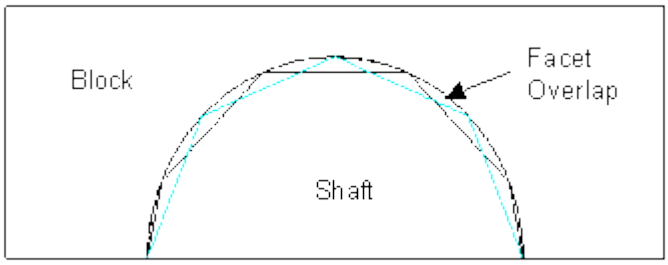



**Figure 2. Possible false find due to overlap (tolerance will prevent finding match)**

**Group** - the surface pairs found can optionally be placed into a group. The name of the group defaults to "overlap_surfaces".

**Set Overlap Group {on|OFF}**

**List** - by default the command lists out each overlapping pair - this can be turned off using the command:

**Set Overlap List {ON|off}**

**Display** - by default the command clears the graphics and displays each overlapping pair - this can be turned off using the command:

**Set Overlap Display {ON|off}**

**Body** - by default the command will not search for overlapping pairs within bodies - only between different bodies. Turn this setting on to search for pairs within bodies. Note however that this will slow the algorithm down.

**Set Overlap [Within] {Body|Volume} {on|OFF}**

**Imprint** - If on, Cubit will imprint the overlapping surfaces that it finds together. This will often force imprints that just imprinting bodies together will miss. For each pair of overlapping surfaces, the containing body of one surface is imprinted with the individual curves of the other surface, until the resulting surfaces no longer overlap.

```
Set Imprint {on|OFF}
```

## Geometry Accuracy

The accuracy setting of the ACIS solid model geometry can be controlled using the following command:

```
[set] Geometry Accuracy <value = 1e-6>
```

Some operations like imprinting can be more successful with a lower accuracy setting (i.e., 0.1 to 1e-5). However, it is not recommended to change this value. ***Be sure to set it back to 1e-6 before exporting the model or doing other operations as a higher setting can corrupt your geometry***.

## Regularizing Geometry

The regularize command removes unnecessary topology, which in effect reverses the imprint operation. This can help clean up the model from extra features that are unnecessary for the geometric definition of the model. The following command regularizes the model:

```
Regularize Body|Group|Volume|Surface|Curve|Vertex <range>[keep {curve
<ids>|vertex <ids>}]
```

The **keep** option allows the user to specify curves and vertices that should not be removed during the regulairze operation.

If you are frequently using web-cutting or other boolean operations to decompose your geometry, it may be convenient to always generate regularized geometry. To set creation of regularized geometry during boolean operations use the following command:

```
Set Boolean Regularize [ON | off]
```

## Stitching Sheet Bodies

The stitch command stitches together the specified sheet bodies into either a larger sheet body or a solid volume(s). The tolerance value can be used when these sheet bodies don't line up exactly along the edges. This is common for IGES and STEP models. Only manifold stitching is performed, i.e., edges will be shared with no more than two surfaces.

```
Stitch {Body|Volume} <id_range> [Tolerance <value>] [No_simplify]
[No_tighten_gaps] [Restricted]
```

This command has three stages to it:

1. **Stitch the surfaces together along overlapping edges** Normally IGES and some STEP files do not contain topological information that links surfaces together to share bounding curves. Stitching is an operation that builds up this topological information.
2. **Simplify geometry** The command replaces splines with analytics where possible.
3. **Tighten up gaps (inaccuracies) between the sheet bodies** The command will build the geometry necessary to tighten the gaps in the model.

When the stitch operation completes, a print statement lets the user know if the resulting body is not a closed, solid body.
The user can choose to omit the second and third options of stitching with the **no_simplify** and **no_tighten_gaps** options respectively. This may be necessary in very large or complex models, where the regular approach fails, or takes an inordinate amount of time.
The **restricted** option limits the stitching operation to the boundary curves of the specified sheet bodies or volumes. All non-boundary curves are ignored by the stitch algorithm. This functionality is intended as a performance enhancement.

## Trimming and Extending Curves

Curves can be trimmed or extended with the following command:

**Trim Curve <id> AtIntersection {Curve|Vertex <id>} Keepside Vertex <id> [near]**

Curves can be trimmed or extended where they intersect with another curve or at a vertex location. When trimming to another curve, the curves must physically intersect unless they both are straight lines in which case the **near** option is available. With the **near** option the closest intersection point is used to the other line - so it is possible to trim to a curve that lies in a different plane. When trimming to a vertex, if the vertex does not lie on the curve, it is projected to the closest location on the curve or an extension of the curve if possible.
The **Keepside** vertex is needed to determine which side of the curve to keep and which side to throw away. This vertex need not be one of the curve's vertices, nor does it need to lie on the curve. However, if it is not on the curve it will be projected to the curve and that location will determine which side of the curve to keep.
If the curve is part of a body or surface, it is simply copied first before trimming/extending. If it is a free curve a new curve is created and the old curve is removed. The figures below show several examples of trimming/extending curves.

*Trimming a Curve*

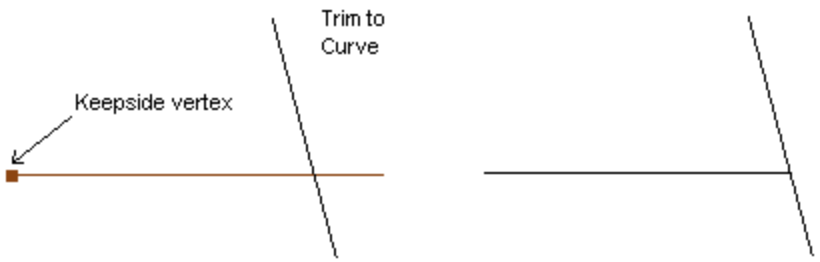



**Figure 1. Trimming a Curve to an Intersecting Curve**

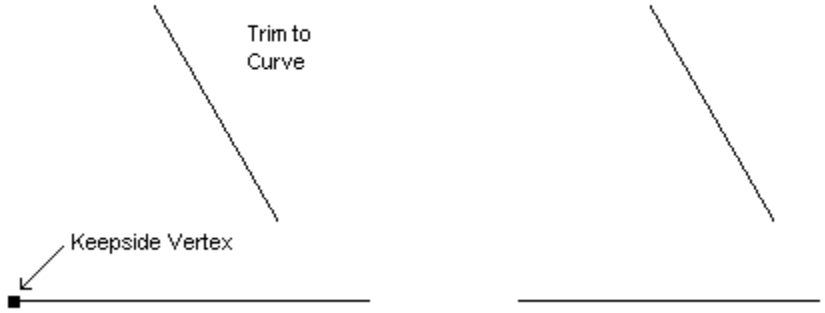



**Figure 2. Trimming a Curve to a Non-Intersecting Curve Using the Near Option**

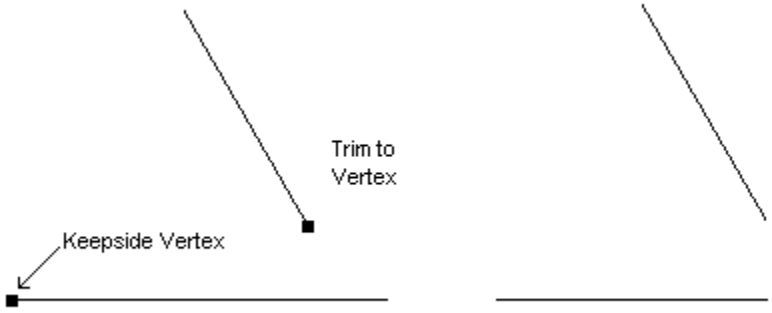



**Figure 3. Trimming a Curve to a Vertex**

*Extending a Curve*

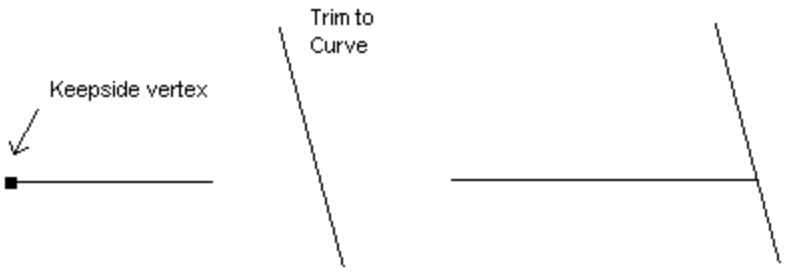



**Figure 4. Extending a Curve to An Intersecting Curve**

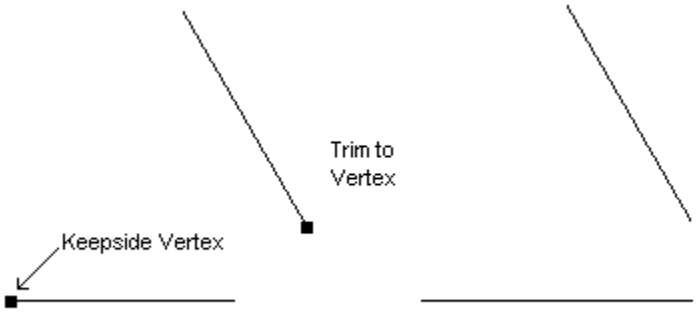



**Figure 5. Extending a Curve to a Non-Intersecting Vertex Using the Near Option**

## Validating Geometry

Detailed checks of geometry and topology can be performed using the validate command:

```
Validate {Body|Volume|Surface|Curve|Vertex|Group} <id_range>

Validate {Volume|Surface|Curve|Vertex} <range> Mesh
```

The **Validate {...} mesh** command performs a connectivity check of the mesh elements to determine the validity of the mesh.
More rigorous checking can be accomplished with the validate geometry commands by specifying a higher check level. Use the following command to accomplish this:

```
set AcisOption Integer 'check_level' <integer>
```

where **integer** is one of the following:

10 = Fast error checks
20 = Level 10 checks plus slower error checks (default)
30 = Level 20 checks plus D-Cubed curve and surface checks
40 = Level 30 checks plus fast warning checks

50 = Level 40 checks plus slower warning checks
60 = Level 50 checks plus slow edge convexity change point checks
70 = Level 60 checks plus face/face intersection checks

You can also get more detailed output from the validate command with (the default is *off*):

```
set AcisOption Integer 'check_output' on
```

Note that some of the ids listed in the output of the validate command are currently meaningless, e.g. those for coedges.
The validate command can also check for consistent surface normals and return a list of offending surfaces. The syntax for the command is as follows:

```
Validate [Body] <body_id> Normal [Reference [Surface] <surface_id>]
[Reverse]
```

Using the "reference" keyword, a reference surface is compared to the normal consistency of all other specified surfaces. Inconsistent surfaces can be reversed using the "reverse" keyword.

## Blunt Tangency

The blunt tangency commands are used to eliminate small angles in the model caused by fillets. The operation 'blunts' the tangency, or small angle, using two different approaches. The first replaces the tangency with a pair of surfaces, essentially moving the vertex to a location with a larger angle. The second approach creates a surface with a larger radius through a tweak operation, increasing the sharp angle at the vertex.

```
Blunt tangency vertex <id> [remove_material] [composite] [angle <value>]
[depth <value>] [preview]
```

```
Blunt tangency tweak vertex <ids> [angle <value>] [offset surface <id>]
[tolerance <value>] [preview]
```

As shown in Figure 1, the **depth** parameter controls the depth of the surfaces that replace the tangency, while the **angle** parameter controls the resultant angle of the new tangency. Figure 2 demonstrates the behavior of the **remove_material** option which removes material instead of adding it.

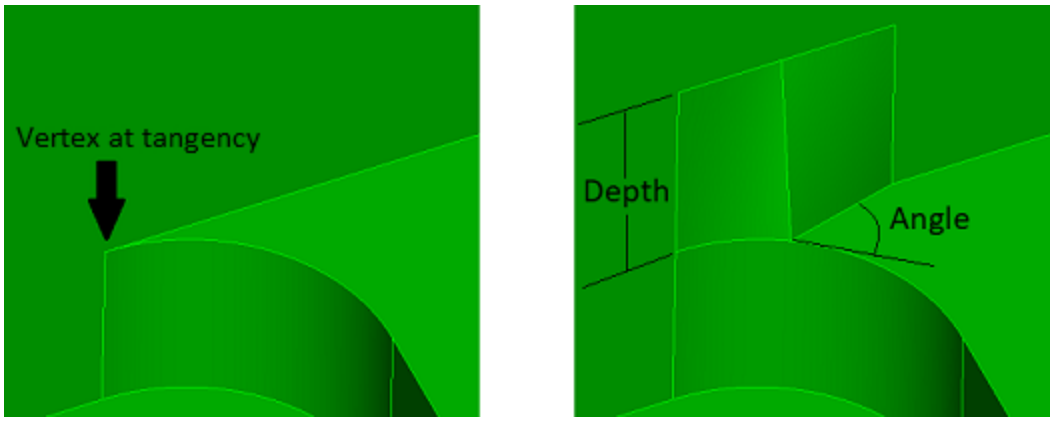



**Figure 1. Blunt Tangency Operation**



**Figure 2. Remove Material Option**

The **composite** option will composite the surfaces created in the blunt tangency operation with the adjacent larger surface, from which they were cut out. See figure 3.

**Figure 3. Blunt Tangency with Composite Option. Highlighted surface is resulting composite surface.**

The second form of the command replaces the surface with the radius at the vertex with a larger radius surface, increasing the angle. The command tweaks the surface that results in the least amount of material removed (Figure 4) or added (Figure 5). The **angle** option works the same as in the first form of the command. The **offset surface** option specifies which surface is tweaked (offset), as sometimes the surface that results in the smallest material added/removed is not correct. The **tolerance** option prevents small curves from being created, snapping to adjacent vertices within the tolerance (Figure 6) or possibly modifying an adjacent surface (Figure 7).




**Figure 4. Blunt Tangency Tweak -- Material Removed**

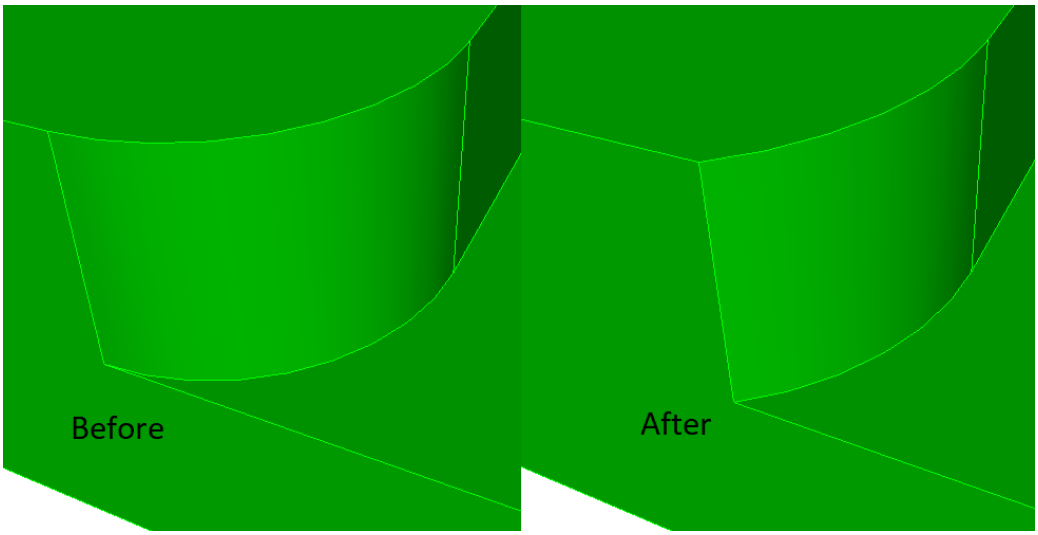



**Figure 5. Blunt Tangency Tweak -- Material Added**

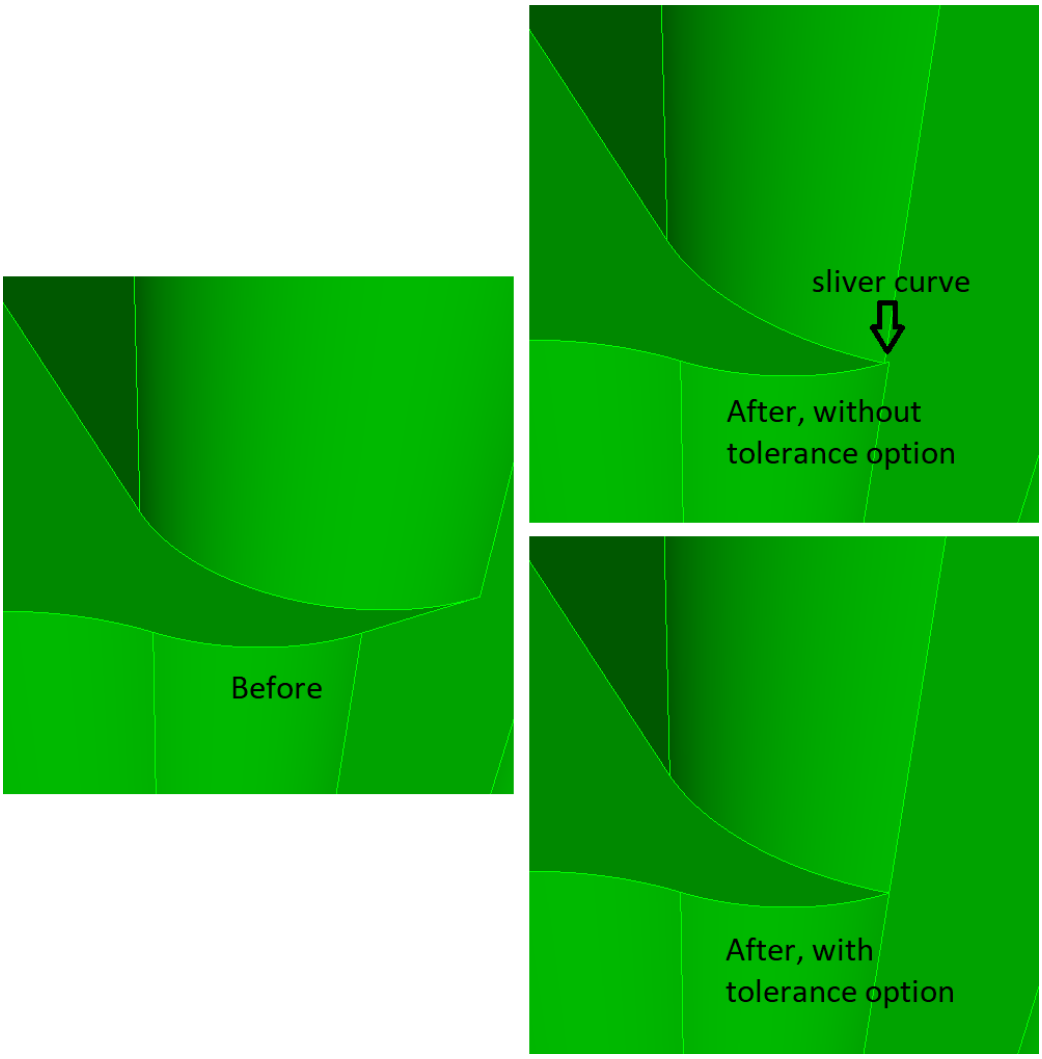



**Figure 6. Blunt Tangency Tweak -- Tolerance**

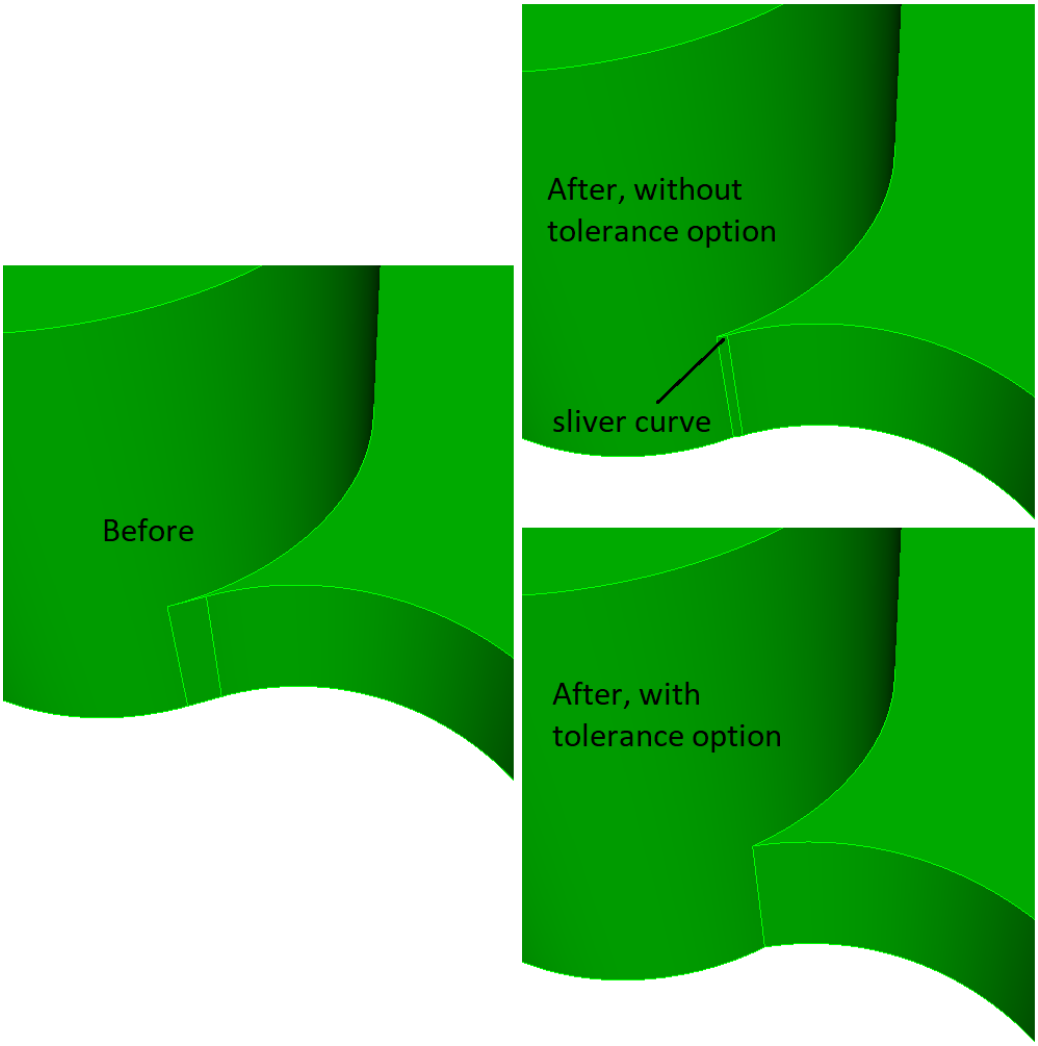



**Figure 7. Blunt Tangency Tweak -- Tolerance**

# imprint merge

Geometry Imprinting and Merging

- Imprinting Geometry
- Merging Geometry
- Examining Merged Entities
- Merge Tolerance
- Unmerging
- Using Geometry Merging to Verify Geometry

Geometry is created and imported in a manifold state. The process of converting manifold to non-manifold geometry is referred to as "geometry merging", since it involves merging multiple geometric entities into single ones. When importing mesh-based geometry, the

merging step can be automatic. Imprinting is a necessary step in the merging process, which ensures that entities to be merged have identical topology.

## Examining Merged Entities

There are several mechanisms for examining which entities have been merged. The most useful mechanism is assigning all merged or unmerged entities of a specified type to a group, and examining that group graphically. This process can be used to examine the outer shell of an assembly of volumes, for example to verify if all interior surfaces have been merged. To put all the merged entities of a given type into a specified group, use the command:

**Group {<`name'>|<id>} add [Surface | Curve | Vertex] with Is_merged**

To put all the unmerged entities of a given type into a specified group, use the command:

**Group {<`name'>|<id>} add [Surface | Curve | Vertex] with Is_merged=0**

Entities can also be labeled in the graphics according to the state of their merge flag. See the Preventing geometry from merging section for information on controlling the merge flag. To turn merge labeling on for a specified entity type, use the command

**Label {Vertex | Curve | Surface} Merge**

## Imprinting Geometry

To produce a non-manifold geometry model from a manifold geometry, coincident surfaces must be merged together (See Geometry Merging); this merge can only take place if the surfaces to be merged have like topology and geometry. While various parts of an assembly will typically have surfaces, which coincide geometrically, an imprint is necessary to make the surfaces have like topology. There are three types of imprinting:

- Regular Imprinting
- Tolerant Imprinting
- Mesh-Based Imprinting

To preview which surfaces can or should be imprinted, or to force imprints that the regular imprint command misses, the Find Overlap command can be used.

*Regular Imprinting*

The commands used to imprint bodies together are:

**Imprint [Volume|BODY] <range> [with [Volume|BODY] <range>] [Keep]**

A body can also be imprinted with curves, vertices or positions, and surfaces can be imprinted with curves. It is useful to imprint bodies or surfaces with curves to eliminate

mesh skew, generate more favorable surfaces for meshing, or create hard lines for paving. Imprinting with a vertex or position can be useful to split curves for better control of the mesh or to create hard points for paving. Impriting a vertex onto a volume allows for a tolerance to be specified, snapping the vertex to the closest location on the volume that is within tolerance.

**Imprint Body <body_id_range> [with] Curve <curve_id_range> [Keep]**

**Imprint Body <body_id_range> [with] Vertex <vertex_id_range> [tolerance <value>] [Keep]**

**Imprint {Volume|Body} [with] Position <coords> [position <coords> ... ]**

**Imprint Surface <surface_id_range> [with] Curve <curve_id_range> [Keep]**

An **Imprint All** will imprint all bodies in the model pairwise; bounding boxes are used to filter out imprint calls for bodies which clearly don't intersect.

**Imprint [Body] All**

*Tolerant Imprinting*

Normal imprinting may be ineffective for some assembly models that have tolerance problems, generating unwanted sliver entities or missing imprints altogether. Tolerant imprinting is useful for dealing with these tolerance challenged assemblies. To determine coincident and overlap entities, tolerant imprinting uses the merge tolerance. The commands also include an optional **tolerance** value that will be used for the purposes of the single command. Specifying an optional **tolerance** value will not change the default, system tolerance value.
A limitation of tolerant imprinting is that it cannot imprint intersecting surfaces onto one another, as normal imprinting can. Tolerant imprinting imprints only *overlapping* entities onto one other.

**Imprint Tolerant {Body|Volume} <range> [tolerance <value>]**

Tolerant imprinting can also be used to imprint curves onto surfaces, provided that the tolerance between surface and curve(s) falls within the merge tolerance. The 'merge' option will merge the owning volume of the specified surface with all other volumes that share any curves with this surface.

**Imprint Tolerant Surface <id> with Curve <id_range> [merge] [tolerance <value>]**

**Imprint Tolerant Surface <id> <id> with Curve <id_range> [merge] [tolerance <value>]**

**Imprint Tolerant Surface <id> <id> [tolerance <value>]**

The second form of the command imprints the specified bounding curves of one surface onto another surface and vice versa. Any specified curves that are not bounding either of the two specified surfaces will not be imprinted. The 'merge' option will merge all the volumes sharing any curve of these two surfaces, after the imprint.
It is recommended that normal imprinting be used when possible and tolerant imprinting be used only when normal imprinting fails.

*Mesh-Based Imprinting*

Another form of the imprint command,

**Imprint Mesh {Body | Volume} <id_list>**

uses coincident mesh entities and virtual geometry to create imprints. See the Partitioned Geometry section for more information on this command.

*Imprint Settings*

After imprint operations, an effort is made to remove sliver entities: sliver curves and surfaces. Previously, all curves in participating bodies less than 0.001 were removed. Newer versions of Cubit changed this because there might be times when the user wants sliver curves/surfaces to be generated during an imprint operation. In order to give the user more control over the cleanup of these sliver entities after imprint operations, a command was implemented so that the user can set an 'imprint sliver cleanup tolerance'. The default tolerance for curves is the merge tolerance 0.0005. The default tolerance for surfaces is a suitable tolerance chosen internally based on the bounding box of the entity. Sliver surfaces are removed whose maximum gap distance among the long edges is smaller than the tolerance and who have at most three long edges. A long edge is an edge whose length is greater than the specified tolerance.

**Set {Curve|Surface} Imprint Cleanup Tolerance <value>**

## Merge Tolerance

Geometric correspondence between entities is judged according to a specified absolute numerical tolerance. The particular kind of spatial check depends on the type of entity. Vertices are compared by comparing their spatial position; curves are tested geometrically by testing points 1/3 and 2/3 down the curve in terms of parameter value; surfaces are tested at several pre-determined points on the surface. In all cases, spatial checks are done comparing a given position on one entity with the closest point on the other entity. This allows merging of entities which correspond spatially but which have different parameterizations.
The default absolute merge tolerance used in CUBIT is 5.0e-4. This means that points which are at least this close will pass the geometric correspondence test used for merging. The user may change this value using the following command:

**Merge Tolerance <val>**

If the user does not enter a value, the current merge tolerance value will be printed to the screen. There is no upper bound to the merge tolerance, although in experience there are few cases where the merge tolerance has needed to be adjusted upward. The lower bound on the tolerance, which is tied to the accuracy of the solid modeling engine in CUBIT, is 1e-6.

*Finding Nearly Coincident Entities*

These commands find vertex-vertex, vertex-curve and vertex-surface pairs whose separation is within the specified tolerance range. If a tolerance range isn't specified the default will be from merge tolerance to 10*merge tolerance. It is useful for determining if you need to expand merge tolerance to accomodate sloppy geometry.

**Find Near Coincident Vertex Vertex {Body|Volume} <id_range> [low_tol <value>] [high_tol <value>]**

**Find Near Coincident Vertex Curve {Body|Volume} <id_range> [low_tol <value>] [high_tol <value>]**

**Find Near Coincident Vertex Surface {Body|Volume} <id_range> [low_tol <value>] [high_tol <value>]**

## Merging Geometry

The steps of the geometry merging algorithm used in CUBIT are outlined below:

1. Check lower order geometry, merge if possible
2. Check topology of current entities
3. Check geometry of current entities
4. If both entities are meshed, check topology of meshes.
5. If geometric topology, geometry, and mesh topology are alike, merge.

Thus, in order for two entities to merge, the entities must correspond geometrically and topologically, and if both are meshed must have topologically equivalent meshes. The geometric correspondence usually comes from constructing the model that way. The topological correspondence can come from that process as well, but also can be accomplished in CUBIT using Imprinting.
If both entities are meshed, they can only be merged if the meshes are topologically identical. This means that the entities must have the same number of each kind of mesh entity, and those mesh entities must be connected in the same way. The mesh on each entity need not have nodes in identical positions. If the node positions are not identical, the position of the nodes on the entity with the lowest ID will be used in the resulting merged mesh.
There are several options for merging geometry in CUBIT.

*Merge geometry automatically*

**Merge All [Group|Body|Surface|Curve|Vertex] [group_results][tolerance <value>]**

All topological entities in the model or in the specified bodies are examined for geometric and topological correspondence, and are merged if they pass the test.
If a specific entity type is specified with the Merge all, only complete entities of that type are merged. For example, if Merge all surface is entered, only vertices which are part of corresponding surfaces being merged; vertices which correspond but which are not part of corresponding surfaces will not be merged. This command can be used to speed up the merging process for large models, but should be used with caution as it can hide problems with the geometry.

*Test for merging in a specified group of geometry*

**Merge {Group|Body|Surface|Curve|Vertex} <id_range>[With {Group|Body|Surface|Curve|Vertex} <id_range>] [group_results] [force] [tolerance<value>]**

All topological entities in the specified entity list, as well as lower order topology belonging to those entities, are examined for merging. This command can be used to prevent merging of entities which correspond and would otherwise be merged, e.g. slide surfaces.

*Force merge specified geometry entities*

**Merge Vertex <id> with Vertex <id> Force**

**Merge Curve <id> with Curve <id> Force**

**Merge Surface <id> with Surface <id> Force**

This command results in the specified entities being merged, whether they pass the geometric correspondence test or not. This command should only be used with caution and when merging otherwise fails; instances where this is required should be reported to the CUBIT development team.

*Preventing geometry from merging*

**Body <id_range> Merge [On | Off]**

**Volume <id_range> Merge [On | Off]**

**Surface <id_range> Merge [On | Off]**

**Curve <id_range> Merge [On | Off]**

**Vertex <id_range> Merge [On | Off]**

These commands provide a method for preventing entities from merging. If merging is set to off for an entity, merging commands (e.g. "merge all") will not merge that entity with any other.

*Other Merge Commands*

**Set Merge Test BBox {on|OFF}**

This is an additional test for merging to see if a pair of surfaces should merge. First, it creates a bounding box for each surface by summing individual bounding boxes of each of the surface's curves. A comparison is then made to see if these two bounding boxes are within tolerance. This can help to weed out any potential incorrect merges that can result from non-tight bounding boxes.

**Set Merge Test InternalSurf {on|OFF|spline}**

This is an extra check when merging surfaces. A point on one surface, closest to its centroid is found. Another point, closest to this point is found on the other surface. If these two points are not within merge tolerance, the two surfaces will not be merged. If set to **on**, all surface types will be included in this check. If set with the **spline** option, then splines are only checked this way; analytic surfaces are excluded. This is another check to prevent incorrect merges from occurring.

## Using Geometry Merging to Verify Geometry

Geometry merging is often used to verify the correctness of an assembly of volumes. For example, groups of unmerged surfaces can be used to verify the outer shell of the assembly (see Examining Merged Entities.) There is other information that comes from the **Merge all** command that is useful for verifying geometry.

In typical geometric models, vertices and curves which get merged will usually be part of surfaces containing them which get merged. So, if a **Merge all** command is used and the command reports that vertices and curves have been merged, this is usually an indication of a problem with geometry. In particular, it is often a sign that there are overlapping bodies in the model. The second most common problem indicated by merging curves and vertices is that the merge tolerance is set too high for a given model. In any event, merged vertices and curves should be examined closely.

## Unmerging

The unmerge command is used to reverse the merging operation. This is often in cases where further geometry decomposition must be done.

**Unmerge {all|<entity_list> [only]}**

Un-merging an entity means that the specified geometric entity and all lower-order (or child) entities will no longer share non-manifold topology with any other entities. For

example, if a body is unmerged, that body will no longer share any surfaces, curves, or vertices with any other body.

**[Set] Unmerge Duplicate_mesh {On|OFF}**

If any meshed geometry is unmerged, the mesh is kept as necessary to keep the mesh of higher-order entities valid. For example, if a surface shared by two volumes is to be unmerged and only one of the volumes is meshed, the surface mesh will remain with whichever surface is part of the meshed volume.
When unmerging meshed entities, the default behavior of the code is that the placement if the mesh is determined by the following rules:

- If neither entity has meshed parent entities, the mesh is kept on one of the two entities.
- If one entity has a meshed parent entity, the mesh is kept on that entity.
- If both entities have meshed parents, the mesh is kept on one and copied on the other.

If **unmerge duplicate_mesh** is turned on, the rules described above are overwritten and whenever a meshed entity is unmerged the mesh is always copied such that both entities remain meshed.
To get back to the default behavior, turn **unmerge duplicate_mesh** off.

# virtual geometry

## Virtual Geometry

- Composite Geometry
- Partitioned Geometry
- Collapsing Geometry
- Simplify Geometry
- Deleting Virtual Geometry

The Virtual Geometry module in CUBIT provides a way to modify the topology of the model without affecting the underlying ACIS geometry representation and without making changes to the actual solid model. Virtual Geometry includes the capability to composite or partition geometry as well as creates new virtual geometric entities. Virtual Geometry operations are most often used as a tool for adjusting the geometry to allow mapping, sub-mapping or sweeping mesh generation schemes to be applied.
The advantage to using Virtual Geometry is that all operations are reversible. With standard geometry modification commands, changes are made to the underlying geometry representation and cannot be changed once effected. With virtual geometry, the original solid model topology can be easily restored. This is useful when geometry modifications are made in order to apply a particular meshing scheme. Virtual geometry can be applied and later removed once the part has been meshed.

# *Collapse Angle*

The collapse command allows the user to collapse small angles using virtual geometry. The command syntax is:

**Collapse Angle at Vertex <id range> [angle <degrees>] [Curve <id1> [Arc_length <length>]] [Curve <id2> [Arc_length <length> | Same_size | Perpendicular | Tangent]] [Composite_vertex <angle>] [Preview]**

The collapse angle command is used to eliminate small angles at vertices, where curves meet at a tangential point. The command will split each curve at a specified distance (δ1 and δ2 ) as shown in Figure 1, and create two new vertices along those curves. The remaining small angle will be composited into its neighboring surface using virtual geometry. One of two methods may be used for specifying options for the **collapse angle** command: **(1) Simple**, and **(2) Complete** as described below:

**(1) Simple**

Curves are not specified for this option. Instead, Cubit will automatically identify the smallest angle at the vertex and collapse it using the **tangent** option described below (see Figure 4). An optional **angle** option may also be specified that controls the arc length (δ1) along curve C1 where the curve will be split. If not specified a value of 30 degrees will be used. For the **simple** option of this command, a range of vertices may be used. The **complete** version of the command requires exactly one vertex and two curves.

**(2) Complete**

The complete options of the command allow you to specify which curves and where to split each curve. You must input a distance for the first curve ( δ1), but the second location can be determined based on the length and direction of the first curve.

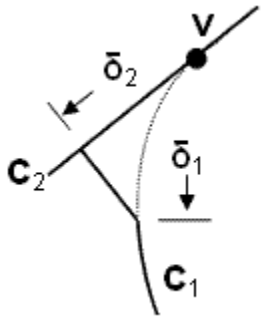



**Figure 1. Collapse angle syntax**

The **arclength** option will split each curve at a specified distance δ1 and δ2, (See Figure 1) measured from the vertex. You must input at least one **arclength** for each of the options listed below.

The **same_size** option will split curve 2 so that the two resulting curves, δ1 and δ2, are the same length as shown in Figure 2.

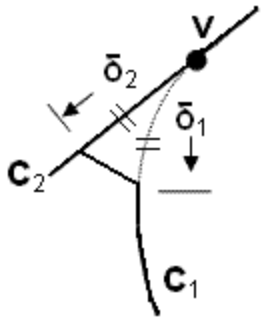



**Figure 2. Collapse angle using the same_size option**

The **perpendicular** option will split curve 2 so it is perpendicular to the split location on curve 1, as shown in Figure 3.

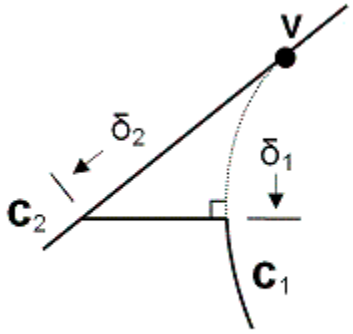



**Figure 3. Collapse angle using the perpendicular option**

The **tangent** option will split curve 2 where a line tangent to curve 1 at the split location intersects curve 2, as shown in Figure 4.

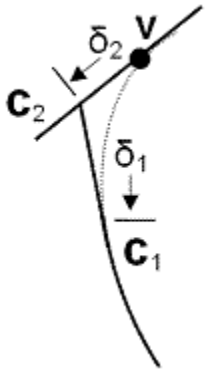



**Figure 4. Collapse angle using the tangent option**

The **composite_vertex** option automatically composites resulting surfaces if there are only two curves left at the vertex, and the angle is less than a specified tolerance.
The **preview** option will preview composited surface before applying changes.

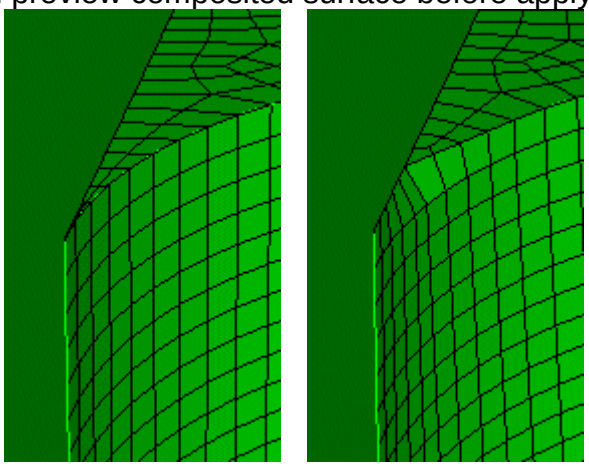


**Figure 5. An example of a meshed surface that is generated after using the collapse angle command.**

# *Collapse Curve*

The collapse curve command allows the user to collapse small curves using virtual geometry. It is intended to be used in cases where removing a small curve to simplify topology will facilitate meshing. The operation can be thought of as reconnecting curves from one vertex on the small curve to the other vertex. If the user doesn't specify which vertex to keep during the operation CUBIT will choose one of the vertices. The operation is performed using virtual partitions and composites on the curves and surfaces surrounding the small curve. The command syntax is:

**Collapse Curve <id> [Vertex <id>] [Ignore] [Real_split]**

The **vertex** keyword allows the user to specify which vertex on the small curve to keep during the operation or in other words which vertex to "collapse to". Depending on the surrounding topological configuration some vertices cannot currently be chosen so if the user specifies a vertex to collapse to that results in a complex topological configuration that CUBIT can't currently handle the user will be notified and encouraged to pick a different vertex. If the user doesn't specify a vertex CUBIT will attempt to choose the "best" vertex to keep based on surrounding topology and geometry. Currently, the collapse curve command only handles curves where the vertex that is NOT retained has a valence of 3 or 4.
The **ignore** keyword allows the user to specify whether or not small portions of surfaces that are partitioned off of one surface and composited with a neighboring surface during the collapse curve operation are considered when evaluating the new composite surface.

By specifying the **ignore** option the user tells CUBIT that these small surfaces will be ignored in future evaluations of the composite surface. This can be beneficial in cases where the small surface makes a sharp angle with the neighboring surface it is being composited with. These first derivative discontinuities of composite surfaces can make it difficult for the meshing algorithms to proceed and ignoring the small surfaces during evaluation can help remedy this problem. By default the small surfaces will not be ignored. The **real_split** option tells CUBIT to use the solid modeling kernel's (ACIS) split surface functionality to do the splitting rather than using virtual partitioning. The result is that you only have virtual composites at the end and no virtual partitions. The main advantage of using this option is that the solid modeling kernel's split operation is often more reliable than the virtual partition.
Figure 1 shows a typical example where the collapse curve command should be used to simplify the topology for meshing.

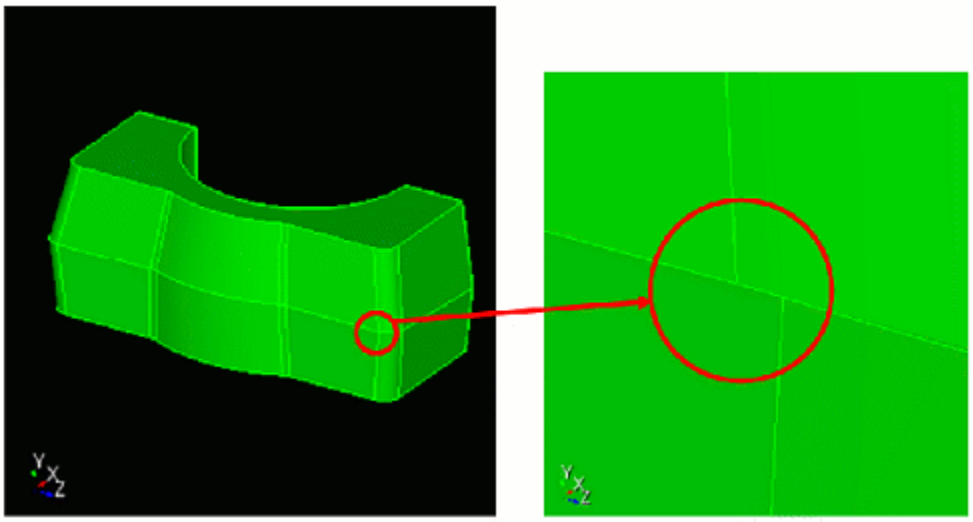


**Figure 1. Example where the collapse curve operation is needed.**

Figure 2 shows the above example after collapsing the small curve

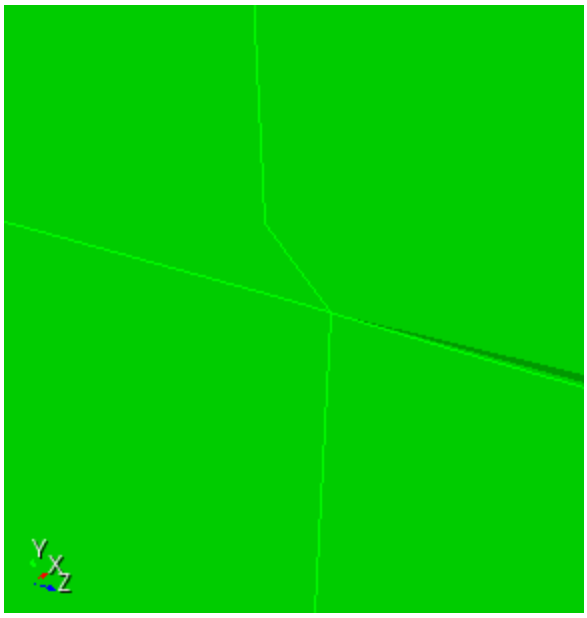



**Figure 2. Above example after collapsing the small curve.**

# *Collapse Geometry*

The collapse geometry commands use virtual geometry to tweak small angles and curves to improve meshability of geometry models. The following options for collapsing geometry are available:

- Collapse Angle
- Collapse Curve
- Collapse Surface

# *Collapse Surface*

The collapse surface command allows the user to remove surface boundaries from the model. This is accomplished by splitting the surface at two given locations and combining it into two adjacent surfaces using virtual geometry operations. The command syntax is:

**Collapse Surface <id> Across Location1 Location 2 With Surface <id_list> [Preview]**

The locations option can use any of the general Cubit location commands. However, the vertex and curve options are among the most useful location options. For example, the command

**collapse surface 15 across vertex 128 curve 40 with surface 26 117**

would split surface 15 by the line that is formed between vertex 128 and the midpoint of curve 40. It would then composite the two parts of surface 15 that are adjacent to surfaces 26 and 117. The result is that three surfaces have been reduced to two.

The collapse surface command is most useful in removing blended surfaces (i.e. fillets and chamfers) from a model. For example, Figure 1 below shows a set of highlighted surfaces on a bracket. By collapsing all these surfaces the model shown in Figure 2 is created. Collapsing the surfaces for this model simplifies the model and allows for the creation of a higher quality mesh.

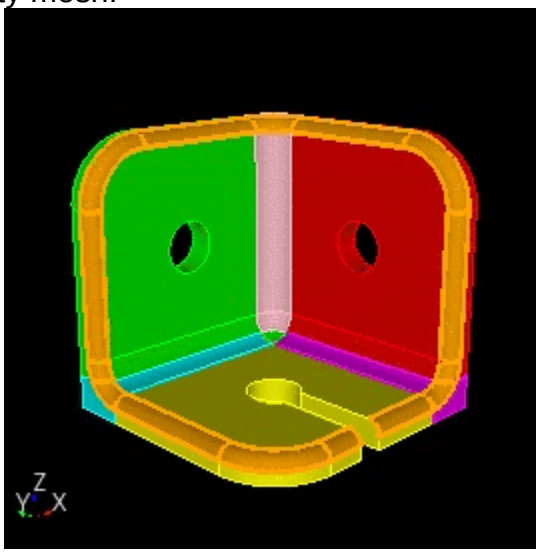


**Figure 1. Bracket with chamfered edges.**

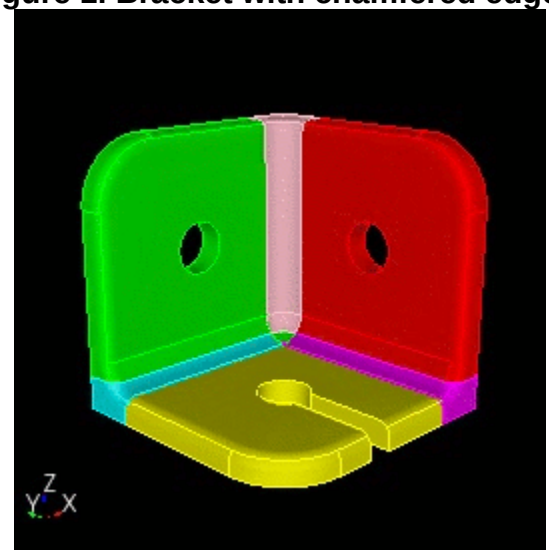


**Figure 2. Bracket after highlighted edges have been collapsed**

# Composite Curves

The full command for the creation of composite curves is:

**Composite Create Curve <id_range> [Keep Vertex <id_list>] [Angle <degrees>]**

The additional arguments provide two methods to prevent vertices from being removed from the model or *composited* over. The first method, **keep vertex** explicitly specifies vertices which are not to be removed. This option can also be used to control which vertex is kept when compositing a set of curves results in a closed curve.
The **angle** option specifies vertices to keep by the angle between the tangents of the curves at that vertex. A value less than zero will result in no composite curves being created. A value of 180 or greater will result in all possible composites being created. The default behavior is an empty list of vertices to keep, and an angle of 180 degrees.

# Composite Geometry

- Composite Curves
- Composite Surfaces

The virtual geometry module has the capability to combine a set of connected curves into a single composite curve, or a set of connected surfaces into a single surface. The general purpose is to suppress or remove the child geometry common to those entities being composited. For example, compositing a set of curves suppresses the vertices common to those curves, thus removing the constraint that a node must be placed at that vertex location.
The basic form of the command to create composites is:

**Composite Create {Surface|Curve} <id_list>**

This command will composite as many surfaces (or curves) as possible, in many cases creating multiple composites.
The entities combined to create the composite must either all be unmeshed or all be meshed. A meshed composite surface can not be removed unless the mesh is first deleted.
Care should be taken when compositing over large $C^1$ discontinuities as it may cause problems for the meshing algorithms and may result in poor quality elements. $C^1$ discontinuities are corners or abrupt changes in the surface normal.
The command to remove a composite is:

**Composite Delete {Surface|Curve} <id>**

# *Composite Surfaces*

The general command for composite surface creation is:

**Composite Create Surface <id_range> [Angle <degrees>] [Nocurves] [Keep [Angle <degrees>] [Vertex <id_list>]]**

**Related Commands**

**Graphics Composite {on|off}**

The **angle** argument prevents curves from being removed from the model or *composited* over. Composites will not be generated where the angle between surface normals adjacent to the curve is greater than the specified angle.
When a composite surface is created, the default behavior is to also to composite curves on the boundary of the new composite surface.
Curves are automatically composited if the angle between tangents at the common vertex is less than 15 degrees. The **nocurves** option can be used to prevent any composite curves from being created.
The **keep** keyword can be used to change the default choice of which curves to composite. The arguments following the **keep** keyword behave the same as for explicit composite curve creation. The **nocurves** and **keep** arguments are mutually exclusive.

#### Controlling the Surface Evaluation Method for Composite Surfaces

It typically takes longer to mesh a single composite surface than to mesh the surfaces used in the creation of the composite. To improve speed, composite surfaces use an approximation method to evaluate the closest point to a trimmed surface. However, this evaluation method may give poor results for composites of highly convoluted surfaces.
The virtual geometry module provides a way to change the way surfaces are evaluated using the following command:

**Composite Closest_pt Surface <id> {Gme|Emulate}**

The default behavior is to use the **emulate** method, as it is typically considerably faster. Specifying the **gme** option will force the specified composite surface to use the exact calculation of the closest point to a trimmed surface, as provided by the solid modeler. The **gme** option, however, can be considerably slower.

#### Composite Determination

The **composite create surface** command is non-deterministic in some circumstances. When three or more adjacent surfaces are to be composited, all the surfaces may not be able to be composited into a single surface as illustrated in Figure 1. In this case different subsets of the surfaces may be composited and the command will choose arbitrary subsets to composite. As an example, there are three surfaces A, B, and C, all adjacent to each other. The common curve between A and B is AB, the common curve between B and C is BC, and the common curve between A and C is CA. If the curve BC cannot be removed, either due to the angle specified in the composite command, or because there is a fourth surface, D, also using that curve, the command will arbitrarily choose to either composite A and B or A and C.

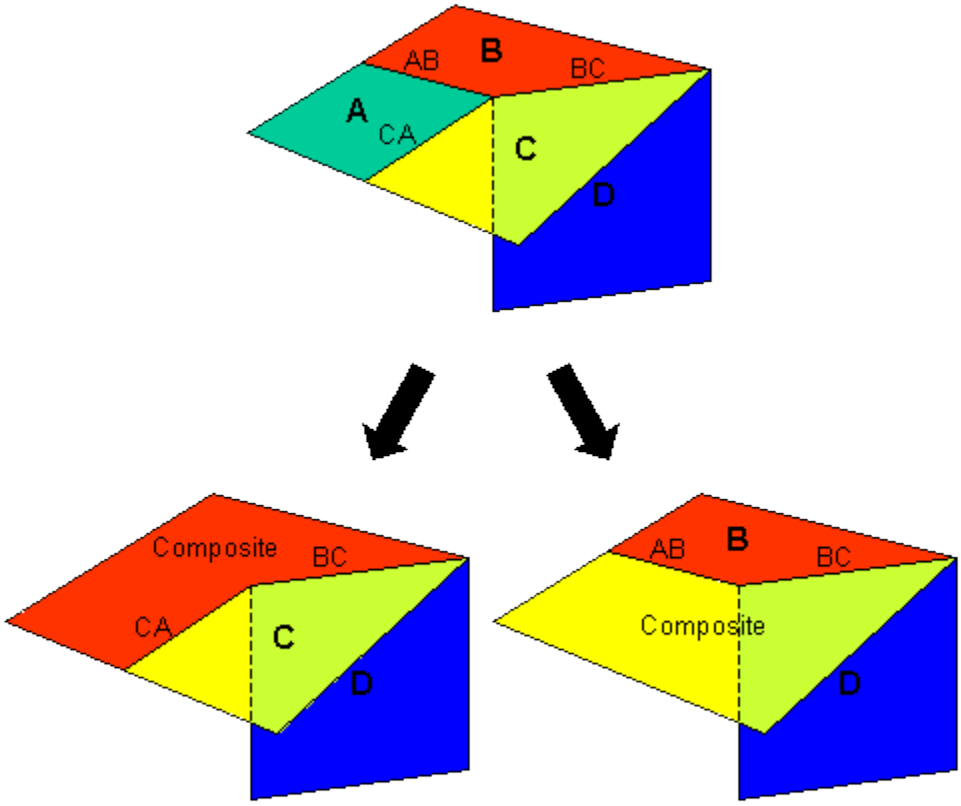



**Figure 1. In some cases, the program will make a determination of which surfaces to composite.**

partitioned geometry

# *Removing Partitions*

There are two commands used to remove partitions:

**Partition Merge {Curve|Surface|Volume} <id_list>**

The command combines existing partitions where possible. This command is similar to the composite create command. The difference is that this command is special-cased for partitions, and will result in more efficient geometric evaluations. If all the partitions of a real solid model entity are merged, such that there is only one partition remaining, the virtual geometry will be removed, and the original solid model geometry will be restored to the model.
The CUBIT delete command can also be used for removing partitions. See Deleting Virtual Geometry for a description of its use.

# *Using Mesh Intersections to Partition Surfaces*

To assist in various mesh editing tasks such as joining, a *mesh-based imprinting* capability is provided. The command

```
Imprint Mesh {Body | Volume} <id_list>
```

determines imprint locations using the mesh on the surfaces of the specified bodies or volumes. Regions of coincidence between the surfaces is determined by searching for coincident nodes in the mesh of the surfaces. Virtual geometry is then used to partition the surfaces and curves at the boundary of these regions of coincident mesh.
The **imprint mesh** functionality differs from a normal geometric imprint in the following ways:

- The location of the imprint is determined from coincidence of mesh nodes.
- The mesh remains intact through the imprint operation.
- Virtual geometry is used to create the imprint.
- The imprinting can be done on all types of geometry (including mesh-based geometry, merged geometry, and virtual geometry.)

The following is a trivial example of this capability. The following commands create two meshed blocks:

```
brick                    width                    10
brick                    width                    6
body          2          move          x          8
volume          1          2          size          1
mesh volume 1 2
```

Figure 1 shows the results of these commands.

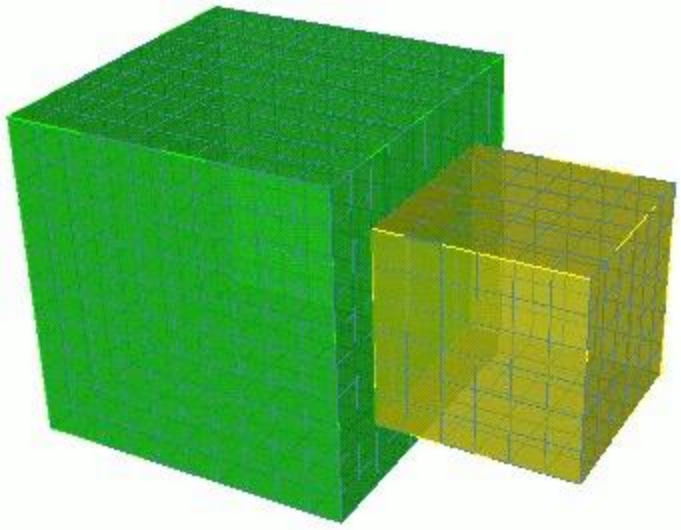


**Figure 1. Two adjacent meshed volumes. The coincident meshes will form the basis of the imprint operation.**

The mesh of the blocks can be joined by first doing a mesh-based imprint and then merging:

```
imprint mesh body 1 2
merge body 1 2
```

Figure 2. shows the results of the imprint operation. A meshed surface is created at the interface between the two meshed volumes. The nodes on the new surface are shared by the neighboring hexahedra of both volumes.

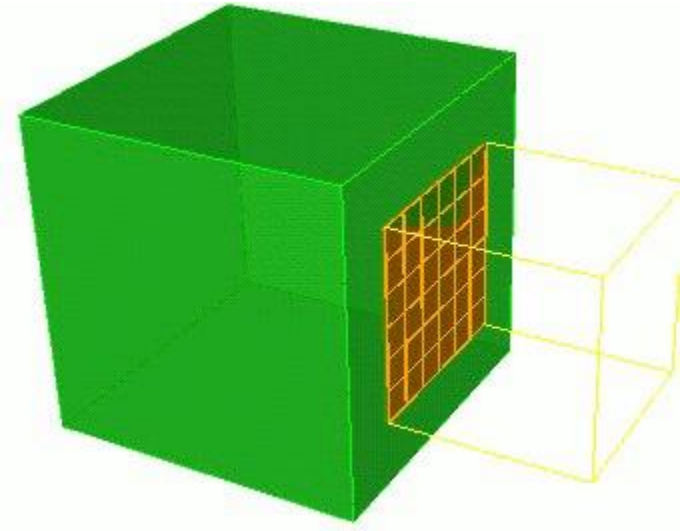


**Figure 2. The imprinted surface. Adjacent volume meshes joined at the interface surface.**

# *Partitioned Curves*

There are four methods for specifying locations at which to partition curves:

**Partition Create Curve <curve_id> {Fraction <fraction_list> | Position <xpos> <ypos> <zpos> | [with] <vertex_list> | <node_list> }**

The first two forms of the command create additional vertices and use those vertices to split a curve. The third form of the command uses existing vertices to split the curve. The fourth form of the command uses existing nodes to split the curve.
Using the **fraction** option, vertices are created at the specified fractions along the curve (in the range [0,1].) Subsequently, the curve is split at each vertex, resulting in n+1 new curves, where n is the number of fraction values specified.
Using the **position** option, vertices are created at the closest location along the curve to each of the specified position. Subsequently, the curve is split at each vertex, resulting in n+1 new curves, where n is the number of positions specified.
If the **node** option is used, meshed curves may be partitioned. The specified nodes must lie on the curve to be partitioned. The curve is split at each node specified, and any other mesh entities are divided appropriately amongst the curve partitions.

# *Partitioned Surfaces*

There are several forms of the command to partition a surface. A surface may be partitioned using hard points, curves, polylines, mesh edges, mesh faces or mesh triangles.

- Partitioning with Vertices or Nodes
- Partitioning with Curves
- Partitioning with Mesh Edges
- Partitioning with Mesh Faces or Triangles

#### Partitioning with Vertices and Nodes

##### Partitioning with Hard Points

There are two methods of partitioning a surface using vertices and nodes. The first method is to create a set of hard points using nodes, vertices, or coordinates that constrain the mesh to particular points on the surface. The syntax is:

**Partition Create Surface <id> Vertex <id_list> [Individual]**

**Partition Create Surface <id> Node <id_list> [Individual]**

##### Partitioning with Polylines

The second method is to define a polyline using a set of vertices or coordinates. This method splits the surface using a polyline defined by the a list of positions specified as

either coordinate triples, or existing vertices. The polyline is projected to the surface to define the curve for splitting the surface. If only one position is specified a zero-length curve with a single vertex will be created The syntax is identical to above WITHOUT the individual option.

**Partition Create Surface <id> Vertex <id_list>**

**Partition Create Surface <id> Position <x> <y> <z> [[Position] <x> <y> <z> ...]**

In the following simple example, the surface is partitioned using both methods. On the left half of the object, the surface is partitioned using the individual option (vertices 11 12 15 13). On the right half, a polyline is used (vertices 9 10 16 14). All of the free vertices can then be deleted, leaving the virtual curves shown in the second picture. Vertices 19 20 21 and 22 are all zero-length curves. The small 'v' in parentheses is to indicate that it is virtual geometry. The resulting mesh is shown in the third picture. Notice that the polyline constrains the entire curve to the mesh, while the hardpoints constrain only that individual point.

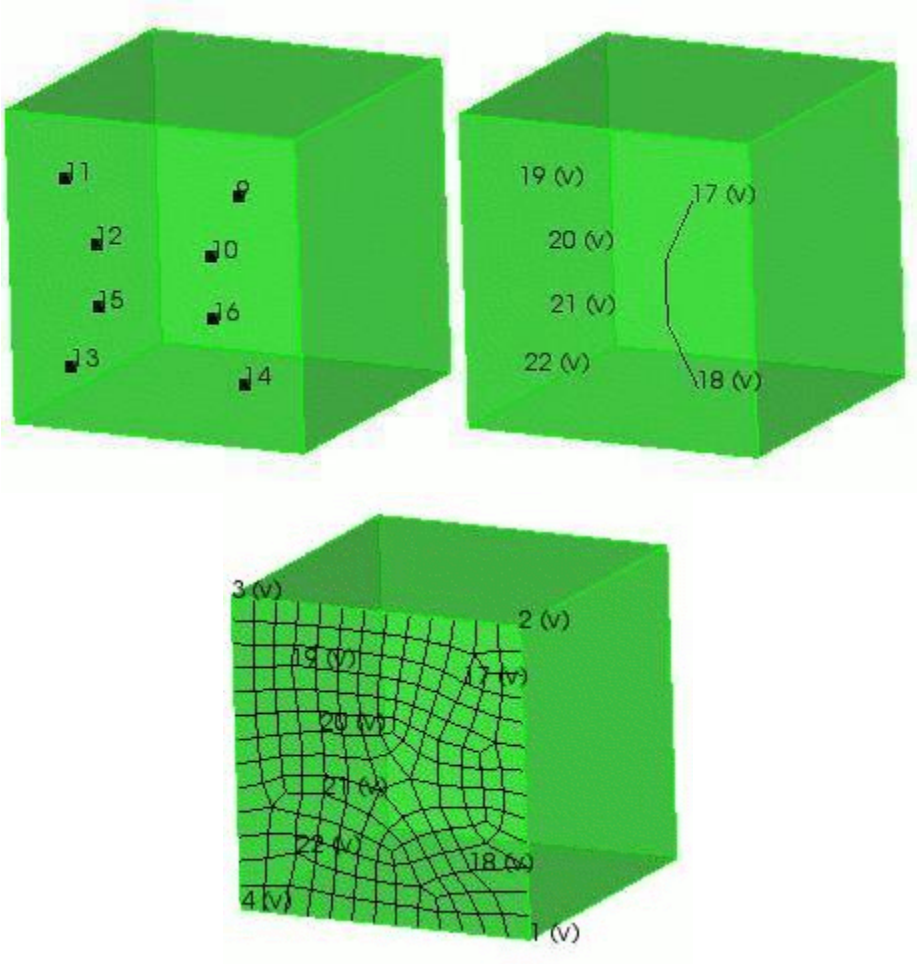



**Figure 1. Partitioning a Surface Using Vertices**

#### Partitioning with Curves

This form of the command splits the existing surface into several surfaces by creating curves that approximate the projection of the specified existing curves onto the surface. The syntax is:

**Partition Create Surface <id> Curve <id_list>**

#### Partitioning with Mesh Edges

Meshed surfaces may be partitioned with mesh edges. The specified mesh edges must be owned by the surface to be partitioned. The shape of the curve(s) used to split the surface is specified by a set of mesh edges.
If the split location is specified by a series of mesh edges, and the specified mesh edges form a closed loop, the node option may be used to control which node the vertex is created at.

**Partition Create Surface <id> Edge <id_list> [Node <node_id>]**

#### Partitioning with Faces or Triangles

Surfaces may also be partitioned by specifying a list of triangles or faces (quads). The boundary of the list will automatically be detected and new curves and vertices created at the appropriate locations. Curves are created from the mesh edges and used to split the surface. The surface mesh is split and assigned to the appropriate surface partitions.

**Partition Create Surface <id> Face|Tri <id_list>**

# *Partitioned Volumes*

To partition a volume by giving a center and radius:

**Partition Create Volume <id> Center [Location] {options} Radius <val>**

This command splits the existing volume into two volumes. All volume elements that lie within the specified radius of the specified center location are identified, and the exterior faces of these elements are used to create a surface and partition the volume. The center can be specified with any of the location options.
Figure 1 shows an example of a partitioned volume. A cube that has been map meshed is partitioned using a center at one of its vertices. The result is two distinct volumes with a surface separating the two. The interface surface is composed of the faces of the interior hex elements.

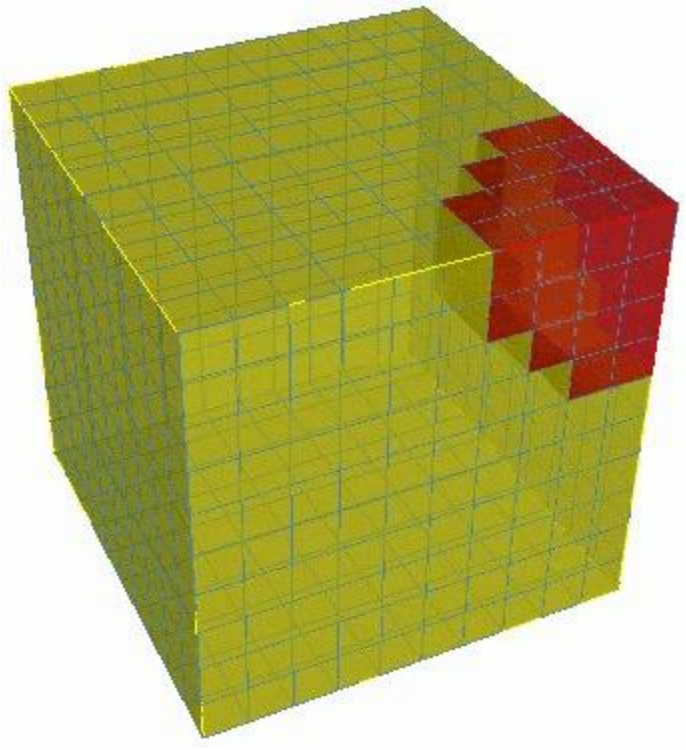


**Figure 1. A partitioned volume**

This command may be useful for separating small regions of a meshed volume so that remeshing or mesh improvement may be performed locally.

# *Partitioned Geometry*

Partitioning provides a method to introduce additional topology into the model, to better constrain meshing algorithms. This is accomplished by splitting, or partitioning, existing curves or surfaces.

- Partitioned Curves
- Partitioned Surfaces
- Partitioned Volumes
- Using Mesh Intersections to Partition Surfaces
- Removing Partitions

## Deleting Virtual Geometry

*Removing Virtual Geometry*

The following command removes all lower-order virtual geometry from the specified entities.

**Virtual Remove <entity_list>**

**Examples:**

**virtual remove surface 5**

Removes all composite and partition curves from surface 5.

**virtual remove body all**

Remove all virtual geometry from all bodies.
For removing individual virtual entities, see the sections of the documentation for each type of virtual entity:

- Composite curves
- Composite surfaces
- Partition curves
- Partition surfaces

*Using The Delete Command With Composites*

If the general **delete** command is invoked for a composite surface, the composite surface will be removed, and the original surfaces used to define the composite will be restored to the model. The defining surfaces are NOT also deleted. As with any other non-virtual surfaces, the **delete** command will fail if the composite has a parent volume.
To delete composite surfaces with a parent volume, the composite delete command can be used. The behavior is analogous for composite curves.
If the **delete** command is used on a volume containing a composite surface or curve, or on a surface containing a composite curve, the entire volume or surface will be deleted, including the original entities used to define the composite, as those entities are also children of the entity being deleted.

*Using the Delete Command With Partitions*

It is recommended that the delete command not be used with partitions, as it may break subsequent usage of the merge and delete forms of the partition command for other partitions of the same real geometry entity. However, if the delete command is used for partitions, the behavior is to delete the specified partition, and when the last partition of the real geometry is deleted, to restore the original geometry.
The **delete** command can also be used on parents of partitions. For example, a volume containing partitioned surfaces, or a surface containing partitioned curves can be deleted. In this case, the specified entity will be deleted along with all of its children, including the partition entities, and the original entities that were partitioned.

## Simplify Geometry

Simplifying topology by compositing individually selected surfaces is often a tedious and time-consuming task. The simplify command addresses the tedium by automatically compositing surfaces and curves based on selected criteria between neighboring entities. Figure 1 shows a typical example of simplify command usage (‘simplify volume 1 angle 15’).

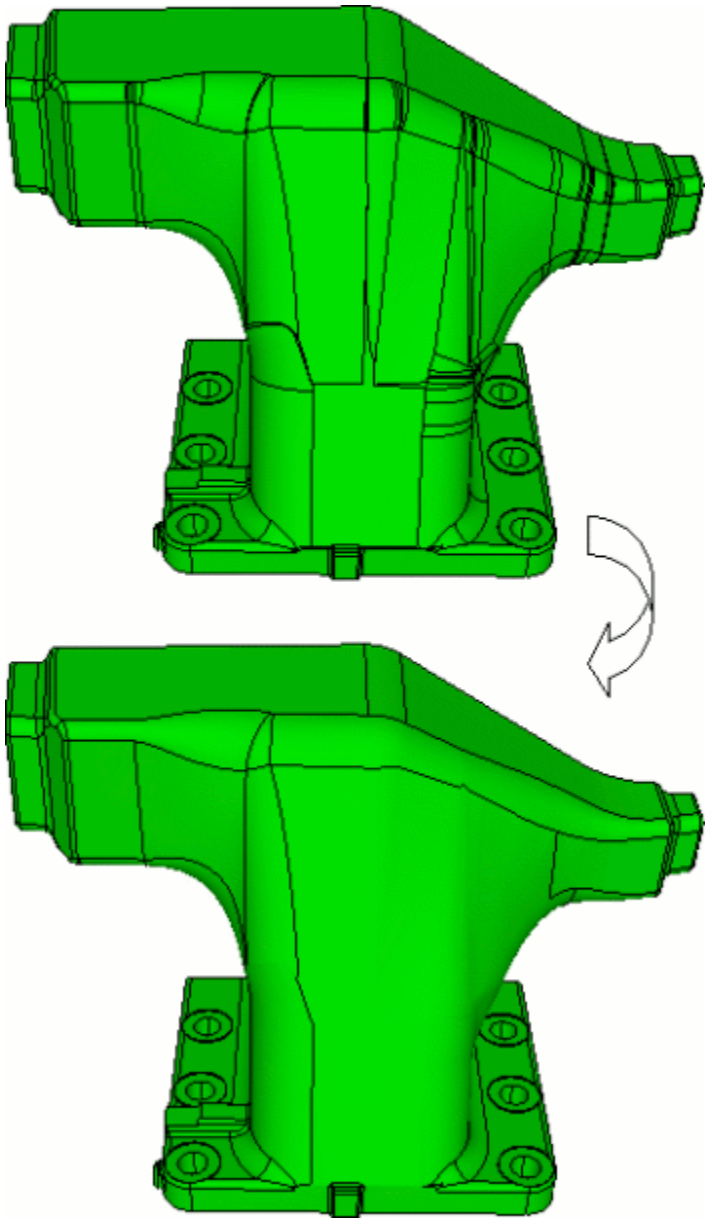


**Figure 1. Typical Simplify command usage**

The command syntax and discussion items are shown below.

**Simplify {Volume|Surface|Curve} <range> [Angle< value >] [Respect {Surface <id_range> | Curve <id_range> | Vertex <id_range>| Imprint | Fillet}] [Local_Normals] [Preview]**

*Feature Angle*

Feature angle is defined as the angle between the average facet normals of two neighboring surfaces. If the angle is less than the specified angle then the two surfaces are composited together (assuming any other specified criteria are met). Feature angle is always used as criteria and if an angle is not specified the value is set to 15 degrees.

*Automatically Compositing Curves*

The simplify command can also be used to automatically composite curves using an angle tolerance. Curves will be composited together only if they are explicitly specified in this command, and not as the result of two surfaces being composited.

*Respecting Vertices, Curves and Surfaces*

Surfaces, curves, and vertices can be specified to prevent geometry features from automatically being composited. Figure 2 show an example of respecting a surface ('simplify vol 1 angle 15 respect surf 289').

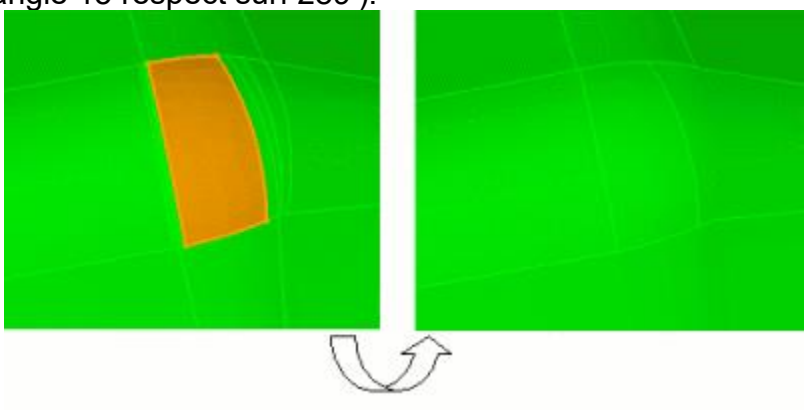


**Figure 2 Respecting a surface**

For complex geometries, it is often useful to preview the simplify command and then add any respected geometry to the command respect lists.

*Respecting Imprints*

Curves created by imprints can automatically be respected by the simplify command. Figure 3 shows an example of geometry with split fillets.

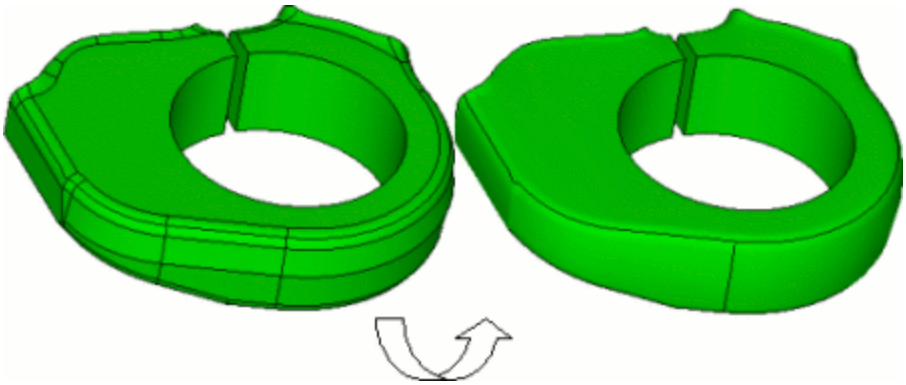


**Figure 3 Respecting imprint geometry**

Notice that in the split curves are respected by the Simplify command (‘simplify vol 1 angle 40 respect imprint’).

*Using Local Normals*

By default the command will compare the *average normal* of two adjacent surfaces to determine whether they should be composited. By issuing the *local_normal* option, the test will be modifed slightly. The modified test will compare the maximum difference between normals along the shared curve(s) for the two surfaces.

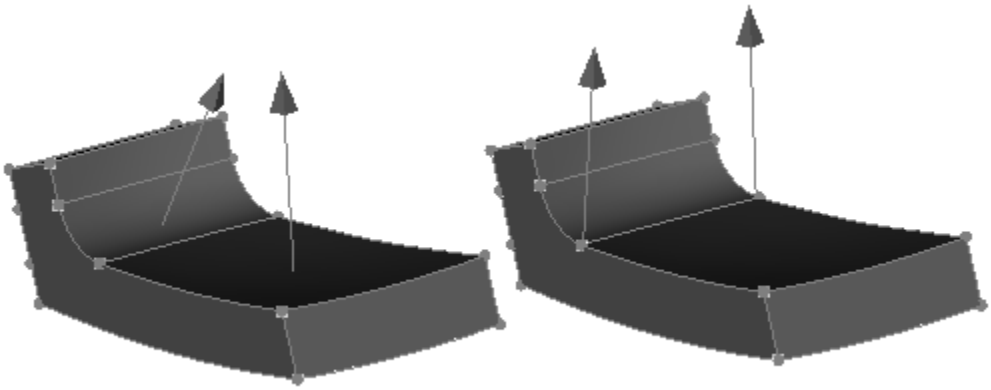


**Figure 4. Comparison of surface normals using the average surface normal method (on the left) and local normal method (on the right).**

*Other Options*

The *preview* option shows what curves are respected without compositing any surfaces. It should also be pointed out that multiple respect specifications can be chained together. For example:

**Simplify volume 1 angle 15 respect curve 1 respect imprint respect fillet preview**

# groups

## Groups

Groups are collections of geometry and mesh entities. Groups can contain other groups. Groups provide a powerful capability for organizing and performing operations on multiple entities with minimal input. Groups can collect entities according to various criteria, such as position, size, or whether they are meshed. Performing an operation on a group is usually the same as performing that operation on all the entities in that group. The following describes the Group operations available in CUBIT:

- Basic Group Operations
- Groups in Graphics
- Propagated Hex Groups
- Quality Groups
- Group Propagation

There are several utilities in CUBIT which use groups as a means of visualizing output. These utilities are described elsewhere, but listed here for reference:

- Webcut results
- Merged and unmerged entities
- Sweep groups
- Interval matching
- Disassociated Meshes
- Importing ACIS, IGES, STEP, Free Meshes

## Basic Group Operations

*Geometry Groups*

The common syntax to create or modify a group is to give it a name and a collection of entities:

**Group ["name"] Equals <list of entities>**

The *Equals* operation assigns the group to contain the given list. If the group already existed, its prior contents are overwritten. (Once a group is created, it can be refered to by id as well.) Here, entities can be geometry entities, mesh entities, or both. For example, the command,

**group "Exterior" equals surface 1 to 2, curve 3 to 5**

will create the group named (names are case sensitive). Any command taking entities can also take a group: e.g., **mesh Exterior**, **list Exterior**, or **draw Exterior**.

Geometry entities may specified by name as well. E.g.,

**group 'Interior' add surface with name 'bill' 'john' 'fred'**

will place the surfaces named 'bill' 'john' and 'fred' in the group Interior.

Wildcards (*) can also be used with names. To add all surfaces with the substring 'bob' in their name, use the command:

**group 'interior' equals surface with name '*bill*'**

There are a variety of operations for modifying the contents of a group, such as adding or removing entities. These can also be used to create a group from scratch by using a new name.

**Group ["name" | <id>] Add <entity list>**

**Group ["name" | <id>] Remove <entity list>**

**Group ["name" | <id>] Xor <entity list>**

*Group Booleans*

Groups may also be created from the contents of existing groups by set booleans: Group A boolean B preposition C. These operations will also overwrite the contents of existing groups. The **intersect** command will create a new group that contains elements common to both groups: A = B ∩ C. The **unite** command collects entities that exist in either group: A = B ∪ C. The **subtract** command collect the entities in one group but not the other: A = C \ B.

**Group {<'name'>|<id>} Intersect Group <id> with Group <id>**

**Group {<'name'>|<id>} Unite Group <id> with Group <id>**

**Group {<'name'>|<id>} Subtract Group <id> from Group <id>**

*Group Copy and Transform*

The contents of a group can be copied and transformed (e.g. moved), or simply transformed. (The group itself is not copied, only the contents.) This works only for groups of geometry entities, or groups of free mesh elements without associated geometry. Groups can collect free mesh entities in much the same way as a geometric entity. If the optional **copy** keyword is provided, the entities in the group will be copied first, and the copy will be transformed. Geometric and mesh groups behave differently. For geometry groups, if the group contains geometric entities that are meshed, the mesh will also be copied and transformed, unless the optional **nomesh** keyword is specified. You can copy a single surface of a volume, but you cannot move just it, because geometric entities cannot be transformed unless the parent entity is also. In addition, all merged partners must be contained in the group. However, any geometric entity can be copied and transformed. For mesh groups, transforming the nodes will implicitly transform the elements containing those nodes. The **copy** keyword is ignored if the group contains mesh entities. These types of mesh groups can only contain free mesh entities; for copying and transforming mesh associated to geometry, use a geometry group. (In the following, recall that an existing group name can always be used in place of 'Group <id>'.)

**Group <id> [Copy [nomesh]] [Move <dx> <dy> <dz>]**

**Group <id> [Copy [nomesh]] [Move {x|y|z} <distance>...]**

**Group <id> [Copy [nomesh]] [Move <direction> [distance]]**

**Group <id> [Copy [nomesh]] [Reflect {x|y|z}]**

**Group <id> [Copy [nomesh]] [Reflect <x> <y> <z>]**

**Group <id> [Copy [nomesh]] [Rotate <angle> About {x|y|z}]**

**Group <id> [Copy [nomesh]] [Rotate <angle> About <x> <y> <z>]**

**Group <id> [Copy [nomesh]] [Rotate <angle> About Vertex <Vertex_id1> <Vertex_id2>]**

**Group <id> [Copy [nomesh]] [Scale <scale> | x <val> y <val> z <val>]**

*Deleting Groups*

Groups can be deleted with the following command:

**Delete Group <id range> [Propagate]**

The option **propagate** will also delete any contained groups, recursively. That is, if group A contains group B and group B contains group C, then **Delete Group A Propagate** will delete groups A, B, and C.

*Cleaning Out Groups*

You can remove all of the entities in a group via the **cleanout** command:

**Group <group_id_range> Cleanout [Geometry|Mesh] [Propagate]**

By default all entities will be removed - optionally you can cleanout just geometry or mesh entities. As in delete, the **propagate** option will cleanout the group specified and all of its contained groups recursively.

## Groups in Graphics

In the GUI version of CUBIT, groups may be picked with the mouse.
When displaying a group containing hexes, only the outside skin of the hexes will be displayed.

## Propagated Hex Groups

- Starting on a Surface

- Starting on a Face

Propagated hex groups are a way of grouping hexes from a hex mesh using sweep-type criteria. For example, creating a group containing all hexes between two specified mesh faces.
Note: the first examples below are based on first executing these commands:

```
brick width 10
volume 1 size 1
mesh volume 1
```

*Propagated Hex Group Starting on a Surface*

Starting on a surface can end at a surface or can end after the number of times the user specifies.

- Ending at a Surface
- Number of Times
- Ending at a Surface with Multiple
- Number of Times with Multiple
- Ending at a Surface with Direction
- Number of Times with Direction

#### Ending at a Surface

```
Group ['name' | <id>] Add Hex Propagate Surface <id> Target Surface <id>
```

**Example**

```
group 2 add hex propagate surface 1 target surface 2
```

**Result**: Group 2 will be created containing 1000 hexes

#### Number of Times

```
Group ['name' | <id>] Add Hex Propagate Surface <id> Times <number>
```

**Example**

```
group 2 add hex propagate surface 1 times 4
```

**Result**: Group 2 will be created containing 400 hexes
Both methods, ending at surface or number of times, can be used with the "multiple" option which will create several groups depending upon the multiple number specified.

#### Ending at a Surface with Multiple

**Group ['name' | <id>] Add Hex Propagate Surface <id> Target Surface <id> Multiple <number>**

**Example**

**group 2 add hex propagate surface 1 target surface 2 multiple 2**

**Result:** Five groups will be created and stored with their respective ids of multiple 2, these groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

#### Number of Times with Multiple

**Group ['name' | <id>] Add Hex Propagate Surface <id> Times <number> Multiple <number>**

**Example**

**group 2 add hex propagate surface 1 times 10 multiple 5**

**Result**: Two groups will be created and stored with their respective ids of multiple 5, these two groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

If number of times is specified and the direction is ambiguous, the surface direction or the node direction can be specified to direct the propagation. If the end surface is specified, only a node direction can be specified to direct the propagation. When specifying the node direction, the node has to be picked such that when the hexes are propagated, the picked node lies in these propagated hexes. If that node is never reached while propagating, the direction is not found and zero hexes will be included in the specified group.
**Note:** for the examples below, the result can be seen by executing these commands:

**brick x 10**
**vol 1 size 1**
**brick width 10**
**body 2 move 10**
**volume all size 1**
**merge all**
**mesh volume all**

#### Ending at Surface with Direction

**Group ['name' | <id>] Add Hex Propagate Surface <id> Times <number> Direction Node <id>**

**Example**

**group 2 add hex propagate surface 6 target surface 12 direction node 1530**

**Result**: Group 2 will be created containing 400 hexes
**Note**: The direction command and the multiple command can be combined (i.e. group 2 add propagate surface 6 times 4 multiple 2 direction node 1530)

#### Number of Times with Direction

**Group ['name' | <id>] Add Hex Propagate Surface <id> Times <number> Direction [surface <id> | node <id>]**

**Example**

**group 2 add hex propagate surface 6 times 4 direction surface 4**

**group 2 add hex propagate surface 6 times 4 direction node 1530**

**Result**: group 2 will be created containing 400 hexes

*Propagated Hex Group Starting on a Face*

When starting on a face, the propagation method can end at a surface, end at a face or can end after the number of times the user specifies:

- Ending at a Surface
- Ending at a Face
- Number of Times
- Ending at a Surface with Multiple
- Ending at a Face with Multiple
- Number of Times with Multiple
- Ending at a Face with Direction
- Ending at a Surface with Direction
- Number of Times with Direction

#### Ending at a Surface

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id range> Target Surface <id>**

**Example**

**group 2 add hex propagate face 1 11 21 target surface 2**

**Result**: Group 2 will be created containing 30 propagated hexes (10 layers of 3 hexes)

#### Ending at a Face

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id> Target Face <id>**

**Example**

**group 2 add hex propagate face 1 target face 1721**

**Result**: Group 2 will be created containing 5 propagated hexes (5 layers of 1 hex)
**Note**: Ending at a face requires starting at one face at one time, but ending at surface allows multiple start faces

#### Number of Times

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id range> Times <number>**

**Example**

**group 2 add hex propagate face 2 times 4**

**Result**: Group 2 will be created containing 4 propagated hexes (4 layers of 1 hex)

All of these methods, ending at surface, end at a face or number of times, can be used with the "multiple" option which will create a grandparent (top-level), parent (mid-level, contained within the grandparent) and child (bottom level, contained within the parent) groups. The child groups will contain each hex layer (specified number of layers per child group), all organized into a single parent group, which is organized underneath the group ID given to the command. Subsequent propagation commands could then be executed adding to the grandparent group, but creating a new parent and child groups. This way multiple propagation "sets" can be stored in one grandparent group, if desired.

#### Ending at a Surface with Multiple

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id> Target Surface <id> Multiple <number>**

**Example**

**group 2 add hex propagate face 1 target surface 2 multiple 1**

**Result:** Ten groups will be created and stored with their respective ids, one for each layer of hexes. These groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2. A subsequent propagation command could be executed adding to group 2 (the grandparent), which would create a single group contained in group 2 (the parent), containing the hex layer groups (the children).

#### Ending at a Face with Multiple

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id> Target Surface <id> Multiple <number>**

**Example**

**group 2 add hex propagate face 1 target face 1721 multiple 1**

**Result**: 5 groups will be created and stored with their respective ids, one for each layer of hexes. These groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2. A subsequent propagation command could be executed adding to group 2 (the grandparent), which would create a single group contained in group 2 (the parent), containing the hex layer groups (the children).

#### Number of Times with Multiple

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id> Times <number> Multiple <number>**

**Example**

**group 2 add hex propagate face 1 times 10 multiple**

**Result**: Two groups will be created and stored with their respective ids, these two groups will be stored in the parent group, Group 3, and Group 3 will be stored in the grand parent group, Group 2.

If the end surface or end face is ambiguous, a node direction can be specified to direct the propagation. When specify the node direction, the node has to be picked such that when the hexes are propagated, the picked node lies in these propagated hexes. If that node is never reached while propagating, the direction is not found and zero hexes will be included in the specified group.

#### Ending at Face with Direction

**Group ['name' | <id>] Add Hex Propagate [source] Face <id> Target Face <id> Direction Node <id>**

**Example**

**group 2 add hex propagate face 1721 target face 1 direction node334**

**Result**: group 2 will be created containing 6 hexes

#### Ending at Surface with Direction

**Group ['name' | <id>] Add Hex Propagate [Source] Face <id range> Target Surface <id> Direction Node <id>**

**Example**

```
group 2 add hex propagate face 1 target surface 2 direction node 334
```

**Result**: group 2 will be created containing 10 hexes
**Note**: The direction command and the multiple command can be used together (i.e. group 2 add propagate face 1721 end face 1 multiple 2 direction node 334)

If number of times is specified and the direction is ambiguous, a surface direction or a node direction can be specified to direct the propagation. The node direction has the same condition as when ending at a surface or face and that is it must lie in the propagated hexes.

#### Number of Times with Direction

```
Group ['name' | <id>] Add Hex Propagate [Source] Face <id> Times <number>Direction [surface <id> | node <id>]
```

**Example**

```
group 2 add hex propagate face 110 times 4 direction surface 2

group 2 add hex propagate face 1 times 4 direction node 269
```

**Result**: group 2 will be created contained 4 hexes
**Note**: The direction command and the multiple command can be used together. (i.e. group 2 add propagate face 1721 times 4 multiple 2 direction surface 1)

# *Naming Convention for Propagated Hex Groups*

A special naming convention can be used for the propagated hex groups, best described by an example.
The following command will create a hierarchy of logically named groups, as follows.

```
group 'W1P1T1' add propagate surf 1 end surf 2 multiple 1
```

The hierarchy looks like this:

```
W1
 W1P1
  W1P1T1
  W1P1T2
  W1P1T3
  ...
```

**W1P1T10**

Where W1P1 is contained within W1, and W1P1T1, W1P1T2, etc.. are contained within W1P1.
The software simply looks for numerical numbers in the group name and parses out the correct grandparent, parent and child names from the substrings. There must be exactly 3 substrings in the group name, each ending with an integer for the command to work properly.
A subsequent command:

**group 'W1P2T1' add propagate surf 3 end surf 5 multiple 1**

will add a parent group to W1, called W1P2, and the subsequent child groups:

**W1**
**W1P1**
**W1P1T1**
**W1P1T2**
**W1P1T3**
**...**
**W1P1T10**
**W1P2**
**W1P2T1**
**W1P2T2**
**W1P2T3**
**...**
**W1P2T10**

## Quality Groups

Groups can also be formed from the hexes or faces obtained from the quality command. Each group formed using quality can be drawn with its associated quality characteristics {i.e. jacobian low .2 high .3} automatically.

**Group {<'name'>|id} {Add|Equals|Remove|Xor} Quality { Hex | Tet | Face | Tri | Volume | Surface | Group } <id_range> { quality metric name (default is SHAPE) } [ High <value> ] [ Low <value> ] [ Top <number> ] [ Bottom <number>]**

The following example illustrates the use of quality groups:

**group 2 add quality volume 1 jacobian**

In this case, if the meshed brick from the section Propagated Hex Groups is used, Group 2 will be created and it will contain 1000 hexes with quality characteristics.
The quality metric names can be found in the Quality Assessment section of the documentation.

## Propagated Groups

Creating propagated groups is a mechanism for joining groups of elements that meet specific criteria. For hex groups it might be grouping hexes from a hex mesh using sweep-type criteria. For surface elements, it might be grouping faces or tris into sidesets based on angle criteria.

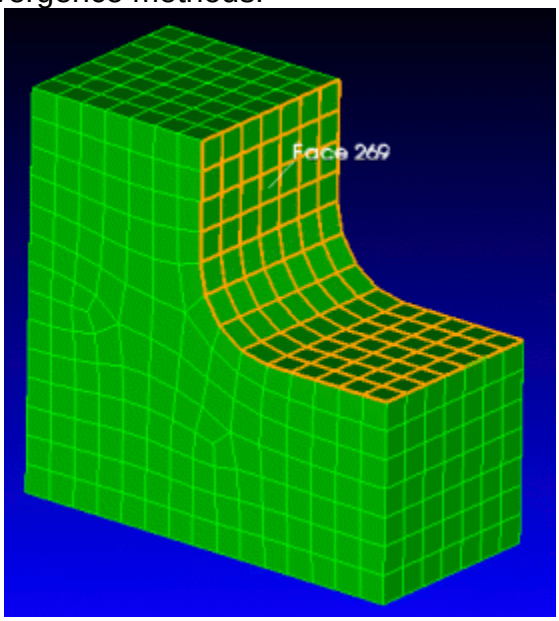
- Propagated Hex Groups
- Propagated Surface Groups using the Seed Method

## Seeded Mesh Groups

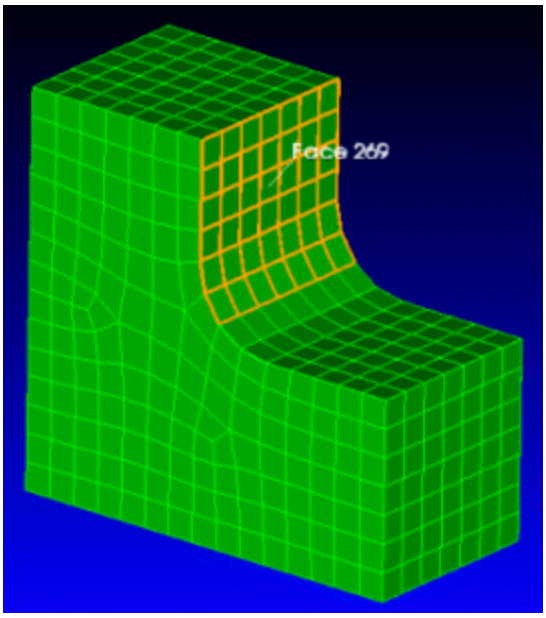
It is also possible to automatically group surface mesh elements based on feature angles. Given a seed element, the algorithm will loop over all adjacent elements and create groups of elements whose surface normals are similar, or which fall within a certain radius. The command syntax is:

**Group {<'name'>|<id>} {Add|Equals|Remove|Xor} Seed <mesh_entities> {Feature_angle <angle> [Divergence]|Depth <number>}**

The seed element may be a quad, tri, or node element. There are two methods of angle comparison for this command. The feature angle option will compare angles of the each element to its adjacent elements by comparing surface normals. In the case of nodes, the seed node surface normal will be the average of the adjacent faces or tris. Nodes will be added if their attached faces meet the angle requirements. The divergence option will compare angles to the original seed element's surface normal. The depth option will add elements within a certain radius.
The following figures illustrate the use of the seed method to create mesh groups using the feature angle and divergence methods.

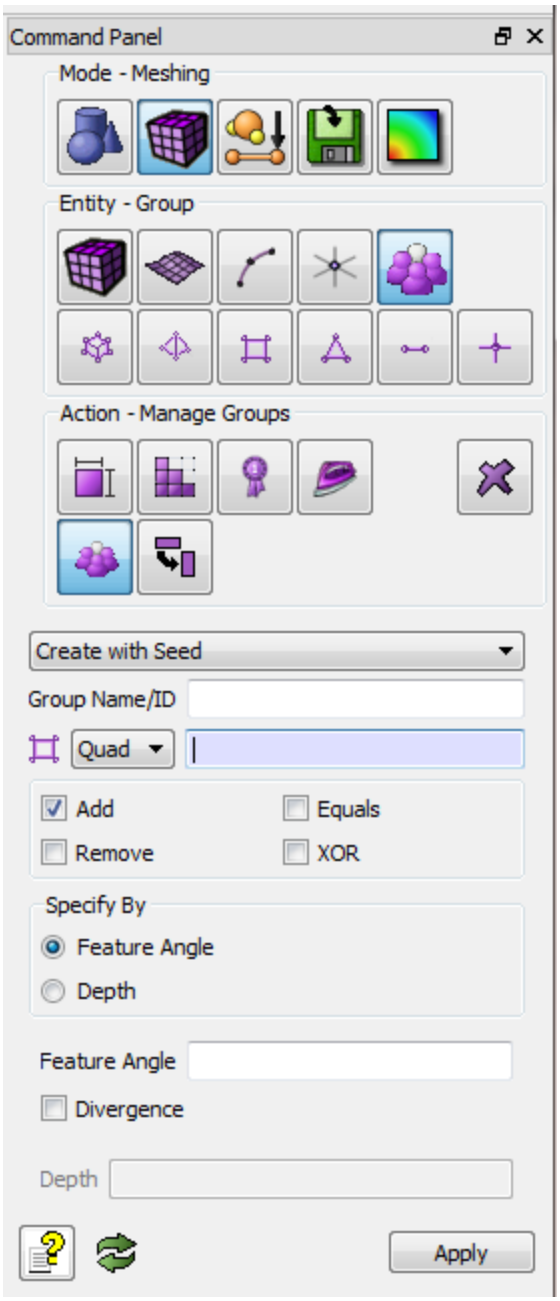



**CUBIT> group 'mygroup1' add seed face 269 feature_angle 45**




**CUBIT> group 'mygroup2' add seed face 269 feature_angle 45 divergence**

The seed method of creating groups is particularly useful for creating groups on free meshes for the purpose of assigning nodesets and sidesets.
The GUI command panel for this command is found by selecting
"Mode-Meshing", "Entity-Group", "Action-Manage Groups", then "Create with Seed." The command panel is shown below:




## Naming Convention for Propagated Hex Groups

A special naming convention can be used for the propagated groups, best described by an example.
The following command will create a hierarchy of logically named groups, as follows.

```
group 'W1P1T1' add propagate surf 1 end surf 2 multiple 1
```

The hierarchy looks like this:

```
W1
 W1P1
```

**W1P1T1**
**W1P1T2**
**W1P1T3**
**...**
**W1P1T10**

Where W1P1 is contained within W1, and W1P1T1, W1P1T2, etc.. are contained within W1P1.
The software simply looks for numerical numbers in the group name and parses out the correct grandparent, parent and child names from the substrings. There must be exactly 3 substrings in the group name, each ending with an integer for the command to work properly.
A subsequent command:

**group 'W1P2T1' add propagate surf 3 end surf 5 multiple 1**

will add a parent group to W1, called W1P2, and the subsequent child groups:

**W1**
**W1P1**
**W1P1T1**
**W1P1T2**
**W1P1T3**
**...**
**W1P1T10**
**W1P2**
**W1P2T1**
**W1P2T2**
**W1P2T3**
**...**
**W1P2T10**

# attributes

## Geometry Attributes

Each geometric topological entity has specific information attached to it. These attributes specify aspects of the entity such as the color that entity is drawn in and the meshing scheme to be used when meshing that entity. This section describes those geometry attributes that are not described elsewhere in this manual.

- Entity Names
- Entity IDs
- Persistent Attributes

persistent attributes

# Attribute Behavior

In this context, attributes are defined as data associated directly with a particular geometry entity. In CUBIT's implementation of attributes, these data can occupy one of three "states" at any given time: they can be stored in data fields on CUBIT's geometry entities; they can be stored in an intermediate representation, using CUBIT's attribute objects; or they can exist only on the ACIS objects. When they are stored on ACIS objects, those attributes are written to and read from disk files with the geometry. This mechanism allows CUBIT-specific information to be stored and retrieved with the geometry data. By default, attribute data is not stored with geometry. To enable the use of attributes, use the commands described in the following sections.

# Attribute Commands

Most non-CUBIT-developer uses of attributes will be to use all or none of the attributes. Therefore, the most common command to enable and disable the use of attributes is:

**Set Attribute {On|Off}**

When this option is on, all defined attributes will be saved with the geometry when the user enters the Export Acis command.
When a geometry is imported into CUBIT, any attributes defined on that geometry and recognized as CUBIT attributes are imported and put into an intermediate representation (that is, this information is not assigned directly to the geometry entities). To find out which attributes are defined on a given set of entities, use the following command:

**List [<entity_list>] Attributes [Type <attribute type>] [All] [Print]**

If no entities are entered, attribute information for all the geometric entities defined in CUBIT is printed.
The **Type** option can be used to list information about a specific attribute type; values for are the same as those in the previous table.
If the **All** option is entered, information about all attribute types will be printed, even if there are none of those attributes defined for the specified entities.
If the **Print** option is entered, the information stored in each attribute will be printed; this command is usually used only by CUBIT developers.

#### Control By Attribute Type or Geometric Entity

Attributes can be enabled or disabled by attribute type, to allow the use of only user-specified attribute types. To turn on or off specific attributes, use the command:

**Set Attribute <attribute type> {On|Off}**

where **<attribute type>** is one of the types shown in the previous table.
Attributes can also be controlled to automatically write (update) and read (actuate) to/from solid model files automatically, using the command:

**Set Attribute <attribute_type> Auto {Actuate|Update} {On|Off}**

Finally, attributes can be manually written to and read from the geometric entities, and removed from cubit entities, using the command

**{geom_list} Attribute {All|Attribute_type}**
**{Actuate|Remove|Update|Read|Write}**

where **geom_list** is a list of geometry entities. This command is recommended only for developers' use.

# *Attribute Types*

The attribute types currently implemented in CUBIT are shown below.

| **Attribute Types** | **Description** |
|---|---|
| Color | Entity Color |
| Composite vg | Used to restore composite virtual topology |
| Genesis entity | Membership in boundary conditions (block, sideset, nodeset) |
| Id | Entity Id |
| Mesh container | Handle to mesh defined for the owner |
| Mesh scheme | Meshing scheme (e.g. paving, sweeping, etc.) |
| Name | Entity name |
| Partition vg | Used to restore partition virtual topology |
| Smooth scheme | Smoothing scheme (e.g. Laplacian, Condition Number) |
| Unique Id | Unique entity id, used to cross-reference other entities |
| Vertex type | Used to define mesh topology at vertex for mapping/submapping |
| Virtual vg | Used to store virtual geometry entity(ies) defined on an entity |

# Persistent Attributes

Typical data assigned to topological entities during a meshing session might include intervals, mesh schemes, group assignments, etc. By default, most of this data is lost between CUBIT sessions, and must be restored using the original CUBIT commands. Using CUBIT's persistent attributes capability, some of this data can be saved with the solid model and restored automatically when the model is imported into CUBIT.


- Attribute Behavior
- Attribute Types
- Attribute Commands
- Using CUBIT Attributes


# Using CUBIT Attributes

A typical scenario for using CUBIT attributes would be as follows.
Construct geometry, merge, assign intervals, groups, etc. (i.e. normal CUBIT session)
Enable automatic use of attributes using the command:

**Set Attribute On**

Export acis file (see Export Acis command).
Subsequent runs:
Enable automatic reading and actuating of attributes:

**set attribute on**

Import ACIS file (see Import Acis command)
Used in this manner, geometry attributes allow the user to store some data directly with the geometry, and have that data be assigned to the corresponding CUBIT objects without entering any additional commands.

## Entity IDs

Topological entities (including groups) are assigned integer identification numbers or ids in CUBIT in ascending order, starting with 1 (one). Each new entity created within CUBIT receives a unique id within the topological entity type. This id can be used for specifying the entity in CUBIT commands, for example "draw volume 3".
There is a separate id space for each type of topological entity. For example, all mesh nodes are given ids from 1 to *n*, where *n* is an integer greater than or equal to the number of nodes in the model. Likewise, all hexahedra are given ids from 1 to *m*, where *m* is an integer greater than or equal to the number of hexahedra in the model.

*Element Ids*

Each mesh entity (hex, tet, face, tri, edge, node, etc.) may also have a *Global Element ID* from an id space which is used for all mesh entities. A mesh entity is only assigned a *Global Element ID* if it is in a block, and is the global id that will be assigned to the element during Exodus export. The *Global Element ID* provides a single id space across all the different element types.

*Gaps in ID space*

After working with a model for some time, various operations will cause gaps to be left in the numbering of the geometric & mesh entities. The **compress ids** commands can be used to eliminate these gaps:

**Compress [ids] [all] [Retainmax] [Sort]**

**Compress [Ids] [All]**
**{Group|Body|Volume|Surface|Curve|Vertex|Element|Hex|Tet|Face|Edge|Node}**
**[Retainmax]**

Typing **compress** with no options or **compress all** will compress the ids of all entities; otherwise, the entity type for which ids should be compressed can be specified. The **retainmax** argument will retain the maximum id for each entity type, so that entities created subsequent to this command will receive ids greater than that value. If the **sort** qualifier is included, the new id of each entity will be determined by its size and location. Small entities are given a lower id than large entities. Entities that are the same size are sorted by their location, with lower x coordinate, then y, then z leading to a lower id. For example, two vertices are always the same size, so they are sorted based on the lowest x coordinate. If they are equal, then lowest y coordinate, etc. If two entities are found to have the same size and location, they are sorted according to their previous ids. This option can be used to restore ids in translated models in a manner which leads to more persistence than purely random id assignment.

*Renumbering IDs*

The renumber command can be used to change the id numbers assigned to meshed entities.

**Renumber {Node|Edge|Tri|Face|Hex|Tet|Wedge|Element} <id_range> Start_id <id> [Uniqueids]**

Any valid range specification can be used to specify the source ids. There is no requirement that the ids being renumbered are consecutively numbered. The new id numbers will be consecutive beginning at the specified start id. For the command to be successful there can be no existing ids within the effective range of the start id. If the resultant destination range is not free of id numbers, the command will fail with an appropriate error.

Using the **uniqueids** keyword will result in the elements to be renumbered such that no element shares the same ID.
For convenience, all elements and nodes in a block can be renumbered with a single command:

**Renumber block <id_range> [node_start_id <id>] [elem_start_id <id>] [localids]**

By default, the *Global Element ID* is renumbered with the **renumber block** command. If **localids** is specified, the hex, tet, face, tri, or edge id is renumbered instead.

*Volume ID*

The volume id command is used to renumber a single volume.

**Volume <old_id> Id <new_id>**

This command replaces the volume's old_id with the new_id if no other is using the new_id number. Entity renaming only works for volumes; it does not work for nodes, curves or surfaces.

## Entity Names

By default, geometric entities in CUBIT are referenced using an entity type (e.g. Surface, Volume) and an id, for example "draw surface 1". However, geometric entities can also be assigned names, to simplify working with specific entities. Once a name is assigned to an entity, that name can be used in any CUBIT command in place of the entity type and number. For example, if surface 1 were named 'mysurf1', the command above would be equivalent to "draw mysurf1". Also, since entity names are saved with the geometry, this also provides a means for persistent identifiers for geometric entities. Names can be added or removed using the following commands.

**{Group|Body|Volume|Surface|Curve|Vertex} {Name | Rename} {`<entity_name>'| Default}**

**{Group|Body|Volume|Surface|Curve|Vertex} Remove Name {`<entity_name>'| All | Default}**

The name of each topological entity appears in the output of the List command. In addition, topological entities can be labeled with their names (see label command). A list of all names currently assigned and their corresponding entity type and id (optionally filtered by entity type) can be obtained with the command

**List Names [{Group|Body|Volume|Surface|Curve|Vertex|All}]**

**Notes:**

- In a merge operation, the surviving entity is given the name(s) of the deleted entity.
- A geometric entity may have multiple names, but a particular name may only refer to a single entity.

*Case-Insensitive Names*

Entity names in CUBIT are case-insensitive. This means that there is no difference between the name 'mysurf1' and MySurf1'. The case of names will be preserved as assigned. For example, a volume named 'MyVolume' will display the name 'MyVolume' but can be referenced by any case version of the name, 'MYVOLUME', 'mYvOlume', etc. Case-sensitivity can be toggled on/off with the command:

**[Set] Case Sensitive Names [on|OFF]**

*Valid and Invalid Names*

Although any string may be used as an entity name, only valid names may be used directly in commands. A name is valid if it begins with a letter or underscore ("_"), followed by any combination of zero or more letters, digits, or the characters ".", "_", or "@". If an attempt is made to assign an invalid name to an entity, CUBIT will generate a valid version of the invalid name by replacing invalid characters with an underscore. Then both the valid and invalid versions of the name are assigned to the entity. For example, assigning the name "123#" to a volume will result in the volume having two names, "123#" and "_23_". The valid name can be used directly in commands (mesh _23_), while the invalid name can only be referenced using a longer, less direct syntax (mesh volume with name "123#").

*Reconciling Duplicate Names*

When an attempt is made to assign the same name to two different entities, a suffix is added to the name of the second entity to make it unique. The suffix consists of the "@" character followed by one or more letters or numbers. For example, the following commands will result in volumes 1 to 3 having the names "hinge", "hinge@A", and "hinge@B", respectively:

```
volume 1 name "hinge"
volume 2 name "hinge"
volume 3 name "hinge"
```

To prevent this automatic "fixing" of names, the *Fix Duplicate Names* flag may be switched to off. If the user attempts to assign a duplicate name while the flag is set to off, the name will remain unchanged.

```
Set Fix Duplicate Names [ON|Off]
```

*Automatic Name Creation*

CUBIT provides an option for automatically assigning names to entities upon entity creation. This option is controlled with the command:

**Set Default Names {On|OFF}**

When this option is on, entities are assigned default names consisting of a geometry type concatenated with the entity id, for example 'cur1', 'surf26', or 'vol62'.

*Automatic Name Propagation*

CUBIT automatically propagates names through webcuts. If an entity that has been assigned the name "Gear" is split through webcuts, the resulting bodies are named "Gear" and "Gear@A". Try the following example.

```
br                                    x                                    10
volume                  1                    name                    "Cube"
webcut                  volume                  1                    xplane
webcut              volume                  1                  2          yplane
webcut          volume          1          2          3          4          zplane
label volume name
```

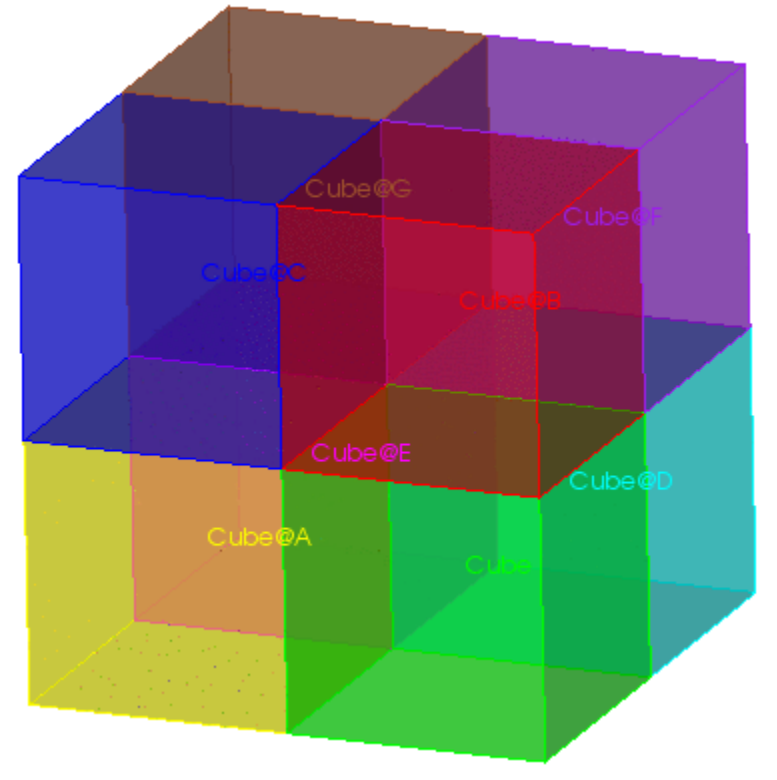



**Figure 1. Name Propagation through Webcuts**

You can operate on these propagated names using wildcards such as:

```
mesh volume with name 'Cube*'
block 1 volume with name 'Cube*'
```

*Naming Merged Entities*

When entities that have the same base name, such as "platform" and "platform@A", are merged, the resulting entities is assigned both names. The **set merge base names on** command tells Cubit that in this situation, it should merge the names too. The command syntax is:

```
Set Merge Base Names [On|OFF]
```

For example:

```
brick x 10
vol 1 copy move 10
surf 6 name 'platform'
surf 10 name 'platform'
```

Surface 10 actually is named platform@A, since we don't want duplicate names

```
merge all
list surf 6
```

You see that surface 6 has both 'platform' and 'platform@A' as names. Now, for the contrasting example

```
brick x 10
vol 1 copy move 10
surf 6 name 'platform'
surf 10 name 'platform'
set merge base names on
merge all
list surf 6
```

You see that surface 6 has only 'platform' as its name.

# metadata

## Parts, Assemblies, and Metadata

*Overview of Parts, Assemblies and Metadata*

A geometric model may be organized into a hierarchy of assemblies, sub-assemblies, and parts. These parts and assemblies can be assigned certain attribute values. The parts, assemblies, and associated attributes are referred to as DART Metadata, or simply metadata. Metadata can be imported from files, or can be created within CUBIT. Metadata can be exported to both mesh and geometry files.
Although useful in its own right, the primary purpose of CUBIT's metadata capabilities is to enable interoperability with the set of applications participating in the DART project (see the Sandia Analysis Workbench Wiki page at https://dart.sandia.gov/wiki/display/SAW/DartMetadataPlugin). DART interoperability

enables CUBIT to preserve assembly relationships and material data through the analysis process.
This section describes the procedures for importing, manipulating and exporting metadata within CUBIT.

- Working with Parts and Assemblies
- Metadata Attributes
- Importing and Exporting Metadata

## Importing and Exporting Metadata

Metadata can be imported from and exported to a file. In most cases metadata will be imported and exported with a data file such as a SAT file or a genesis file. CUBIT is also compatible with DART artifacts, including artifact dependency tracking.

- Importing Metadata
- Exporting Metadata
- Importing and Exporting DART Artifacts

*Importing Metadata*

Parts and assemblies can be created and associated with geometry by importing a DART Metadata file along with a geometry file, using the XML option of the import command. At this time the only two geometry formats which support metadata import are STEP and ACIS:

**Import {Step|Acis} "<filename>". . . [XML "<xml_filename>"]**

To successfully associate the contents of the geometry file with the parts described in the metadata, the XML file must follow the DART Metadata 3.0 XML schema found at https://dart.sandia.gov/wiki/display/SAW/DartMetadataPlugin, and the geometry file must contain extra DART data. A suitable STEP file and a corresponding metadata file can be exported from Pro/E using the Pro/E (Creo) extension, see the Metadata plugin page for details. A SAT file and corresponding metadata file can be obtained by exporting them from CUBIT using the XML option of the export command.

*Exporting Metadata*

Some export commands include an XML option. Including this option in the export command instructs CUBIT to write out a DART metadata file, in addition to the traditional data file. The metadata file includes the data required to enable interoperability with other DART-compliant applications.
The only geometry export command which supports the XML option is ACIS export:

**Export Acis “<acis_filename>” [XML “<xml_filename>”]**

When an ACIS file exported with metadata, the specified XML file includes a description of the assembly hierarchy as it appears in CUBIT.
Metadata can also be written to an XML file when exporting mesh. The only mesh export command which supports the XML option is genesis export:

**Export {Genesis|Mesh} "<mesh_filename>" [XML '<xml_filename>']**

The XML file generated during mesh export includes the same information in a geometry metadata file, but also includes mesh-related data such as mappings between parts and element blocks, and includes any block, nodeset, or sideset names or descriptions which have been defined.

*Importing and Exporting DART Artifacts*

The DART project has defined a specific way to package data files with corresponding metadata files. A correctly packaged set of data files with a corresponding metadata file is called an *artifact*. An artifact's metadata file is always located in the same directory as the primary data file, and is always named *artifact.dta*.
Within the DART environment, dependencies between artifacts may be tracked by placing tracking information into metadata files. CUBIT supports automated artifact dependency tracking. Tracking information in an input metadata file is automatically reflected in any output metadata file written by CUBIT.
If input is correctly packaged as an artifact, CUBIT can automatically locate and read the metadata file corresponding to a particular input data file. To have CUBIT do this, select the "Import as Artifact" checkbox in the Open File dialog.
CUBIT can also package output as an artifact. To do so, select the "Export as Artifact" checkbox in the export dialog box.
When importing or exporting artifacts using the command line, include the *XML* option in the import or export command, specifying the xml file called *artifact.dta* in the same directory as the main data file.
For dependency tracking purposes, it may be necessary to import an artifact's metadata file by itself. For example, it may be necessary to import an artifact consisting of an IGES file. Since the *Import IGES* command does not support the *XML* option, the metadata file must be imported separately. To do so, use the command:

**Import XML "<xml_filename>"**

When working with correctly packaged artifacts, the XML filename will always be artifact.dta.

## Metadata Attributes

Each part and assembly has several attributes, including its name and description. In addition, there are several attributes which do not describe any particular part or assembly. The "global" attributes describe the assembly tree as a whole, or the metadata as a whole.
These sections describe how to view and edit metadata attributes.

- Part and Assembly Metadata Attributes
- Viewing Part and Assembly Metadata Attributes
- Modifying Part and Assembly Metadata Attributes
- Viewing and Modifying Global Metadata Attributes


*Part and Assembly Metadata Attributes*

Each part and assembly has several attributes. Some attributes apply to both parts and assemblies, while other attributes apply to only parts. The attributes are listed in the following table:

| **Attribute Name** | **Attribute Description** | **Applies To:** | |
|---|---|---|---|
| | | **Part** | **Assembly** |
| Name | Name of Part or Assembly | **x** | **x** |
| Description | Description of Part or Assembly | **x** | **x** |
| Instance | Instance Number | **x** | **x** |
| File | The name of the file containing the original version of this entity. Often a reference to a PDM system. | **x** | **x** |
| Units | The unit system of this part or assembly. | **x** | **x** |
| Material_Description | The name or description of the material of which this part is composed. | **x** | |
| Material_Specification | The formal specification number of the material of which this part is composed. | **x** | |
| Density | The density of the material of which this part is composed. Setting it to a non-positive value will clear the attribute, as if there were no value assigned. | **x** | |
| Material_Volume | The volume of the region enclosed by this part. The material_volume is not calculated from the volumes associated with the part. It will often differ from the actual volume enclosed by this part's associated geometric | **x** | |

| | | | |
|---|---|---|---|
| | volumes, and can also be manually set to any non-negative value. Setting it to a non-positive value will clear the attribute, as if there were no value assigned. | | |
| Elemental_Composition | A string value describing the composition of the material, typically expressed as percentages of given elements. | **x** | |

*Viewing Part and Assembly Metadata Attribute Values*

The easiest way to view a part or assembly's metadata attribute values is to select the item in the entity tree. The item's metadata attributes are listed in the property page.
A part or assembly's metadata attribute values can also be viewed using the **Metadata List** command:

**Metadata List [<attribute_name>] {Part|Assembly} "<path>"**

The attribute_name should be one of the attribute names in the table above. If no attribute name is included in the command, all metadata attributes are listed.
Metadata attributes can also be listed based on a volume.

**Metadata List [<attribute_name>] Volume <id>**

This volume-based command works just like the part-based command, but lists the metadata for the part with which the volume is associated.

*Modifying Metadata Attributes*

A part or assembly's metadata attributes can be modified in the property page. Simply select the part or assembly in the entity tree, then click in the appropriate text field in the property page.
A part or assembly's metadata attributes can also be modified using the **Metadata Modify** command:

**Metadata Modify <attribute> "new value" {Part|Assembly} "<path>"**

where **attribute** is one of the attributes listed in the table above. The specified attribute value will be changed to **new_value**.
There is also a volume-based version of the **Metadata Modify** command:

**Metadata Modify <attribute> "new_value" Volume <id>**

The volume-based command works just like the part-based command, operating on the part with which the volume is associated. Note that if the specified volume is not associated with a part, a new part will be created and associated with the volume.

*Viewing and Modifying Global Metadata*

There are several attributes which do not describe any particular part or assembly. These "global" attributes describe the metadata as a whole:

| **Attribute Name** | **Description** |
|---|---|
| Classification_Level | The level of sensitivity of the metadata. Usually one of the following:<br>• Secret<br>• Confidential<br>• Unclassified |
| Classification_Category | The classification category. Usually one of the following:<br>• Not Restricted<br>• Restricted Data (RD)<br>• Formerly Restricted Data (FRD)<br>• National Security Information (NSI) |
| Weapon_Category | Sigma 1 through Sigma 15 |

Global metadata values can be viewed using the **Metadata List** command:

**Metadata List <attribute_name>**

Global metadata values can be modified using the **Metadata Modify** command:

**Metadata Modify <attribute_name> "new_value"**

For both commands, **attribute_name** should be one of the attribute names in the table above.

## Working With Parts and Assemblies

Volumes can be organized into a hierarchical tree of parts, assemblies, and sub-assemblies. Assemblies may contain parts and other assemblies. Parts, on the other hand, may not contain sub-entities.

Each part and assembly has a name and an optional description. Other attributes may also be assigned, such as a material specification or a link to an entry in a PDM system. See Metadata Attributes.

The relationship between the geometric model and the assembly is determined by associating parts with volumes. A single part can be associated with any number of

volumes, including zero volumes. A volume, however, can be associated with only one part.
As volumes are modified, CUBIT automatically maintains the appropriate relationships with parts. If a volume is associated with a part, and that one volume is split into multiple volumes through a webcut or some other operation, each of the resulting volumes is automatically associated with the original volume's part. Copying a volume will also result in the new volume being associated with the same part as the original volume.


- Identifying Parts and Assemblies
- Creating Parts and Assemblies
- Deleting Parts and Assemblies
- Associating Parts with Volumes
- Viewing All Assembly Information at Once
- Assemblies Tool in Power Tools


*Identifying Parts and Assemblies*

A part or assembly is identified by its assembly path. An assembly path is much like a directory path in a file system. It consists of the name of each ancestor in the assembly tree, separated by a forward slash. For example, a part named "p1" contained within the top-level assembly "a1" would be identified by the path "/a1/p1". If the part "p2" is part of the assembly "a2", and "a2" is a sub-assembly of "a1", then "p2" has the path "/a1/a2/p2". More than one part or assembly may have the same name. To differentiate between parts or assemblies with the same name and path, each part also has an instance number. If two entities have the same name, they will not have the same instance number. For example, two parts named "p1" may be "p1 instance 1" and "p1 instance 2".
Instance numbers may be incorporated into assembly paths by placing the instance number in angled braces after a part or assembly name. For example, "p1 instance 3" is identified in a path as "p1<3>". Other examples of instance numbers in assembly paths include "/a1<1>/a2<1>/p1<3>" and "/a1/a2<1>/p1". Assembly paths are always allowed to incorporate instance numbers, but are only required to include as many instance numbers as it takes to avoid ambiguity. Note that some commands do accept ambiguous paths, selecting a random entity which matches the path.
Most commands which accept assembly paths also allow the path to be followed by an "instance" command option (for example, metadata list part "/a1/p1" instance 3). The instance option always refers to the instance number of the last item in the path (p1 in the example).

*Creating Parts and Assemblies*

Parts and assemblies can be created using the following commands:

**Metadata Create {Assembly|Part} "<absolute_path>" [Instance <instance>]**

If the **instance** option is not included, CUBIT will assign an appropriate instance number to the new entity. If the instance option IS included, an entity with the specified name and instance number must not already exist or the command will fail.
Note that the path must be absolute, identifying each ancestor of the new entity. Any ancestors of the new entity which do not already exist are automatically created.

*Deleting Parts and Assemblies*

To delete a part or an assembly, use the Metadata Remove command:

**Metadata Remove {Part “<path>” | Assembly “<path>" [propagate]}**

This will remove the specified part or assembly. If the **propagate** option is specified when removing an assembly, all contained parts and subassemblies will be removed automatically before the assembly itself is removed. Otherwise, assemblies will only be removed if they have no contents.
It is also possible to remove all parts and assemblies that have no association with geometric volumes in the model:

**Metadata Clean**

This can be extremely useful when importing geometry which has been simplified with metadata which has not been simplified. For example, eMatrix currently writes out the full assembly hierarchy even when exporting a simplified representation of the geometry.

*Associating Parts with Volumes*

The relationship between the geometric model and the assembly is determined by associations between parts and volumes. As stated previously, a part may be associated with any number of volumes, while a volume may be associated with only one part. The easiest way to associate a volume with a part is to use the entity tree in the user interface. Drag a volume in the tree onto a part in the tree, and the volume and part are now associated. Since a volume can only be associated with one part at a time, any previous association between that volume and a part is removed.
Part-to-volume associations can be created on the command line using the **Metadata Modify Path** command:

**Metadata Modify Path “<part_path>” Volume <ids>**

The specified volume or volumes will be associated with the part specified by part_path. Any volumes already associated with the specified part will retain their association with the part.
Associations can be removed using the **Metadata Remove** command:

**Metadata Remove Volume <ids>**

After the Metadata Remove command has been issued, the specified volumes are no longer associated with any part.
The set of volumes associated with a given part can be modified using the **Metadata Replace** command:

**Metadata Replace Part “<part_path>” Volume <ids>**

When the Metadata Replace command is issued, all associations the part may have had with any volumes are removed. New associations are then created with the specified volume or volumes.

*Viewing All Assembly Information at Once*

Once an assembly tree is created, all assemblies, parts, and part-to-volume associations can be viewed using the command:

**Metadata List Tree**

This will print the names of all parts and assemblies in the output window, along with the IDs of the volumes associated with each part.
It is also possible to view all parts, their properties, and their volume associations using a spreadsheet application such as Microsoft Excel. This is done by generating a file using the command:

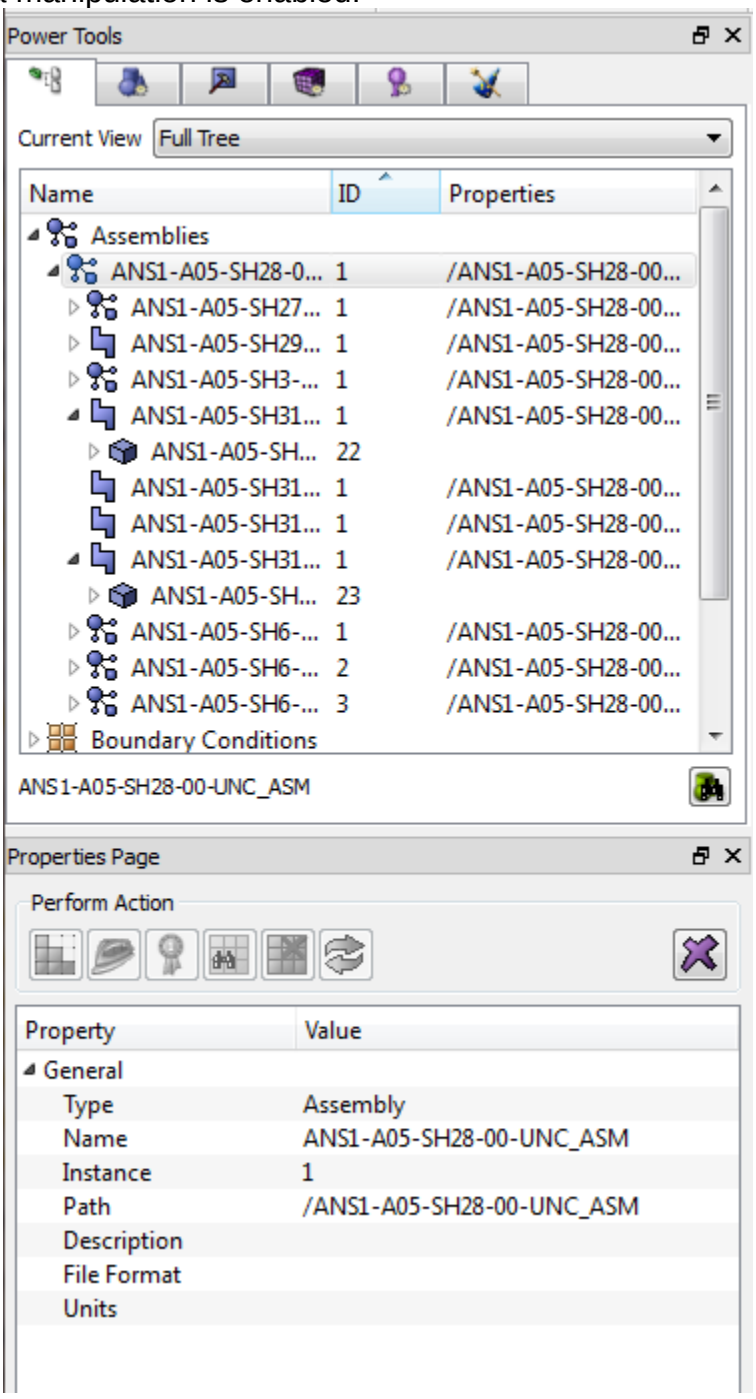
**Export Part_List "<filename>" [OverWrite]**

This command writes an XML file in a format that Excel can convert to a spreadsheet. To do this, simply import the XML file into Excel as an XML List. The data can then be sorted and filtered by any of the parts' properties.
The **Export Part_List** command is particularly useful for identifying parts which are not correctly associated with parts. Among the fields that can be filtered is the **is-part** field. This field is FALSE for each volume that is not associated with a part. Filtering on this value will show a list of all volumes that are not associated with any part. The **volume-ids** field will show the ID of each unassociated volume, and the **volume-name** field will show each unassociated volume's name, if any.
It is equally easy to identify parts that are not associated with volumes. Display only those rows with a blank value in the **volume-ids** field to see a list of parts that have no associated volume.
Similar methods can be used to identify missing materials information. Fields can also be sorted to group the parts by material.

*Metadata in the GUI*

Metadata may be displayed and manipulated in the GUI. The tree view includes a category for metadata. The category is labelled "Assemblies" in the tree view. Users are able to drag volumes into parts on the tree.  Also, selecting an Assembly or Part on the

tree will cause the attributes for the entity to be displayed in the property page where further data manipulation is enabled.




# import

## Importing Geometry

- Importing ACIS Models
- Importing FASTQ Models

- Importing STEP Files
- Importing IGES Files
- Importing SGM Files
- Importing Facet Files
- Other Formats

*Other Formats*

Internally, CUBIT represents geometry as either ACIS solid model geometry or mesh-based geometry. CUBIT can import ACIS geometry in the native "sat" file format. CUBIT can also import STEP and IGES files and internally converts them into ACIS solid model geometry. For compatibility with Sandia legacy applications, CUBIT can import FASTQ input decks to create ACIS geometry, as well. CUBIT also contains experimental support for using SGM for representing geometry.
If you have geometry that has been created in another format, such as in SolidWorks, you will need to translate that geometry into something that Cubit can read. Many solid modeling packages have an Export ACIS .sat command, which is probably the easiest way of translating your model. If you do not have that option, there are some other possibilities.

- Try a different file format, such as STEP or IGES.
- As a last resort, contact the Cubit team. They might have other options for importing your file.

See Also
**Importing a Mesh**

## Importing ACIS Files

The command used to read an ACIS file is:

**Import Acis '<acis_filename>' [No_bodies][No_surfaces] [No_curves][No_vertices][Group {'<name>'|<id>}] [Binary|Ascii] [Show_Each] [Sort] [XML '<xml_filename>'] [Attributes_On] [Separate_Bodies] [merge_gloabally] [Heal]**

The **import ACIS** command is the primary mechanism for generating geometry within CUBIT. ACIS parts can be generated and saved with CUBIT, but in most cases are developed within a 3rd party CAD package and exported for use in CUBIT. CUBIT provides the capability to import ACIS solid models and make modifications to them so they can be meshed. CUBIT incorporates the commercial ACIS libraries developed and maintained by Spatial Inc. for reading and writing ACIS format files. IGES and STEP format files can also be imported and exported to/from CUBIT using the Spatial's libraries.

*Import Options*

It is possible to include *free* entities (vertices, curves and surfaces) in the file. The default operation is to read all entities in the file whether they are included as part of a body or are free. By using any of the options **no_bodies, no_surfaces, no_curves,** or **no_vertices**, the user may exclude certain types of *free* entities.
The **group** option of the import command will allow the user to create a group for each set of imported geometry. The newly created group can later be accessed using the name or id specified with the group option.
The import capability of ACIS files supports both the ASCII format (.sat) and binary format (.sab). When importing, the filename extension will determine the default file type, be it ASCII or binary. A (.sat) extension will default to ASCII, while a (.sab) extension will default to binary. If you use a different file extension you can specify the type with the **[binary|ascii]** option. Binary files can be significantly faster but are not guaranteed to be upward compatible, nor cross-platform compatible. Therefore, it is recommended that models be archived in ASCII format.
Normally the numerical IDs of the geometric entities contained in the ACIS model are used directly within CUBIT. The sort option provides the capability to compress the IDs read from the ACIS file. The **sort** option does the same thing as the compress ids sort command, but combines it with the import command to remove a step in the process.
The **show_each** option is a graphics option that applies to how the volumes are shown as they are imported. If there are multiple volumes in the file, the graphics display will be updated between each volume during import.
The **xml** option will read assembly information and other metadata from an XML file in the DART metadata XML format. See the metadata documentation and the Analyst's Home Page for details.
The **attributes_on** option will enable attribute support for the file. Attributes include properties like entity color, entity id, and meshing scheme. Including the attributes option will only affect the current import. The settings will be restored to their previous settings after importing.
To retain any possible merge information when importing an ACIS file use the **attribute_on** command.
The **separate_each** option creates a separate body for each volume that is imported, preventing multi-volume bodies from being imported.
When importing, the use may specify the scope of the merge using **merge_globally**. The default behavior is to merge within the scope of the file being imported. With the **merge_globally** option, imported entities will merge with anything, including entities already in the Cubit session that have merge attributes on them.
Use the **heal** option to heal the entities when importing.

*Case-Insensitive Entity Names*

Entity names in Cubit are case-insensitive. This means that there is no difference between 'myvolume' and MyVolume'. When importing entity names as attributes any names that are case-sensitive duplicates will be made unique. This is done by appending one or more underscores, '_', followed by a number 1 to 9 or a letter A to Z. For example, two names 'myvolume' and 'MyVolume' encountered in that order will result in the names 'myvolume' and MyVolume_1'. Case-sensitivity can be toggled on/off with the command:

**[Set] Case Sensitive Names [on|OFF]**

*Importing ACIS files at startup*

ACIS files can also be imported using the **"-solid"** option when starting CUBIT from the UNIX command prompt. (See Execution Command Syntax for details.) Note that the filename must be enclosed in single or double quotes. This command will create as many bodies within CUBIT as there are bodies in the input file.
See also Exporting ACIS Files.

## Importing Facet Files

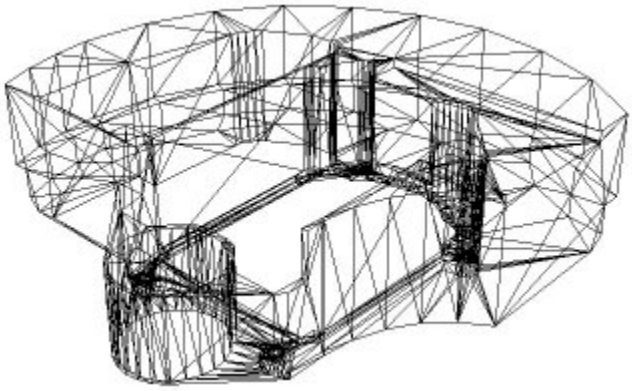
CUBIT provides the capability to import a model composed of facets to create geometry. The command to import facets from a file is:

**Import [Facets|AVS] "<filename>" [Feature_Angle <angle>] [LINEAR||Spline] [MERGE|No_merge] [Make_elements] [Stitch] [Improve]**

**Import STL "<filename>" [Feature_Angle <angle>] [Surface_Feature_Angle <angle>] [LINEAR|Gradient|Quadratic|Spline] [MERGE|No_merge] [Make_elements] [Stitch]**

**Import OBJ "<filename>" [Feature_Angle <angle>] [Surface_Feature_Angle <angle>] [Make_elements]**

**Facets** are simply triangles that have been stitched together to form surfaces. Faceted geometry representations are commonly used for graphics, bio-medical, geotechnical and many other applications that output a discrete surface representation. Upon import, the resulting geometry representation is Mesh-Based Geometry. Figure 1. shows an example of a faceted model and the resulting geometry created in CUBIT.



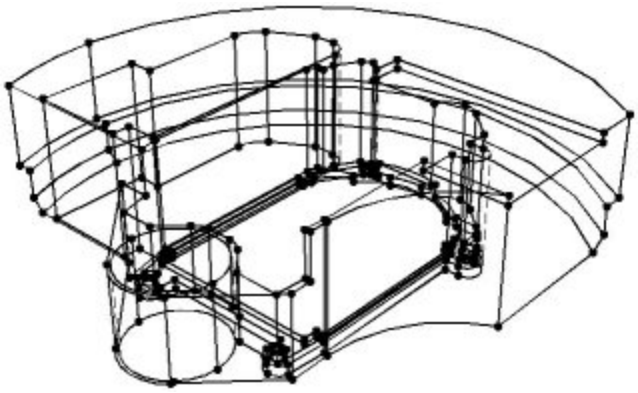


**Figure 1. Example of faceted model and the resulting solid model created in CUBIT from the facets.**

For convenience, the import facet command currently supports three different formats, facet, AVS and STL

- **Facet format**: The facet file format is a simple ASCII file that contains vertex coordinates and connectivities. The facet file format is described below.
- **AVS format**: The AVS format is a general geometry format that can support a variety of polygonal shapes. In CUBIT's implementation of the AVS import, it will support only triangles.
- **STL format**: Perhaps the most common format in the industry is Stereolithography (STL). CUBIT supports both ASCII and binary forms of the STL format. While the STL format is adequate for graphics and visualization, it can be problematic for geometry applications such as CUBIT. Each triangle in the STL format is represented independently. This means that multiple definitions of a single vertex are included in the file. CUBIT will attempt to merge duplicate vertices to form a water-tight surface. In cases where the vertex locations may not correspond exactly, an optional **tolerance** argument may be used on the import command. The **tolerance** option is used only for STL format files.
- **OBJ format**: A geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors. Cubit only supports importing OBJ files that contain tessellations with polygonal faces.

*Facet File Format*

The format for the ASCII facet file is as follows

| | | | |
|---|---|---|---|
| n | | | m |
| id1 | x1 | y1 | z1 |
| id2 | x2 | y2 | z2 |
| id3 | x3 | y3 | z3 |
| . | | | |
| . | | | |
| . | | | |

```
idn xn yn zn
fid1 id<1> id<2> id<3> [id<4>]
fid2 id<1> id<2> id<3> [id<4>]
fid3 id<1> id<2> id<3> [id<4>]
.
.
.
fidm id<1> id<2> id<3> [id<4>]
```

Where:

```
n = number of vertices
m = number of facet
id<i> = vertex ID of vertex i
x<i> y<i> z<i> = location of vertex i
fid<j> = facet ID of facet j
id<1> id<2> id<3> = IDs of facet vertices
[id<4>] = optional fourth vertex for quads
```

As noted above, the facets can be either quadrilaterals or triangles. Upon import, the facets serve as the underlying representation for the geometry. By default, the facets are not visible once the geometry has been imported. To view the facets, use the following command:

```
draw surf <id range> facets
```

*Feature Angle*

The **feature angle** option is used to specify the angle at which surfaces will be split by a curve or where curves will be split by a vertex. 180 degrees will generate a surface for every facet, while 0 degrees will define a single, unbroken surface from the shell of the mesh. The default angle is 135 degrees. This feature is identical to the feature angle option available when importing Exodus II files.

For the stl format, it is possible to independently control the feature angle for surfaces and curves. If **Surface_Feature_Angle** is specified, it controls the angle at which surfaces will be split by a curve, while **Feature_Angle** controls the angle at which curves will be split by a vertex. If **Surface_Feature_Angle** is not specified, **Feature_Angle** will control the angle for both surfaces and curves.

*Smooth Curves and Surfaces*

This option permits the use of a higher order approximation of the surface when remeshing/refining the resulting geometry. Default is to use the original facets themselves as the curve and surface geometry representation. If the facet model to be imported is to represent geometry with curved surfaces, it may be useful to apply this option. If the Spline option is selected, it will use a 4th order B-Spline approximation to the surface [Walton,96]. More information on using smooth approximation of the facets is available in Importing an Exodus II File.

*Merge*

This option allows the user to either merge or not merge the resulting surfaces. The default option is to merge adjacent surfaces. This results in non-manifold topology, where neighboring surfaces share common curves. The **no_merge** option, adjacent surfaces will generate distinct/separate curves.

*Make elements*

This option creates mesh elements from each of the facets on the facet surface.

*Stitch*

The stitch option is used with the facet or avs format files to try to merge vertices and triangles that are close. Figure 2 shows an example of where this might be employed. The model on the left contains facets that are not connected between the red and blue groups. In this case, the surfaces will not be water-tight, even though the vertices on the boundary between the two groups may be coincident. The **stitch** option attempts to eliminate the extra edge and vertex between the groups to form the model on the right. This option can be useful when importing facet files for 3D meshing. CUBIT's 3D meshing algorithms require a water-tight (closed) set of surfaces.

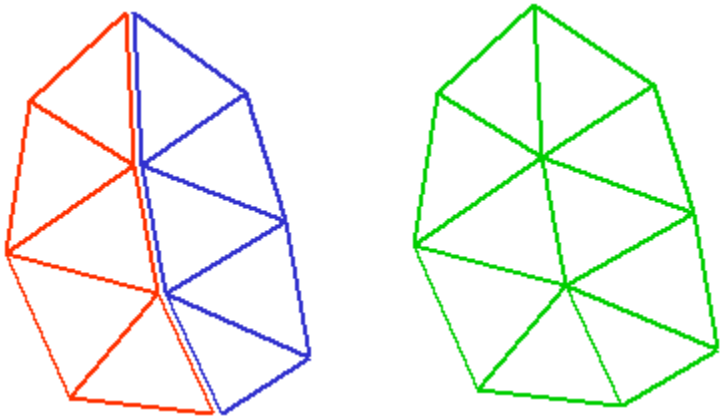
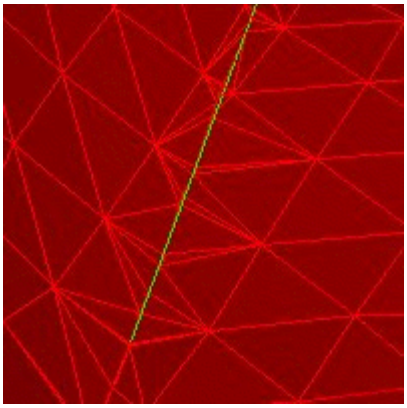


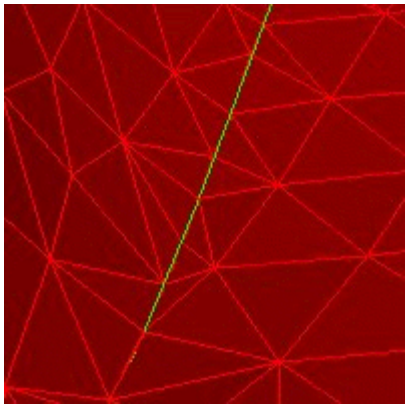
**Figure 2. Example use of the stitch option on import.**

*Improve*

The **improve** option will collapse short edges on the boundary of the triangulation that are less than 30% the length of the average edge length in the model. In some cases, short edges are the result of discrete boolean operations on the triangulation which may result in edges that are of negligible length. This option is particularly useful for boundaries where multiple surfaces come together at an edge. Figure 3. shows an example of where the improve option improved the quality of the triangles at the boundary. This option is especially useful if the facets themselves will be used for the FEA mesh.



**Triangles near a boundary that have not been used the improve option**



**The same set of triangles where improve option has collapsed edges**

**Figure 3. Example use of the improve option**

## Importing FASTQ Files

CUBIT can read a FASTQ file and convert it into an ACIS model:

```
Import Fastq '<fastq_filename>'
```

Note that the filename must be enclosed in single or double quotes.
FASTQ is an older, 2d meshing tool; (Blacker 88.) FASTQ files are a series of commands much like a CUBIT journal file. All FASTQ commands are fully supported except for the "Body" command (it is unnecessary and ignored), the "corn" (corner) line type, and some of the specialized mapping primitive "Scheme" commands. Standard mapping, paving, and triangle primitive scheme commands are handled. The pentagon, semicircle, and transition primitives are not handled directly, but are meshed using the paving scheme. The FASTQ input file may have to be modified if the Scheme commands use any non-alphabetic characters such as `+', `(`, or `)'. Circular lines with non-constant radius are generated as a logarithmic decrement spiral in FASTQ; in CUBIT they will be generated as an elliptical curve.
Since a FASTQ file by definition will be defined in a plane, it must be projected or swept to generate three dimensional geometry. CUBIT supports sweeping options to convert imported FASTQ geometries into volumetric regions.

## Importing Granite Files

As of version 13.0, native Granite models are no longer supported.

## Importing IGES Files

The ACIS IGES translator provides bi-directional functionality for data translation between ACIS and the IGES (Initial Graphics Exchange Specification) format.
The commands to import IGES files are:

**Import Iges '<iges_filename>' [No_bodies] [No_surfaces] [No_curves] [No_vertices] [Group {'<name>'|<id>}] [Nofreesurfaces] [HEAL|noheal] [Logfile ['filename'] [Display]] [Show_Each] [Sort]**

*Import Options*

It is possible to include free entities (vertices, curves and surfaces) in the file. Default operation is to read all entities in the file whether they are included as part of a body or are free. By using any of the options **no_bodies, no_surfaces, no_curves**, or **no_vertices**, the user may exclude certain types of *free* entities.
The **group** option of the import command will allow the user to create a group for each set of imported geometry. The newly created group can later be accessed using the name or id specified with the group option.
The **nofreesurfaces** option will automatically convert free surfaces to bodies. By default this option is off.
By default, bodies are automatically healed when imported - if this causes problems, you can disable this option by using the **noheal** argument.
The **logfile** option specifies a file where informational messages generated during import of the STEP file will be written. The display option will display the file.
The **show_each** option is a graphics option that applies to how the volumes are shown as they are imported. If there are multiple volumes in the file, the graphics display will be updated between each volume during import.
Normally the numerical IDs of the geometric entities contained in the ACIS model are used directly within CUBIT. The **sort** option provides the capability to compress the IDs read from the ACIS file. The sort option does the same thing as the compress ids sort command, but combines it with the import command to remove a step in the process.
Note that the IGES import and export functionality might not be available on all 64-bit platforms.
See also Exporting IGES Files.

## Importing STEP Files

The ACIS STEP translator provides bi-directional functionality for data translation between ACIS and the file format standard STEP AP203.
STEP AP203 is an international standard which defines a neutral file format for representation of configuration control design data for a product.
The command used to import a STEP file are:

**Import Step '<step_filename>' [No_bodies][No_surfaces] [No_curves] [No_vertices] [HEAL|Noheal] [Logfile ['filename'] [Display]] [Show_Each] [Group {'<name>'|<id>}] [Sort] [XML '<xml_filename>']**

*Import Options*

- It is possible to include free entities (vertices, curves and surfaces) in the file. The default operation is to read all entities in the file whether they are included as part

of a body or are free. By using any of the options **no_bodies**, **no_surfaces**, **no_curves**, or **no_vertices**, the user may exclude certain types of free entities.
- By default, bodies are automatically healed when imported - if this causes problems, you can disable this option by using the **noheal** argument.
- The **logfile** option specifies a file where informational messages generated during import of the STEP file will be written. The display option will display the file.
- The **show_each** option is a graphics option that applies to how the volumes are shown as they are imported. If there are multiple volumes in the file, the graphics display will be updated between each volume during import.
- The **group** option of the import command will allow the user to create a group for each set of imported geometry. The newly created group can later be accessed using the name or id specified with the group option.
- Normally the numerical IDs of the geometric entities contained in the STEP model are used directly within CUBIT. The **sort** option provides the capability to compress the IDs read from the STEP file. The sort option does the same thing as the compress ids sort command, but combines it with the import command to remove a step in the process.
- The **xml** option will read assembly information and other metadata from an XML file in the DART metadata XML format. See the metadata documentation and the Analyst's Home Page for details.
- If the geometric entities in the STEP file have names, they will be read in.
- Beginning with version 13.0, Cubit will read assembly information embedded in the imported STEP file. No additional arguments are required. The resultant assembly/part structure will be displayed in the GUI's main entity tree.

*Import Settings*

By default, names on bodies in STEP files are not read in. To change this, the following command is available:

```
[set] Read Step Body Names [on|OFF]
```

*Exporting a STEP file from Pro/Engineer*

To export a STEP file from Pro/ENGINEER, from the Export STEP Dialog, Press Options. In the file step_config.pro add the following:

```
STEP_EXPORT_FORMAT AP203_CD.
```

Also be sure your export option is set to Solids. If the geometry has problems in CUBIT, you may need to increase the geometry accuracy in Pro/ENGINEER.
See also Exporting STEP Files.

## Importing SGM Files

CUBIT contains experimental support for importing SGM and STEP files using the Scalable Geometric Modeler (SGM) to represent the geometry.

The commands to import SGM files are:

**Import SGM '<filename>' [restore_ids] [read_colors]**

*Import Options*

The **Import SGM** command can import either .sgm files or .step files. If given a .sgm file, the **restore_ids** option may be used to restore ids as they were in a previous SGM session. Restoration of IDs is not supported if a model was previously imported into SGM. The **read_colors** option is available if one wants to read colors found in the file. If **read_colors** is not given, colors will be automatically assigned by CUBIT based on the ID of the volumes.
Other CUBIT capabilities are limited when it comes to working with models imported into SGM. Visualization, list commands, and machine learning classification will work, but meshing, export and other operations are not yet supported.

# export

## Exporting Geometry

Geometry can be exported from CUBIT in a variety of formats, including the ACIS ".sat" and ".sab" formats as well as in more portable exchange formats like STEP and IGES.

- Exporting ACIS Files
- Exporting STEP Files
- Exporting IGES Files
- Exporting Facet Files

## Exporting ACIS Files

Geometry can be exported from within CUBIT to the ACIS "sat" (ASCII) and "sab" (binary) formats. These formats can be used to exchange geometry between ACIS-compliant applications. The command used to export geometry is:

**Export Acis [Debug] 'filename' [<geometry_entity_list>] [Binary|Ascii] [Current] [Overwrite]**

The filename should be enclosed in single or double quotes. By convention, binary and ASCII ACIS files use the .sab and .sat filename extensions, respectively. If a geometry entity list is not specified, the entire ACIS model is exported. A geometry entity list is specified in the same format used for other CUBIT commands (See Entity Specification). Note that the model is saved as manifold geometry, and will have that representation when imported back into CUBIT (See Non-Manifold Topology and Geometry Merging.)
When exporting, the filename extension will determine the default file type, either ASCII or binary. A .sat extension will default to ASCII; a .sab extension will default to binary. If you use a different file extension you can specify the type with the **[binary|ascii]** option

(with an unsupported extension exporting will default to ASCII but importing requires the type to be specified). Binary files can be significantly faster but are not guaranteed to be upward compatible nor cross-platform compatible (although testing has determined compatibility between NT and HP/UX).
In the GUI version, the **current** option will set the default filename for autosave (cntrl-S or File->Save (auto inc)) to the imported filename. Also, the filename is then set in the window titlebar.
When exporting with the "**file overwrite**" option on, the software will check to see if the file exists already, and if it does, exporting will fail in the command line version or ask to confirm the overwrite in the GUI version of CUBIT. The **overwrite** option will override this option and overwrite the file. The "file overwrite" option defaults to ON in the GUI version, OFF in the command line version.
When exporting, you can set the version of the Acis geometry. This allows backwards compatibility to previous versions of Cubit or other Acis-based applications. The command to change the Acis geometry engine version is:

**Set Geometry Version [version_number]**

where **version_number** can be one of the following:106, 107, 201, 300, 301, 401, 402, 403, 500, 501, 502, 503, 600, 601, 602, 603, 700, 701, 702, 703, 704, 705, 800, 1007, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2100, 2200, 2401, 2502. Note that you cannot set a version number that is higher than that of your current engine. For example, Cubit 6.0 was based on Acis 6.2, so you cannot set a geometry version of 700.
To retain any merging information during export of an ACIS file, **set attribute on** and **set attribute off** need to be used before and after the **export acis** command.
See also Importing ACIS Models.

## Exporting Facet Files

Facet files may be exported directly, or by converting from an ACIS representation. The syntax for exporting facet files is:

**Export Facets 'filename' <entity_list> [Overwrite]**

The overwrite function allows you to overwrite an existing facet file.
STL facet files may be generated from geometry or from a triangle mesh. The syntax for exporting to the STL format is:

**Export STL [ASCII|binary] 'filename' [<entity_list>] [tri <id_range>] [angle=15] [mesh|fast] [sidesets|sideset<ids>] [Overwrite]**

The **[entity_list]** option is a list of geometric entities (bodies, volumes, or surfaces). By default, the graphics facets for the geometric entities will be written to the STL file. The **[angle]** keyword specifies the dihedral angle used during facet generation. By default, a "water-tight" set of graphics facets is exported for solid volumes. If a water-tight set of facets is not of interest to the user and performance is more important, the **fast** option

can be used. It generates a faceting that does not take the extra step of ensuring water-tightness. To export the triangle mesh on the geometric entities, instead of the graphics facets, specify the **[mesh]** keyword. Note that STL export of quad meshes is not supported.
The **[sidesets]** or **[sideset <ids>]** options will export all or the specified sidesets respectively in your model as surface designation for any or all triangles in the file. If present, one sideset will be generated for each surface designation in the STL file. Following is an example surface designation in an STL file. It would appear following all triangles.

```
surface 1
  0 1 2 3 4 5 6 7 8 9
  10 11 12 13 14 15 16 17 18 19
  20 21 22 23
endsurface 1
```

The id following the surface designation will be used as the sideset ID.
Alternatively, a list of mesh triangles can be specified for export. If neither geometry entities nor mesh are specified, all volumes and sheet bodies are written out.

## Exporting IGES Files

The ACIS IGES translator provides bi-directional functionality for data translation between ACIS and the IGES (Initial Graphic Exchange Standard) format. The command to export IGES files is:

**Export Iges 'filename' [<geometry_entity_list>] [Solid] [Logfile ['filename'] [Display]] [Overwrite]**

As with ACIS file export, you can specify which individual entities to export. If unspecified, all ACIS entities are exported.
The **logfile** option is used to save information regarding the conversion to IGES format. This information saved to a file with the name specified by the user, or named 'iges_export.log' by default. When running the GUI version of CUBIT, the logfile can be displayed in a dialog window by using the **display** option.
The **solid** option allows solid volumes to be exported as Manifold Solid B-Rep Objects (MSBO). Without this option, the iges file is simply a collection of stand-alone surfaces.
The overwrite option works the same as with ACIS file export.
See Importing IGES Files for information on setting up the IGES import and export functionality.
Note that the IGES import and export functionality might not be available on all 64-bit platforms.

## Exporting STEP Files

CUBIT can export geometry to the STEP format, an emerging standard for storing geometry and other information. The STEP AP203 and STEP AP214 standards are supported. It is recommended to use AP214 for exchange of geometry information with CUBIT. The command used to export a STEP file is:

**Export Step 'filename' [<geometry_entity_list>] [Logfile ['filename'] [Display]] [Overwrite]**

As with ACIS file export, you can specify which individual entities to export. If unspecified, all ACIS entities are exported.
The **logfile** option is used to save information regarding the conversion to STEP format. This information saved to a file with the name specified by the user, or named 'step_export.log' by default. When running the GUI version of CUBIT, the logfile can be displayed in a dialog window by using the **display** option.
The overwrite option works the same as with ACIS file export.
If bodies, volumes, surfaces, curves, or vertices have names, they will be written into the STEP file.
See Importing STEP Files for information on setting up the STEP import and export functionality.

# Geometry Deletion

Geometry can be deleted from the model using the following command:

**Delete [Body | Surface | Curve | Vertex] <id_range>**

Any type of Body can be deleted, whether it is based on solid model geometry or another representation. Other entities (Surface, Curve, Vertex) can be deleted when they are "free", i.e. when they are not contained in an entity of higher topological order (Body, Surface or Curve, respectively); this type of geometry is often created from the lowest order topology up.

# Geometry Orientation

The orientation of surface and curve geometry is the direction of the normal and tangent vectors respectively.
Each surface has a forward (or top) side. The evaluation of the surface normal at any point on the surface will return a vector at that point, orthogonal to the surface and directed towards the forward side of the surface. The mesh faces generated on each surface will have the same normal direction as their owning surface.
Each curve has a forward direction and a corresponding start and end vertex. The direction of the curve is from start to end vertex. The evaluation of the tangent vector of the curve at any point along the curve will result in a vector that is both tangent to the curve and pointing in the forward direction of the curve (towards the end vertex along the path of the curve.) The mesh edges created on each curve will be oriented in the same direction as their owning curve. The exported nodes and edges of a curve mesh will be written in the order they occur along the path of the curve.
Higher-dimension geometry has uses lower-dimension geometry with an associated sense (forward or reversed) for each lower-dimension entity. For example, a volume as a sense for each surface used to bound the volume. If the surface normal points outside the volume, then the volume uses the surface with a forward sense. If the surface normal points into the interior of the volume, the volume uses the surface with a reversed sense.

Similarly a surface is bounded by a set of curves forming a loop such that the direction of the loop and the sense of each curve results in a cycle that is counter-clockwise around the surface normal.

## Adjusting Orientation

By default, a surface is oriented so that its normal points OUT of the volume of which it is a part. For a merged surface (a surface which belongs to more than one volume) or a free surface (a surface that belongs to no volume, also known as a sheet body), the orientation of the surface is arbitrary. The orientation of a surface influences the orientation of any elements created on that surface. All surface elements have the same orientation as the surface on which they are created. The following commands are available to adjust the normal-direction for a surface:

**Surface <id_range> Normal Opposite**

**Surface <id_range> Normal Volume <id>**

The orientation of a surface can be flipped from its current orientation by using the "Opposite" keyword. The orientation of a merged surface can be set to point OUT of a specific volume by specifying that volume in the "Volume" keyword.
Occasionally, volumes will be created "inside-out". The command:

**Reverse {Body|Volume|Surface} <id_range>**

will turn a given volume, surface, or body inside out. This should be equivalent to reversing the normals on all the surfaces. This shouldn't be encountered very often, as it is a very rare condition.
The following commands are available to adjust the tangent direction of a curve:

**Curve <id_range> Tangent Opposite**

**Curve <id_range> Tangent {Forward|Reverse} Surface <id>**

**Curve <id_range> Tangent {Start|End} Vertex <id>**

The first command reverses the tangent direction of the curve. The second command sets the tangent direction such that it is used by a specific surface with a specified sense. The third command sets the tangent direction of the curve such that the curve starts or ends with the specified vertex. For the latter two forms of the command, the curve must be adjacent to the specified surface or vertex.
The below command can be used to change the orientation of multiple curves at once. With the direction option, the curve will be oriented along the specified direction. With the location option, the vertex closest to the give location becomes the start vert in the oriented curve. The curve orientation can be reversed using the opposite argument. Also, a vertex id can be specified to make it the start vertex in the oriented curve.

**Curve <id_range> Orient Sense {direction (options)|location (options)|vertex <id_range>} [Opposite]**

The above command is useful in changing the orientation of multiple curves at once using various options described. This becomes helpful, e.g., when bias is applied on multiple curves. By default, bias depends on the orientation of the curve, i.e., bias begins at start vertex.

# Entity Measurement

To output various properties of entities, the following **Measure** command options are available.

- Measure Between
- Measure Small
- Measure Angle
- Measure Void
- Measure Volume
- Measure Surface

## Measure Between

**Measure Between { { Vertex|Curve|Surface |Volume|Node} <id1> | Location <options> | Plane <options> | Axis <options> } With { {Vertex|Curve|Surface|Volume|Node} <id2> | Location <options> | Plane <options> | Axis <options> }**

**Measure Between {Surface|Curve} <id1 > [Surface|Curve] <id2> [Node]**

**Measure Between {Vertex|Curve|Surface|Volume|Node|Edge|Face|Tri|Hex|Tet} <id1> With {Vertex|Curve|Surface|Volume|Node|Edge|Face|Tri|Hex|Tet} <id2>**

The **Measure Between** command outputs the distance from one entity, location, plane, or axis to the next. The two entities in the command should be separated by the word "with". The result will always be the minimum distance between entities. For example, measuring between two spheres will output the minimum distance between them, not the distance between centroids. The example shown below will output the minimum distance between vertex 1 and surface 2.

**measure between vertex 1 surface 2**

The second form of the command is just for surfaces or curves and contains the **Node** argument. This argument attempts to measure between corresponding nodes on a pair of surfaces or curves. The command tries to determine a one-to-one mapping of nodes between the pair. It returns the greatest distance between any two nodal pairs, least distance between any two nodal pairs, and average distance between all of the nodal pairs. The mapping algorithm works best on surfaces if they are parallel.

The last form of the command measures between any geometry or mesh entities. The measurement to the mesh entities is to their center (i.e. the averaged vector location of all of the nodes belonging to the mesh entity).
With 2 entities selected in the graphics window, the user can right click one of the entities and measure the distance between the entities.

## Measure Small

**Measure Small {Length|Area|Volume|All} {Body|Surface} <id_list>**

The **Measure Small** command locates all of the lengths, areas, or volumes smaller than the **Measure Small Tolerance** setting. Entities meeting the small tolerance criteria are listed in the output window and typically highlighted in the view port. The following two commands set the small tolerance to 0.1 and output all of the curves within body 1 with lengths at or below the small tolerance.

```
set measure small tolerance 0.1
measure small length body 1
```

## Measure Angle

**Measure Angle { Direction <options> | Plane <options> | Axis <options> } With { Direction <options> | Plane <options> | Axis <options> }**

The **Measure Angle** command displays the interior angle between the two entered entities. When a plane and a direction are specified, the angle between the direction vector and its projection into the plane is displayed. The measured angle represents the distance between the orientations of entities, and does not require the entities to intersect. Angles of model features can be measured by using the various options associated with the Direction, Planes, and Axis commands.

```
measure angle direction tangent curve 1 with plane surf 1
```

## Measure Void

**Measure Void [Face | Tri] <range>[No_Checks]**

The **Measure Void** command takes a closed list of quadrilaterals or triangles and calculates the volume of the internal region defined by the given list of elements. This command assumes that the normals on the given elements are consistently ordered. If the normals are pointing away from the interior of the void, the reported volume may be negative. This command will check to ensure that the given elements do form a closed, manifold shell, otherwise an error is reported. Common uses will be to calculate the volume of an internal void for use in determining bulk element properties for a thermal analysis.
Rather than issuing an error, the **no_checks** option does not check for closure of the faces and will compute a void volume regardless of their watertightness. This is useful if faces are all touching, but may not have complete topological closure.

## Measure Volume

**Measure Volume <range> [Overlap | Shell]**

The **Measure Volume** command prints summary information about the specified volumes and surfaces of these volumes, such as average volume, minimum volume, angles, average surface area, etc. If the **shell** option is specified information about the shells of the volumes is additionally printed. The **overlap** option does not print any summary information, but only reports pairs of intersecting volumes.

## Measure Surface

**Measure Surface <range>**

The **Measure Surface** command prints summary information about the specified surfaces and curves of these surfaces, such as average area, minimum area, angles, minimum curve length, etc.

# Mesh Generation

## Mesh Generation


- Meshing the Geometry
- Interval Assignment
- Meshing Schemes
- Mesh Quality Assessment
- Mesh Modification
- Mesh Validity
- Mesh Adaptivity and Sizing Functions
- Mesh Deletion
- Free Meshes
- Skinning a Mesh


The methods used to generate a mesh on existing geometry are discussed in this chapter. The definitions used to describe the process are first presented, followed by descriptions of interval specification, mesh scheme selection, and available curve, surface, and volume meshing techniques. The chapter concludes with a description of the mesh editing capabilities, and the quality metrics available for viewing mesh quality.

### Element Types

For each entity topology-type in the model geometry, CUBIT can discretize the entity using one, or several, types of basic elements, for each order entity in the geometry (vertex, curve, etc.). CUBIT uses a basic element designator to describe the corresponding entity, or entities, in the mesh, and a given geometric topology entity can be discretized with one, or several, of basic elements types in CUBIT. For example, a geometric surface in CUBIT is discretized into a number of faces, where faces is the basic element designator for surfaces. These faces can consist of two types of basic elements, quadrilaterals or triangles. The basic element designators corresponding to each type of geometric entity, along with the types of basic elements supported in CUBIT, are summarized in the table below.

For each basic element, CUBIT also supports several element type definitions, whose use depends on the level of accuracy desired in the finite element analysis. For example, CUBIT can write both linear (4-noded) and quadratic (8- or 9-noded) quadrilaterals. The element type definition is specified after meshing occurs, as part of the boundary condition specification. See Finite Element Model Definition for a description of that process and the various element types available in CUBIT.

Each mesh entity is associated with a geometric entity which "owns" it. This associativity allows the user to mesh, display, color, and attach attributes to the mesh through the geometry. For example, setting a mesh attribute on a surface affects all faces owned by that surface.

### Mesh Generation Process

Starting with a geometric model, the mesh generation process in CUBIT consists of four primary steps:

**Set interval size** and count for individual entities or groups

The size or interval is always applied to a specific geometric entity. For example:
**volume 1 size 2.0**
**Set mesh schemes**
CUBIT supports numerous meshing schemes for meshing solid model entities. For example:
**volume 1 scheme sweep**
**Generate the mesh** for the model
Use the mesh command to generate the mesh on a specified geometric entity. For example:
**mesh volume 1**
**Inspect mesh for quality** and suitability for targeted analysis
CUBIT provides various quality metrics for the user to verify the suitability of the mesh for analysis. The quality command can be used to check the elements generated on a specific geometric entity. For example:
**quality volume 1**
There are also mechanisms for improving mesh quality locally using smoothing and local mesh topology changes and refinement. For complex models, this process can be iterative, repeating all of the steps above.
The mesh for any given geometry is usually generated hierarchically. For example, if the mesh command is issued on a volume, first its vertices are meshed with nodes, then curves are meshed with edges, then surfaces are meshed with faces, and finally the volume is meshed with hexes. Vertex meshing is of course trivial and thus the user is given little control over this process. However, curve, surface, and volume meshing can be directly controlled by the user. Each of the steps listed are described in detail in the following sections.

| Geometry Entity Type | Basic Element Designator | Basic Element(s) In CUBIT |
|---|---|---|
| Vertex | Node | Node |
| Curve | Edge | Edge |
| Surface | Face | Quadrilateral, Triangle |
| Volume (or Body) | Element | Hexahedron, Tetrahedron, Pyramid, Wedge |

# Meshing the Geometry

After assigning interval or sizing attributes to a geometric entity and a meshing scheme is applied, the geometry is ready to be meshed. To mesh a geometric entity, use the command:

**Mesh <entity> <id_range> [GLOBAL|Individual]**

The **<entity>** to be meshed may be any one of the following:

**Body**
**Volume**
**Surface**
**Curve**
**Vertex**

The **Global** and **Individual** options affect how the constraints are gathered for interval matching. With the Global option, the interval constraint equations are calculated from all entities in the entity list. The Individual option calculates the interval constraint equations from each entity individually. The Global option is the default.

### Default Scheme and Interval Selection

If either interval settings or schemes have not already been set on the entities being meshed, CUBIT will do its best to automatically set one or both of these attributes. See Auto Scheme Selection and Auto Specification of Intervals for a description of how CUBIT chooses these attributes. In cases where the automatic scheme selection algorithm fails to select a scheme for the geometry, the meshing operation will fail. In this case explicit specification of the meshing scheme and/or further geometry decomposition may be necessary.

### Continuing Meshing After a Mesh Failure

Frequently when meshing large assemblies containing a number of volumes, the mesh command can be applied to a group of volumes with the same mesh command. Typically, if a mesh failure is detected, the meshing operation will continue to mesh the remaining volumes specified at the command line. The following command permits the user to override this feature to discontinue meshing additional volumes and return to the command line immediately after a mesh failure is detected:

**Set Continue Meshing [ON|Off]**

The default for this command is **ON**.
Turning this setting **OFF** is useful when meshing assemblies where a meshing failure of one volume would adversely affect the meshing of adjoining volume(s). This occurs frequently when meshing a sweep group using the sweep scheme.

# interval assignment

## Interval Assignment

- Interval Firmness
- Setting Interval Sizes
  - Explicit Specification of Intervals
  - Explicit Specification of Intervals Using Interval Size
  - Periodic Intervals
  - Relative Intervals
  - Automatic Specification of Intervals
  - Vertex Sizing and Automatic Curve Biasing
- Additional Interval Constraints
- Match Intervals Solver

- Examining Intervals
  - Listing Intervals
  - Mesh Preview

"Intervals" means the number of mesh edges on a curve (or across a periodic surface). "Interval matching" is the process of assigning intervals that are close to the user-desires, and satisfy the constraints imposed by the various quad and hex meshing algorithms (e.g., equal intervals on opposite sides of a mapped surface). The desired number of intervals (goals) are set on geometric entities by either specifying the interval count or size. "Soft" user-set and auto-set mesh sizes and interval counts are goals. Any hard counts are part of the constraints. The constraints are determined by the meshing schemes, as well as the vertex types (corners) of surfaces, and the edge-types of volumes, and sweep directions. The user may set additional constraints, including upper and lower bounds.

## Additional Interval Constraints

*Curve Minimum* | *Maximum*

Rather than specifying a hard interval count, which may overconstrain the interval matcher, the user can specify an upper and lower bound that is acceptable. Typical uses are sweeping a complex assembly where the normal compromises lead to too many (or few) intervals on specific curves, or thin layers.

```
{Group|Body|Volume|Surface} <range>
{Interval | Size | Periodic Interval} {[Lower]|Upper}
Bound {On|Off|<bound>}
```

```
Curve <range> {Interval | Size} {[Lower]|Upper}
Bound {On|Off|<bound>}
```

*Loop Minimum* | *Maximum*

The user can specify the minimum and maximum number of intervals on a loop. This is mostly used for small loops, such as holes drilled in a plate, to ensure they have at least 4 or 6 intervals. If no entties are specified, then the setting is applied globally to all current and future loops in the model.

```
set [{group|volume|surface|curve} <range>]
interval loop minimum {<count>|default}
```

```
set [{group|volume|surface|curve} <range>]
interval loop maximum {<count>|default}
```

```
list [{group|volume|surface|curve} <range>]
interval loop [minimum] [maximum] [default]
```

*Manual Constraints*

#### General Constraints

**Curve <curve_id_range> Interval {Equal_to|Greater_than_equal|Less_than_equal} [Curve] <curve_id_range> [ Extra <intervals>]**

**Curve <curve_id_range> Interval {Equal_to|Greater_than_equal|Less_than_equal} Extra <intervals>**

These sets a constraint that the interval matcher resolves when it is run. E.g., the command "curve 2 3 greater_than_equal curve 4 5 extra 4" stores the inequality constraint "c2 + c3 >= c4 + c5 + 4" in the interval matcher. While this can resolve quality issues, it is also an easy way to make the interval matching problem infeasible.

#### Sets of Curves with the Same Intervals

Interval **same** is a two way constraint that is resolved immediately. If the user subsequently changes the interval on a curve in the set, then the other curves are changed immediately. One problems is if the user hard sets an interval on one curve and then sets a size on another, the hard set interval on the other curve is not changed.

**Curve <range> Interval {Same|Different}**

**List Curve [ <curve_id_range> ] Interval Same**

Specifying that curves have the "same" intervals stores them in a set. More curves may be added to an existing set, and sets merged, by future commands. The current contents of the affected sets are printed after each command. A curve may be removed from a set by specifying that its intervals are "different."

#### Even

The user can also constrain the parity of intervals on curves:

**{Curve|Surface|Volume} <range> Interval {Even | Odd}**

If **Even** is specified, then during subsequent interval setting commands and during interval assignment, curves are forced to have an even number of intervals. If the current number of intervals is odd, then it is increased by one to be even. If **Odd** is specified then intervals may be either even or odd. Setting intervals to even is useful in problems where adjoining faces are paved one by one without global interval assignment.

## Vertex Sizing and Automatic Curve Biasing

Sizes can now be specified on vertices to control biasing along curves. If a curve has a bias scheme the vertex sizes will be honored, even if it is inherited from parent geometry. Set a size on a vertex  with the following command:

**vertex <id> size <size>**

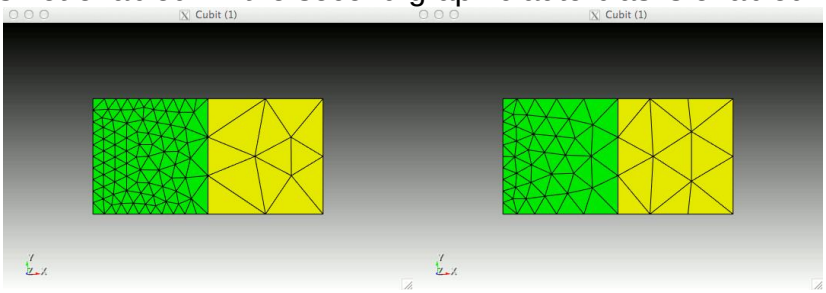
Bias can be turned on with:

**curve <id> scheme bias**

For tri/tet meshing, curve biasing is on by default to generate higher quality tri/tet meshes. Not only is the difference noticeable when setting sizes on vertices, but it is also noticeable when setting various sizes on connected curves, surfaces, or volumes. To turn curve biasing off issue the following command:

**curve<id> scheme equal**

In the following examples, the surfaces have been given sizes. In the first graphic auto bias is not enabled. In the second graphic auto bias is enabled.




When auto bias is enabled sizes on vertices are respected. If a size hasn't been directly set on a vertex the size is inherited from the parent(s). If there are multiple parents the inherited size is averaged. In the examples shown above the sizes of the vertices attached to both surfaces was an average of the two surface sizes. That affected the biasing while curve meshing.

## Explicit Specification of Intervals

The density of mesh edges along curves is specified by setting the actual number of intervals or by specifying a desired interval size. The number of intervals can be explicitly set curve by curve, or implicitly set by specifying the intervals on a surface or volume containing that edge. For example, setting the intervals for a volume sets the intervals on all curves in that volume.

The command to specify the number of intervals at the command line is:

**{Curve|Surface|Volume|Body|Group} <range> Interval <intervals>**

When setting interval counts for surfaces, volumes, bodies and groups, an interval's firmness of ***soft*** is assigned to the owned curves. When setting the interval count for a curve, a firmness of ***hard*** is assigned.

The user can scale the current intervals with the following commands. Scaling is done on an entity by entity basis.

**{Curve|Surface|Volume|Body|Group} <range> Interval Factor <factor>**

## Explicit Specification of Intervals Using Interval Size

The number of intervals along curves can be specifying by setting a desired interval size. The interval size can be explicitly set curve by curve, or indirectly set by specifying the interval size on a surface or volume containing that curve. The size for an entity is determined with the following method. If the entity has a size explicitly set then that size is used. Otherwise the entity averages the size determined for its parents. If an entity doesn't have any parents then a size is automatically calculated from all of the geometry

in the model. If the auto size functionality is turned off then a default size of 1.0 is used. Some meshing algorithms may calculate a different default size.
For example, Suppose you have two volumes that share a face and corresponding curves. If the size on volume one is set to 1.0 and the size on volume two is set to 3.0 then the size for the common face will be set to 2.0. The size for the remaining faces on volume one and two will be 1.0 and 3.0 respectively. The size for the common curves will be set to 2.0.
The command to specify the interval size at the command line is:

**{Curve|Surface|Volume|Body|Group} <range> [Interval] Size <interval_size>**

Interval sizes set directly on an entity are given the type “user_set”. Interval sizes determined from parents or automatically calculated are give the type “calculated”.
When interval matching or meshing the interval count for each curve is computed by dividing the curve's arc length by the specified interval size. Interval counts calculated in this manner are considered to have a default firmness of soft.
The user can scale the current intervals or size with the following commands. Scaling is done on an entity by entity basis.

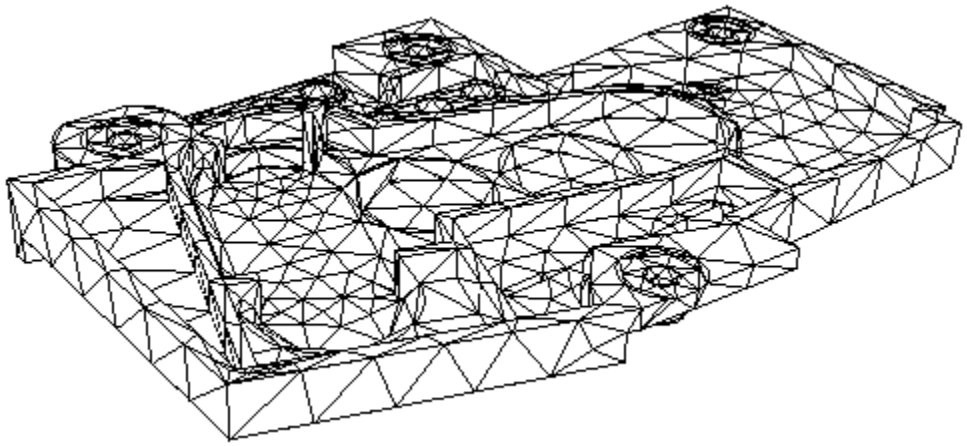
**{Curve|Surface|Volume|Body|Group} <range> [Interval] Size Factor <factor>**

## Automatic Specification of Interval Size

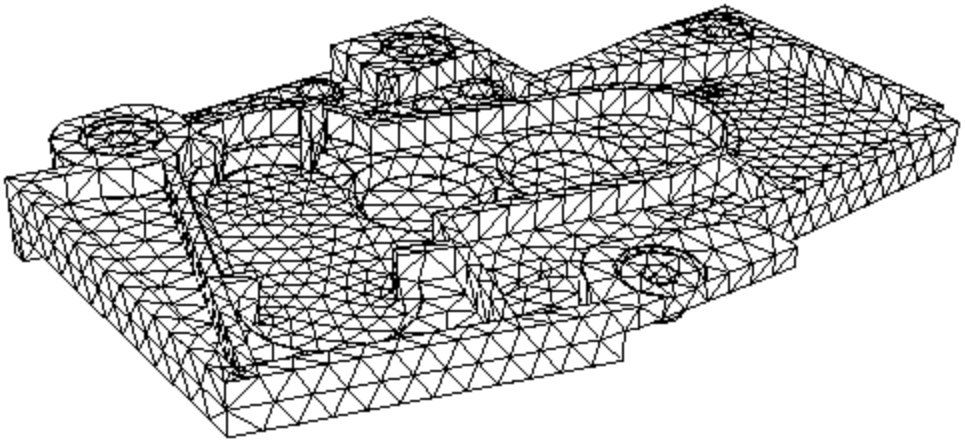
In addition to specifying intervals explicitly based on a known count or size, CUBIT is able to compute interval sizes automatically based on characteristics of the model geometry. The following automatic interval size setting command can be used:

**{geom_list} Size Auto [Factor <factor> ] [Individual] [Propagate]**

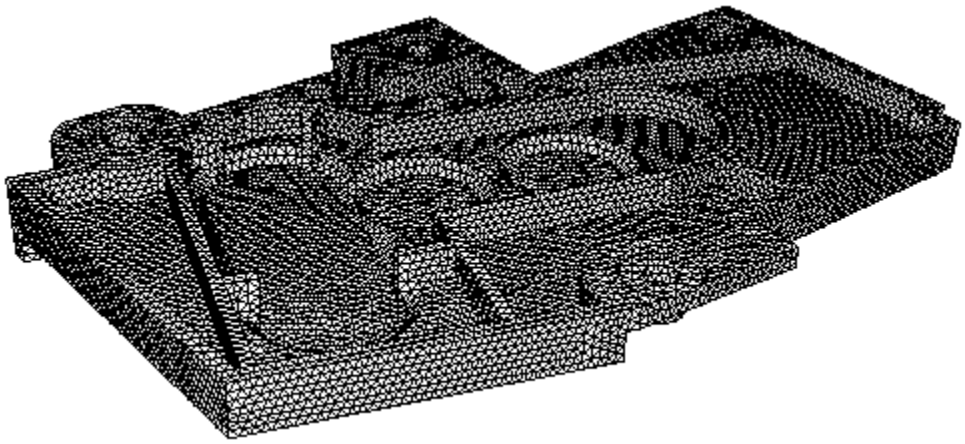
Vertices are not valid in the **geom_list** for this command. Automatic interval size assignment works by examining the geometric characteristics of the entities in the **geom_list** and assigning a heuristic size to the entities and their child entities. The factor may be a floating point number between 1.0 and 10.0, where 1.0 represents a fine interval size and 10.0 represents a coarse size. Figure 1 shows an example of different auto size specification on a CAD model.



**(a) auto size factor = 7.0**



**(b) auto size factor = 5.0**



**(c) auto size factor = 1.0**

The user may assign the interval size to be the arc length of the smallest curve contained in the specified entity or entities using the following command:

**{geom_list} Size Smallest Curve**

Vertices are not allowed in the **geom_list** for this command. This command assigns a ***soft*** interval firmness.

*Automatic Interval Size Specification*

An automatic interval size with an auto size factor of 5 will automatically be computed and applied to any curve for which the following is true:

1) Intervals have not been explicitly defined by the user for a curve or its owning entities.

2) An Interval size has not been explicitly defined by the user for a curve and it is not possible to determine an interval size from its owning entities.

This automatic interval size is based upon all the geometry in the model. The automatic interval size specifications can be overridden easily by specifying another auto size factor or an explicit interval size.
If an auto size factor of 5 is undesirable for most meshing operations, the default factor may be changed by using the following command:

**Set Auto Size Default <value>**

where **value** is a number from 1 to 10. This will be the default auto size factor used when either a factor has not been specified on the size auto command or when an automatic interval size specification is used.
In previous versions of CUBIT a default interval of 1 was assigned to all entities. If this behavior is still desired, the following command may be used to enforce this condition:

**Set Default Autosize [ON|off]**

*Maximum Spanning Angle on Arcs*

On many CAD models, arcs or small holes require that a finer mesh be specified around these entities in order to maintain reasonable mesh quality. To facilitate this, the user may specify the maximum angle an element edge may span on an arc. To change or list the maximum arc span, use the following commands

**Set Maximum Arc_Span <angle>**
**List Maximum Arc_Span**

The angle parameter must be a positive value less than 360. The maximum arc span setting will only be used if there is not already a user defined interval set on the arc, and if the interval setting produces mesh edges which exceed the maximum spanning angle. Figure 2 shows the effect of three different maximum arc_span settings on a small hole using the pave scheme.

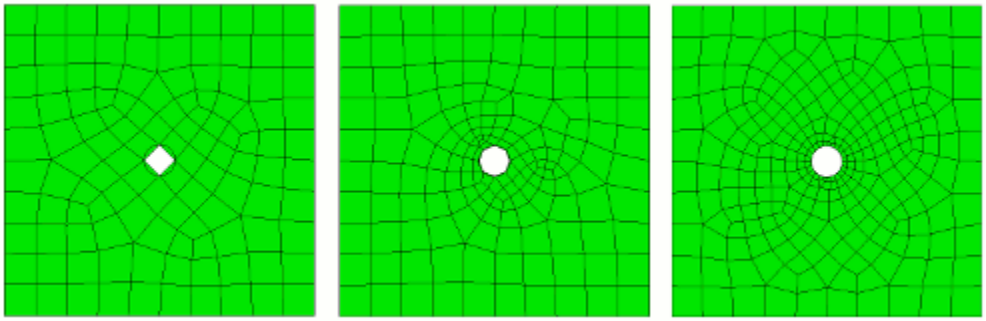


**Figure 2. Maximum arc_span settings of 90, 45 and 15 degrees respectively.**

**Default arc span setting**: In addition to setting an automatic size factor, if there are otherwise no user-defined interval sizes defined on an arc and no **maximum arc_span** has been set by the user when a tetrahedral mesh or triangle mesh is defined, a maximum spanning angle of **60 degrees** will be used. Removing the use of the arc_span setting can be accomplished with the following:

**Set Maximum Arc_Span Default**

Note that once interval sizes have been defined when the entity has been meshed, it may be necessary to reset the interval settings **(reset {geom_list})** to use a new maximum arc span setting when remeshing.

## Interval Matching

Each meshing scheme in CUBIT imposes constraints on the intervals assigned to the curves bounding the surface or volume. For example, map meshing a surface requires that intervals on opposite sides are equal. Meshing any surface with quadrilaterals, regardless of scheme, requires an even number of intervals on its boundary. For connected surfaces and assemblies, these interval constraints must be resolved globally to ensure that each surface and volume will be meshable with the assigned scheme. Beyond satisfying these constraints, the number of intervals (mesh size) should be what the user wants, or as close to it as possible. The global solution technique implemented in CUBIT is referred to as "interval matching."

Interval matching is automatically performed by the **mesh** command before generating elements. Invoking interval matching manually is useful if the user wishes to mesh only some entities but wants to ensure it will be possible to mesh all entities later: e.g. match intervals on all the volumes, but just mesh surface 3 for now. Interval matching can also be called to check whether the assigned schemes, corners, edge types, and intervals are compatible.

The command syntax for manually matching intervals is the following:

**Match Intervals {Surface|Volume|Body|Group} <range>**

Here the entity list can be any mixed collection of groups, bodies, volumes, surfaces and curves.

*Troubleshooting*

If the quality of the interval solution is poor, try changing the interval count on individual entities, or, e.g., changing some schemes from map to pave to provide more freedom, and rerunning interval matching.

There may be no interval solution. To improve the chances of finding a solution, at least as a test, try setting intervals to soft firmness. Even with soft intervals, a solution might not exist if the mapping corners or sweep directions do not line up right.

If there is no solution, the following command may help in determining the cause (this is not yet supported by IIA in Cubit 15.6):

**Match Intervals {Surface|Volume|Body|Group} <range> [Seed Curve <range>] [Assign Groups [Only|Infeasible]] [Map|Pave]**

Specifying **Assign Groups** will create groups that contain independent subproblems of the global problem. Specifying **Assign Groups Only** will group independent subproblems, but the algorithm will not attempt to solve these subproblems. **Assign Groups Infeasible** will put each independent subproblem with no solution into specially named groups. Often poor corner choices and surface meshing schemes will be

illuminated this way. If **Map** or **Pave** is specified, then only subproblems involving mapping or paving constraints will be considered. If a **Seed Curve** is specified, then only those subproblems containing that curve will be considered.

*Interval Solver Version*

For consistent results running old journal files, the user may want to use the old solver.

**set interval version {<cubit version number>|default}**

**list interval version**

Setting version >= 15.6 uses IIA (2020 solver) and enables small-loop bounds. Using a prior version uses BBIA (1997 solver). Both solvers solve the problem

| | | |
|---|---|---|
| min | | f(x,g) |
| s.t. | | Ax=b |
| x | in | [lo,hi] |
| A, x, b integer | | |

but the objective f is slightly different so the solution may be different. General problems of this form are known to be difficult because the solution must be integer valued, and neither solver is guaranteed to find the global minimum.
Here x are the intervals and internal variables; b are the bounds on those variables; A and b are the constraints, including hardsets; g are the goals, the user-desired number of intervals for each curve. The objective f models the desire to minimize the maximum relative change in mesh size. In mathematical language, for the new IIA solver f is the maximum lexicographic vector of the ratio achieved:goal if achieved>goal, else goal:achieved. The older solver uses an approximation to IIA's f.

#### IIA Solver >= Cubit 15.6

Cubit 15.6 in 2020 introduced "Incremental Interval Assignment (IIA)" based on integer linear algebra. It robustly and quickly finds an integer solutions to Ax=b, and tries to improve it by adding integer vectors from A's nullspace to satisfy the bounds and come close to the goals.
IIA solutions tend to be slightly coarser than BBIA, often one interval less, and closer to the user goals and the global optimum. IIA is faster than BBIA, dramatically faster (6000x) in some cases involving submapping and hard-sets. Please send the Cubit team any example which takes longer than 1 second.

#### BBIA Solver < Cubit 15.6

The solution method that Cubit introduced in 1996 was to use linear programming to find a floating point solution to Ax=b, then use branch-and-bound rounding to try to find a nearby integer solution. The first part is fast and robust, the second part slow and error prone. Linear programming is limited to linear objective functions, so the goal:achieved ratio is only approximated and extreme size differences can give unexpected compromises.

For BBIA, advanced users may wish to experiment with the following:

**Set Match Intervals Rounding {on|off}**

**Set Match Intervals Fast {on|off}**

**Set Match Intervals Delta <interval_difference = 0.>**

If **set match intervals rounding** is set to **on**, the intervals will be rounded to the nearest integer. If the setting is **off**, the intervals will be rounded toward the user specified intervals.
If **set match intervals fast** is set to **off** a single curve will be fixed per branch and bound iteration. Note in rare cases this may produce better meshes, but will generally be slower. If set **on**, multiple curves will be fixed per iteration.
**Set match intervals delta** affects the convergence tolerance for the optimization. Here **delta** means the difference between a curve's goal and assigned intervals. A larger value makes matching intervals faster, but the quality of the solution may be worse. The default is 0.0. Hint: try 1.0.

## Interval Firmness

Before describing the methods used to set and change intervals, it is important that the user understand the concept of interval firmness. An interval firmness value is assigned to a geometry curve along with an interval count or size; this firmness is one of the following values:

**hard**: interval count is fixed and is not adjusted by interval size command or by interval matching
**soft**: current interval count is a goal and may be adjusted up or down slightly by interval matching or changed by other interval size commands.
**default**: default firmness setting, used for detecting whether intervals have been set explicitly by the user or by other tools

Interval firmness is used in several ways in CUBIT. Each curve is assigned an interval firmness along with an interval count or size. Commands and tools which change intervals also affect the interval firmness of the curves. Those same commands and tools which change intervals can only do so if the curves being changed have a lower-precedence interval firmness. The firmness settings are listed above in order of decreasing precedence. For example, some commands are only able to change curves whose interval firmness is soft or default ; curves with hard firmness are not changed by these commands.
More examples of interval setting commands and how they are affected by firmness are given in the following sections.
A curve's interval firmness can be set explicitly by the user, either for an individual curve or for all the curves contained in a higher order entity, using the command:

**{geom_list} Interval {Default | Soft | Hard}**

All curves are initialized with a firmness of default. Any command that changes intervals (including interval assignment) upgrades the firmness to at least soft.

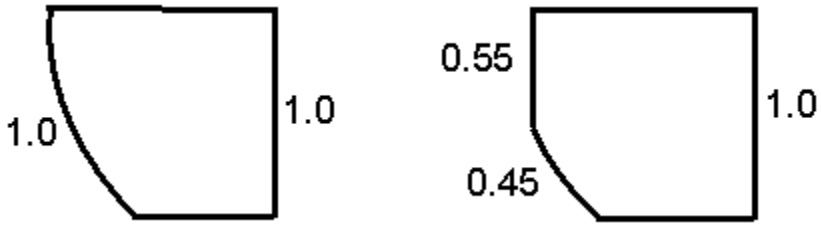
*Precedence*

If a size is specified multiple times for a single entity, the following precedence is used:

- The highest firmness command takes precedence.
  Hard commands include "curve <id> interval <val>", and "{geometry_list} interval hard" will fix the size at the current size.
- Within a given firmness, the last-issued command takes precedence.
  For example, if the user commands "surface 1 size 1" then "volume 1 size 2", and surface 1 is part of volume 1, then surface 1 will have a size of 2.

## Relative Intervals

If the user needs fine control over mesh density, then for curvy or slanted sides of swept geometries, it is often useful to treat curves as if they had a different length when setting interval sizes. For example, the user may wish to specify that a slanting side curve and a straight side curve have the same "relative" length, despite their true length as shown in the following figure. These are not interval matching constraints; interval matching may change intervals so that the user-specified ratio does not hold exactly.




The relative lengths of curves are set with the following command:

**{geom_list} Relative Length <size>**

The following command is used to assign intervals proportional to these lengths:

**{geom_list} Relative Interval <base_interval>**

For a curve with relative length x, setting a relative interval of y produces xy intervals, rounded to the nearest integer.

## Mesh Interval Preview

It is sometimes useful to view the nodal locations/intervals on curves graphically before meshing (which can take considerably more time). The command to do this is:

**Preview Mesh {Body|Volume|Surface|Curve|Vertex} <id_range> [Hard] [color <color>]**

To clear the display of the temporary nodes, simply issue a "**display**" command. The purpose of the **hard** option is that only curves that have an interval firmness of hard will be previewed.

### Periodic Intervals

The number of intervals on a periodic surface, such as a cylinder, in the dimension that is not represented by a curve is usually set implicitly by the surface size.
However, periodic intervals and firmness can be specified explicitly by the following commands:

**Surface <range> Periodic Interval <intervals>**

**Surface <range> Periodic Interval {Default|Soft|Hard}**

# meshing schemes

## Meshing Schemes

Meshing schemes in CUBIT can be divided into four broad categories.

- Traditional Meshing Schemes
- Free Meshing Schemes
- Conversional Meshing Schemes
- Duplication Meshing Schemes

In addition, Cubit supports two parallel meshing applications, pCamal and Sculpt

- Parallel Meshing

If no scheme is selected, Cubit will attempt to assign a scheme using the automatic scheme selection methods.

- Automatic Scheme Selection

*Traditional Meshing Schemes*

Traditional meshing schemes are used to apply a mesh to an existing geometry using the methods described in Meshing the Geometry (i.e. setting a scheme, applying interval sizes, and meshing). Traditional meshing schemes are available for all geometry types.

- Bias, Dualbias
- Circle
- Curvature
- Equal
- Hole
- Mapping
- Pave
- STransition
- Stretch
- Submap
- Sweep
- Tetmesh
- Tetprimitive
- Tridelaunay

- Pentagon
- Pinpoint
- Polyhedron
- Sphere
- TriAdvance
- Trimap
- Trimesh
- Tripave
- Triprimitive

*Free Meshing Schemes*

Free meshing schemes will create a free-standing mesh without any prior existing geometry. The final mesh will have mesh-based geometry.

- Radialmesh

*Conversional Meshing Schemes*

Conversional meshing schemes are used to convert an existing mesh into a mesh of different element type or size. For example, the THex scheme will convert a tetrahedral mesh into a hexahedral mesh.

- HTet
- QTri
- THex
- TQuad

*Duplication Meshing Scheme*

The duplication meshing scheme is used to copy an existing mesh from one geometry onto another similar geometry.

- Copy

*General Meshing Information*

Information on specific mesh schemes available in CUBIT is given in this section. The following sections have important meshing-related information as well, and should be read before applying any of the mesh schemes described below.
In most cases, meshing a geometric entity in CUBIT consists of three steps:

- Set the interval number or size for the entity (See Interval Assignment.)
- Set the scheme for the object, along with any scheme-specific information, using the scheme setting commands described below.
- Mesh the object, using the command:

**Mesh {geom_list}**

This command will match intervals on the given entity, then mesh any unmeshed lower order entities, then mesh the given entity.
After meshing is completed, the mesh quality is automatically checked (see Mesh Quality Assessment), then the mesh is drawn in the graphics window.
The following table classifies the meshing schemes with respect to their applicable geometry.

| Curves | Surfaces | Volumes |
|---|---|---|
| Bias/Dualbias | Circle | Copy |
| Copy | Copy | HTet |
| Curvature | | Mapping |
| | Hole | Polyhedron |
| Equal | Mapping | Sphere |
| Pinpoint | | Submap |
| Stretch | Pave | Sweep |
| | Pentagon | TetMesh, TetINTRIA |
| | Polyhedron | Tetprimitive |
| | QTri | THex |
| | Submap | |
| | TriDelaunay | |
| | Triprimitive | |
| | TriMap | |
| | TriMesh | |
| | TriAdvance | |
| | TriPave | |
| | STransition | |

duplication

# *Copying a Mesh*

**Applies to**: Curves, Surfaces, Volumes
**Summary**: Copies the mesh from one entity to another
**Syntax**:

**Curve <range> Scheme Copy source Curve <range> [Source Percent [<percentage> | auto]] [Source [combine|SEPARATE]] [Target [combine|SEPARATE]] [Source Vertex <id_range>] [Target Vertex <id_range>]]**

**Surface <id> Scheme Copy Source Surface <id> Source Curve <id> Target Curve <id> Source Vertex <id> Target Vertex <id> [Nosmoothing] [mirror]**

**Volume <range> Scheme Copy [Source Volume] <id> [[Source Surface <id> Target Surface <id>] [Source Curve <id> Target Curve <id>] [Source Vertex <id> Target Vertex <id>]][Nosmoothing]**

**Copy Mesh Curve <id> Onto Curve <curve_id_range> [Source Node <starting node id> <ending node id>] [Source Percent [<percentage>|auto]] [Source Vertex <id_range>] [Target Vertex <id_range>]**

**Copy Mesh Surface <surface_id> Onto Surface <surface_id> Source Vertex <id> Target Vertex <id> Source Curve <id> Target Curve <id> [interior (pair vertex <id> <id>) ...] [smooth] [mirror] [preview]**

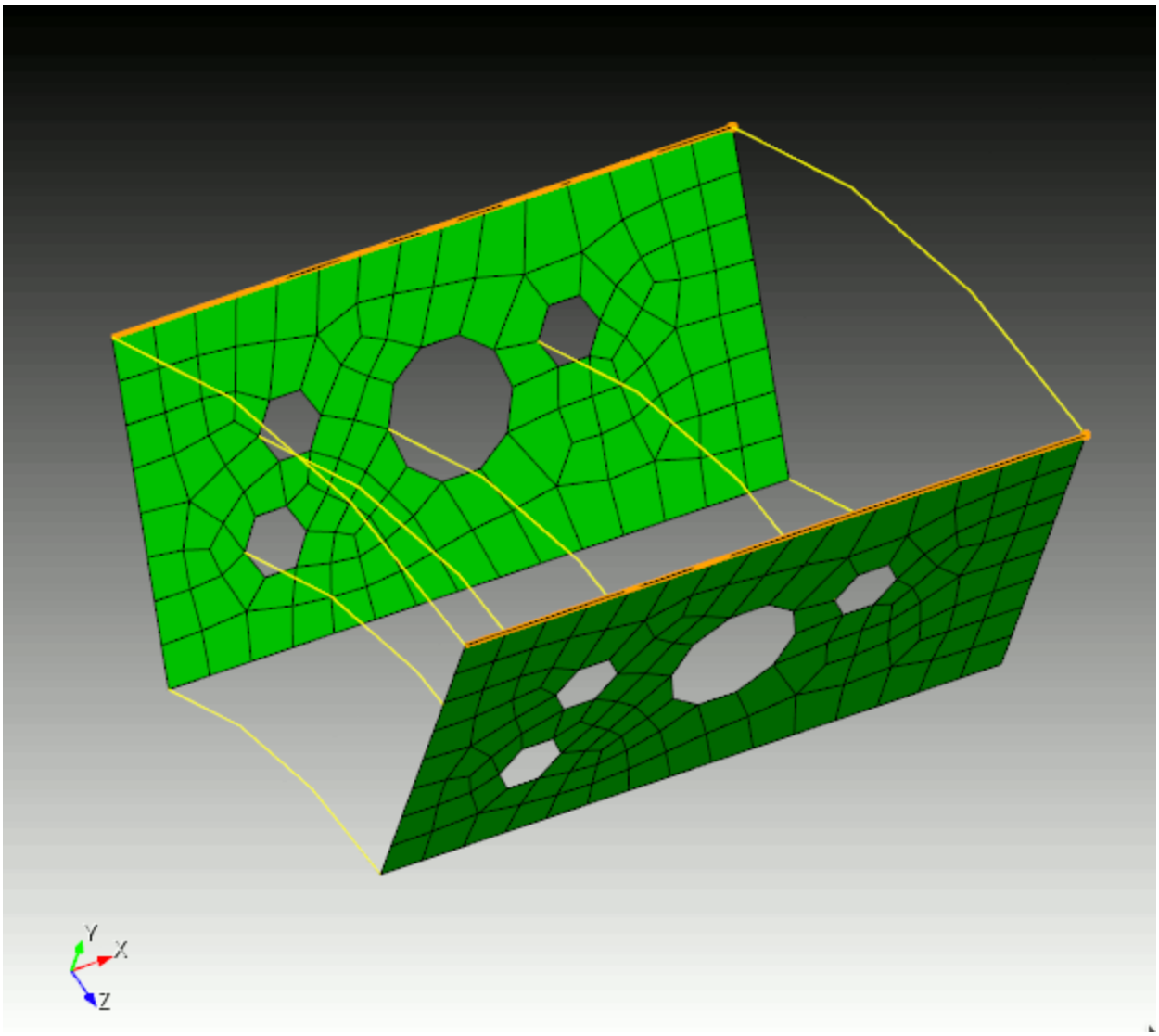
**Copy Mesh Volume <volume_id> Onto Volume <volume_id> [Source Vertex <vertex_id> Target Vertex <vertex_id> [Source Curve <curve_id> Target Curve <curve_id>] [Nosmoothing]**

**Related Commands**:

**Set Morph Smooth {on | off}**

**Discussion**:
If the user desires to copy the mesh from a surface, volume, curve, or set of curves that has already been meshed, the copy mesh scheme can be used. Note that this scheme can be set before the source entity has been meshed; the source entity will be meshed automatically, if necessary, before the mesh is copied to the target entity. The user has the option of providing orientation data to specify how to orient the source mesh on the target entity. For example, when copying a curve mesh, the user can specify which vertex on the source (the source vertex) gets copied to which vertex on the target (the target vertex). If you need to reference mesh entities for the copy, use the **Copy Mesh** commands. If no orientation data is specified, or if the data is insufficient to completely determine the orientation on the target entity, the copy algorithm will attempt to determine the remaining orientation data automatically. If conflicting, or inappropriate, orientation data is given, the algorithm attempts to discard enough information to arrive at a proper mesh orientation.
Curve mesh copying has certain options that allow the copying of just a section of the source curves' mesh. These options are accessed through the extra keyword options. The **percent** option allows the user to specify that a certain percentage of the source mesh be copied--in this context the auto keyword means that the percentage will be calculated based on the ratio of lengths of the source and target curves. The **combine** and **separate** keywords relate to how the command line options are interpreted. If the user wishes to specify a group of target curves that will each receive an identical copy of a source mesh, then the **target separate** option should be used (this is the default). If, however, the user wishes the source mesh to be spread out along the range of target

curves, then the **target combine** option should be used. The source curves are treated in a similar fashion.
Surface mesh copying with multiple holes in the surface may require matching up interior pair vertices. This will be required if the algorithm cannot match them up spatially. Interior pair vertices can be specified with the option **Interior pair vertex <id> <id> ...**
Volume mesh copying depends on the surface copying scheme. Because of this, the target volume must not have any of its surfaces meshed already.
An exact copy of the mesh may not always happen. Dissimilar geometry or smoothing may cause inexact copies. If the geometry is similar, the smoothing option may be turned off to get an exact copy of the mesh, by either specifying **Nosmoothing** or by omitting **Smooth**. If the geometry is dissimilar, the user may set the morph smoothing flag on, which will activate a special smoother that will match up the meshes as closely as possible.
**Example**:
As an example, the following copy is done with the command
**copy mesh surf 23 onto surf 14 source curve 1 source vertex 1 target curve 24 target vertex 20**
The source and target vertices match up, and are highlighted, while the source and target curves match up and are highlighted. Matching the source and target curves/vertices help define the orientation.




conversion

# *HTet*

**Applies to:** Volumes
**Summary:** Converts an existing hex mesh into a conforming tetrahedral mesh.
**Syntax:**

**HTet Volume <range> {UNSTRUCTURED | structured}**

**Discussion:**
Unlike other meshing schemes in this section, The HTet command requires an existing hexahedral mesh on which to operate. Rather than setting a meshing scheme for use

with the mesh command, the HTet command works after an initial hex mesh has been generated.
Two methods for decomposing a hex mesh into tetrahedra are available. Set the method to be used with the optional arguments unstructured and structured. The unstructured method is the default. Figure 1 shows the difference between the two methods:

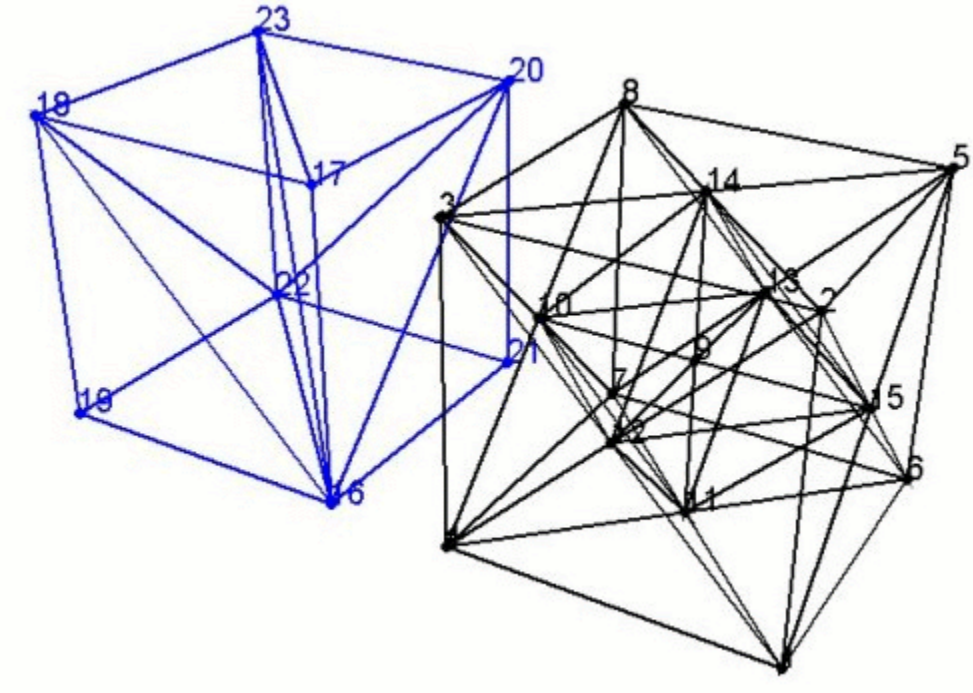



**Figure 1. Left: Unstructured method creates 6 tets per hex. Right: Structured method creates 28 tets per hex**

#### Unstructured

This method creates 6 tetrahedra for every hexahedra. No new nodes will be generated. The orientation of the 6 hexahedra will be based upon the element node numbering, as a result orientations may change if node numbering changes. This method is referred to as unstructured because the number of tetrahedra adjacent each node will be relatively arbitrary in the final mesh. Tetrahedral element quality is generally sufficient for most applications, however the user may want to verify quality before performing analysis.

#### Structured

With this approach, 28 tetrahedra are generated for every hexahedra in the mesh. This method adds a node to each face of the hex and one to the interior. Although this method generates significantly more elements, the orientation and quality of the resulting tetrahedra are more consistent. Each previously existing interior node in the mesh will have the same number of adjacent tetrahedra.

# *QTri*

**Applies to**: Surfaces
**Summary**: Meshes surfaces using a quadrilateral scheme, then converts the quadrilateral elements into triangles.
**Syntax**:

**Surface <range> Scheme Qtri [Base Scheme quad_scheme>]**

**QTri { Surface <range> | Face <range> }**

**Set QTri Split [2|4]**

**Set QTri Test {Angle|Diagonal}**

**Discussion**:
QTri is used to mesh surfaces with triangular elements. The surface is, first, meshed with the quadrilateral scheme, and, then, the generated quads are split along a diagonal to produce triangles. The first command listed above sets the meshing scheme on a surface to QTri. The second form sets the scheme and generates the mesh in a single step.
In the first command, the user has the option of specifying the underlying quadrilateral meshing scheme using the base scheme <quad_scheme> option. If no base scheme is specified, CUBIT will automatically select a scheme. For non-periodic surfaces, the base scheme will be set to scheme pave. For periodic surfaces, the base scheme will be set to scheme map.
Generally, the second command, Qtri Surface <range>, is used on surfaces that have already been meshed with quadrilaterals. If, however, this command is used on a surface that has not been meshed, a base scheme will automatically be selected using CUBIT's auto-scheme capabilities. The user can over-ride this selection by specifying a quadrilateral meshing scheme prior to using the qtri command (using the Surface <range> Scheme <quad_scheme> command). QTri may also be performed on quadrilateral elements on a surface or a subset of quadrilateral elements on a surface. To split existing quadrilaterals, the QTri command can be given a list of faces.
In addition to the default 2 tris per quad, the set qtri split command may alter the QTri scheme so that it will split the quad into 4 triangles per quad. Where the 4 option is used, an additional mesh node is placed at the centroid of each quad.
There are two methods that may be used to calculate the best diagonal to use for splitting the quadrilateral elements: angle or diagonal. The angle measurement uses the largest angle, while the diagonal option uses the shortest diagonal. The largest angle measurement will be more accurate but takes more time.
Also, the QTri scheme is used in the TriMesh command as a backup to the TriAdvance triangle meshing scheme.

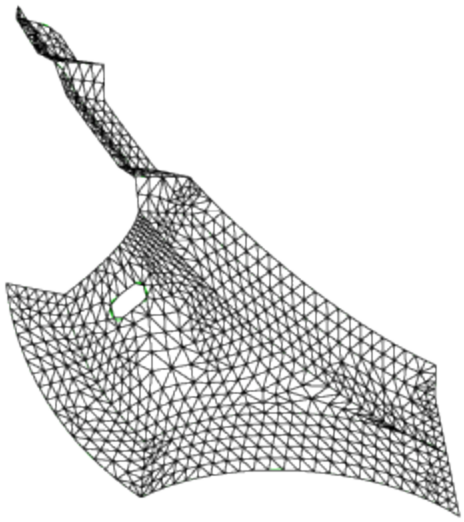


**Figure 1. Surface meshed with scheme QTri**

# *THex*

**Applies to**: Volumes
**Summary**: Converts a tetrahedral mesh into a hexahedral mesh.
**Syntax**:

```
THex Volume <range>
```

**Discussion**:
The THex command splits each tetrahedral element in a volume into four hexahedral elements, as shown in Figure 1. This is done by splitting each edge and face at its midpoint, and then forming connections to the center of the tet.
When THexing merged volumes, all of the volumes must be THexed at the same time, in a single command. Otherwise, meshes on shared surfaces will be invalid. An example of the THex algorithm is shown in Figure 2.

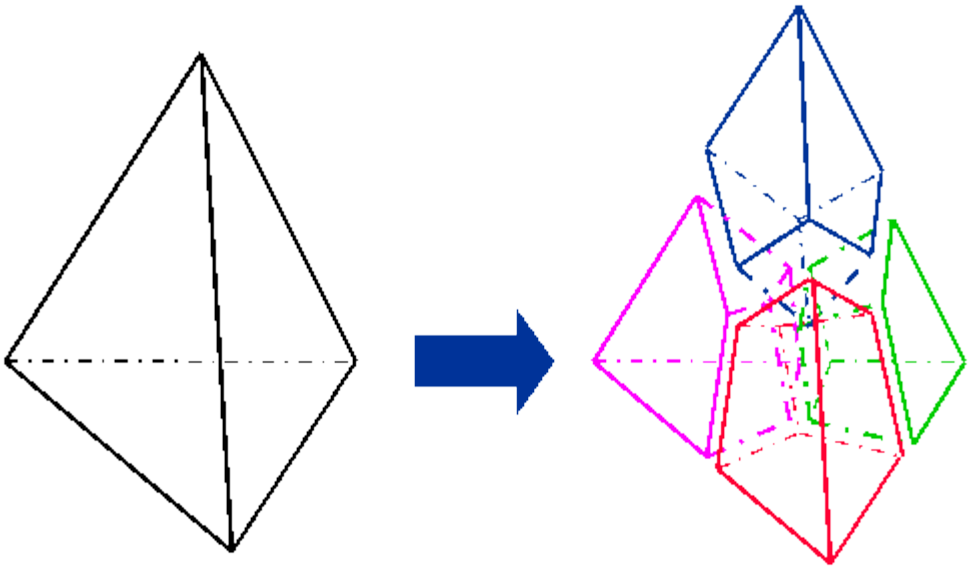
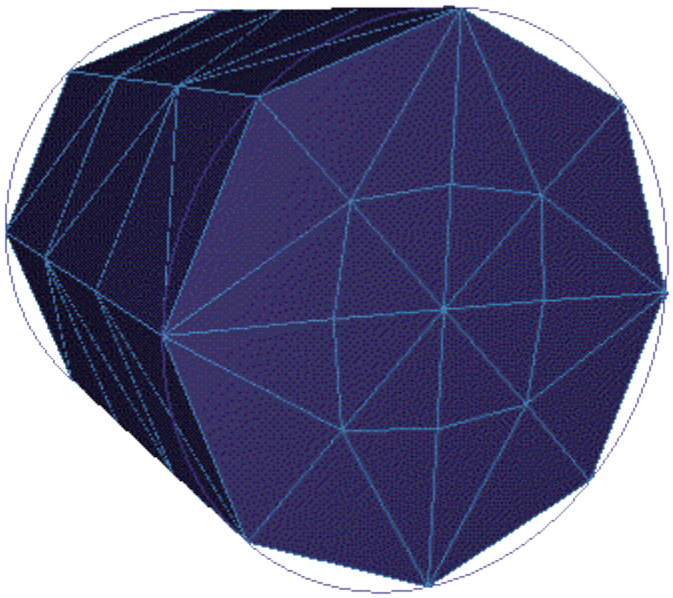


**Figure 1. Conversion of a tetrahedron to four hexahedra, as performed by the THex algorithm.**

.



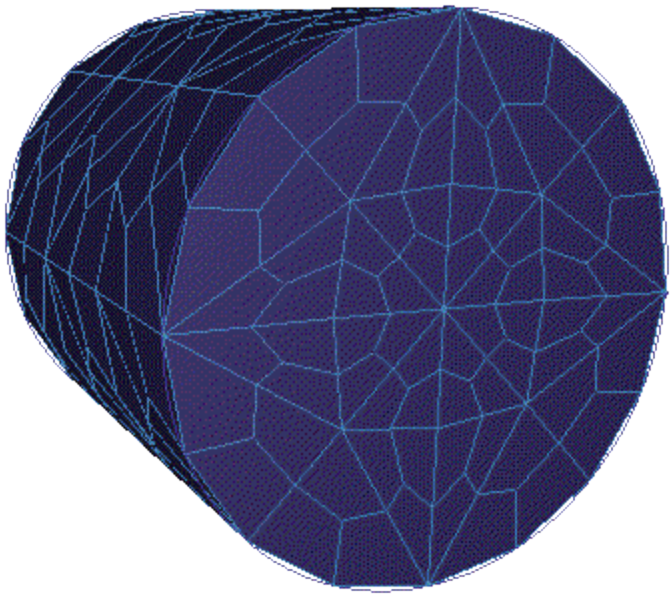


**Figure 2. A cylinder before and after the THex algorithm is applied.**

# *TQuad*

**Applies to**: Surfaces
**Summary**: Converts a triangular surface mesh into a quadrilateral mesh.
**Syntax**:

```
TQuad Surface <range>
```

**Discussion**:
The TQuad command splits each triangular surface element in four quadrilateral elements, as shown in Figure 1. This is done by splitting each edge at its midpoint, and then forming connections to the center of the triangle. The result is the same as using the THex algorithm, but only applies to surfaces. In general it is better to use a mapped or paved mesh to generate quadrilateral surface meshes. However, the TQuad scheme may be useful for converting facet-based triangular meshes to quadrilateral meshes when remeshing is not possible.

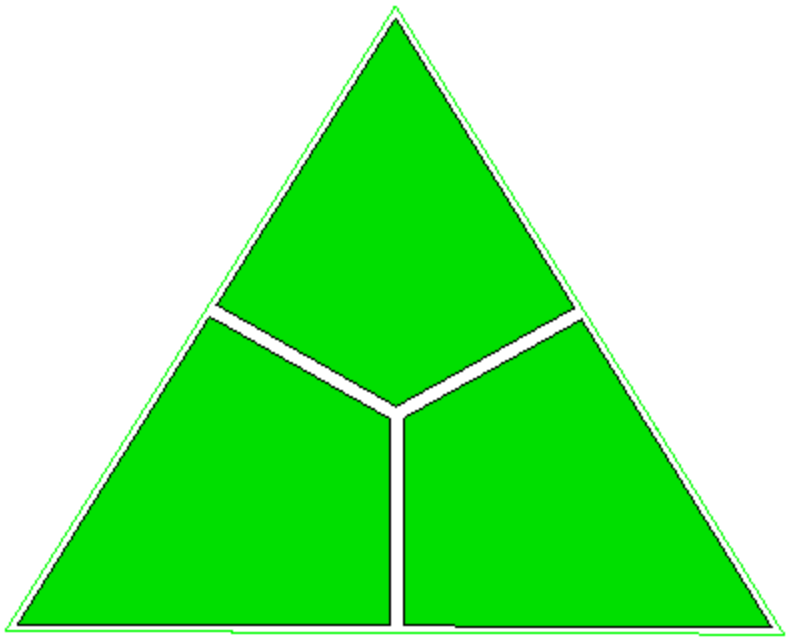


**Figure 1. A triangle split into 3 quads using the TQuad scheme**

traditional

# *Bias, Dualbias*

**Applies to**: Curves
**Summary**: Meshes a curve with node spacing biased toward one or both curve ends.
**Syntax**:

**Curve <range> Scheme Bias**

**Curve <range> Scheme Bias {Factor|First_Delta|Fraction} <double> [Start Vertex <range>] [preview]**

**Curve <range> Scheme Dualbias {Factor|First_Delta|Fraction} <double> [preview]**

**Curve <range> Scheme Bias Fine Size <double>**
**{Coarse Size <double> | Factor <double>} [Start Vertex <range>] [preview]**

**Curve <range> Scheme Dualbias Fine Size <double>**
**{Coarse Size <double> | Factor <double>} [preview]**

**Curve <range> Scheme Multi_bias Start <size>**
**[Fraction <value> <size>]... End <size>**
**[Start Vertex <id>][Respect_intervals][preview]**

**Related Commands**:

**Curve <range> Reverse Bias**

**Set Maximum Interval <int>**

**See also Surface Sizing Function Type Bias**

**See also Curve Scheme Stretch**

The main differences between scheme bias and stretch are the following: scheme stretch does not use strict geometric series for node placement. If you specify scheme bias or dualbias using the "fine size" form, the interval count will be hard-set to a value that fills in the curve.

**Auto Bias**

When using the command '**curve <range> scheme bias**' with no additional parameters, an auto setting will be enabled by default for tet and tri meshing. This scheme honors sizes at a curve's vertices and that vertex size will be used to create a biased edge mesh. For example, two volumes with different sizes set on the volumes are merged. The size at the vertices (averaged from sizes on the parent entities) will be used to create the biased edge mesh.

A user can set a size on a vertex with the following command:

**Vertex <id> Size <size>**

**More Discussion**:

The Bias and DualBias schemes space the curve mesh unequally, placing more nodes towards (or away from) the ends of the curve according to a geometric progression. The ratio of successive edges is the "factor," which may be greater than or less than one. For bias, the series starts at the first vertex of the curve, or the "start vertex" if specified. For dualbias, the series starts at both ends of the curve and meets in the middle.

The command behaves differently depending on which set of parameters are specified. There are three basic variables: the interval count, the bias factor, or the first edge size. The curve length is a given, fixed quantity. The user can specify any two of these variables, and the third will be automatically determined.

If the "{Factor|First_Delta|Fraction}" form is specified, then the interval count is taken as a given. The interval count is whatever was specified previously by an interval count or size command (see Interval Assignment). If "Factor" is specified, then the first edge size will be automatically chosen so that the geometric progression of edges "fit" onto the curve. If "first_delta" is specified, then the first edge length is exactly that absolute value, and the "factor" is automatically chosen. If "fraction" is specified, then the first edge length is the curve length times that fraction, and again the "factor" is automatically chosen.

If the "fine size" is specified, then the first edge length is exactly that absolute value. If the "factor" is specified, then the interval count is automatically chosen. If an approximate coarse size is specified, then this also determines the factor, and again the interval count is automatically chosen. If a surface sizing function type bias is used, then the curves of the surface are sized using similar formulas.
If no start or end vertex is specified, the curve's start vertex is used as the starting point of the bias. (A curve's start vertex can be identified by listing the curve from the "CUBIT>" prompt.)
If a curve needs to have its nodes distributed towards the opposite end, it can be easily edited using the reverse bias command. Reversing the curve bias using this command is equivalent to setting a bias factor equal to the inverse of the original bias factor. Reversing the bias can be performed on both meshed and unmeshed curves.
The maximum interval setting allows the user to set a maximum number of intervals on any bias curve. This value is doubled for a curve with a dualbias scheme. It can be easy to accidentally specify a very large number of intervals and this setting allows the user to place an upper limit the number of intervals.
The preview option will allow the user to preview mesh size and distribution on the curve before meshing.
The following figure shows the result of meshing edges with equal, bias and dualbias schemes.

**Multi Bias**

The multi-bias scheme allows several biases to be created on a single curve by specifying desired sizes at multiple locations along the curve (see Figure 1 below). The "start" and "end" sizes must be specified, and any number of fraction-size pairs may be specified, where the fraction value is between 0 - 1. If "Start Vertex" is given, the specified vertex is considered to have the fraction value of 0, with the opposite vertex having the fraction value of 1. If "respect_intervals" is not specified, the scheme will choose an appropriate number of intervals for the curve based on the given sizes.

**Figure 1. Curve with scheme multi_bias.**

If the "respect_intervals" option is given, the multi-bias scheme will try to get as close to the desired sizes as possible, but will always respect the number of intervals set on the curve and adjust sizing as necessary (see Figure 2).

**Figure 2. Same curve as in Figure 1, but with the respect_intervals option. Note the areas of relatively dense mesh correspond to the dense mesh in the original.**

# *Circle*

**Applies to**: Surfaces
**Summary**: Produces a circle-primitive mesh for a surface

**Syntax**:

**Surface <range> Scheme [Sector] Circle [Interval <int>] [fraction <double>]**

**Discussion**:
The Circle scheme is used in regions that should be meshed as a circle. A "circle" consists of a single loop of bounding curves containing an even number of intervals. Thus, the circle scheme can be applied to circles, ellipses, ovals, and regions with "corners" (e.g. polygons). The bounding curves should enclose a convex region. Non-planar bounding loops can also be meshed using the circle primitive provided the surface curvature is not too great. The mesh resembles that obtained via polar coordinates except that the cells at the "center" are quadrilaterals, not triangles. See Figure 1 for an example of a circle mesh. Radial grading of the mesh may be achieved via the optional [intervals] input parameter. The Fraction option has the range 0 < fraction < 1 and defaults to 0.5. Fraction determines the size of the inner portion of the circle mesh relative to the total radius of the circle. The sector option was added to revert to legacy behavior which is not recommended.

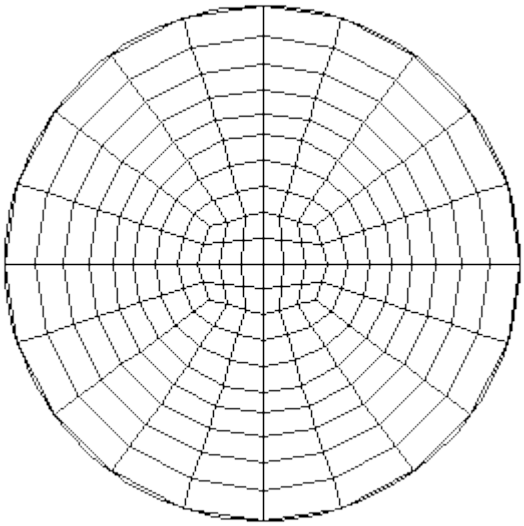


**Figure 1. Circle Primitive Mesh**

# *Curvature*

**Applies to**: Curves
**Summary**: Meshes curves by adapting the interval size to the local curvature.
**Syntax**:

**Curve <range> Scheme Curvature <double>**

**Discussion**:
The value of <double> controls the degree of adaptation. If zero, the resulting mesh will have nearly equal intervals. If greater than zero, the smallest intervals will correspond to the locations of largest curvature. If less than zero, the largest intervals will correspond to the locations of largest curvature. The default value of <double> is zero. Straight lines and circular arcs will produce meshes with near-equal intervals. The method for generating this mesh is iterative and may sometimes not converge. If the method does not converge, either the <double> is too large (over-adaptation) or the number of intervals is too small. Currently, the scheme does not work on periodic curves.

# *Equal*

**Applies to**: Curves
**Summary**: Meshes a curve with equally-spaced nodes
**Syntax**:

**Curve <range> Scheme Equal**

**Discussion**:
See Interval Assignment for a description of how to set the number of nodes or the node spacing on a curve.

# *Hole*

**Applies to**: Annular Surfaces
**Summary**: Useful on annular surfaces to produce a "polar coordinate" type mesh (with the singularity removed).
**Syntax**:

**Surface <surface_id_range> Scheme Hole [Rad_intervals <int>] [Bias <double>] [Pair Node <id> With Node <id>]**

**Discussion**:
A polar coordinate-like mesh with the singularity removed is produced with this scheme. The azimuthal coordinate lines will be of constant radius (unlike scheme map) The number of intervals in the azimuthal direction is controlled by setting the number of intervals on the inner and outer bounding loops of the surface (the number of intervals must be the same on each loop). The number of intervals in the radial direction is controlled by the user input, rad_intervals (default is one).
A bias may be put on the mesh in the radial direction via the input parameter bias. The default bias of 0 gives a uniform grading, a bias less than zero gives smaller radial

intervals near the inner loop, and a bias greater than zero gives smaller radial intervals near the outer loop.
The correspondence between mesh nodes on the inner and outer boundaries is controlled with the pair node "<loop node-id> with node <loop node-id>" construct. One id on the inner loop and one id on the outer loop should be given to connect the two nodes by a radial mesh line. Not choosing this option may result in sub-optimal node pairings with possible negative Jacobians. To use this option, mesh the inner and outer curve loops and then determine the mesh node ids.

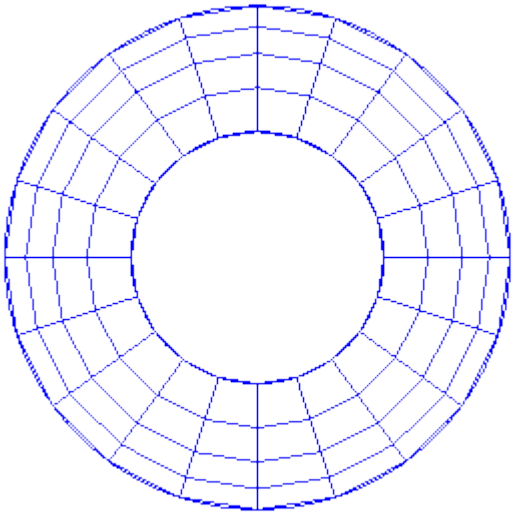


**Figure 1. Example of Hole Scheme**

# *Mapping*

**Applies to**: Surfaces, Volumes
**Summary**: Meshes a surface/volume with a structured mesh of quadrilaterals/hexahedra.
**Syntax**:

**Volume <range> Scheme Map**

**Surface <range> Scheme Map [Direction {Options}]**

**Discussion**:
A structured mesh is defined as one where each interior node on a surface/volume is connected to 4/6 other nodes. Mappable surfaces contain four logical sides and four logical corners of the map; each side can be composed of one or several geometric curves. Similarly, mappable volumes have six logical sides and eight logical corners; each side can consist of one or several geometric surfaces. For example, in Figure 1 below,

the logical corners selected by the algorithm are indicated by arrows. Between these vertices the logical sides are defined; these sides are described in Table 1.

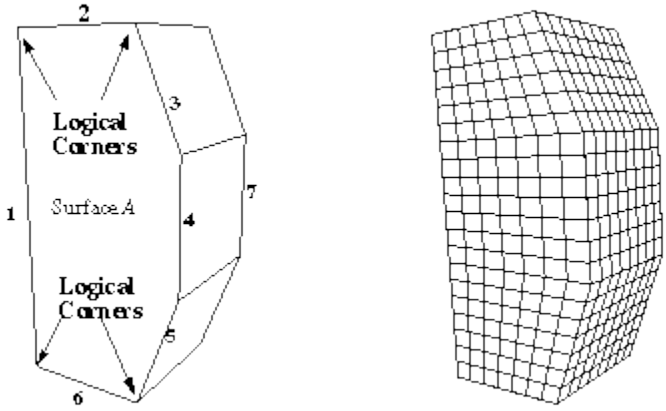



**Figure 1. Scheme Map Logical Properties**

**Table 1. Listing of Logical Sides**

| Logical Side | Curve Groups |
|---|---|
| Side 1 | Curve 1 |
| Side 2 | Curve 2 |
| Side 3 | Curve 3, Curve 4, Curve 5 |
| Side 4 | Curve 6 |

Interval divisions on opposite sides of the logical rectangle are matched to produce the mesh shown in the right portion of Figure 1. (i.e. The number of intervals on logical side 1 is equated to the number of intervals on logical side 3). The process is similar for volume mapping except that a logical hexahedron is formed from eight vertices. Note that the corners for both surface and volume mapping can be placed on curves rather than vertices; this allows mapping surfaces and volumes with less than four and eight vertices, respectively. For example, the mapped quarter cylinder shown in Figure 2 has only five surfaces.

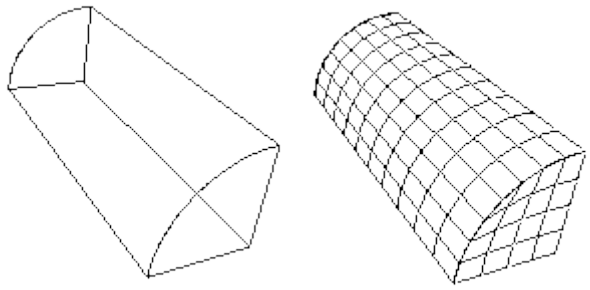


**Figure 2. Volume Mapping of a 5-surfaced volume**

The mapper works on a bicubic interpolation of the points on the boundary to represent the surface. There may be times that those points may not be on the surface exactly if the surface is not suitable for bicubic interpolation. The Mapping Constraint flag tells the mapper to relax the nodes to the geometry or not.

**Set Mapping Constraint {ON|off}**

When on, the mapping constraint relaxes the node to the nearest point on the geometry. In some situations, the nearest point might be incorrect for the intent of the mesh. To help the mesher find the correct location, a projection direction may optionally be specified for surfaces.

**Surface <range> Scheme Map [Direction {Options}]**

If a projection direction is specified, the nodes are moved to the geometry in a straight line along the given direction. The direction can be specified using any of the direction options.

# *Pave*

**Applies to**: Surfaces
**Summary**: Automatically meshes a surface with an unstructured quadrilateral mesh.
**Syntax**:

**Surface <range> Scheme Pave Related Commands:**

**[Set] Paver Diagonal Scale <factor (Default = 0.9)> [set] Paver Grid Cell <factor (Default = 2.5)>[set] Paver LinearSizing {Off | ON} Surface <range> Sizing Function Type ...**

**[Set] Paver Smooth Method {DEFAULT | Smooth Scheme | Old}**

**[Set] Paver Cleanup {ON|Off|Extend}**

**Discussion**:
Paving (Blacker, 91; White, 97) allows the meshing of an arbitrary three-dimensional surface with quadrilateral elements. The paver supports interior holes, arbitrary boundaries, hard lines, and zero-width cracks. It also allows for easy transitions between dissimilar sizes of elements and element size variations based on sizing functions. Figure 1 shows the same surface meshed with mapping (left) and paving (right) schemes using the same discretization of the boundary curves.

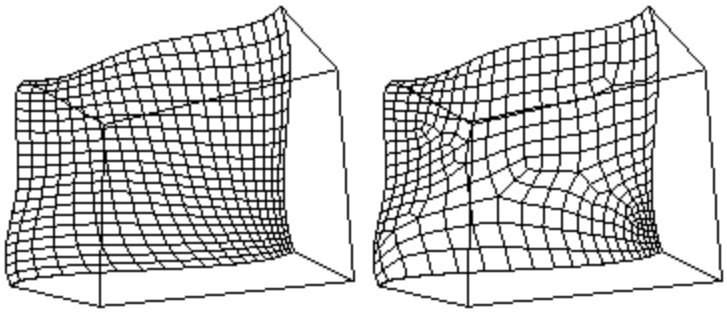


Figure 1. Map (left) and Paved (right) Surface Meshes

#### Element Shape Improvement

When meshing a surface geometry with paving, clean-up and smoothing techniques are automatically applied to the paved mesh. These methods improve the regularity and quality of the surface mesh. By default the paver uses its own smoothing methods that are not directly-callable from CUBIT. Using one of CUBIT's callable smoothing methods in place of the default method will sometimes improve mesh quality, depending on the surface geometry and specific mesh characteristics. If the paver produces poor element quality, switching the smoothing scheme may help. This is done by the command:

**[set] Paver Smooth Method {DEFAULT | Smooth Scheme | Old}**

When the "Smooth Scheme" is selected, the smoothing scheme specified for the surface will be used in place of the paver's smoother. See "Mesh Smoothing" for more information about the available smoothing schemes in CUBIT.

#### Controlling Flattening of Elements

The smoothers flatten elements, such as inserted wedges, that have two edges on the active mesh front. In meshes where this "corner" is a real corner, flattening the element may give an unacceptable mesh. The following command controls how much the diagonal of such an element is able to shrink.

**[set] Paver Diagonal Scale <factor (Default = 0.9)>**

The range of for the scale factor is 0.5 to 1.0. A scale factor of 1.0 will force the element to be a parallelogram as long as it is on the mesh front. A value of 0.5 will allow the diagonal to be half its calculated length. The element may became triangular in shape with the two sides on the mesh front being collinear.

#### Controlling the Grid Search for Intersection Checking

The paver divides the bounding box of a surface into a number of cells based on the average length of an element. It uses these cells to speed intersection checking of new element edges with the existing mesh. If both very long and very short edges fall in the same area, it is possible that a long edge which spans the search region is excluded from

the intersection check when it does intersect the new element. The following command allows the user to adjust the size of the grid cells.

**[set] Paver Grid Cell <factor (Default = 2.5)>**

The grid cell factor is a multiplier applied to the average element size, which then becomes the grid cell size. The surface's bounding box is divided by this cell size to determine the number of cells in each direction. A larger cell size means each cell contains more nodes and edges. A smaller cell size means each cell has fewer nodes and edges. A larger cell size forces the intersection algorithm to check more potential intersections, which results in long paver times. A smaller cell size gives the intersection algorithm few edges to check (faster execution) but may result in missed intersections where the ratio of long to short element edges is great. Increase this value if the paver is missing intersections of elements.

#### Controlling the Paver Sizing Function

The paving algorithm will automatically select a "linear" sizing function if the ratio the largest element to the smallest is greater than 6.0 and no other sizing function is specified for the surface. This is usually desirable. When it is not, the user can change this behavior with the command:

**[set] Paver LinearSizing {Off | ON}**

Setting paver linear sizing to "off" will keep the default behavior. The size of the element will be based on the side(s) of the element on the mesh front. For a discussion of sizing functions, including how to automatically set up size transitions, see Adaptive Meshing.

#### Controlling Paver Cleanup

The paver uses a mesh clean-up process to improve mesh quality after the initial paving operation. Clean-up applies local connectivity corrections to increase the number of interior mesh nodes that are connected to four quadrilaterals. Sometimes it fails to improve the mesh. The following command allows the user to control some aspects of the clean-up process.

**[Set] Paver Cleanup {ON|Off|Extend}**

The default option is to clean-up the mesh. The off option will turn clean-up off and may give an invalid mesh. The extend option enables a non-local topology replacement algorithm. The command without any option will list the current setting.
The extend option attempts to group several defective nodes in a region that may be replaced with a template that has fewer defects. The images below show a mesh before and after using this option.

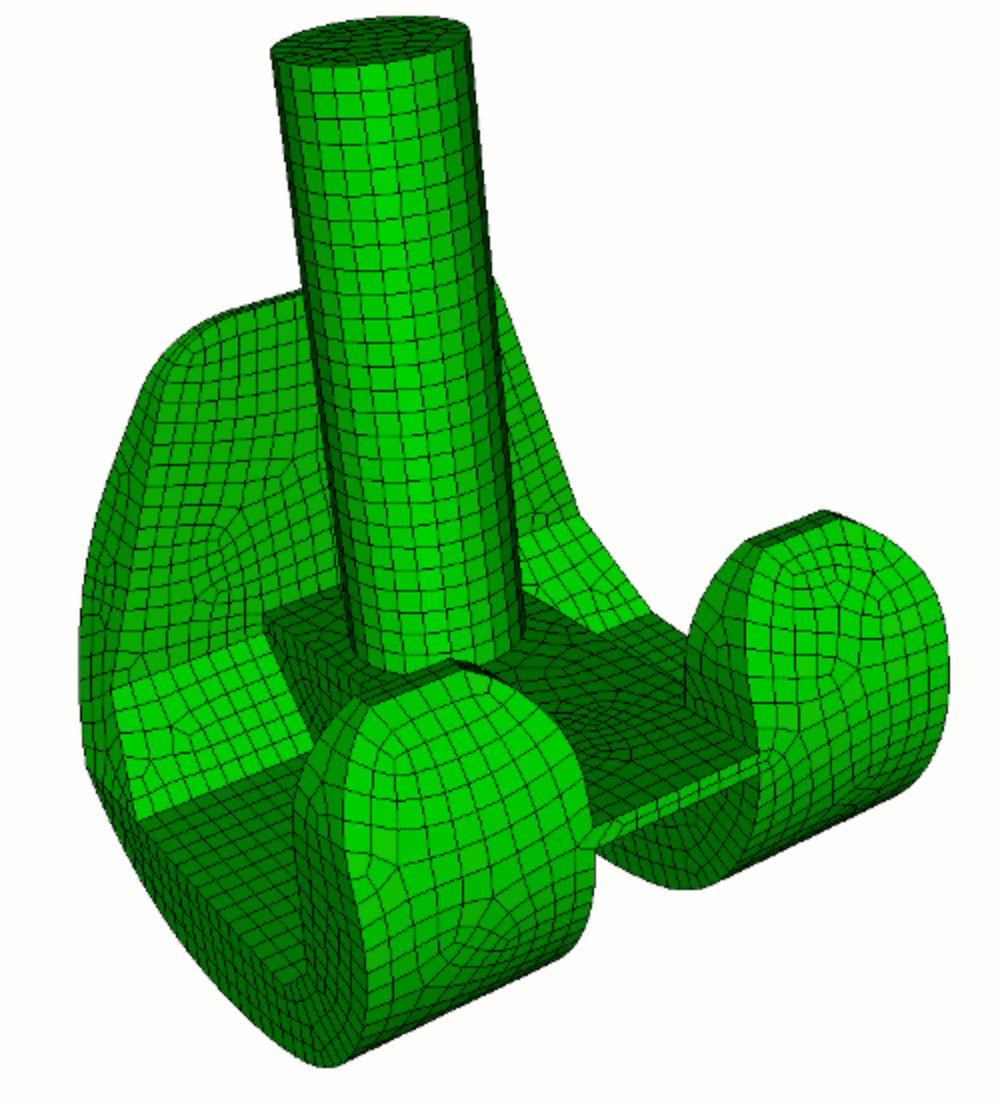


**Figure 2. Paved mesh before using cleanup extend**

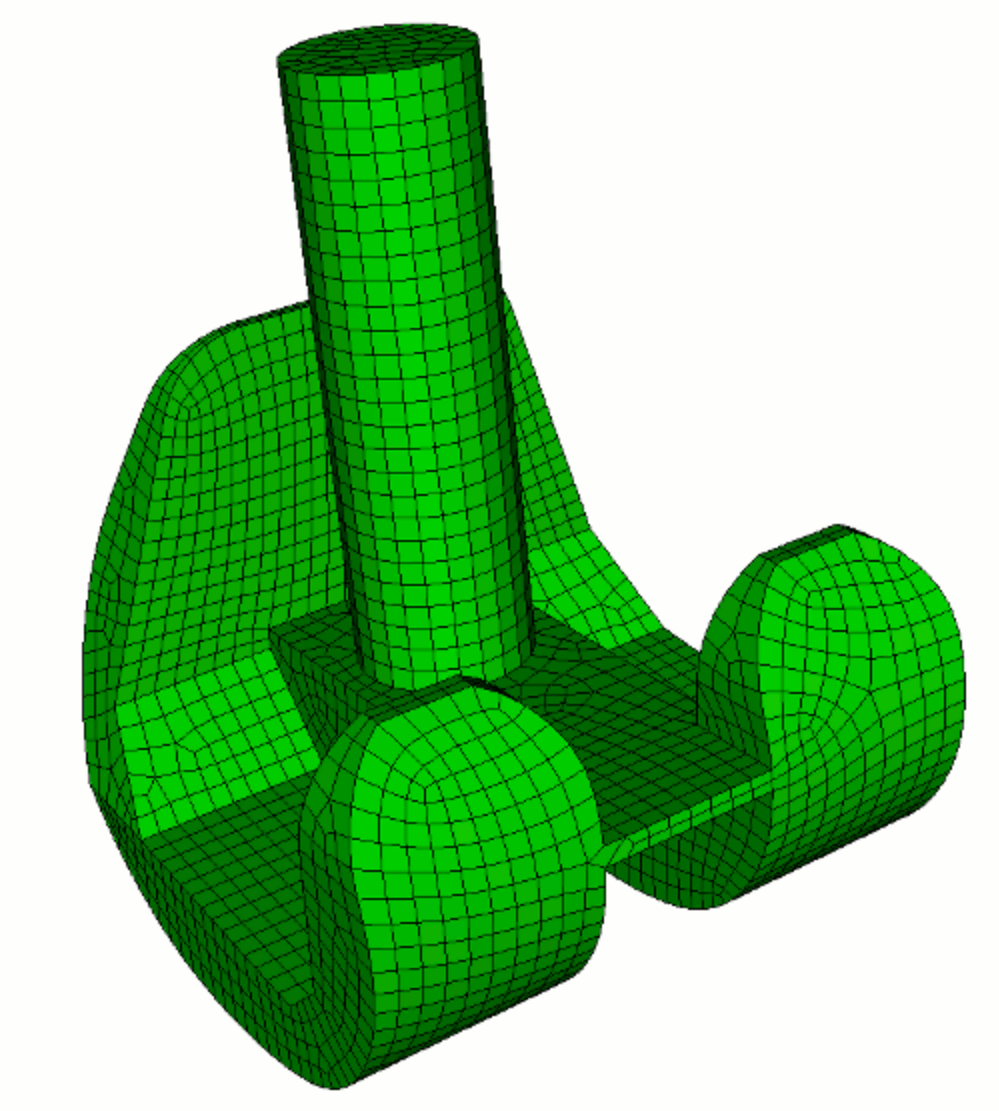


**Figure 3. Paved mesh after using cleanup extend**

# *Pentagon*

**Applies to**: Surfaces
**Summary**: Produces a pentagon-primitive mesh for a surface
**Syntax**:

```
Surface <range> Scheme Pentagon
```

**Discussion**:

The pentagon scheme is a meshing primitive for 5-sided regions. It is similar to the triprimitive and polyhedron schemes, but is hard-coded for 5 sided surfaces.
The pentagon scheme indicates the region should be meshed as a pentagon. The scheme works best if the shape has 5 well-defined corners; however shapes with more corners can be meshed. The algorithm requires that there be at least 10 intervals (2 per side) specified on the curves representing the perimeter of the surface. In addition, the sum of the intervals on any three connected sides must be at least two greater than the sum of the intervals on the remaining two sides. Figure 1 shows two examples of pentagon meshes.

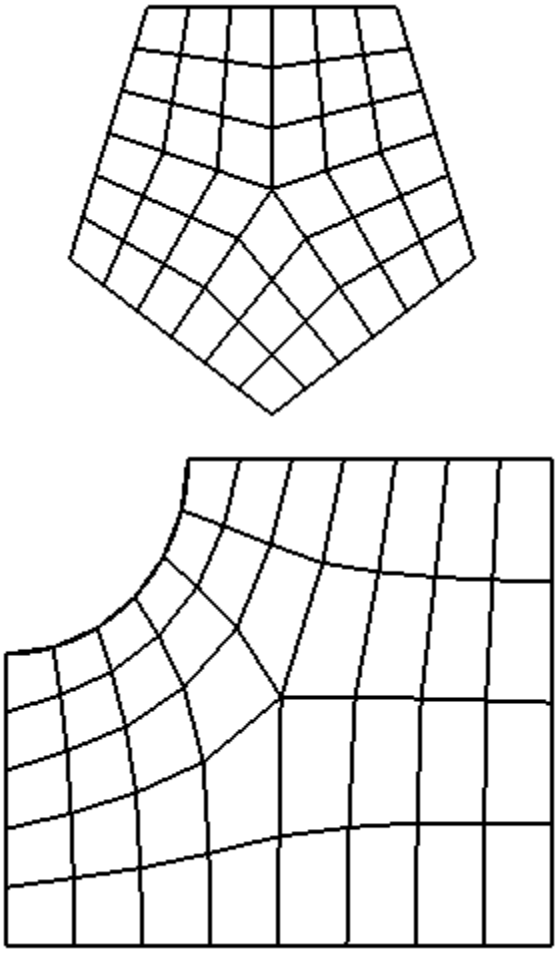


**Figure 1. Examples of Pentagon Scheme Meshes**

# *Pinpoint*

**Applies to**: Curves
**Summary**:Meshes a curve with node spacing specified by the user.

**Syntax:**

**Curve <range> Scheme Pinpoint Location <list of doubles>**

**Discussion**:
The **Pinpoint** scheme allow the user to specify exactly where on a curve to place nodes. The list of doubles are absolute positions, measured from the start vertex. The user can enter as many as needed, and they do not need to be in numerical order. Below is an example of a curve that has been meshed using the following scheme:

**curve 2 scheme pinpoint location 1 4 5 6 6.2 6.4 6.6 9:**



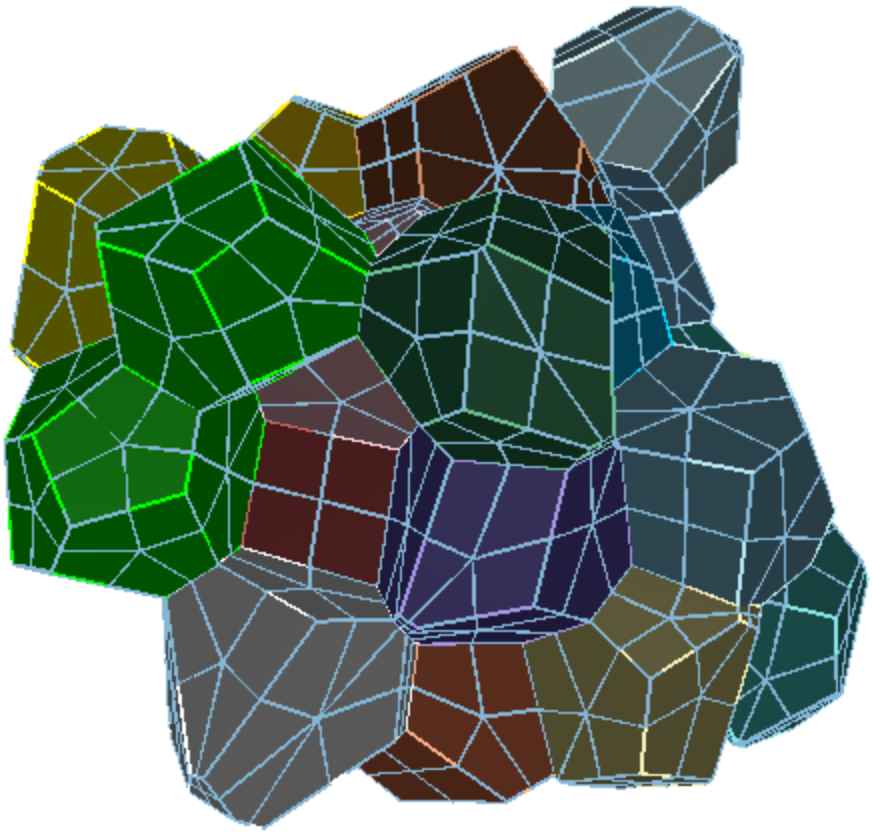
# *Polyhedron*

**Applies to**: Surfaces and Volumes.
**Summary**: Produces an arbitrary-sided block primitive mesh for a surface or volume.
**Syntax**:

**Volume <range> Scheme Polyhedron**

**Surface <range> Scheme Polyhedron**

**Discussion**:
The polyhedron scheme is a meshing primitive for 2d and 3d n-sided regions. This is similar to the triprimitive , tetprimitive, and pentagon schemes, except rather than 3, 4, or 5 sides, it allows an arbitrary number of sides. The scheme works best on convex regions. Surfaces must have only one loop, and each vertex must be connected to exactly two curves on the surface (e.g., no hardlines). Volumes must have only one shell, each vertex must be connected to exactly three surfaces on the volume, and each surface should be meshed with scheme polyhedron. There are some interval assignment requirements as well, which should be automatically handled by CUBIT.
If the polyhedron scheme is specified for the volume, then the surfaces of the volume are automatically assigned scheme polyhedron as well, unless they were hard-set by the user. Schemes should be specified on all volumes of an assembly prior to meshing any of them. Scheme polyhedron attaches extra data to volumes; if Cubit is behaving strangely, the user may need to explicitly remove that data with a **reset volume all**, or similar command.
Scheme polyhedron was designed for assemblies of material grains, where each volume is roughly a Voronoi region, and the assembly is a periodic space-filling model (tile). Figure 1 shows two examples of polyhedron meshes.



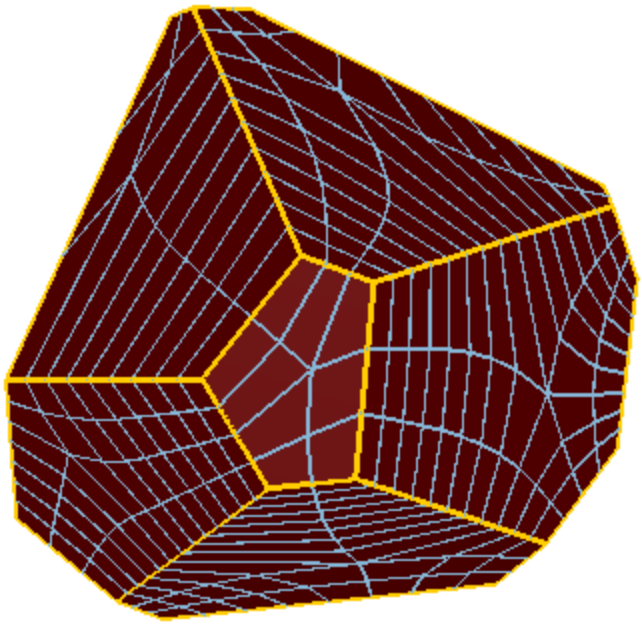


**Figure 1. Examples of Polyhedron Scheme Meshes**

# *Sphere*

**Applies to**: Volumes topologically equivalent to a sphere and having one surface.
**Summary**: Generates a radially-graded hex mesh on a spherical volume.
**Syntax**:

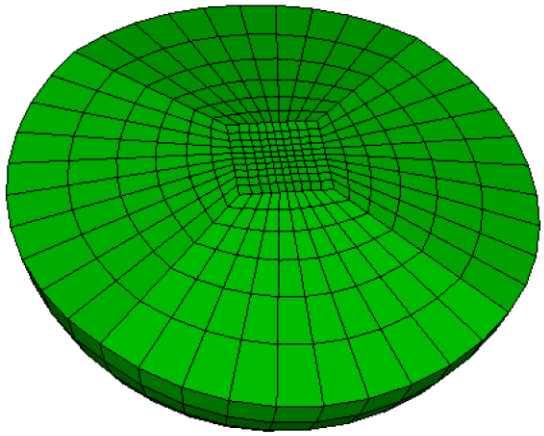
**Volume <range> Scheme Sphere [Graded_interval <int>] [Az_interval <int>] [Bias <val>] [Fraction <val>] [Max_smooth_iterations <int=2>]**

**Discussion**:
This scheme generates a radially-graded mesh on a spherical volume having a single bounding surface. The mesh is a straightforward generalization of the circle scheme for surfaces. The mesh consists of an inner region and an outer region. The inner region is a mapped mesh of a cube and the outer region contains fronts that trasition from the cube surface to the sphere surface. The following describes the parameters that control the sphere mesh.
**Graded_interval**:
The number of intervals in the outer region from the inner mapped mesh to the surface of the sphere is controlled by the **graded_interval** input parameter. Azimuthal mesh lines in the outer portion of the sphere will have approximately constant radius. If **graded_interval** is not specified, a default number of intervals will be computed based on the interval size value assigned to the sphere volume.
**Az_interval**:
The number of azimuthal intervals around the equator is controlled by the **az_interval** input parameter. To maintain symmetry, the **az_interval** will be rounded to the nearest multiple of 8.
If **az_interval** is not specified, a default number of intervals will be computed either based on the the the interval value or on the mesh size value assigned to the volume. If the interval value is set (**volume 1 interval 40**, for example), the interval value will be used to define the number of azimuthal intervals. Otherwise, the mesh size will be used as the approximate size for elements on the inner mapped mesh.
**Bias**:
The **bias** parameter controls the amount of radial grading in the outer region of the mesh from the inner mapped mesh to the sphere surface. A **bias = 1** will results in equal size intervals, while a **bias < 1** will generate smaller intervals towards the sphere interior and a **bias > 1** will generate smaller elements towards the sphere surface. If the **bias** parameter is not specified, a default **bias** will be computed so that element size gradually increases from the inner mapped mesh to the sphere surface. The default **bias** value will also be based on the interval size assigned to the sphere volume as it attempts to maintain approximately isotropic elements throughout the sphere.
**Fraction**:

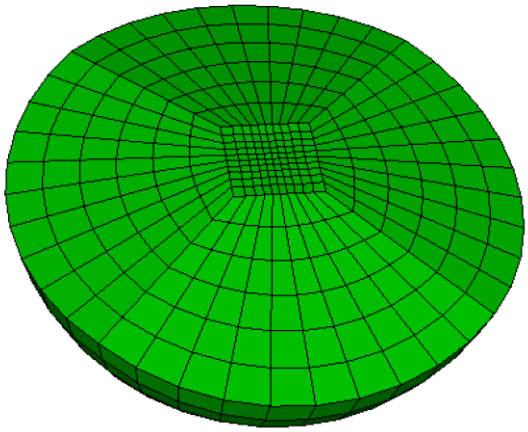
The **fraction** parameter (between 0 and 1) determines what fraction of the sphere is occupied by the inner mapped mesh. The **fraction** is defined as ratio of the diagonal of the cube containing the mapped mesh to sphere's diameter. The default value for **fraction** is 0.5. Interval sizes in the inner mapped mesh are normally constrained by the **az_intervals**. If **az_intervals** are not specified, element sizes in this region will be based upon the interval size assigned to the sphere volume.

**Max_smooth_iterations**:

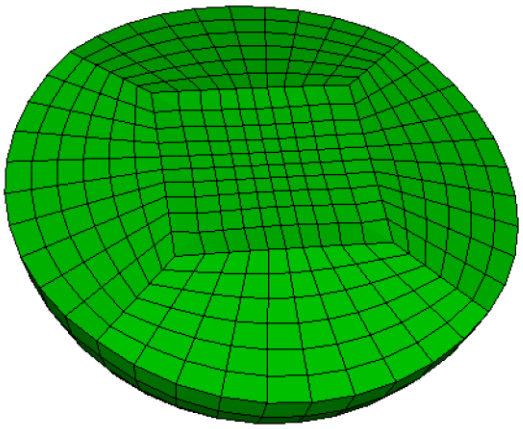
The **Max_smooth_iterations** parameter determines the number of smoothing iterations following initial definition of the sphere mesh. By default, the number of smoothing interations is set to 0, which will result in a symmetric mesh. Note that smoothing can improve the quality of the mesh, however, it may disturb the bias and fraction. When bias and fraction are critical then smoothing iterations should be set to 0.



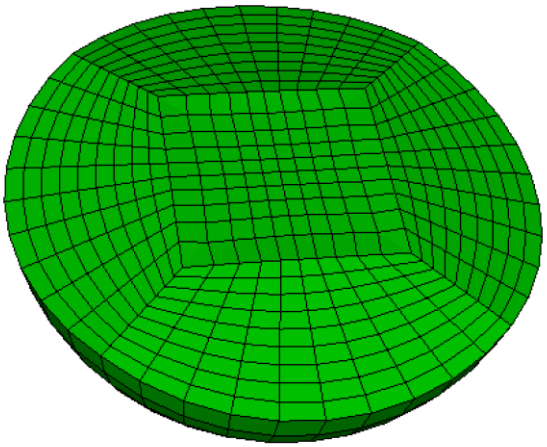
**SPHERE MESH: fraction 0.3 graded_interval 6 az_interval 40 bias 0.8 max_smooth_iterations 0**



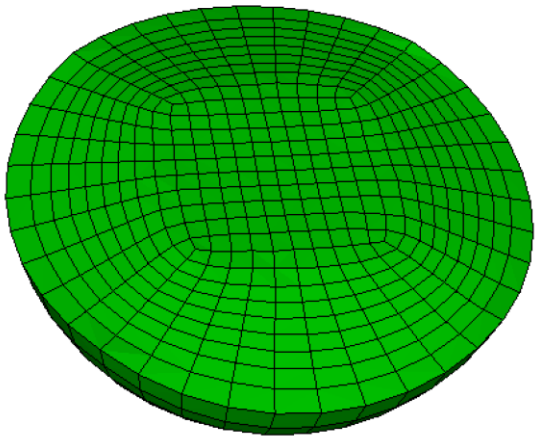
**BIAS (uniform): fraction 0.3 graded_interval 6 az_interval 40 bias 1.0 max_smooth_iterations 0**



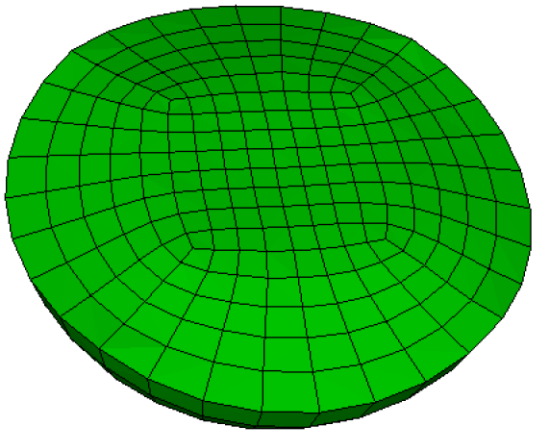
**FRACTION: fraction 0.7 graded_interval 6 az_interval 40 bias 1.0 max_smooth_iterations 0**



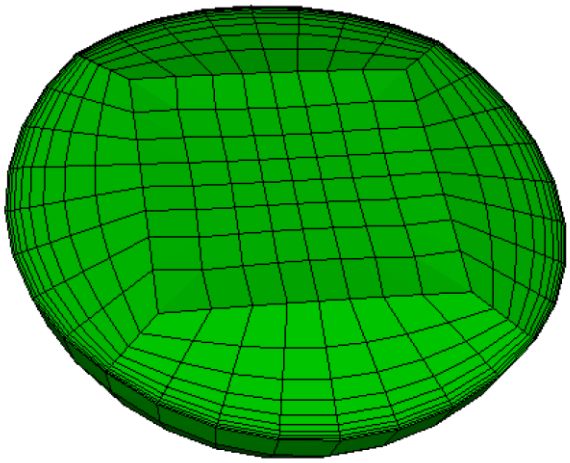
**INTERVAL: fraction 0.7 graded_interval 9 az_interval 40 bias 1.0 max_smooth_iterations 0**



**SMOOTHING: fraction 0.7 graded_interval 9 az_interval 40 bias 1.0 max_smooth_iterations 2**



**AZIMUTHAL (mesh coarseness): fraction 0.7 graded_interval 5 az_interval 32 bias 1.0 max_smooth_iterations 2**



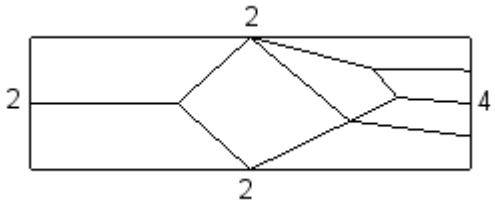
**BIAS (graded): fraction 0.9 graded_interval 9 az_interval 32 bias 1.5 max_smooth_iterations 0**

# *STransition*

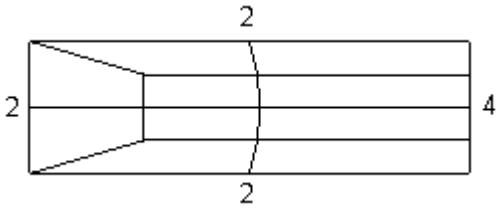
**Applies to**: Surfaces
**Summary:**
Produces a simple transitional mapped mesh.
**Syntax:**

**Surface <surface_id_range> Scheme STransition [Triangle] [Coarse]**

**Discussion:**
The STransition scheme transitions a mesh from one element density to another across a surface. This scheme is particularly helpful when the Paving scheme produces a poor mesh. The following two figures show a specific case where the STransition scheme may offer an improvement.

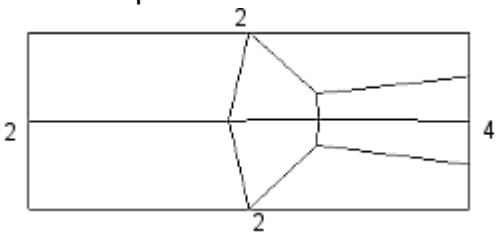



Pave scheme



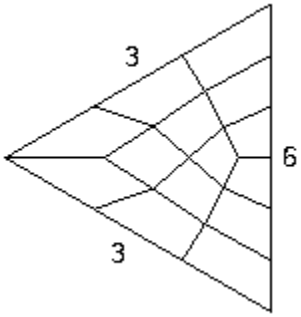

STransition scheme

The **coarse** option forces the mesh to transition to a coarser mesh in the first layer.



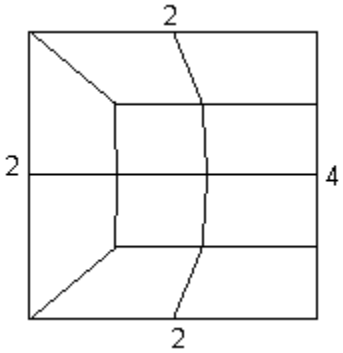

STransition scheme with coarse option

For triangular surfaces, the STransition scheme with the **triangle** option will produce similar results when compared to the Triprimitive scheme. However, STransition is capable of handling more varied interval settings. The following triangle fails when using the Triprimitive scheme but succeeds with the STransition scheme.

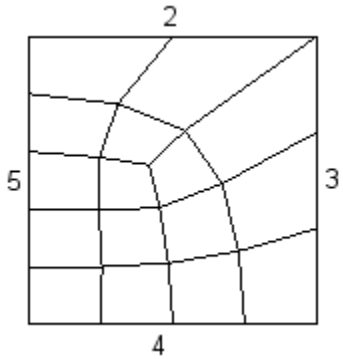



STransition scheme on a triangular surface with intervals set to 3, 3, and 6.

The figures below show the STransition meshing scheme response to different shapes and interval settings.

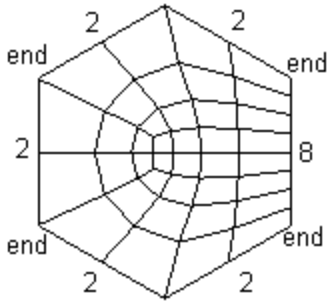



STransition scheme on a rectangular surface with three intervals set to 2 and one set to 4.




STransition scheme on a rectangular surface with intervals set to 2, 3, 4, and 5.

The user also has the option of specifying END or SIDE surface vertex types.




STransition scheme on a hexagon surface with five intervals set to 2, one interval set to 8, and user specified endpoints.

Note, that the Centroid Area Pull smoothing algorithm sometimes gives better results than the default Winslow smoothing algorithm for STransition meshes.

# *Stretch*

**Applies to**: Curves
**Summary**: Permits user to specify the exact size of the first and/or last edges on a curve.
**Syntax**:

**Curve <range> Scheme Stretch [First_size <double>] [Last_size <double>] [Start Vertex <id>]**

**Curve <range> Scheme Stretch [Stretch_factor <double>] [Start Vertex <id>]**

**Related Commands**:
Scheme Bias and Dualbias.
**Discussion**:
This scheme allows the user to specify the exact length of the first and/or last edge on a curve mesh. Intermediate edge lengths will vary smoothly between these input values. Reasonable values for these parameters should be used (for example, the sizes must be less than the total length of the curve). If last_size is input, first_size must be input also. If stretch_factor is input, neither first_size nor last_size can be input. This scheme does not currently work on periodic curves.

# *Submap*

**Applies to**: Surfaces, Volumes
**Summary**: Produces a structured mesh for surfaces/volumes with more than 4/6 logical sides
**Syntax**:

**{Surface|Volume} <range> Scheme Submap**

**Related Commands**:
**{Surface|Volume} <range> Submap Smooth <on|off>**
**Discussion**:
Submapping (Whiteley, 96) is a meshing tool based on the surface mapping capability discussed previously, and is suited for mesh generation on surfaces which can be decomposed into mappable subsurfaces. This algorithm uses a decomposition method to break the surface into simple mappable regions. Submapping is not limited by the number of logical sides in the geometry or by the number of edges. The submap tool, however is best suited for surfaces and volumes that are fairly blocky or that contain interior angles that are close to multiples of 90 degrees.
An example of a volume and its surfaces meshed with submapping is shown in Figure 1.

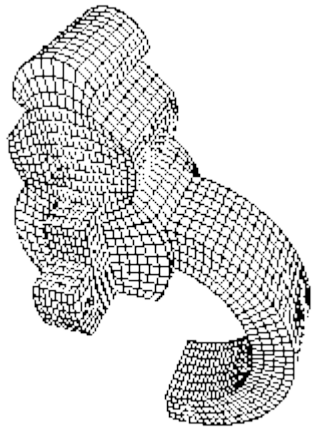


**Figure 1. Quadrilateral and Hexahedral meshes generated by submapping**

Like the mapping scheme, submapping uses vertex types to determine where to put the corners of the mapped mesh (See Surface Vertex Types). For surface submapping, curves on the surface are traversed and grouped into " logical sides " by a classification of the curves position in a local "i-j" coordinate system.
Volume submapping uses the logical sides for the bounding surfaces and the vertex types to construct a logical "i-j-k" coordinate system, which is used to construct the logical sides of the volume. For surface and volume submapping, the sides are used to formulate the interval constraints for the surface or volume.
Figure 2 shows an example of this logical classification technique, where the edges on the front surface have been classified in the i-j coordinate system; the figure also shows the submapped mesh for that volume.

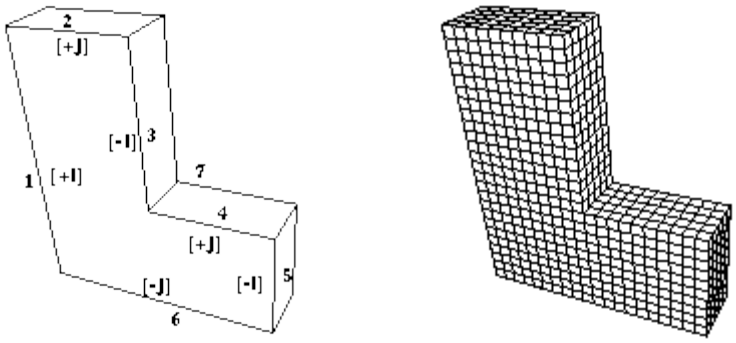



**Figure 2. Scheme Submap Logical Properties**

In special cases where quick results are desired, submap cornerpicking can be set to OFF. The corner picking will be accomplished by a faster, but less accurate algorithm which sets the vertex types by the measured interior angle at the given vertex on the surface. In most cases this is not recommended.

**Set Submap CornerPicking {ON|off}**

In special cases where 4 corners will be selected for a four-sided mapped region, but the region has more than 4 reasonable locations for the 4 corners, one may choose between the submapping or mapping corner picking algorithms to determine the 4 locations for 4 corners. In most cases this is not recommended. The following commands may be used.

**Set Cornerpicking_MapAsSubmap {on|OFF}**

**Set Cornerpicking_SubmapAsMap {on|OFF}**

**List Cornerpicking_MapAsSubmap**

**List Cornerpicking_SubmapAsMap**

After submapping has subdivided the surface and applied the mapped meshing technique mentioned above, the mesh is smoothed to improve mesh quality. Because the decomposition performed by submapping is mesh based, no geometry is created in the process and the resulting interior mesh can be smoothed. Sometimes smoothing can decrease the quality of the mesh; in this case the following command can turn off the automatic smoothing before meshing:

**{Surface|Volume} <range> Submap Smooth <on|off>**

Surface submapping also has the ability to mesh periodic surfaces such as cylinders. An example of a periodic surface meshed with submapping is shown in Figure 3. The requirement for meshing these surfaces is that the top and bottom of the cylinder must have matching intervals.

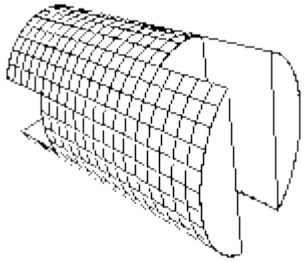


**Figure 3. Periodic Surface Meshing with Submapping**

For periodic surfaces, there are no curves connecting the top and bottom of the cylinder. Setting intervals in this direction on the surface can be done by setting the periodic interval for that surface (see Interval Assignment). No special commands need to be given to submap a periodic surface, the algorithm will automatically detect the fact that the surface is periodic. Currently, periodic surfaces with interior holes are not supported.

*Surface Vertex Types*

- Surface Vertex Commands
- Listing and Drawing Vertex Types
- Triangle Vertex Types
- Adjusting the Automatic Vertex Type Selection Algorithm
- Volume Curve Types

Several meshing algorithms in CUBIT "classify" the vertices of a surface or volume to produce a high quality mesh. This classification is based on the angle between the edges meeting at the vertex, and helps determine where to place the corners of the map, submap or trimesh, or the triangles in the trimap or tripave schemes. For example, a surface mapping algorithm must identify the four vertices of the surface that best represent the surface as a rectangle. Figure 1 illustrates the vertex angle types for mapped and submapped surfaces, and the correspondence between vertex types and the placement of corners in a mapped or submapped mesh.

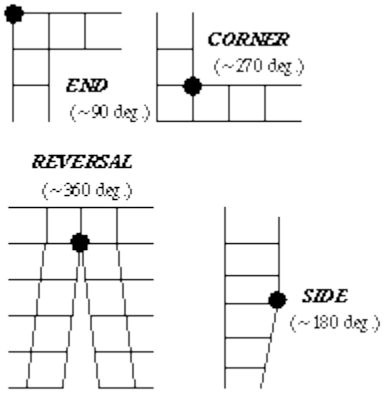


**Figure 1. Angle Types for Mapped and Submapped Surfaces: An End vertex is contained in one element, a Side vertex two, a Corner three, and a Reversal four.**

The surface vertex type is computed automatically during meshing, but can also be specified manually. In some cases, choosing vertex types manually results in a better quality mesh or a mesh that is preferable to the user. Vertex types can be specified directly as End, Side, Corner, or Reversal, or can be specified by giving the desired interior angle as 90, 180, 270, or 360, respectively.
Vertex types have a firmness, just as meshing schemes do. Automatically selected vertex types are **soft**, while user-set vertex types are **hard**.

#### Surface Vertex Commands

Vertex types are set using the following commands:

**Surface <surface_id> [Vertex <vertex_id_range> [Loop_index <int>]] Type {End|Side|Corner|Reversal}**

**Surface <surface_id> [Vertex [<vertex_id_range> [Loop_index <int>]] Angle <value>**

**Surface <surface_id> [Vertex <vertex_id_range> [Loop_index <int>]] Type {Default|Soft|Hard}**

If no vertices are specified, the command is applied to all vertices of each surface.
Note that a vertex may be connected to several surfaces and its classification can be different for each of those surfaces.
The influence of vertex types when mapping or submapping a surface is illustrated in Figure 2. There, the same surface is submapped in two different ways by adjusting the vertex types of ten vertices.

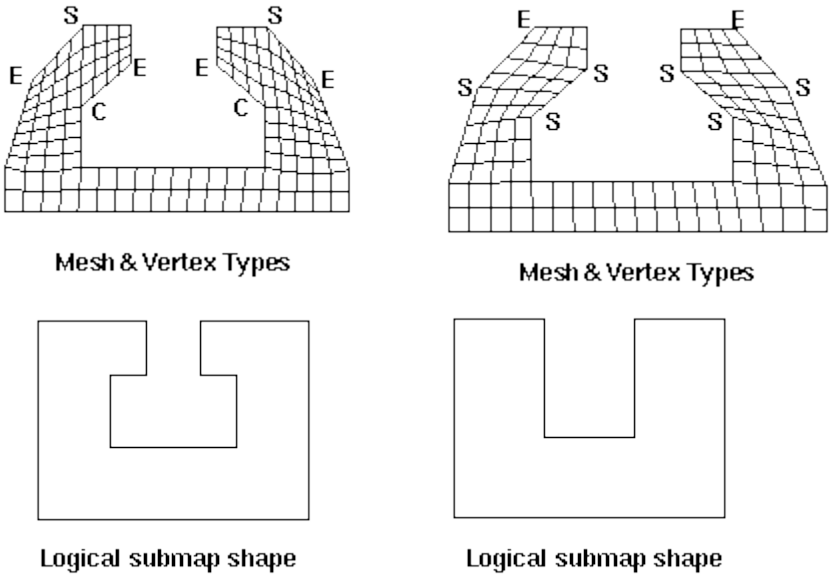


**Figure 2. Influence of vertex types on submap meshes; vertices whose types are changed are indicated above, along with the mesh produced; logical submap shape shown below.**

The **loop_index** is an advanced option used only for vertices where the boundary of a single surface passes through the same vertex more than once. This case is not common. If no loop index is specified for such a vertex, the specified vertex type is assigned to all occurrences of the vertex. The loop index for a specific occurrence of a vertex can be determined by listing the surface (**list surface <id>**) to show the list of curves in each loop bounding the surface, with the start and end vertex listed for each curve. The loop index begins at zero for the first curve in the first loop, and is incremented by one for subsequent curves through the last curve in the last loop. The loop index values corresponding to a specific vertex will be the loop index of each curve whose start vertex is the desired vertex.

#### Listing and Drawing Vertex Types

Listing a surface lists the types of the vertices. The vertex type settings may also be drawn with the following commands:

**Draw Surface <surface_id_range> {Vertex Angle|Vertex Type}**

#### Triangle Vertex Types

For a surface that will be meshed with scheme trimap or tripave, the user may specify the angle below which triangles are inserted:

**Surface <surface_id_range> Angle <angle>**

The user may also set whether to add a triangle at a particular vertex:

**Surface <surface_id> [Vertex <vertex_id_range> [Loop_index <int>]] Type {Triangle|Nontriangle}**

#### Adjusting the Automatic Vertex Type Selection Algorithm

The user may specify the maximum allowable angle at a corner with the following command:

**Set {Corner|End} Angle <degrees>**

The user may also give greater priority to one automatic selection criteria over the others by changing the following absolute weights. The **corner weight** considers how large angles are at corners. The **turn weight** considers how L-shaped the surface is. The **interval weight** considers how much intervals must change. The **large angle weight** affects only auto-scheme selection: surfaces with a large angle will be paved instead. Each weight's default is 1 and must be between 0 and 10. The bigger a weight the more that criteria is considered.

**Set Corner Weight <value>**

**Set Turn Weight <value>**

**Set Interval Weight <value>**

**Set Large Angle Weight <value>**

An illustration of a mesh produced by the submapping algorithm is shown in Figure 2. The meshes produced by submapping on the left and right result from adjusting the vertex types of the eight vertices shown.

#### Volume Curve Types

When sweeping, a 2.5 dimensional meshing scheme, curves perpendicular to the sweep direction can have a type with respect to the volume. These types are usually automatically selected. The following commands are useful:

**Draw Volume <surface_id_range> {Curve Angle|Curve Type}**

**List Volume <volume_id> Curve Type**

**Volume <volume_id> [Curve <curve_id_range>] Type {End|Side|Corner|Reversal}**

**Volume <volume_id> [Curve <curve_id_range>] Type {Default|Soft|Hard}**

# *Sweep*

**Applies to**: Volumes
**Summary**: Produces an extruded hexahedral mesh for 2.5D volumes.
**Syntax**:

**Volume <range> Scheme Sweep [Source [Surface] <range>] [Target [Surface] <range>]**
**[Propagate bias]**
**[Sweep smooth {auto | smart affine | linear | residual | winslow} ]**
**[Sweep transform {LEAST SQUARES | Translate}] [Autosmooth target {ON|off} ]**

**Volume <range> Scheme Sweep Vector <xval yval zval>**

**Volume <range> autosmooth target [off|ON]**
**fixed imprints [on|OFF]**

**smart smooth [ON|off] tolerance <val 0.0 to 1.0=0.2>**
**nlayers <val >=0=5>**

**Related Commands**:

**Set Multisweep [On|Off]**

**Multisweep Smoothing {ON|Off}**

**Multisweep Volume <range> Remove**

**Volume <range> Redistribute Nodes {ON|off}**

**[Set] Legacy Sweeper {On|Off}**

**Discussion**:
The sweep algorithm can sweep general 2.5D geometries and can also do pure translation or rotations. A 2.5D geometry is characterized by source and target surfaces which are topologically similar. The hexahedral mesh is swept (extruded) between source and target along a single logical axis. Bounding the swept hexahedra between source and target surfaces, are the linking surfaces. Figures 1 and 2 show examples of source, target and linking surfaces.
Command Options: The user can specify the source and target surfaces. The user can also specify a geometric vector approximating the sweep direction, and let CUBIT determine the source and target surfaces. The user can specify just the source surfaces, and let cubit guess the target, or "scheme auto" can also be used.

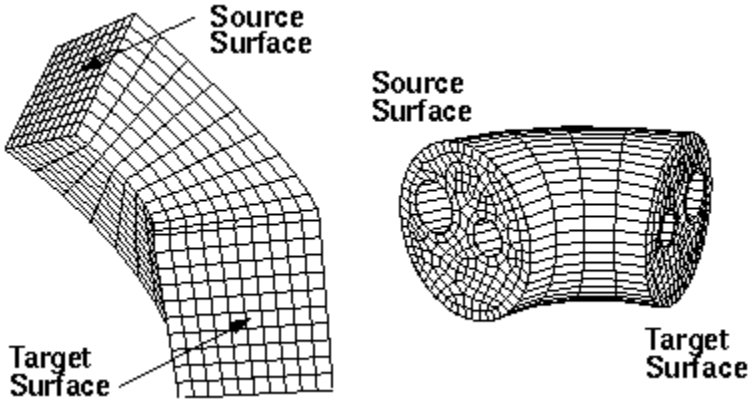



**Figure 1. Sweep Volume Meshing**

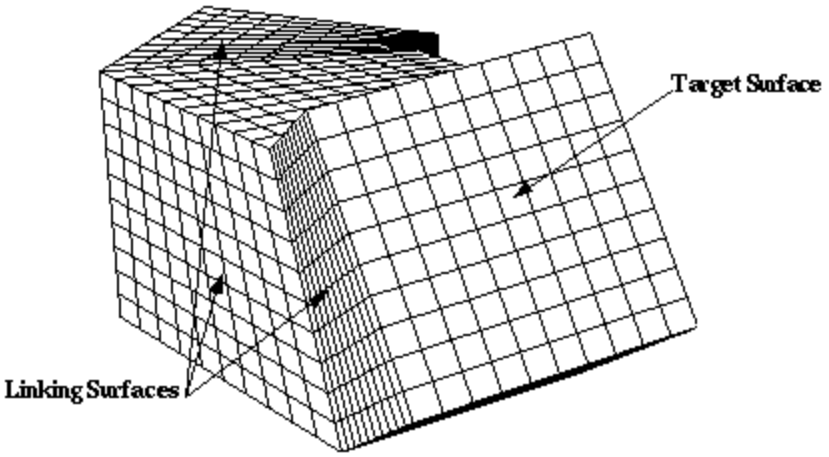



**Figure 2. Multiple Linking Surface Volume Meshing with Scheme Sweep**

In general, the procedure for using the sweep scheme is to first mesh the source surfaces. Any surface meshing scheme may be employed. Figure 1 displays swept meshes involving mapped and paved source surfaces. Linking surfaces must have either mapping or submapping schemes applied. The sweep algorithm can also handle multiple surfaces linking the source surface and the target surfaces. An example of this is shown in Figure 2. Note that for the multiple- linking-surface meshing case, the interval requirement is that the total number of intervals along each multiple edge path from the source surface to the target surface must be the same for each path. Once the appropriate mesh is applied to the source surface and intervals assigned, the **mesh** command may be issued.

In many cases auto-scheme selection can simplify this process by recognizing sweepable geometries and automatically select source and target surfaces. If the source and target surfaces are not specified, CUBIT attempts to automatically select them. CUBIT also automatically sets curve and vertex types in an attempt to make the mesh of the linking surfaces lead from a source surface to a target surface. These automatic selections may occasionally fail, in which case the user must manually select the source/target surfaces, or some of the curve and vertex types. After making some of these changes, the user should again set the volume scheme to sweep and attempt to mesh. In some cases of 1-1 sweeps, the source and target are swapped. Precedence for which surface to use as the source is {meshed, merged, specified as source}. If the user wants to avoid swaps and enforce that a particular surface is the source, then they can mesh that prior to sweeping.

Occasionally the user must also adjust intervals along curves, in addition to the usual surface interval matching requirements. For a given pair of source/target surfaces, there must be the same number of hexahedral layers between them regardless of the path taken. This constrains the number of edges along curves of linking surfaces. For example, in Figure 1 right, the number of intervals through the holes must be the same as along the outer shell.

**Propagate bias Option**: The propagate bias option attempts to preserve the source bias by propagating bias mesh schemes from the curves of the source surface to the curves of the target surface. It also propagates bias from one linking curve to all other linking curves.

**Sweep transform Option**: Swept meshes are created by projecting points between the source and target surfaces using affine transformations and then connecting them to form hexahedra. The method used to calculate the affine transformations is set using the sweep transform option.

**Least squares**: If the least squares option is selected then affine transformations between the source and target are calculated using a least squares method.

**translate**: If the translate option is selected then a simple translate affine transformation is calculated based upon the centroid of the source and target.

**Sweep smooth Option**: *Note: This option is available only in Legacy mode. The command '**set legacy sweeper on|off** controls the mode. Legacy mode is OFF by default.*

To ensure adequate mesh quality, optional smoothing schemes are available to reposition the interior nodes. The sweep tool permits five types of smoothing that are set with the following command prior to meshing a volume whose mesh scheme is sweep:

**Linear**: If this option is selected, no layer smoothing is performed. The node positions are determined strictly by the affine transformation from the previous layer. Good quality swept meshes can be constructed using “linear” provided the volume geometry and meshed linking surfaces permit the volume mesh to be created by a translation, scaling, and/or rotation of the source mesh. Volumes for which this is nearly true may also produce acceptable quality with “linear”. As one would expect, this option generates swept meshes more quickly than the other sweep smooth options. This option is rarely needed since the next option produces better results with little time penalty.

**Smart affine**: The “smart affine” option does minimal smoothing of the interior nodes. Affine transformations are used to project the source and target surfaces to the middle surface of the volume. The position of the middle surface nodes is the average of the projected nodes from the source and target surfaces. The error in projecting from source and target is computed, and this error is linearly distributed back to the source and target.

**Residual**: The “residual” method is often used for meshing volumes that cannot be swept with the “smart linear” method. It tends to produce better quality meshes than the “smart linear” method while running faster than the Winslow-based smoother. The sweeping algorithm uses an affine transformation to calculate the interior nodes’ positions, but the mesh on the linking surface determines the positions of the nodes on the boundary of the layer. For the “residual” method, CUBIT calculates corrective adjustments for interior nodes using the “residuals” from boundary nodes. The “residual” is defined as the distance between the boundary node’s position (as determined by the surface mesh) and the boundary node’s ideal position (as determined by the affine transformation of the previous layer). Cubit computes the residual forward from the source and backward from the target to get best the possible node position.

**Winslow**: Smooth scheme “winslow” smooths each layer using a weighted, elliptic smoother. The weights are computed from the source mesh; they help maintain any biased spacing that occurs on the source mesh. For example, one might want to use the “winslow” option if the source was a biased mesh that was created using scheme circle. The biasing of the outer elements of the source mesh may be destroyed if one

of the other smooth options is used. The interior nodes are initially place using the residual method.

**AUTO**: This is the default for the sweep smooth option. “auto” causes the Sweeper to automatically choose between “smart affine” and “residual.” Auto will choose “off” if the layer needs little or no smoothing or “residual” if it needs smoothing. Scheme “auto” does not guarantee that no negative Jacobians are produced. This option produces acceptable results in most cases. If it fails to produce a quality mesh, then choose one of the other sweep smooth options.
If none of these smooth schemes result in adequate mesh quality, one can consider trying one of the volume smoothing schemes such as condition number or mean ratio.

**Autosmooth target Option and Command**

During sweeping, a quad mesh is placed on each source surface. Then the collection of nodes & quads from all the source surfaces is projected onto the target surface. The **autosmooth target** command or sweep command options control the placement of the nodes onto the target surface.

**Volume <range> autosmooth target [off|ON]**
**fixed imprints [on|OFF]**
**smart smooth [ON|off] tolerance <val 0.0 to 1.0=0.2>**
**nlayers <val >=0=5>**

Issuing the command “Volume <id> autosmooth target off”, or using these options in the sweep command, will project the source nodes onto the target without any subsequent smoothing to improve quality. The result is that the relative placement of the nodes on the target will be as close to identical as possible to the relative placement of the node on the sources. This should be used when sweeping models that are very thin, and smoothing of the target could result in significant skew introduced in the thin layers in the sweep. Axisymmetric models might also want to turn OFF the autosmooth target so that the nodes are identically placed on the symmetry plane surfaces.
Issuing the command “Volume <id> autosmooth target on”, or using it as an option in the sweep command, will call a surface smoother after the initial projection of the nodes onto the target in order to improve surface element quality. This smoothing does not consider hex element quality, only quality of the target surface mesh. This command will smooth all nodes on the target surface. Adding the **“fixed imprint on”** keyword onto the command will cause the target nodes which are projections of source nodes on source curves and vertices to remain fixed during smoothing. Only target nodes, which are projections of source surface nodes will be smoothed. The “**smart smooth on**” option provides further control to the user. If “smart smooth” is turned on, target surface smoothing will only move nodes which are within “nlayers” of a target surface quad element that has a scaled Jacobian quality measure less than the specified “tolerance” value.

#### Multisweep

While the basic sweeping algorithm requires a *single* target surface, the sweeping algorithm can also handle *multiple* target surfaces. The multisweep algorithm works by recognizing possible mesh and topology conflicts between the source and target surfaces and works to resolve these conflicts through the use of the virtual geometry capabilities

in CUBIT. Figure 4 shows some examples of volumes which have been meshed with the multisweep algorithm.

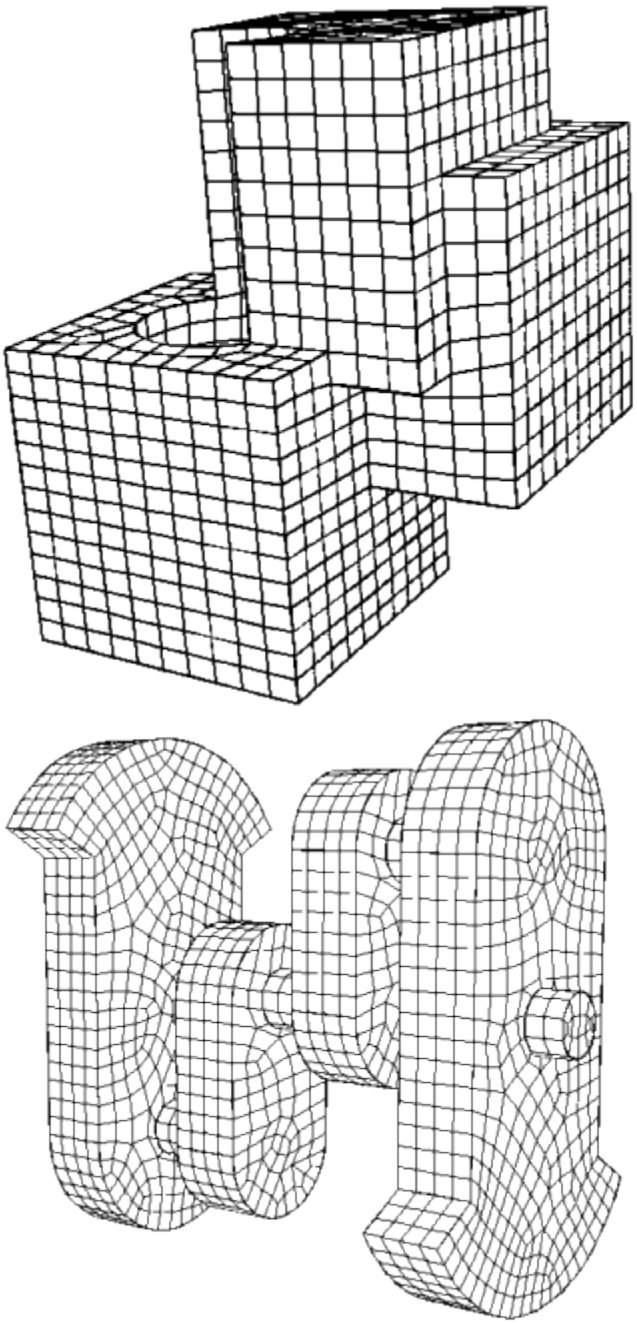


**Figure 4. Examples of Multisweep meshes.**

**Linear**: If this option is active and/or target surfaces are omitted from the scheme setting command, CUBIT will determine source and target surfaces (See Automatic Scheme Selection). Sweeping can be further automated using the "sweep groups" command.

- Limitations: Not all geometries are sweepable. Even some that appear sweepable may not be, depending on the linking surface meshes. Highly curved source and target surfaces may not be meshable with the current sweep algorithm.
- **Grouping Sweepable Volumes**

Swept meshing relies on the constraint that the source and target meshes are topologically identical or the target surface is unmeshed. This results in there being dependencies between swept volumes connected through non-manifold surfaces; these dependencies must be satisfied before the group of volumes can be meshed successfully. For example, if the model was a series of connected cylinders, the proper way to mesh the model would be to sweep each volume starting at the top (or bottom) and continuing through each successive connected volume.
With larger models and with models that contain volumes that require many source surfaces, the process of determining the correct sweeping ordering becomes tedious. The sweep grouping capability computes these dependencies and puts the volumes into groups, in an order which represents those dependencies. The volumes are meshed in the correct order when the resulting group is meshed.
To compute the sweep dependencies, use the command:

**Group Sweep Volumes**

This will create a group named "sweep_groups", which can then be meshed using the command:

**Mesh sweep_groups**

In some automated meshing systems, the source and target surfaces are named using a naming pattern. For example, all source surfaces might be given names "xxx.source" and all target surfaces might be named "xxx.target". This allows the automated setting of the sweep direction based on predetermined names rather than ids. The following command is used to set the source and targets based on the naming pattern.

**Set {Source|Target} Surface Pattern '<pattern>' [Include Volume Name]**

The **pattern** is checked against all surfaces in the model using a simple case-sensitive substring match. All surfaces which contain that string of letters in their name will be designated as either a source or target surface, depending on which option the user specifies. For example:

```
br x 10
surface 1 name 'brick.top'
surface 2 name 'brick.bottom'
```

```
set source surface pattern 'top'
set target surface pattern 'bottom'
volume 1 scheme sweep
list volume 1 brief
```

**Node Redistribution**

**Volume <range> redistribute nodes {ON|off}**

With redistribute set to ON, the boundary nodes of a mappable surface are moved until the spacing between the nodes are equivalent on the two opposing curves. In other words, the parametric values of the nodes lying on the two opposite curves are matched.
Redistribute option ON will assist in avoiding the skewness of the mapped mesh. In the below examples, the linking surfaces are meshed using mapped scheme, and with redistribute option ON, the skewness is significantly avoided (see figures (4) and (5)).
Note:

1. Redistribute option ON will affect all mapped surfaces, not just the linking surfaces of a swept volume. Even though the example below shows a swept volume, the command can be used independent of the sweeping command. That is, it can be used while meshing surface models that contain mappable surfaces.
2. If the linking surfaces of a swept mesh contain submappable surfaces, then the affect of redistribute option ON is generally not seen. The current implementation is restricted to mappable surfaces only and doesn't handle submappable surfaces. In the future, we should be able to easily extend the redistribute option to submappable surfaces.

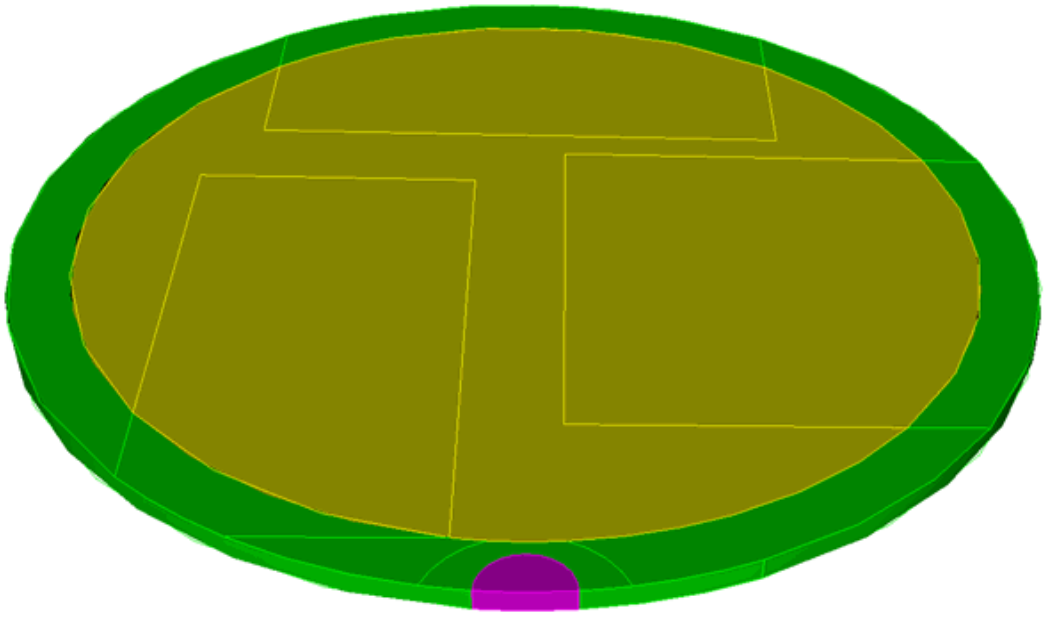


**Figure 1 - Linking surfaces of a many-to-one sweepable solid (shown in green) is mappable**

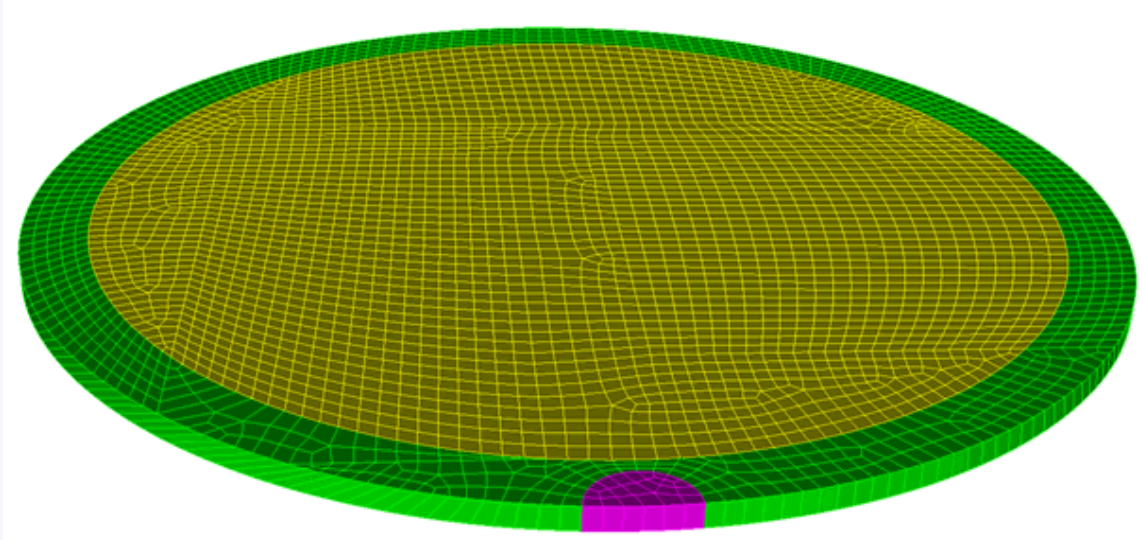


**Figure 2 - Highly skewed elements on the linking mapped surface with 'redistribute nodes OFF'**

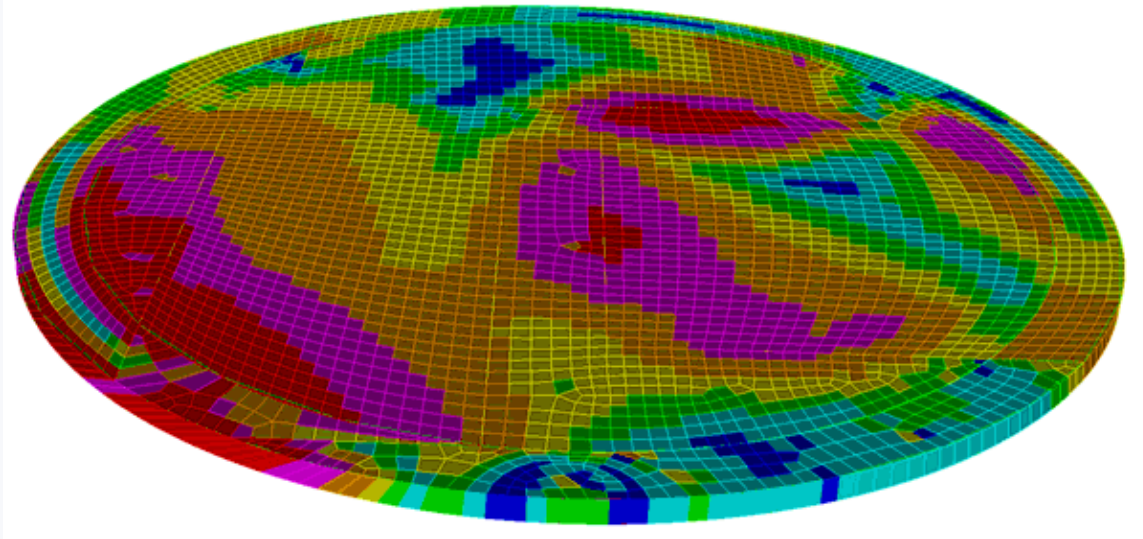


**Figure 3 - Quality of mesh with 'Redistribute Nodes OFF'**

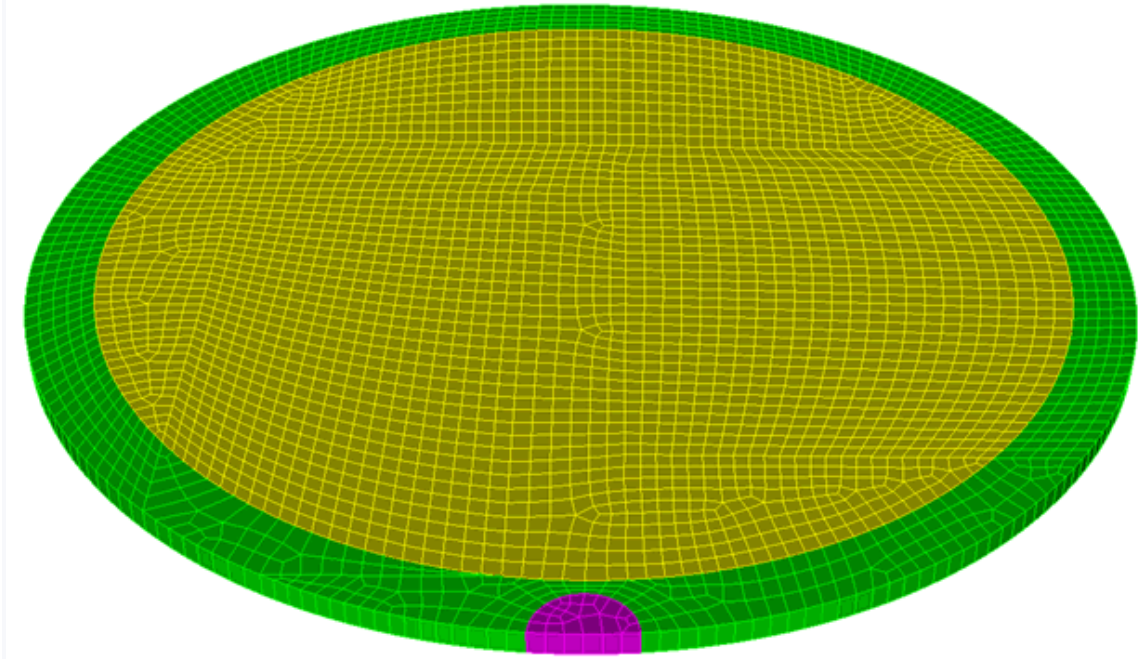


**Figure 4 - High skew on the linking mapped surface can be avoided with 'Redistribute Nodes ON'**

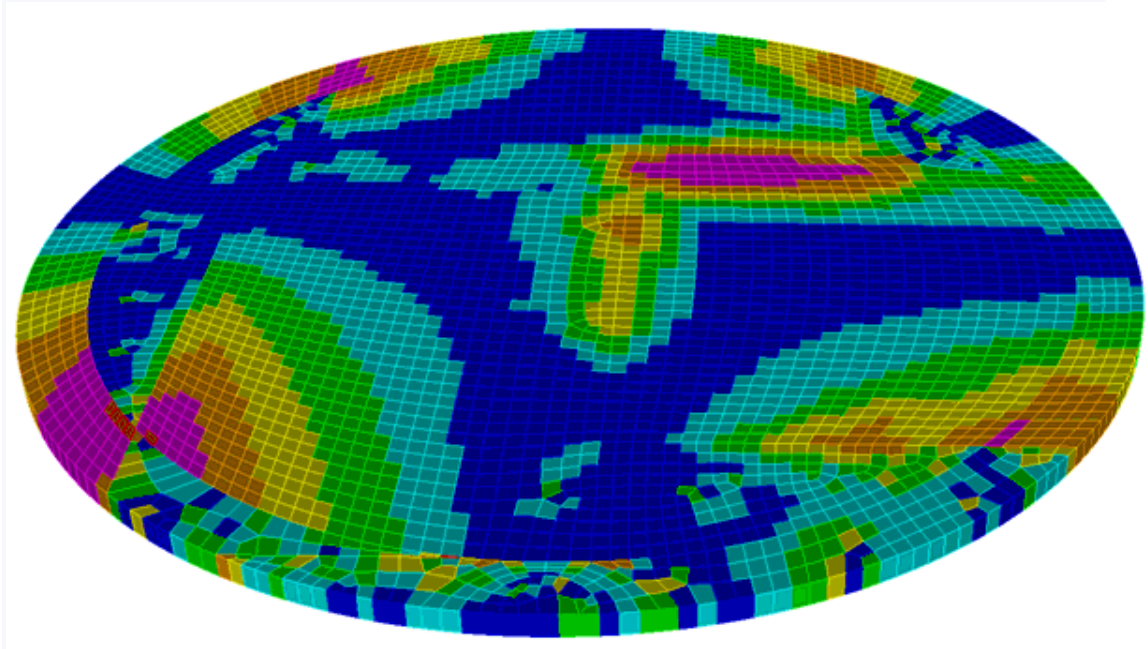


**Figure 5 - Quality of mesh with 'Redistribute Nodes ON'**

# *TetMesh*

**Applies to**: Volumes
**Summary**: Automatically meshes a volume with an unstructured tetrahedral mesh.
**Syntax**:

**Volume <range> Scheme TetMesh [Proximity Layers {on[<num_layers>]|OFF}] [Geometry Approximation Angle <angle>]**

**Related Commands**:

**[Set] Tetmesher Add mid_edge_nodes {on|OFF}**

**[Set] Tetmesher Optimize Surface mid_edge_nodes {on|OFF}**

**[Set] Tetmesher Anisotropic layers {on|OFF [<layers=2>]}**

**[Set] Tetmesher Boundary Recovery {on|OFF}**

**[Set] Tetmesher HPC {ON|off} [Threads <value=4>]**

**[Set] Tetmesher HPC minimum size [<size>]**

**[Set] Tetmesher Interior Points {ON|off}**

**[Set] Tetmesher Optimize { { [Level <level> [Overconstrained Edges {on|OFF}] [Overconstrained Tetrahedra {on|OFF}] [Sliver {on|OFF}] } | Default }**

**[Set] Trimesher Surface Gradation <value>**

**[Set] Trimesher Volume Gradation <value>**

**[Set] Trimesher Geometry Sizing {ON|off}**

**[Set] Trimesher Split Overconstrained Edges {on|OFF}**

**THex Volume All**

**Volume <volume_id> Tetmesh Respect {Face|Tri|Edge} <range>**

**Volume <volume_id> Tetmesh Respect Node <range> [Size <size>]**

**Volume <volume_id> Tetmesh Respect Clear**

**Volume <volume_id> Tetmesh Respect File '<filename>'**

**Volume <volume_id> Tetmesh Respect Location (options)**

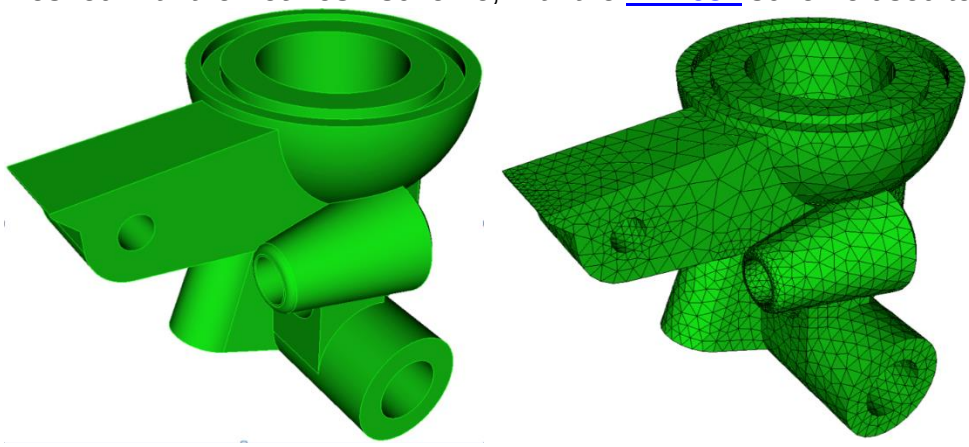
**Tetmesh Tri <range> [Make {Block|Group} [<id>]]**

**Tetmesh Tri <range> {Add|Replace} {Block|Group} <id>**

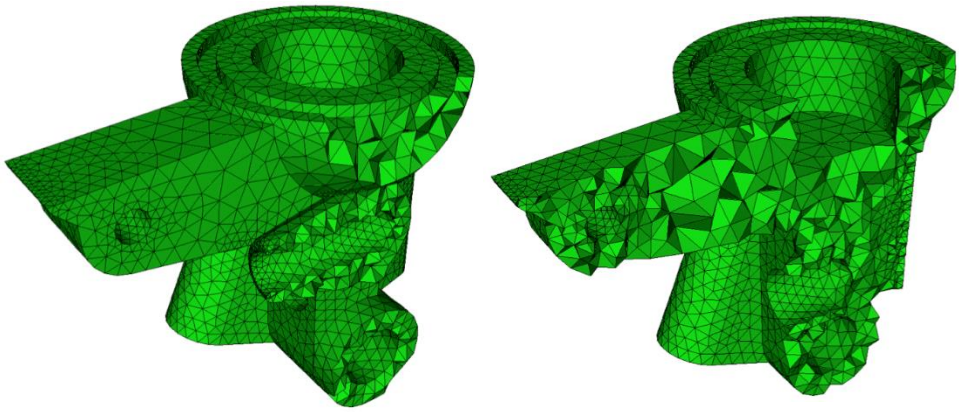
**Volume <id_range> Tetmesh growth_factor <value 1.0 to 10.0 = 1.05>**

**Discussion**

The **TetMesh** scheme fills an arbitrary three-dimensional volume with tetrahedral elements. The surfaces are first triangulated with one of the triangle schemes (TriMesh, TriAdvance or TriDelaunay) or a quadrilateral scheme with the quadrilaterals being split into two triangles (QTri). If a meshing scheme has not been applied to the surfaces, the TriMesh scheme will be used.
Included in Cubit is a third party software library for generating tetrahedral meshes called MeshGems. This is a robust and fast tetrahedral mesher developed by the French laboratory INRIA and distributed by Distene. It utilizes an algorithm for automatic mesh generation based upon the Voronoi-Delaunay method. Figure 1 shows a CAD model meshed with the **TetMesh** scheme, with the TriMesh scheme used to mesh the surfaces.



(a) (b)



(c) (d)

**Figure 1. Tetrahedral mesh generated with the TetMesh scheme using default settings. (a) Initial CAD geometry (b) CAD model with surface mesh generated with TriMesh scheme. (c) and (d) Cut-away views of the interior tetrahedral mesh**

The **TetMesh** scheme is usually very good at generating a mesh with its default settings. In most cases no adjustments to default settings are necessary. Using the size assigned to the volume, either assigned explicitly or defined with an auto size, the **TetMesh** scheme will attempt to maintain the assigned size, except where features smaller than the specified size exist. In this case, smaller tets will automatically be generated to match the feature size. The tet mesher will then generate a smooth gradation from the small tets used to capture features, to the size specified on the volume. This effect is shown in figure 1 where internal transitions in tetrahedra size can be seen. User defined sizes and intervals can also be assigned to individual surfaces and curves for more specific control of element sizes.

A **sizing function** can also be used with the **TetMesh** scheme to control element sizes, however the algorithm used for meshing surfaces will automatically revert to the **TriAdvance** scheme. This is because the **TetMesh** scheme provides built-in capabilities for adaptively controlling the element sizes based on geometry. More details can be found in Geometry Adaptive Sizing for TriMesh and TetMesh Schemes

When using the **TetMesh** and **TriMesh** schemes, recommended practice is to mesh all surfaces and volumes simultaneously. This provides the greatest flexibility to the algorithms to determine feature sizes and their effect on neighboring surfaces and volumes.

#### TetMesh Scheme Options

The **Tetmesh** options described below can be set to adjust the default behavior of the tet mesher. Scheme options are assigned independently to each volume as part of the **scheme tetmesh** command.

**Proximity Layers {on[<num_layers>]|OFF}**

In some thin regions of the model, it may be necessary to ensure a minimum number of element layers through the thickness to better capture physical properties. Using the **proximity layers** setting, the specified minimum **num_layers** of tetrahedra will be placed in thin regions, even if the tetrahedra sizes drop below the size assigned to the volume. The default setting for **proximity layers** is **OFF** where element sizes will not be affected in thin regions.

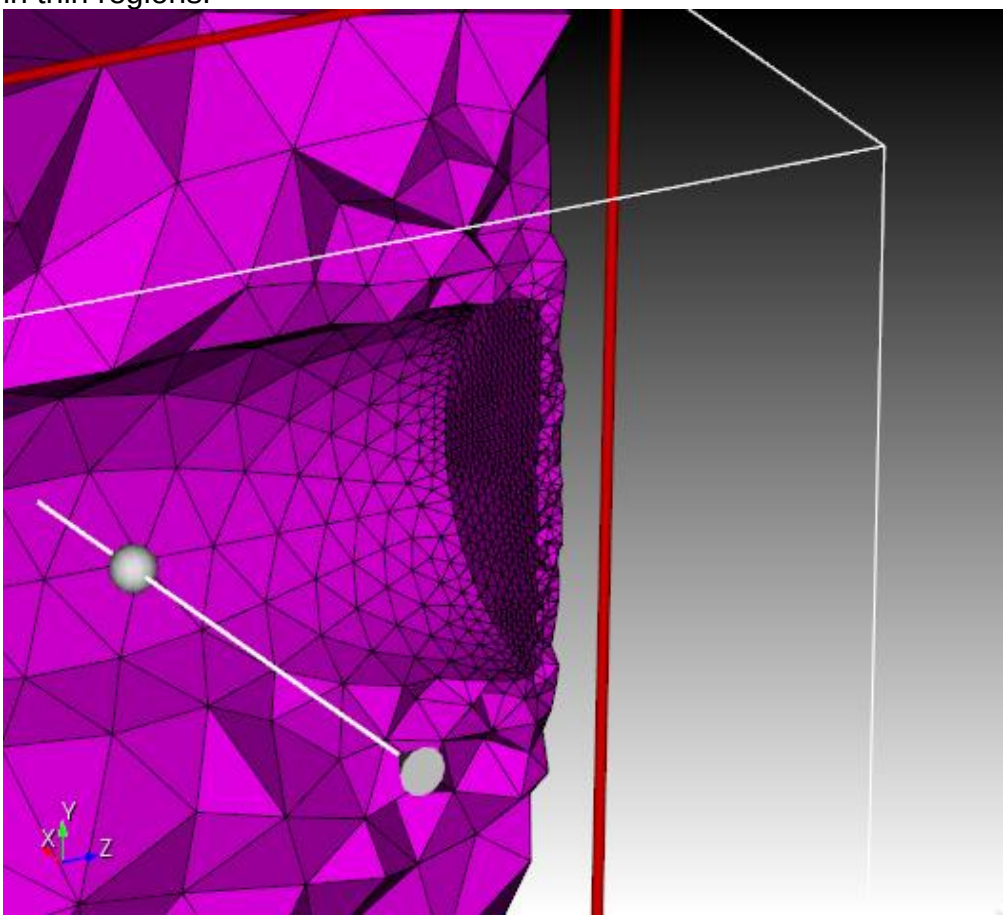



**Figure 2. Demonstrates the effect of using proximity layers on a cut-away section of a volume. Note the layers of smaller tets placed in the thin region.**

### Geometry Approximation Angle <angle>

For non-planar CAD surfaces, an approximation must always be made to capture the curved features using the linear faces of the tetrahedra. When a **geometry approximation angle** is specified, the tet mesher will adjust element sizes on curved surfaces so that the linear edges of the tetrahedra will deviate no greater than the specified **angle** from the geometry. Figure 3 illustrates how the geometry approximation angle is determined. If the red curve represents the geometry and the black segments represent the mesh, the angle **θ** is the angle between the tangent plane at point **A** and the plane of a triangle at **A**. **θ** represents the maximum deviation from the geometry that the mesh will attempt to capture. As shown in figure 2(b), a smaller geometry

approximation angle will normally result in more elements, but it will more closely approximate the actual geometry. The default approximation angle is 15 degrees.

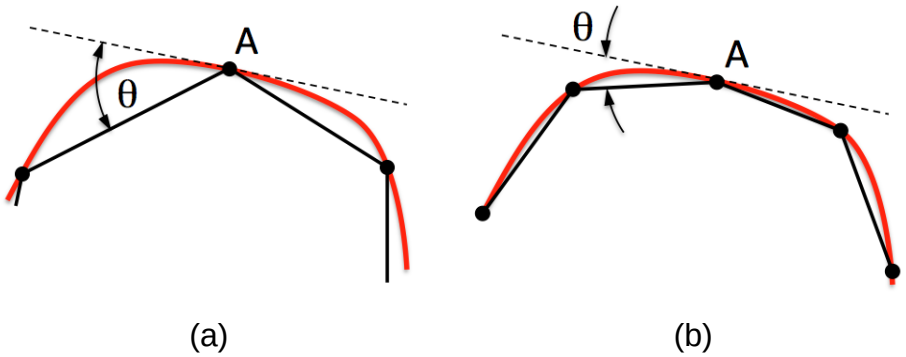



**Figure 3. The geometry approximation angle θ is shown as the maximum deviation between the tangent plane at A and the plane of a triangle at A.**

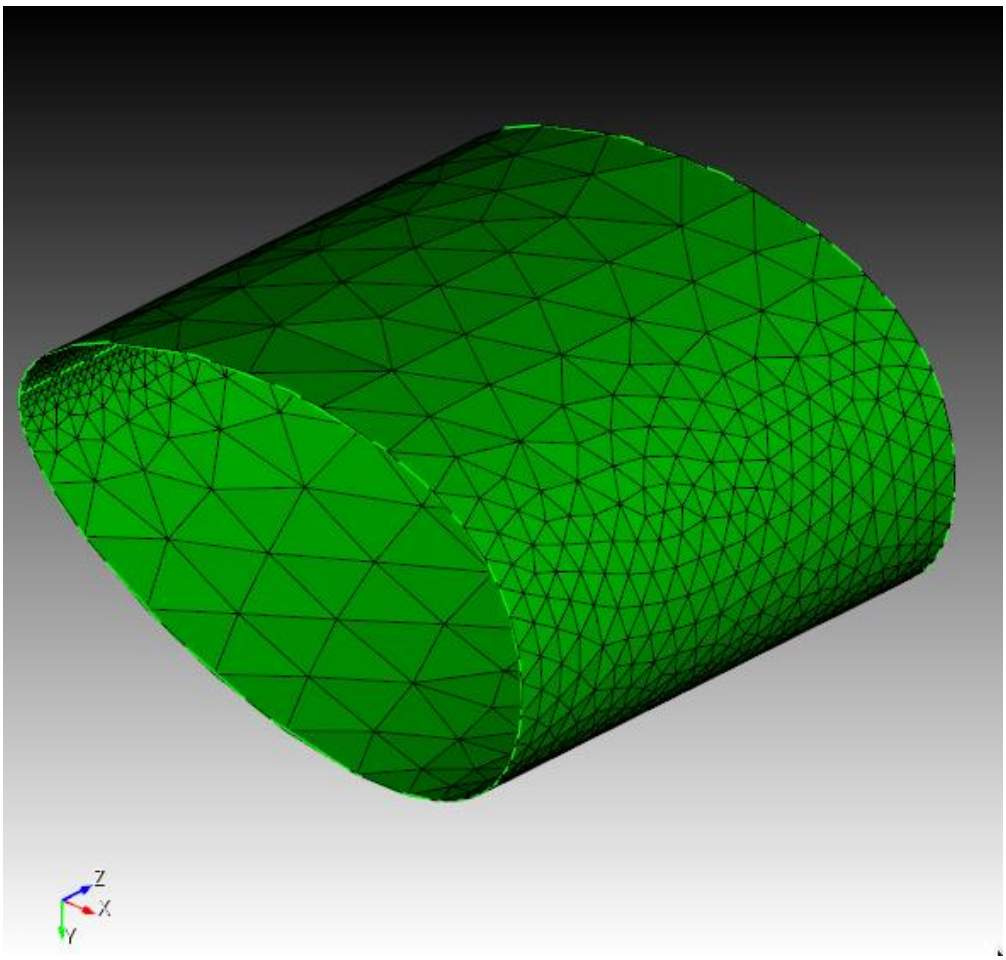



**Figure 4. Demonstrates the effect of the geometry approximation angle set on the volume. Triangle sizes on the interior of surfaces will be adjusted to better capture curvature.**

#### Global Tetmesher Options

The user may set options that control the operation of the tet-meshing algorithms. These tetmesher options are global settings and apply to all tetmeshes generated when the scheme is set to **TetMesh** until the option is changed by the user.

**[Set] Tetmesher Add mid_edge_nodes {on|OFF}**

If the triangle mesh given to tetmeshing has quadratic (mid-edge) nodes, tetmeshing can automatically create quadratic edge nodes while generating the tets if this option has been turned on. By performing this step during tetmeshing, these nodes can be placed optimally by the Meshgems tetmesher, improving element quality. If triangle or face elements have been set to be 'respected' in the tetmesh, they must also have quadratic edge elements or meshing will fail. The default value for this option is off. If set to off, quadratic edge nodes will be placed after meshing, exactly half way along the linear edge.

**[Set] Tetmesher Optimize Surface mid_edge_nodes {on|OFF}**

If the triangle mesh given to tetmeshing has quadratic (mid-edge) nodes, tetmeshing can also automatically optimize the locations of these mid edges nodes during the tetmeshing operation to achieve improved quality. create quadratic edge nodes while generating the tets if this option has been turned on. By performing this step during tetmeshing, these nodes can be placed optimally by the Meshgems tetmesher, improving element quality. If triangle or face elements have been set to be 'respected' in the tetmesh, they must also have quadratic edge elements or meshing will fail. The default value for this option is off. If set to off, quadratic edge nodes will be placed after meshing, exactly half way along the linear edge.

**[Set] Tetmesher Anisotropic Layers {on|OFF [<layers=2>]}**

The **Anisotropic Layers** setting attempts to place the specified number of layers of tetrahedra through thin regions of the volume while respecting the volume mesh size in the thick direction. The default number of layers is two. This option is currently under development and can sometimes generate high aspect ratio tetrahedra. The number of layers generated can sometimes exceed the number of layers specified..

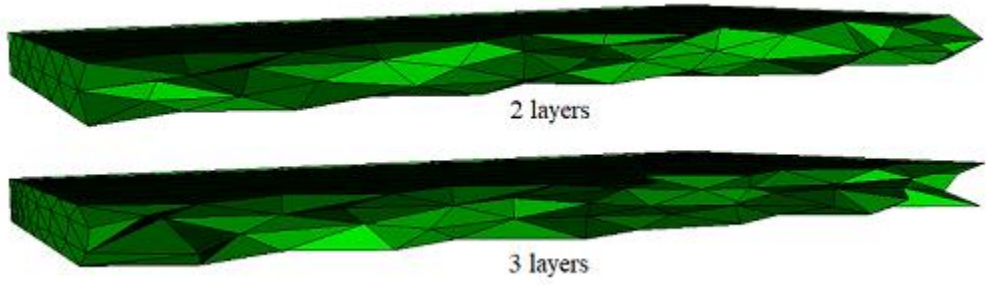



**Figure 5. Anisotropic Volume Meshing**

### [Set] Tetmesher Boundary Recovery {on|OFF}

The **TetMesh** scheme includes a specialized module known as *Boundary Recovery*. Normally if the quality of the surface mesh is good, the boundary recovery module is not used and the resulting tet mesh will conform exactly to the triangles defined on the surfaces without additional processing. In some cases where the surface mesh contains triangles that are of poor quality (ie. highly stretched or sliver shaped triangles) the tet mesher is unable to generate sufficiently good quality elements. When this occurs, the boundary recovery module is automatically invoked. This module does additional processing to temporarily modify boundary triangles so that reasonable quality tets may be inserted. The boundary adjustment is done as an intermediate phase and in most cases the boundary triangulation remains unchanged following meshing. The **TetMesh** scheme in Cubit will automatically invoke the boundary recovery module if the minimum surface mesh quality drops below a condition number of 0.2. However, if the the boundary recovery option is set to **ON**, the tet mesher will use the boundary recovery module regardless of surface mesh quality. Turning this setting **ON** will normally increase the time to generate the mesh, but may result in improved mesh quality. The default setting is **OFF**.

### [Set] Tetmesher HPC {ON|off} [Threads <value=4>]

This option turns on or off MeshGems-Tetra HPC, the multithread or distributed tetrahedral volume mesh generator. The MeshGems-Tetra HPC software is an automatic multithread or distributed tetrahedral mesh generator based on the constrained VORONOI-DELAUNAY method. Using the **threads** option, one can specify the maximum number of threads MeshGemsTetra HPC will use to generate the mesh. The effective number of threads used will be determined by the number of parallel subdomains used, the default of 4, and the maximum of 8. If HPC is off, the older serial tetmesher MeshGems-Tetra is used. The default setting is **ON**.

### [Set] Tetmesher HPC minimum size [<size>]

Sets the minimum edge length in tetmeshing, when using Distene's MeshGems-Tetra HPC.

### [Set] Tetmesher Interior Points {ON|off}

Infrequently, the user desires a model with as few interior points as possible. The **Interior Points** command allows the user to enable or disable, or turn **OFF** the insertion of interior points. If interior points are disabled, the tetmesher will attempt to mesh the volume using only the exterior points. This may not be possible and a few points will be inserted to allow tet-meshing to complete. The default setting is **ON**, meaning that interior points will be inserted according to the specified element size.

### [Set] Tetmesher Optimize Level <level>

The **Tetmesher Optimize Level** command allows the user to control the degree of optimization used to automatically improve element quality following the initial generation of tetrahedra. The optimization level is an integer in the range 0 to 6, which represent how aggressively the algorithm will attempt to improve element quality by automatically adjusting element connectivity and smoothing. The integers 0 to 6 can also be represented as **none (0), light (1), medium (2), standard (3), strong (4), heavy (5), and extreme (6)**. Greater values will result in greater computation time, however may result in improved mesh quality. The default is 3 or standard optimization.

**[Set] Tetmesher Optimize Overconstrained Edges {on|OFF}**

This option controls the splitting of overconstrained edges. An edge is considered overconstrained when it connects two surface nodes but does not belong to the surface. This condition may not be desirable for some FEA analysis. Splitting edges can useful to guarantee two elements through the thickness. When using MeshGems-Tetra, this option cannot be used by itself; it must be used with the **optimize tetrahedra** option. If using MeshGems-Tetra HPC, it can be used by itself. The default for **optimize overconstrained edges** is **OFF**.

**[Set] Tetmesher Optimize Overconstrained Tetrahedra {on|OFF}**

In some cases, the default mesh generated with the **TetMesh** scheme may result in cases where more than one triangle face of a single tetrahedra lies on the same geometric surface. This condition may not be desirable for some FEA analysis. The default for **optimize overconstrained tetrahedra** is **OFF**.

**[Set] Tetmesher Optimize Sliver {on|OFF}**

A sliver tetrahedra is one in which the four nodes of the tet are nearly co-planar. Sliver tets are a common occurrence when using the Delaunay method, but are normally removed by standard optimization. In some cases, sliver tets may still remain even after optimization. To facilitate removal of all sliver-shaped tets, the **optimize sliver** option may be set to **ON**. In this event, additional processing will be done on the mesh to attempt to identify and remove all sliver-shaped tets from the mesh. Since this step may take additional time, and in most cases is not needed, the default setting is **OFF**.

**[Set] Tetmesher Optimize Default**

The **Tetmesher Optimize Default** command restores the default optimization values: level = 3 (standard), overconstrained edges = off, overconstrained tetrahedra = off, and sliver = off.

#### Using tets as the basis of an unstructured hexahedral mesh

Tet meshing can be used to generate hexahedral meshes using the **THex** command. Each of the tetrahedron can be converted into 4 hexes, producing a fully conformal

hexahedral mesh, albeit of poorer quality. These meshes can often be used in codes that are less sensitive to mesh quality and mesh directionality. The **THex** command requires that all tets in the model be converted to hexahedra with the same command.

#### Conforming the tetmesh to internal features

In some cases it is necessary for the finite element mesh to conform to internal features of the model. The tetmesh scheme provides this capability provided the **tetmesh respect** command has been previously issued to define the features that will be respected.

```
Volume <volume_id> Tetmesh Respect {Face|Tri|Edge} <range>

Volume <volume_id> Tetmesh Respect Node <range> [Size <size>]
```

The tetmesh respect command allows the user to specify mesh entities that will be part of a tetrahedral mesh. These faces, triangles, edges, or nodes are inside the volume since all surface mesh features will appear in the final tetrahedral mesh by default. These mesh entities specified to be respected can be generated from other meshing commands on free vertices, curves, or surfaces.

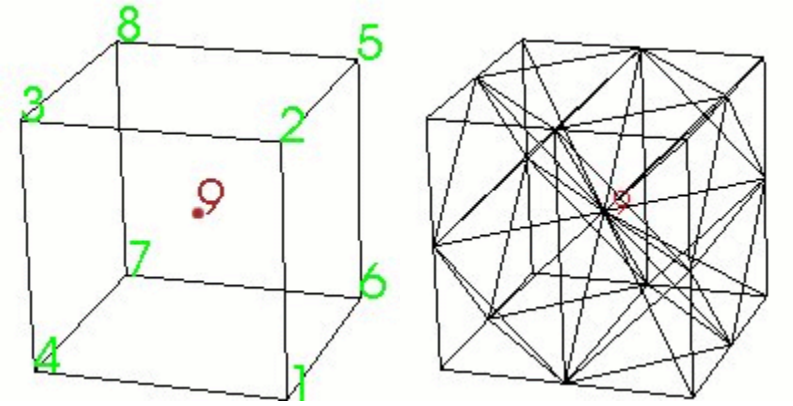



**Figure 2. Example of using tetmesh respect to ensure node 9 is captured in the tetmesh.**

Figure 2 is an example of using the **tetmesh respect** command to enforce a node at the center of a cube. Node 9 in this example was generated by first creating a free vertex at the center location and meshing the vertex. (mesh vertex 9). The following commands would then be used to generate the tetmesh that respected node 9.

```
volume 1 scheme tetmesh
tetmesh respect node 9
mesh volume 1
```

The **tetmesh respect** command can also be used to enforce multiple mesh entities. To accomplish this, the **tetmesh respect** command may be issued multiple times. For example, If node 12 and a triangle 2 inside volume 3 was to appear in the volumetric mesh, the following commands could be used:

```
volume 3 scheme tetmesh
volume 3 tetmesh respect node 12
```

**volume 3 tetmesh respect tri 2 mesh volume 1**

The **tetmesh respect** command can also be given a **size** value with a node. When given a size, the generated tet elements surrounding the node will have sizes matching the given size. This may be useful to provide refinement at given locations within a tet mesh. Unlike the **tetmesh respect** command described above, the **tetmesh respect file** and **tetmesh respect location** commands do not require underlying geometry.

**Volume <volume_id> Tetmesh Respect File '<filename>'**

**Volume <volume_id> Tetmesh Respect Location (options)**

These two commands create mesh data that only the tetmesher knows about. Thus, to respect a point at (1.0, 0.0, -1.0) in your model, enter the command

**volume 1 tetmesh respect location 1 0 -1**

This is much simpler than creating the vertex, meshing it, and then respecting it.
If the model has many points that must be respected, use the file version of the command. First generate a file with all of the points, edges, and triangles that should be respected. The format of the file is the format used by the facet file. Now, use the following command to respect all of the information in the file for the given volume.

**volume 2 tetmesh respect file 'my_points.facet'**

Finally, the following command is used to remove the respected data from an entity.

**Volume <volume_id> Tetmesh Respect Clear**

The tetmesh respect clear command is the only way to remove respected data from a volume without deleting the volume. Unfortunately, it removes all respected data from the volume. Therefore, if the model has a lot of data to be respected, it is best to put it in a file or keep a journal file that can be edited.

#### Controling the gradation of the mesh size inside the volume

**Volume <id_range> Tetmesh growth_factor <value 1.0 to 10.0 = 1.05>**

The **growth_factor** option controls how fast the tetrahedra sizes can change when transitioning from small to larger sizes within the volume. For example a value of 1.5 will attempt to limit the ratio between 2 adjacent tetrahedral edges. Valid values for gradation should be greater than or equal to 1.0 and usually less than 2 or 3. The larger the value, the faster the transition is. Likewise, values closer to 1.0 will result in a more uniform mesh. The default setting for **growth_factor** is 1.05, allowing for a somewhat slow transition between sizes within a volume. The size at the interior of a volume can be controlled using the **Volume <range> [Interval] Size <interval_size>** command.
Gradation of the triangles on the surfaces can also be controlled independently using the global settings **[set] trimesher surface gradation** and **[set] trimesher volume gradation**.

#### Generating a Tetmesh from a Skin of Triangles

**Tetmesh Tri <ids> [growth_factor <value>] [Make {Block|Group} [<id>]]**

**Tetmesh Tri <ids> [growth_factor <value>] {Add|Replace} {Block <id>|Group <id>}**

The **Tetmesh Tri** command generates a tetrahedral mesh from the list of triangles entered. The triangles must form a closed surface. The command fails if they do not. The list of triangles may be a skin, and thus a command such as **tetmesh tri in block 1** would be acceptable, should block 1 be a previously defined skin.
The first command form has optional arguments. If the **make** option and its arguments are present, then the specified block or group will contain the generated tet elements. The command fails with the make option if the specified block or group already exists. If the block or group id is omitted, the next available block or group id is used.
The second command form has two options, **add** and **replace**. Each option requires specifying an existing block or group. If the block or group does not exist, the command fails. The **add** option appends the tet elements to the block or group. The **replace** option removes any existing mesh from the block or group before adding the tet elements.
The **growth_factor** option helps control the transition from small to larger sizes within the mesh. The value specified will be the approximate ratio in the size of adjacent tets going from the boundary into the interior of the mesh. For example, a **growth_factor** of 1.0 will give near-constant sizing, while a **growth_factor** of 1.3 allows approximately 30% growth in each layer of adjacent tets.

# *Tetprimitive*

**Applies to**: Volumes
**Summary**: Meshes a 4 "sided" object with hexahedral elements using the standard tetrahedron primitive.
**Syntax**:

**Volume <range> Scheme Tetprimitive [Combine Surface <range>] [Combine Surface <range>] [Combine Surface <range>] [Combine Surface <range>]**

**Discussion**:
The tetprimitive scheme is used to create a hexahedral mesh in a volume which fits the shape of a tetrahedral primitive. The **Tetprimitive** scheme assumes that each of the four surfaces have been meshed with the triprimitive, or similar, meshing scheme. If more than four surfaces form the tetrahedron geometry, the surfaces forming a logical side can be combined using the **combine** option.

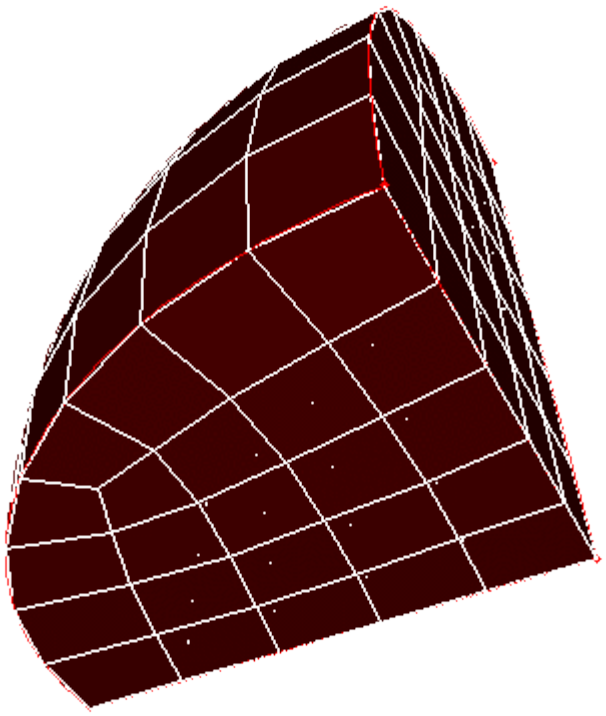
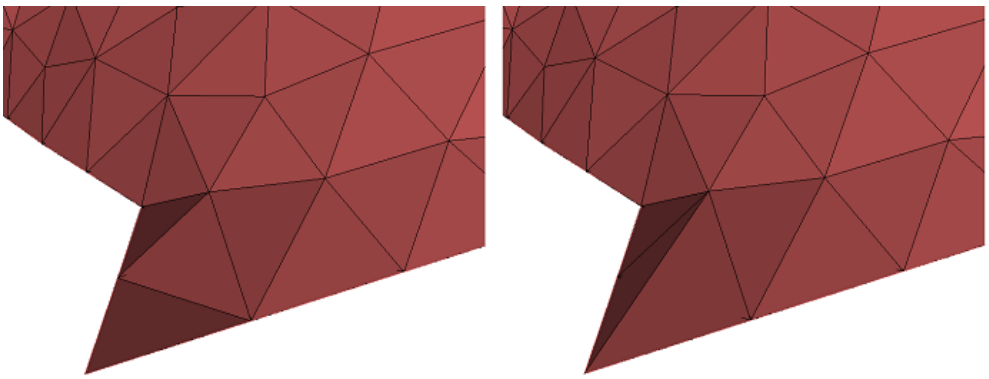


**Figure 1. Sphere octant hex meshed with scheme Tetprimitive, surfaces meshed using scheme Triprimitive**

# *TriAdvance*

**Applies to:** Surfaces
**Summary**: Automatically meshes surface geometry with triangle elements.
**Syntax:**
**Surface <range> Scheme TriAdvance**
**Discussion:**
The triangle meshing scheme TriAdvance fills an arbitrary surface with triangle elements. It is an advancing front algorithm which allows holes in the surface and transitions between dissimilar element sizes. It can use a sizing function like the pave scheme if one is defined for the surface. Future development will add hard lines to this scheme's capabilities. You specify this scheme for a surface by giving the command:
To insure that coplanar (two-tris-sharing-three-node condition) or nearly coplanar tris at sharp geometric vertices are not generated using this scheme, a corrective edge-swap is automatically done.



Without edge swap correction  With edge swap correction

# *TriDelaunay*

**Applies to:** Surfaces
**Summary:** Automatically meshes parametric surface geometry with triangle elements.
**Syntax:**

**Surface <range> Scheme TriDelaunay**

**Discussion:**
The scheme TriDelaunay is a parametric meshing algorithm. It can be run in two modes. The default mode (**asp**) combines the Delaunay [Watson,81] criterion for connecting nodes into triangles with an advancing-front approach for inserting nodes into the mesh. This method maximizes the number of regular triangles in the mesh but does not guarantee the minimum angle quality of the triangles. A guaranteed quality (**gq**) mode can be used for planar surfaces (*only*). This mode refines the initial Delaunay configuration by placing points at the centroids of the worst triangles until the mesh has an acceptable density. To switch between the two modes, use the following setting command.

**[Set] Tridelaunay point placement {gq | guaranteed quality | asp}**

TriDelaunay can also utilize a sizing function if one is defined for the surface.

Note: This algorithm is unstable for periodic surfaces which include a singularity point, E.G. spheres with poles, cone tips and some types of toruses. Use scheme TriMesh, TriAdvance or QTri to mesh non-parametric or periodic parametric surfaces.

# *TriMap*

**Applies to**: Surfaces
**Summary**: Places triangle elements at some vertices, and map meshes the remaining surface.
**Syntax**:

**Surface <range> Scheme Trimap**

**Related Commands**:

**Surface <range> Vertex <range> Type {Triangle|Notriangle}**

**Discussion**:
Some surfaces contain bounding curves which meet at a very acute angle. Meshing these surfaces with an all-quadrilateral mesh will result in a very skewed quad to resolve that angle. In some cases, this is a worse result than simply placing a triangular element to resolve that angle. This scheme resolves this situation by placing a triangular element in these tight corners, and filling the remainder of the surface with a mapped mesh.
The algorithm can automatically compute whether a triangular element is necessary, along with where to place that element. To override the choice of where triangular elements are used, the following command can be issued:

**Surface <range> Vertex <range> Type {Triangle|Notriangle}**

# *TriMesh*

**Applies to:** Surfaces
**Summary**: Automatically meshes surface geometry with triangle elements using the third part *meshgems* tool.
**Syntax:**

**Surface <range> Scheme TriMesh [Geometry Approximation Angle <angle>] [Meshgems] [Minimum Size <value>]**

**Related Commands**:

**[Set] Trimesher Minimum Size <value>**

**[Set] Trimesher Surface Gradation <value>**

**[Set] Trimesher Volume Gradation <value>**

**[Set] Trimesher Geometry Sizing {ON|off}**

**[Set] Trimesher Clean Discrete {on|OFF}**

**[Set] Trimesher Discrete Composites {on|OFF}**

**[Set] Trimesher Split Overconstrained Edges {on|OFF}**

**[Set] Trimesher Ridge Angle {<value=100>}**

**[Set] Trimesher Anisotropic layers {on|OFF [<layers=2>]}**

**[Set] Trimesher Surface Proximity {on|OFF [<ratio=1>]}**

**Discussion:**

The **TriMesh** scheme fills a surface of arbitrary shape with triangle elements. The **TriMesh** scheme serves as the default method for meshing the surfaces of volumes for the TetMesh scheme.

Included in Cubit is a third party software library for generating triangle meshes called MeshGems. This is a robust and fast triangle mesher developed and distributed by Distene. Figure 1 shows a CAD model where surfaces have been meshed with the **TriMesh** scheme. The triangle mesh was then used as input to the TetMesh scheme.

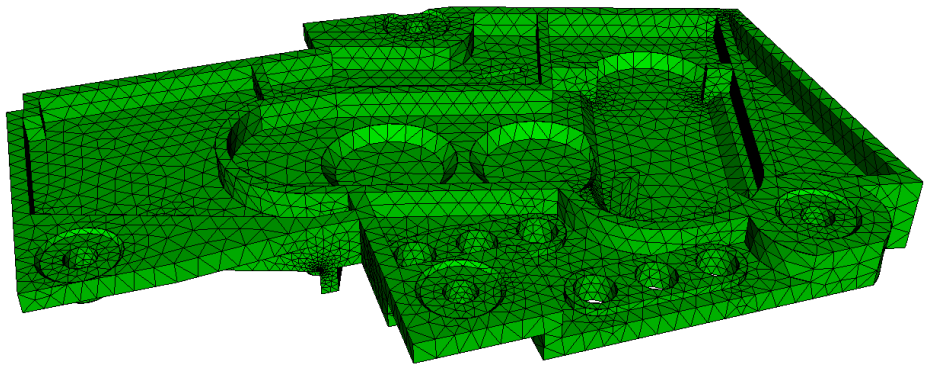


**Figure 1. Triangle meshes generated with the TriMesh scheme using default settings on the surfaces of a CAD model.**

The **TriMesh** scheme is usually very good at generating a mesh with its default settings. In most cases no adjustments to default settings are necessry. Using the size assigned to the surface, either assigned explicitly or defined with an auto size, the **TriMesh** scheme will attempt to maintain the assigned size, except where features smaller than the specified size exist. In this case, smaller triangles will automatically be generated to match the feature size. The triangle mesher will then generate a smooth gradation from the small triangles used to capture features, to the size specified on the surface. This effect is shown in figure 1 where the transitions in triangle sizes can be seen. If no size is specified on the surface, it will use the size that was set on its parent volume. User defined sizes and intervals can also be assigned to individual curves for more specific control of element sizes.

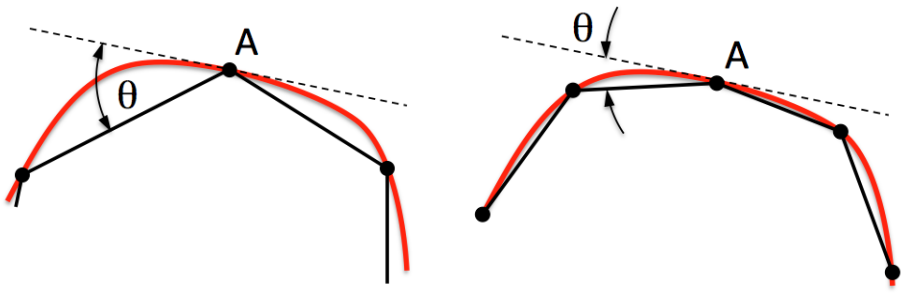
Although rare, if meshing fails when using the **TriMesh** scheme, Cubit will automatically attempt to mesh the surface with the **TriDelaunay** scheme. Subsequent mesh failures will also attempt meshing with the **TriAdvance** and **QTri** schemes.
A **sizing function** can also be used with the **TriMesh** scheme to control element sizes, however the algorithm used for meshing will automatically revert to the **TriAdvance** scheme. This is because the MeshGems algorithm provides built-in capabilities for adaptively controlling the element sizes based on geometry. More details can be found in Geometry Adaptive Sizing for TriMesh and TetMesh Schemes
When using the **TriMesh** and **TetMesh** schemes, recommended practice is to mesh all surfaces and volumes simultaneously. This provides the greatest flexibility to the algorithms to determine feature sizes and their effect on neighboring surfaces and volumes.

#### TriMesh Scheme Options

The **TriMesh** options described below can be set to adjust the default behavior of the tri mesher. Scheme options are assigned independently to each surface as part of the **scheme TriMesh** command. Note that the options described here will apply only if the **TriMesh** scheme is used. **TriDelaunay** and **TriAdvance** schemes will not utilize these options when meshing.

### Geometry Approximation Angle <angle>

For non-planar CAD surfaces and non-linear curves, an approximation must always be made to capture the curved features using the linear edges of the triangle. When a **geometry approximation angle** is specified, the triangle mesher will adjust triangle sizes on curved boundaries so that the linear edges of the triangle will deviate from the geometry by no greater than the specified **angle**. Figure 2 illustrates how the geometry approximation angle is determined. If the red curve representes the geometry and the black segments represent the mesh, the angle **θ** is the angle between the tangent plane at point **A** and the plane of a triangle at **A**. **θ** represents the maximum deviation from the geometry that the mesh will attempt to capture. As shown in figure 2(b), a smaller geometry approximation angle will normally result in more elements, but it will more closely approximate the actual geometry. The default approximation angle is 15 degrees.




(a) (b)

**Figure 2. The geometry approximation angle θ is shown as the maximum deviation between the tangent plane at A and the plane of a triangle at A.**

Note that the **geometry approximation angle** is also effective in controlling the element size on the interior of surfaces as illustrated in figure 3. This is most useful when used in conjunction with the **TetMesh** Scheme where smaller tets will be placed in regions of higher curvature.

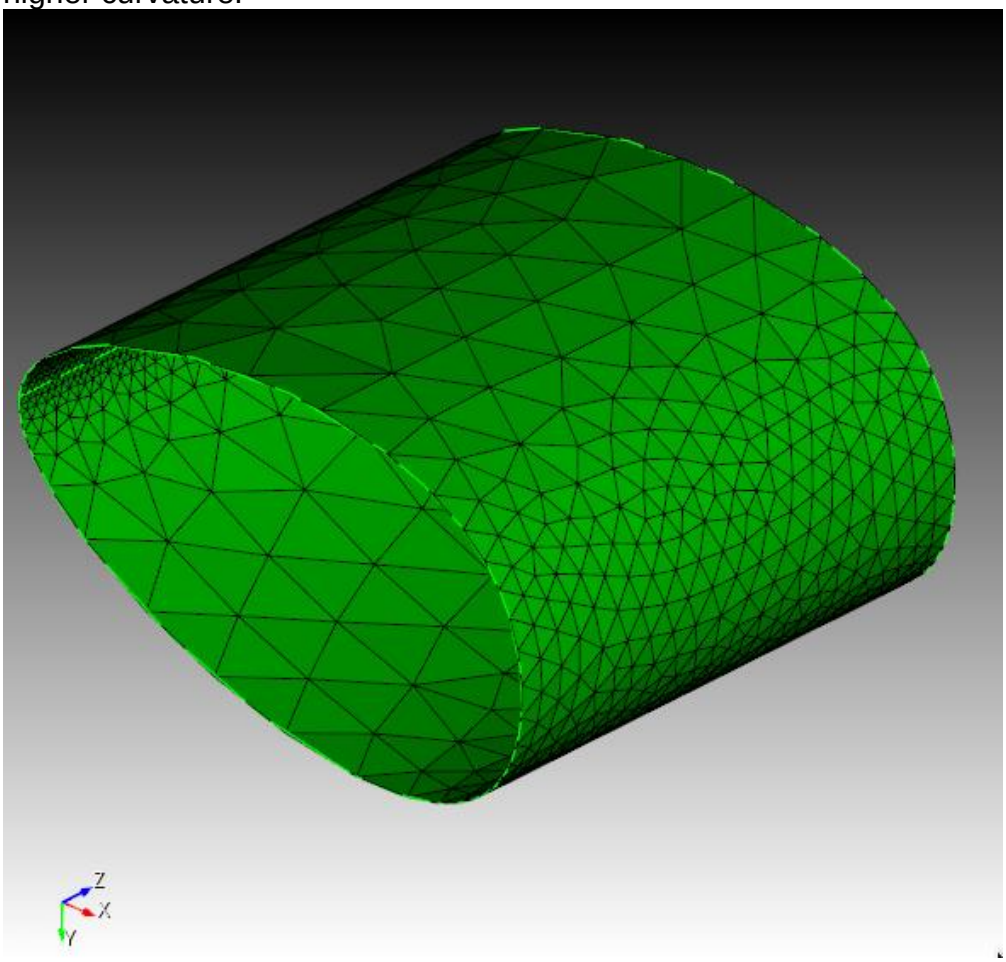



**Figure 3. Demonstrates the effect of the geometry approximation angle to better capture surface curvature on the interior of surfaces.**

**Minimum Size <value>**

By specifying a **minimum size**, the tri mesher will attempt to prevent creating elements smaller than this specified size. It should be noted that there may still be a small number of elements with a size slighly less that this value; it is not an exact setting.
The **MeshGems** option will use only the MeshGems triangle mesher on the specified surfaces. It will not revert upon failure to the **TriDelaunay** or **TriAdvance** schemes.

**Global Trimesher Gradation Options**

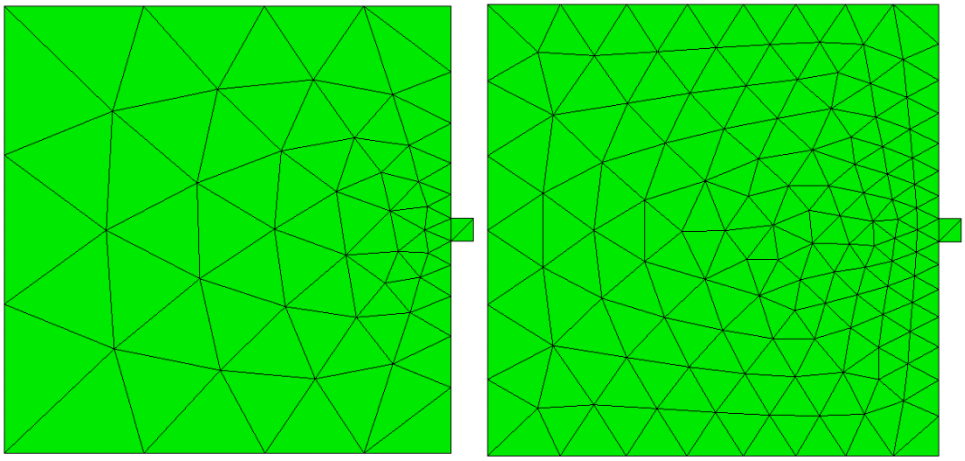
The user may set options that control the gradation of the tri-meshing algorithms. These trimesher options are global settings and apply to all trimeshes generated when the scheme is set to **TriMesh** until the option is changed by the user.
The **minimum size** setting controls the smallest edge length generated during triangle meshing.

**[Set] Trimesher Minimum Size <value>**

The global **gradation** options control how fast the triangle sizes can change when transitioning from small to larger sizes. For example a value of 1.5 will attempt to limit the change in element size of adjacent triangles to no greater than a factor of 1.5. Valid values for gradation should be greater than 1.0 and usually less than 2 or 3. The larger the value, the faster the transition resulting in fewer total elements. Likewise, values closer to 1.0 can result in significantly more elements, especially when small features are present. The default setting for **gradation** is 1.3. Gradation can be controlled for both surfaces and volumes.

**[Set] Trimesher Surface Gradation <value>**

**Surface gradation** will control the growth of triangles where element size has been determined by bounding curves. For example, Figure 4 shows a small feature where element sizes have been determined locally by the length of the small curves. A gradation is applied so that triangle sizes increase away from the small feature. A surface gradation of 1.3 is shown on the left, while a surface gradation of 1.1 is shown on the right.



(a) (b)

**Figure 4. Demonstrates the effect of changing the default gradation, where (a) is the default gradtion of 1.3, compared with (b) using a gradation of 1.1. Note that both images use the same interval size setting for the surface.**

**[Set] Trimesher Volume Gradation <value>**

**Volume gradation** will control the growth of triangles where element size has been determined by the proximity of other nearby surfaces. For example, Figure 5a and 5b

shows a brick with a small void where the surface meshes are generated with the **TriMesh** scheme. The surface gradation has been adjusted to a large number so its effect is negligible. The small element size determined for the void is propagated to the exterior surfaces. The resulting gradation of the nearby triangles on the surface is determined by the **trimesh volume gradation** setting.

Note that the **trimesh volume gradation** command is different than the growth factor control setting. The **trimesh volume gradation** controls the gradation of triangles on the surface due to nearby features where small tets will exist, whereas the **volume <range> tetmesh growth_factor** command controls the gradation of the interior tet elements.

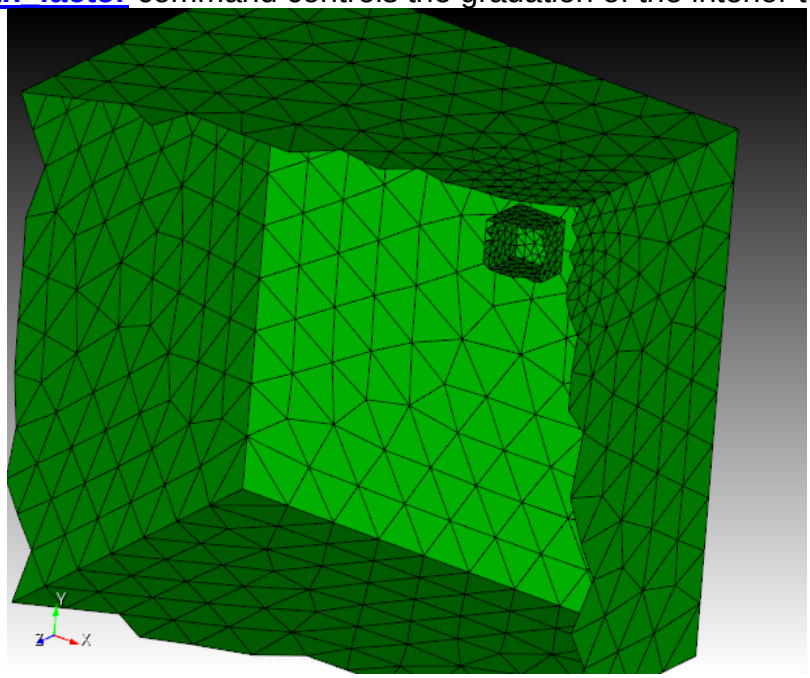



**Figure 5a. An example of a cut-away mesh with a volume gradation, where the small size on the interior void propagates to the exterior surfaces**

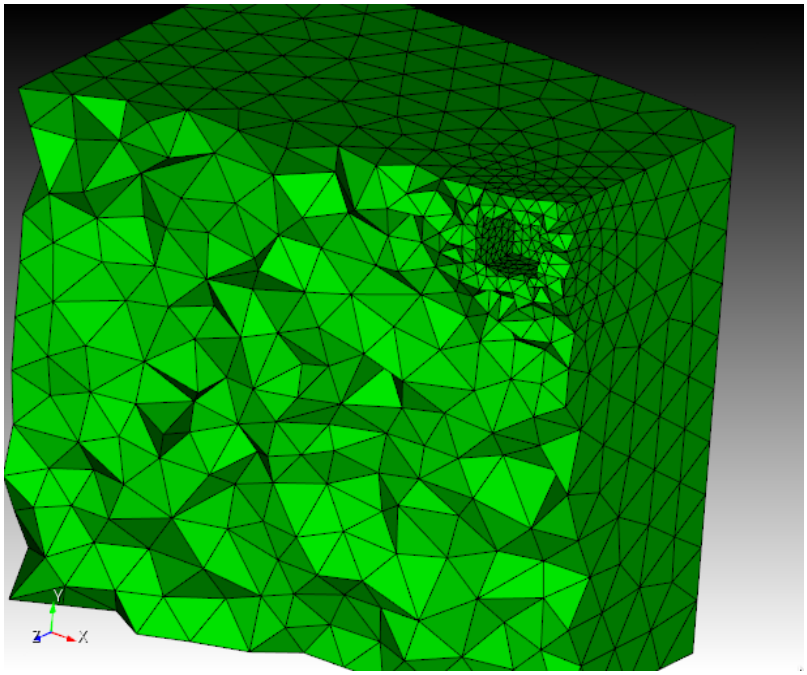


**Figure 5a. An example of a cut-away mesh with a volume gradation, where the small size on the interior void propagates to the exterior surfaces**

### [Set] Trimesher Geometry Sizing {ON|off}

The global **Geometry Sizing** setting can be toggled on or off. If set to **on**, the element size will be influenced by the **geometry approximation angle**. If set to **off**, **geometry approximation angle** will not be involved in the computation of element size. See geometry approximtion angle for more information.

### [Set] Trimesher Discrete Composites {on|OFF}

The option **Discrete Composites** forces trimeshing to convert the composite into a discrete (faceted) representation, from which the trimesher generates a mesh. The default behavior is to use the underlying geometry, if the composite has it, to generate a mesh. Using the underlying geometry is typically a more robust and reliable approach, not dependent on good faceting. However, it should be noted that composites with mesh curves will be triangle meshed with the discrete method, as this is not supported with the default method at this time but shortly will be.

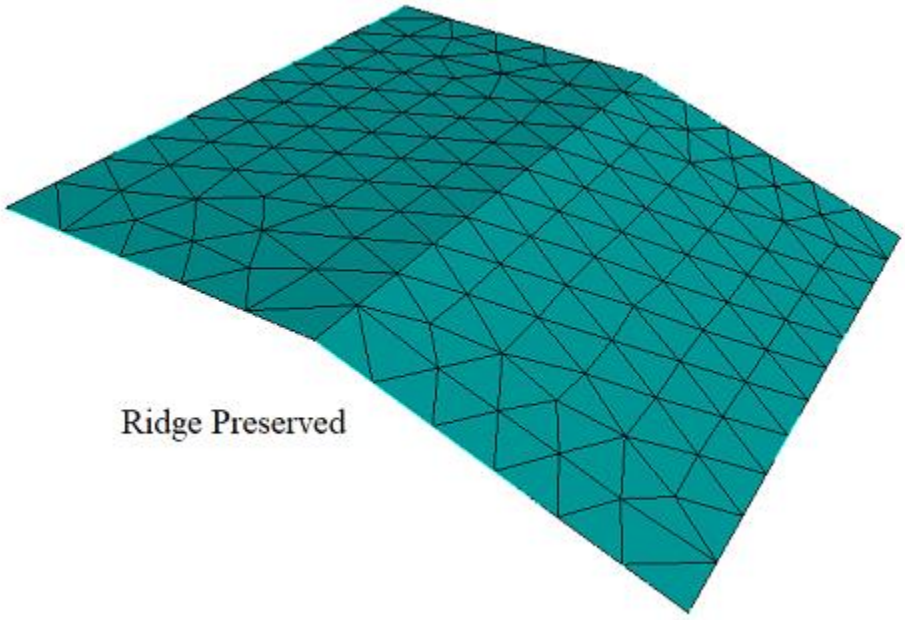
### [Set] Trimesher Clean Discrete {on|OFF}

The option **Clean Discrete** performs a step to 'clean' the faceting given to the trimesher for discrete surfaces (mesh-based geometry and composites), previous to triangle meshing. The 'cleaning' is actually a meshing of the facets which typically generate a better, more optimal set of facets representing the surface. This 'clean' step uses

geometry approximation, with angle of 8 degrees to generate the new facets. The option is OFF by default.

**[Set] Trimesher Ridge Angle {<value=100>}**

The **ridge_angle** setting is only used when meshing discrete surfaces (composites or facet/mesh-based geometry surfaces). It is the threshold for deteremining when to preserve lines (ridges) defined in the discrete surface. Lines in the discrete surface having a dihedral angle larger than **ridge_angle** will be preserved in the generated triangle mesh. In Figure 7, the composite surface has a dihedral angle of 17° at its ridge. In the first image **ridge_angle** is set to less than 17° so the ridge is preserved. In the second image **ridge_angle** is set to greater than 17° so the ridge is not preserved.



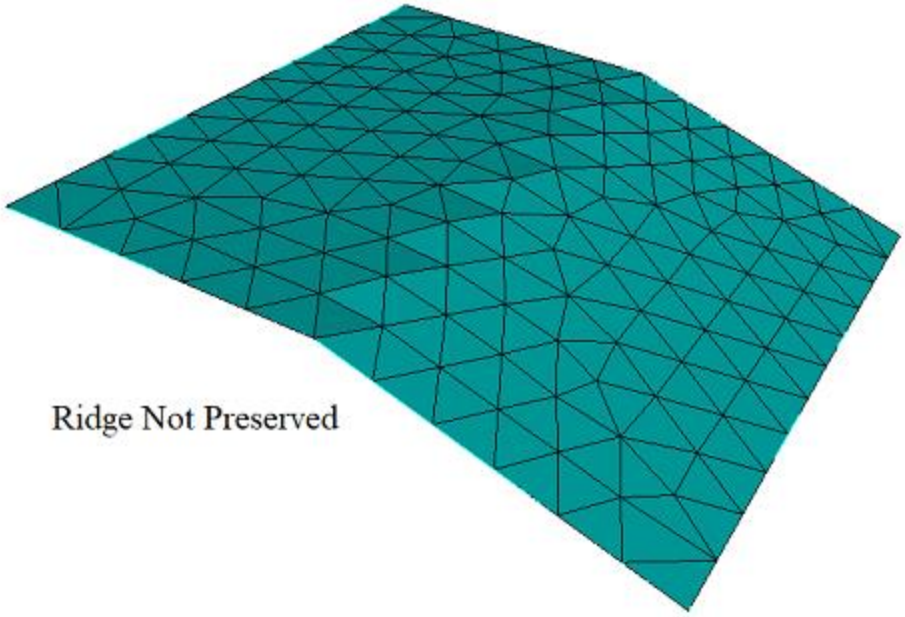



**Figure 6. Ridge Angle Setting**

### [Set] Trimesher Split Overconstrained Edges {on|OFF}

The global **Split Overconstrained Edges**, if set to **on**, splits edges owned by the surface, but with both nodes on curves. This feature can help when two elements through the thickness of the mesh is desired. Figure 7 shows the effect of this option.

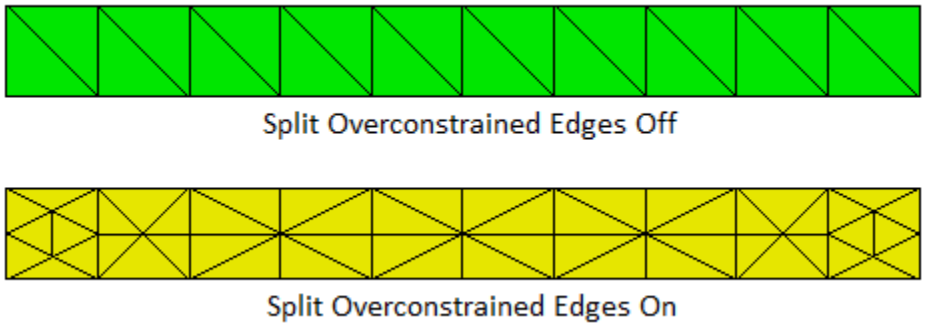



**Figure 7. Split overconstrained edges**

### [Set] Trimesher Anisotropic Layers {on|OFF [<layers=2>]}

The **Anisotropic Layers** setting attempts to place the specified number of layers triangle through thin regions of the surface while respecting the surface mesh size in the thick direction. The default number of layers is two. This option is currently under development and can sometimes generate ill-formed triangles. The number of layers generated can sometimes exceed the number of layers specified..

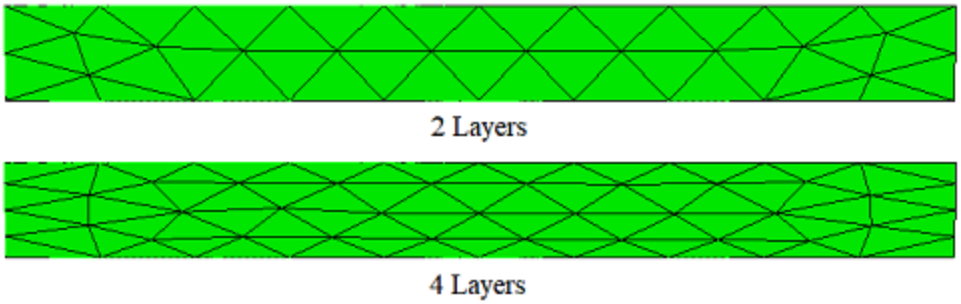



**Figure 8. Anisotropic Surface Meshing**

**[Set] Trimesher Surface Proximity {on|OFF [<ratio=2>]}**

The **Surface Proximity** setting will add refinment to thin regions of surfaces. The ratio option can be used to multiply a scale factor to a size computed from proximity. By default, Surface Proximity is not enabled and if enabled, the default ratio is 1.0.

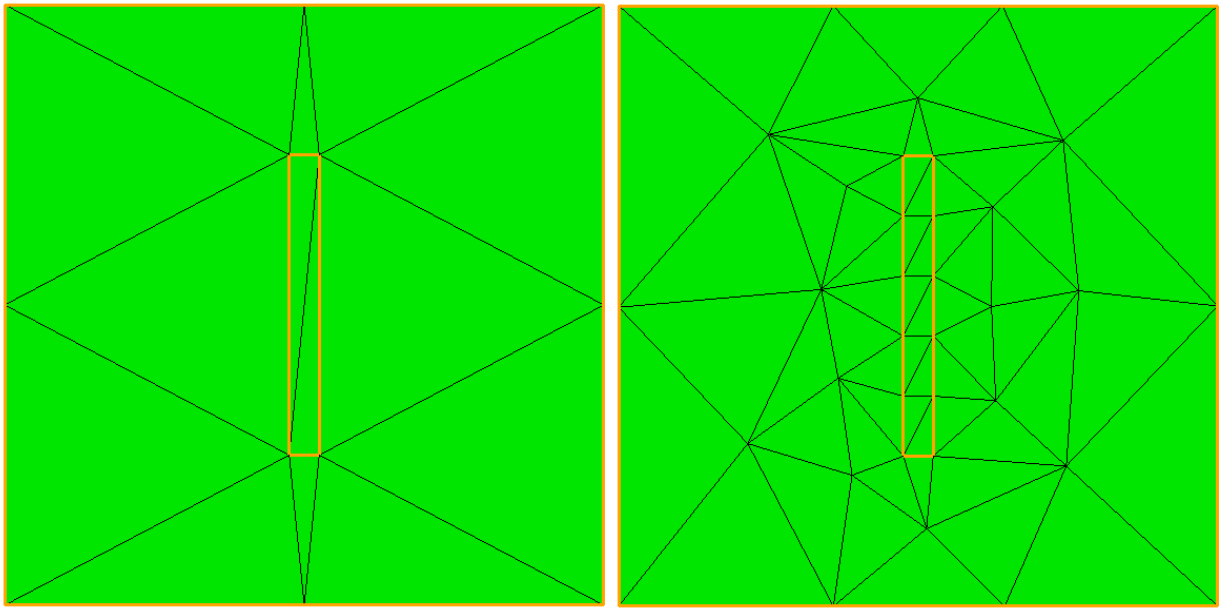


**Figure 9. Surface Proximity disabled on the left, and enabled on the right**

# *TriPave*

**Applies to**: Surface
**Summary**: Places triangle elements at some vertices, and paves the remaining surface.
**Syntax**:

**Surface <range> Scheme Tripave**

**Related Commands:**

**Surface <range> Vertex <range> Type {triangle|notriangle}**

**Discussion**:
Similar to the trimap algorithm, but uses paving instead of mapping to fill the remainder of the surface with quadrilaterals.
The algorithm can automatically compute whether a triangular element is necessary, along with where to place that element. To override the choice of where triangular elements are used, the following command can be issued:

**Surface <range> Vertex <range> Type {triangle|notriangle}**

# *TriPrimitive*

**Applies to**: Surfaces
**Summary**: Produces a triangle-primitive mesh for a surface with three logical sides
**Syntax**:

**Surface <range> Scheme Triprimitive [SMOOTH | nosmoothing]**

**Discussion**:
The triprimitive scheme indicates that the region should be meshed as a triangle. A surface may use the triprimitive scheme if three "natural", or obvious, corners of the surface can be identified. For instance, the surface of a sphere octant (shown in the figure below) is handled nicely by the triprimitive scheme. The algorithm requires that there be at least 6 intervals (2 per side) specified on the curves representing the perimeter of the surface and that the sum of the intervals on any two of the triangle's sides be at least two greater than the number of intervals on the remaining side. The following figure illustrates a triprimitive mesh on a 3D surface.
By default, the triprimitive algorithm will smooth the mesh with an iterative smoothing scheme. This smoothing can be disabled by using the "nosmoothing" option with this command. The quality of the mesh will often be significantly degraded by disabling smoothing, but in certain cases the unsmoothed mesh may be preferred.

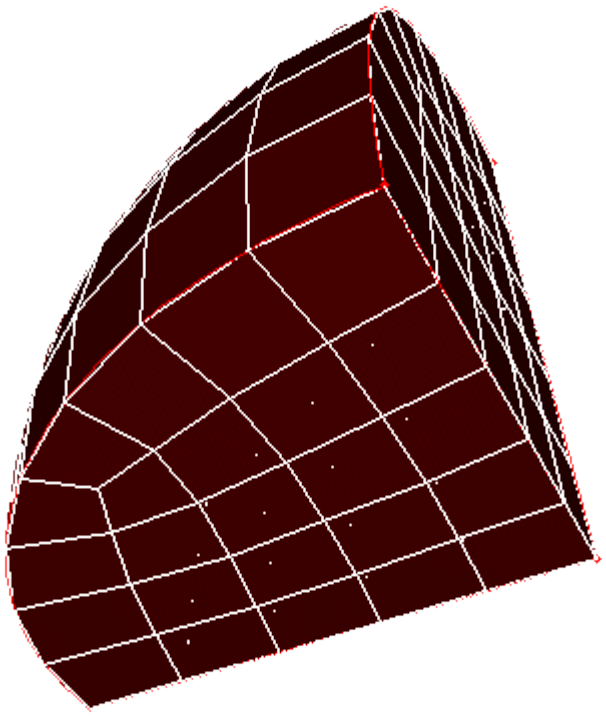


**Figure 1. Surfaces meshed with scheme Triprimitive**

parallel

# *pCamal*

pCamal is an application written and maintained by the Cubit development team. It is designed to work in a distributed computing environment to generate 3D hex elements of a sweep mesh. It first uses the serial Cubit application to generate the 2D quad elements. These elements are written to a file that can then be used by pCamal to generate the most time consuming and memory intensive portion of the mesh: the 3D hex elements. The following describes how to set up the necessary inputs to pCamal using Cubit's sweeping command.
To set up for pCamal, first use the **parallel meshing** setting:

**Set Parallel Meshing {on|OFF}**

You would then use the sweep scheme and mesh your 3D volumes as normal. When Cubit performs the mesh operation on a volume that has a **sweep** scheme applied when the **parallel meshing** option is **ON**, only the surface entities will be meshed, leaving the hex elements for pCamal. Surfaces will be meshed with appropriate source, target and linking surface designations.

#### Exporting a Parallel Mesh for pCAMAL

The following command can be used for exporting a mesh in exodus format for use with pCAMAL

**Export Parallel "<filename>" [Block <id_list>] [Overwrite] [Processor <number>]**

The options are the same as those for the **export genesis** command except for the addition of the processor option.
The processor option allows the user to specify the number of processors that will be used to mesh the volume with the pCAMAL option. This same option exists in the pCAMAL application and is more often used there since the number of available processors is known then rather than when the output file is created in Cubit.
If the processor option is given, Cubit attempts to balance the number of sweepable volumes to run on *N* processors by converting many-to-one sweeps to one-to-one sweeps, subdividing the sweep volume along its sweep direction, or partitioning the source surface of a one-to-one sweep if the number of source quads is much larger than the number of layers.
To determine if you are currently in parallel meshing mode you may list the current status using the List Parallel command.

**List Parallel Meshing**

**Note:** pCamal is not currently distributed with the current release of Cubit. Contact the Cubit developers if you are interested in obtaining a copy of the executable for linux operating systems.

# *Sculpt*

Sculpt is a separate parallel application designed to generate all-hex meshes on complex geometries with little or no user interaction. Sculpt was developed as a separate application so that it can be run independently from Cubit on high performance computing platforms. It was also designed as a separable software library so it can be easily integrated as an in-situ meshing solution within other codes. Cubit provides a front end command line and GUI for the Sculpt application. The command will build the appropriate input files based on the current geometry and can also automatically invoke Sculpt to generate the mesh and bring the mesh back to Cubit.


- Preparing to Use Sculpt
- Sculpt Command
- Controlling the Execution of Sculpt
- Sculpt Help Command
- Sculpt Path Command
- Sculpt Examples

- Sculpt Technical Description
- Sculpt Application Documentation

#### Preparing to Use Sculpt

##### Platforms

Sculpt is available for Windows, Mac and Linux operating systems.

##### Sculpt Installation

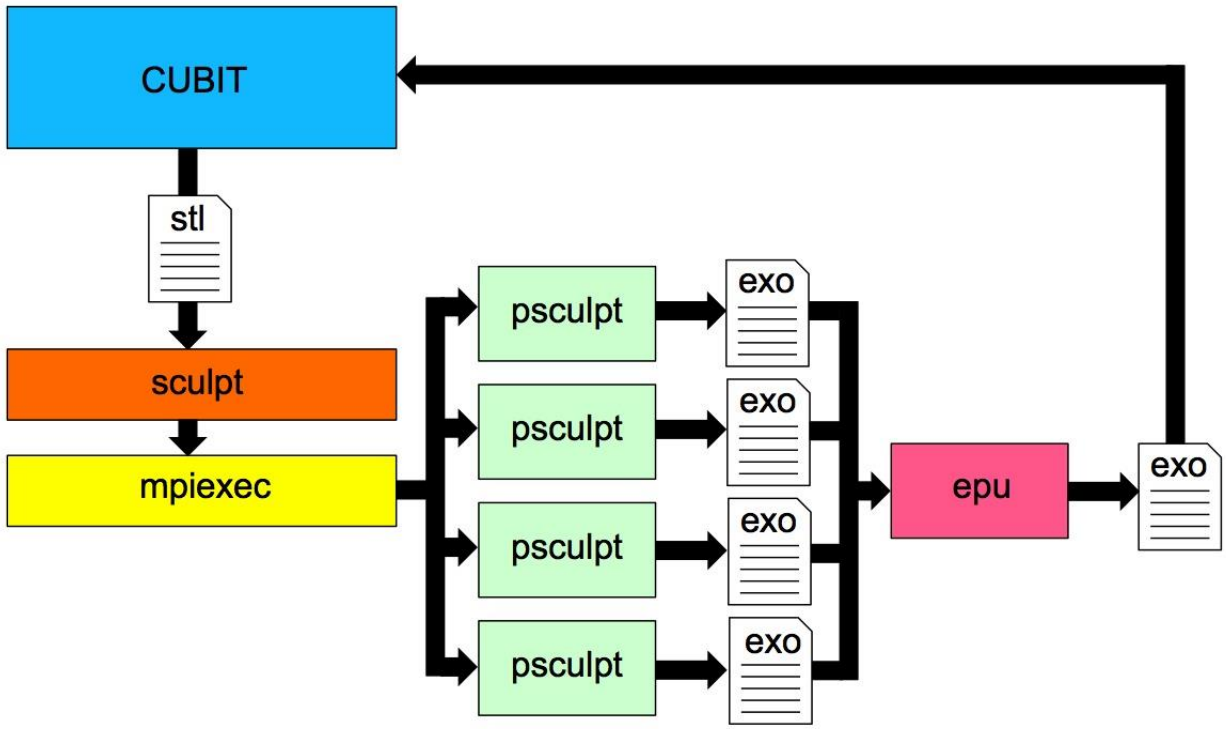
Sculpt is a stand-alone executable, separate from Cubit. In order for Cubit to start up Sculpt, it must be on your system and accessible to Cubit. The default installation of Cubit should install files in the correct locations for this to occur. Check with Cubit support if it did not come with your installation or you are not able to locate it or any of its supporting applications.
To run Sculpt from Cubit, four executable files are needed:

- **sculpt:** Application that controls start-up of **mpiexec** and **psculpt**. Main entry point from Cubit, that checks for the existence and compatibility of either the system **mpiexec** application or will use a local cubit instalation of **mpiexec**.
- **psculpt:** The main mpi-based Sculpt application. Requires **mpiexec** to run.
- **mpiexec:** Standard application available on most linux-based operating systems for starting up mpi-based applications on multiple processors. This should be available with your Cubit installation, but is also available from open-mpi.org
- **epu:** Used for combining multiple exodus files, generated with Sculpt, into a single exodus file. This executable is optional, but is useful for importing the resulting mesh into Cubit for viewing. It is part of the SEACAS tool suite developed by Sandia National Laboratories and is also included with your Cubit installation. It can also be obtained in open source form from sourceforge.net.

To view the current path to these executables that Cubit will use, issue the following command from the Cubit command window

**Sculpt Path List**

See the **Sculpt Path** Command for more info on setting and customizing these paths.
The following image illustrates the process flow when the **sculpt** command is used in Cubit.




For the Sculpt meshing process, a set of files, including a facet-based stl file are written to disk. The **sculpt** application is then started up which in turn invokes **mpiexec** to start up multiple instances of **psculpt** in parallel. **psculpt** then performs the meshing and writes one exodus file per processor. These files can then be combined using **epu** and then imported back into Cubit for viewing.

#### Setting your Working Directory

When using the **Sculpt** command in Cubit, several temporary files will be written to the current working directory. Because of this, it is important to set your working directory before using Sculpt to a desired location where you want these files placed.

#### Sculpt Command

The **Sculpt Command** in Cubit invokes the sculpt application. You can designate the target geometry, which can be existing volumes or blocks within the current Cubit run or imported from external files (e.g., STL, diatom, or microstructure files). The command syntax for preparing a model for Sculpt is as follows:

**Sculpt [parallel]**

**Geometry Options**

**{[volume <ids>] [block <ids>] [stl_file "<filename.stl>"] [diatom_file "<filename.diatom>"] [input_vfrac "<filename.e>"] [input_micro "<filename.tec>"] [input_cart_exo "<filename.e>"] [input_spn "<filename.spn>"]}**

**[spn_xyz_order ""] [input_stitch "<filename.st>"] [stitch_field "<field_name>"] [stitch_timestep {<timestep_value>|first|last}] [stitch_timestep_id <timestep_ID>] [stitch_info]**

**Process Options**

**[processors <value>] [fileroot "<rootfilename>"] [{OVERWRITE|no_overwrite}] [absolute_path] [{EXECUTE|no_execute}] [{COMBINE|no_combine}] [{IMPORT [unique_genesis_ids]|no_import}] [{SHOW|no_show}] [{CLEAN|no_clean}] [{gen_input_file <string>|no_gen_input_file}] [{debug <value>] [quiet] [preview]**

**Grid Options**

**[{size <value>|autosize <value>}] [box {align|location <options>|expand <value>}] [{xintervals|nelx} <value> {yintervals|nely} <value> {zintervals|nelz} <value>] [input_mesh "<filename.g>"] [input_mesh_material <value>] [input_mesh_pamgen "<filename.pam>"]**

**Mesh Options**

**[void <value>] [void_block <value>] [separate_void_blocks] [stair <value>] [htet <value>] [htet_material <value>] [htet_method <value>] [periodic <value>]**

**Smoothing Options**

**[smooth <value>] [csmooth <value>] [num_laplace <value>] [max_opt_iters <value>] [opt_threshold <value>] [curve_opt_thresh <value>] [max_pcol_iters <value>] [pcol_threshold <value>] [max_gq_iters <value>] [gq_threshold <value>] [max_deg_iters <value>] [deg_threshold <value>] [geo_smooth_max_deviation <value>]**

**Improve Options**

**[pillow <value>] [pillow_surfaces] [pillow_curves] [pillow_curve_layers <value>] [pillow_curve_thresh <value>] [pillow_boundaries] [pillow_smooth_off] [defeature <value>] [min_vol_cells <value>] [defeature_bbox] [defeature_iters <value>] [capture <value>] [capture_angle <value>] [capture_side <value>] [thicken_material <value>...] [thicken_void <value>] [remove_bad <value>] [wear_method <value>] [crack_min_elem_thickness <value>] [temp_use_sipe_depth <value>] [min_num_layers <value>]**

**Adapt Options**

**[adapt_type <value>] [adapt_threshold <value>] [adapt_levels <value>] [adapt_material <value>...] [adapt_export] [adapt_non_manifold] [adapt_load_balance]**

**Boundary Condition Options**

**[gen_sidesets <value>] [material_name <value>... <string>...] [sideset_name <value>... <string>...] [nodeset_name <value>... <string>...] [sideset_definition <value>... <string>...] [nodeset_definition <value>... <string>...] [free_surface_sideset <value>...] [match_sidesets <value>...] [match_ss_nodeset <value>...]**

**Output Options**

**[exodus <string>] [large_exodus] [xtranslate <value>] [ytranslate <value>] [ztranslate <value>] [xscale <value>] [yscale <value>] [zscale <value>] [volfrac_file <string>] [quality <string>] [write_geom] [write_mbg] [compare_volume] [compute_ss_stats]**

#### Controlling the Execution of Sculpt in Cubit

The following command options can be used to control the execution of Sculpt from within Cubit when used with the **sculpt** command. Follow the links above for other options that control he behavior of the sculpt algorithms and methods.

**volume <ids> | block <ids>**
List of volumes or blocks to include in the mesh. One file containing a faceted representation (STL) per volume will be generated and saved in the current working directory to be used as input to Sculpt. Each volume will be treated as a separate material within sculpt and a conforming mesh will be generated where volumes touch. If the Block command is used, one file per block will be used. Each block represents a separate material in Sculpt.

**fileroot '<root filename>'**
Root of file names for output. When the **sculpt** command is executed, Cubit will generate multiple files in the working directory used for input to the Sculpt application. The '<root filename>' will be used as the basis for naming these files.

**processors <value>**
Specify the number of processors that MPI will use to execute the Sculpt application. If not specified, the maximum number of available processors on the local machine will be used up to a maximum of 16.

**OVERWRITE | no_overwrite**
By default, Cubit will overwrite an existing set of files with the same '<root filename>'. To over-ride, use the **no_overwrite** option.

**absolute_path**
By default, Cubit will write the relative path names of files used in the .run and .diatom files. To force absolute path names to be written, use the **absolute_path** option

**EXECUTE | no_execute**

By default, Cubit will attempt to run sculpt in parallel on the machine Cubit is currently running on. To generate just the required input to run Sculpt at a later time or on another machine, use this option. A file of the form <root filename>.run will be generated in the current working directory. (for example "model.run"). Executing the .run file from the linux command line should run sculpt in parallel. It can also be used to run sculpt on a cluster where a Cubit executable may not be available.

**size <value> | autosize <value>**

The **size** option is the absolute cell size for the Cartesian grid and is the same as the cell_size option in sculpt. The **autosize** option is a value between 0 and 10. It represents a model independent size where 1 is the small size and 10 is large. This is the same scaling factor used in Cubit's auto sizing but is divided by ten. A size value will be computed from the selected autosize and used as the absolute cell size for the base Cartesian grid.

**box location <options>**

When using a Cartesian grid as the overlay grid definition, the **location** options define the bounds of the Cartesian grid. The first Location <option> defines the minimum Cartesian coordinate of the grid and the second, the maximum. The <options> can be any valid method for defining a coordinate location in cubit. In most cases the **position** option can be used. The default is computed as an enclosing bounding box with 2.5 additional cells on each side.

Note that the **xmin, ymin, zmin, xmax, ymax, zmax** options can also be used for specifying the bounds of the Cartesian grid from the command line.

**COMBINE | no_combine**

If the **no_combine** option is used, following execution of Sculpt, the resulting exodus meshes will not be combined using the **epu seacas** tool. Otherwise the default will automatically combine the meshes generated by each processor into a single mesh. Note that epu should be installed on your system and the path to epu defined using the **sculpt path** command. Epu is a code developed by Sandia National Laboratories and is part of the SEACAS tool suite. It combines multiple Exodus databases produced by a parallel application into a single Exodus database. The epu program should be included with distributions of Cubit beginning with Version 15.0.

**IMPORT | no_import**

If the **no_import** option is used, following execution of Sculpt, the result will be not be imported into Cubit as a free mesh. The default **IMPORT** option will automatically import the mesh that was generated in Sculpt. If the **no_combine** option has been used, then multiple free meshes will be imported with duplicate nodes and faces at processor domain boundaries. Otherwise a single free mesh, the result of the **epu** code, will be imported. Note that the resulting mesh will not be associated with the original geometry, however Block (material) definitions will be maintained. In addition, a separate group will be generated for each imported mesh (One per processor). The default will automatically import the mesh following mesh generation in Sculpt.

**SHOW | no_show**

If the **no_show** option is used, while the external Sculpt process is running, no ouput from the Sculpt application will be displayed to the command window. Otherwise, the default **SHOW** is used and output from the Sculpt application will be echoed to the Cubit command window. This option is only effective if the **no_execute** is not used.

**CLEAN | no_clean**
If the **clean** option is used, temporary files generated during the **sculpt** command will be deleted. This includes any exodus mesh files, .stl, .diatom, .log and .run files. The default for this option is **CLEAN**, therefore, use the **no_clean** option to keep any temporary files generated as part of the current Sculpt run.
**gen_input_file <file name> | no_gen_input_file**
An input file with the given file name will be generated when the command is executed. This is a text file containing all sculpt options used in the command. The input file is intended to be used for batch execution of sculpt. To run sculpt from the operating system command line you would use the **-i** option. For example: **sculpt -i myinputfile.i -j 4** where **myinputfile.i** is the name of the input file specified with the **gen_input_file** option and **-j 4** is the number of processors to use.
**debug <value>** The debug option is used only as a developer debugging tool. It will set the debug processor and sleep upon execution to allow a debugger to be attached to the process.
**quiet**
The **quiet** option used as an option in the **sculpt** command will suppress output from the sculpt application to the Cubit output window.
**preview**
When used with the **sculpt** command, the **preview** option will print the contents of the sculpt input file to the Cubit output window, based on the currently defined options in the **sculpt** command. It will not execute sculpt.

#### Sculpt Help Command

Help about any of the Sculpt options can be printed to the output window using the following command:

**Sculpt [parallel] print_help [<"sculpt_option">]**

where <"sculpt_option"> is any valid sculpt application option listed above.

#### Sculpt Path Command

The command for letting Cubit know where the Sculpt and related applications are located is:

**Sculpt [Parallel] Path [List|Psculpt|Epu|Mpiexec]**

This command defines the path to **psculpt**, **epu** and **mpiexec** on your system. In most cases, however, these paths should be automatically set provided Sculpt was successfully installed with your Cubit installation. The **list** option will list the current paths that Cubit will use for these tools. If an alternate path to these executables is desired, it is recommended that this command be used in the .cubit initialization file so that it wont be necessary to define these parameters every time Cubit is run.

#### Sculpt Mesh Quality Control

In most cases, the Sculpt tool can be used without adjusting default values. Depending on the characteristics of the geometry to be meshed, the default values may not yield adequate mesh quality. Upon completion, Sculpt reports to the command line, a summary of the mesh that was generated. This includes a summary of the mesh quality. Care should be taken to review this summary to ensure the minimum mesh quality is in a range suitable for analysis.
The element metric used for computing mesh quality in Sculpt is the Scaled Jacobian. This is a value between -1 and 1 that is a relative measure of the angles at the element's nodes. A value of 1 indicates a perfect 90 degree angle between each of its edges. In most cases a value less than zero, or negtive Jacobian element, indicates an unusable mesh. Sculpt's default settings try to achieve a minimum Scaled Jacobian of 0.2, which is normally usable in most analysis. The following discussion provides several options for adjusting the model or Sculpt parameters to help improve mesh quality.

1. **Locating poor mesh quality:** When the Sculpt mesh has been imported back into CUBIT it is a good idea to display the element quality. You can do this with variations of the following commands:

   ```
   quality hex all scaled jacobian
   quality hex all draw mesh
   ```

   Identify regions where hexes are poor quality and zoom in to these regions.

2. **Modifying the geometry:** Zooming in to poor quality elements may reveal that the mesh does not adequately represent the underlying geometry. In some cases you may find that small features, or small gaps between parts may be on the order of the size of the Sculpt cell size. If these features are not important to the analysis, you may consider using Cubit's geometry modification tools to remove features or close small gaps.
3. **Modifying the cell size:** In cases where small geometric features or gaps are important to the simulation, it may be necessary to use a smaller base cell size. Use the size or autosize input parameters or increase the numbers of intervals. Normally to adequately capture a feature you would want the cell size to be no greater than about 1/3 to 1/2 the size of the smallest feature you would want to represent in the simulation.
4. **Turning on Pillowing for multiple materials:** For models that have more than one material that share an interface, unless the geometry is precisely aligned with the global axis, it is usually a good idea to turn on pillowing. Pillowing automatically inserts an additional layer of hexes at interface boundaries to improve mesh quality. Without pillowing may notice inverted or poor quality elements at curve interfaces where 2 or more materials meet.
5. **Modifying smoothing parameters:** Sculpt includes a tiered approach to smoothing to improve element quality. It starts by applying smoothing to all nodes in the mesh and progressively restricts the smoothing operations to only those nodes that fall below a user-defined scaled Jacobian threshold. Default numbers of iterations and thresholds for each smoothing phase have been tuned

for general use, however it may be worthwhile to adjust these parameters. The three smoothing phases include:

- **Laplacian Smoothing:** Applied to all elements. Very inexpensive fast approach to improve quality, but can result in degraded element quality if applied to excess. A fixed default of 2 iterations is applied to all hexes. Increasing the num_laplace parameter can improve some cases, especially convex shapes
- **Optimization Smoothing:** Applied only to elements who's scaled Jacobian falls below the opt_threshold parameter (default 0.6) and their surrounding elements. This approach uses a more expensive optimization technique to improve regions of elements simultaneously. The max_opt_iters parameter can control the maximum number of iterations applied (default is 5). Iterations will terminate, however, if no further improvement is detected. Because this method optimizes node locations simultaneously, neighboring nodes with competing optimum can sometimes limit mesh quality.
- **Spot Optimization:** Also known as *parallel coloring*, is applied only to elements who's element quality falls below the pcol_threshold parameter (default 0.2). This technique is the most expensive of the techniques, but focusses only on nodes that are immediately adjacent to poor quality hexes. Each node is smoothed independently of its neighbors, and may require a high number of iterations using the max_pcol_iters to achieve desired results. Increasing the pcol_threshold and max_pcol_iters may yield improved results.

Observing the mesh quality output to the command line following each smoothing iteration can provide some insight on the effect of modifying smoothing parameters.

6. **Creating degenerate hexes:** Some geometries will not permit a usable mesh with a traditional all-hex mesh. Sculpt includes the option to automatically and selectively collapse element edges to improve low-quality elements. The max_deg_iters and the deg_threshold values are used to control the creation of degenerates. Degenerate elements are treated as standard hex elements, but use repeated nodes in the eight-node connectivity array.
7. **Creating hex-dominant mesh** Another option for avoiding mesh quality issues is to generate a few tet elements in the mesh using the htet option. With this option you can specify a scaled Jacobian threshold value below which hexes will be converted to tet elements. The interface between hex and tet elements is managed by an automatically defined set of nodesets and sidesets that describe where multi-point constraints will be applied.
8. **Defeaturing** The defeature option does an initial filter on the cells of the base grid and attempts to reassign the material ID for cells that meet certain criteria. These are cases where a small grouping of cells form a small volume, or where protrusions exist that would otherwise be difficult or impossible to mesh with good quality elements. By reassigning the cells in these locations, in many cases

it will allow the mesh to be acceptable. This operation may result in small changes to the boundary or surface definitions, however usually small enough to still be a reasonable approximation.

#### Sculpt Examples

- Basic Sculpt
- Size and Bounding Box
- Meshing the Void
- Automatic Sideset Definition
- Running Sculpt Stand-Alone
- Meshing Multiple Materials With Sculpt

The following examples use this simple geometry. Execute these commands prior to performing the example **Sculpt** command line operations

```
sphere rad 1
sphere rad 1
vol 2 mov x 2
cylinder rad 1 height 2
vol 3 rota 90 about y
vol 3 mov x 1
unite vol all
```

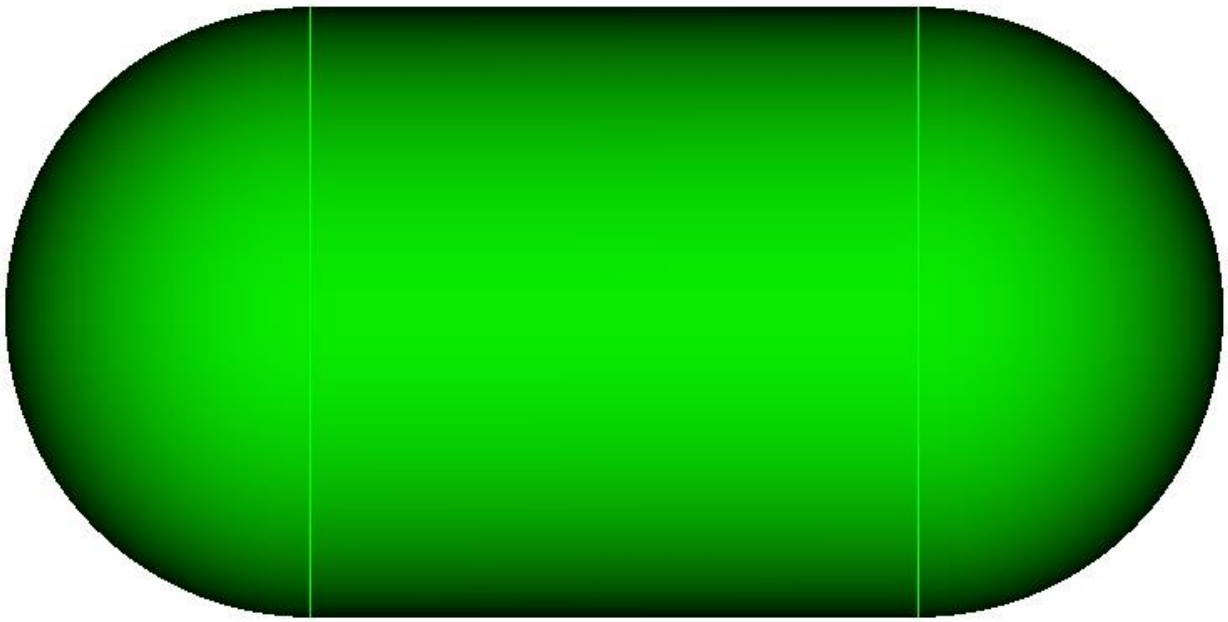


**Figure 1. Geometry created from the above commands and used for the following examples.**

#### Basic Sculpt

This example illustrates use of Sculpt with all default options. So that we can view the result, we will also use the **overwrite**, **combine** and **import** options.

```
sculpt volume 1
draw block all
```

The result of this operation is shown in Figure 2. For this example, behind the scenes, Cubit built an input file for Sculpt, ran it on 4 processors, combined the resulting 4 meshes, and subsequently imported the resulting mesh into Cubit. Note that Volume 1 remains "unmeshed" and we have created a free mesh that is not associated with a volume. The result of any Sculpt command is always an unassociated free mesh.

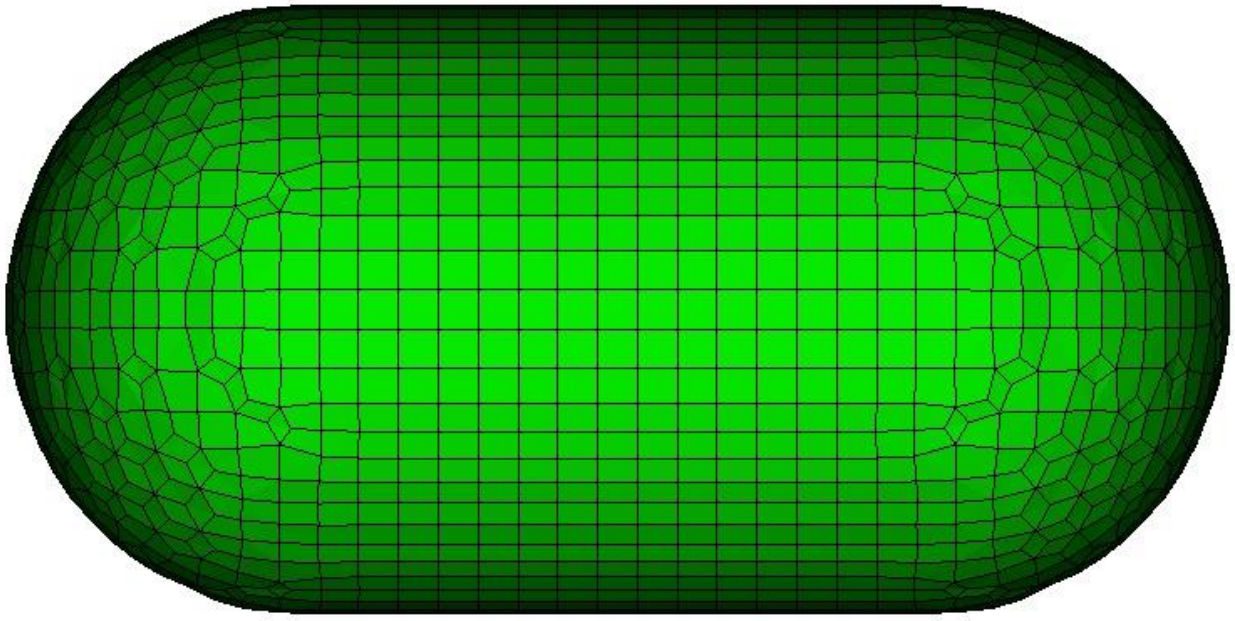


**Figure 2. Free mesh generated from sculpt command**

#### Size and Bounding Box

This example illustrates the use of the **size** and **box** options

```
delete mesh
sculpt volume 1 size 0.1 box location position -1.5 0 -1.5 location position 1
1.5 0
draw block all
```

In this case we have used the **size** option to define the base cell size for the grid. We have also used the **box** option to define a bounding box in which the mesh will be generated. Any geometry falling outside of the bounding box is ignored by Sculpt. Figure 3 shows the mesh generated with this command.

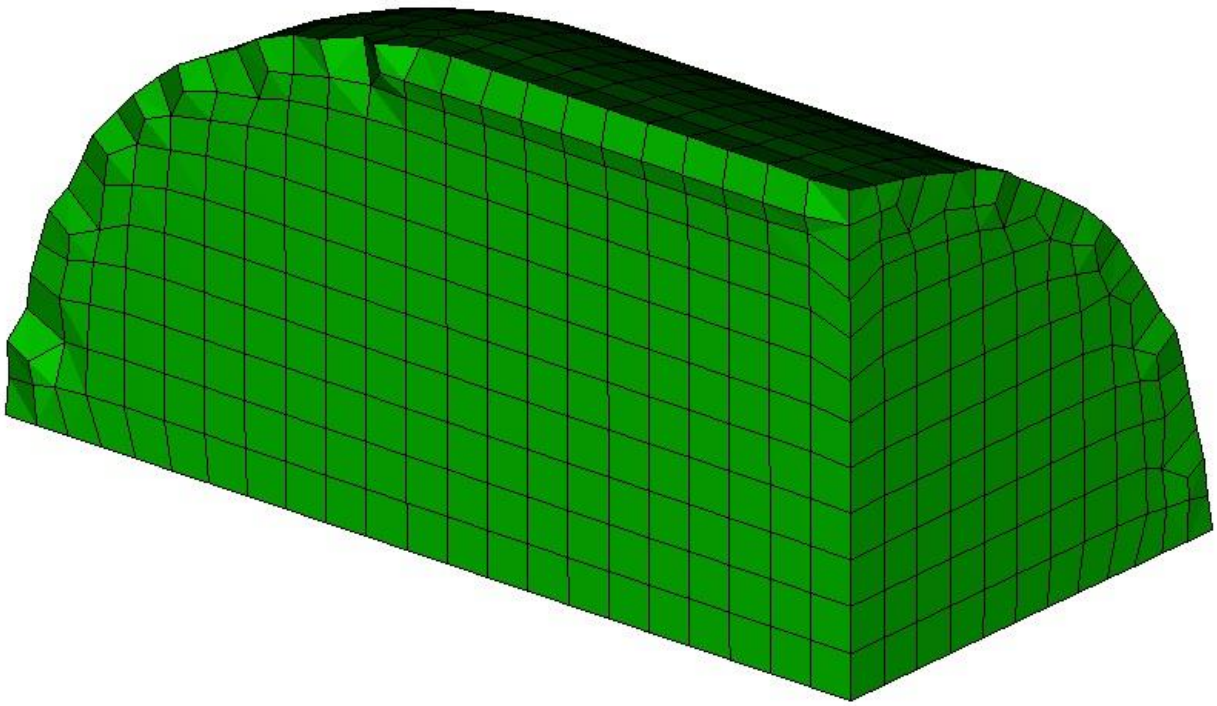


**Figure 3. Sculpt "box" option limits the extent of the generated mesh.**

#### Meshing the Void

In this example we illustrate the use of the **void** option:

```
delete mesh
sculpt volume 1 size 0.1 box location position -1.5 0 -1.5 location position 1
1.5 0 void 1
draw block all
```

The result is shown in figure 4. Notice that this example is precisely the same as the last with the exception of the addition of the **void** option. Mesh is generated in the space surrounding the volume out to the extent of the bounding box. In this case, an additional material block is defined and automatically assigned an ID of 2. The nodes and element faces at the interface between the two blocks are shared between the two materials.

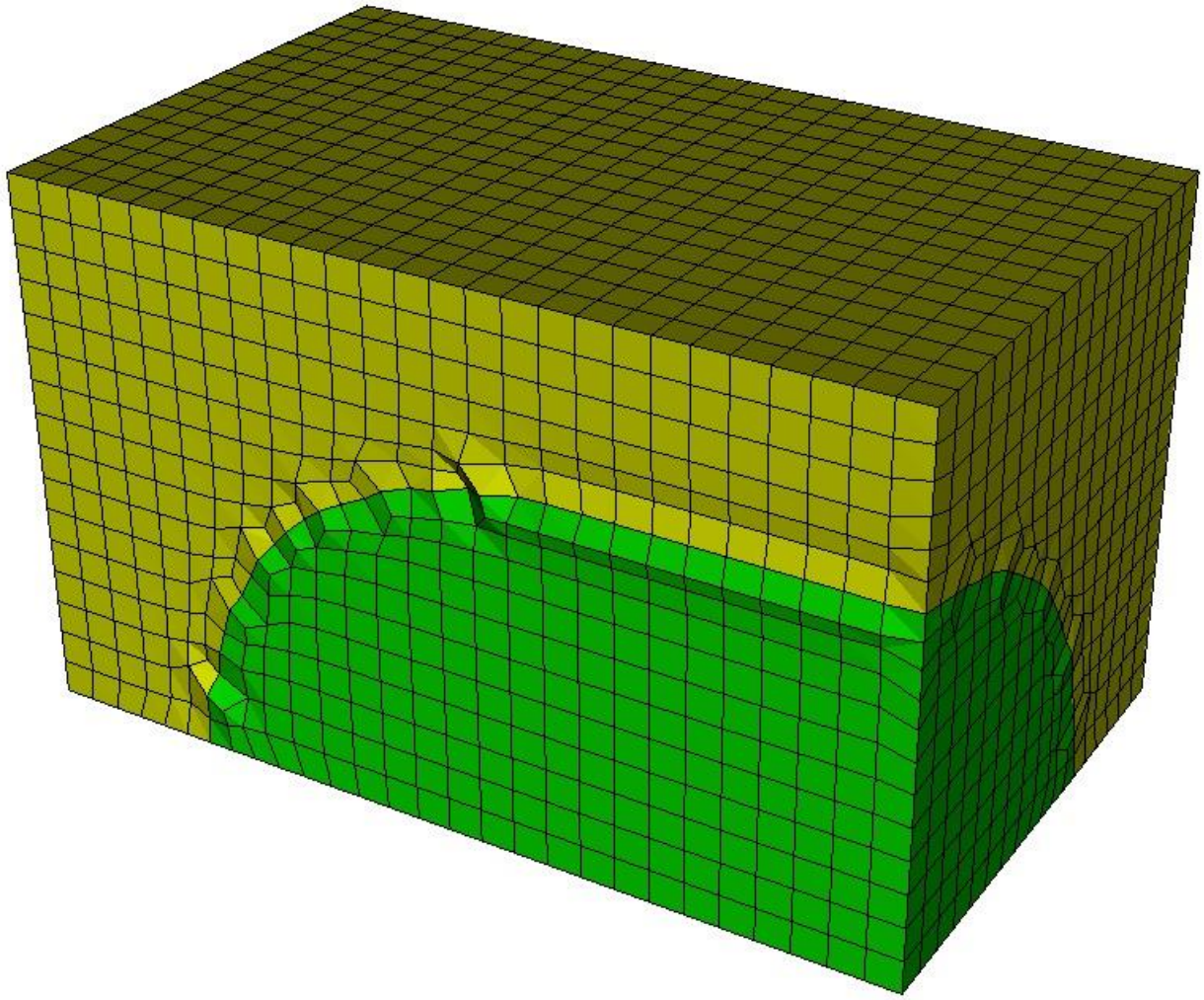


**Figure 4. Sculpt "void" operation generates mesh outside the volume.**

#### Automatic Sideset Definition

In this example we illustrate the use of the **gen_sidesets** option.
**Generating sidesets on the free mesh with Cubit:** Sideset boundary conditions can be manually created on the resulting free mesh from Sculpt using the standard **Sideset <sideset_id> Face <id_range>** syntax. The **Group Seed** command is also useful in grouping faces based on a feature angle to be used in a single sideset.
**Generating sidesets in Sculpt:** Sculpt also provides several options for defining sidesets as part of the Sculpt run. The following illustrates one option:

```
delete mesh
sculpt volume 1 size 0.1 box location position -1.5 0 -1.5 location position 1
1.5 0 void 1 gen_sidesets 2
list sideset all
draw sideset all
```

Once again we use the same syntax but add the **gen_sidesets 2** option to automatically generate a series of sidesets. The **list** command should reveal that 10 sidesets were defined for this example with IDs 1 to 10. Figure 5 shows the result of the **draw** command showing all of the sidesets in different colors. Note that for the **gen_sidesets 2** option, sidesets are created with the following criteria:

- Interfaces between materials
- Exterior surfaces
- Surfaces at the domain boundary

See the gen_sidesets option above for a description of other options for generating sidesets in Sculpt.

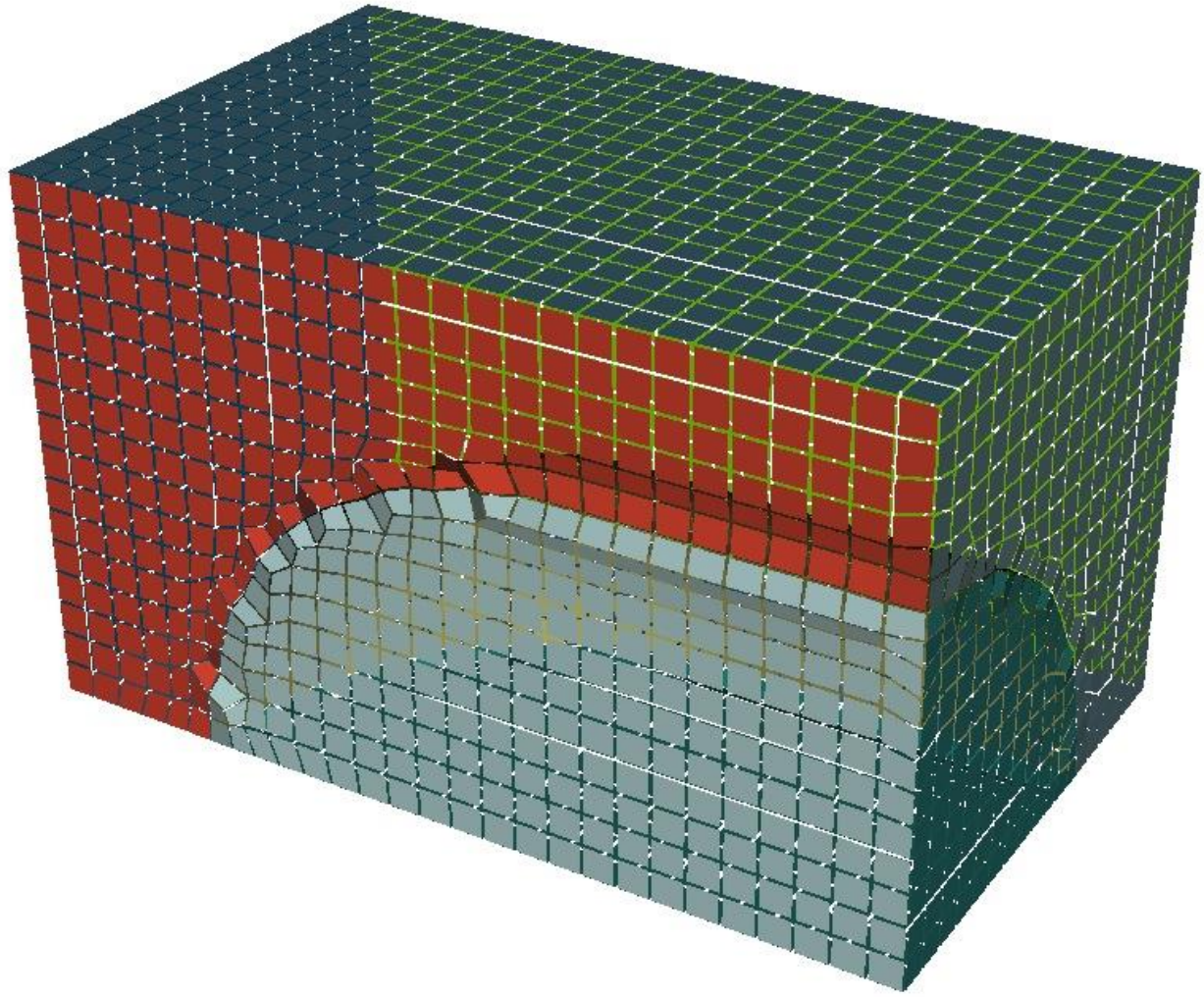


**Figure 5. Automatic sidesets created using Sculpt**

**Running Sculpt Stand-Alone**

This example illustrates how to set up the files necessary to run Sculpt as a stand-alone process. This can be done on the same desktop machine or moved to a larger cluster machine more suited for parallel processing.
Begin by setting your working directory to a location that is convenient for placing example files

```
cd "path/to/my/sculpt/examples"
```

Next we issue the basic **sculpt** command to mesh the volume

```
delete mesh
sculpt volume 1 processors 8 fileroot "bean" over no_execute no_clean
```

In this case, we used the **no_execute** option which does not invoke the Sculpt application. Instead it will write a series of files to the working directory. The **fileroot** option defines the base file name for the files that will be written; in this case **"bean"**. We also use the **processors** option to set the number of processors to be used to 8. Finally, since the default **clean** option will remove temporary files after execution of sculpt, we use the **no_clean** option to ensure they will persist.
To see the files that Cubit placed in the working directory, bring up a terminal window on your desktop and change directories to the current working directory (ie. **cd path/to/my/sculpt/examples**). A directory listing should reveal 3 files as shown in Figure 6.

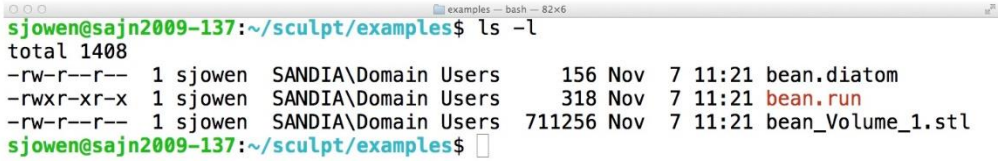



**Figure 6. Directory listing of files written from Cubit**

The following describes the purpose of each of the resulting files:

- **bean.diatom:** *Diatoms* is a file format used by Sandia's CTH and Alegra analysis programs that includes a rich constructive solid geometry definition. A series of directives for constructing and orienting primitives to build a complete solid model can be used. Included in the Diatom description is an STL import option. While any standard Diatom description may be used as input to Sculpt, for Cubit's purposes, only the STL option is used. This file contains a listing of all STL files that will be used as input to Sculpt.
- **bean.run:** The *.run* file contains the unix command line for running sculpt. Note that the file permissions have been set to execute to allow this file to be used as a unix script. Figure 7 shows the .run file for this example. Note that the command uses **mpiexec** and the **psculpt** executables, along with their full path. These paths may need to be edited when running on a different machine. It also includes the default parameters for setting the sizes, bounding box and smoothing parameters that have been computed by Cubit.

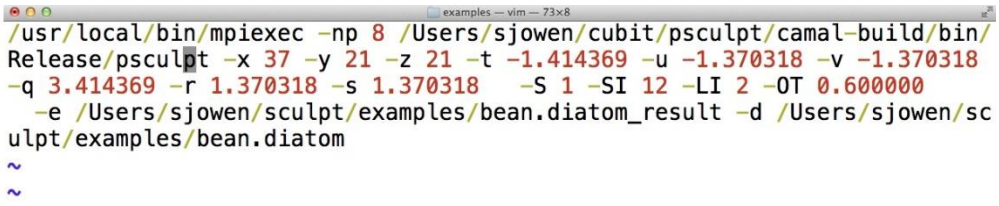



**Figure 7. Unix command line for running Sculpt generated by Cubit**

- **bean_Volume_1.stl:** The STL file is a copy of the geometric model. In our case, it is a representation of the cylinder and sphere object we have been working with. The STL format is a set of triangles that describe the surfaces of the object. One STL file will be generated for each Volume. If the Block option is used, then one file for each Block would be created.

To run sculpt on the same machine, from the terminal window in your current working directory you would issue the following command:

```
./bean.run
```

If Sculpt is to be run on a different machine, copy the files in the working directory to the other machine and issue the same command. Remember to change the path to the **mpiexec** and **psculpt** executables to match those on the new machine. For running on cluster machines that have scheduling of resources, check with your system administrator for how to submit a job for running.
After running Sculpt, Figure 8 shows the resulting files that would be written to the current working directory.

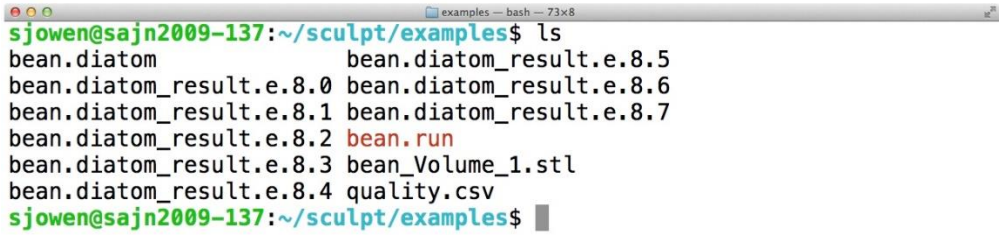



**Figure 8. 8 Exodus files were generated and placed in working directory**

Note that 8 exodus files have been generated, 1 from each processor. These files can be used by themselves or used as-is for use in a simlation, or they can be combined into a single file. The exodus files produced by Sculpt include all appropriate parallel communication information as defined by the **Nemesis** format. Nemesis is an extension of Sandia's Exodus II format that also includes appropriate parallel communication information.
To combine the resulting exodus files into a single file, we can use the **epu** tool. Epu should be included in your Cubit distribution, but may require you to set up appropriate paths for it to be recognized. To run epu on this model, use the following command from a unix terminal window:

```
epu -p 8 bean.diatom_result
```

The result should be a single file with the name **bean.diatom_result.e**. The mesh in this file can then be imported into Cubit. Switch back to your Cubit application and from the command line type the following command:

```
import mesh "bean.diatom_result.e" no_geom
```

The result should be the same mesh we generated previously that is shown in Figure 2.

#### Meshing Multiple Materials With Sculpt

This example illustrates using Sculpt to mesh models with multiple materials. To begin with, we will modify our current model by adding some additional volumes. Use the following commands:

```
delete mesh
cylinder rad 0.5 height 3
cylinder rad 0.5 height 3
vol 5 mov x 2
```

The resulting geometry should look like the image in Figure 9.

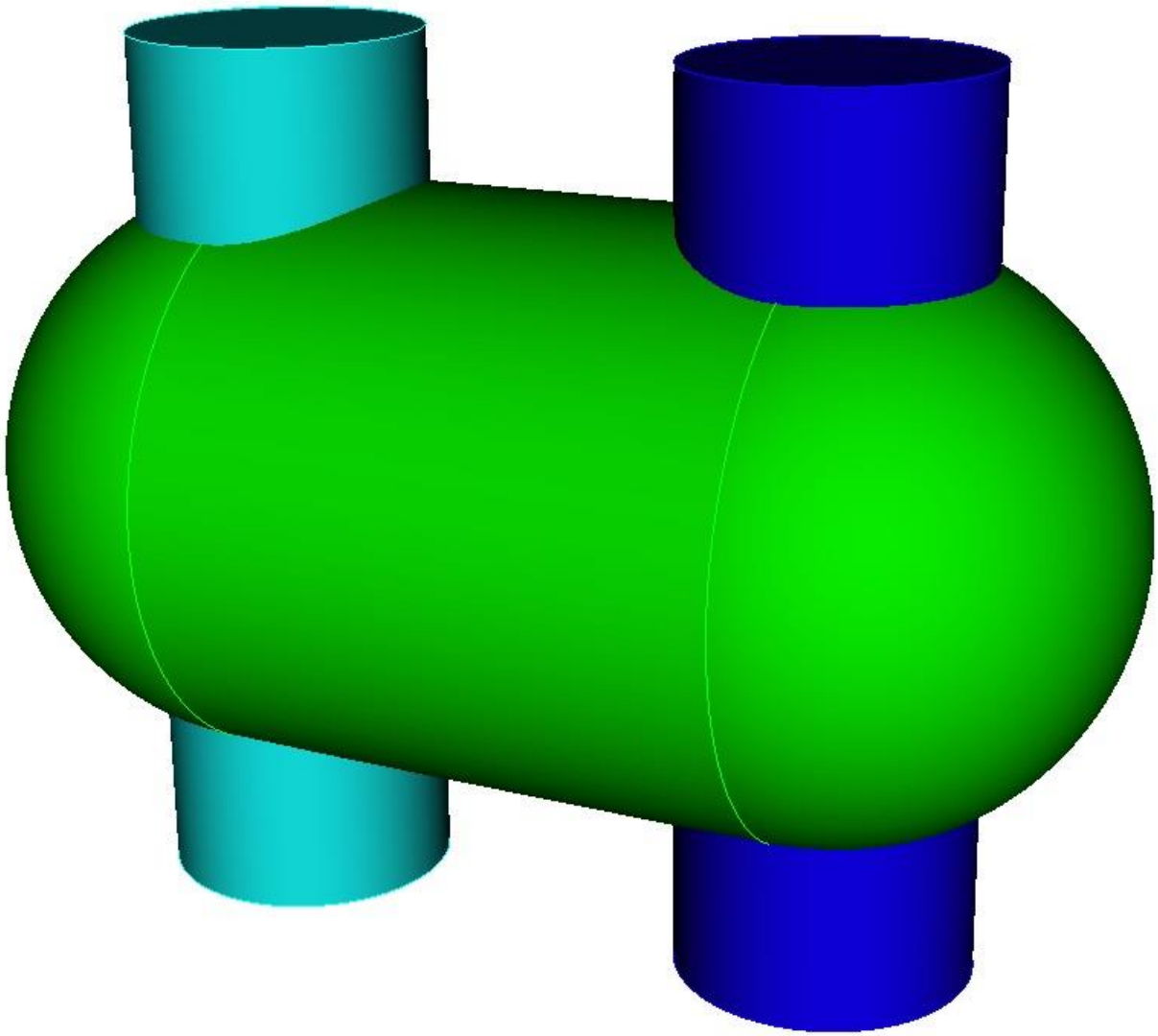


**Figure 9. Geometry used to demonstrate multiple materials with Sculpt**

Use this geometry to generate a mesh using Sculpt.

```
sculpt volume all size 0.075
draw block all
```

The resulting mesh should look like the image in Figure 10.

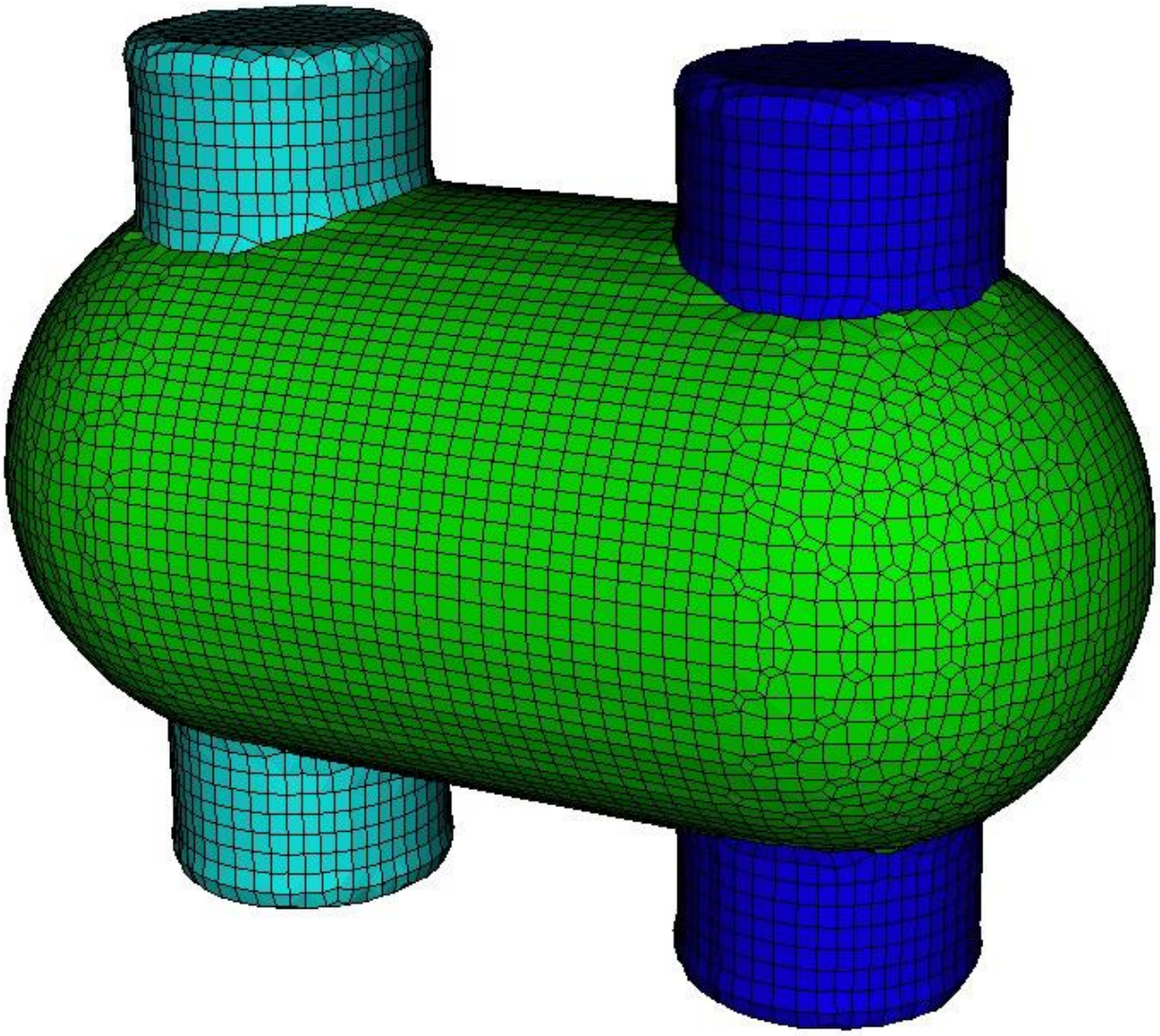


**Figure 10. Mesh generated on multiple materials**

Notice that one mesh block per volume was created. We should also note that no boolean operations were performed prior to building the mesh with Sculpt. In fact, volumes 4 and 5 were significantly overlapping volume 1. This would be an invalid condition for normal Cubit meshing operations. Figure 11 shows a cut-away image of the mesh using the Clipping Plane tool.

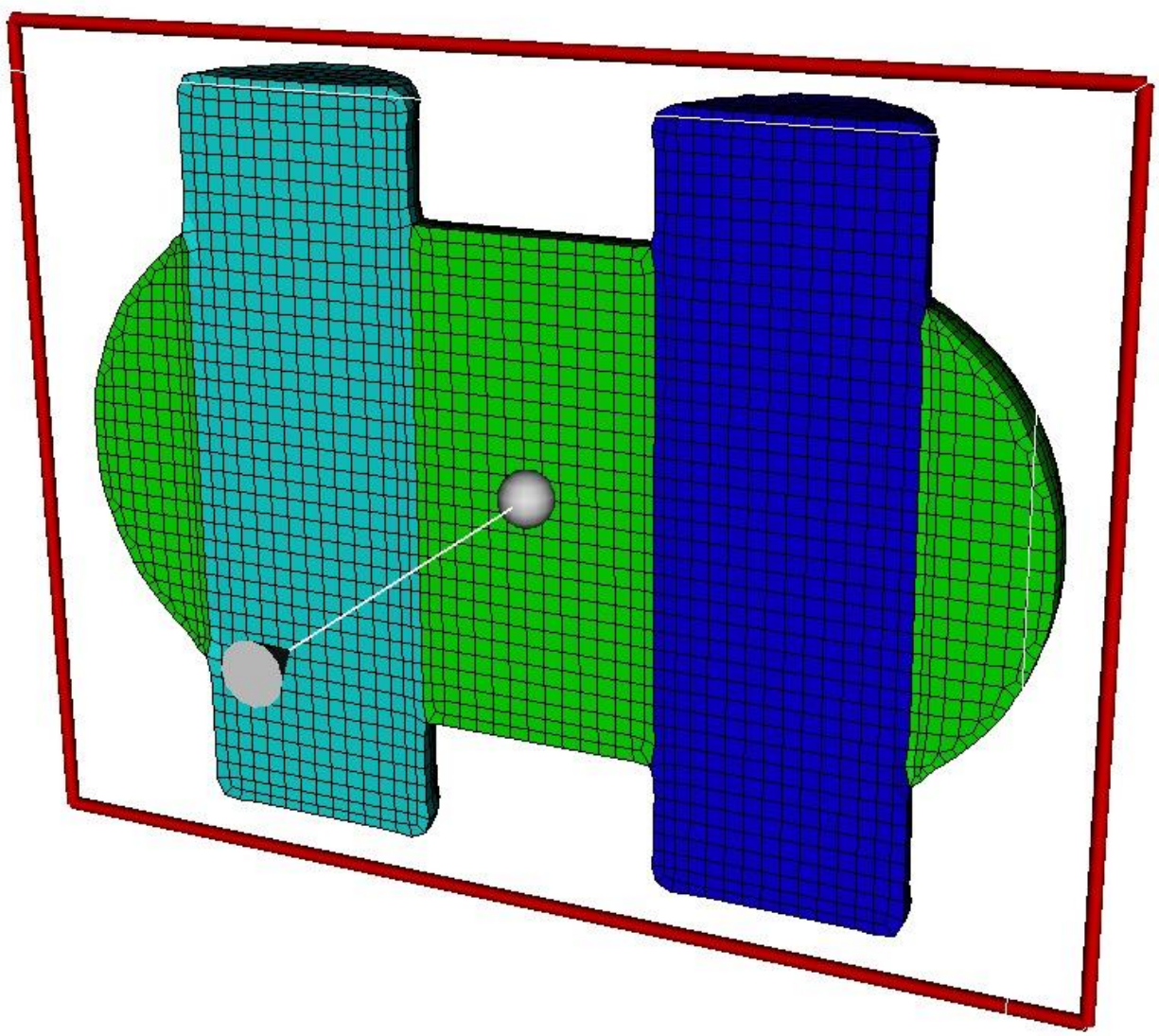


**Figure 11. Cut-away of mesh generated on multiple materials**

We should also note that imprint/merge operations typically needed, were also not required. While it is usually best to avoid overlaps to avoid ambiguities in the topology, Sculpt is able to generate a mesh giving precedence to the most recently defined materials. Merging is performed strictly by geometric proximity. Volumes closer than about one half the user input size will normally be automatically merged.

Next, we will examine the mesh quality of the free mesh. The following command will display a graphical representation of the Scaled Jacobian metric.

```
quality hex all scaled jacobian draw mesh
```

The result is shown in Figure 12. Note the elements (colored red) at the interface between materials are unacceptable for simulation. This is caused by the Sculpt algorithm projecting nodes to a common curve interface shared by the materials.

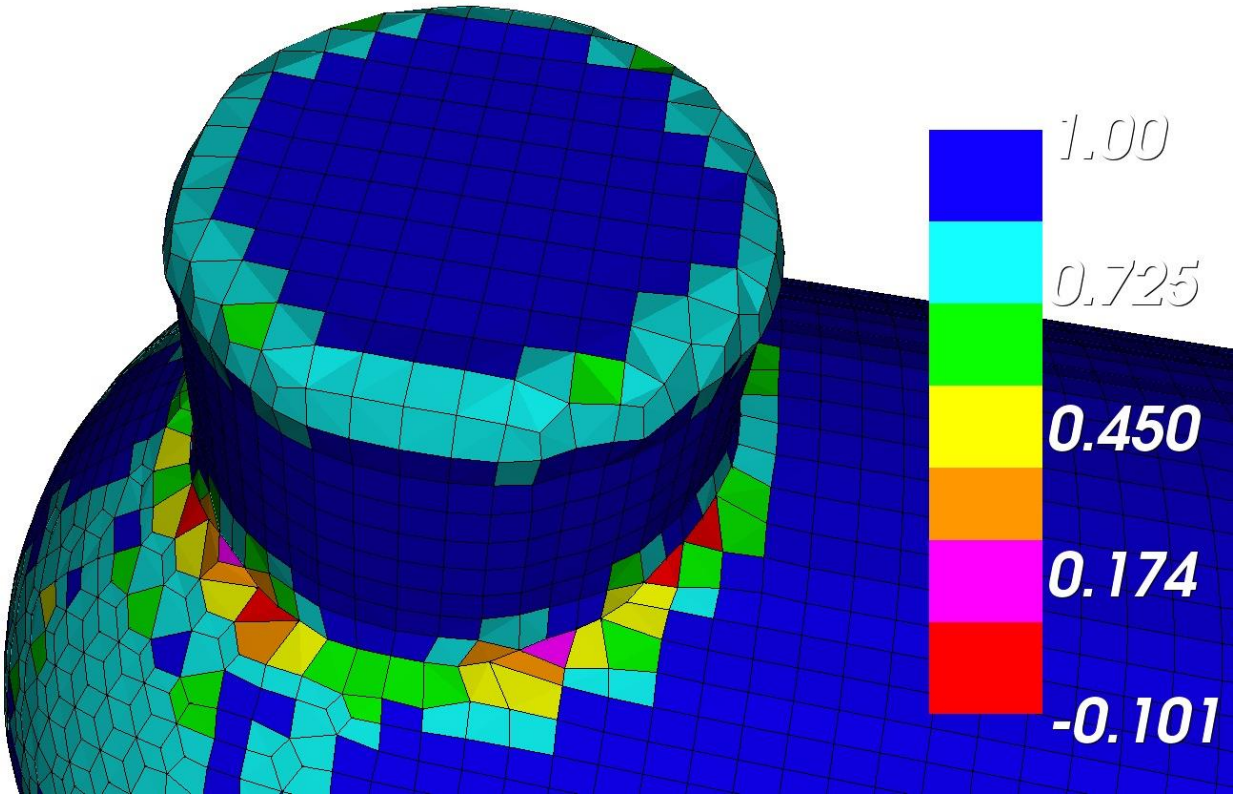



**Figure 12. Mesh quality of multi-material mesh.**

In most cases, the poor element quality at material interfaces can be improved by using the **pillow** option. Adding this option will direct Sculpt to add an additional layer of elements surrounding each surface. To see the result of pillowing, issue the following commands:

```
delete mesh
sculpt volume all size 0.075 over combine import pillow 1
quality hex all scaled jacobian draw mesh
```

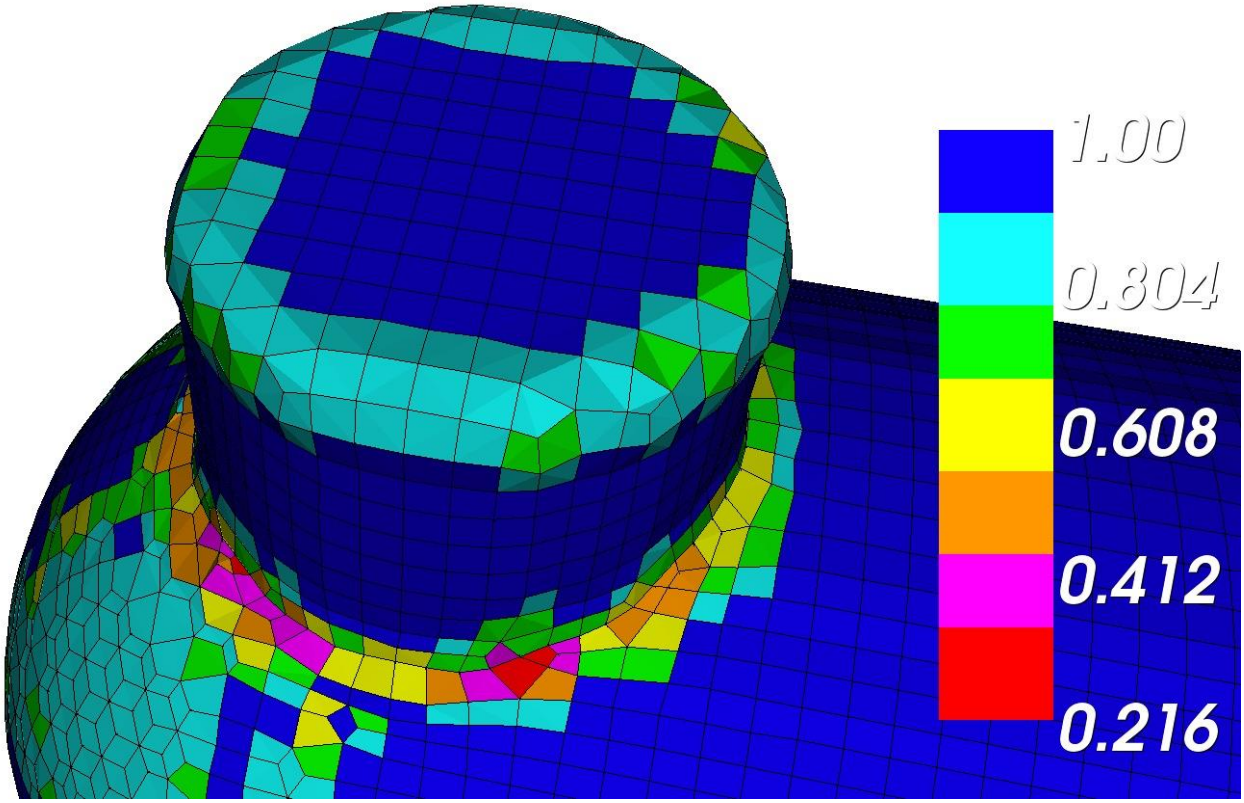



**Figure 13. Mesh quality of multi-material mesh using pillow option**

The resulting mesh is showed in Figure 13. Note the improved mesh quality at the shared curve interface. A closer look at the mesh, shown in Figure 14. reveals the additional layer of hexes surrounding each surface that allows for improved mesh quality when compared with Figure 11. When generating meshes with multiple materials that must share common interfaces, the **pillow** option is usually recommended.

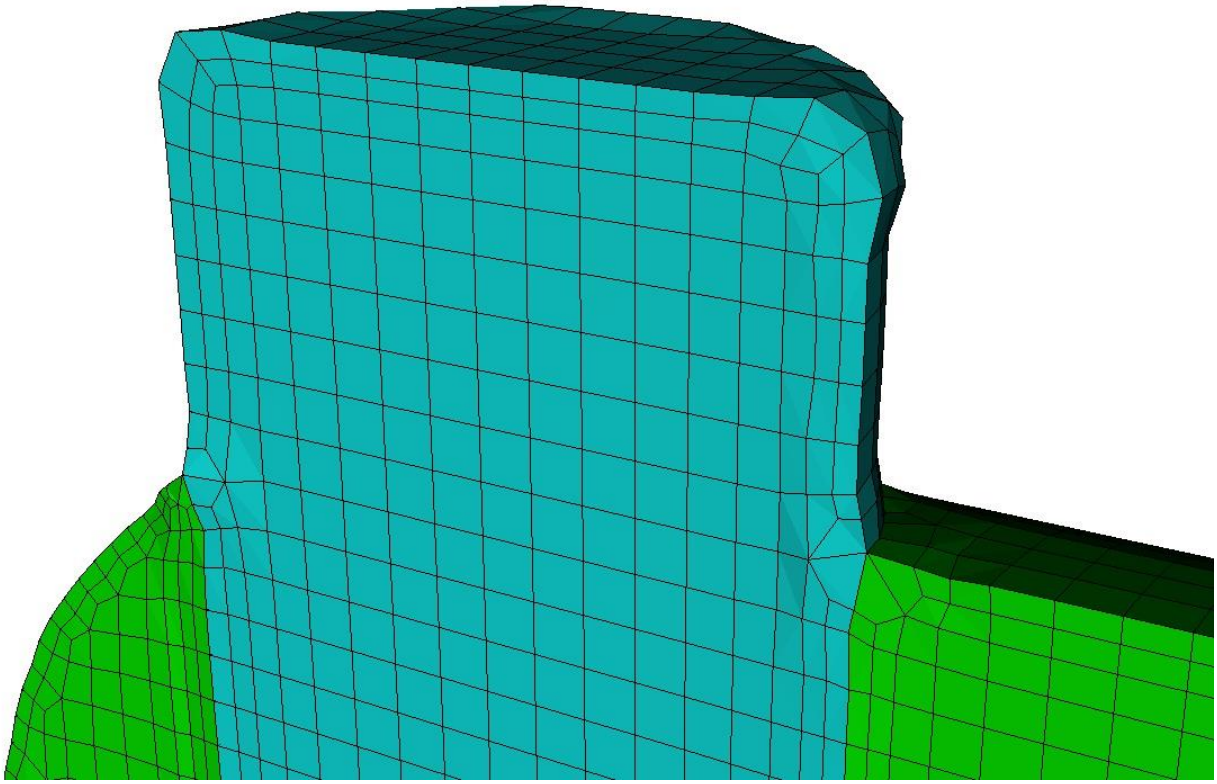


**Figure 14. Cutaway of mesh reveals the additional layer of hexes surrounding each surface when the pillow option is used.**

# *Sculpt Technical Description*

This document provides a brief technical overview of the Sculpt application, a separate companion application to Cubit designed to generate all-hex meshes of complex geometries. Details on command arguments to Sculpt may be found here. Also information for using Cubit to set up input for Sculpt may be found here.

The method for generating an all-hex mesh employed by Sculpt is often referred to in the literature as an *overlay-grid* or *mesh-first* method. This differs significantly from the algorithms employed by Sweeping and Mapping, which are classified as *geometry-first* methods. Mapping and Sweeping start with the geometry, carefully fitting logical groupings of hexes to conform to a recognized topology. In contrast, the Sculpt method begins with a base Cartesian grid encompassing the geometry which is used as the basis for the mesh. Geometric features are carved or sculpted from the Cartesian grid and boundaries smoothed to create the final hex mesh. The obvious benefit of the Sculpt (*mesh-first*) method over Mapping and Sweeping (*geometry-first*) methods is there is no need to decompose the geometry into mappable or sweebable components, a process that can often be very time consuming, tedious and sometimes impossible. Input to Sculpt can be any geometry regardless of features and complexity.

The basic Sculpt procedure is illustrated in figure 1. Beginning with a Cartesian grid as the base mesh, shown in figure 1(a), a geometric description is imposed. Nodes from the

base grid that are near the boundaries are projected to the geometry, locally distorting the nearby hex cells (figure 1(b)). A pillow layer of hexes is then inserted at the surfaces by duplicating the interface nodes on either side of the boundaries and inserting hexes (figures 1(c) and (d)). While constraining node locations to remain on the interfaces, smoothing procedures can now be employed to improve mesh quality of nearby hexes (figure 1(e)).

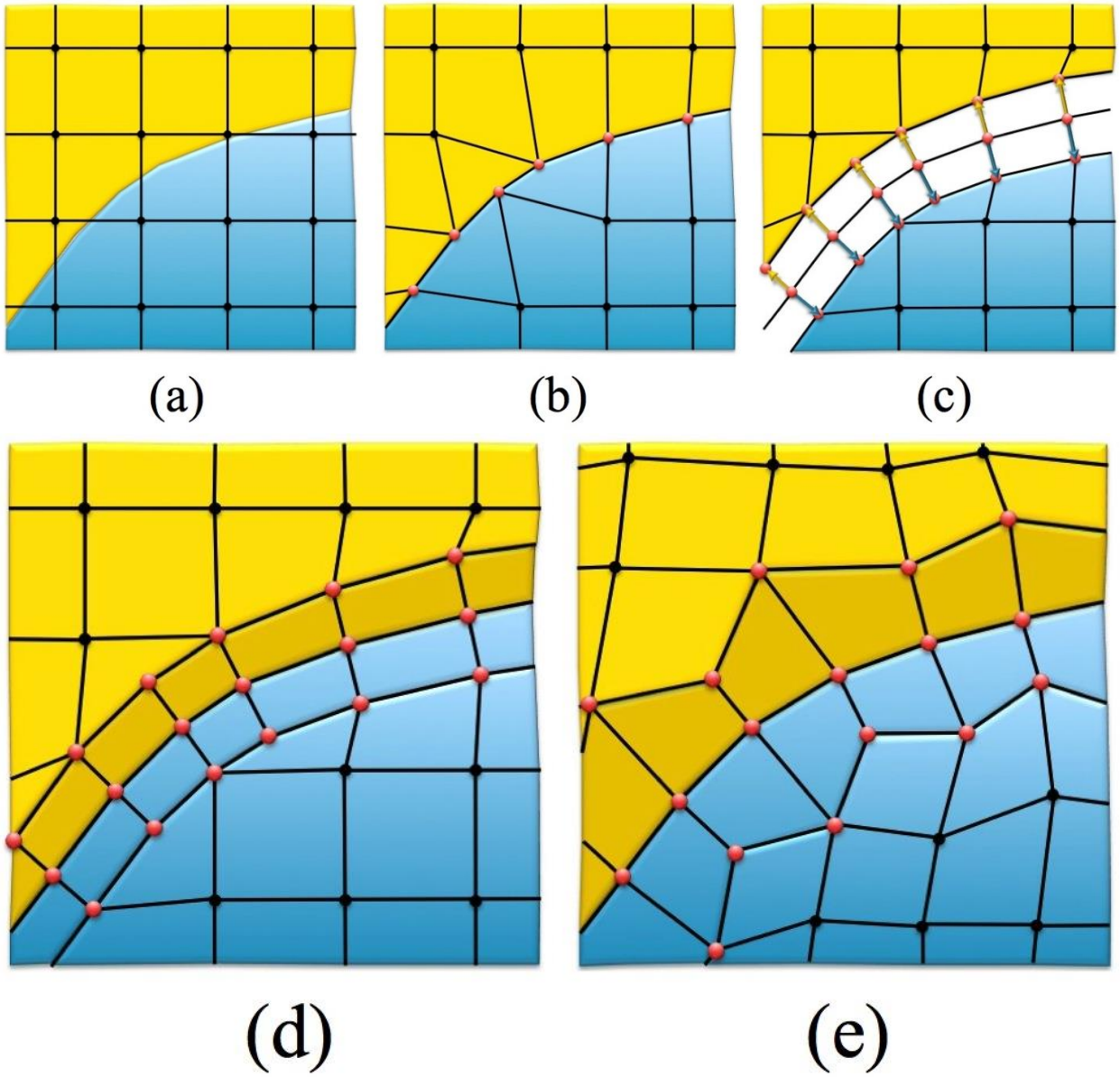



**Figure 1. The procedure for generating a hex mesh using the Sculpt overlay grid method**

Sculpt is limited to capturing geometric features with the available resolution of the selected base mesh. Because of this, care should be taken in selecting an appropriate cell size. In addition, no attempt is made by the Sculpt procedure to capture sharp exterior features. Figure 2 shows an example of a sculpt mesh of a CAD model. Note that exterior corner features are rounded, however the effect of sharp feature capture becomes less pronounced as resolution increases as demonstrated in figures 3(a) and (b).

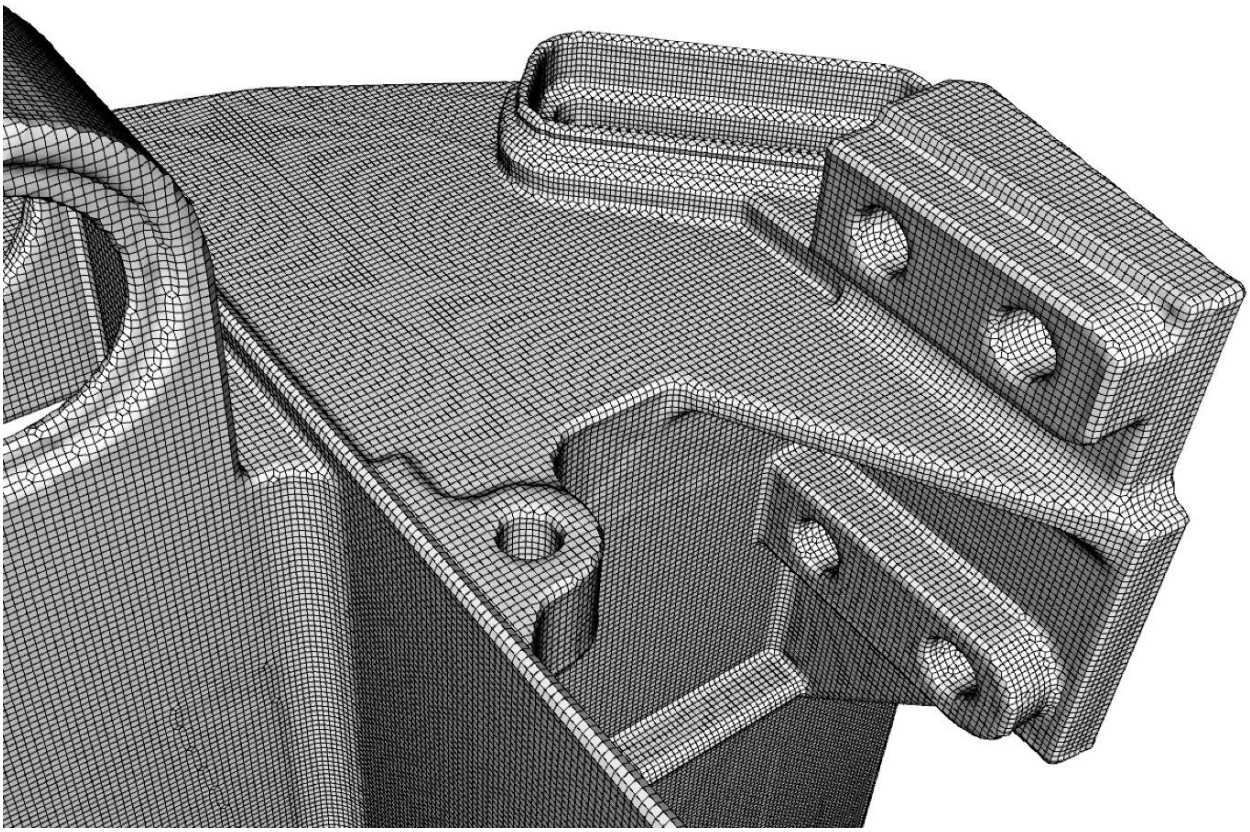
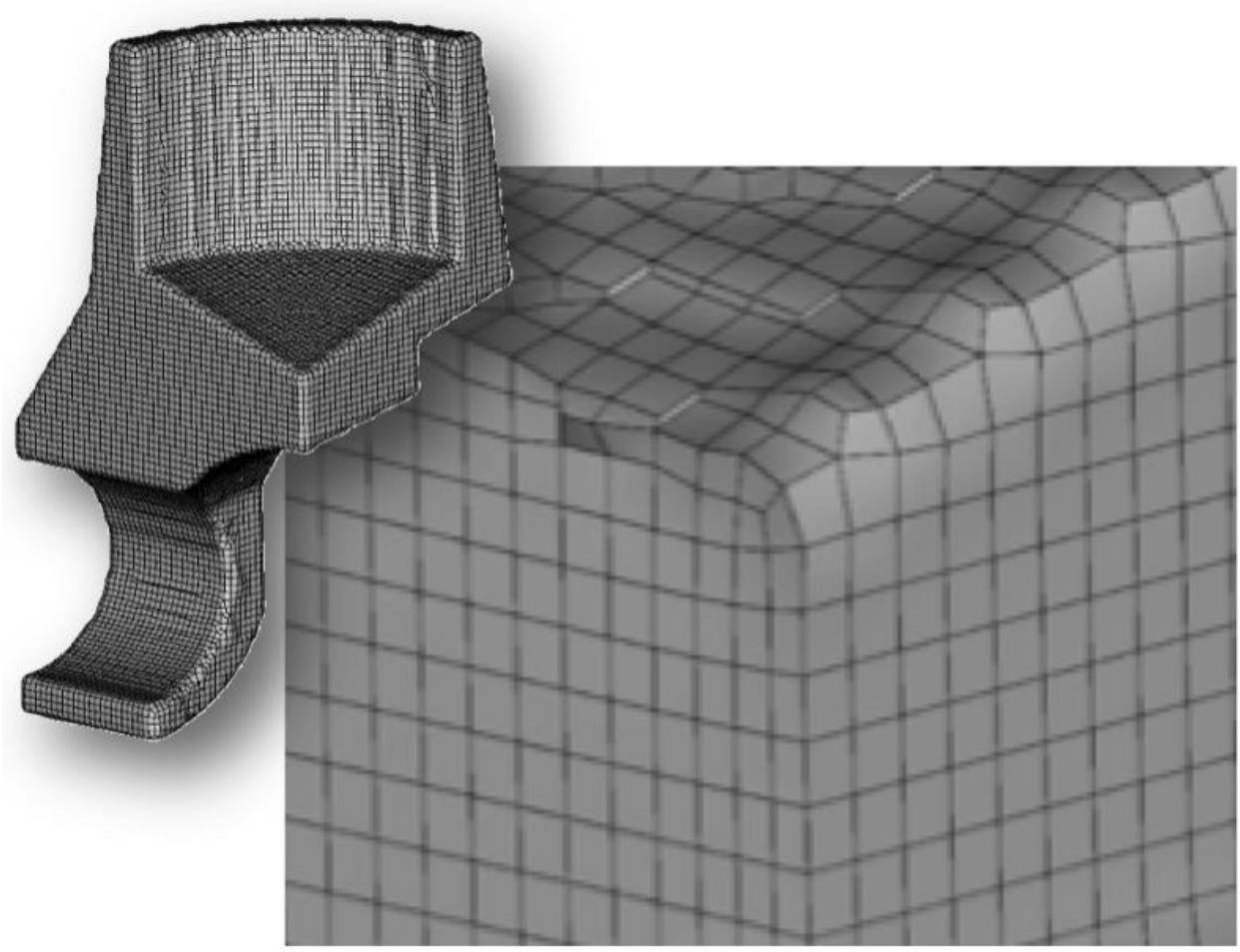


**Figure 2. Hex mesh generated using the Sculpt overlay grid procedure**



(a)

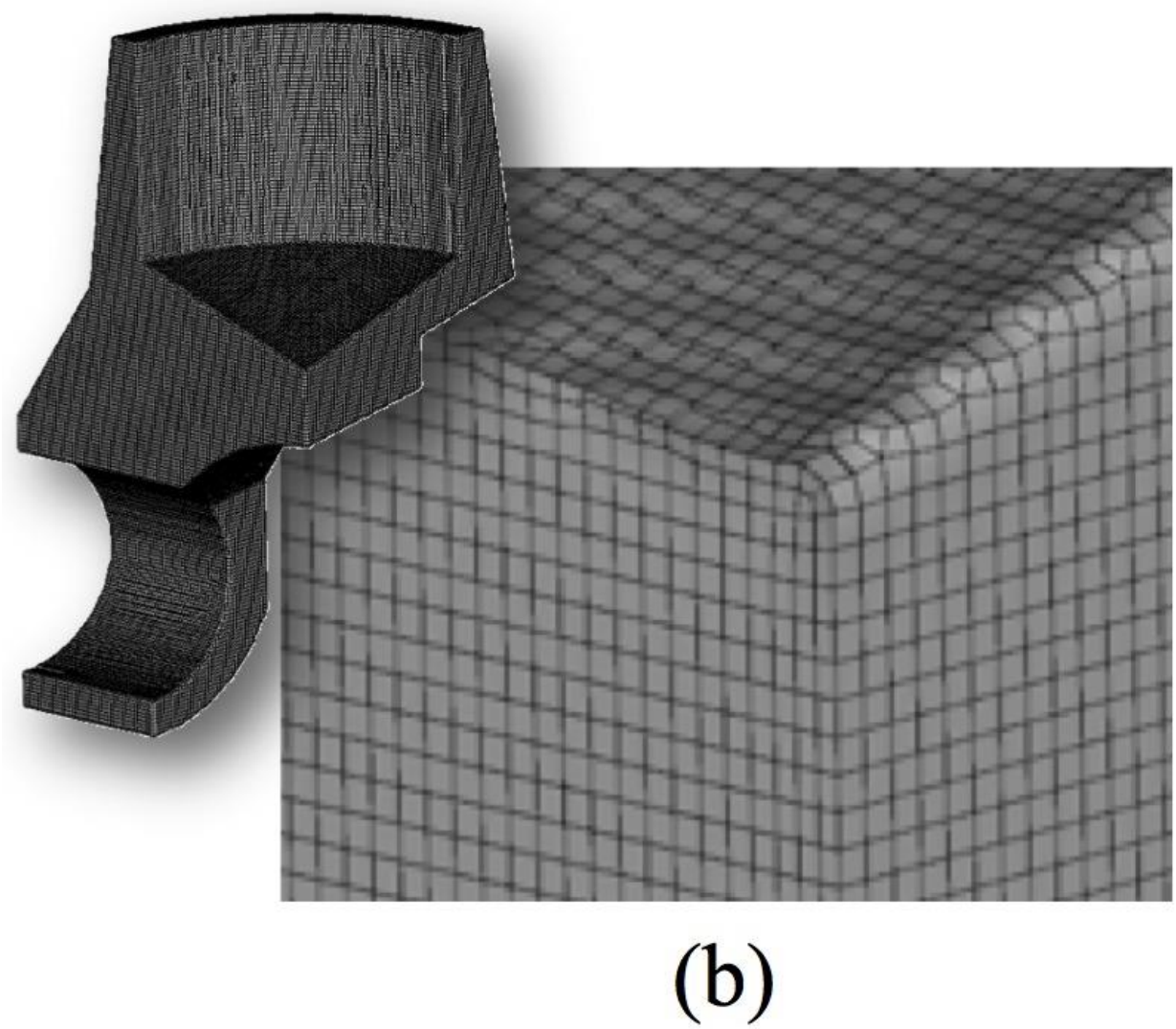



**Figure 3. Examples of the same model meshed at two different resolutions showing a cutaway view of the mesh.**

Another aspect of model preparation for computational simulation involves geometry cleanup and simplification. In most cases, geometry-first methods, such as Sweeping, require an accurate non-manifold boundary representation before mesh generation can begin. Small, sometimes unseen gaps, overlaps and misalignments can result in sliver elements or mesh failure. Tedious manual geometry simplification and manipulation is often required before meshing can commence. Sculpt, however employs a solution that avoids much of the geometry inaccuracy issues inherent in CAD design models. Using a faceted representation of the solid model, a voxel-based volume fraction representation is generated. Figure 4 illustrates the procedure where a CAD model serving as input (figure 4(a)) is processed by a procedure that will generate volume fraction scalar data for each cell of an overlay Cartesian grid (figure 4(b)). One value per material per cell is computed that represents the volume fraction of material filling the cell. A secondary geometry representation is then extracted using an interface tracking technique from which the final hex mesh is generated (figure 4(c)). While similar to its initial facet-based representation, the new secondary geometry description developed from the volume

fraction data results in a simplified model that tends to wash over small features and inaccuracies that are smaller than the resolution of the base cell size.

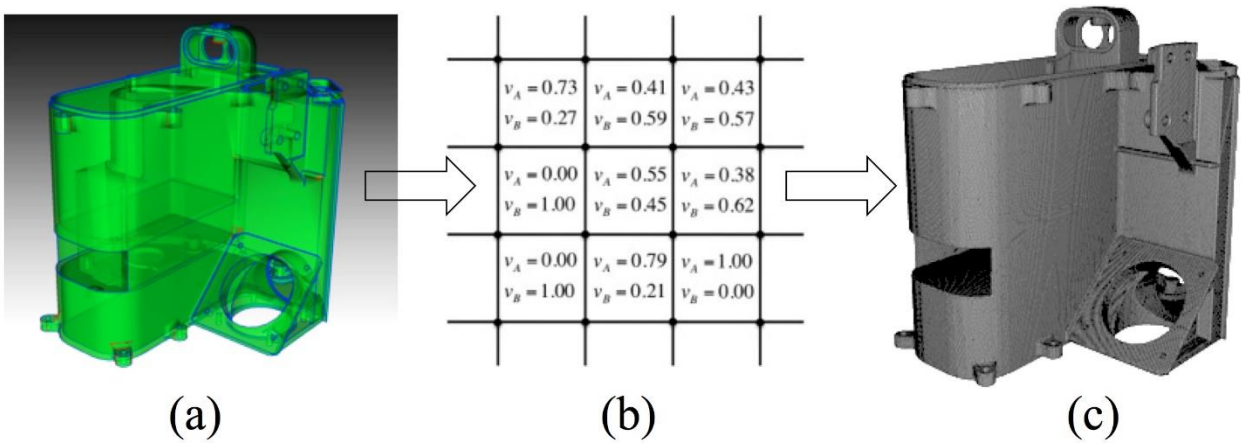



**Figure 4. A representation of the procedure used to generate a hex mesh with Sculpt using Volume Fractions.**

While acknowledging some loss in model fidelity in this new volume-fraction based geometric model, the advantage and time-savings to the analyst of being able to ignore troublesome geometry issues is enormous. At the same time it may be important to understand what the additional discrete approximations will make to solution accuracy and employ relevant engineering judgement in the use of this technology.

**References**

The following technical papers, written by the author of Sculpt, describe the Sculpt procedure in more depth. These papers were presented at the International Meshing Roundtable and are external links to pdf documents.

**Parallel Hex Meshing from Volume Fractions:** Describes the basic algorithms and mathematics used in the Sculpt procedure.

**Parallel Smoothing for Grid-Based Methods:** A brief description of the smoothing procedures used in Sculpt.

**Validation of Grid-Based Hex Meshes with Computational Solid Mechanics:** Describes a study where computational results from Sculpt meshes are compared with Sweep meshes using the Sierra Solid Mechanics Tool as a comparison.

**A Template-Based Approach for Parallel Hexahedral Two-Refinement:** Describes the refinement procedures used for generating adapted Sculpt meshes.

# *Sculpt Application*

This page describes the Sculpt application, a separate companion application to Cubit designed to run in parallel for generating all-hex meshes of complex geometry. Sculpt was developed as a separate application so that it can be run independently from Cubit on high performance computing platforms. It was also designed as a separable software library so it can be easily integrated as an in-situ meshing solution within other codes. As installed with Cubit, Sculpt can be set up and run directly from Cubit, in a batch process

from the unix command line or from a user-defined input file. This documentation describes the input file and command line syntax for the Sculpt Application when running in batch mode. See this page for using Cubit to set up input for Sculpt. A brief technical description of Sculpt may also be found here.


- Sculpt System Requirements
- Running Sculpt
- Sculpt Command Summary
  - Process Control
  - Input Data Files
  - Output
  - Overlay Grid Specification
  - Mesh Type
  - Boundary Conditions
  - Adaptive Meshing
  - Smoothing
  - Mesh Improvement
  - Mesh Transformation
  - Boundary Layers
- Sculpt Examples


#### Sculpt System Requirements

Sculpt is currently built for windows, linux and mac operating systems. Current supported OS versions should be the same as those supported by Cubit. It is designed to take advantage of 64 bit multicore and distributed memory computers, using open-mpi as the basis for parallel communications.

#### Running Sculpt

Sculpt can be run using one of two excutables:

1. **psculpt**
   1. requires the use of **mpiexec** to start the process. Number of processors to use is specified by the -np argument to mpiexec. psculpt and its input parameters are also used as input to mpiexec. For example:

      ```
      mpiexec -np 8 psculpt -stl myfile.stl -cs 0.5
      ```

   2. If appropriate system paths have not been set, you may need to use full paths when referring to mpiexec and psculpt.
2. **sculpt**
   1. This application assumes that mpiexec is included in the standard CUBIT installation directory. The number of processors to use is specified by the -j option. For example:

      ```
      sculpt -j 8 -stl myfile.stl -cs 0.5
      ```

2. If the -j option is not used, sculpt will default to a single processor for execution. The -mpi option can also be used with the sculpt application to indicate a specific mpi installation that is not included with CUBIT. For example:

```
sculpt -j 8 -mpi /path/to/mpiexec -stl myfile.stl -cs 0.5
```

3. If the path specified by the -mpi option does not exist or the mpi version is incompatible, sculpt will attempt to use the local CUBIT-installed mpiexec or else the system mpiexec in the PATH environment.

#### Sculpt Examples

The following illustrate simple use cases of the Sculpt application. To use these examples, copy the following stl and diatom files to your working directory

1.

```
brick1.stl
brick2.stl
bricks.diatom
```

##### Example 1

```
sculpt -j 4 -stl brick1.stl -cs 0.5
```

Runs sculpt with 4 processors with geometry input from brick1.stl. Uses a base Cartesian cell size of 0.5. The bounding box and all other parameters will be defaulted. The result should be the 4 exodus files:

```
brick1.stl_results.e.4.0
brick1.stl_results.e.4.1
brick1.stl_results.e.4.2
brick1.stl_results.e.4.3
```

These files can be combined into a single file using the SEACAS tool **epu**

```
epu -p 4 brick1.stl_results
```

The result of this operation should be a single file:

```
brick1.stl_results.e
```

To view the resulting mesh in Cubit, use the import free mesh command. For example:

1. **import mesh "brick1.stl_results.e" no_geom**

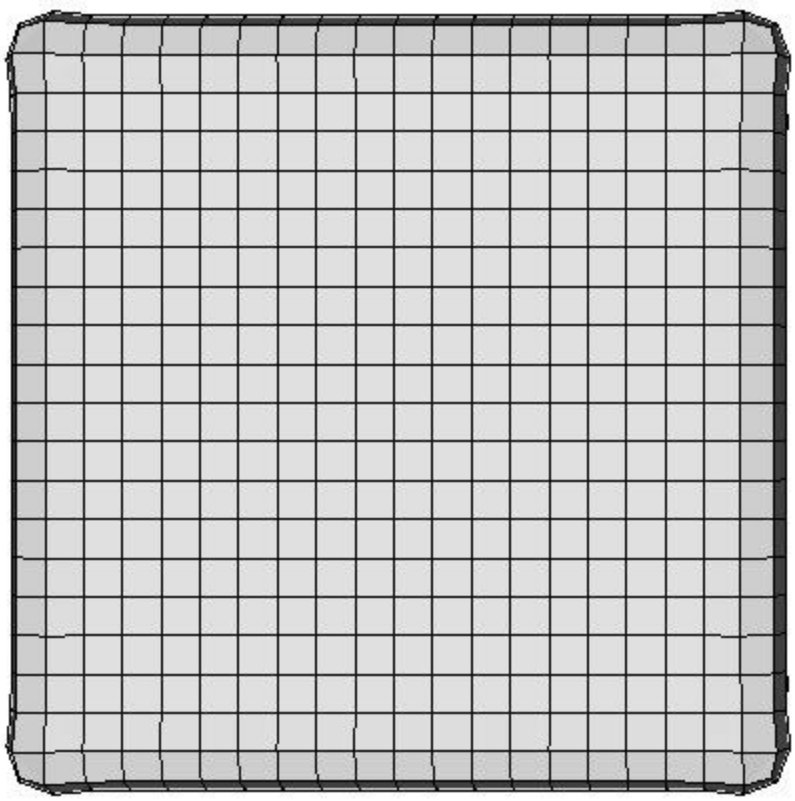


**Figure 1. Example 1 mesh**

#### Example 2

```
mpiexec -np 4 psculpt -x 46 -y 26 -z 26 -t -6.5 -u -6.5 -v -6.5 -q 16.5 -r
6.5 -s 6.50 -d bricks.diatom
```

In this case we use **mpiexec** to start 4 processes of **psculpt**. We explicitly define the number of Cartesian intervals and the dimensions of the grid. Rather than using the -stl option, we use the -d option which allows us to specify the diatom file, bricks.diatom. This file allows us to specify multiple stl files, where each one represents a different material. In this case we use both brick1.stl and brick2.stl, which are called out in bricks.diatom.

We can use similar commands as used in Example 1 to combine and import the free mesh into Cubit for display.

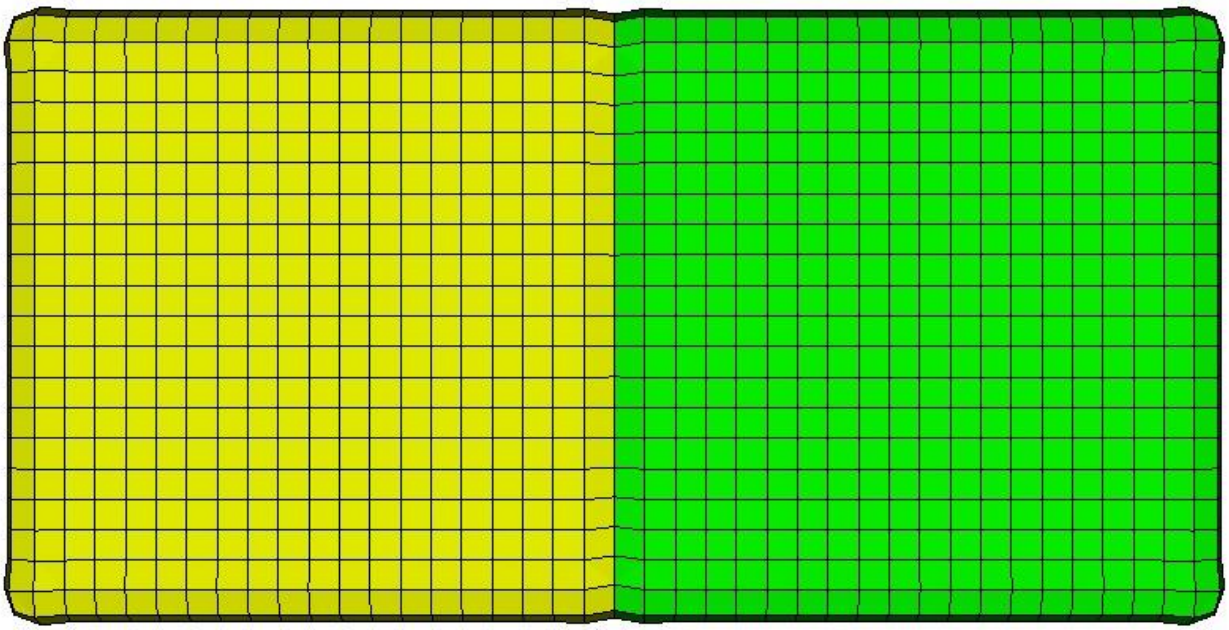


**Figure 2. Example 2 mesh**

# *Parallel Meshing*

Cubit has been designed as a serial application, using a single CPU to generate its meshes. In some cases, where memory or time constraints are critical, parallel meshing may be necessary. Cubit currently provides a few separate applications designed to run in parallel either on a desktop or on massively parallel cluster machines. In these cases, Cubit can be used as a pre-processor to manipulate geometry and set up for meshing, however the actual meshing procedure is performed as a separate process or on another machine. The following two parallel meshing applications are available:

- pCamal
- Sculpt

A separate application for parallel refinement is also available:

- STK_Adapt

# *Sculpt Mesh Transformation*

Sculpt options for applying transformations to the mesh following mesh generation. For cases where the initial geometry description may not be at the desired scale or bounds, the transformation options provide the ability to apply transformations to the node

locations following the mesh generation procedure. This can be effective for microstructure models, where the size and location may be defined by the given intervals of the data.

```
Mesh Transformation           -tfm     --transform
  --xtranslate                -xtr  <arg> Translate final mesh coordinates in
X
  --ytranslate                -ytr  <arg> Translate final mesh coordinates in
Y
  --ztranslate                -ztr  <arg> Translate final mesh coordinates in
Z
  --xscale                    -xsc  <arg> Scale final mesh coordinates in X
  --yscale                    -ysc  <arg> Scale final mesh coordinates in Y
  --zscale                    -zsc  <arg> Scale final mesh coordinates in Z
```

Sculpt Command Summary

---

#### Translate Mesh Coordinates in X

```
Command: xtranslate     Translate final mesh coordinates in X

Input file command:   xtranslate <arg>
Command line options: -xtr <arg>
Argument Type:        floating point value
```

Command Description:

Translate all mesh coordinates written to Exodus file by X delta distance.

---

#### Translate Mesh Coordinates in Y

```
Command: ytranslate     Translate final mesh coordinates in Y

Input file command:   ytranslate <arg>
Command line options: -ytr <arg>
Argument Type:        floating point value
```

Command Description:

Translate all mesh coordinates written to Exodus file by Y delta distance.

---

#### Translate Mesh Coordinates in Z

```
Command: ztranslate     Translate final mesh coordinates in Z

Input file command:   ztranslate <arg>
Command line options: -ztr <arg>
Argument Type:        floating point value
```

Command Description:

Translate all mesh coordinates written to Exodus file by Z delta distance.

---

#### Scale X Mesh Coordinates

```
Command: xscale     Scale final mesh coordinates in X

Input file command:   xscale <arg>
Command line options: -xsc <arg>
Argument Type:        floating point value
```

Command Description:

Scale all mesh X coordinates written to Exodus file by given factor

---

#### Scale Y Mesh Coordinates

```
Command: yscale     Scale final mesh coordinates in Y

Input file command:   yscale <arg>
Command line options: -ysc <arg>
Argument Type:        floating point value
```

Command Description:

Scale all mesh Y coordinates written to Exodus file by given factor

---

#### Scale Z Mesh Coordinates

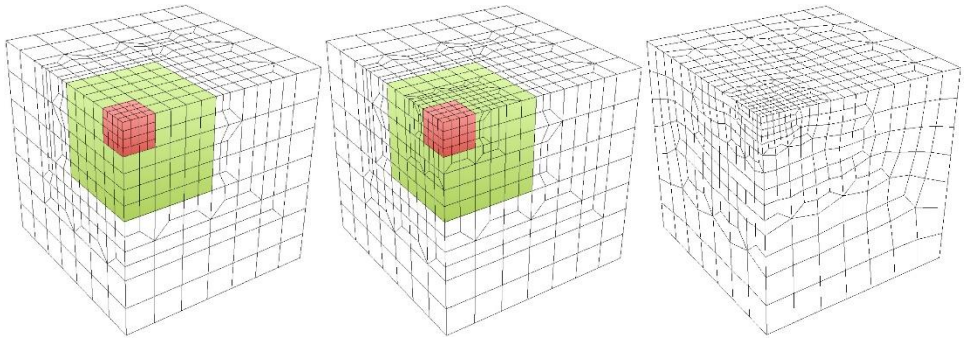
```
Command: zscale     Scale final mesh coordinates in Z

Input file command:   zscale <arg>
Command line options: -zsc <arg>
Argument Type:        floating point value
```

Command Description:

Scale all mesh Z coordinates written to Exodus file by given factor

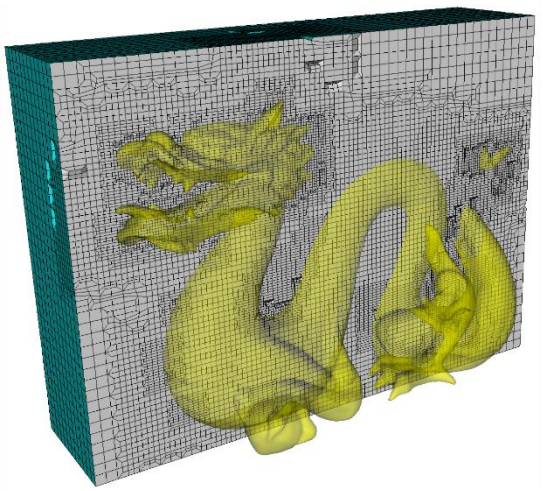
# *Sculpt Adaptive Meshing*

Sculpt options for specifying adaptive meshing. Sculpt uses an initial overlay Cartesian grid that serves as the basis for the all-hex mesh. The default mesh size will roughly follow the constant size cells of the overlay grid. The adaptivity option allows the user to automatically split cells of the Cartesian grid based on geometric criteria, resulting in smaller cells in regions with finer details. The adapted grid is then used as the basis for the Sculpt procedure.

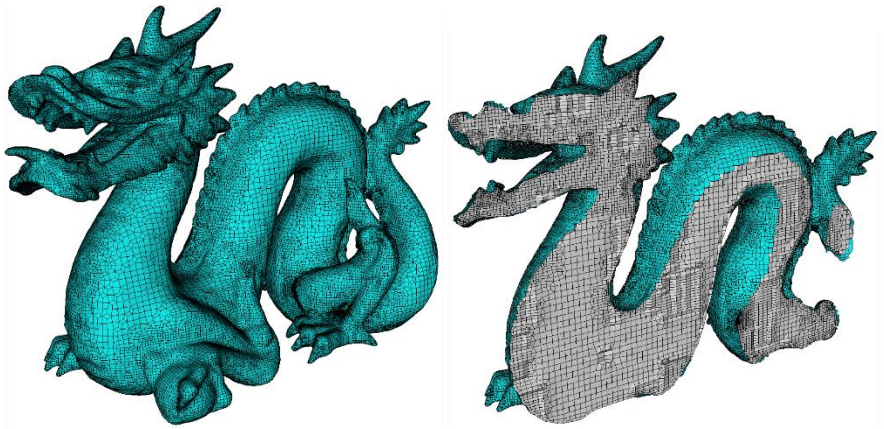


Adaptive mesh begins with constant size coarse Cartesian grid. Cells are recursively split based on geometry criteria and transitions added between levels. Projections and smoothing are performed to improve element quality.

Three options are used for controlling the adaptivity in sculpt: adapt_type, adapt_levels and adapt_threshold. The adapt_type option controls the method and geometric criteria used for deciding which cells to split in the grid, while the adapt_levels option controls the the maximum number of times any one cell can be split. Depending upon the adapt_type selected, the adapt_threshold is used as the specific geometric threshold value at which the decision is made to split any given cell.



Initial cut-away view of adapted grid from dragon model before performing Sculpt operations.



The final mesh of the dragon model and cutaway view of the mesh is shown with up to 4 levels of adaptive refinement.

```
Adaptive Meshing                -adp     --adapt
  --adapt_type                  -A     <arg> Adaptive meshing type
  --adapt_threshold             -AT    <arg> Threshold for adaptive meshing
  --adapt_levels                -AL    <arg> Number of levels of adaptive
refinement
  --adapt_material              -AM    <arg> Info for adapting material
  --adapt_export                -AE          Export exodus mesh of refined grid
  --adapt_non_manifold          -ANM         Refine at non-manifold conditions
  --adapt_load_balance          -ALB         Adaptive parallel load balancing
  --adapt_memory_stats          -AMS         Write memory usage stats for
adaptivity
```

Sculpt Command Summary

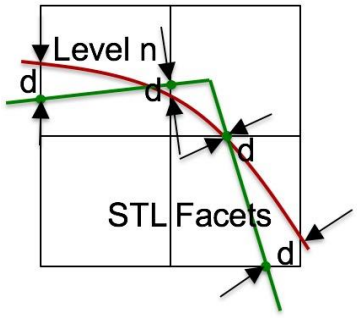
#### Adaptive Refinement Type

```
Command: adapt_type     Adaptive meshing type

Input file command:   adapt_type <arg>
Command line options: -A <arg>
Argument Type:        integer (0, 1, 2,...7)
Input arguments: off (0)
                 facet_to_surface (1)
                 surface_to_facet (2)
                 surface_to_surface (3)
                 vfrac_average (4)
                 coarsen (5)
                 vfrac_diff (6)
                 vfrac_difference (6)
                 resample (7)
                 material (8)
                 region (12)
```

Command Description:

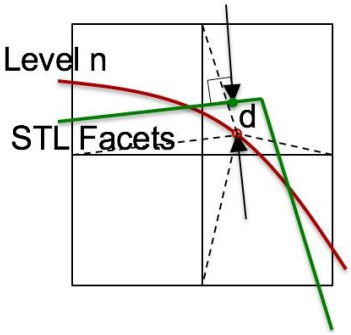
This option will automatically refine the mesh according to a user-defined criteria. Without this option, a constant cell size will be assumed everywhere in the model. To build the mesh, Sculpt uses an approximation to the exact geometry of the CAD model by interpolating mesh surfaces from volume fraction samples in each cell of the Cartesian grid. In general, the, higher the resolution of the Cartesian grid, the more sampling is done and the more accurate the mesh will represent the initial geometry. The adapt_type selected will control the criteria used for refining the mesh. If the criteria is not satisfied, the refinement will continue until a threshold indicated by the adapt_threshold parameter is satisfied everywhere, or the maximum number of levels (adapt_levels) is reached. The following criteria for refinement are available:

- **off (0):** Cartesian grid is defined only by **nelx**, **nely**, and **nelz** or **cell_size** which is used as the basis for the sculpt mesh. No refinement will be performed.
- **facet_to_surface (1):** This option will evaluate every location where an edge in the Cartesian grid intersects a triangle of the STL model and measures the closest distance to the approximated geometry. The cells adjacent to intersecting edges where the measured distance is greater than the **adapt_threshold** will be identified for uniform refinement. This is done for each refinement level where a new approximated geometry is then computed based upon the finer resolution grid. The refinement will continue until all measured distances are less than the adapt_threshold, or the maximum number of levels (**adapt_levels**) is reached. This option can only be used if input comes from an STL file. Microstructures and diatoms are currently not supported.



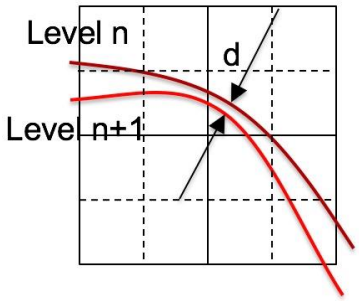

Distance from STL Facet to Approximated Geometry. The distance d is measured between the facets (green) where they cross the edges of the grid, to the closest point on the interpolated geometry. If d > adapt_threshold then the cell is split.

- **surface_to_facet (2):** This criteria is similar to **facet_to_surface** (1) except that the locations selected for sampling are chosen from the vertices representing the approximated surfaces. The closest distance measured to any of the facets in the STL model is used as the criteria for refinement. Those cells at vertices where the distance measured exceeds the **adapt_threshold** are identified for refinement. This option is generally faster than 1, but may miss features if the initial resolution of the grid is too coarse. This option can also only be used if input geometry comes from an STL file. Microstructures and diatoms are currently not supported.



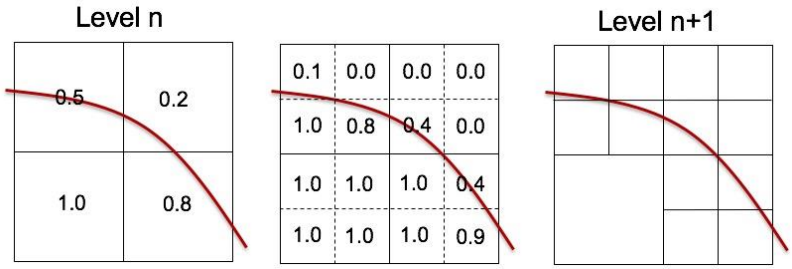

Distance from Approximated Geometry to STL Facet. The distance d is measured between points on the interpolated geometry corresponding to a projected point on the grid, to its closest point on one of the STL facets. If d > adapt_threshold then the cell is split.

- **surface_to_surface (3):** This criteria will test each cell to compute the local interpolated surface for the cell and compare with the surface interpolated for its eight subdivided child cells. If the distance between these two approximated surfaces is greater the the user defined **adapt_threshold**, then the cell will be uniformly refined. This option can be used with STL and diatom input geometry, but not with Microstructures.




Distance Between Child and Parent Approximated Geometry. After computing the interpolated geometry for level n and level n+1, d is the distance between the two geometry representations. Cells where d > adapt_threshold are split.

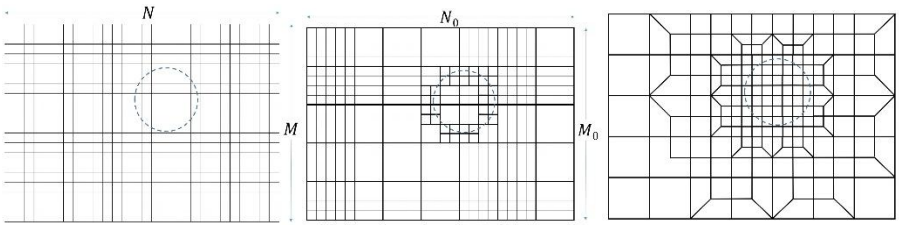
- **vfrac_average (4):** Each cell of the Cartesian grid is tested to determine if it should be subdivided into eight cells. The volume fraction of the parent cell is compared with the average volume fraction of its eight child cells. If the absolute difference between the average child volume fraction and its parent volume fraction is greater than the user defined **adapt_threshold** then the cell is uniformly refined. The **adapt_threshold** for this case should be a number between 0 and 1. A smaller number will be more sensitive to changes in geometry, usually resulting in more refinement at interfaces.




Difference of Cell Volume Fraction. Volume fractions are evaluated for the 8 child cells of a cell at level n. This example shows where one or more of the volume fractions at level n+1 of the lower left cells does not differ by more than a threshold d, so it remains unsplit.

- **coarsen (5):** Given a dense set of data on a Cartesian Grid, Sculpt will begin at a coarse resolution and refine to capture changes in the data. It uses the adapt_levels option to determine the coarseness of the initial grid. For example, a dense grid of LxMxN cells will begin with an initial resolution of L/2^a x M/2^a x N/2^a, where a is the user defined adapt_levels value. Cells will be identified for refinement if the volume fraction of any material in a cell is greater than the user defined **adapt_threshold** and less than 1.0-adapt_threshold. This option is available only for **input_spn** and **input_micro** formats. It is most useful for cases where very dense data is initially provided which would be too fine to serve as an

FEA mesh. This method will effectively coarsen the mesh on the interior and exterior of solids, but maintain a fine resolution at geometry boundaries.




Refine to Dense Data (Coarsening). Initial grid at resolution N X M is coarsened to N0 X M0 based on the adapt_levels value. Coarse cells are then split similar to criteria in adapt_type = 4.

- **vfrac_difference (6):** Each cell of the Cartesian grid is tested to determine if it should be subdivided into eight cells. The volume fraction of the parent cell is compared with each of the volume fractions of its eight child cells. If the absolute difference between any of the child volume fractions and its parent volume fraction is greater than the user defined **adapt_threshold** then the cell is uniformly refined. The **adapt_threshold** for this case should be a number between 0 and 1. A smaller number will be more sensitive to changes in geometry, usually resulting in more refinement at interfaces.
- **resample (7):** Input volume fraction data from microstructure file formats (**input_micro, input_cart_exo and input_spn**) is down sampled, averaging volume fraction data across multiple cells according to the **adapt_levels** set. To resample at 1/2 resolution, use **adapt_levels = 1**, For 1/4 resolution, use **adapt_levels = 2**, etc. For example, if the input data has a resolution of 100X100X100, **adapt_levels = 1** will result in effectively a 50X50X50 grid resolution where each 2X2X2 group of cells in the original data is averaged into a single larger cell. Similarly, adapt_type = 2 will result in a 25X25X25 grid resolution, where each 4X4X4 group of cells makes up a single cell in the resample. Default **adapt_levels** is 2.
- **material (8):** Refinement will be done in cells where the predominant volume fraction is a user defined material ID. Must be used with the **adapt_material** option to indicate a material to be refined.

To maintain a conforming mesh, transition elements will be inserted to transition between smaller and larger element sizes. Default for the **adapt_type** option is **off (0)** (or that no adaptive refinement will take place).
In all cases the initial Cartesian grid defined by xint, yint and zint or the **cell_size** value will be used as the basis for refinement and will define the approximate largest element size in the mesh.

---

**Adaptive Refinement Threshold**

```
Command: adapt_threshold     Threshold for adaptive meshing

Input file command:   adapt_threshold <arg>
Command line options: -AT <arg>
Argument Type:        floating point value >= 0.0
```

Command Description:

This value controls the sensitivity of of the adaptivity. The value used should be based upon the adapt_type:

- **facet_to_surface (1)**
  **surface_to_facet (2)**
  **surface_to_surface (3)**
  For these options, the **adapt_type** selected represents an absolute distance between surfaces or facets. Where the distance exceeds **adapt_threshold** the nearby cell or cells will be identified for refinement. The smaller this number the more sensitive will be the adaptation and greater the resulting number of elements. If not specified, the **adapt_threshold** will be determined as follows:

  ```
  adapt_threshold = 0.25 * cell_size / adapt_levels^2
  ```

- **vfrac_average (4)**
  **coarsen (5)**
  The **adapt_threshold** value in this case represents the maximum difference in volume fraction between a parent cell and the average of its eight child cells. This value should be between 0.0 and 1.0. The smaller the number, the more sensitive will be the adaptation and the greater the number of resulting elements. A default **adapt_threshold** of 0.01 is used if not specified.

- **material (8)**

  Refinement occurs if volume fraction of material in a cell exceeds the threshold value. A separate **threshold** can also be set for each **adapt_material** overriding the the global value for threshold.

Note that the user defined adapt_threshold may not be satisfied everywhere in the mesh if the value defined for adapt_levels is exceeded.

---

#### Number of Adaptive Levels

```
Command: adapt_levels     Number of levels of adaptive refinement

Input file command:   adapt_levels <arg>
Command line options: -AL <arg>
Argument Type:        integer >= 0
```

Command Description:

The maximum number of levels of adaptive refinement to be performed. One level of refinement will split each Cartesian grid cell identified for uniform refinement into eight child cells. Two levels of refinement will split each cell again into eight, resulting in sixty-four child cells, three levels into 256, and so on. The maximum number of subdivision per cell is give as:

```
number of cells = 8^adapt_levels
```

The minimum edge length for any cell will be given by:

```
min cell edge length = cell_size / adapt_levels^2
```

The actual number of refinement levels used will be determined by whether all cells meet the **adapt_threshold**, or the adapt_levels value is exceeded. The default adapt_levels is 2. Note that setting the **adapt_levels** more than 4 or 5 can result in long compute times.

---

**Info for Adapt Material**

```
Command: adapt_material     Info for adapting material

Input file command:   adapt_material <arg>
Command line options: -AM <arg>
Argument Type:        5 optional values
```

Command Description:

Specifies material information to be used with the **adapt_type = material** option. This option expects up to 5 ordered values representing the following:

1. **material id**: ID of material to be refined
2. **levels**: (optional) Number of levels to uniformly refine within the specified material. If not specified or **levels** <= 0, the global value for **adapt_levels** will be used.
3. **threshold**: (optional) Threshold used to determine whether to uniformly refine. Refinement occurs if volume fraction of material in a cell exceeds the threshold value. If not specified or **threshold** <= 0, the global value for **adapt_threshold** will be used.
4. **expansion distance**: (optional) Absolute distance beyond the specified material that will be uniformly refined. If not specified or **expansion distance** = 0, uniform refinement will be restricted to the material. A negative integer value for **expansion distance** will be interpreted as the number of layers of cells beyond the material that will be refined.
5. **region**: (optional) specify a region id as defined by an **adapt_region** specification. Refinement will be limited to the adapt_region overlapping the specified region. If not specified or **region** <= 0, the full material domain will be refined.

This option may be used multiple times in the same input, once per material to be adapted.

---

#### Export Refined Cartesian Grid

```
Command: adapt_export     Export exodus mesh of refined grid

Input file command:   adapt_export
Command line options: -AE
```

Command Description:

Export an exodus mesh containing the refined Cartesian grid. Interface reconstruction, boundary layer insertion and smoothing have not yet been applied to this mesh. It is the base mesh used as input to Sculpt. One file per processor will be exported in the form "vfrac_adapt.e.x.x". The exodus mesh produced will also contain the computed volume fractions for each material present in the model represented as element variables.
This option is primarily used for debugging the refinement option. However the mesh produced with this option can be used as the base mesh when used with the **input_mesh** option. For example, instead of Cartesian grid options, the input mesh may be specified as **input_mesh = vfrac_adapt.e.1.0**. Sculpt will use the refined mesh and the volume fraction element variables to build the final mesh.

---

#### Adapt Cells at Non-manifold Nodes

```
Command: adapt_non_manifold     Refine at non-manifold conditions

Input file command:   adapt_non_manifold
Command line options: -ANM
```

Command Description:

If refinement results in a non-manifold condition at a node, the surrounding cells will be identified for refinement. In some cases, using this option will result in a closer match to geometry for thin layers or small features. Using this option will normally result in more elements at material interfaces. Note that in all cases non-manifold conditions will be resolved even without this option in a subsequent step, however without this option, the resulting solution may not match geometry as accurately.
The **adapt_non_manifold** option is off by default. It is currently only implemented for adapt_type that use an STL geometry definition. (**adapt_type = 1,2,3**)

---

#### Adaptive Parallel Load Balancing

```
Command: adapt_load_balance     Adaptive parallel load balancing

Input file command:   adapt_load_balance
Command line options: -ALB
```

Command Description:

Do adaptive load balancing during refinement. Uses Zoltan's Recursive Coordinate Bisection (RCB) algorithm to repartition parallel domains following each adaptive level. Default (OFF), refinement, will use roughly spatially equal volume domains on each processor. This can cause significant imbalance in work between processors, sometimes causing failure, especially where adpativity is localized. When using **adapt_load_balance**, Sculpt will repartition the parallel domains after each adaptivity level so that each processor maintains approximetly equal numbers of elements to equalize work and memory among processors.

---

#### Write Memory Usage Stats for Adaptivity

```
Command: adapt_memory_stats     Write memory usage stats for adaptivity

Input file command:   adapt_memory_stats
Command line options: -AMS
```

Command Description:

When the **adapt_type** option is enabled, it allows users to monitor the memory usage and availability during the adaptivity procedures. By setting this option, both minimum and maximum memory usage for the current processor arrangement will be recorded. This information can be particularly useful when building meshes with many levels of refinement or when there are significant variations in the spatial distribution of refinement regions.

# *Sculpt Boundary Conditions*

Sculpt options for specifying the methods for generating nodesets, sidesets and blocks on the mesh. Several automatic methods for generating nodesets and sidesets are provided in Sculpt using the gen_sidesets option. Where multiple blocks are required, Block IDs are normally defined using the material ID in the diatom file. Each STL file can be associated with a different block ID. If the mesh_void option is used, the ID for the block of elements in the void region can be set using the void_mat option.

For other input formats such as volume fraction microstructure data or Cartesian Exodus files, the Block IDs are defined by the individual formats.

```
Boundary Conditions              -bc     --boundary_condition
  --void_mat                     -VM   <arg> Void material ID (when
mesh_void=true)
  --separate_void_blocks         -SVB        Separate void into unique block IDs
  --material_name                -mn   <arg> Label Material (Block) with Name
  --sideset_name                 -sn   <arg> Label Sideset with Name
  --nodeset_name                 -nn   <arg> Label Nodeset with Name
  --sideset                      -sid  <arg> User Defined Sideset
  --nodeset                      -nid  <arg> User Defined Nodeset
  --gen_sidesets                 -SS   <arg> Generate sidesets
  --free_surface_sideset         -FS   <arg> Free Surface Sideset
  --match_sidesets               -mss  <arg> Sidesets ids of matching pairs
  --match_sidesets_nodeset       -msn  <arg> Nodeset defining match_sidesets
```

Sculpt Command Summary

---

#### Void Material ID

```
Command: void_mat     Void material ID (when mesh_void=true)

Input file command:   void_mat <arg>
Command line options: -VM <arg>
Argument Type:        integer > 0
```

Command Description:

When the **mesh_void** option is used, this value is the material (block) ID assigned to all elements in the void region. If void_mat option is not used, the material ID of elements in the void region will be the maximum material ID in the model + 1. Note that the void_mat may be the same as an existing material in another part of the model.

---

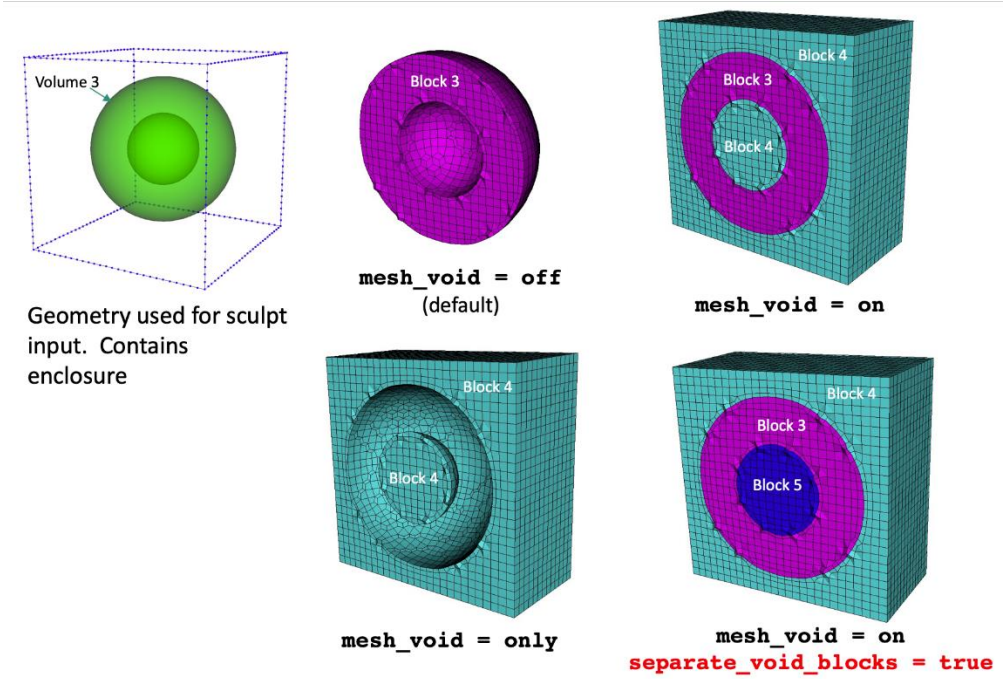
#### Separate Void Blocks

```
Command: separate_void_blocks     Separate void into unique block IDs

Input file command:   separate_void_blocks
Command line options: -SVB
```

Command Description:

When the **mesh_void** option is used, the default case will generate a single block ID for all elements in the void region. Turning this option **ON** will separate the elements into

unique block IDs where the elements are contiguous. For example, elements in a void region completely enclosed and interior to a volume will be assigned a different block ID to those exterior to the volume. The **void_mat** option can be used to specify the ID of the first void block. Subsequent blocks will be numbered incrementally from the first void mat ID. If **void_mat** is not defined, the next highest IDs not used by a material will be used for the void block IDs.




Example of the effect of using **separate_void_blocks** with the **mesh_void** option

---

**Material Name**

```
Command: material_name     Label Material (Block) with Name

Input file command:   material_name <arg>
Command line options: -mn <arg>
Argument Type:        integer and string
```

Command Description:

Optionally assign a name to a material. Specify a valid material ID followed by a name. This option can be used multiple times to label one or more materials. The material name will be assigned as the block name to be written to the output exodus file. A valid integer representing a material defined in the input should be used. For example, the name will be associated with the material ID identified in the diatom file for a given package. The command will be ignored if a valid matching material is not found.

Note: When using the **input_mesh** option and **input_mesh_material = blocks**, block names may be included in the input exodus mesh as part of the exodus block description. If the **material_name** option is used and exodus blocks names are included in the input file, the specified material name in the sculpt input will be used.

---

#### Sideset Name

```
Command: sideset_name     Label Sideset with Name

Input file command:   sideset_name <arg>
Command line options: -sn <arg>
Argument Type:        integer and string
```

Command Description:

Optionally assign a name to a sideset. Specify a valid sideset ID followed by a name. This option can be used multiple times to label one or more sidesets. For example:

```
sideset_name 10 left_faces
sideset_name 20 right_faces
sideset_name 30 bottom_faces
sideset_name 40 top_faces
```

The assigned sideset name will be written to the output exodus file. A valid integer representing a sideset to be generated should be used. The sideset ID should refer to an existing sideset defined using **gen_sidesets** or **sideset** options.
**Note:** When using the **input_mesh** option and **input_mesh_material = blocks**, sideset names may be included in the input exodus mesh as part of the exodus sideset description. If the **sideset_name** option is used and exodus sideset names are included in the input file, the specified sideset name in the sculpt input will be used.

---

#### Nodeset Name

```
Command: nodeset_name     Label Nodeset with Name

Input file command:   nodeset_name <arg>
Command line options: -nn <arg>
Argument Type:        integer and string
```

Command Description:

Optionally assign a name to a nodeset. Specify a valid nodeset ID followed by a name. This option can be used multiple times to label one or more nodesets. For example:

```
nodeset_name 10 left_nodes
```

```
    nodeset_name 20 right_nodes
    nodeset_name 30 bottom_nodes
    nodeset_name 40 top_nodes
```

The assigned nodeset name be will written to the output exodus file. A valid integer representing a nodeset to be generated should be used. The nodeset ID should refer to an existing sideset defined using **gen_sidesets** or **nodeset** options.
**Note:** When using the **input_mesh** option and **input_mesh_material = blocks**, nodeset names may be included in the input exodus mesh as part of the exodus nodeset description. If the **nodeset_name** option is used and exodus nodeset names are included in the input file, the specified nodeset name in the sculpt input will be used.

---

#### User Defined Sideset

```
Command: sideset     User Defined Sideset

Input file command:   sideset <arg>
Command line options: -sid <arg>
Argument Type:        integer and string
```

Command Description:

Define a sideset (collection of mesh faces) based on a user-defined specification. Requires one integer representing a unique sideset ID and a character string representing a geometry specification. Current supported geometry specifications include: **xmin, xmax, ymin, ymax, zmin, zmax**. For a mesh defined with an axis-aligned orientation this option will place sidesets at the bounding box boundaries. For example:

```
    sideset 10 xmin
    sideset 20 xmax
    sideset 30 ymin
    sideset 40 ymax
```

**Note:** See also option **gen_sidesets = RVE**. The gen_sidesets option will automatically define sidesets at boundaries for Cartesian base meshes.

---

#### User Defined Nodeset

```
Command: nodeset     User Defined Nodeset

Input file command:   nodeset <arg>
Command line options: -nid <arg>
Argument Type:        integer and string
```

Command Description:

Define a nodeset (collection of nodes) based on a user-defined specification. Requires one integer representing a unique nodeset ID and a character string representing a geometry specification. Current supported geometry specifications include: **xmin, xmax, ymin, ymax, zmin, zmax**. For a mesh defined with an axis-aligned orientation this option will place nodesets at the bounding box boundaries. For example:

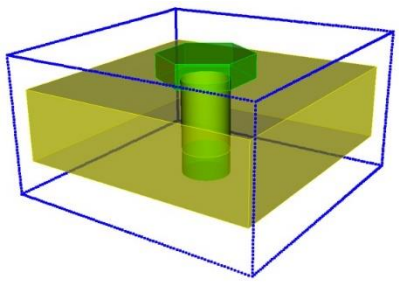
```
nodeset 10 xmin
nodeset 20 xmax
nodeset 30 ymin
nodeset 40 ymax
```

---

#### Generate Sidesets

```
Command: gen_sidesets     Generate sidesets

Input file command:   gen_sidesets <arg>
Command line options: -SS <arg>
Argument Type:        integer (0, 1, 2, 3, 4, 5)
Input arguments: off (0)
                 fixed (1)
                 variable (2)
                 geometric_surfaces (3)
                 geometric_sidesets (4)
                 rve (5)
                 input_mesh_and_stl (6)
                 input_mesh_and_free_surfaces (7)
                 rve_variable (8)
                 input_mesh (9)
```

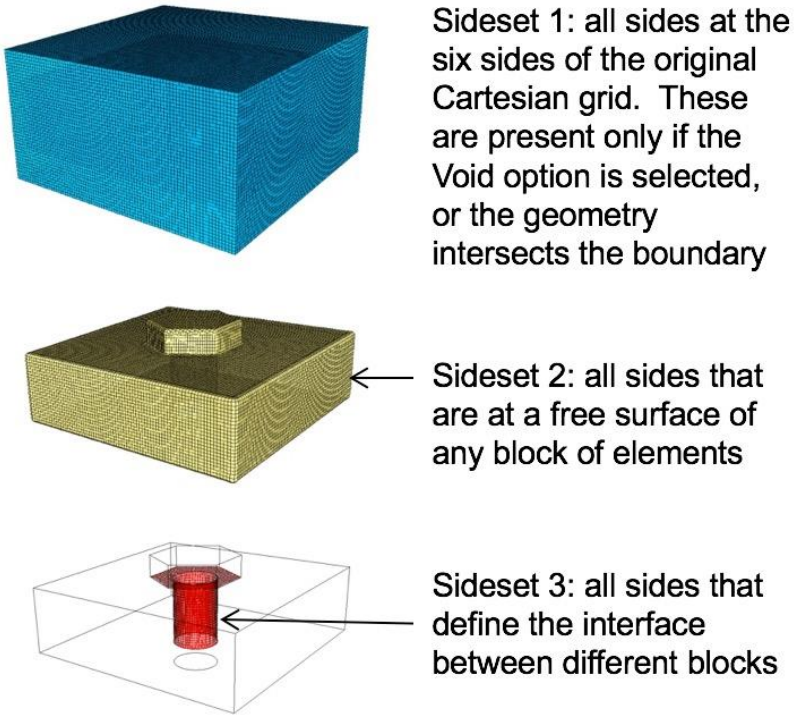
Command Description:



Geometry used in sideset examples below.

Generate exodus sidesets using one of the following options:
**off (0):** No sidesets will be generated
**fixed (1):** Exactly 3 sidesets will be generated according to the following:

- Sideset 1: All sides at the domain boundary. Sides will only be present in this sideset if the model intersects the enclosing bounding box or the void option is used.
- Sideset 2: All sides at the model boundary. Any side on the model that is not interior will be included. This should represent a full enclosure of the model if it does not intersect the domain boundary.
- Sideset 3: All sides at material interfaces. Includes sides on the interior where adjacent blocks are different.

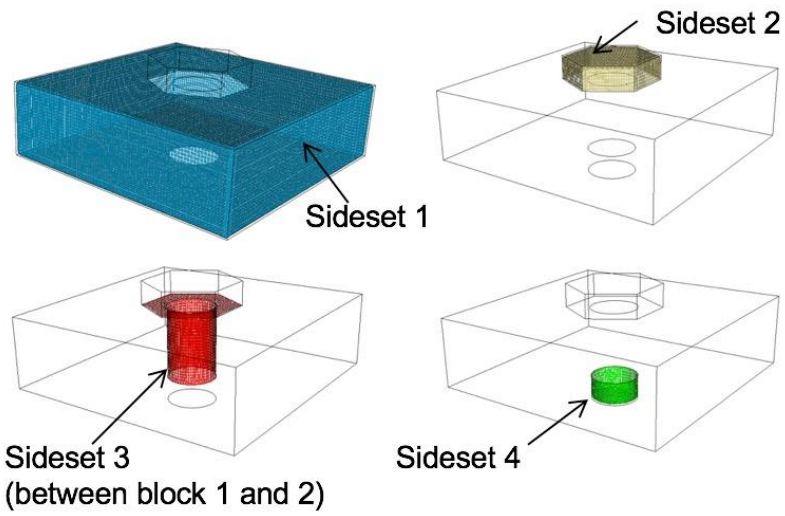



Example of fixed(1) sidesets.

**variable (2):** A variable number of sidesets will be generated with the following characteristics:

- Surfaces at the domain boundary
- Exterior material surfaces
- Interfaces between materials

Unlike Fixed sidesets, grouping of sides will be contiguous. A separate sideset will be generated for each set of contiguous sides.

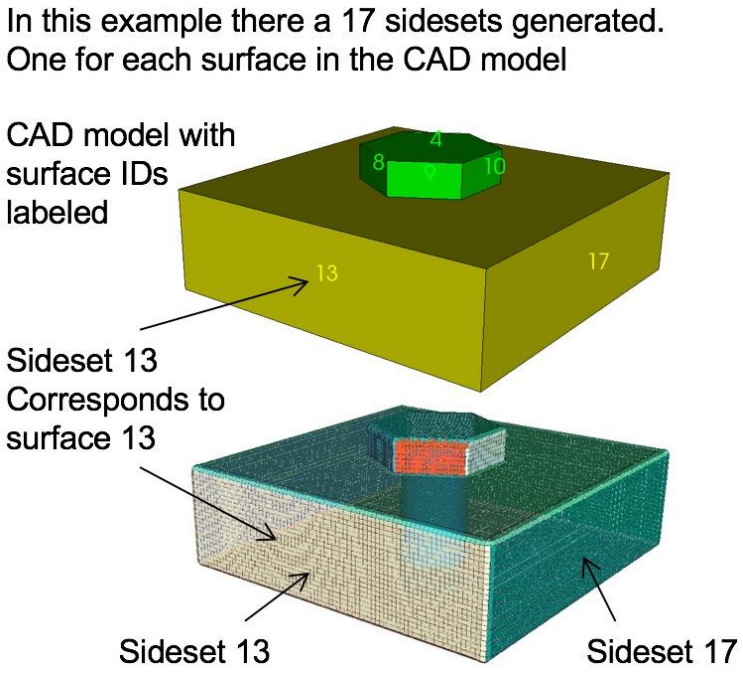



Example of variable(2) sidesets.

**geometric_surfaces (3):** Sidesets will be generated according to imported surface ID information. STL files may include an optional surface designation for any or all triangles in the file. Surface information may be written automatically from Cubit based on geometric surface IDs or sideset IDs. See the cubit sculpt parallel sideset option for more details. Alternatively, use the "export stl ..." command with the "sidesets" option to export all sidesets in a Cubit model as surface information. If present, one sideset will be generated for each surface designation in the STL file. Following is an example surface designation in an STL file. It would appear following all triangles.

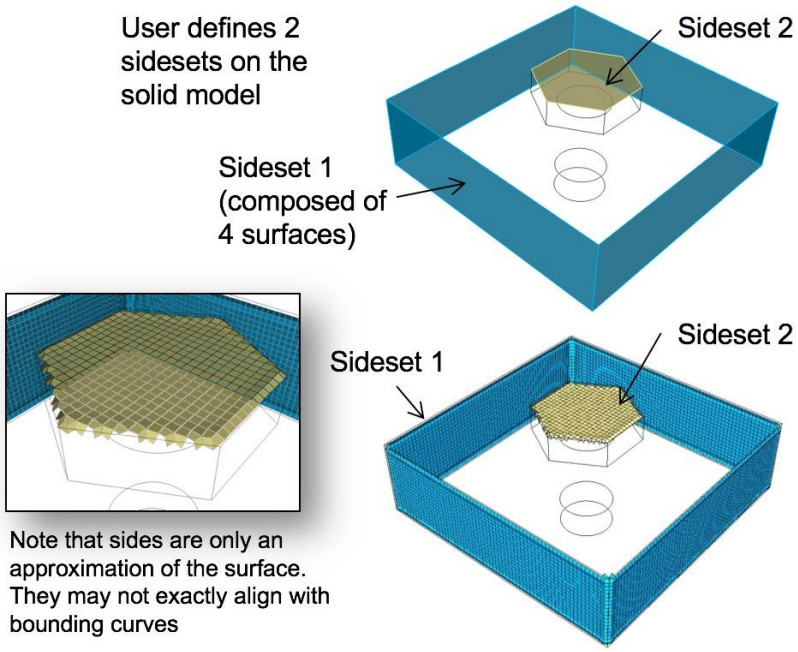
```
surface 1
  0 1 2 3 4 5 6 7 8 9
  10 11 12 13 14 15 16 17 18 19
  20 21 22 23
endsurface 1
```

The id following the surface designation will be used as the sideset ID. Up to 10 triangle IDs, per line may be assigned to the surface. Triangle IDs are assigned based on order they appear in the STL file. Any number of surfaces may be defined. For this option, the assumption is that all triangles included in the STL files will be included in at least one surface designation.




Example of all geometric surfaces (3) defining sidesets.

**geometric_sidesets (4):** Similar to **geometric_surfaces**, except that only a portion of the triangles may be designated as sideset surfaces. This option is useful when using Cubit to identify specific surfaces as sidesets.

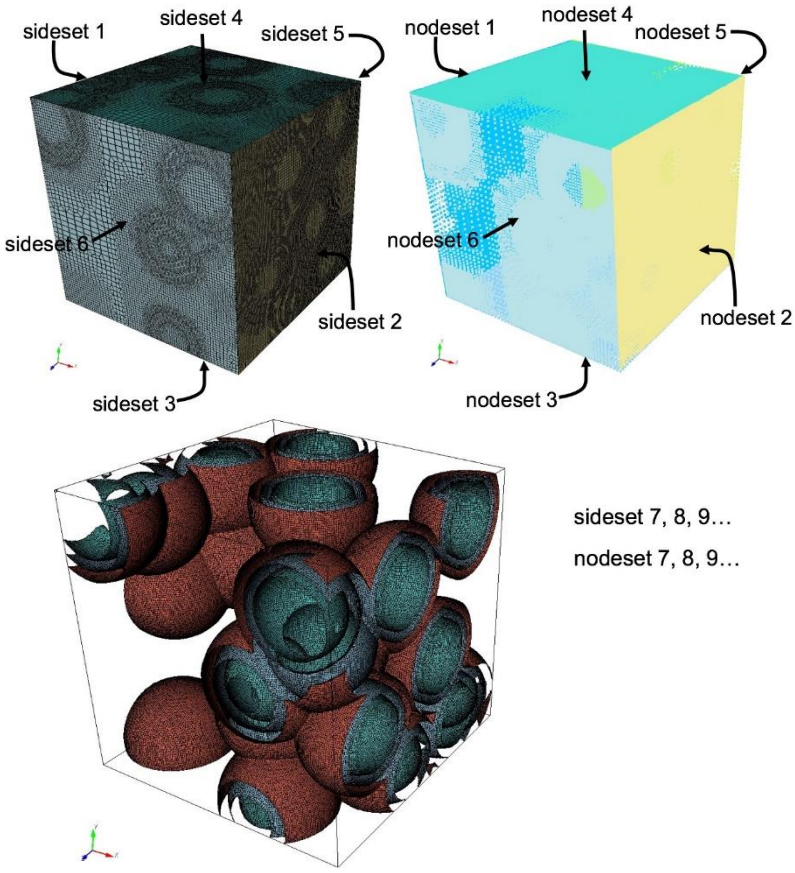



Example of selected geometric sidesets (4) in Cubit defining sidesets in Sculpt.

**RVE (5):** When using the full bounding box, such as representative volume elements (RVE) for microstructures, the nodesets and sidesets with IDs 1 to 6 are reserved for the six faces of the bounding box. They are assigned as follows:

| Nodeset/Sideset ID | Contains nodes/faces |
|---|---|
| 1 | on minimum X domain boundary |
| 2 | on maximum X domain boundary |
| 3 | on minimum Y domain boundary |
| 4 | on maximum Y domain boundary |
| 5 | on minimum Z domain boundary |
| 6 | on maximum Z domain boundary |

In addition, a nodeset and sideset will be generated on interior surfaces for each unique pair of adjacent material IDs. One final nodeset will also be generated along interior curves at all internal triple junctions (curves where at least 3 surfaces share a common curve).

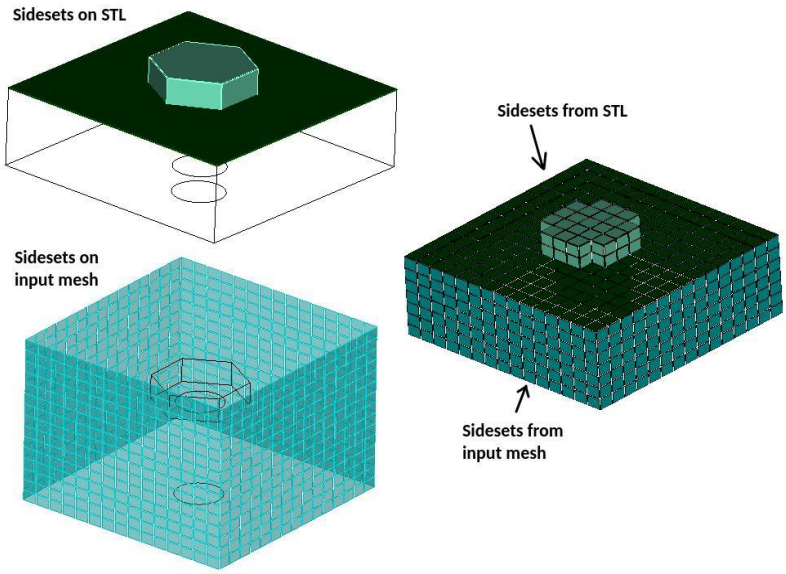



Example of automatically defined sidesets at domain boundaries of an RVE and at all interface surfaces between materials.

**input_mesh (9):** Used with the **input_mesh** option where an exodus file is used as the base grid. Only sidesets and nodesets defined in the input exodus mesh are transferred to the output mesh.

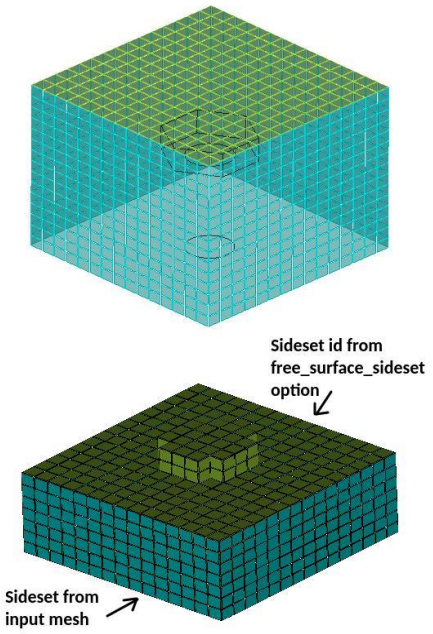
**input_mesh_and_stl (6):** Used with the **input_mesh** option where an exodus file is used as the base grid. Sidesets and nodesets defined in the input exodus mesh are transferred

to the output mesh if the surface is not an interior surface. Sidesets defined in the augmented STL input file are transferred to the output mesh for interior surfaces. See also the **free_surface_sideset** option for prescribing a sideset on interior surfaces cut by the STL definition when using the **input_mesh** option.




Example of sidesets defined in the input mesh and corresponding domain boundary sidesets in the output mesh.

**input_mesh_and_free_surfaces (7):** Used with the **input_mesh** option where an exodus file is used as the base grid. Sidesets and nodesets defined in the input exodus mesh are transferred to the output mesh if the surface is not an interior surface. Sidesets defined in the **free_surface_sideset** option are used to define sidesets for interior surfaces.




Example of sidesets defined in the input mesh and corresponding domain boundary sidesets in the output mesh.

**rve_variable (8):** Nodesets 1-6 and Sidesets 1-6 are defined at the boundaries as described in the **gen_sidesets = rve (5)** option. With the **rve_variable** option, additional nodesets and sidesets at material interfaces on the interior of the mesh are defined similar to the **gen_sidesets = variable (2)** option. Grouping of interior sides in a sidesets will be contiguous, where a separate sideset will be generated for each unique set of contiguous sides. Nodesets will be generated in a similar manner.

---

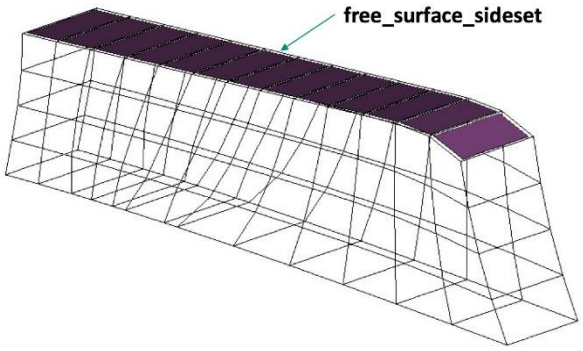
#### Free Surface Sidesets

```
Command: free_surface_sideset     Free Surface Sideset

Input file command:   free_surface_sideset <arg>
Command line options: -FS <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

Given exodus sidesets are treated as interior surfaces for STL projection.
Used with the **input_mesh** option when using an exodus mesh as the base grid. This may be useful if the **capture** option is enabled and some of the STL surfaces are close in proximity to the boundaries of the input exodus mesh. When close in proximity, sculpt will by default not project those boundary nodes to the STL surface but keep them on the domain boundary. If a list of sideset IDs are given here, the sideset faces will be projected

to the STL. The sideset IDs should refer to sidesets that are defined in the specified input_mesh exodus file.




Example of free_surface_sideset defined on the top surface faces of an input mesh

---

**Match Sideset Ids**

```
Command: match_sidesets     Sidesets ids of matching pairs

Input file command:   match_sidesets <arg>
Command line options: -mss <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

If used with an unstructured base grid (input mesh), this option allows the user to define a crack in the input mesh, where the faces of each vertical side (wall) of the crack are each in a different sideset. The faces at the bottom of the crack share a common edge (V-bottom) or face (square-bottom). Sculpt will match or equalize the volume fractions of the bottom cells on either side of the crack. This produces a uniform, higher quality mesh at the crack. The sidesets must be specified in a pairwise order. This option must be used with the **--input_mesh** (-im) option.

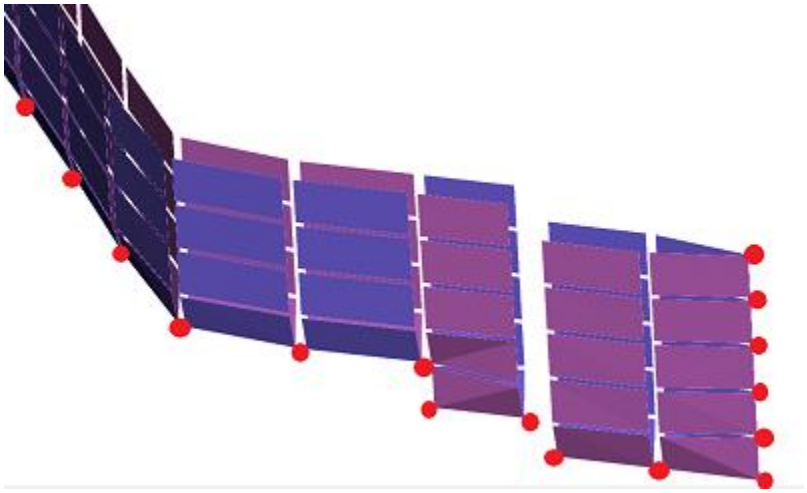
---

**Match Sideset Nodeset Id**

```
Command: match_sidesets_nodeset     Nodeset defining match_sidesets

Input file command:   match_sidesets_nodeset <arg>
Command line options: -msn <arg>
Argument Type:        integer >= 0
```

Command Description:

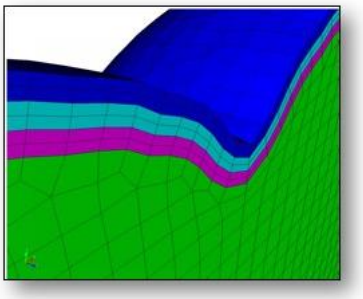
If using the match_sidesets option, this option is used to explicitly define the seam between match_sideset pairs, instead of deriving it. If deriving, the seam is where sideset boundaries share common edges, at the bottom of a crack. But at times the seam can be inadvertently found along the crack wall, if the sideset have been swapped. Explicitly specifing the seam through a nodeset prevents this.



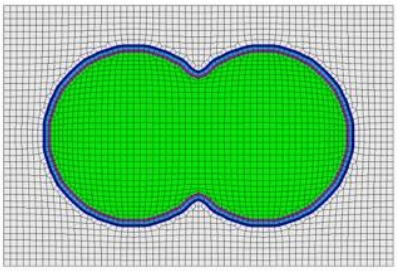
Specifying match_sideset_nodeset to explicitly define crack bottom.

# *Sculpt Boundary Layers*

Sculpt options for defining boundary layers in the mesh. Boundary layers are thin hex layers that can be defined at surfaces, extending either inward or outward from a material. The user may specify the number and thickness of the hex layers as well as the material ID of the layers. Layer thicknesses should normally be "thin" with respect to the size of the cells. Layers will not intersect, so should be defined on surfaces where nearby layers will not overlap. Boundary layers are specified based upon a material ID, where hex layers will be placed at surfaces where the material interfaces with other materials, or at free surfaces.



Example of boundary layers.



Boundary layers defined at the surfaces of a material.

```
Boundary Layers                -bly     --boundary_layer
  --begin                      -beg  <arg> Begin specification blayer or
blayer_block
  --end                        -zzz  <arg> End specification blayer or
blayer_block
  --material                   -mat  <arg> Boundary layer material
specification
  --num_elem_layers            -nel  <arg> Number of element layers in blayer
block
  --thickness                  -th   <arg> Thickness of first element layer in
block
  --bias                       -bi   <arg> Bias of element thicknesses in
blayer block
```

Sculpt Command Summary

---

**Boundary Layer Begin**

```
Command: begin     Begin specification blayer or blayer_block

Input file command:   begin <arg>
Command line options: -beg <arg>
Argument Type:        blayer, blayer_block
```

Command Description:

Defines the beginning of a specification block. Must be closed with "end" argument. Currently supports the following specifications:

**blayer**

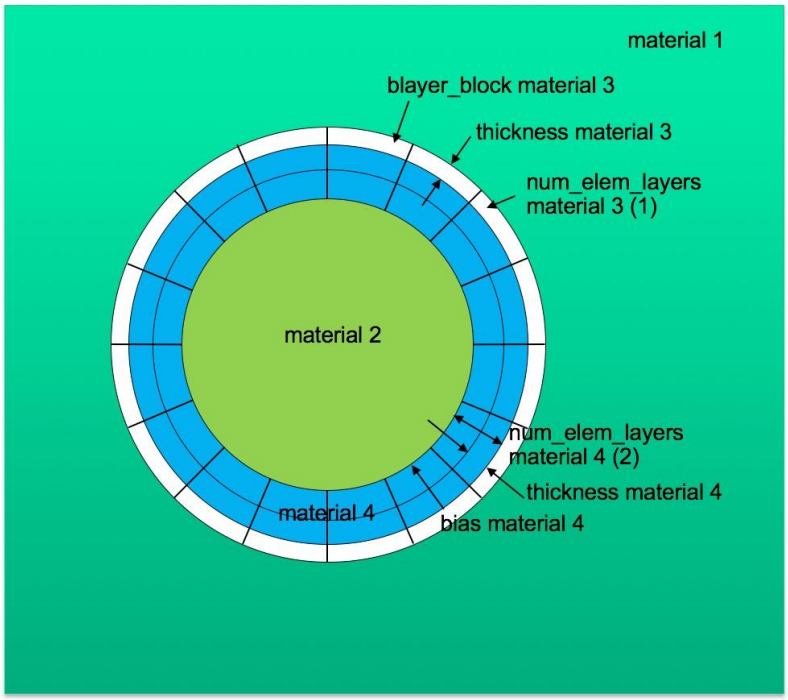
Defines a boundary layer specification. Layers of hex elements are placed at the interface of a given material. Valid argumnts used within a blayer specification include: material, and begin blayer_block.

**blayer_block**

Defines a set of element layers within a given blayer definition that share a common material ID. Valid arguments used within a blayer_block specification include: material, num_elem_layers, thickness and bias.

**Example:**
The following example shows a boundary layer specification in a sculpt input file. In this example, two boundary layer blocks are defined at the interface of materials 1 and 2. Two material blocks with ID 3 and 4 are generated with 1 and 2 element layers respectively.

```
BEGIN BLAYER
  MATERIAL = 1 2
  BEGIN BLAYER_BLOCK
    MATERIAL = 3
    NUM_ELEM_LAYERS = 1
    THICKNESS = 0.1
  END BLAYER_BLOCK
  BEGIN BLAYER_BLOCK
    MATERIAL = 4
    NUM_ELEM_LAYERS = 2
    THICKNESS = 0.2
    BIAS = 1.3
  END BLAYER_BLOCK
END BLAYER
```




Example schema for boundary layers corresponding to input file below.

---

**Boundary Layer End**

```
Command: end     End specification blayer or blayer_block

Input file command:   end <arg>
```

```
Command line options: -zzz <arg>
Argument Type:        blayer, blayer_block
```

Command Description:

Defines the end of a specification block. Must be preceded with "begin" argument. Currently supports arguments blayer and blayer_block.

---

#### Boundary Layer Material

```
Command: material     Boundary layer material specification

Input file command:   material <arg>
Command line options: -mat <arg>
Argument Type:        integer > 0
```

Command Description:

Defines a material ID in a boundary layer specification. When used within a BLAYER specification, it references one or two existing materials in the input where boundary layers will be generated. If a single material is specified, hex layers will be generated at all interfaces of the designated material with any adjacent material. If two material IDs are specified, layers will be generated only at interfaces where the two materials are adjacent. In most cases, the material ID(s) in the BLAYER specification refer to material IDs defined in the diatom file for specific geometry inserts such as STL files or diatom primitives. It can also be defined as the void material ID (VOID_MAT) or a material in a volume fraction description such as input_vfrac, input_micro, input_cart_exo or input_spn.
When used within a BLAYER_BLOCK specification, it refers to a new block that will be generated for which all elements in the blayer_block will be assigned. Normally it refers to a unique material ID that is not already referenced in the input. Where the material ID is already used, elements in the blayer block will be added to the existing material.
A material ID must be defined for both a BLAYER and BLAYER_BLOCK. This value does not have a default.

---

#### Number of Element Layers in Boundary Layer

```
Command: num_elem_layers     Number of element layers in blayer block

Input file command:   num_elem_layers <arg>
Command line options: -nel <arg>
Argument Type:        integer > 0
```

Command Description:

Number of element layers to be defined within a BLAYER_BLOCK specification. num_elem_layers must be defined for all BLAYER_BLOCKs.

---

**Boundary Layer Thickness**

```
Command: thickness     Thickness of first element layer in block

Input file command:   thickness <arg>
Command line options: -th <arg>
Argument Type:        floating point value
```

Command Description:

Thickness of the first layer defined in a BLAYER_BLOCK. Value is an absolute distance. No default is provided and must be defined for all BLAYER_BLOCKs

---

**Boundary Layer Bias**

```
Command: bias     Bias of element thicknesses in blayer block

Input file command:   bias <arg>
Command line options: -bi <arg>
Argument Type:        floating point value
```

Command Description:

Bias factor applied to additional layers of a BLAYER_BLOCK. Used in conjunction with the THICKNESS parameter (thickness of first layer) it defines a multiplier for the thickness for subsequent element layers defined within the same BLAYER_BLOCK. Default BIAS is 1.0 and is optional.

# *Sculpt Command Summary*

Following is a listing of the available input commands to either sculpt or psculpt. When used from the unix command line, commands may be issued using the short form argument, designated with a single dash(-), or with the longer form, designated with two dashes (--). When used in an input file, only the long form may be used, omitting the two dashes (--)

```
Process Control                -pc     --process
  --num_procs                  -j    <arg> Number of processors requested
  --input_file                 -i    <arg> File containing user input data
```

```
  --debug_processor          -D    <arg> Sleep to attach to processor for
debug
  --debug_flag               -dbf  <arg> Dump debug info based on flag
  --quiet                    -qt         Suppress output
  --print_input              -pi         Print input values and defaults then
stop
  --version                  -vs         Print version number and exit
  --threads_process          -tpp  <arg> Number of threads per process
  --iproc                    -ip   <arg> Number of processors in I direction
  --jproc                    -jp   <arg> Number of processors in J direction
  --kproc                    -kp   <arg> Number of processors in K direction
  --build_ghosts             -bg         Write ghost layers to exodus files
for debug
  --vfrac_method             -vm   <arg> Set method for computing volume
fractions

Input Data Files             -inp     --input
  --stl_file                 -stl  <arg> Input STL file
  --diatom_file              -d    <arg> Input Diatom description file
  --input_vfrac              -ivf  <arg> Input from Volume Fraction file base
name
  --input_micro              -ims  <arg> Input from Microstructure file
  --input_cart_exo           -ice  <arg> Input from Cartesian Exodus file
  --input_spn                -isp  <arg> Input from Microstructure spn file
  --spn_xyz_order            -spo  <arg> Ordering of cells in spn file
  --compress_spn_ids         -csp  <arg> Compress IDs from SPN file
  --input_stitch             -ist  <arg> Input from Stitch file
  --stitch_timestep          -stt  <arg> Timestep in Stitch file to read
  --stitch_timestep_id       -stn  <arg> Timestep ID in Stitch file to read
  --stitch_field             -stf  <arg> Field in Stitch file to read
  --stitch_info              -sti        List header info for Stitch file
  --lattice                  -l    <arg> STL Lattice Template File

Output                       -out     --output
  --exodus_file              -e    <arg> Output Exodus file base name
  --large_exodus             -le   <arg> Output large Exodus file(s)
  --volfrac_file             -vf   <arg> Output Volume Fraction file base
name
  --quality                  -Q    <arg> Dump quality metrics to file
  --export_comm_maps         -C          Export parallel comm maps to debug
exo files
  --write_geom               -G          Write geometry associativity file
  --write_mbg                -M          Write mesh based geometry file
<beta>
  --compare_volume           -cv         Report vfrac and mesh volume
comparison
  --compute_ss_stats         -css        Report sideset statistics

Overlay Grid Specification   -ovr     --overlay
  --nelx                     -x    <arg> Num cells in X in overlay Cartesian
grid
  --nely                     -y    <arg> Num cells in Y in overlay Cartesian
grid
  --nelz                     -z    <arg> Num cells in Z in overlay Cartesian
grid
  --xmin                     -t    <arg> Min X coord of overlay Cartesian
grid
```

```
  --ymin                        -u    <arg> Min Y coord of overlay Cartesian
grid
  --zmin                        -v    <arg> Min Z coord of overlay Cartesian
grid
  --xmax                        -q    <arg> Max X coord of overlay Cartesian
grid
  --ymax                        -r    <arg> Max Y coord of overlay Cartesian
grid
  --zmax                        -s    <arg> Max Z coord of overlay Cartesian
grid
  --cell_size                   -cs   <arg> Cell size (nelx, nely, nelz ignored)
  --align                       -a          Automatically align geometry to grid
  --bbox_expand                 -be   <arg> Expand tight bbox by percent
  --input_mesh                  -im   <arg> Input Base Exodus mesh
  --input_mesh_blocks           -imb  <arg> Block ids of Input Base Exodus mesh
  --input_mesh_material         -imm  <arg> Material definition with input mesh
  --input_mesh_pamgen           -imp  <arg> Input Base mesh defined by Pamgen
  --join_parallel               -jp   <arg> Join parallel files

Mesh Type                       -typ     --type
  --stair                       -str  <arg> Generate Stair-step mesh
  --mesh_void                   -V    <arg> Mesh void
  --trimesh                     -tri        Generate tri mesh of geometry
surfaces
  --tetmesh                     -tet  <arg> Under Development
  --deg_threshold               -dg   <arg> Convert hexes below threshold to
degenerates
  --max_deg_iters               -dgi  <arg> Maximum number of degenerate
iterations
  --htet                        -ht   <arg> Convert hexes below quality
threshold to tets
  --htet_method                 -hti  <arg> Method used for splitting hexes to
tets
  --htet_material               -htm  <arg> Convert hexes in given materials to
tets
  --htet_transition             -htt  <arg> Transition method between hexes and
tets
  --htet_pyramid                -htp  <arg> Local transition pyramid
  --htet_tied_contact           -htc  <arg> Local transition tied contact
  --htet_no_interface           -htn  <arg> Local transition none
  --periodic                    -per        Generate periodic mesh
  --check_periodic              -cp   <arg> Check for periodic geometry
  --check_periodic_tol          -cpt  <arg> Tolerance for checking periodicity
  --periodic_axis               -pax  <arg> Axis periodicity is about
  --periodic_nodesets           -pns  <arg> Nodesets ids of master/slave
nodesets

Boundary Conditions             -bc     --boundary_condition
  --void_mat                    -VM   <arg> Void material ID (when
mesh_void=true)
  --separate_void_blocks        -SVB        Separate void into unique block IDs
  --material_name               -mn   <arg> Label Material (Block) with Name
  --sideset_name                -sn   <arg> Label Sideset with Name
  --nodeset_name                -nn   <arg> Label Nodeset with Name
  --sideset                     -sid  <arg> User Defined Sideset
  --nodeset                     -nid  <arg> User Defined Nodeset
  --gen_sidesets                -SS   <arg> Generate sidesets
```

```
  --free_surface_sideset     -FS   <arg> Free Surface Sideset
  --match_sidesets           -mss  <arg> Sidesets ids of matching pairs
  --match_sidesets_nodeset   -msn  <arg> Nodeset defining match_sidesets

Adaptive Meshing             -adp     --adapt
  --adapt_type               -A    <arg> Adaptive meshing type
  --adapt_threshold          -AT   <arg> Threshold for adaptive meshing
  --adapt_levels             -AL   <arg> Number of levels of adaptive
refinement
  --adapt_material           -AM   <arg> Info for adapting material
  --adapt_export             -AE         Export exodus mesh of refined grid
  --adapt_non_manifold       -ANM        Refine at non-manifold conditions
  --adapt_load_balance       -ALB        Adaptive parallel load balancing
  --adapt_memory_stats       -AMS        Write memory usage stats for
adaptivity

Smoothing                    -smo     --smoothing
  --smooth                   -S    <arg> Smoothing method
  --csmooth                  -CS   <arg> Curve smoothing method
  --laplacian_iters          -LI   <arg> Number of Laplacian smoothing
iterations
  --max_opt_iters            -OI   <arg> Max. number of parallel Jacobi opt.
iters.
  --opt_threshold            -OT   <arg> Stopping criteria for Jacobi opt.
smoothing
  --curve_opt_thresh         -COT  <arg> Min metric at which curves won't be
honored
  --max_pcol_iters           -CI   <arg> Max. number of parallel coloring
smooth iters.
  --pcol_threshold           -CT   <arg> Stopping criteria for parallel color
smooth
  --max_gq_iters             -GQI  <arg> Max. number of guaranteed quality
smooth iters.
  --gq_threshold             -GQT  <arg> Guaranteed quality minimum SJ
threshold
  --geo_smooth_max_deviation -GSM  <arg> Geo Smoothing Maximum Deviation

Mesh Improvement             -imp     --improve
  --pillow                   -p    <arg> Set pillow criteria (1=surfaces)
  --pillow_surfaces          -ps         Turn on pillowing for all surfaces
  --pillow_curves            -pcv        Turn on pillowing for bad quality at
curves
  --pillow_boundaries        -pb         Turn on pillowing at domain
boundaries
  --pillow_curve_layers      -pcl  <arg> Number of elements to buffer at
curves
  --pillow_curve_thresh      -pct  <arg> S.J. threshold to pillow hexes at
curves
  --pillow_smooth_off        -pso        Turn off smoothing following pillow
operations
  --capture                  -c    <arg> Project to facet geometry <beta>
  --capture_angle            -ca   <arg> Angle at which to split surfaces
<beta>
  --capture_side             -sc   <arg> Project to facet geometry with
surface ID
  --defeature                -df   <arg> Apply automatic defeaturing
  --min_vol_cells            -mvs  <arg> Minimum number of cells in a volume
```

```
  --defeature_bbox            -dbb        Defeature Filtering at Bounding Box
  --defeature_iters           -dfi  <arg> Maximum Number of Defeaturing
Iterations
  --thicken_material          -thm  <arg> Expand a given material into
surrounding cells
  --thicken_void              -thv  <arg> Insert void material to remove
overlap
  --micro_expand              -me   <arg> Expand Microstructure grid by N
layers
  --micro_shave               -ms         Remove isolated cells at micro.
boundaries
  --remove_bad                -rb   <arg> Remove hexes with Scaled Jacobian <
threshold
  --wear_method               -wm   <arg> Method for removing void at free
surface
  --crack_min_elem_thickness -cmet <arg> Minimum element thickness in crack
  --min_num_layers            -mnl  <arg> Minimum number of layers to keep
using wear_method=2

Mesh Transformation           -tfm     --transform
  --xtranslate                -xtr  <arg> Translate final mesh coordinates in
X
  --ytranslate                -ytr  <arg> Translate final mesh coordinates in
Y
  --ztranslate                -ztr  <arg> Translate final mesh coordinates in
Z
  --xscale                    -xsc  <arg> Scale final mesh coordinates in X
  --yscale                    -ysc  <arg> Scale final mesh coordinates in Y
  --zscale                    -zsc  <arg> Scale final mesh coordinates in Z

Boundary Layers               -bly     --boundary_layer
  --begin                     -beg  <arg> Begin specification blayer or
blayer_block
  --end                       -zzz  <arg> End specification blayer or
blayer_block
  --material                  -mat  <arg> Boundary layer material
specification
  --num_elem_layers           -nel  <arg> Number of element layers in blayer
block
  --thickness                 -th   <arg> Thickness of first element layer in
block
  --bias                      -bi   <arg> Bias of element thicknesses in
blayer block
```

# *Sculpt Mesh Improvement*

Sculpt options for modifying the mesh to improve mesh quality.
Automatic smoothing provides an effective method for improving element quality. However there may be some cases that cannot be improved with smoothing alone. The options included in this section will apply changes to the underlying hex mesh or to the volume fraction data to increase the opportunity for smoothing to produce a good quality mesh.

```
Mesh Improvement                -imp     --improve
  --pillow                      -p    <arg> Set pillow criteria (1=surfaces)
  --pillow_surfaces             -ps         Turn on pillowing for all surfaces
  --pillow_curves               -pcv        Turn on pillowing for bad quality at
curves
  --pillow_boundaries           -pb         Turn on pillowing at domain
boundaries
  --pillow_curve_layers         -pcl  <arg> Number of elements to buffer at
curves
  --pillow_curve_thresh         -pct  <arg> S.J. threshold to pillow hexes at
curves
  --pillow_smooth_off           -pso        Turn off smoothing following pillow
operations
  --capture                     -c    <arg> Project to facet geometry <beta>
  --capture_angle               -ca   <arg> Angle at which to split surfaces
<beta>
  --capture_side                -sc   <arg> Project to facet geometry with
surface ID
  --defeature                   -df   <arg> Apply automatic defeaturing
  --min_vol_cells               -mvs  <arg> Minimum number of cells in a volume
  --defeature_bbox              -dbb        Defeature Filtering at Bounding Box
  --defeature_iters             -dfi  <arg> Maximum Number of Defeaturing
Iterations
  --thicken_material            -thm  <arg> Expand a given material into
surrounding cells
  --thicken_void                -thv  <arg> Insert void material to remove
overlap
  --micro_expand                -me   <arg> Expand Microstructure grid by N
layers
  --micro_shave                 -ms         Remove isolated cells at micro.
boundaries
  --remove_bad                  -rb   <arg> Remove hexes with Scaled Jacobian <
threshold
  --wear_method                 -wm   <arg> Method for removing void at free
surface
  --crack_min_elem_thickness -cmet <arg> Minimum element thickness in crack
  --min_num_layers              -mnl  <arg> Minimum number of layers to keep
using wear_method=2
```

Sculpt Command Summary

---

### Pillow

```
Command: pillow     Set pillow criteria (1=surfaces)

Input file command:   pillow <arg>
Command line options: -p <arg>
Argument Type:        integer (0, 1, 2, 3)
Input arguments: off (0)
                 surfaces (1)
                 curves (2)
                 domain_boundaries (3)
```

```
surfaces_no_smoothing (100)
curves_2_layers (212)
curves_3_layers (213)
curves_4_layers (214)
curves_5_layers (215)
curves_2_layers_no_smoothing (202)
curves_3_layers_no_smoothing (203)
curves_4_layers_no_smoothing (204)
curves_5_layers_no_smoothing (205)
```

Command Description:

For models that have more than one material that share an interface, unless the geometry is precisely aligned with the global axis, it is usually a good idea to turn on pillowing. Pillowing automatically inserts an additional layer of hexes at interface boundaries to improve mesh quality. Without pillowing you may notice inverted or poor quality elements at curve interfaces where 2 or more materials meet.
The pillow option will generate an additional layer of hexes at surfaces as a means to improve element quality near curve interfaces. This is intended to eliminate the problem of 3 or more nodes from a single hex face lying on the same curve. Use one or more of the following options to set up pillowing:

- **pillow_surfaces:** Pillow around all surfaces
- **pillow_curves:** Pillow bad quality at curves
- **pillow_boundaries:** Pillow at domain boundaries
- **pillow_curve_layers:** Number of element layers to buffer curves
- **pillow_smooth_off:** Turn OFF smoothing following pillow operations

See help on the above options for more information

---

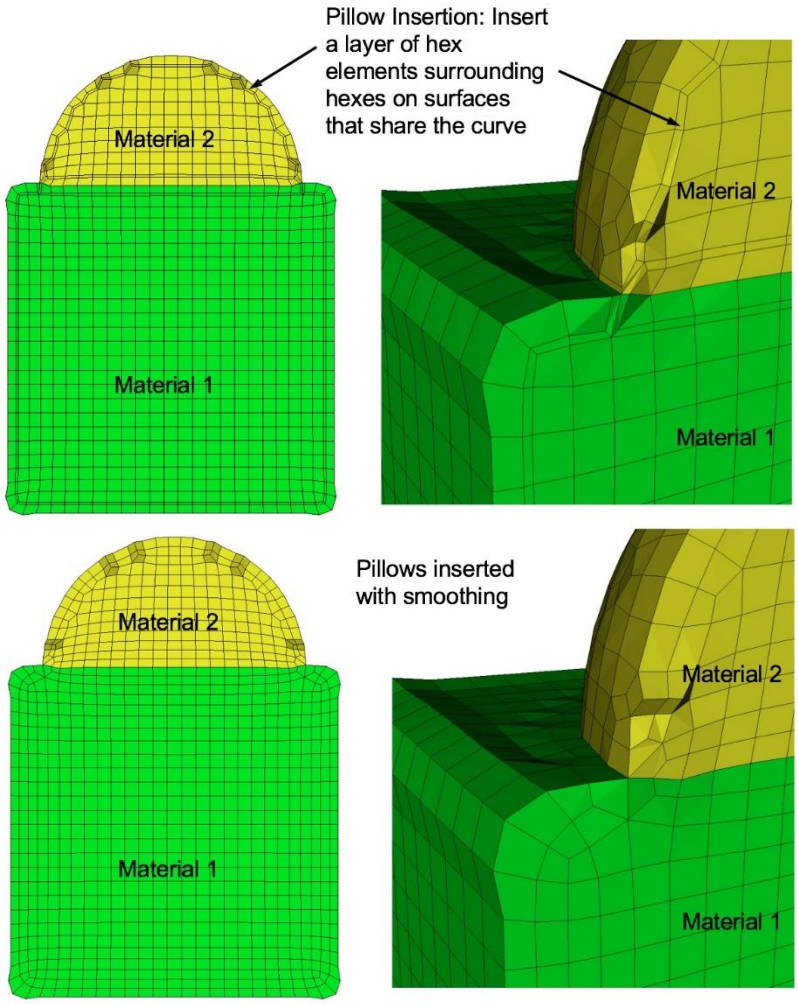
#### Pillow All Surfaces

```
Command: pillow_surfaces     Turn on pillowing for all surfaces

Input file command:   pillow_surfaces
Command line options: -ps
```

Command Description:

Pillow option to insert a layer of hexes surrounding each internal surface in the mesh. Where two volumes share a common interface is defined as a surface. All hexes that have at least one of its faces on a surface are defined as the "shrink set" of hexes. A separate shrink set is defined for each unique surface. Hexes in the set are shrunk away from their hex neighbors not in the shrink set. A layer of hexes is then inserted surrounding all hexes in each set. This enforces the condition where no more than one hex edge will

lie on any single curve thus allowing more freedom for the smoother to improve element quality.




Example of surface pillowing, before and after smoothing

Surface pillowing is off by default. If both **pillow_curves** and **pillow_surfaces** options are used, curve pillowing will be performed before surface pillowing.
See the **pillow** option for more information on setting additional options for pillowing.

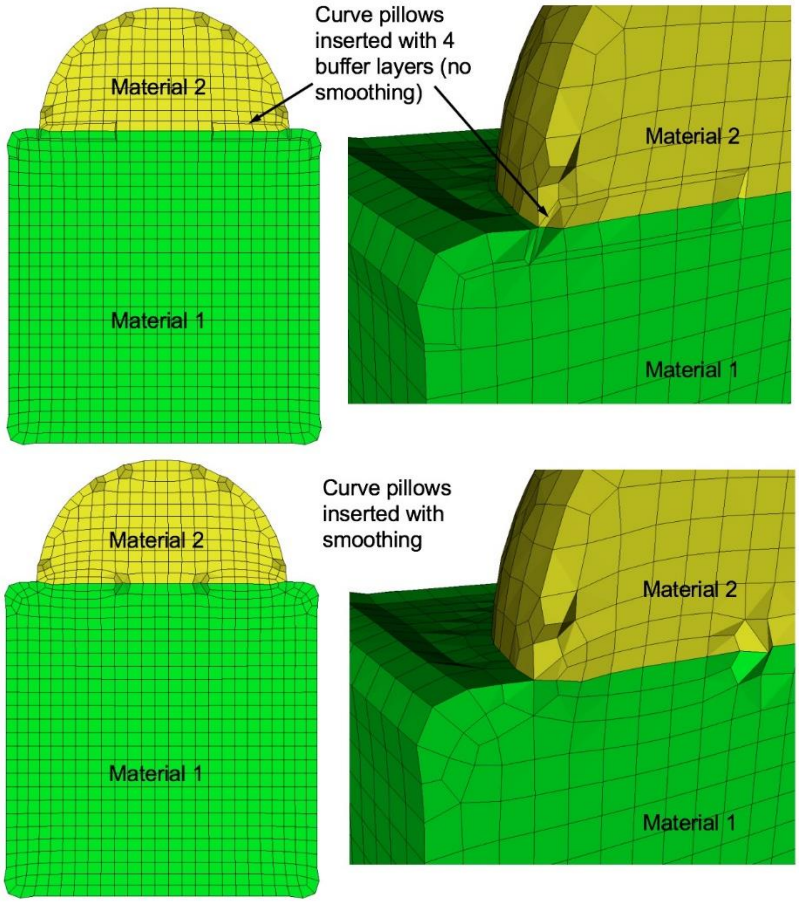
---

#### Pillow Bad Quality at Curves

```
Command: pillow_curves     Turn on pillowing for bad quality at curves

Input file command:   pillow_curves
Command line options: -pcv
```

Command Description:

Pillow option to selectively pillow hexes at curves. Only hexes that have faces with 3 or more nodes on a curve will be pillowed. Additional buffer layers of hexes beyond the poor quads at the curves will be included in the pillow region. The number of buffer layers beyond the curve can be controlled with the **pillow_curve_layers**, where the default will be 3 layers.




Example of curve pillowing with four pillow_curve_layers, before and after smoothing

Curve pillowing is off by default. If both **pillow_curves** and **pillow_surfaces** options are used, curve pillowing will be performed before surface pillowing.
See the **pillow** option for more information on setting additional options for pillowing.

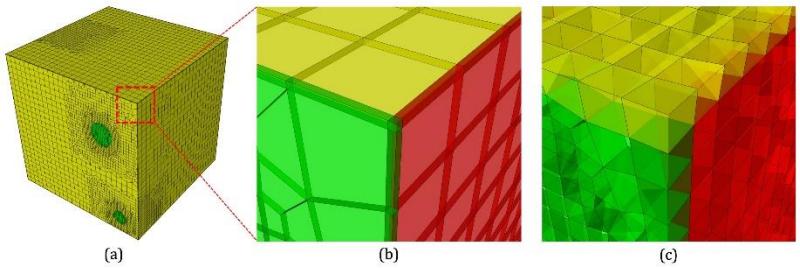
---

**Pillow at Domain Boundaries**

```
Command: pillow_boundaries     Turn on pillowing at domain boundaries

Input file command:   pillow_boundaries
Command line options: -pb
```

Command Description:

Pillow option to insert pillow layers at domain boundaries of the initial Cartesian grid definition. One layer of hexes is inserted on each of the six faces of the Cartesian Domain. This option is useful where the void option is used to generate a mesh in the full Cartesian grid and where the adapt option has been used. Without this option, it is likely that hexes with two faces on the same domain boundary will occur if the adaptation extends to the boundary. Turning on the **pillow_boundaries** option should correct for these cases.




Example of pillowing at boundaries on a microstructure RVE. (b) before smoothing (c) after smoothing

Boundary pillowing is off by default. The **pillow_boundaries** option may be used in the same input as **pillow_surfaces** or **pillow_curves**. The **pillow_boundaries** option must also be used with the **mesh_void** option to ensure hexes will exist at the Cartesian domain boundary. See the **pillow** option for more information on setting additional options for pillowing.

---

#### Number of Element Layers to Buffer Curves

```
Command: pillow_curve_layers     Number of elements to buffer at curves

Input file command:   pillow_curve_layers <arg>
Command line options: -pcl <arg>
Argument Type:        integer > 0
```

Command Description:

Used for setting the number of buffer hex layers when the **pillow_curves** option is used. When **pillow_curves** is used a shrink set is formed from hexes that would otherwise have two or more edges on the same curve. This value will control the extent to which neighboring hexes will be included in the shrink set. The default **pillow_curve_layers** is 3. Setting this value lower will localize the modifications to the hex mesh, whereas, more layers will extend the region that is affected in correcting the poor quality at curves.

---

#### Scaled Jacobian Threshold for Curve Pillowing

```
Command: pillow_curve_thresh     S.J. threshold to pillow hexes at curves

Input file command:   pillow_curve_thresh <arg>
Command line options: -pct <arg>
Argument Type:        floating point value (-1.0->1.0)
```

Command Description:

Used for setting the quality threshold for pillowing hexes at curves. When determining hexes to include in the shrink set, the **pillow_curves** option will look for hexes with more than two nodes of a hex on the same curve. If this condition is satisfied, it will test the mesh quality of quads on the adjacent surfaces that share the common curve. If at least 3 nodes are on a common curve and the Scaled Jacobian of any of the attached quads falls below the, **pillow_curve_thresh** scaled Jacobian metric, then the associated hexes will be included in the shrink set.
Default for **pillow_curve_thresh** is 0.3. Increasing this value will tend to increase the total number of hexes added to the mesh, but may result in better mesh quality after smoothing. Lowering this value may reduce the number of additional hexes but could potentially result in more hexes with poor or bad Scaled Jacobian metrics.

---

#### Turn OFF Smoothing Following Pillow Operations

```
Command: pillow_smooth_off     Turn off smoothing following pillow operations

Input file command:   pillow_smooth_off
Command line options: -pso
```

Command Description:

Controls the smoothing following pillow operations. To maximize element quality at pillowed hexes, smoothing is always performed after inserting the hex layers. The smoothing step may be omitted if **pillow_smooth_off** is set. This option can be useful for visualizing the pillow layers that have been inserted, but in most cases will generate poor quality or inverted elements.

---

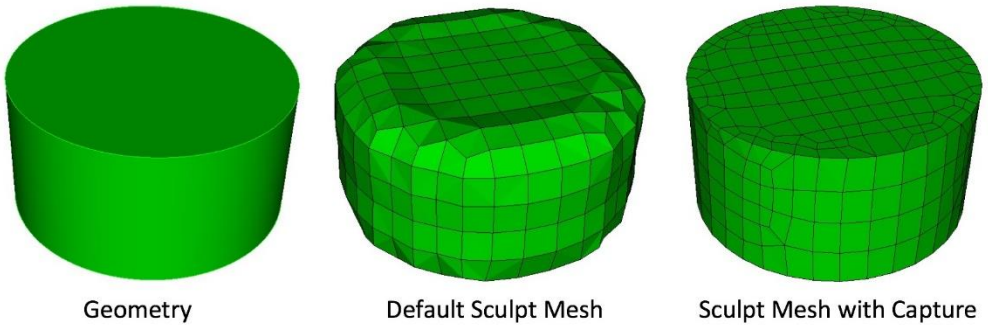
#### Capture

```
Command: capture     Project to facet geometry <beta>

Input file command:   capture <arg>
Command line options: -c <arg>
```

```
Argument Type:        integer (0, 1, 2)
Input arguments: off (0)
                 on (1)
                 external_surfaces (2)
                 projections_only (3)
                 feature_angle_smooth (4)
                 topology_smooth (5)
```

Command Description:

This is an experimental option still in development. Nodes at the surfaces of a default sculpt mesh will not necessarily exactly lie on the geometric surfaces prescribed by the input STL geometry. While this characteristic can provide additional flexibility for defeaturing and element quality, there are cases where a more exact surface representation may be desired. The capture option attempts to address this by extracting sharp features and/or projecting nodes to the facet geometry.




Simple example illustrating the effect of the **capture = 5** option. Options **smooth = to_geometry** and **pillow_curves = true** are also used for this example.

Several options are currently being studied as possible solutions. They include the following:
0 = (off) Capture option is off. No attempt is made at capturing sharp features.
1 = (on) STL geometry is used as basis for feature capture. A user defined feature angle is used (capture_angle) to first generate groups of facets from the STL geometry based on capture_angle. Topological curves are defined based on projections to closest surface facets and edges. With default smoothing option, the surface nodes will be projected to the closest STL surfaces as a final step before exporting the exodus mesh. Consider using **smooth = to_geometry** option.
2 = (exterior_surfaces) Only exterior surfaces are captured. Uses the same procedure as described in capture = 1, except that interior surfaces (those with two adjacent volumes), will be ignored in the capture and projections stage.
3 = (projections_only) For this option, additional topology based on feature angle is not extracted. Only the final projection of surface nodes to the STL facets is done. Note that this option is useful for organic shapes that do not have sharp features, or where sharp features should be ignored.

4 = (feature_angle_smooth) This option uses the procedure outlined in **capture = 1**, except that the **smooth = to_geometry** is used by default. Note that **capture = 1** used with **smooth = to_geometry** should be identical to this option.
5 = (topology_smooth) Curve topology is defined similar to **capture = 1**, except that element face topology is first determined based on closest assigned facet. Curve topology is then extracted based on adjacent element face associativity. Surface node projections are only done for nodes that have unambiguous neighbor associativity. This provides for a tolerant approach to resolving topology that may result in defeaturing. (i.e. where the STL facet topology may be locally more complex than can be resolved by the prescribed resolution). This option also uses the **smooth = to_geometry** option as default for smoothing. Also note that **capture = 5** it is only currently available for serial execution (j=1)

---

#### Capture Angle

```
Command: capture_angle     Angle at which to split surfaces <beta>

Input file command:   capture_angle <arg>
Command line options: -ca <arg>
Argument Type:        floating point value (0 -> 360)
```

Command Description:

This is an experimental option still in development. Feature angle for capture option.

---

#### Capture Side

```
Command: capture_side     Project to facet geometry with surface ID

Input file command:   capture_side <arg>
Command line options: -sc <arg>
Argument Type:        integer > 0
```

Command Description:

Similar to the capture option, the capture_side option will project nodes to the initial triangle facets, however projections will be limited only to surface nodes closest to the surface ID specified by the argument. Note that the input STL file can identify and group facets according to a surface ID. However surface IDs are utilized only when using the gen_sidesets option with arguments 3 and 4. When using Cubit, the STL file written when using the sculpt parallel command with sideset options 3 and 4 will include surface identification for surfaces in the STL file. A workflow for using the capture_side option might include the following:

1. Generate or import CAD model into Cubit.
2. Identify and group selected surfaces into a single sideset with unique ID.
3. In the Sculpt GUI set the sideset generation option to 4.
4. Turn on the option: Export Run Files Only.
5. In your working directory edit the input file (.i).
6. Add the option capture_side = <id> to the input file.
7. Run sculpt in batch using the input file to control execution.

The result should be a mesh where surface nodes closest to the surfaces identified by the unique sideset ID will lie precisely on their closest surface.

---

#### Defeature

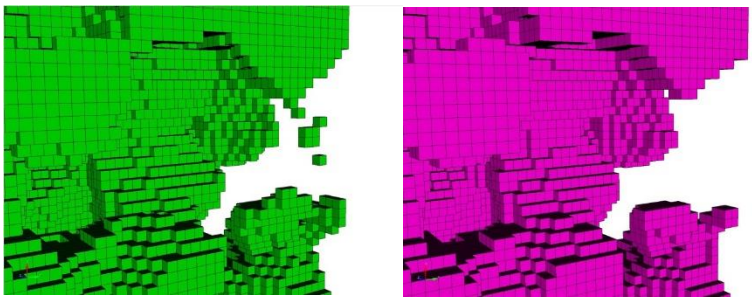
```
Command: defeature     Apply automatic defeaturing

Input file command:   defeature <arg>
Command line options: -df <arg>
Argument Type:        integer (0, 1, 2, 3)
Input arguments: off (0)
                 filter (1)
                 collapse (2)
                 filter_and_collapse (3)
```

Command Description:

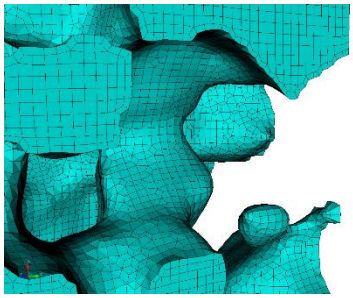
Option to automatically detect and remove small features. Primarily used for defeaturing microstructure data, however can be used with any input format. The following options are available:

- **off (0):** No defeaturing performed (default)
- **filter (1):** Filters the Cartesian grid data so that groupings of cells of a common material with less than **min_vol_cells** will be reassigned to the predominant neighboring material. If the **min_vol_cells** argument is not specified, the minimum number of cells in a volume will be set to 5. This has the effect of removing small volumes that would otherwise be generated. This option will also remove protrusions, where a cell surrounded on 4 or 5 sides by another material ID will be reassigned to the predominant neighboring material. This option is available with multiple processors.

  See also the **defeature_iters** and **defeature_bbox** options for additional control of the **defeature = filter** option. The **compare_volume** option can also be used to validate that changes made to material volumes are within acceptable limits.

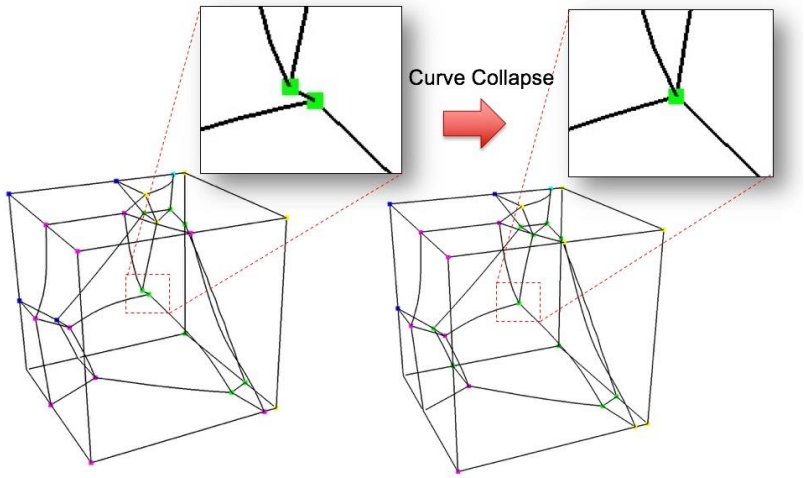


Example grid cells before and after defeaturing has been applied



Final mesh after using defeaturing.

- **collapse (2):** Curve and surface collapses are performed. This option is only available when used with the **trimesh** option. After geometry has been extracted and built from the volume fraction data curves containing exactly one mesh edge are collapsed into a single vertex. Surfaces that are identified with exactly 2 curves, each of which have 2 mesh edges are collapsed into a single curve. Only available as serial option (-j 1)




Example collapsing of small curve on microstructure model when using defeature=2 and trimesh option

- **filter_and_collapse (3):** Performs both option **filter (1)** and **collapse (2)** on a **trimesh**. Only available as serial option (-j 1)

---

#### Minimum Number of Cells in a Volume

```
Command: min_vol_cells     Minimum number of cells in a volume

Input file command:   min_vol_cells <arg>
Command line options: -mvs <arg>
Argument Type:        integer >= 0
```

Command Description:

When used with **defeature** options **filter (1)** or **filter_and_collapse (3)**, specifies the minimum number of cells below which a volume will be eliminated. The cells of small volumes will be absorbed into the predominant material of the neighboring cells. If not specified and defeature options **filter (1)** or **filter_and_collapse (3)** are used, the **min_vol_cells** value will be set to 5.

---

#### Defeature at Bounding Box

```
Command: defeature_bbox     Defeature Filtering at Bounding Box

Input file command:   defeature_bbox
Command line options: -dbb
```

Command Description:

The **defeature_bbox** option is used in conjunction with **defeature = filter (1)**. It is used to modify the defeature filter criteria at cells that are immediately adjacent to the Cartesian grid's domain boundary. It is most effective for microstructure data but can be used with any input format. The **defeature = filter (1)** option will remove protrusions identified by cells that are surrounded on 4 or 5 sides by another material. For cells that are at the domain boundary, cells will have missing adjacent cells on at least one face. If the **defeature_bbox=true** option is used, the missing adjacent cells are considered a different material and counted in the 4 or 5 surrounding cells with a different material. In contrast, the **defeature_bbox=false** option will not count the missing adjacent cells. Using the **defeature_bbox=true** has the effect of more aggressively modifying cells at the domain boundaries to avoid protrusions. The default for this option is **defeature_bbox=false**. It will be ignored if **defeature = filter (1)** is not used.

---

#### Maximum Number of Defeature Iterations

```
Command: defeature_iters     Maximum Number of Defeaturing Iterations

Input file command:   defeature_iters <arg>
Command line options: -dfi <arg>
Argument Type:        integer >=0
```

Command Description:

Used with the **defeature** option. Controls the maximum number of iterations of defeature filtering that will be performed. Setting this value greater than the default of 10 can be useful for very noisy data where a significant number of iterations will need to be performed to resolve the geometry.
When performing non-manifold resolution, the defeature state of some of the cells may be effected. As a result, the defeaturing and non-manifold resolution procedures are performed in a loop until no further changes can be made. The **defeature_iters** sets the maximum number of defeature and non-manifold resolution procedures that will be performed. Note that if defeaturing reaches the maximum iteration value without completely resolving all non-manifold conditions, that subsequent sculpt procedures may not succeed. Set this value higher to allow the defeaturing and non-manifold resolution to run to completion. The **stair = 1** option can be used to interrogate the model to see where non-manifold conditions may still exist.

---

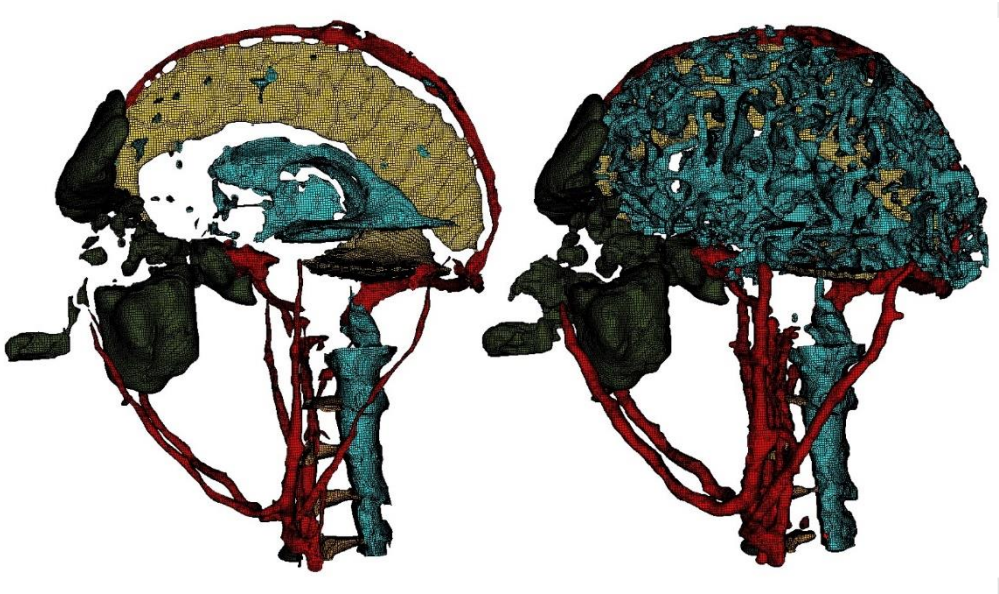
#### Thicken a material

```
Command: thicken_material     Expand a given material into surrounding cells

Input file command:   thicken_material <arg>
Command line options: -thm <arg>
Argument Type:        integer >= 0 floating point value (0.0->1.0)
```

Command Description:

Add additional cells at the boundary of a given material. Takes two input values, a material and a volume fraction between 0 and 1. This option is useful for noisy input data that may not form contiguous volumes. Thickening a material may close small gaps making the material continuous. To perform the thicken operation, cells in adjacent materials are removed and reassigned to the indicated material. This option requires both a valid material ID and volume fraction value, where the volume fraction represents the amount of material to be added to each neighboring cell. For example:
**thicken material = 1 0.2**
**thicken_material = 2 0.5**
each neighboring cell to material 1 will change approximately 20 percent of its volume to be material 1. Other materials present in the cell will be decreased accordingly to maintain

a sum of 1.0 for each cell. Additional material is accumulated in neighboring cells from each adjacent cell it shares with material 1, so that if for example a neighbor cell shares faces with three cells of material 1, it will add 0.6 (0.2 X 3) of material 1 volume fraction to the neighbor. If more than one **thicken_material** option is used, the thicken operation will be performed in the order they appear in the input. For the above example, material 1 would first be thickened, followed by material 2. If materials 1 and 2 are adjacent, thickening in this case, material 2 would take precedence, potentially removing cells from material 1 at their interface.



Bitmap input is used on a Cartesian base grid to generate the mesh for complex head and brain anatomy. Left: Some of the materials prior to applying the thicken_material option. Right: After applying the thicken_material option.

---

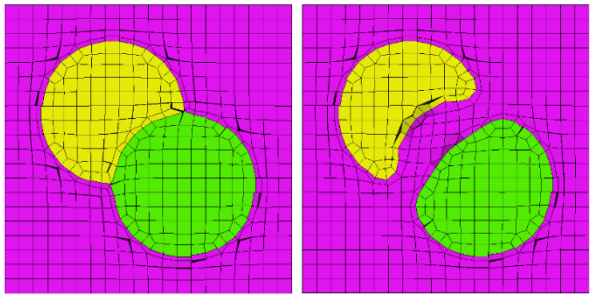
**Thicken void material**

```
Command: thicken_void     Insert void material to remove overlap

Input file command:   thicken_void <arg>
Command line options: -thv <arg>
Argument Type:        floating point value (0.0->1.0)
```

Command Description:

Add additional void material when non-void material is detected as touching or immediately adjacent. Takes one input value, a volume fraction normally about 1.0 that indicates the quantity of volume fraction inserted at each node if the input grid where non-void material adjacency is detected. A value of 1.0 indicates void material equal to the volume of one cell will be added at the nodes, reducing the volume fractions of other

materials present in adjacent cells. Smaller input values will generate a smaller gap between materials, but can run the risk of materials bleeding into one another.
This option is useful when it is known that non-void materials in the model should not touch, instead should have a gap where they would otherwise touch or overlap. For example:
**thicken_void = 1.0**
each node where its adjacent cells have two or more non-void materials present will have additional void material added. In this case, if 8 adjacent cells are assumed, a contribution of 1/8 void volume fraction will be added to each adjacent cell to the node. Other materials present in the cells will be decreased accordingly to maintain a sum of 1.0 for each cell.



Left: Initial mesh without **thicken_void**. Right: Mesh with **thicken_void=1**. Void material (Magenta elements) is inserted between the yellow and green materials to ensure separation.

---

#### Microstructure Expansion

```
Command: micro_expand     Expand Microstructure grid by N layers

Input file command:   micro_expand <arg>
Command line options: -me <arg>
Argument Type:        integer >= 0
```

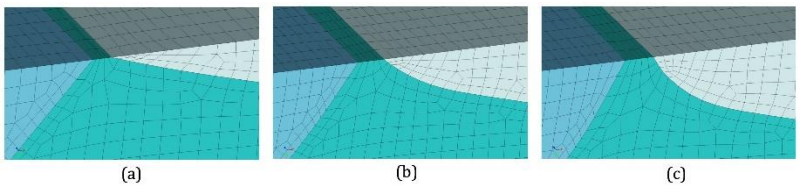
Command Description:

This option expands the Cartesian grid by a specified number of layers. It can be used with any of the following input options:

```
--input_micro
--input_cart_exo
--input_spn
```

In some cases the interior material interfaces may intersect the domain boundaries at small acute angles. When this occurs it may be difficult or impossible to achieve computable mesh quality at these intersections. To address this problem, one or more layers of hexes may be added to the Cartesian grid. The volume fractions from cells at the boundary are copied to generate additional layers. This has the effect of increasing

the angle of intersection for any material interfaces intersecting the domain boundary. Usualy a value of 1 or 2 is sufficient to sufficiently improve quality.
Note that the resulting mesh in the expanded layers serves only to improve mesh quality and will only duplicate existing data at the boundaries. It may not reflect the actual material structure within the expansion layers.




(a) Initial mesh (b) One expansion layer added (c) Two expansion layers added

---

**Microstructure Shave**

```
Command: micro_shave     Remove isolated cells at micro. boundaries

Input file command:   micro_shave
Command line options: -ms
```

Command Description:

This option potentially modifies the outermost layer of Cartesian cells of a microstructures file. It will identify isolated cells where the assigned material is unique from all of its surrounding cells at the boundary. When this occurs, the cell material is reassigned to the dominant nearby material.
This option is useful if it is noted that a cell structure just barely grazes the exterior planar boundary surface. Poor quality elements can often result with this condition. The micro_shave option will, in effect, remove material from the cell structure, but will result in better quality elements by removing the intersection region with the boundary.
micro_shave can be used with any of the following input options:

```
--input_micro
--input_cart_exo
--input_spn
```

---

**Remove bad elements below threshold**

```
Command: remove_bad     Remove hexes with Scaled Jacobian < threshold

Input file command:   remove_bad <arg>
Command line options: -rb <arg>
```

```
Argument Type:        floating point value -1.0 >= 1.0
```

Command Description:

Remove hexes below the specified scaled Jacobian metric.

---

#### Method for removing void at free surface

```
Command: wear_method     Method for removing void at free surface

Input file command:   wear_method <arg>
Command line options: -wm <arg>
Argument Type:        integer (0, 1, 2)
Input arguments: cell (0)
                 cell (1)
                 sheet (2)
```

Command Description:

This option defines how the mesh removes void elements that are at the **free_surface_sideset**. Used with the **free_surface_sideset**, **input_mesh** and **capture=5** options. Normally with the default **wear_method = cell (0)** option, elements in the input_mesh outside of the specified **free_surface_sideset** are removed when the percent of its volume (volume fraction) lying outside of the free_surface_sideset exceeds 50 percent. If the **wear_method = sheet (2)** option is set, volume fraction of continuous layers of elements (sheets) that lie at the **free_surface_sideset** can be removed. The entire sheet of elements will be retained or removed based on the total volume fraction of the elements in the sheet.
The **wear_method = sheet (2)** option is most useful when used with a swept **input_mesh** where the **free_surface_sideset** is approximately orthogonal to the sweep direction. It can especially improve mesh quality when using the **capture=5** option where small geometric features, such as cracks, are encountered near the **free_surface_sideset**. See also the **crack_min_elem_thickness** option, to control how cracks are captured at the free_surface_sideset.

---

#### Minimum element thickness of crack

```
Command: crack_min_elem_thickness     Minimum element thickness in crack

Input file command:   crack_min_elem_thickness <arg>
Command line options: -cmet <arg>
Argument Type:        floating point value
```

Command Description:

Defines the minimum allowed thickness of the elements resolving the side of a crack. Used with the **wear_method = sheet (2)** and **match_sideset** options to potentially improve mesh quality at cracks near the **free_surface_sideset**. Cracks are normally identified using the **match_sideset** and **match_sidesets_nodset** options. The distance from the bottom of the crack to the **free_surface_sideset** is measured to determine the element thickness. If the thickness is below the specified **crack_min_elem_thickness** value, the crack walls are merged together at this location. If a **crack_min_elem_thickness** is not specified, cracks near the **free_surface_sideset** will only be collapsed when the surrounding volume fraction of the sheet drops below 50 percent.

---

#### Minimum number of layers to keep using wear_method=2

```
Command: min_num_layers     Minimum number of layers to keep using
wear_method=2

Input file command:   min_num_layers <arg>
Command line options: -mnl <arg>
Argument Type:        integer >= 0
```

Command Description:

Defines the minimum number of swept layers of the reference mesh to retain when using sculpt with the **wear_method = sheet (2)**. These layers will be retained even if their volume fraction indicates that they should be discarded.

# *Sculpt Input Data Files*

Options for specifying input files to Sculpt. Sculpt uses a method for representing geometry based upon volume fractions defined on a Cartesian or unstructured grid. Sculpt will accept facet-based (STL) or analytic (diatom) geometry, but will first convert the input geometry to the required volume fraction description before generating the hexahedral mesh. Various formats for volume fraction data can also be imported directly into Sculpt and used as the basis for hex meshing. The following formats for geometry are currently supported in Sculpt:

- STL (stl_file)
- Diatom (diatom_file)
- Volume Fractions
    - Exodus element variables (import_vfrac)
    - Exodus blocks (import_cart_exo)
    - Ascii Voxel Data (import_spn)

- Ascii Volume Fraction Data (import_micro)

```
Input Data Files                -inp     --input
  --stl_file                    -stl  <arg> Input STL file
  --diatom_file                 -d    <arg> Input Diatom description file
  --input_vfrac                 -ivf  <arg> Input from Volume Fraction file base
name
  --input_micro                 -ims  <arg> Input from Microstructure file
  --input_cart_exo              -ice  <arg> Input from Cartesian Exodus file
  --input_spn                   -isp  <arg> Input from Microstructure spn file
  --spn_xyz_order               -spo  <arg> Ordering of cells in spn file
  --compress_spn_ids            -csp  <arg> Compress IDs from SPN file
  --input_stitch                -ist  <arg> Input from Stitch file
  --stitch_timestep             -stt  <arg> Timestep in Stitch file to read
  --stitch_timestep_id          -stn  <arg> Timestep ID in Stitch file to read
  --stitch_field                -stf  <arg> Field in Stitch file to read
  --stitch_info                 -sti        List header info for Stitch file
  --lattice                     -l    <arg> STL Lattice Template File
```

Sculpt Command Summary

---

**STL File**

```
Command: stl_file     Input STL file

Input file command:   stl_file <arg>
Command line options: -stl <arg>
Argument Type:        file name with path
```

Command Description:

File name of a single STL (facet geometry) file to be used as input. Either an stl_file or diatom_file designation should be included to run Sculpt. The stl_file option will support a single STL file. To use multiple STL files, where each file represents a different material, use the diatom_file file option where multiple file names may be specified.
It is recommended that STL files used as input to Sculpt be "water-tight". While in many cases non-watertight geometries will be successful, unexpected or incorrect results may result. It is recommended practice to use Cubit to first import the STL geometry and allow the sculpt parallel command to write a new STL geometry file for use in Sculpt. Cubit's sculpt parallel command will attempt to stitch and repair any triangle facets that are not completely closed. Other commercial tools are available for STL geometry that may be effective in repairing the geometry prior to use in Sculpt.

---

**Diatom File**

```
Command: diatom_file     Input Diatom description file

Input file command:   diatom_file <arg>
Command line options: -d <arg>
Argument Type:        file name with path
```

Command Description:

File name of a diatom file to be used as input to Sculpt. Both stl_file and diatom_file cannot be used simultaneously. A diatom file is a constructive solid geometry description containing primitives for generating a full geometric definition of the model. Diatoms are commonly used as input to Sandia's CTH and Alegra codes. Multiple STL files can also be defined in a Diatom file. The following is a simple example of a diatom file that would read 3 different STL files:

```
diatoms
  package 'blue_material'
    material 1
    insert stl
      file = 'blue_part1.stl'
    endinsert
    insert stl
      file = 'blue_part2.stl'
    endinsert
  endpackage
  package 'red_material'
    material 2
    insert stl
      file = 'red_part1.stl'
    endinsert
  endpackage
enddiatom
```

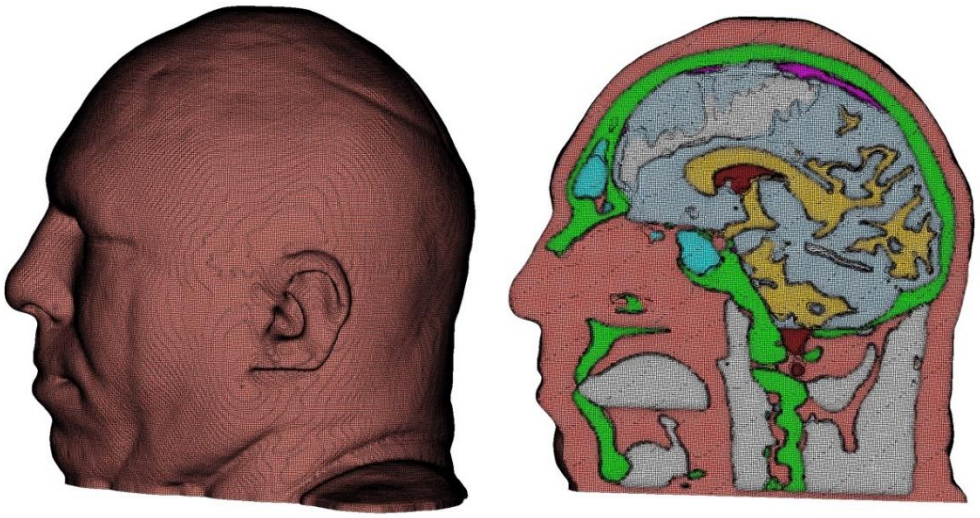
Note that the first two files, blue_part1.stl and blue_part2.stl belong to the same material. As a result, elements generated within the geometry of these files will belong to block 1. Likewise, the elements generated within the geometry of red_part1.stl will belong to block 2.

**Bitmap Files**

The Diatom format will also support bitmap files. These are binary files that set each cell either on or off for the specified material. The following is an example diatom specification for a bitmap file. Note that the bitmap specification includes nx, ny, nz dimensions for the size of the input file.

```
diatoms
  package 'Skull'
    material 1
    insert bitmap
      file = 'skull_bitmap_file'
      nx = 680
      ny = 408
      nz = 236
    endinsert
  endpackage
```

```
enddiatom
```



Example mesh generated with Diatom bitmap option

For a full description of the diatom format see the CTH or Alegra documentation.
D. A. Crawford, A. L. Brundage, E. N. Harstad, K. Ruggirello, R. G. Schmitt, S. C. Schumacher and J. S. Simmons, "CTH User's Manual and Input Instructions, Version 10.3", CTH Development Project, Sandia National Laboratories, Albuquerque, New Mexico 87185, February 14, 2013

---

#### Input Volume Fraction File

```
Command: input_vfrac     Input from Volume Fraction file base name

Input file command:   input_vfrac <arg>
Command line options: -ivf <arg>
Argument Type:        base file name with path
```

Command Description:

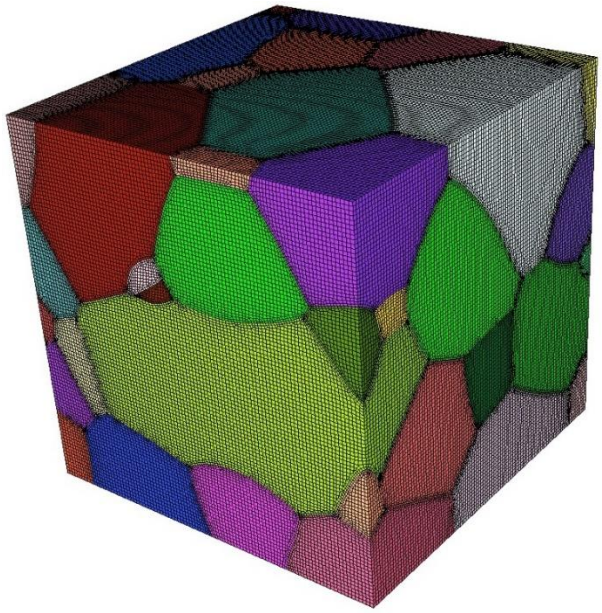
Sculpt can optionally take an exodus file containing volume fraction data stored as element variables. Normally the exodus file has initially been written using the --volfrac_file (-vf) option. Since the exodus file will be a Cartesian grid spread across multiple processors, the base filename for the parallel series of exodus files is used as the argument for this command. The input volume fraction file(s) would be used instead of an STL or diatom file. Since computing volume fractions from geometry can be time consuming, precomputing the volume fractions and reading them from a file can be advantageous if multiple meshes are to be generated from the same volume fraction data.

---

**Input Microstructure File**

```
Command: input_micro     Input from Microstructure file

Input file command:   input_micro <arg>
Command line options: -ims <arg>
Argument Type:        file name with path
```

Command Description:



Example all-hex mesh of microstructure

A microstructure file is an ascii text file containing volume fraction data for each cell of a Cartesian grid. The format for this file includes header information followed by data for each cell. The following is an example:

```
TITLE = triple line system
VARIABLES = x y z, phi_1, phi_2, phi_3
ZONE i = 2 , j = 2 , k = 2
0.0000       0.0000       0.0000      0.5000      0.5000      0.0000
1.0000       0.0000       0.0000      0.3333      0.3333      0.3334
0.0000       1.0000       0.0000      1.0000      0.0000      0.0000
1.0000       1.0000       0.0000      0.0000      1.0000      0.0000
0.0000       0.0000       1.0000      0.2000      0.4000      0.4000
1.0000       0.0000       1.0000      0.6000      0.1000      0.3000
0.0000       1.0000       1.0000      0.0000      0.0000      1.0000
1.0000       1.0000       1.0000      0.9000      0.0000      0.1000
```

The header information should contain the following:
TITLE: any descriptive character string
VARIABLES: a list of variables separated by spaces or commas. It should include x, y, z as the first three variable names. The remaining names are arbitrary. The number of

variable names listed must correspond to the number of data values for each cell of the Cartesian grid.
ZONE: Specify the number of cells in the i, j and k directions (corresponding to x, y, and z respectively)
The body of the file will contain one line per cell of the grid. The first three values correspond to the centroid location of a cell in the grid. The remaining values represent volume fractions for the cell for each variable listed. The sum of the volume fractions for each individual cell should be 1.0
Currently this format assumes that cell sizes are exactly 1.0 x 1.0 x 1.0 and the minimum cell centroid location is always 0.0, 0.0, 0.0. This results in a Cartesian grid with minimum coordinate = (-0.5, -0.5, -0.5) and maximum coordinate = (i-0.5, j-0.5, k-0.5). If a size other than 1x1x1 is required consider using the scale and/or translate options.
Example usage of this command is as follows:

```
  sculpt -j 8 -ims my_micro_file.tec -p 1
```

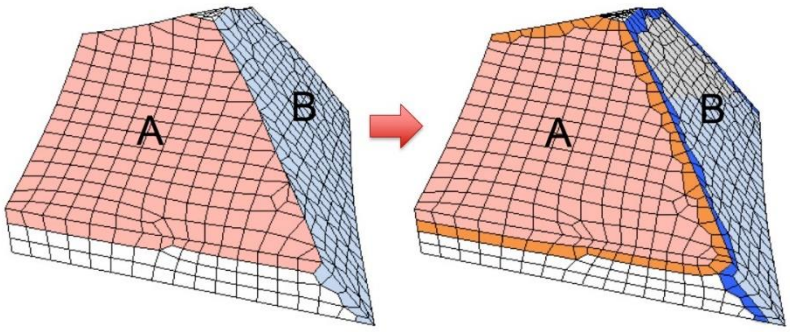
Smoothing: Sculpt will set automatic defaults for smoothing if user options have not been defined. These include:

```
--smooth 9 (surface smoothing option - no surface projection)
--csmooth 2 (curve smoothing option - hermite interpolation)
```

These options will generally provide a smoother curve and surface representation but may not adhere strictly to the volume fraction geometric definition. To over-ride the defaults, consider using the following options:

```
--smooth 8 (surface smoothing option - projection to interpolated surface)
--csmooth 5 (curve smoothing option - projection to interpolated curve)
```

Pillowing: For most 3D models it is recommended using pillowing since triple junctions (curves with at least 3 adjacent materials) will typically be defined where malformed hex elements would otherwise be generated. Surface pillowing (option 1) is usually sufficient to remove poor quality elements at triple junctions.




Pillows (hex layers) inserted at surfaces to improve element quality around curves. Note mesh quality at curve between surfaces A and B.

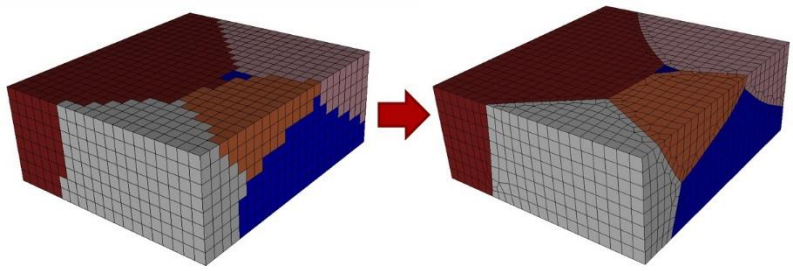
---

**Input Cartesian Exodus File**

```
Command: input_cart_exo     Input from Cartesian Exodus file

Input file command:   input_cart_exo <arg>
Command line options: -ice <arg>
Argument Type:        file name with path
```

Command Description:

An exodus mesh containing a Cartesian grid of elements can also be used as the source of a sculpt mesh. For this option the following conditions must be met:



Example Cartesian Exodus file and the resulting hex mesh.

1. A single (non-parallel) exodus II format file.
2. Contains only hex elements configured as a Cartesian grid.
3. All hex elements must be exactly equilateral cubes.
4. Each hex element has been assigned to exactly one block. (Any number of blocks may be defined in the file)

Provided these conditions are met, sculpt will treat each block as a separate material and generate a smooth conforming mesh between the materials. This option is useful for converting a stair-step mesh into a smooth conforming mesh. The resulting sculpt mesh will have the same dimensions as the original exodus mesh, but will add layers of hexes at material interfaces.
Example usage of this command is as follows:

```
  sculpt -j 8 -ice my_cartesian_file.e -p 1
```

Smoothing: Sculpt will set automatic defaults for smoothing if user options have not been defined. These include

```
--smooth 9 (surface smoothing option - no surface projection)
--csmooth 2 (curve smoothing option - hermite interpolation)
```

These options will generally provide a smoother curve and surface representation but may not adhere strictly to the volume fraction geometric definition. To over-ride the defaults, consider using the following options:

```
--smooth 8 (surface smoothing option - projection to interpolated surface)
--csmooth 5 (curve smoothing option - projection to interpolated curve)
```

Pillowing: For most 3D models it is recommended using pillowing since triple junctions (curves with at least 3 adjacent materials) will typically be defined where malformed hex elements would otherwise be generated. Surface pillowing (option 1) is usually sufficient to remove poor quality elements at triple junctions.

---

#### Input Microstructure SPN File

```
Command: input_spn     Input from Microstructure spn file

Input file command:   input_spn <arg>
Command line options: -isp <arg>
Argument Type:        file name with path
```

Command Description:

A .spn file is an optional method for importing volume fraction data into sculpt for meshing. This format is a simple ascii text file containing one integer per cell of a Cartesian grid. Each integer represents a unique material identifier. Any number of materials may be used, however for practical purposes, the number of unique materials should not exceed more than about 50 for reasonable performance.
An example file containing a 3 x 3 x 3 grid with 2 materials may be defined as follows:

```
1 1 2 1 2 1 1 1 1
1 2 2 1 2 2 1 1 2
2 1 1 1 2 1 1 2 2
```

Any unique integer may be used to identify a material. All cells with the same ID will be defined as a continuous block with the same exodus block ID in the final mesh. All integers should be separated by a space or newline. The number of integers in the file should exactly correspond to the size of the Cartesian grid. The dimensions of the Cartesian grid must be specified on the command line as part of the input. The following is an example:

```
sculpt -j 8 -x 10 -y 24 -z 15 -isp "my_spn_file.spn" -p 1
```

The default order of the cells in the input file will be read according to the following schema:

```
for (i=0; i<nx; i++)
  for (j=0; j<ny; j++)
    for (k=0; k<nz; k++)
       // read next value from file
```

Where nx, ny, nz are the number of cells in each Cartesian direction. This ordering can be changed to nz, ny, nx using the spn_xyz_order option. The initial size of the Cartesian grid will be exactly nx X ny X nz with the minimum coordinate at (0.0, 0.0, 0.0). If a size other than the default is required, consider using the scale and/or translate options.
Smoothing: Sculpt will set automatic defaults for smoothing if user options have not been defined. These include:

```
--smooth 9 (surface smoothing option - no surface projection)
```

```
--csmooth 2 (curve smoothing option - hermite interpolation)
```

These options will generally provide a smoother curve and surface representation but may not adhere strictly to the volume fraction geometric definition. To over-ride the defaults, consider using the following options:

```
--smooth 8 (surface smoothing option - projection to interpolated surface)
--csmooth 5 (curve smoothing option - projection to interpolated curve)
```

Pillowing: For most 3D models it is recommended using pillowing since triple junctions (curves with at least 3 adjacent materials) will typically be defined where malformed hex elements would otherwise be generated. Surface pillowing (option 1) is usually sufficient to remove poor quality elements at triple junctions.

---

#### XYZ ordering of cells in SPN File

```
Command: spn_xyz_order     Ordering of cells in spn file

Input file command:   spn_xyz_order <arg>
Command line options: -spo <arg>
Argument Type:        integer (0 to 5)
Input arguments: xyz (0)
                 xzy (1)
                 yxz (2)
                 yzx (3)
                 zxy (4)
                 zyx (5)
```

Command Description:

This option is valid with the 'input_spn' option. The default order of the cells in the spn input file will be read according to the following schema:

```
    for (i=0; i<nx; i++)
      for (j=0; j<ny; j++)
        for (k=0; k<nz; k++)
           // read next value from file
```

If the spn file has the cells in a different order, use this option to specify the order. 0 (xyz) is the default.

---

#### Compress IDs in SPN file

```
Command: compress_spn_ids     Compress IDs from SPN file

Input file command:   compress_spn_ids <arg>
Command line options: -csp <arg>
Argument Type:        no argument
```

Command Description:

This option is valid with the **input_spn** or **input_stitch** options. The default will use the integers in the spn or stitch file as the final block IDs in the resulting mesh file. Turning this option ON will compress the IDs so that block IDs will start at 1 and be contiguous through the number of materials. If used, check the Sculpt output for a listing of the correspondance between the block IDs and the IDs used in the SPN file.

---

#### Input Stitch File

```
Command: input_stitch     Input from Stitch file

Input file command:   input_stitch <arg>
Command line options: -ist <arg>
Argument Type:        file name with path
```

Command Description:

Stitch is a new I/O system that has been added to Sandia's SPPARKS (Stochastic Parallel PARticle Kinetic Simulator) tool. It was specially developed for additive manufacturing (AM) simulations. Using Stitch, an output database for microstructure simulations is created incrementally by appending lattice sites much in the same way new material is added to an AM part. See the **dump stitch** (https://spparks.github.io/doc/dump.html) and **set stitch** (https://spparks.github.io/doc/set.html) commands in the SPPARKS docs for details.

---

#### Stitch Timestep

```
Command: stitch_timestep     Timestep in Stitch file to read

Input file command:   stitch_timestep <arg>
Command line options: -stt <arg>
Argument Type:        floating point value
Input arguments: first (-10)
                 last (-20)
```

Command Description:

Used with the **input_stitch** option to specify a floating point value timestep to extract from the stitch file. Either a **stitch_timestep** or **stitch_timestep_id** should be used (not both). If neither is specified, the first timestep in the stitch file will be used. Keywords "first" or "last" may also be used in place of a floating value indicating the first or last timestep in the stitch file.

---

#### Stitch Timestep ID

```
Command: stitch_timestep_id     Timestep ID in Stitch file to read

Input file command:   stitch_timestep_id <arg>
Command line options: -stn <arg>
Argument Type:        integer > 0
```

Command Description:

Used with the **input_stitch** option to specify an integer ID representing the timestep to extract from the stitch file. Either a **stitch_timestep_id** or **stitch_timestep** should be used (not both). The **stitch_timestep_id** should be an integer, **N**, representing the **N**th timestep encountered in the stitch file. **stitch_timestep_id = 1** represents the first timestep in the file. If neither **stitch_timestep_id** or **stitch_timestep** is specified, the first timestep encountered in the stitch file will be used.

---

#### Stitch Field

```
Command: stitch_field     Field in Stitch file to read

Input file command:   stitch_field <arg>
Command line options: -stf <arg>
Argument Type:        character string
```

Command Description:

Used with the **input_stitch** option to specify the field to extract from the stitch file. If not specified, the first field in the stitch file will be used.

---

#### List Stitch header info

```
Command: stitch_info     List header info for Stitch file

Input file command:   stitch_info
Command line options: -sti
```

Command Description:

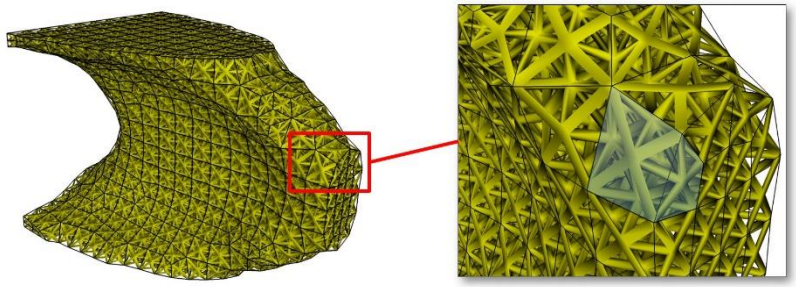
Used with the **input_stitch** option to list the header information for the given stitch file. If **stitch_info** is present (no arguments), Sculpt execution will stop after listing stitch information.

---

#### STL Lattice Template File

```
Command: lattice     STL Lattice Template File

Input file command:   lattice <arg>
Command line options: -l <arg>
Argument Type:        file name with path
```

Command Description:



Lattice geometry generated from exodus mesh.

Generate a lattice structure from a hex mesh. This command takes the name of an STL format template file which defines the lattice over a unit cube. To generate a valid lattice structure, the facets should be symmetric to the three coordinate planes. The lattice structure will be transformed and copied into each hex of the mesh. The result will be an STL file containing lattice geometry for the mesh.
This option currently requires the name of an exodus mesh on which to define the lattice. Use the --exodus_file (-e) option to specify its path. The current implementation is limited to one block, however if a second block is contained in the Exodus file it will be treated as a solid and stl facets will be generated at the skin of the block.
The name of the output STL file may also be defined by using the --stl_file (-stl) option. If no stl file is specified, the output will use the name of the input exodus file with the extension "_lattice.stl" appended.
In addition to the full lattice geometry, an additional file containing only the lattice from the first layer of hexes will be written. This may be useful in reducing the size of the STL file for visualization purposes only. The name of this file will be the name of the full STL geometry file with the extension ".vis.stl" appended.
The following is an example input file using the lattice option:

```
BEGIN SCULPT
  lattice = lattice_template.stl  $contains unit cube with triangles
  exodus_file = file.e $ hex mesh containing one or two element blocks
```

```
    stl_file = file.stl $ name of output stl file
  END SCULPT
```

Note that this option is currently limited to serial execution (-j 1)

# *Sculpt Mesh Type*

Sculpt options for specifying the type of mesh that will be generated. The default mesh type that will be produced from Sculpt is an unstructured all-hex mesh that will attempt to conform as closely as possible to the input geometry. Sculpt will normally generate its mesh on the interior of the input geometry, however with the **mesh_void** option, it can also generate the mesh on the exterior of the geometry, out to the extent of the user-defined Cartesian overlay grid.
In addition to the default hex mesh, other types of meshes may be produced. This includes the stair-step mesh where the cells of the Cartesian grid inside or intersecting the geometry are used directly as the mesh without projections or smoothing. A triangle mesh may also be generated, which can be used as the basis for a facet-based geometry representation. Other methods include the capabilities to generate a hex-dominant mesh with hexes and tets as well as the ability to include degenerate elements.

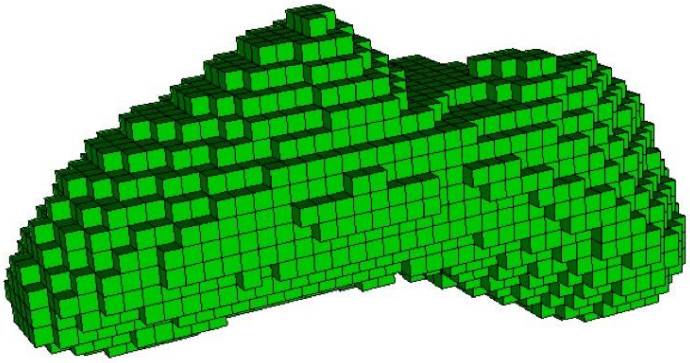
```
Mesh Type                    -typ     --type
  --stair                    -str  <arg> Generate Stair-step mesh
  --mesh_void                -V    <arg> Mesh void
  --trimesh                  -tri       Generate tri mesh of geometry
surfaces
  --tetmesh                  -tet  <arg> Under Development
  --deg_threshold            -dg   <arg> Convert hexes below threshold to
degenerates
  --max_deg_iters            -dgi  <arg> Maximum number of degenerate
iterations
  --htet                     -ht   <arg> Convert hexes below quality
threshold to tets
  --htet_method              -hti  <arg> Method used for splitting hexes to
tets
  --htet_material            -htm  <arg> Convert hexes in given materials to
tets
  --htet_transition          -htt  <arg> Transition method between hexes and
tets
  --htet_pyramid             -htp  <arg> Local transition pyramid
  --htet_tied_contact        -htc  <arg> Local transition tied contact
  --htet_no_interface        -htn  <arg> Local transition none
  --periodic                 -per       Generate periodic mesh
  --check_periodic           -cp   <arg> Check for periodic geometry
  --check_periodic_tol       -cpt  <arg> Tolerance for checking periodicity
  --periodic_axis            -pax  <arg> Axis periodicity is about
  --periodic_nodesets        -pns  <arg> Nodesets ids of master/slave
nodesets
```

Sculpt Command Summary

**Stair**

```
Command: stair     Generate Stair-step mesh

Input file command:   stair <arg>
Command line options: -str <arg>
Argument Type:        integer (0, 1, 2, 3)
Input arguments: none (0)
                 off (0)
                 on (1)
                 full (1)
                 interior (2)
                 fast (3)
```

Command Description:



Example stair-step mesh on STL geometry.

The stair option generates a stair-step mesh where the cells of the Cartesian grid are used in the final mesh without projection or smoothing to the material interfaces. Cells selected from the Cartesian grid to be used in the final mesh will have volume fraction greater than 0.5. Several different options for the stair argument are available:
**off (0):** Stair option is off (default)
**full (1):** Stair-step mesh is generated, but additional processing is done to ensure material interfaces are manifold. This option may add or subtract cells from the basic mesh (where volume fraction > 0.5) to ensure no non-manifold connections between nodes and edges exist in the final mesh.
**interior (2):** The exterior boundary will be smooth while internal material interfaces will be stair-step. This option also ensures manifold connections between elements.
**fast (3):** Generates the final mesh based only on volume fraction criteria. No additional processing is done to ensure manifold connections between edges and nodes.

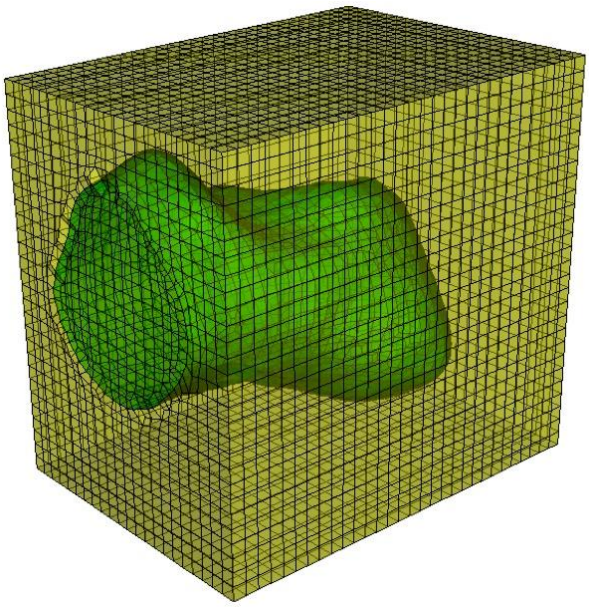
---

**Mesh Void**

```
Command: mesh_void     Mesh void
```

```
Input file command:   mesh_void <arg>
Command line options: -V <arg>
Argument Type:        true/false or only
Input arguments: off (0)
                 false (0)
                 on (1)
                 true (1)
                 only (2)
```

Command Description:



Mesh is generated in the void region surrounding the STL geometry.

The mesh_void accepts the following parameters:
**off (0):** No mesh is generated in the void region
**on (1):** Mesh is generated in the void region
**only (2):** Mesh is generated only in the void region and not in the material
If **mesh_void** option is set to on or only, then the void space surrounding the geometry will be treated as a separate material. Elements will be generated in the void to the extent of the Cartesian grid boundaries. If void_mat option is not used, the material ID of elements in the void region will be the maximum material ID in the model + 1. See also the **separate_void_blocks** option to separate the void elements into contiguous blocks.

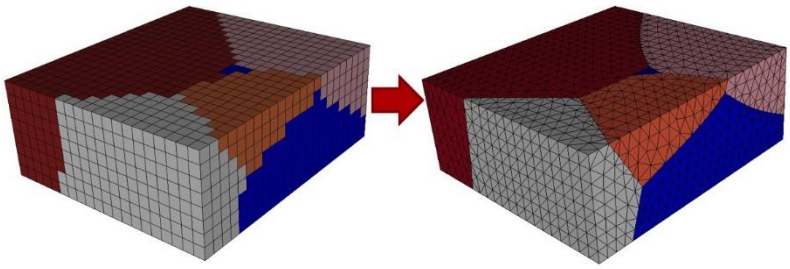
---

**Trimesh**

```
Command: trimesh     Generate tri mesh of geometry surfaces

Input file command:   trimesh
```

```
Command line options: -tri
```

Command Description:



Trimesh generated from voxel microstructure data.

Generate a triangle mesh of the surface geometry. Surface geometry will be defined based on input grid resolution as well as user defined smoothing smoothing parameters. Resulting exodus mesh will contain only TRI elements. All TRI elements will be assigned to the same block in the exodus file.
This option is most often used in conjunction with the --write_geom option used to build a mesh-based geometry in Cubit. Use the following command in Cubit to import a Sculpt trimesh exodus file and s2g file (produced from --write_geom)

```
  import s2g <root filename>
```

See write_geom for more information on s2g files.

---

**Tetmesh**

```
Command: tetmesh     Under Development

Input file command:   tetmesh <arg>
Command line options: -tet <arg>
Argument Type:        none
Input arguments: off (0)
                 on (1)
                 true (1)
                 meshgems (2)
```

Command Description:

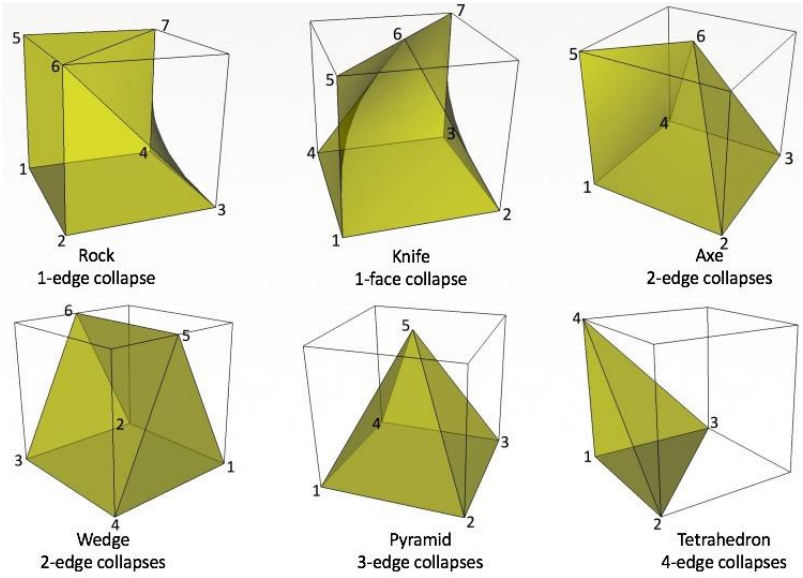
Under Development - uses space-filling tets as base grid. Size and extent is defined by bounding box options.
The **meshgems (2)** option uses a third party tet mesher to place interior tets. Triangle mesh is defined by splitting quads on surface. Both tetmesh options are currently only implemented for serial execution.

---

**Degenerate (Edge Collapse) Threshold**

```
Command: deg_threshold     Convert hexes below threshold to degenerates

Input file command:   deg_threshold <arg>
Command line options: -dg <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:




Examples of degenerates hexes where select edges have been collpased.

Some geometries will not permit a usable mesh with a traditional all-hex mesh. Sculpt includes the option to automatically and selectively collapse element edges to improve low-quality elements. The max_deg_iters and the deg_threshold values are used to control the creation of degenerates. Degenerate elements are treated as standard hex elements, but use repeated nodes in the eight-node connectivity array.
The deg_threshold value indicates scaled Jacobian threshold for edge collapses. Nodes at hexes below this threshold will be candidates for edge collapses, provided doing so will improve the minimum scaled Jacobian at the neighboring hexes. Default is -1.0.

---

**Maxmimum Degenerate Iterations**

```
Command: max_deg_iters     Maximum number of degenerate iterations

Input file command:   max_deg_iters <arg>
Command line options: -dgi <arg>
Argument Type:        integer >= 0
```

Command Description:

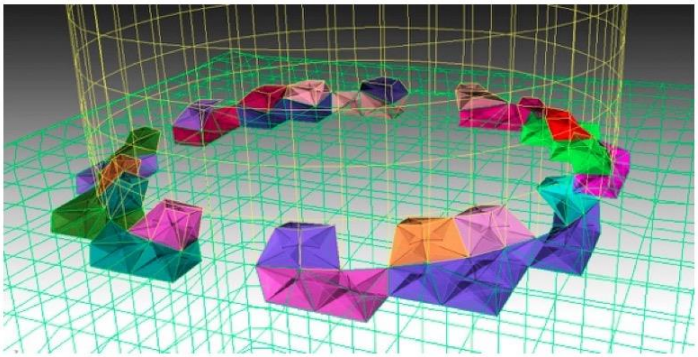
Maximum number of edge collapse iterations to perform to create degenerate hex elements. Default is 0. See also **deg_threshold**

---

**HTet**

```
Command: htet     Convert hexes below quality threshold to tets

Input file command:   htet <arg>
Command line options: -ht <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:



Tet elements generated where quality drops below threshold.

Automatically generate tets in place of poor quality elements. This option can be used to eliminate poor quality hex elements by replacing each hex that falls below the user defined Scaled Jacobian with 6 or 24 tets. The method used for splitting is controlled by the **htet_method** option. The default threshold value for **htet** is -1.0, which turns off the generation of all tets. A value of 1.0 will split all hexes into tets.
If a neighboring element is a hex, and will not be split, one may choose whether to use pyramid transitions or have hanging nodes. The default is to have hanging nodes with a tied contact condition being created. The transition type may be specified with the **htet_transition** command.
If tet blocks are created, their ids will be the material id plus an offset based on the maximum material id. Likewise, any pyramid blocks created will be offset as well, with their ids coming after hex block ids if there are no tets, or with their ids coming after tet blocks.

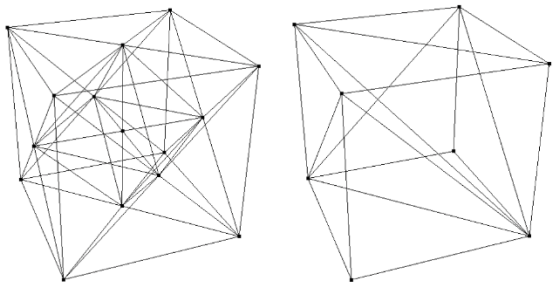
---

**HTet Method**

```
Command: htet_method     Method used for splitting hexes to tets

Input file command:   htet_method <arg>
Command line options: -hti <arg>
Argument Type:        integer (1, 2)
Input arguments: none (0)
                 structured (1)
                 unstructured (2)
```

Command Description:

Specifies which method is used for splitting hexes into tets:
**structured (0)**: Each hex is subdivided into 24 tets. Additional nodes are The 24 tets are formed by inserting one node at the center of each face and one on the interior.
**unstructured (1)**: Each hex is subdivided into 6 tets. No additional nodes are inserted. Note that the unstructured method does not currently support the **htet_transition** options **pyramid** and **tied_contact**.
Default **htet_method** is **structured** (0).



Left: Structured **htet_method** subdivides each hex into 24 tets. Right: Unstructured **htet_method** subdivides each hex into 6 tets

---

#### HTet Material

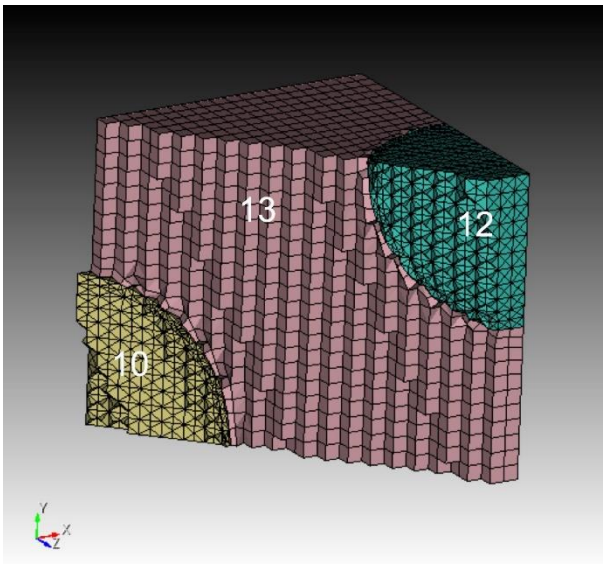
```
Command: htet_material     Convert hexes in given materials to tets

Input file command:   htet_material <arg>
Command line options: -htm <arg>
Argument Type:        integer >= 0
```

Command Description:

Generate tets in place hexes in a given material. This option can be given multiple times to specify multiple materials. Each hex in a material is replaced with 24 tets. The 24 tets are formed by inserting one node at the center of each face and one on the interior.

If an neighboring element is a hex, and will not be split, one may choose whether to use pyramid transitions or have hanging nodes. The default is to have hanging nodes with a tied contact condition being created. The transition type may be specified with the **htet_transition** command.
If tet blocks are created, their ids will be the material id plus an offset based on the maximum material id. Likewise, any pyramid blocks created will be offset as well, with their ids coming after hex block ids if there are no tets, or with their ids coming after tet blocks.

```
htet_material = 10
htet_material = 12
htet_transition = pyramid
htet_no_interface =  10 13
```




Simple example of the use of hybrid tet-hex capability using the above example input. Materials 10 and 12 use tet elements while 13 remains hexes. The default transition is to use pyramids, while the specific interface between 10 and 13 has no interface.

---

**HTet Transition**

```
Command: htet_transition     Transition method between hexes and tets

Input file command:   htet_transition <arg>
Command line options: -htt <arg>
Argument Type:        none/pyramid/tied_contact
Input arguments: none (0)
                 pyramid (1)
                 tied_contact (2)
```

Command Description:

When generating tets adjacent to hexes, the transition type between the two elements can be defined. Possible options are:

- **none (0):** No transition between hex and tet
- **pyramid (1):** Pyramid transition between hex and tet
- **tied_contact (2):** Tied contact condition between hex and tet

If pyramid transition is specified, the hex may be split into 1 pyramids and 20 tets, 2 pyramids and 16 tets, 3 pyramids and 12 tets, and so forth. The mesh will remain conformal if pyramid transition is specified.
A tied contact condition can be defined to ensure continuity of the neighboring tets and hexes. To facilitate this, one additional nodeset and sideset will be generated and output to the exodus file if the gen_sidesets = variable (2) option is specified. The sideset and nodeset will be identified with the following IDs:
Sideset 10000 = the set of hex faces that interface a set of 4 tets.
Nodeset 1000 = the set of nodes at the interface between hexes and tets. One node per face in Sideset 10000 will be included.

---

#### Local HTet Transition Pyramid

```
Command: htet_pyramid     Local transition pyramid

Input file command:   htet_pyramid <arg>
Command line options: -htp <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

When generating tets adjacent to hexes, pyramid transitions can be specified for a given material or material interface. To specify a material interface, two material ids are given to specify pyramid transition between the two materials. To specify multiple materials or multiple material interfaces, this command may be used multiple times.

---

#### Local HTet Transition Tied Contact

```
Command: htet_tied_contact     Local transition tied contact

Input file command:   htet_tied_contact <arg>
Command line options: -htc <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

When generating tets adjacent to hexes, tied contact transitions can be specified for a given material or material interface. To specify a material interface, two material ids are given to specify tied contact transition between the two materials. To specify multiple materials or multiple material interfaces, this command may be used multiple times.

---

#### Local HTet Transition None

```
Command: htet_no_interface     Local transition none

Input file command:   htet_no_interface <arg>
Command line options: -htn <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

When generating tets adjacent to hexes, no transition can be specified for a given material or material interface. To specify a material interface, two material ids are given to specify no transition between the two materials. To specify multiple materials or multiple material interfaces, this command may be used multiple times.

---

#### Generate Periodic Mesh

```
Command: periodic     Generate periodic mesh

Input file command:   periodic
Command line options: -per
```

Command Description:

Generates a periodic mesh for either Cartesian or unstructured mesh input. Ensures that resulting mesh nodes and faces are precisely matching on opposite sides of the mesh.
**Unstructured mesh input:** When used with the **--input_mesh** option opposite sides of the mesh must be identified using pairs of leading and trailing nodesets using the **--periodic_nodesets** (-pns) option. Nodes in the nodeset pairs must be separated by a constant translation or rotation. If a rotation is used between leading and trailing nodesets, the **--periodic_axis** (-pax) option must be used. If not used, then the transformation is assumed to be pure translation. Input geometry is assumed to be periodic with a period equal to that of the input mesh. Results from non-periodic geometry used with the **periodic** option may be unpredictable. The following is an example of an input file that uses the periodic option on an unstructured input mesh:

```
BEGIN SCULPT
   diatom_file = geometry_file.diatom
```

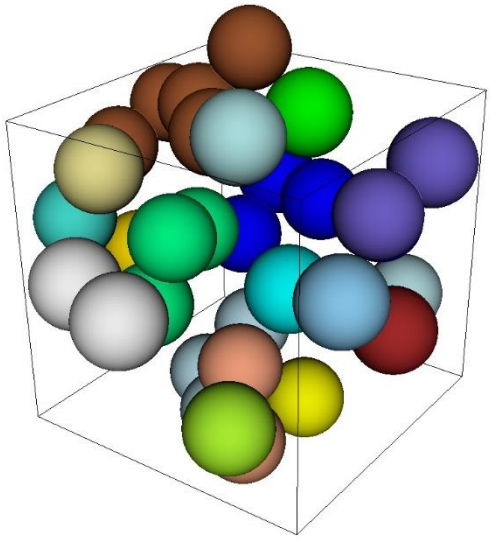
```
   input_mesh = input_exodus_file.g
   exodus_file = output_exodus_file
   smooth = to_geometry
   capture = 5
   capture_angle = 10
   free_surface_sideset = 1000
   gen_sidesets = input_mesh_and_free_surfaces
   periodic = true
   periodic_nodesets = 3224 3225
   periodic_axis = 0 0 0 0 1 0
END SCULPT
```

**Cartesian grid input:** This option is often used for computational materials modeling. Sculpt can generate a true periodic mesh in a representative volume element (RVE) where meshes on all opposite faces of the RVE will precisely match. When used with a Cartesian grid, the **--periodic_nodesets** and **--periodic_axis** options are ignored. The following is an example sculpt input file that utilizes the **--periodic** option on a Cartesian grid with geometry defined in a diatom file. It also utilizes the **--adapt_type** option to automatically refine and the **gen_sidesets = RVE** option to generate sidesets at the six RVE faces.

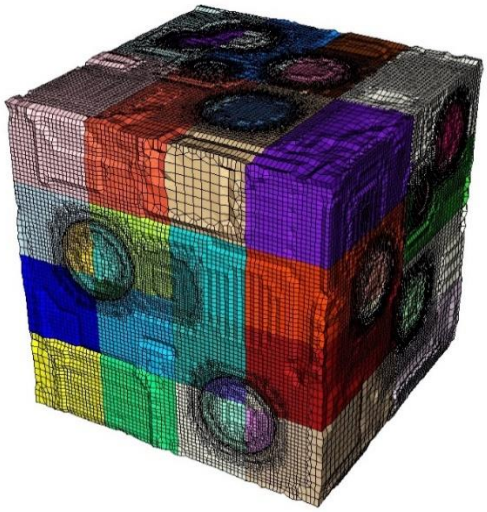
```
BEGIN SCULPT
   diatom_file = spheres_periodic.diatom
   xmin = -18.705510
   ymin = -18.705510
   zmin = -18.705510
   xmax = 18.705510
   ymax = 18.705510
   zmax = 18.705510
   nelx = 38
   nely = 38
   nelz = 38
   periodic = true
   defeature = 1
   min_vol_cells = 10
   adapt_type = vfrac_average
   adapt_levels = 2
   adapt_threshold = 0.00001
   gen_sidesets = RVE
   exodus_file = spheres_periodic
   mesh_void = true
END SCULPT
```

**Geometry Requirements:** In order to generate a valid periodic mesh, the input geometry must also be periodic and the bounding box parameters should span exactly one period of the geometry. To check the periodicity of the geometry and prescribed bounding box, see the **check_periodic** option.

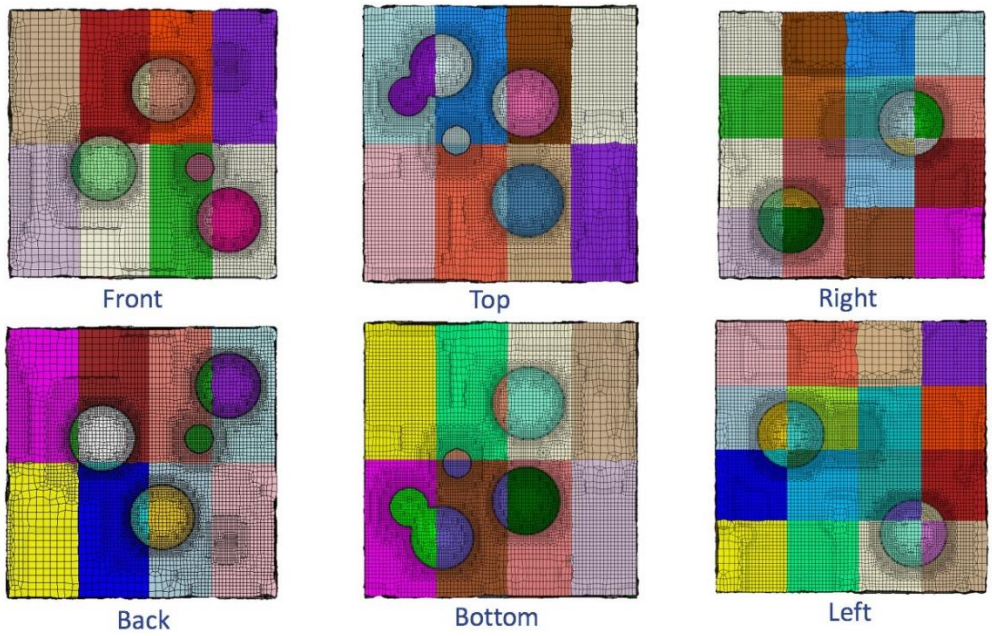
**Note:** The resulting mesh at the boundaries of the Cartesian grid (RVE) will not be projected to the planes of the bounding box. The result will be a "ragged" boundary in order to maintain periodicity between nodes on opposite sides of the mesh. Also note that results from the use of the **periodic** option may be undefined or unstable when used with non-periodic input geometry.



Periodic geometry used for example described in diatom file. RVE boundary shown with respect to the geometry.



Resulting periodic mesh generated from example input.




Six faces of the RVE from above example illustrating periodicity on a 32 processor decomposition. Note that top three images are a mirror image of the bottom three images.

---

**Check for periodic geometry**

```
Command: check_periodic     Check for periodic geometry

Input file command:   check_periodic <arg>
Command line options: -cp <arg>
Argument Type:        on, off, only
Input arguments: off (0)
                 on (1)
                 only (2)
```

Command Description:

When using the **periodic** option with a Cartesian base grid, the input geometry must be periodic with respect to the grid bounding box in order to meet the minimum requirements of a valid periodic mesh. The bounding box must span exactly one period in each dimension. If this requirement is not met, a valid mesh may still be generated, however, periodicity will not be guaranteed. The **check_periodic** option is used to check this requirement. See also **check_periodic_tol** to set the tolerance for checking periodicity.
Options:

- **ON:** The **check_periodic** option is ON by default to ensure periodicity is enforced. Sculpt will fail if the geometry and bounding box do not meet the requirements for periodicity.

- **OFF:** Turning this option OFF will by-pass this check and attempt to generate the mesh even if periodic requirements are not met.
- **ONLY:** The ONLY option will perform a check for periodic requirements and report diagnostics. An exodus file (or files) will be produced with the name "check_periodic.0.0.x.x". A stair-step mesh of the domain will be produced with an additional 2 blocks: 999 and 998. Block 999 shows master cells that are not matched. Block 998 shows paired ghost cells that should have the same volume fraction as those in block 999, but do not. Sculpt will immediately stop execution after producing the "check_periodic.0.0.x.x" mesh. Note that 2 additional layers on all sides of the Cartesian grid will be present in the mesh. These are used internally in Sculpt for parallel ghosting.

The **check_periodic** option is ignored if the **periodic** option is OFF or set to false.

---

#### Tolerance used for periodic check

```
Command: check_periodic_tol     Tolerance for checking periodicity

Input file command:   check_periodic_tol <arg>
Command line options: -cpt <arg>
Argument Type:        floating point value
```

Command Description:

Used on conjunction with the **check_periodic** option. It specifies a tolerance value when checking periodicity. Check periodic option checks the difference between computed volume fractions for cells on the overlay grid that are separated by exactly one period. The periodic tolerance is the allowable volume fraction difference between cells separated by one period. Default value is 1e-6.

---

#### Periodic Mesh Axis

```
Command: periodic_axis     Axis periodicity is about

Input file command:   periodic_axis <arg>
Command line options: -pax <arg>
Argument Type:        six floating point values
```

Command Description:

For an unstructured base grid, specifies an axis about which the nodes in the master (leading) nodesets will be rotated about to produce the slave (trailing) nodesets. Six floating point numbers are specified, the first three define the origin of the axis and the

last three define the axis direction. This option must be used with **--periodic** (-per), **--periodic_nodesets** (-pns), and **--input_mesh** (-im) options. If the **--periodic** (-per) option is used without the --periodic_axis option, the transformation between leading and trailing nodesets is assumed to be pure translation.

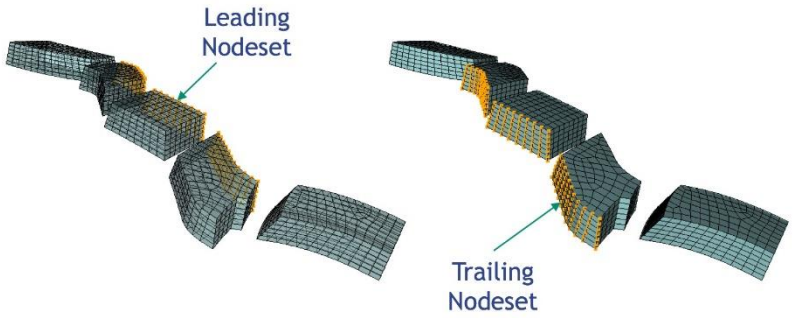
---

#### Periodic Nodeset Ids

```
Command: periodic_nodesets     Nodesets ids of master/slave nodesets

Input file command:   periodic_nodesets <arg>
Command line options: -pns <arg>
Argument Type:        integer(s) >= 0
```

Command Description:

For an unstructured base grid, specifies the master-slave (leading-trailing) nodeset pairs. Master nodesets should be able to be translated or rotated about a specified axis to produce the nodes in the slave nodesets. Nodesets must be specified in pairs, where each master (leading) nodeset corresponds to a single slave (trailing) nodeset. Each nodeset pair must maintain an identical translation or rotation. If a rotation is used, the axis and origin of rotation must be specified with the **--periodic_axis** (-pax) option. This option should be used with **--periodic** (-per), **--periodic_nodesets** (-pns), and **--input_mesh** (-im) options.)




Unstructured input mesh used to generate periodic mesh. Matching leading and trailing nodesets are defined in the exodus file.

# *Sculpt Output*

Sculpt options for specifying output. The primary format for the hex meshes produced from Sculpt is Exodus II. One exodus file will be produced for each processor based upon the -j or num_procs argument. If required, the exodus files can be joined using the epu utility.

Other options for export include the ability to dump the volume fraction representation of the input geometry as well as the ability to write geometry files for use in Cubit.

```
Output                          -out     --output
  --exodus_file                 -e    <arg> Output Exodus file base name
  --large_exodus                -le   <arg> Output large Exodus file(s)
  --volfrac_file                -vf   <arg> Output Volume Fraction file base
name
  --quality                     -Q    <arg> Dump quality metrics to file
  --export_comm_maps            -C          Export parallel comm maps to debug
exo files
  --write_geom                  -G          Write geometry associativity file
  --write_mbg                   -M          Write mesh based geometry file
<beta>
  --compare_volume              -cv         Report vfrac and mesh volume
comparison
  --compute_ss_stats            -css        Report sideset statistics
```

Sculpt Command Summary

---

**Exodus File**

```
Command: exodus_file     Output Exodus file base name

Input file command:   exodus_file <arg>
Command line options: -e <arg>
Argument Type:        character string
```

Command Description:

The base file name of the resulting exodus mesh. Exodus files will be in the form <exodus_file>.e.<nproc>.<iproc>. For example, if the number of processors used is 3 and the exodus_file argument is "model" the following files would be written:

```
model.e.3.0
model.e.3.1
model.e.3.2
```

If no exodus_file argument is used, output files will be in the form <stl_file>_diatom_results.e.<nprocs>.<iproc>. For example, if the number of processors used is 3 and the stl_file (or diatom_file) is "model.stl", the following files would be written:

```
model_diatom_results.e.3.0
model_diatom_results.e.3.1
model_diatom_results.e.3.2
```

A full path may be used when specifying the base exodus file name, otherwise files will be placed in the current working directory. If the exodus_file option is not used, exodus files will be placed in the same directory as the input diatom or stl file.

---

#### Large Exodus Output

```
Command: large_exodus     Output large Exodus file(s)

Input file command:   large_exodus <arg>
Command line options: -le <arg>
Argument Type:        no argument
Input arguments: off (0)
                 false (0)
                 on (1)
                 true (1)
```

Command Description:

Generate output Exodus file(s) to allow IDs greater than 2^31 (2.14 Billion). This option should be used if the intent is to generate billions of elements in parallel on HPC platforms. This option will approximately double the size of output exodus files, so is normally only used for very large parallel applications.

---

#### Volume Fraction File

```
Command: volfrac_file     Output Volume Fraction file base name

Input file command:   volfrac_file <arg>
Command line options: -vf <arg>
Argument Type:        character string
```

Command Description:

Optionally generate exodus files containing a hex mesh of the Cartesian grid containing volume fraction data as element variables. This series of parallel exodus files can later be used as direct input to sculpt using the --input_vfrac (-ivf) command. If not specified, no volume fraction data files will be generated.

---

#### Quality

```
Command: quality     Dump quality metrics to file

Input file command:   quality <arg>
Command line options: -Q <arg>
Argument Type:        file name with path
```

Command Description:

Specify a filename to write mesh quality metrics. If the file already exists, metrics will be appended. Quality metrics and other details of the run will be written to this file. This option is currently off by default.

---

#### Export Communication Maps

```
Command: export_comm_maps     Export parallel comm maps to debug exo files

Input file command:   export_comm_maps
Command line options: -C
```

Command Description:

Used for debugging and verification. Exodus files of the mesh containing the communication nodes and faces at processor boundaries will be written as nodes and side sets. This provides a way to visually check the validity of the parallel communication maps.

---

#### Write S2G Geometry File

```
Command: write_geom     Write geometry associativity file

Input file command:   write_geom
Command line options: -G
```

Command Description:

An s2g (Sculpt to Geometry) file, with the pattern <fileroot>.s2g, will be produced when this argument is used where fileroot is the string specified by the --exodus_file or -e option. An s2g file includes geometry associativity for the exodus file that is written. If used with Cubit's "import s2g <fileroot>" a mesh-based geometry will be generated in Cubit with geometric entities prescribed by Sculpt through the s2g file.
When used with the --trimesh option, the s2g file can provide information to Cubit to build a set of mesh-based geometry volumes where only the surfaces are meshed. This is useful for using the tet meshing capabilities in Cubit to mesh the discrete geometry that was generated in Sculpt. For example, a tet mesh may be constructed from microstructures spn data (see import_spn) with the following workflow:

1. Run Sculpt to generate an exodus and s2g file. An example input file may look like the following:

```
begin sculpt
  import_spn = myfile.spn
  trimesh = true
  write_geom = true
  pillow = 1
end sculpt
```

2. Import the file into Cubit to generate a mesh based geometry:

```
import s2g myfile
```

3. Delete the triangle mesh, set sizes and mesh:

```
delete mesh
vol all scheme tetmesh
vol all size 2.0
mesh vol all
```

Note that the write_geom and trimesh options are still in development and will currently only work with a single processor (-j 1).

---

#### Write Mesh Based Geometry

```
Command: write_mbg     Write mesh based geometry file <beta>

Input file command:   write_mbg
Command line options: -M
```

Command Description:

An MBG (Mesh Based Geometry) file will be produced when this argument is used with the pattern <fileroot>.mbg, where fileroot is the string specified by the --exodus_file or -e option. An MBG file includes the surface and topology definition defined by sculpt as a result of the interface reconstruction process. It will correspond to the boundary of the 3D elements that are generated in the exodus file, or the surface elements generated with the --trimesh option.
An MBG file can be be imported into Cubit using the following Cubit command line options:

```
import mbg "<fileroot>.mbg"
```

---

#### Report VFrac to Mesh Volume Comparison

```
Command: compare_volume     Report vfrac and mesh volume comparison

Input file command:   compare_volume
Command line options: -cv
```

Command Description:

A report will be generated and printed to the terminal following the mesh summary that compares the input volume fraction of the geometry with that of the final finite element mesh. If a volume fraction format is not used as input, the volume fractions will be computed on the refined base grid and used as comparison. Note that exact geometric volumes of the STL or analytic geometry are not used for comparison, rather the volume fraction approximation of the geometry on the refined Cartesian grid.

```
======================================== VOLUME COMPARISON ========================================
Block ID        Num Elems       Sum VFrac       Elem Vol        Diff            Percent Err     VFrac
1               1460156         863.819         868.16           4.34073        0.502504        0.159767
2               9171            3.095           2.90369         -0.191313       6.18135         0.000534363
3               148952          53.268          50.0251         -3.24293        6.08796         0.00920606
4               194233          117.063         115.3           -1.76334        1.50632         0.0212185
5               553             0.089           0.0697202       -0.0192798      21.6627         1.28305e-05
6               95367           37.029          37.0632          0.0341906      0.0923346       0.0068207
7               545329          176.477         162.542         -13.9348        7.89612         0.0299125
8               847440          493.112         491.576         -1.53645        0.311582        0.0904642
9               1397057         656.51          668.967          12.4568        1.89743         0.123109
10              1305491         912.571         912.232         -0.339128       0.0371618       0.167877
11              2947876         2121.99         2125.09          3.1019         0.146179        0.391078
------------------------------------------------------------------------------------------------------
Total           8951625         5435.02         5433.93          1.09365        0.0201223       1
```

Example output from the **compare_volume** command.

The following is a brief description of each column:

- **Block ID:** ID of material/block
- **Num Elems:** Number of hex elements assigned to block in final mesh
- **Sum VFrac:** Sum of input volume fraction for block. For STL or diatom geometry, approximates the volume fraction. For 3D image data (ie. bitmap, input_spn) sums the exact volume fraction input.
- **Elem Vol:** Sum of final mesh volume for block
- **Diff:** Absolute difference between input and output volume fractions for block. (Elem Vol - Sum VFrac)
- **Percent Err:** Percent error represented by Difference between input and output volume fractions for block
- **VFrac:** Total volume fraction represented by Elem Vol. for block. VFrac volume should sum to 1.0.

---

#### Report Sideset Statistics

```
Command: compute_ss_stats     Report sideset statistics

Input file command:   compute_ss_stats
```

```
Command line options: -css
```

Command Description:

A report will be generated and printed to the terminal following the mesh summary that displays the statitics for the surface areas of the sidesets in the final FEA mesh. This is often useful for microstructures use cases where it may be important to know the interface area between materials. This should be used with the **gen_sidesets** option to be effective.

# *Sculpt Process Control*

Options for controlling the execution of Sculpt. Sculpt is a parallel application that uses MPI to distribute and build the hex mesh on multiple processors. The -j or num_procs option is normally used to specify the number of processors to use. Sculpt will write a separate exodus file for each processor, which can be joined into a single file using the epu utility. While any number of processors may be used, you would normally use a -j value less than or equal to the number of cores available on your hardware.
Sculpt options can be specified directly from the command line using the "short" commands, or from an input file where the longer forms of the commands are used. Since an input file can be commented and modified, it is generally the recommended method for running Sculpt.

```
Process Control                 -pc     --process
  --num_procs                   -j    <arg> Number of processors requested
  --input_file                  -i    <arg> File containing user input data
  --debug_processor             -D    <arg> Sleep to attach to processor for
debug
  --debug_flag                  -dbf  <arg> Dump debug info based on flag
  --quiet                       -qt         Suppress output
  --print_input                 -pi         Print input values and defaults then
stop
  --version                     -vs         Print version number and exit
  --threads_process             -tpp  <arg> Number of threads per process
  --iproc                       -ip   <arg> Number of processors in I direction
  --jproc                       -jp   <arg> Number of processors in J direction
  --kproc                       -kp   <arg> Number of processors in K direction
  --build_ghosts                -bg         Write ghost layers to exodus files
for debug
  --vfrac_method                -vm   <arg> Set method for computing volume
fractions
```

Sculpt Command Summary

---

**Number of Processors**

```
Command: num_procs     Number of processors requested

Input file command:   num_procs <arg>
Command line options: -j <arg>
Argument Type:        integer > 0
```

Command Description:

The number of processors that Sculpt will use to generate the mesh. For a structured, Cartesian base grid, the domain will be automatically divided into roughly equal sized rectangular regions based on this value.
For unstructured input (see **--input_mesh**), to utilize more than one processor, the base mesh must first be docomposed into the same number of regions specified by **num_procs**. The **decomp** tool, part of the Sandia, **SEACAS** tool suite can be used to break up an exodus mesh into multiple regions suitable for sculpt input.
An independent mesh of a portion of the domain is generated on each processor. Continuity across processor boundaries is maintained with MPI (Message Passing Interface). Each processor will write a separate Exodus II file to disk containing its portion of the domain. The Sandia SEACAS tool, **epu** can be used to join parallel files into a single file if desired.
If not specified on the command line, the number of processors used will be 1.
For additional control on the arrangement of processor domains on a Cartesian base grid, see arguments **iproc, jproc, kproc.**

---

#### Input File

```
Command: input_file     File containing user input data

Input file command:   input_file <arg>
Command line options: -i <arg>
Argument Type:        file name with path
```

Command Description:

Rather than specifying a complicated series of arguments on the command line, an input file may also be used. An input file is a simple text file containing all arguments and parameters to be used in the current sculpt run. Input files are normally expected to have a ".i" extension. Arguments used in the input file are limited to the Long Names indicated for each command.
User comments can also be made anywhere in the file but must follow a "$" sign. The argument assignments that are intended to be read must be contained within a "begin sculpt" and "end sculpt" block. All arguments may use upper or lower case and can optionally use "=" between the command and its parameter. The following is an example input file:

```
  BEGIN SCULPT
```

```
    stl_file = "mygeom.stl"
    cell_size = 0.5
    exodus_file = "mymesh"
    mesh_void = true
  END SCULPT
```

The following is an example of using an input file with sculpt:

```
  sculpt -j 4 -i myinput.i
```

Note that the number of processors (-j) should always be used on the command line and cannot be included in the input file. Relative or absolute paths for files may also be used.

---

#### Debug Processor

```
Command: debug_processor     Sleep to attach to processor for debug

Input file command:   debug_processor <arg>
Command line options: -D <arg>
Argument Type:        integer >= 0
```

Command Description:

Used for debugging. All processes will sleep until the designated process is attached to a debugger. Note: value of 0 corresponds to first processor, 1 to second, etc.

---

#### Debug Flag

```
Command: debug_flag     Dump debug info based on flag

Input file command:   debug_flag <arg>
Command line options: -dbf <arg>
Argument Type:        integer >= 0
Input arguments: off (0)
                 lost_nodes (1)
                 non-manifold (2)
                 defeature (3)
                 thickening (4)
                 initial-projections (5)
                 reversal (6)
                 gq-color (8)
                 gq-full (9)
                 hostname (10)
                 test_large_global_ids (11)
                 use_bbox_for_unstructured (12)
                 allow_input_mesh_sidesets (13)
```

Command Description:

Used for debugging. Set flag to dump specific info based on the following:
**0** (off) Default, No debug output
**1** (lost_nodes) Dump processor lost node info
**2** (non-manifold) Export Non-manifold resolution state as exodus file after each inner and outer iteration.
**3** (defeature) Export Defeature state as exodus file after each inner and outer iteration.
**4** (thickening) Export the Thickened st:wqate as exodus file after each material has been thickened.
Guaranteed Quality:
**5** (initial-projections) Turn off initial minimizer projection.
**6** (reversal) Use Non-manifold reversal case
**7** Combine debug_flag 5 and 6
**8** (gq-color) Use guaranteed quality laplacian color smoothing
**9** (gq-full) Combine debug_flags 5,6 and 8
**10** (hostname) Display the host name for each processor
**11** (test_large_global_IDs) starts the global node and element ID numbering at INT_MAX (2^31) for testing large_exodus option.
**12** (use_bbox_for_unstructured) Use the xmin, ymin, ... options to limit the domain on an input_mesh
**13** (allow_input_mesh_sidesets) Don't restrict use of gen_sidesets = 6, 7 or 9 if using parallel. (This may result in poor elements at proc boundaries).

---

#### Quiet

```
Command: quiet     Suppress output

Input file command:   quiet
Command line options: -qt
```

Command Description:

Suppress any output to the command line from Sculpt as it is running.

---

#### Print Input

```
Command: print_input     Print input values and defaults then stop

Input file command:   print_input
Command line options: -pi
```

Command Description:

Display all input parameters and defaults used in the current Sculpt run to the output window and then stop. No mesh (or volume fractions) will be generated.

---

#### Version

```
Command: version     Print version number and exit

Input file command:   version
Command line options: -vs
```

Command Description:

Prints Sculpt version information and exits.

---

#### Threads Per Processor

```
Command: threads_process     Number of threads per process

Input file command:   threads_process <arg>
Command line options: -tpp <arg>
Argument Type:        integer > 0
```

Command Description:

This option is currently experimental and under development. Sculpt may use shared memory parallelism to improve performance. When built with the Kokkos library, some algorithms in sculpt will use shared memory parallel threads in addition to MPI distributed memory parallelism (MPI+X). Currently this option is implemented only for surface and volume Laplacian smoothing algorithms. This option may not be available requiring a custom build of sculpt to be used. Check with developers if you would like to use this option.

---

#### Number of processors in I

```
Command: iproc     Number of processors in I direction

Input file command:   iproc <arg>
Command line options: -ip <arg>
Argument Type:        integer > 0
```

Command Description:

Arguments iproc, jproc and kproc provide user control over the processor decomposition in I, J, and K directions respectively. iproc * jproc * kproc must equal the number of processors specified on the command line using the -j option. In some cases, where it is known that significant refinement will be done in a localized region of the domain, it may be worth manually specifying the number of cell layers, or intervals that each processor will contain. This should provide rough processor load balacing so that regions with an expected large number of elements will have a smaller physical domain, but roughly similar element counts. This may avoid single processor memory limitations on large HPC machines. To specify processor intervals, optionally, the exact interval count in I, J, K directions should be specified. The number of intervals specfied for each direction should be the value of **iproc**, **jproc**, and **kproc** respectively. In addition, the sum of the intervals in each direction should equal the user specified **nelx**, **nely** and **nelz** values respectively. For example, for a 24 processor (-j 24) arrangement with nelx = nely = 28 and nelz = 27, one possible arrangment of processors would be as follows:

```
nelx = 28
nely = 28
nelz = 27
iproc = 2 14 14
jproc = 2 14 14
kproc = 6 7 6 5 4 3 2
```

For this example the I and J directions are equally subdivided into 2 processors with 14 intervals in both directions. The K direction, however is subdivided into 6 processors, where the intervals are progressively thinner, starting with 7 intervals at the bottom (smallest z) and ending with 2 intervals at the top (largest z). Note that the number of intervals in any direction must always be at least 2 or greater. If processor intervals are omitted, Sculpt will attempt to roughly equally space the intervals in each direction. If at least one direction of intervals is detected in the input, then all directions should be specified. Note that this option is only available when using a Cartesian base grid specification and cannot be used with an unstructured base grid using the **input_mesh** option.

---

#### Number of processors in J

```
Command: jproc     Number of processors in J direction

Input file command:   jproc <arg>
Command line options: -jp <arg>
Argument Type:        integer > 0
```

Command Description:

Arguments iproc, jproc and kproc provide user control over the processor decomposition in I, J, and K directions respectively. iproc * jproc * kproc must equal the number of processors specified on the command line using the -j option. In some cases, where it is

known that significant refinement will be done in a localized region of the domain, it may be worth manually specifying the number of cell layers, or intervals that each processor will contain. This should provide rough processor load balacing so that regions with an expected large number of elements will have a smaller physical domain, but roughly similar element counts. This may avoid single processor memory limitations on large HPC machines. To specify processor intervals, optionally, the exact interval count in I, J, K directions should be specified. The number of intervals specfied for each direction should be the value of **iproc**, **jproc**, and **kproc** respectively. In addition, the sum of the intervals in each direction should equal the user specified **nelx**, **nely** and **nelz** values respectively. For example, for a 24 processor (-j 24) arrangement with nelx = nely = 28 and nelz = 27, one possible arrangment of processors would be as follows:

```
nelx = 28
nely = 28
nelz = 27
iproc = 2 14 14
jproc = 2 14 14
kproc = 6 7 6 5 4 3 2
```

For this example the I and J directions are equally subdivided into 2 processors with 14 intervals in both directions. The K direction, however is subdivided into 6 processors, where the intervals are progressively thinner, starting with 7 intervals at the bottom (smallest z) and ending with 2 intervals at the top (largest z). Note that the number of intervals in any direction must always be at least 2 or greater. If processor intervals are omitted, Sculpt will attempt to roughly equally space the intervals in each direction. If at least one direction of intervals is detected in the input, then all directions should be specified. Note that this option is only available when using a Cartesian base grid specification and cannot be used with an unstructured base grid using the **input_mesh** option.

---

#### Number of processors in K

```
Command: kproc     Number of processors in K direction

Input file command:   kproc <arg>
Command line options: -kp <arg>
Argument Type:        integer > 0
```

Command Description:

Arguments iproc, jproc and kproc provide user control over the processor decomposition in I, J, and K directions respectively. iproc * jproc * kproc must equal the number of processors specified on the command line using the -j option. In some cases, where it is known that significant refinement will be done in a localized region of the domain, it may be worth manually specifying the number of cell layers, or intervals that each processor will contain. This should provide rough processor load balacing so that regions with an

expected large number of elements will have a smaller physical domain, but roughly similar element counts. This may avoid single processor memory limitations on large HPC machines. To specify processor intervals, optionally, the exact interval count in I, J, K directions should be specified. The number of intervals specfied for each direction should be the value of **iproc**, **jproc**, and **kproc** respectively. In addition, the sum of the intervals in each direction should equal the user specified **nelx**, **nely** and **nelz** values respectively. For example, for a 24 processor (-j 24) arrangement with nelx = nely = 28 and nelz = 27, one possible arrangment of processors would be as follows:

```
nelx = 28
nely = 28
nelz = 27
iproc = 2 14 14
jproc = 2 14 14
kproc = 6 7 6 5 4 3 2
```

For this example the I and J directions are equally subdivided into 2 processors with 14 intervals in both directions. The K direction, however is subdivided into 6 processors, where the intervals are progressively thinner, starting with 7 intervals at the bottom (smallest z) and ending with 2 intervals at the top (largest z). Note that the number of intervals in any direction must always be at least 2 or greater. If processor intervals are omitted, Sculpt will attempt to roughly equally space the intervals in each direction. If at least one direction of intervals is detected in the input, then all directions should be specified. Note that this option is only available when using a Cartesian base grid specification and cannot be used with an unstructured base grid using the **input_mesh** option.

---

#### Write the ghost layers for debug

```
Command: build_ghosts     Write ghost layers to exodus files for debug

Input file command:   build_ghosts
Command line options: -bg
```

Command Description:

If set, this option will dump the ghost hexes at the boundaries of processor domains to the exodus files. This is used only for debugging.

---

#### Volume Fraction Calculation Method

```
Command: vfrac_method     Set method for computing volume fractions

Input file command:   vfrac_method <arg>
```

```
Command line options: -vm <arg>
Argument Type:        integer (1, 2, 3)
Input arguments: cth (0)
                 cth (1)
                 r3d (2)
                 winding (3)
```

Command Description:

Sets the method used for computing volume fractions from geometry input. Three options are currently available:
**CTH (1):** The default method. It uses the CTH third party library from Sandia Laboratories for approximating intersections using an adaptive ray firing method to determine inside-outside status of multiple locations within a grid cell. This method can be used with STL and all valid primitive types defined by the diatom format.
**R3D (2):** Uses the R3D third party library developed by Los Alamos Laboratories. Machine precision intersection calculations are performed to generate accurate volume fractions from the STL description. This method is valid for STL and diatom input packages specifying STL input files. Non STL format geometry defined in the diatom file will be ignored for this format.
**Winding (3):** Uses the fast winding number from the igl third party library, which should gracefully handle non-watertight stl files. R&D is in progress.

# *Sculpt Overlay Grid Specification*

Sculpt options for setting up the overlay grid. Sculpt is an overlay-grid method that requires a base mesh that it will modify to generate the final mesh. The base mesh can be in the form of a Cartesian grid, but can also be any general unstructured hexahedral mesh defined in an exodus file (see the **input_mesh** option). Pamgen can also be used to generate an unstructured base mesh (see **input_mesh_pamgen**).
When an overlay Cartesian grid is used as the basis for the all-hex mesh that will be produced, the bounds and size of the cells defining the grid must be specified. The Cartesian grid can be defined in one of two ways:

1. Define the bounding box and number of intervals in each coordinate direction. (**xmin, ymin, zmin, xmax, ymax, zmax, nelx, nely, nelz**)
2. Define a **cell_size**. Sculpt will then automatically define the Cartesian grid coordinates and intervals by evaluating the bounding box of the input geometry and adding a small number of cells in each coordinate direction.

Other options for setting up the Cartesian base grid include **align** and **expand** which are normally used with the second method. The **align** option will automatically rotate the grid to best match the characteristic direction of the geometry rather than maintaining alignment with the global Cartesian directions. The **expand** option over-rides the default expansion of the Cartesian grid beyond the bounding box of the geometry and allow the user to specify a specific expansion percentage.

```
Overlay Grid Specification   -ovr     --overlay
  --nelx                     -x    <arg> Num cells in X in overlay Cartesian
grid
  --nely                     -y    <arg> Num cells in Y in overlay Cartesian
grid
  --nelz                     -z    <arg> Num cells in Z in overlay Cartesian
grid
  --xmin                     -t    <arg> Min X coord of overlay Cartesian
grid
  --ymin                     -u    <arg> Min Y coord of overlay Cartesian
grid
  --zmin                     -v    <arg> Min Z coord of overlay Cartesian
grid
  --xmax                     -q    <arg> Max X coord of overlay Cartesian
grid
  --ymax                     -r    <arg> Max Y coord of overlay Cartesian
grid
  --zmax                     -s    <arg> Max Z coord of overlay Cartesian
grid
  --cell_size                -cs   <arg> Cell size (nelx, nely, nelz ignored)
  --align                    -a          Automatically align geometry to grid
  --bbox_expand              -be   <arg> Expand tight bbox by percent
  --input_mesh               -im   <arg> Input Base Exodus mesh
  --input_mesh_blocks        -imb  <arg> Block ids of Input Base Exodus mesh
  --input_mesh_material      -imm  <arg> Material definition with input mesh
  --input_mesh_pamgen        -imp  <arg> Input Base mesh defined by Pamgen
  --join_parallel            -jp   <arg> Join parallel files
```

Sculpt Command Summary

---

#### Number of Intervals X

```
Command: nelx     Num cells in X in overlay Cartesian grid

Input file command:   nelx <arg>
Command line options: -x <arg>
Argument Type:        integer > 0
```

Command Description:

Defines the number of intervals in the x direction of the base Cartesian grid used for defining the volume fraction definition and meshing For best results the intervals specified should result in approximately equilateral cells.
See also nely, nelz

---

#### Number of Intervals Y

```
Command: nely     Num cells in Y in overlay Cartesian grid

Input file command:   nely <arg>
Command line options: -y <arg>
Argument Type:        integer > 0
```

Command Description:

Defines the number of intervals in the y direction of the base Cartesian grid used for defining the volume fraction definition and meshing For best results the intervals specified should result in approximately equilateral cells.
See also nelx, nelz

---

#### Number of Intervals Z

```
Command: nelz     Num cells in Z in overlay Cartesian grid

Input file command:   nelz <arg>
Command line options: -z <arg>
Argument Type:        integer > 0
```

Command Description:

Defines the number of intervals in the z direction of the base Cartesian grid used for defining the volume fraction definition and meshing For best results the intervals specified should result in approximately equilateral cells.
See also nelx, nely

---

#### Xmin Bounding Box Range

```
Command: xmin     Min X coord of overlay Cartesian grid

Input file command:   xmin <arg>
Command line options: -t <arg>
Argument Type:        floating point value
```

Command Description:

Defines the minimum x coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also ymin, zmin, xmax, ymax, zmax.

---

#### Ymin Bounding Box Range

```
Command: ymin     Min Y coord of overlay Cartesian grid

Input file command:   ymin <arg>
Command line options: -u <arg>
Argument Type:        floating point value
```

Command Description:

Defines the minimum y coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also xmin, zmin, xmax, ymax, zmax.

---

#### Zmin Bounding Box Range

```
Command: zmin     Min Z coord of overlay Cartesian grid

Input file command:   zmin <arg>
Command line options: -v <arg>
Argument Type:        floating point value
```

Command Description:

Defines the minimum z coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also xmin, ymin, xmax, ymax, zmax.

---

#### Xmax Bounding Box Range

```
Command: xmax     Max X coord of overlay Cartesian grid

Input file command:   xmax <arg>
Command line options: -q <arg>
Argument Type:        floating point value
```

Command Description:

Defines the maximum x coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also xmin, ymin, zmin, ymax, zmax.

---

#### Ymax Bounding Box Range

```
Command: ymax     Max Y coord of overlay Cartesian grid

Input file command:   ymax <arg>
Command line options: -r <arg>
Argument Type:        floating point value
```

Command Description:

Defines the maximum y coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also xmin, ymin, zmin, xmax, zmax.

---

#### Zmax Bounding Box Range

```
Command: zmax     Max Z coord of overlay Cartesian grid

Input file command:   zmax <arg>
Command line options: -s <arg>
Argument Type:        floating point value
```

Command Description:

Defines the maximum z coordinate of the bounding box or range of the Cartesian mesh to be used for meshing.
See also xmin, ymin, zmin, xmax, ymax.

---

#### Cell Size

```
Command: cell_size     Cell size (nelx, nely, nelz ignored)

Input file command:   cell_size <arg>
Command line options: -cs <arg>
Argument Type:        floating point value
```

Command Description:

Defines a target edge size for the cells of the base Cartesian grid. Both interval and cell_size can not be specified simultaneously. If cell_size is used without a range specification, a bounding box of the geometry will be computed and used as the default range

---

#### Align

```
Command: align     Automatically align geometry to grid

Input file command:   align
Command line options: -a
```

Command Description:

The align option will attempt to orient the Cartesian grid with the main dimensions of the geometry. This is done by defining a tight bounding box around the geometry using an optimization procedure where the objective is to minimize the difference in volume between an enclosing box and the geometry. Using the align command will override any bounding box parameters previously entered and will build an "aligned" bounding box around the full geometry. It is currently only implemented for STL geometry and will ignore any other diatom definitions. Note that this option will also write temporary stl and diatom files to the working directory.

---

#### Bounding Box Expansion Factor

```
Command: bbox_expand     Expand tight bbox by percent

Input file command:   bbox_expand <arg>
Command line options: -be <arg>
Argument Type:        floating point value
```

Command Description:

Sculpt will measure a tight bounding box of the input model and expand the box by the specified percentage in x, y and z. Input value can be any positive or negative floating point value where 1.0 represents 100 percent expansion. If not specified, the default will add about 2.5 cell widths to the bounding box on each side. This option should be used with the cell_size option. It will be ignored if a specific bounding box has been defined (ie. xmin, ymin, etc...).

---

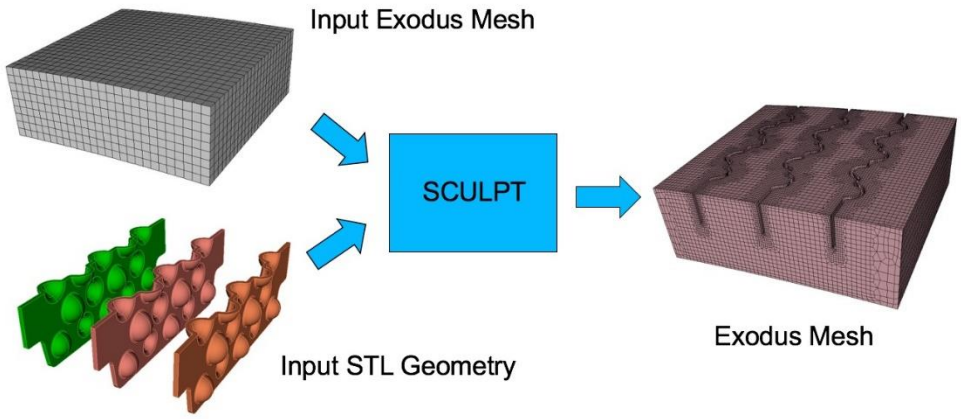
#### Input Base Exodus Mesh

```
Command: input_mesh     Input Base Exodus mesh

Input file command:   input_mesh <arg>
Command line options: -im <arg>
```

```
Argument Type:        file name with path
```

Command Description:

Option to import an Exodus file to use as the base mesh for Sculpt. Sculpt's meshing procedure requires a base mesh from which geometry is recovered and captured. The default base mesh is a Cartesian grid that is defined by specifying a bounding box and intervals. The **input_mesh** option permits a general hexahedral mesh to be used as the base mesh instead of a Cartesian grid. This option currently supports a serial and parallel Exodus files containing HEX8 elements with any number of blocks.




An exodus file is used as the base mesh for Sculpt and STL files describe the geometry to be sculpted.

The **input_mesh** option can also be used in parallel. Sculpt currently requires the mesh to be decomposed prior to running sculpt. The SEACAS **decomp** tool can be used to pre-process any exodus mesh to break it into multiple meshes ready for use in sculpt. SEACAS is an open source library available on github. For example, when using four processors with sculpt, you would use the following command:

```
decomp -p 4 simple-mesh.g
```

The result would be the four meshes:

```
simple-mesh.g.4.0
simple-mesh.g.4.1
simple-mesh.g.4.2
simple-mesh.g.4.3
```

Once the base mesh has been decomposed, Sculpt can be run. In this case, the **input_mesh** option would use the root **simple-mesh.g** as the argument.

```
input_mesh = simple-mesh.g
```

If the **-j 4** option is used, sculpt will look for 4 meshes in the current working directory with the appropriate root and extension.

Four different options are supported for describing the geometry when using the **input_mesh** option:

- **stl_file:** A single file containing a water-tight faceted description of the geometry. Note that only the portion of the STL file completely contained within the base mesh will be represented in the final mesh.
- **diatom_file:** May contain analytic descriptions of geometric primitives and/or references to multiple STL files.
- **input_spn:** The materials of the cells in the spn file are mapped onto the elements of the input mesh using inverse distance-weighted interpolation. As with the stl and diatom files, only the portion of the spn file completely contained within the base mesh will be represented in the final mesh. The **input_mesh_blocks** option can be used in conjunction with the spn_file option to limit the scope of the mapping of material from the spn file to the mesh file. If this options is used, only elements in the specified blocks will get mapped to. For more details, see the **input_mesh_blocks** option.
- **Element Variables:** The geometry may also be described by element variables in the Exodus file. Element variables should represent material volume fractions where the sum of element variables for any one cell should be between 0.0 and 1.0. Any number of element variables may be used where each unique variable defined will describe an element block in the final Exodus mesh produced. if the sum of element variables is less than 1.0 for any one element, a void material will be assumed and removed from the base mesh unless the **mesh_void** option is used.

**Limitations:**

- An STL file and element variables cannot be used in the same input. If element variables are present in the Exodus file and an STL or Diatom file is used, the element variables will be ignored.
- If an input mesh is used, any Cartesian grid specifications will be ignored (ie. **nelx**, **xmin**, **xmax**).
- The **adapt_type** option will work only for an exodus input mesh that defines a mapped mesh. Adapt types **vfrac_average (4)** and **vfrac_difference (6)** are currently the only criteria supported with the **input_mesh** option.

---

#### Blocks of Input Base Exodus Mesh

```
Command: input_mesh_blocks     Block ids of Input Base Exodus mesh

Input file command:   input_mesh_blocks <arg>
Command line options: -imb <arg>
Argument Type:        integers > 0
```

Command Description:

This option is valid when specifying both **input_mesh** and **spn_file**. Using this option, the materials of the cells in the spn file are mapped onto only the elements of the specified

blocks in the input_mesh file. The remaining blocks are treated as void. The behavior without this option maps the materials of the cells in the spn file onto elements of all blocks in the input_mesh file.

---

#### Material Definition with Input Mesh

```
Command: input_mesh_material     Material definition with input mesh

Input file command:   input_mesh_material <arg>
Command line options: -imm <arg>
Argument Type:        integers > 0
Input arguments: geometry (0)
                 blocks (1)
                 proximity_blocks (2)
                 geometry_and_blocks (3)
```

Command Description:

This option is valid when specifying an 'input_mesh' . Using this option, the material definition in the final mesh may be defined based on the material definitions on the geometry, or based on the block ids of the input mesh. For example, a diatom file defining geometry would have materials defined which are used to define the materials in the final mesh. The default is to use material definitions on the geometry. Possible options are:

- **geometry (0):** Material defined by geometry. Block definitions in input file will be ignored. Instead, only the geometry defined from the STL or diatom geometry definition will be respected.
- **blocks (1):** Material defined by blocks in input mesh. Both the geometry definition defined by the STL/diatom file and by the block definition in the input file will be used to construct the mesh. Block definitions used in the final mesh will be defined only by the input_mesh blocks, ignoring any material definitions in the diatom file. Note that pillowing may be required to maintain mesh quality where interior intersecting interfaces between STL/diatom geometry and block geometry occur. When used with the **mesh_void** option, element block groupings are made only for the interior of the STL/diatom geometry. Elements in the void region are assigned to the designated **void_mat**
- **proximity_blocks (2):** Material defined by blocks in input mesh based on proximity method. Geometric interfaces in the final mesh will be defined based on the STL/diatom definition, similar to the **geometry (0)** method. Material/block assignment for elements is defined only by the proximity of individual elements contained within the regions defined by blocks in the input mesh. Block assignment is made by collecting all elements in the final mesh whose centroid falls within a given block in the input mesh. Note that this method avoids the interior intersections and subsequent poor mesh quality that may result from the **blocks (1)** method described above. As a result, interior interfaces may be non-

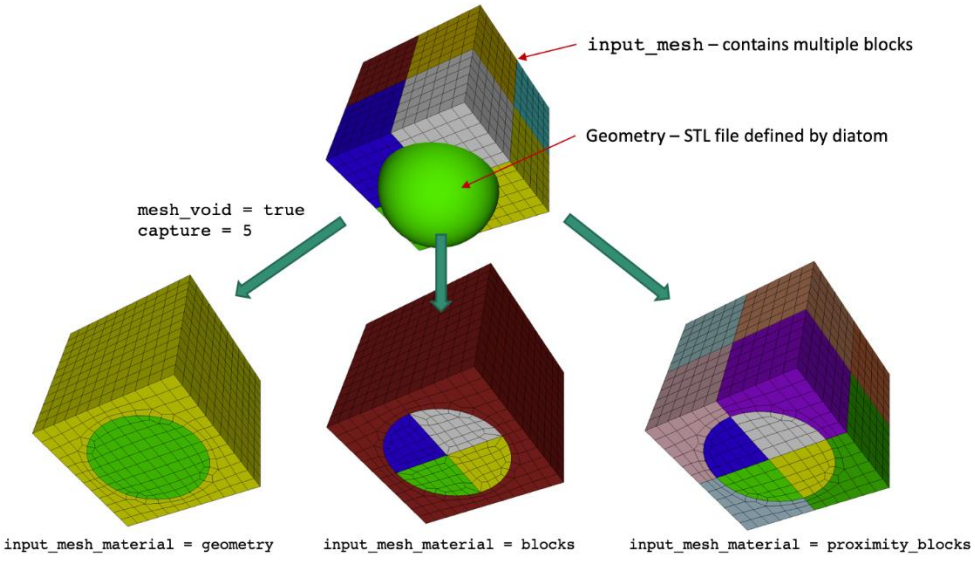
continuous (stair-step). In order to avoid collisions between the material ID defined in the diatom file and those defined in the input_mesh, block IDs in the final mesh are assigned based on the following definition:

```
ID = (diatom_material_ID-1)*max_input_mesh_block_ID +
input_mesh_block_ID
```

- If the max_input_mesh_block_ID is less than the max_diatom_material_ID, then the following definition is used:

```
ID = (input_mesh_block_ID-1)*max_diatom_material_ID +
diatom_material_ID
```

- **geometry_and_blocks (3):** Material defined by the stl geometry and the blocks in input mesh. Both the geometry definition defined by the STL/diatom file and by the block definition in the input file will be used to construct the mesh. Block definitions used in the final mesh will be defined by both the material IDs defined in the diatom file and the input_mesh blocks. Where stl geometry overlaps a material block defined in the input_mesh, the stl geometry material will take precedence.




Example of the resulting meshes using the different options for **input_mesh_material**. In this case, geometry is defined by an STL file (sphere) and **input_mesh** is a brick containing 8 different material blocks.

**Input Base mesh defined by Pamgen**

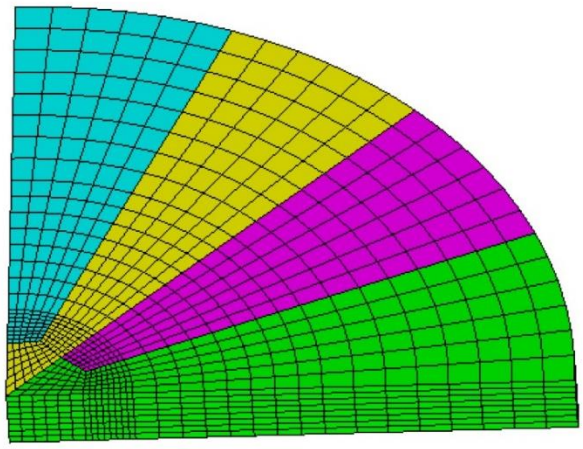
```
Command: input_mesh_pamgen     Input Base mesh defined by Pamgen

Input file command:   input_mesh_pamgen <arg>
Command line options: -imp <arg>
Argument Type:        file name with path
```

Command Description:

Option to use Pamgen to create a base mesh for Sculpt. Pamgen is an open source meshing tool developed at Sandia for generating hexahedral meshes from geometric primitives. In addition to being a stand-alone meshing solution, it is a parallel tool that is integrated as an inline meshing tool for Sandia's shock physics simulation tool, Alegra. Pamgen has also been integrated in Sculpt as a solution for automatically defining a base mesh.
The **input_mesh_pamgen** option permits a mesh defined my Pamgen input parameters to define the base mesh. A limited set of brick and cylinder primitives are supported by Pamgen. The name of an ascii file containing the pamgen mesh definition is used as the argument for this option. The following is a simple example of a pamgen mesh description. It generates a partial cylinder with a span of 90 degrees and height of 1.0. Other parameters allow for specific interval and sizing specifications as well as block/material identification.

```
mesh
    radial trisection
        trisection blocks, 2
        zmin -0.00075
        numz 1
            zblock 1 1. interval 8
        numr 3
            rblock 1 2.0 interval 8
            rblock 2 3.0 interval 8
            rblock 3 4.0 interval 8
        numa 1
            ablock 1 90. interval 24
    end
    set assign
        nodeset, ilo, 100
        block sideset, ihi, 45, 2
    end
end
```

For a full description of Pamgen and input parameters see the following document:
David M. Hensinger, Richard R. Drake, James G. Foucar, Thomas A. Gardiner, "Pamgen, a Library for Parallel Generation of Simple Finite Element Meshes", Sandia Report SAND2008-1933 (2008)



Base mesh generated by pamgen using the above input parameters. Colors represent 4 different processors when used in parallel mode.

Similar to the **input_mesh** option, the same geometry input options are available. They include **stl_file**, **diatom_file** and **input_spn**. See the **input_mesh** option for additional details and limitations.

---

**Join Parallel Files**

```
Command: join_parallel     Join parallel files

Input file command:   join_parallel <arg>
Command line options: -jp <arg>
Argument Type:        true/false
Input arguments: off (0)
                 false (0)
                 on (1)
                 true (1)
```

Command Description:

Import a set of files, one per processor, using the base name defined by the **import_mesh** option. When not used (default), the assumption of input exodus meshes is to include parallel (nemesis) data where parallel relationships between neighboring processors has already been established. This is normally done by using the SEACAS **decomp** tool to decompose a single exodus mesh into multiple files.
If the **join_parallel** option is used, sculpt assumes the parallel relationships are not included and will establish these relationships based on proximity of node locations at processor boundaries. Note that the file naming convention should follow the standard exodus parallel file naming convention. For example, a mesh spread across 4 files would be named:

```
    brick.e.4.0
```

```
brick.e.4.1
brick.e.4.2
brick.e.4.3
```

This option is currently only implemented for axis aligned rectangular processor domains. This option can also be used to just stitch exodus files and dump resulting files without any additional sculpt operations. The following is an example of a sculpt input file that does simple stitching without any additional sculpt operations:

```
begin sculpt
  input_mesh = brick.e
  exodus_file = brick_ouput
  join_parallel  = true
  input_mesh_material = blocks
  stair = fast
end sculpt
```

# *Sculpt Smoothing*

Sculpt options for specifying how the mesh will be smoothed following mesh generation. Sculpt includes a tiered approach to smoothing to improve element quality. It starts by applying smoothing to all nodes in the mesh and progressively restricts the smoothing operations to only those nodes that fall below a user-defined scaled Jacobian threshold. Default numbers of iterations and thresholds for each smoothing phase have been tuned for general use, however it may be worthwhile to adjust these parameters. The three smoothing phases include:

- **Laplacian Smoothing:** Applied to all elements. Very inexpensive fast approach to improve quality, but can result in degraded element quality if applied to excess. A fixed default of 2 iterations is applied to all hexes. Increasing the num_laplace parameter can improve some cases, especially convex shapes.
- **Optimization Smoothing:** Applied only to elements who's scaled Jacobian falls below the opt_threshold parameter (default 0.6) and their surrounding elements. This approach uses a more expensive optimization technique to improve regions of elements simultaneously. The max_opt_iters parameter can control the maximum number of iterations applied (default is 5). Iterations will terminate, however, if no further improvement is detected. Because this method optimizes node locations simultaneously, neighboring nodes with competing optimum can sometimes limit mesh quality.
- **Spot Optimization:** Also known as parallel coloring, is applied only to elements who's element quality falls below the pcol_threshold parameter (default 0.2). This technique is the most expensive of the techniques, but focusses only on nodes that are immediately adjacent to poor quality hexes. Each node is smoothed independently of its neighbors, and may require a high number of iterations using the max_pcol_iters to achieve desired results. Increasing the pcol_threshold and max_pcol_iters may yield improved results.

```
Smoothing                          -smo     --smoothing
  --smooth                         -S     <arg> Smoothing method
  --csmooth                        -CS    <arg> Curve smoothing method
  --laplacian_iters                -LI    <arg> Number of Laplacian smoothing
iterations
  --max_opt_iters                  -OI    <arg> Max. number of parallel Jacobi opt.
iters.
  --opt_threshold                  -OT    <arg> Stopping criteria for Jacobi opt.
smoothing
  --curve_opt_thresh               -COT   <arg> Min metric at which curves won't be
honored
  --max_pcol_iters                 -CI    <arg> Max. number of parallel coloring
smooth iters.
  --pcol_threshold                 -CT    <arg> Stopping criteria for parallel color
smooth
  --max_gq_iters                   -GQI   <arg> Max. number of guaranteed quality
smooth iters.
  --gq_threshold                   -GQT   <arg> Guaranteed quality minimum SJ
threshold
  --geo_smooth_max_deviation       -GSM   <arg> Geo Smoothing Maximum Deviation
```

Sculpt Command Summary

---

**Smooth**

```
Command: smooth     Smoothing method

Input file command:   smooth <arg>
Command line options: -S <arg>
Argument Type:        integer (0, 1, 2, 3)
Input arguments: off (0)
                 default (1)
                 on (1)
                 fixed_bbox (2)
                 no_surface_projections (3)
                 to_geometry (4)
                 to_geom (4)
                 geo_smooth (5)
                 geometry_smoothing (5)
                 geo_smoothing (5)
```

Command Description:

Automatic adjustment of node locations following meshing to improve element quality. Controls the combined Laplacian and optimization smoothing procedures applied to volume and surface nodes (see csmooth for curve smoothing options) Uses the **laplacian_iters**, **max_opt_iters**, **opt_threshold**, **max_pcol_iters**, **pcol_threshold**, **mqx_gq_iters** and **gq_threshold** arguments to control the sensitivity and aggressiveness of the smoothing operations. In most cases, the default options for these

parameters are sufficient, however increasing iterations or threshold values, while potentially causing longer run times, may result in improved mesh quality.
Smoothing will adjust the location of nodes on surfaces, projecting them to an approximated surface representation defined by interface reconstruction from volume fractions. In addition to turning smoothing **on** and **off**, the surface projection characteristics can be adjusted using the **bbox_fixed** and **no_surface_projections** options.

- **off (0):** No volume and surface smoothing is performed.
- **on/default (1):** (Default) Combined Laplacian/Optimization (Hybrid) smoothing both surface and volumes. Automatic boundary buffer layer improvement is performed at interior surfaces intersecting the domain boundary.
- **fixed_bbox (2):** Uses standard hybrid smoothing procedure (option 1), however nodes at the the domain boundaries will be projected to one of the six planes of the bounding box or unstructured **input_mesh**. This option turns off the automatic boundary buffer improvement.
- **no_surface_projections (3):** Uses the **fixed_bbox** method, however interior surfaces are not projected. This can result in smoother interior surface representations for microstructures models. This is effective in smoothing noisy surface data, but can potentially reduce overall volume. The **laplacian_iters** option will control the amount of smoothing that will occur, with higher numbers of iterations resulting in a smoother surface representation, but also resulting in more reduction in geometric volume. This method is default for microstructures file formats.
- **to_geometry (4):** When used with the **capture** option, smoothing will also move nodes to the closest geometry entity. It must currently be used with **capture** to ensure that curves and surfaces are first identified and associated with boundary mesh entities. This option will only work with STL or diatom input that contains STL geometry.
- **geo_smooth (5):** Similar to the **smooth = to_geometry** option, it is also intended to be used with the **capture** option. With the **geo_smooth** option, nodes are initially projected to the closest geometry entity. However, following the initial projection, Sculpt will utilize the Laplacian smoothing operation, similar to the **smooth = no_surface_projections** option, to effectively remove noise and local deviations in the geometry definition. The **laplacian_iters** option will control the amount of geometric smoothing that will occur. The optional command **geo_smooth_max_deviation** can also be used to control the maximum distance any individual node location can deviate from its original geometry definition.

**Boundary Buffer Improvement:** Sculpt's smoothing procedures will use an automatic boundary buffer improvement method. It will attempt to improve the quality of hexes where interior surfaces are close to tangent with the bounding box. This can result in nodes that may not lie precisely on the planes of the domain boundary. The **fixed_bbox (2)** and **no_surface_projections (3)** options will turn off the automatic boundary buffer improvement.

---

#### Curve Smoothing

```
Command: csmooth     Curve smoothing method

Input file command:   csmooth <arg>
Command line options: -CS <arg>
Argument Type:        integer (0, 1, 2, ...6)
Input arguments: off (0)
                 circle (1)
                 hermite (2)
                 average_tangent (3)
                 neighbor_surface_normal (4)
                 vfrac (5)
                 linear (6)
```

Command Description:

The csmooth option controls the smoothing method used on curves. In most cases the default should be sufficient, however it may be useful to experiment with different options. The default curve smoothing option is **vfrac (5)**. The following curve smoothing options are available:

- **off (0):** No curve smoothing will be performed.
- **circle (1):** Nodes projected to a fitted circle defined current node and its two neighbors.
- **hermite (2):** Nodes projected based on Hermite interpolation. Note that this method can only be used in serial (-j 1)
- **average_tangent (3):** Nodes projected based on average tangent of neighbors. Note that this method may not be parallel serial consistent.
- **neighbor_surface_normal (4):** Nodes projected based on neighboring surface normals and the resulting intersecting planes.
- **vfrac (5):** (Default) Nodes projected to initial curve interface defined from the original volume fraction data.
- **linear (6):** Nodes projected to the linear segment defined by the node and its two immediate neighbors.

---

#### Laplacian Iterations

```
Command: laplacian_iters     Number of Laplacian smoothing iterations

Input file command:   laplacian_iters <arg>
Command line options: -LI <arg>
Argument Type:        integer >= 0
```

Command Description:

Number of Laplacian smoothing iterations performed when Hybrid smoothing option is used. Default value is 2.

---

#### Maximum Optimization Iterations

```
Command: max_opt_iters     Max. number of parallel Jacobi opt. iters.

Input file command:   max_opt_iters <arg>
Command line options: -OI <arg>
Argument Type:        integer >= 0
```

Command Description:

Indicates the maximum number of iterations of optimization-based smoothing to perform. May complete sooner if no further improvement can be made. Default is 5

---

#### Optimization Threshold

```
Command: opt_threshold     Stopping criteria for Jacobi opt. smoothing

Input file command:   opt_threshold <arg>
Command line options: -OT <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:

Indicates the value for scaled Jacobian where Optimization smoothing will be performed. Elements with scaled Jacobian less than opt_threshold and their neighbors will be smoothed. Default value is 0.6

---

#### Curve Optimization Threshold

```
Command: curve_opt_thresh     Min metric at which curves won't be honored

Input file command:   curve_opt_thresh <arg>
Command line options: -COT <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:

Indicates the value for scaled Jacobian where if a node that falls on a curve has neighboring quads less than this value, then the smoothing will no longer honor the curve definition. Instead the optimization smoother will attempt to place the node to optimize the neighboring mesh quality, without regard for its placement on its owning curve.
Normally this value should be set close to zero to avoid too many nodes from floating off of their owning curves, however, if mesh quality is constrained by curve geometry, setting this value higher can help to avoid bad or poor quality elements. Default for this value is 0.1.

---

#### Maximum Parallel Coloring Iterations

```
Command: max_pcol_iters     Max. number of parallel coloring smooth iters.

Input file command:   max_pcol_iters <arg>
Command line options: -CI <arg>
Argument Type:        integer >= 0
```

Command Description:

Maximum number of spot smoothing (also known as parallel coloring) iterations to perform. May complete sooner if no further improvement can be made. Default is 100. See also pcol_threshold.

---

#### Parallel Coloring Threshold

```
Command: pcol_threshold     Stopping criteria for parallel color smooth

Input file command:   pcol_threshold <arg>
Command line options: -CT <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:

Indicates scaled Jacobian threshold for spot smoothing (also known as parallel coloring). A parallel coloring algorithm is used to uniquely identify and isolate nodes to be improved using optimization. Default is 0.2.

---

#### Maximum Guaranteed Quality Iterations

```
Command: max_gq_iters     Max. number of guaranteed quality smooth iters.
```

```
Input file command:   max_gq_iters <arg>
Command line options: -GQI <arg>
Argument Type:        integer >= 0
```

Command Description:

Maximum number of guaranteed quality smoothing iterations to perform. Guaranteed quality smoothing performs a constrained Laplacian smoothing algorithm to adjust node locations. If the result of a smoothing operation results in adjacent element quality falling below the specified **gq_threshold** value, then move distance is cut until minimum threshold is achieved or the metric is improved. To achieve parallel consistency, a parallel coloring methodology is employed. The **max_gq_iters** defines the maximum number of parallel color iterations employed. Default is 0 (off).
Note that guaranteed quality can be utilized in conjunction with other smoothing methods (Laplacian, Optimization and Parallel Coloring), however to be effective it is normally used independent from other smoothing. For example, to use guaranteed quality the following is suggested:

```
laplacian_iters = 0
max_opt_iters = 0
max_pcol_iters = 0
max_gq_iters = 100
```

---

#### Guaranteed Quality Threshold

```
Command: gq_threshold     Guaranteed quality minimum SJ threshold

Input file command:   gq_threshold <arg>
Command line options: -GQT <arg>
Argument Type:        floating point value (-1.0 -> 1.0)
```

Command Description:

Indicates scaled Jacobian threshold for guaranteed quality smoothing. Default is 0.2. see also **max_gq_iters**

---

#### Geo Smooth Max Deviation

```
Command: geo_smooth_max_deviation     Geo Smoothing Maximum Deviation

Input file command:   geo_smooth_max_deviation <arg>
Command line options: -GSM <arg>
Argument Type:        floating point value (>= 0.0
```

Command Description:

Used only in conjunction with the **smooth = geo_smooth** option. It controls the maximum distance any individual node can deviate from the geometry definition. If a smoothing operation computes a location that will move a node further than the prescribed **geo_smooth_max_deviation** value from the geometry, the node movement will be artificially limited by this value. If not specificied, no limitations will be placed on node movement due to smoothing operations when the **smooth = geo_smooth** is used. A value of zero (0) will constrain all nodes at interfaces to lie on the geometry, similar to the **smooth = to_geometry** option. When using this option, the maximum deviation from the geometry for any individual node will be reported in the MESH SUMMARY in the Sculpt ouput.

## Automatic Scheme Selection


- Default Scheme Selection
- Automatic Scheme Selection General Notes
- Surface Auto Scheme Selection
- Volume Auto Scheme Selection


For volume and surface geometries the user may allow CUBIT to automatically select the meshing scheme. Automatic scheme selection is based on several constraints, some of which are controllable by the user. The algorithms to select meshing schemes will use topological and geometric data to select the best quad or hex meshing tool. Auto scheme selection will not select tet or tri meshing algorithms. The command to invoke automatic scheme selection is:

```
{geom_list} Scheme Auto
```

Specifically for surface meshing, interval specifications will affect the scheme designation. For this reason it is recommended that the user specify intervals before calling automatic scheme selection. If the user later chooses to change the interval assignment, it may be necessary to call scheme selection again. For example, if the user assigns a square surface to have 4 intervals along each curve, scheme selection will choose the surface mapping algorithm. However if the user designates opposite curves to have different intervals, scheme selection will choose paving, since this surface and its assigned intervals will not satisfy the mapping algorithm's interval constraints. In cases where a general interval size for a surface or volume is specified and then changed, scheme selection will not change. For example, if the user specified an interval size of 1.0 a square 10X10 surface, scheme selection will choose mapping. If the user changes the interval size to 2.0, mapping will still be chosen as the meshing scheme from scheme selection. If a mesh density is not specified for a surface, a size based on the smallest curve on the surface will be selected automatically.

*Default Scheme Selection*

If the user does not set a scheme for a particular entity and chooses to mesh the entity, CUBIT will automatically run the auto scheme selection algorithm and attempt to set a scheme. In cases where the auto scheme selection fails to choose a scheme, the meshing operation will fail. In this case explicit specification of the meshing scheme and/or further geometry decomposition may be necessary.
The default scheme selection in CUBIT, unless otherwise set, will attempt to set a quadrilateral or hexahedral meshing scheme on the entity. If tet or tri meshing will always be the desired element shape, the following command can be used:

```
Set Default Element [Tet|Tri|HEX|QUAD|None]
```

Setting the default element to **tet** or **tri** will bypass the auto scheme selection and always use either the triadvance or tetmesh schemes if the scheme has not otherwise been set by the user. The default settings of **quad** or **hex** will use the automatic scheme selection. Previous functionality of CUBIT used a default scheme of map and interval of 1 for all surface and volume entities. For backwards compatibility and if this behavior is still desired, the **none** option may be used on the **set default element** command.

*Auto Scheme Selection General Notes*

In general, automatic scheme selection reduces the amount of user input. If the user knows the model consists of 2.5D meshable volumes, three commands to generate a mesh after importing or creating the model are needed. They are:

```
volume all size <value>
volume all scheme auto
mesh volume all
```

The model shown in the following figure was meshed using these three commands (part of the model is not shown to reveal the internal structure of the model).

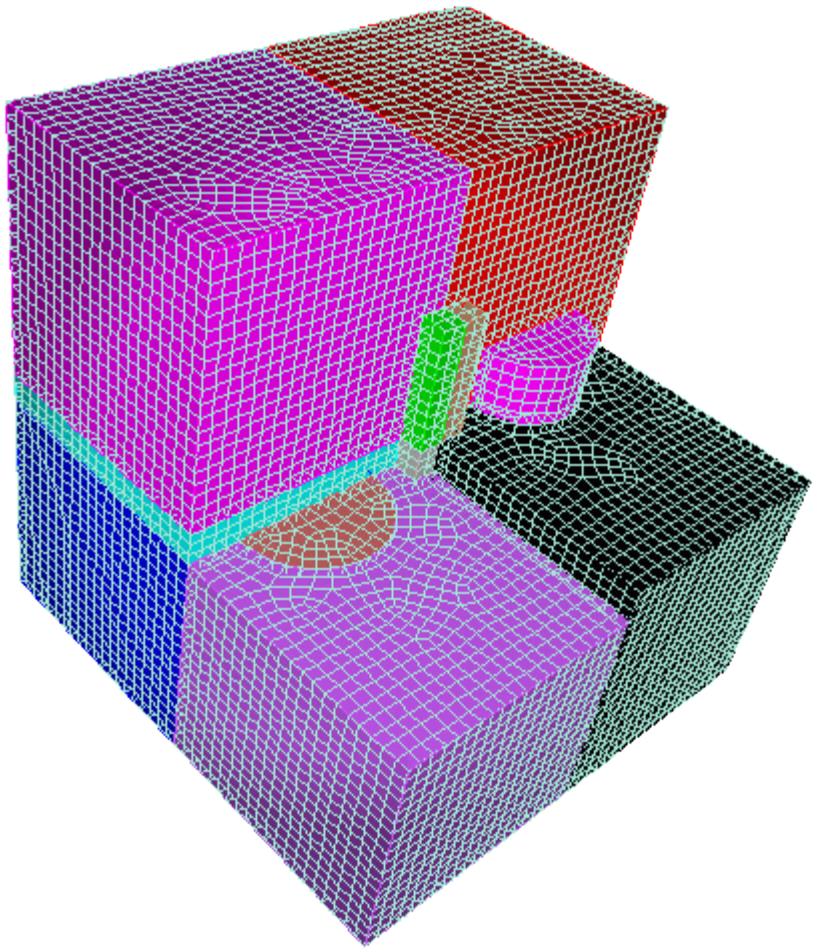


**Figure 1. Non-trivial model meshed using automatic scheme selection**

*Scheme Firmness*

Meshing schemes may be selected through three different approaches. They are: default settings, automatic scheme selection, and user specification. These methods also affect the scheme firmness settings for surfaces and volumes. Scheme firmness is completely analogous to interval firmness.
Scheme firmness can be set explicitly by the user using the command

```
{geom_list} Scheme {Default | Soft | Hard}
```

Scheme firmness settings can only be applied to surfaces and volumes.
This may be useful if the user is working on several different areas in the model. Once she/he is satisfied with an area's scheme selection and doesn't want it to change, the firmness command can be given to hard set the schemes in that area. Or, if some surfaces were hard set by the user, and the user now wants to set them through automatic scheme selection then she/he may change the surface's scheme firmness to soft or default.

*Surface Auto Scheme Selection*

Surface auto scheme selection (White, 99) will choose between Pave, Submap, Triprimitive, and Map meshing schemes, and will always result in selecting a meshing scheme due to the existence of the paving algorithm, a general surface meshing tool (assuming the surface passes the even interval constraint).
Surface auto scheme selection uses an angle metric to determine the vertex type to assign to each vertex on a surface; these vertex types are then analyzed to determine whether the surface can be mapped or submapped. Often, a surface's meshing scheme will be selected as Pave or Triprimitive when the user would prefer the surface to be mapped or submapped. The user can overcome this by several methods. First, the user can manually set the surface scheme for the "fuzzy" surface. Second, the user can manually set the "vertex types" for the surface. Third, the user can increase the angle tolerance for determining "fuzziness." The command to change scheme selection's angle tolerances is:

**[Set] Scheme Auto Fuzzy [Tolerance] {value} (value in degrees)**

The acceptable range of values is between 0 and 360 degrees. If the user enters 360 degrees as the fuzzy tolerance, no fuzzy tolerance checks will be calculated, and in general mapping and submapping will be chosen more often. If the user enters 0 degrees, only surfaces that are "blocky" will be selected to be mapped or submapped, and in general paving will be chosen more often.

*Volume Auto Scheme Selection*

When automatic scheme selection is called for a volume, surface scheme selection is invoked on the surfaces of the given volume. Mesh density selections should also be specified before automatic volume scheme selection is invoked due to the relationship of surface and volume scheme assignment.
Volume scheme selection chooses between Map, Submap and Sweep meshing schemes. Other schemes can be assigned manually, either before or after the automatic scheme selection.
Volume scheme selection is limited to selecting schemes for 2.5D geometries, with additional tool limitations (e.g. Sweep can currently only sweep from several sources to a single target, not multiple targets); this is due to the lack of a completely automatic 3D hexahedral meshing algorithm. If volume scheme selection is unable to select a meshing scheme, the mesh scheme will remain as the default and a warning will be reported to the user.
Volume scheme selection can fail to select a meshing scheme for several reasons. First, the volume may not be mappable and not 2.5D; in this case, further decomposition of the model may be necessary. Second, volume scheme selection may fail due to improper surface scheme selection. Volume schemes such as Map, Submap, and Sweep require certain surface meshing schemes, as mentioned previously.

free

# *Radialmesh*

**Summary**: Creates a free cylindrical mesh with precise node locations based on input radii, angles, and offsets, then creates mesh-based geometry to fit the mesh.
**Syntax**:

```
Create Radialmesh \
   NumZ <val> [Span <val>] \
      Zblock 1 [<offset val>] \
            {Interval|Bias|Fraction|First Size} <val> \
           [{Interval|Bias|Fraction|Last Size} <val>] \
      Zblock 2 [<offset val>] \
            {Interval|Bias|Fraction|First Size} <val> \
           [{Interval|Bias|Fraction|Last Size} <val>] \
      ... NumZ \

   NumR <val> {Trisection|Initial Radius<val>} \
      Rblock 1 <offset radius val> \
            {Interval|Bias|Fraction|First Size} <val> \
           [{Interval|Bias|Fraction|Last Size} <val>] \
      Rblock 2 <offset radius val> \
            {Interval|Bias|Fraction|First Size} <val> \
           [{Interval|Bias|Fraction|Last Size} <val>] \
      ... NumR \

   NumA <val> [Full360] [Span <val>] \
      Ablock 1 [<offset angle val>] \
            {Interval|Bias|Fraction|First Angle} <val> \
           [{Interval|Bias|Fraction|Last Angle} <val>] \
      Ablock 2 [<offset angle val>] \
            {Interval|Bias|Fraction|First Angle} <val> \
           [{Interval|Bias|Fraction|Last Angle} <val>] \
      ... NumA
```

**Discussion**:
The purpose of the **radialmesh** command is to create a cylindrical mesh with precise node locations. Unlike all other meshing commands which place nodes using smoothing algorithms to optimize element quality, node locations for the radialmesh command are calculated based on the input radii, angles, and offsets. In addition, the radialmesh command does not mesh existing geometry. Rather, it creates a mesh based on the input parameters, after which a mesh-based geometry is created to fit the free mesh.
The radialmesh command requires input for the 3 coordinate directions (Z, radial, angular). The number of blocks in each direction is specified with the numZ, numR, and numA values in the command. Each block forms a new volume in the final mesh. All bodies in the mesh are merged to form a conformal mesh between blocks.

The Radialmesh command can create meshes which span any angle greater than 0.0 up to 360 degrees. In addition, meshes can model either a tri-section (see Figure 1), or a non-trisection mesh (see Figure 2).

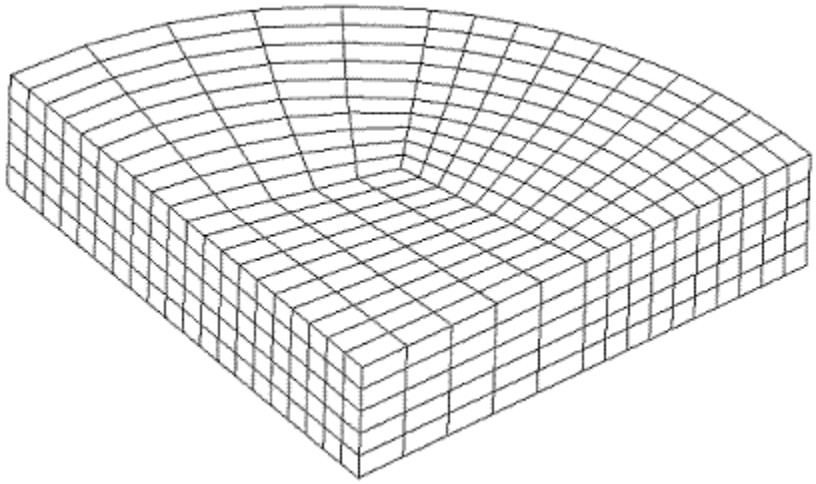


**Figure 1. Tri-section Radialmesh**

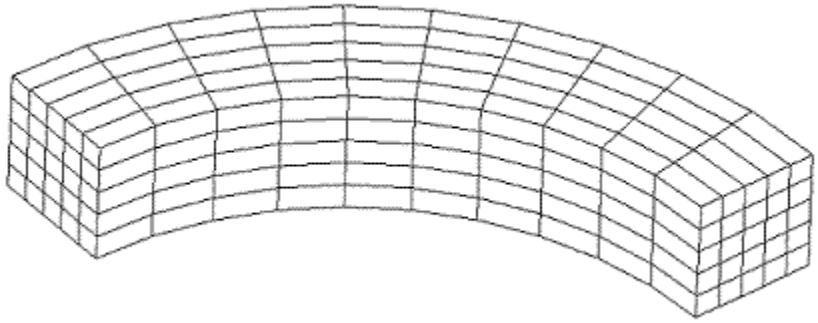


**Figure 2. Non-tri-section Radialmesh**

The command to generate the mesh in Figure 1 is:

```
create                                  radialmesh                                  \
    numZ        1        zblock        1        1        interval        5        \
    numR      3      trisection      rblock      1      2      interval      5      \
              rblock          2          3          interval          5          \
              rblock          3          4          interval          5          \
    numA 1 span 90 ablock 1 interval 10
```

The command to generate the mesh in Figure 2 is:

```
create                                  radialmesh                                  \
    numZ        1        zblock        1        1        interval        5        \
    numR    1    initial    radius    3    rblock    1    4    interval    5    \
    numA 1 span 90 ablock 1 interval 10
```

A mesh can span an entire 360 degrees by using the “full360” keyword. For example, the mesh in Figure 3 was generated with the following command:

```
create radialmesh numZ 1 zblock 1 1 interval 5 \
  numR 3 trisection rblock 1 1 interval 5 \
    rblock 2 2 interval 5 \
    rblock 3 3 interval 5 \
  numA 5 full360 span ablock 1 interval 5 \
    ablock 2 interval 5 \
    ablock 3 interval 5 \
    ablock 4 interval 5
```

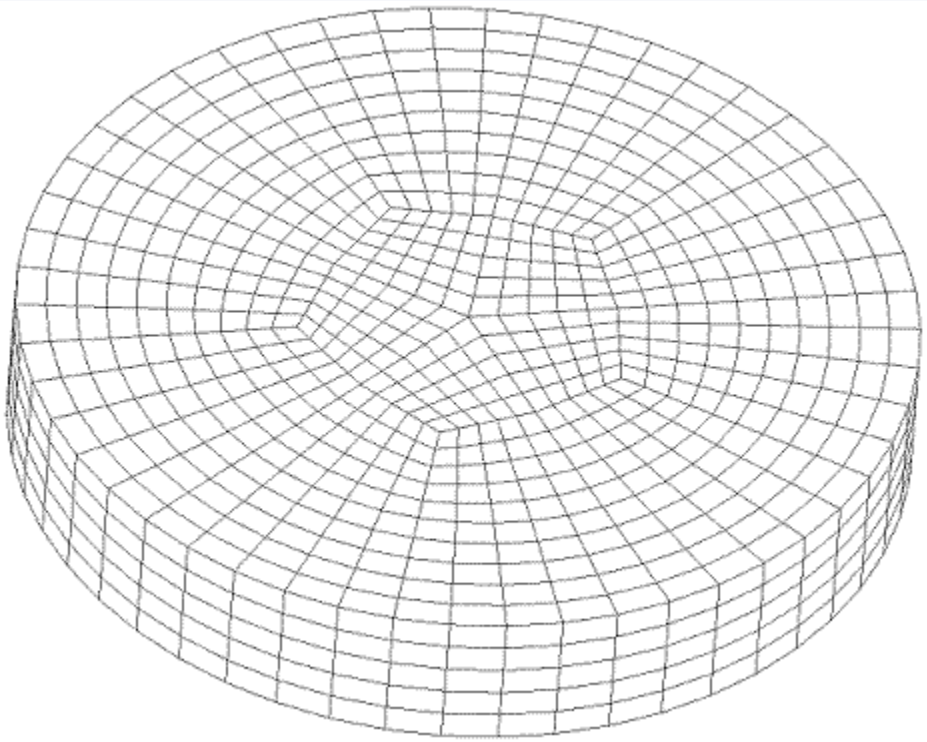


**Figure 4. Radialmesh using full360 option**

After the mesh is generated, the radialmesh command fits the mesh with mesh based geometry. The surfaces created to fit the mesh are given special names according to their location on the geometry. To see the names of the surfaces, issue the command **label surface name** after creating a radialmesh. Also, if you create a tri-section mesh, the edges on the center axis are given names. To see these names issue the command **label curve name** after creating a trisection Radialmesh.

The user can control the number of intervals and the spacing of these intervals using the optional parameters in each rblock, zblock and ablock. There are 11 combinations that these can be combined as listed below:

- **Interval Only**- Example: "interval 5." The block will be meshed with 5 equally spaced intervals.

- **First Size Only**- Example: “first size 2.5.” The block will be meshed with intervals of approximately 2.5 in length. The total number of intervals is internally calculated and depends on the overall block length.
- **Fraction Only**- Example: “fraction 0.3333.” The block will be meshed with intervals approximately 0.3333*overall block length.
- **Interval and Bias**- Example: “interval 5 bias 1.5.” There will be 5 intervals on the block, which each interval being 1.5 times the previous one. The length of each interval is calculated internally.
- **Interval and Fraction**- Example: “interval 5 fraction 0.25.” There will be 5 intervals on the block, the first being .25 of the length of the block with the remaining decreasing in size.
- **Interval and First Size**- Example: “interval 5 first size 0.2.” There will be 5 intervals on the block, the first being 0.2 in length. The remaining intervals will increase or decrease to fill the blocks length.
- **First Size and Last Size**- Example: “first size 0.2 last size 0.4.” The first interval will be 0.2 in length. The last interval will be 0.4 in length. The total number of intervals is internally calculated to allow for transition between the 2 specified sizes.
- **First Size and Bias**- Example “first size 0.2 bias 0.85.” The first interval will be 0.2 in length and the remaining intervals will scale by a factor of 0.85 from one to the next until the block is filled. The total number of intervals is internally calculated and depends on the overall block length.
- **Fraction and Bias**- Example “fraction 0.25 bias 1.25.” The first interval will be 0.25 of the overall block length and the remaining intervals will scale by a factor of 1.25 from one to the next until the block is filled. The total number of intervals is internally calculated and depends on the overall block length.
- **Interval and Last Size**- Example: “last size 1.5 interval 5.” The last interval will be 1.5 in length. The remaining intervals will scale up or down to fit 5 intervals in the block.
- **Last Size and Bias**- Example: “last size 2.0 bias 1.1.” The last interval will be 2.0 in length. The remaining intervals will scale by 1.1 until the block is filled. The total number of intervals is internally calculated and depends on the overall block length.

Figure 5 shows an example of a bias spaced mesh with the following command:

```
create radialmesh numZ 2 zblock 1 1 first size 0.2 \
    zblock 2 10 first size 0.2 last size 1.0 \
  numR 3 trisection rblock 1 1 interval 5 \
    rblock 2 2 first size .25 \
    rblock 3 5 first size .25 bias 2.0 \
  numA 1 span 90 ablock 1 interval 5
```

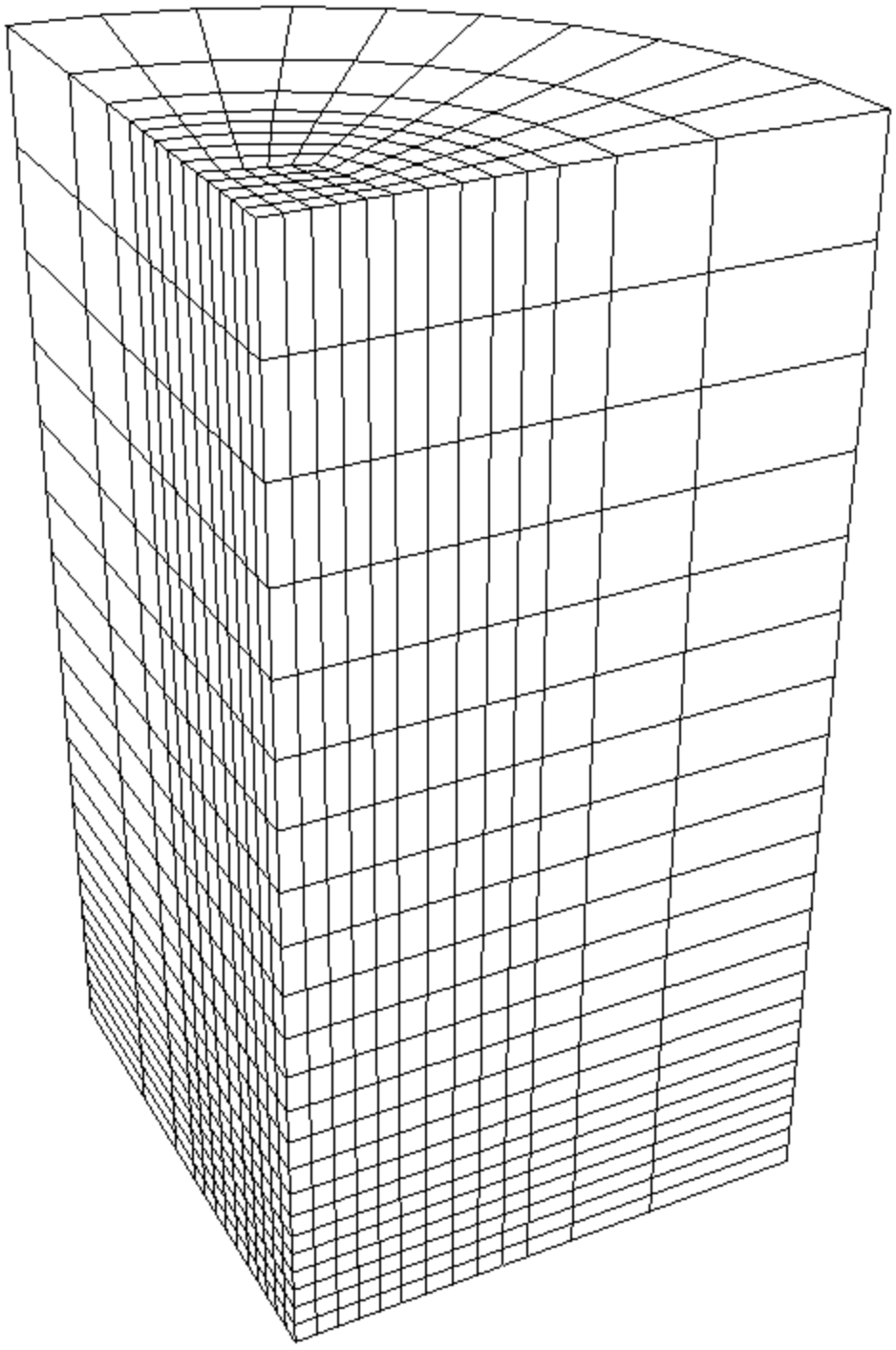


**Figure 5. Radialmesh created with biased spacing**

# mesh quality assessment

## Automatic Mesh Quality Assessment

CUBIT performs an automatic calculation of mesh quality which warns users when a particular meshing scheme or other meshing operation has created a mesh whose quality may be inadequate. These warnings are supplied in case the user forgets to manually check the mesh quality.
CUBIT automatically calculates the SHEAR quality of hexahedral and quadrilateral elements and the SHAPE quality of tetrahedral and triangular elements. The SHEAR metric measures element skew and ranges between zero and one with a value of zero signifying a non-convex element, and a value of one being a perfect, right-angled element. The SHAPE metric also ranges between zero and one with a value of zero signifying a degenerate or inverted element and a value of one signifying a perfect, equilateral element. The quality of the mesh is then defined to be the minimum value of the shear metric for hexahedral and quadrilateral elements and the shape metric for tetrahedral and triangular elements, with the minimum taken over the elements in the mesh.
If the quality of the mesh is zero, the code reports *"ERROR: Negative Jacobian Element Generated"* to the command window. By default, if the quality of the mesh is positive but less than a certain threshold, the code reports *"WARNING: Poorly-Shaped Element Generated"* to the command window. Also reported in this case is the ID of the offending element, the value of its shear (or shape) metric, and the value of the threshold to which it was compared. The default value of the threshold parameter is 0.2. Users may change the threshold value by issuing the command

**Set Quality Threshold <double=0.2>**

The user may also change what type of message is printed in the case of a poor quality, but positive Jacobian mesh. This message can be printed as a warning (the default) or an error or can be turned off completely using the command

**Set Print Quality { WARNING|Error|Off }**

The above commands only affect the message generated for meshes with a quality greater than zero and less than the given threshold value; an error will always be generated for meshes with a quality of zero (that is, for meshes containing negative Jacobian elements).

## Coincident Node Check

The ability to check for coincident nodes in the model is available in CUBIT. It uses an efficient octal hash tree to make the comparisons. The command is:

**Quality Check Coincident Node [ In ] [Group|Body|Volume|Surface|Curve|Vertex <id_range> ] [ Merge [Delete] ] [ HIGHLIGHT|Draw [color <number>]] [List] [Into Group [names|id] ]**

If no entity list is given, the command works on all the nodes in the model. If an entity list is given, then it compares the nodes on those entities *with the rest of the nodes in the model*. By default the command highlights the coincident nodes in the graphics window

and lists the total number of coincident nodes found. You can also have it clear the graphics and draw the nodes, and/or list the coincident node ids. Optionally, the coincident nodes found can be placed in a group.
If the model being operated on is from an imported universal file (i.e., no geometry exists in the model), you can merge the coincident nodes with the *merge* option. In this case *delete* allows you to delete the extra nodes (recommended). If you do not delete them they are placed into an output group.
You can control the tolerance used to check between nodes with the following setting (default = 1e-8):

**set Node Coincident Tolerance [<value>]**

## Controlling Mesh Quality

If the quality of a model after meshing isn't acceptable, there are two options available to improve that quality. The user can ask for more smoothing, or delete the mesh and start over. There are some commands that the user can invoke before meshing the model which can help to improve mesh quality. Some of them are discussed here.

*Skew Control*

The philosophy behind the skew control algorithm is one of subdividing surfaces into blocky, four-sided areas which can be easily mapped. The goal of this subdivide-and-conquer routine is to lessen the skew that a mesh exhibits on submapped regions. By controlling the skew on these surfaces, the mesh of the underlying volume will also demonstrate less skew.
The commands for skew control are:

**Control Skew Surface <surface_id_range> [Individual]**

**Delete Skew Control Surface {surface_list} [Propagate]**

The keyword **Individual** is deprecated. Its purpose is to specify that surfaces should be processed without regards to the other surfaces in the given list. This is not necessary, and could lead to problems with the final mesh. When the command is entered, the algorithm immediately processes the surfaces, inserting vertices and setting interval constraints on the resulting subdivided curves. In this way, the mesh is more constrained in its generation, and the resulting skew on the model can be lessened. The only surfaces that can utilize this algorithm are those that lend themselves to a structured meshing scheme, although future releases might lessen this restriction.
The user also has the ability to delete the changes that the skew control algorithm has made. This is done by using the **delete skew control** command.
When the user requests the deletion of the skew control changes on a given surface, every curve on that surface will have the skew control changes deleted, even if a given curve is shared with another surface on which skew control was performed. If the user

wishes to propagate the deletion of skew control to all surfaces which are affected by one (or more) particular surfaces, the keyword **propagate** should be used.

*Propagate Curve Bias*

When a bias mesh scheme is applied to a curve, this sometimes creates skewing of the surface mesh that is attached. Sometimes the user will want to ensure that the same bias is applied to curves on attached surfaces so that this skewing is minimized. The command for doing this is:

**Propagate Curve Bias [Surface|Volume|Body|Group <id_list>]**

This command will search out all simply mappable surfaces in the input list, find which curves of those have a bias scheme set, and will propagate that bias across the mappable surfaces.

*Adjust Boundary*

**Adjust Boundary {Surface|Group} <id_range> [Angle <double>]**

This command can be used to improve element quality for mapped or submapped surface meshes. Often, due to vertex positions, the curve meshing for a surface will lead to a poor quality surface mesh. This command can be used to adjust the curve meshes in an attempt to generate a better quality surface mesh. The command works by looking at the angle the mesh edges leave the boundary. In a perfect mapped or submapped mesh, the mesh edges will be orthogonal to the boundary, or will go off at 90 degree angles. The adjust boundary command looks at the deviation of the mesh edges, and if it is greater than the prescribed angle deviation, it will move the node location such that it is 90 degrees, if possible. The deviation angle by default is 5 degrees and can be changed by the user through the **[Angle <double>]** option in the command. In order to modify the curve meshes, the surface meshes are first deleted then later remeshed after the curve meshes have been repositioned and fixed. This command assumes that the volumes attached to the surface have not been meshed, if they have been, the command will return an error message. It should be noted that this command, while useful, may not always work due to interval constraints (i.e., you may need to change the intervals on the surface), or if the surfaces are not very blocky.

## Metrics for Edge Elements

The metrics used for edge elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range |
|---|---|---|---|
| Length | L^0 | 0 to inf | None |

*Quality Metric Definitions:*

**Length**: Distance between beginning and ending nodes of an edge

*Comments on Algebraic Quality Measures*

1. The quality command for edge length only accepts edge elements as input; it does not accept geometry as input.
2. The length metric is currently only available for edge elements. Edge elements are created by default when curves and surfaces are meshed. Edge elements are not created for interior volume elements.

## Metrics for Hexahedral Elements

The metrics used for hexahedral elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range | Reference |
|---|---|---|---|---|
| Aspect Ratio | L^0 | 1 to inf | 1 to 4 | 1 |
| Skew | L^0 | 0 to 1 | 0 to 0.5 | 1 |
| Taper | L^0 | 0 to +inf | 0 to 0.4 | 1 |
| Stretch | L^0 | 0 to 1 | 0.25 to 1 | 2 |
| Diagonal Ratio | L^0 | 0 to 1 | 0.65 to 1 | 3 |
| Dimension | L^1 | 0 to inf | None | 1 |
| Condition No. | L^0 | 1 to inf | 1 to 8 | 5 |
| Mass Increase Ratio | L^0 | 1 to inf | None | |
| Node Distance | L^1 | -inf to inf | None | |
| Scaled Jacobian | L^0 | -1 to +1 | 0.5 to 1 | 5 |
| Shear | L^0 | 0 to 1 | 0.3 to 1 | 5 |
| Shape | L^0 | 0 to 1 | 0.3 to 1 | 5 |
| Relative Size | L^0 | 0 to 1 | 0.5 to 1 | 5 |
| Shear & Size | L^0 | 0 to 1 | 0.2 to 1 | 5 |
| Shape & Size | L^0 | 0 to 1 | 0.2 to 1 | 5 |
| Timestep | Seconds | 0 to inf | None | 6 |
| **High Order Metrics** | | | | |
| Distortion | L^0 | 0 to 1 | 0.6 to 1 | 7 |
| Element Volume | L^3 | -inf to inf | None | 1 |
| Jacobian | L^3 | -inf to inf | None | 5 |

*Hexahedral Quality Definitions*

With a few exceptions, as noted below, Cubit supports quality metric calculations for linear hexahedral elements only. When calculating quality metrics, that only support linear elements, for a higher order hexahedral element, Cubit will only use the corner nodes of the element.
**Aspect Ratio**: Maximum edge length ratios at hex center.
**Skew**: Maximum |cos A| where A is the angle between edges at hex center.
**Taper**: Maximum ratio of lengths derived from opposite edges.
**Stretch**: Sqrt(3) * minimum edge length / maximum diagonal length.
**Diagonal Ratio**: Minimum diagonal length / maximum diagonal length.
**Dimension**: Pronto-specific characteristic length for stable timestep calculation. Char_length = Volume / 2 grad Volume.
**Condition No**. Maximum condition number of the Jacobian matrix at 8 corners.
**Mass Increase Ratio**: This metric stems from the global target time step and the element time step. The density required to fulfill the target time step (via mass scaling) divided by the block density is termed the mass increase ratio. Because the density within each element is constant, a ratio in the element density is equivalent to a ratio in the element mass. This metric calculates the requisite density for each element to attain the prescribed target time step. If that density is greater than the defined density, the metric yields a value greater than one. This desired global time step is set by the user with the command:

**[Set] Target Timestep <value>**

As stated, this metric computes the element based timestep metric and consequently element blocks must be defined with material properties of Young's modulus, Poisson's ratio, and a target timestep must be set.
If this metric is computed in the context of a block ('quality block 1 mass increase ratio') an accompanying printout of the mass increase per block is given.
**Node Distance**: Minimum distance between any two adjacent corner nodes.
**Scaled Jacobian**: For linear elements the minimum Jacobian divided by the lengths of the 3 edge vectors.
**Shear**: 3/Mean Ratio of Jacobian Skew Matrix
**Shape**: 3/Mean Ratio of weighted Jacobian Matrix
**Relative Size**: Min(J, 1/J), where J is the determinant of weighted Jacobian matrix
**Shear & Size**: Product of Shear and Size Metrics
**Shape & Size**: Product of Shape and Size Metrics
**Timestep**: The approximate maximum timestep that can be used with this element in explicit transient dynamics analysis. This critical timestep is a function of both element geometry and material properties. To compute this metric on hexes, the hexes must be contained in an element block that has a material associated to it, where the materials poisson's ratio, elastic modulus, and density are defined.

#### High Order Elements

The preceding metrics will measure quality based only on the 8 corner nodes of the hexahedron. The following metrics also take into account the mid nodes.

**Distortion**: {min(|J|)/actual volume}*parent volume, parent volume = 8 for hex. Cubit also supports Distortion calculations for hex20 elements.
**Element Volume**: For linear hexes, the jacobian at hex center. For higher-order hexes, the hex is subdivided into sub-tets, the volumes of which are summed.
**Jacobian**: Minimum pointwise volume of local map at 8 corners at center of hex. Cubit also supports Jacobian calculations for hex27 elements.

*References for Hexahedral Quality Measures*


1. (Taylor, 89)
2. FIMESH code
3. Unknown
4. (Knupp, 00)
5. P. Knupp, Algebraic Mesh Quality Metrics for Unstructured Initial Meshes, to appear in Finite Elements for Design and Analysis.
6. Flanagan, D.P. and Belytschko, T., 1984, “Eigenvalues and Stable Time Steps for the Uniform Hexahedron and Quadrilateral,” Journal of Applied Mechanics, Vol. 51, pp.35-40.
7. SDRC/IDEAS Simulation: Finite Element Modeling - User's Guide


## Mesh Quality Assessment


- Metrics for Edge Elements
- Metrics for Triangular Elements
- Metrics for Quadrilateral Elements
- Metrics for Tetrahedral Elements
- Metrics for Hexahedral Elements
- Metrics for Wedge Elements
- Metrics for High Order Elements
- Mesh Quality Command Syntax
- Mesh Quality Example Output
- Automatic Mesh Quality Assessment
- Controlling Mesh Quality
- Coincident Node Check
- Mesh Topology Check
- Find Intersecting Mesh
- Measuring Number of Tets Through the Thickness


The `quality' of a mesh can be assessed using several element quality metrics available in CUBIT. Information about the CUBIT quality metrics can be obtained from the command

**Quality Describe {Hex | Hexahedral | Tet | Tetrahedral | Face | Quad | Quadrilateral | Tri | Triangular}**

which gives data on the quality metrics for each of the above element types. The following pages discuss the mesh quality assessment capabilities in CUBIT.

## Mesh Quality Example Output

The typical summary output from the command **quality surface 24** is shown in Figure 1. Figure 2 shows the corresponding histogram. The colored element display resulting from the **command quality surface 1 draw `Skew'** is shown Figure 3. A color legend is also printed to the console as shown in Figure 4.

```
Surface 24 Quad quality, 292 elements:
-------------------------------------
  Function Name    Average    Std Dev     Minimum   (id)    Maximum  (id)
----------------  ---------  ---------   --------------    -------------
   Aspect Ratio 1.339e+00  3.374e-01   1.001e+00 (244)  3.662e+00 (132)
          Skew  1.848e-01  1.461e-01   7.986e-04 (212)  6.440e-01 (284)
         Taper  1.342e-01  9.397e-02   8.689e-03 (164)  5.500e-01 (133)
       Warpage  9.991e-01  4.465e-03   9.283e-01 (14)  1.000e+00 (82)
   Element Area 6.075e-04  4.725e-04   4.941e-05 (248)  2.202e-03 (274)
       Stretch  7.276e-01  1.233e-01   3.266e-01 (147)  9.587e-01 (161)
  Maximum Angle 1.099e+02  1.329e+01   9.079e+01 (82)  1.738e+02 (14)
  Minimum Angle 7.143e+01  1.185e+01   3.373e+01 (135)  8.955e+01 (82)
  Condition No. 1.250e+00  6.244e-01   1.003e+00 (161)  1.107e+01 (14)
      Jacobian  5.125e-04  4.273e-04   9.696e-06 (14)  1.918e-03 (274)
Scaled Jacobian 9.044e-01  1.104e-01   1.072e-01 (14)  9.999e-01 (82)
         Shear  9.045e-01  1.104e-01   1.072e-01 (14)  9.999e-01 (82)
         Shape  8.436e-01  1.314e-01   9.033e-02 (14)  9.966e-01 (161)
  Relative Size 3.036e-01  2.531e-01   3.226e-03 (248)  9.710e-01 (45)
 Shear And Size 2.789e-01  2.361e-01   1.477e-03 (14)  9.389e-01 (45)
 Shape And Size 2.609e-01  2.234e-01   1.245e-03 (14)  9.389e-01 (45)
     Distortion 8.118e-01  1.352e-01   9.654e-02 (14)  9.864e-01 (82)
```

**Figure 1. Typical Summary for a Quality Command**

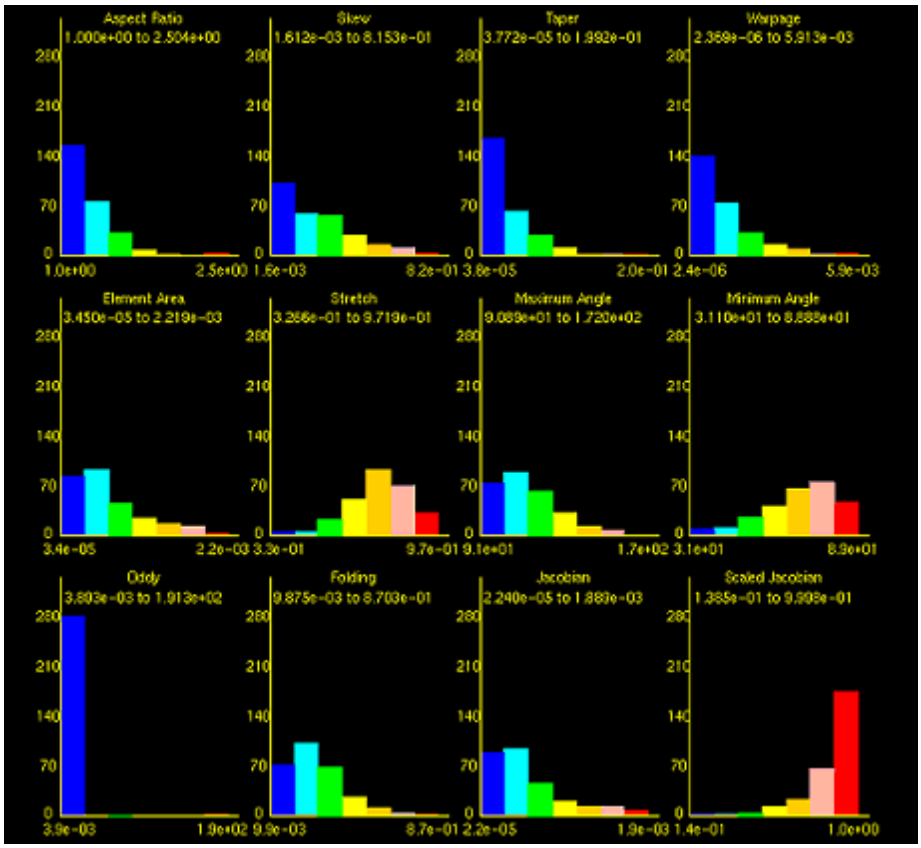



**Figure 2. Histogram output from command "Quality Surface 24 Draw Histogram"**

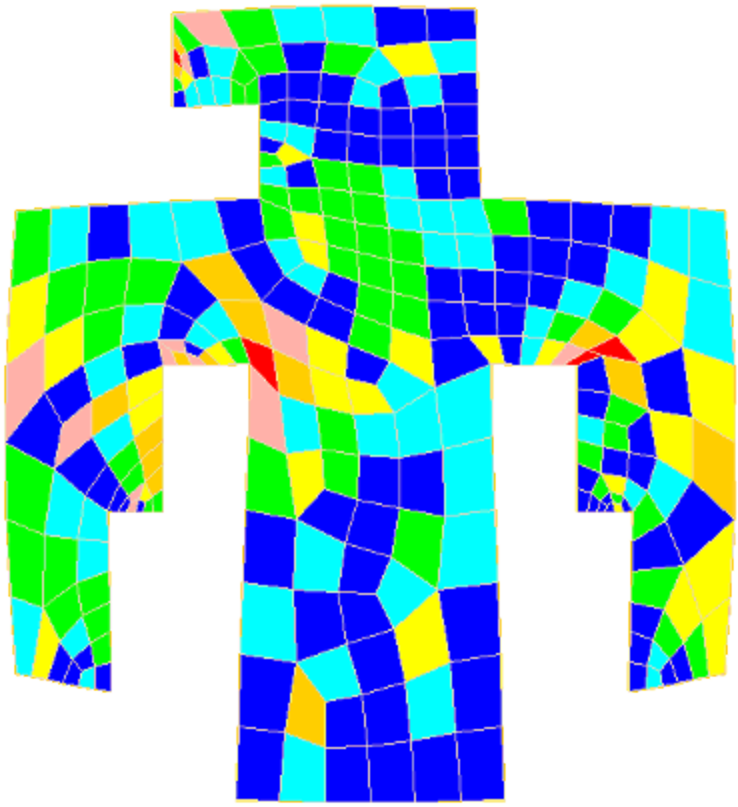


**Figure 3. Graphical output of quality metric for command "Quality Surface 24 Skew Draw Mesh"**

```
Surface 24 Quad quality, 280 elements:
    Skew ranges from 1.612e-03 to 8.153e-01 (280 entities)
    Blue ranges from 1.612e-03 to 1.178e-01 (102 entities)
    Cyan ranges from 1.178e-01 to 2.341e-01 (60 entities)
   Green ranges from 2.341e-01 to 3.503e-01 (58 entities)
  Yellow ranges from 3.503e-01 to 4.666e-01 (29 entities)
DkYellow ranges from 4.666e-01 to 5.828e-01 (15 entities)
    Pink ranges from 5.828e-01 to 6.990e-01 (12 entities)
     Red ranges from 6.990e-01 to 8.153e-01 (4 entities)
```

**Figure 4. Legend for command "Quality Surface 1 Skew Draw Mesh"**

## Mesh Quality Command Syntax

The base command to view the quality of a mesh is the following:

**Quality {geom_and_mesh_list} [metric name] [quality options] [filter options]**

Where the list contains surfaces and volumes and groups that have been meshed with faces, triangles, hexes, and tetrahedra; the list can also specify individual mesh entities or ranges of mesh entities.
If a specific metric name is given, only that metric or metrics are computed for the specified entities. Note that the metric given must be one which applies to the given entities. To see a list of quality metrics for individual entities see the Mesh Quality Assessment section and select the desired entity type: hexahedral, tetrahedral, quadrilateral, triangle. or edge
The metric name can also be more general than a specific metric. Four generalized options for metric name can be used:
**Allmetrics**: All of the metrics corresponding to the element type of the geom_and_mesh_list will be computed and reported.
**Algebraic**: All algebraic metrics corresponding to the element type of the geom_and_mesh_list will be computed and reported (e.g., Shape, Shear, Relative Size).
**Robinson**: All Robinson metrics corresponding to the element type of the geom_and_mesh_list will be computed and reported (e.g., Aspect Ratio, Skew, Taper).
**Traditional**: All the traditional Cubit metrics corresponding to the element type of the geom_and_mesh_list will be computed and reported (e.g., area, volume, angle, stretch, dimension).
If no metric name is supplied, the default metric is "**Shape**".

*Quality Options*

The quality options are:

**Scope**

**[ Global | Individual ]**

If the user specifies **individual**, one quality summary is generated for each entity specified on the command line. If the user specifies **global**, or specifies neither, then one quality summary is generated for each mesh element type.

**Draw**

**[ Draw [Histogram] [Mesh] [Monochrome] [Add] ]**

If the user specifies **draw histogram**, then histograms are drawn in a separate graphics window. The window contains one histogram for each quality metric. If the user specifies draw mesh, then the mesh elements are drawn in the default graphics window. A color-coded scale will appear in the graphics window. The histogram and mesh graphics are color coded by quality: a small metric value corresponds to red, a large metric value to blue and in-between values according to the rainbow. You can grab the side of color bar and resize it. The text gets smaller as the color bar width decreases. You can also grab in the middle of the color bar and move it around. It can be repositioned to the bottom or top and it will automatically change orientations. See Figure 1.

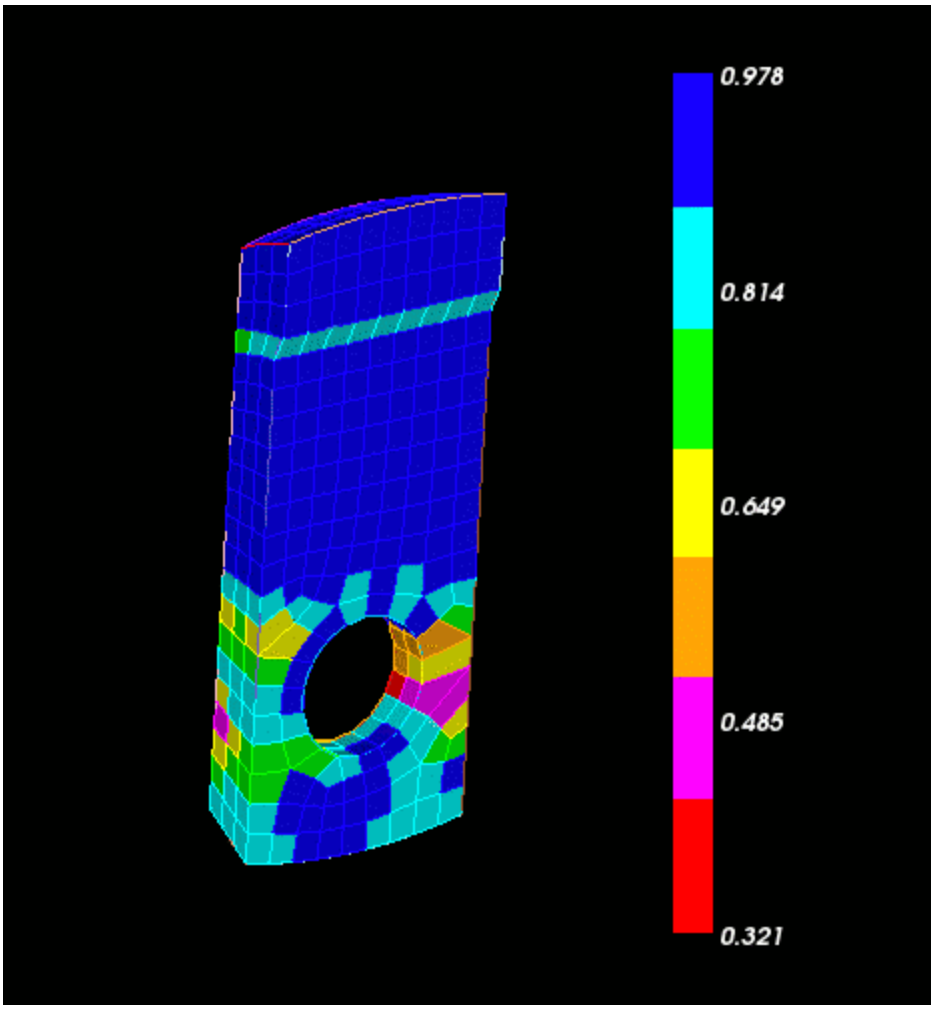



**Figure 1. Quality Scale**

If monochrome is specified, then the graphics are not color-coded. If add is specified, then the current display is not cleared before drawing the mesh elements.

#### List

**[ List [Detail] [Id] [Verbose Errors] ] [Geometry]**

If the user specifies **List**, then the quality data is summarized in text form. **List Detail** lists the mesh elements by ascending quality metric. **List Id** lists the ids of the mesh elements. If **Verbose Errors** is specified, then details about unacceptable quality elements are printed out above the summaries. If **Geometry** is specified, then a list of the geometric entities that own the elements will be printed.

#### Filter

There are several options available to *filter* the output of the quality command, using the following filter options :

**[High <value>] [Low <value>]**

Discards elements with metric values above or below value; either or both can be used to get elements above or below a specified value or in a specified range.

**[Top <number>] [Bottom <number>]**

Keeps only number elements with the highest or lowest metric values. For example, " **Quality hex all aspect ratio top 10** " would request the elements with the 10 highest values of the aspect ratio metric.

## Metrics for Quadrilateral Elements

The metrics used for quadrilateral elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range | Reference |
|---|---|---|---|---|
| Aspect Ratio | L^0 | 1 to inf | 1 to 4 | 1 |
| Skew | L^0 | 0 to 1 | 0 to 0.5 | 1 |
| Taper | L^0 | 0 to +inf | 0 to 0.7 | 1 |
| Warpage | L^0 | 0 to 1 | 0.9 to 1.0 | NEW |
| Stretch | L^0 | 0 to 1 | 0.25 to 1 | 2 |
| Minimum Angle | degrees | 0 to 90 | 45 to 90 | 3 |
| Maximum Angle | degrees | 90 to 360 | 90 to 135 | 3 |
| Condition No. | L^0 | 1 to inf | 1 to 4 | 4 |
| Deviation | L^2 | 0 to inf | None | |
| Jacobian | L^2 | -inf to inf | None | 4 |
| Scaled Jacobian | L^0 | -1 to +1 | 0.5 to 1 | 4 |
| Shear | L^0 | 0 to 1 | 0.3 to 1 | 5 |
| Shape | L^0 | 0 to 1 | 0.3 to 1 | 5 |
| Relative Size | L^0 | 0 to 1 | 0.3 to 1 | 5 |
| Shear & Size | L^0 | 0 to 1 | 0.2 to 1 | 5 |
| Shape & Size | L^0 | 0 to 1 | 0.2 to 1 | 5 |
| **High Order Metrics** | | | | |
| Distortion | L^2 | -1 to 1 | 0.6 to 1 | 6 |
| Element Area | L^2 | -inf to inf | None | 1 |

*Quadrilateral Quality Definitions*

**Aspect Ratio**: Maximum edge length ratios at quad center

**Skew**: Maximum |cos A| where A is the angle between edges at quad center
**Taper**: Maximum ratio of lengths derived from opposite edges
**Warpage**: Cosine of Minimum Dihedral Angle formed by Planes Intersecting in Diagonals
**Element Area**: Jacobian at quad center
**Stretch**: Sqrt(2) * minimum edge length / maximum diagonal length
**Minimum Angle**: Smallest included quad angle (degrees).
**Maximum Angle**: Largest included quad angle (degrees).
**Condition No**. Maximum condition number of the Jacobian matrix at 4 corners
**Jacobian**: Minimum pointwise volume of local map at 4 corners & center of quad
**Scaled Jacobian**: For linear elements the minimum Jacobian divided by the lengths of the 2 edge vectors
**Shear**: 2/Condition number of Jacobian Skew matrix
**Shape**: 2/Condition number of weighted Jacobian matrix
**Relative Size**: Min( J, 1/J ), where J is determinant of weighted Jacobian matrix
**Shear and Size**: Product of Shear and Relative Size
**Shape and Size**: Product of Shape and Relative Size
**Distortion**: {min(|J|)/actual area}*parent area, parent area = 4 for quad
**Deviation**: Absolute distance from quad centroid to its associated surface

*Comments on Algebraic Quality Measures*

Shape, Relative Size, Shape & Size, and Shear are algebraic quality metrics that apply to quadrilateral elements. Cubit encourages the use of these metrics since they have certain nice properties (see reference 5 below). The metrics are referenced to a square-shaped quadrilateral element, thus deviations from a square are measured in various ways.
Shape measures how far skew and aspect ratio in the element deviates from the reference element.
Relative size measures the size of the element vs. the size of reference element. If the element is twice or one-half the size of the reference element, the relative size is one-half. The reference element for the Relative Size metric is a square whose area is determined by the average area of all the quadrilaterals on the surface mesh under assessment
Shape and size metric measures how both the shape and relative size of the element deviate from that of the reference element.
The SHEAR metric is based on the condition number of the skew matrix. SHEAR is really just an algebraic skew metric but, since the word skew is already used in the list of quad quality metrics, Cubit has chosen to use the word 'shear.'
Shear = 1 if and only if quadrilateral is a rectangle.
The Robinson 'skew' metric equals the ideal (zero) if the quad is a rectangle. It also attains the ideal if the quad is a trapezoid, a kite, or even triangular!

*References for Quadrilateral Quality Measures*

1. (Robinson, 87)
2. FIMESH code.

3. Unknown.
4. (Knupp, 00)
5. P. Knupp, Algebraic Mesh Quality Metrics for Unstructured Initial Meshes, submitted for publication.
6. 6. SDRC/IDEAS Simulation: Finite Element Modeling--User's Guide

*Details on Robinson Metrics for Quadrilaterals*

The quadrilateral element quality metrics that are calculated are aspect ratio, skew, taper, element area, and stretch. The calculations are based on metrics described in (Robinson, 87). An illustration of the shape parameters is shown in Figure 1, below. The stretch metric is calculated by dividing the length of the shortest element edge divided by the length of the longest element diagonal.

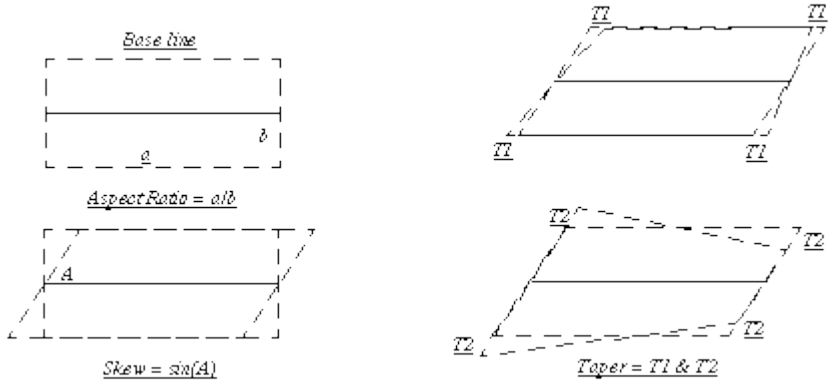



**Figure 1. Illustration of Quadrilateral Shape Parameters (Quality Metrics)**

## Metrics for Tetrahedral Elements

The metrics used for tetrahedral elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range | Reference |
|---|---|---|---|---|
| Aspect Ratio Beta | L^0 | 1 to inf | 1 to 3 | 1 |
| Aspect Ratio Gamma | L^0 | 1 to inf | 1 to 3 | 1 |
| Condition No | L^0 | 1 to inf | 1 to 3 | 2 |
| Node Distance | L^1 | -inf to inf | None | |
| Scaled Jacobian | L^0 | -1 to 1 | 0.2 to 1 | 2 |
| Shape | L^0 | 0 to 1 | 0.2 to 1 | 3 |
| Relative Size | L^0 | 0 to 1 | 0.2 to 1 | 3 |
| Shape and Size | L^0 | 0 to 1 | 0.2 to 1 | 3 |

| **High Order Metrics** | | | | |
|---|---|---|---|---|
| Distortion | L^0 | -1 to 1 | 0.6 to 1 | |
| Element Volume | L^3 | -inf to inf | None | 1 |
| Inradius | L^1 | -inf to inf | None | None |
| Jacobian | L^3 | -inf to inf | None | 2 |
| Normalized Inradius | L^0 | -1 to 1 | 0.15 to 1 | |
| Mean Ratio | L^0 | 0 to 1 | 0.2 to 1 | |
| Mass Increase Ratio | L^0 | 1 to inf | None | |
| Timestep | Seconds | 0 to inf | None | 4 |

*Tetrahedral Quality Definitions*

With a few exceptions, as noted below, Cubit supports quality metric calculations for linear tetrahedral elements only. When calculating quality metrics, that only support linear elements, for a higher order tetrahedral element, Cubit will only use the corner nodes of the element.
**Aspect Ratio Beta**: CR / (3.0 * IR) where CR = circumsphere radius, IR = inscribed sphere radius
**Aspect Ratio Gamma**: Srms**3 / (8.479670*V) where Srms = sqrt(Sum(Si**2)/6), Si = edge length
**Condition No.**: Condition number of the Jacobian matrix at any corner
**Inradius**: For all tets but tetra10s, the radius of the smallest, fully contained sphere of the linear tet. For tetra10s, the mid-edge nodes are used to subdivide the tet into 12 linear sub-tets. The inradius is the smallest inradius of the 12 linear sub-tets * 2.3.
**Jacobian**: Minimum pointwise volume at any corner. Cubit also supports Jacobian calculations for tetra15 elements.
For tetra15 elements, all 15 nodes are included for the Jacobian calculation. For all other tet types, only the corner nodes are considered.
**Node Distance**: Minimum distance between any two adjacent corner nodes.
**Scaled Jacobian**: For linear elements the minimum Jacobian divided by the lengths of 3 edge vectors
**Shape**: 3/Mean Ratio of weighted Jacobian Matrix
**Relative Size**: Min(J, 1/J), where J is the determinant of the weighted Jacobian matrix
**Shape & Size**: Product of Shape and Relative Size Metrics

#### High Order Elements

The preceding metrics will measure quality based only on the 4 corner nodes of the tetrahedron. The following metrics also take into account the mid nodes.

**Distortion**: {min(|J|)/actual volume}*parent volume, parent volume = 1/6 for tet. Cubit also supports Distortion calculations for tetra10 elements.
For tetra10 elements, the distortion metric can be used in conjunction with the shape metric to determine whether the mid-edge nodes have caused negative Jacobians in the element. The shape metric only considers the linear (parent) element. If a tetra10 has a non-positive shape value then the element has areas of negative Jacobians. However, for elements with a positive shape metric value, if the distortion value is non-positive then the element contains negative Jacobians due to the mid-side node positions.
**Element Volume**: For linear tets, (1/6) * Jacobian at corner node. For higher order tets, the tet is subdivided into sub-tets, the volumes of which are summed.
**Normalized Inradius**: Ratio of minimum subtet inner radius to tet outer radius (circumsphere). Subtets are defined by subdividing the tet into 12 smaller tets by using a common point at the centroid of the tet and the 6 mid-edge nodes as shown in Figure 1. The minimum in-radius of any of these 12 tets normalized by its parent outer-radius and a constant is used to determine this metric. The Normalized Inradius metric is also valid for linear elements, except that all mid-edge nodes are defined as the midpoint of their corner nodes.

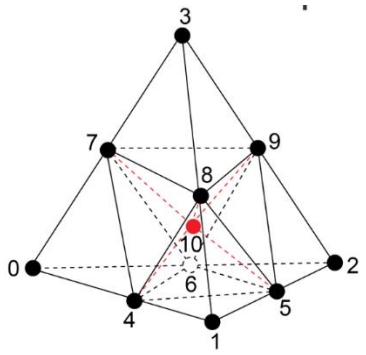



**Figure 1. Subtet subdivision used for determining Normalized Inradius quality metric**

**Mean Ratio**: General description: Mean ratio quality metric measures the deviation of a tetrahedral element from an equilateral tetrahedron through the root-mean-squared edge length. In this context, we employ a volume ratio. For a 4-node tet, the volume is compared to the cube of root-mean-squared length of the six edges. For the 10-node tet, 12 sub-tets are formed and minimum mean ratio of the 12 is returned. Unlike the normalized inradius, the mean ratio is quite sensitive to a single, highly-elongated sub-tet. We note that for an equilateral 10-node tetrahedral element, there are two families of sub-tetrahedra. Sub-tets connected to the corner nodes or parent nodes of the 4-node tet have a mean ratio of 1 by construction. They are equilateral tets. The other family of sub-tets are not equilateral tets. These interior sub-tets connected entirely to mid-edge nodes are scaled such that all sub-tetrahedra have a mean ratio of 1 for an equilateral tet.
**Mass Increase Ratio**: This metric stems from the global target time step and the element time step. The density required to fulfill the target time step (via mass scaling) divided by the block density is termed the mass increase ratio. Because the density within each element is constant, a ratio in the element density is equivalent to a ratio in the element mass. This metric calculates the requisite density for each element to attain the

prescribed target time step. If that density is greater than the defined density, the metric yields a value greater than one. This desired global time step is set by the user with the command:

**[Set] Target Timestep <value>**

As stated, this metric computes the element based timestep metric and consequently element blocks must be defined with material properties of Young's modulus, Poisson's ratio, and a target timestep must be set.
If this metric is computed in the context of a block ('quality block 1 mass increase ratio') an accompanying printout of the mass increase per block is given.
**Timestep**: The approximate maximum timestep that can be used with this element in explicit transient dynamics analysis. This critical time step is a function of both element geometry and material properties. To compute this metric on tets, the tets must be contained in an element block that has a material associated to it, where the materials poisson's ratio, elastic modulus, and density are defined.
Note that, for tetrahedral elements, there are several definitions of the term "aspect ratio" used in literature and in software packages. Please be aware that the various definitions will not necessarily give the same or even comparable results.

*References for Tetrahedral Quality Measures*


1. (Parthasarathy, 93)
2. (Knupp, 00)
3. P. Knupp, Algebraic Mesh Quality Metrics for Unstructured Initial Meshes, to appear in Finite Elements for Design and Analysis.
4. Flanagan, D.P. and Belytschko, T., 1984, "Eigenvalues and Stable Time Steps for the Uniform Hexahedron and Quadrilateral," Journal of Applied Mechanics, Vol. 51, pp.35-40.
5. SDRC/IDEAS Simulation: Finite Element Modeling - User's Guide


## Mesh Topology Check

The ability to check for non-manifold topology among mesh entities is given with the following command.

**Quality Check Topology [[Hex <range>] [Tet <range>] [Face <range>] [Tri <range>]]**

If no entity list is given, it will check the entire model. Multiple element types are also allowed. The command checks for non-manifold boundaries (edges) in the element set entered. For quads and tris the command lists and highlights all edges that have more than two tris or faces connected.

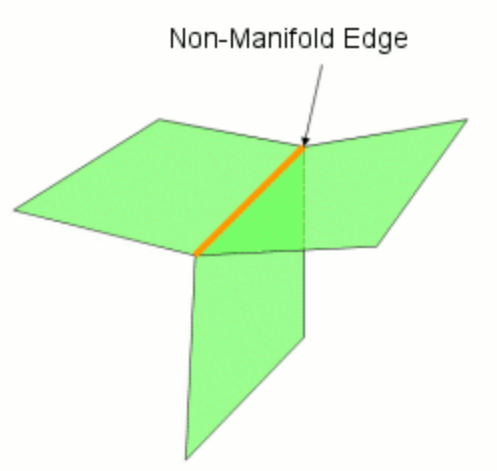



**Figure 1. Topology check for quads and tris**

For hexes and tets it looks for edges with two or more elements connected that do not share common faces.

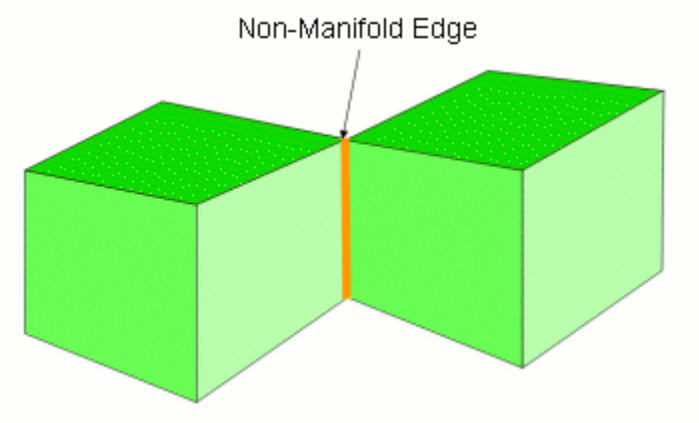



**Figure 2. Topology check for hexes and tets**

Additional topology checks fall into three categories:

- - model edges
- - coincident nodes
- - coincident quadrilateral(faces) or triangles

**Model Edge Check**

The model edge check will find edges with adjoining quadrilaterals or triangles whose angles between the surface normals exceed a specified value. The default angle is 40 degrees.

The following commands check for model edges:

**Topology check model edge {group|volume|surface|curve} <id_range> [angle <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check model edge {block|sideset|nodeset} <id_range> [angle <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check model edge {hex|tet|face|tri|edge} <id_range> [angle <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

The optional **angle** parameter allows the user to specify a custom angle value against which the check will be performed. The default angle is **40 degrees**.
By default, the command will draw the model edges.
By default, very little information is output to the command line. The optional **verbose** parameter will output a list of the flagged model edges.
By default, the model edges will be written to the group **'model_edges'**. Optionally, the user may specify no grouping, or the user may specify the name or id of an existing group into which the model edges will be written. The contents of the existing group will be replaced by the model edges.

**Interface Checks**

Cubit will verify the interfaces between sections of a model. The existence of coincident nodes, for example, may not necessarily be an error in the model if the nodes are in sliding contact or are constrained by some type of multi-point constraint. The existence of coincident quadrilaterals or triangles may indicate that the model is not correctly joined. The following commands check for coincident nodes.

**Topology check coincident node {group|volume|surface|curve|vertex} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident node {block|sideset|nodeset} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident node {hex|tet|face|tri|edge|node} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

The optional **tolerance** parameter allows the user to specify a custom tolerance value against which the check will be performed. The **default tolerance is 1.0 e-6**.
The default group name is **'coincident_nodes**.'
All other options behave similarly to those described above under Model Edge Check.
The following commands check for coincident quadrilaterals.

**Topology check coincident quad {group|volume|surface} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident quad {block|sideset|nodeset} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident quad {hex|tet|face} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

The default group name is '**coincident_quads**.'
All other optional parameters behave similarly to those described above.
The following commands check for coincident triangles.

**Topology check coincident tri {group|volume|surface} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident tri {block|sideset|nodeset} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

**Topology check coincident tri {hex|tet|face|tri} <id_range> [tolerance <value>] DRAW|nodraw|highlight] [BRIEF|verbose] [RESULT GROUP[{<name>|{<id>}|nogroup]**

The default group name is '**coincident_tris**.'
All other optional parameters behave similarly to those described above.

## Metrics for Triangular Elements

The metrics used for triangular elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range | Reference |
|---|---|---|---|---|
| Maximum Angle | degrees | 60 to 180 | 60 to 90 | 1 |
| Minimum Angle | degrees | 0 to 60 | 30 to 60 | 1 |
| Aspect Ratio | L^0 | 1 to inf | 1 to 3 | |
| Aspect Ratio Alpha | L^0 | 1 to inf | 1 to 3 | |
| Condition No | L^0 | 1 to inf | 1 to 1.3 | 2 |
| Deviation | L^2 | 0 to inf | None | |
| Scaled Jacobian | L^0 | -1 to 1 | 0.2 to 1 | 2 |

| | | | | |
|---|---|---|---|---|
| Relative Size | L^0 | 0 to 1 | 0.25 to 1 | 3 |
| Shape | L^0 | 0 to 1 | 0.25 to 1 | 3 |
| Shape and Size | L^0 | 0 to 1 | 0.25 to 1 | 3 |
| **High Order Metrics** | | | | |
| Distortion | L^2 | -1 to 1 | 0.6 to 1 | 4 |
| Element Area | L^2 | 0 to inf | None | 1 |
| Normalized Inradius | L^2 | 0 to 1 | None | |

*Approximate Triangular Quality Definitions:*

**Maximum Angle**: Maximum included angle in triangle
**Minimum Angle**: Minimum included angle in triangle
**Aspect Ratio**: Ratio of circumcirle to inradius
**Aspect Ratio Alpha**: Ratio of longest edge length to inradius
**Condition No**: Condition number of the Jacobian matrix
**Deviation**: Absolute distance from triangle centroid to associated surface
**Scaled Jacobian**: Minimum Jacobian divided by the lengths of 2 edge vectors
**Relative Size**: Min( J, 1/J ), where J is determinant of weighted Jacobian matrix
**Shape**: 2/Condition number of weighted Jacobian matrix
**Shape & Size**: Product of Shape and Relative Size
**Distortion**: {min(|J|)/actual area}*parent area, parent area = 1/2 for triangular element
**Element Area**: (1/2) * Jacobian at corner node
**Normalized Inradius**: 4.0 * minimum_subtri_inradius / radius of circle containing three corner nodes

*Comments on Algebraic Quality Measures*

Relative Size, Shape, and Shape & Size are algebraic metrics, which have well behaved properties. Cubit encourages the use of these metrics over other metrics. These metrics are referenced to an ideal element which, in the case of triangular elements, is an equilateral triangle. Thus deviations from an equilateral triangle are measured in various ways by the algebraic metrics.
Relative size measures the size of the element vs. the size of reference element. If the element is twice or one-half the size of the reference element, the relative size is one-half. By default, the size of the reference element is the average size of all the elements that the quality command is currently evaluating.
The shape and size metric measures how both the shape and relative size of the element deviate from that of the reference element.

*References for Triangular Quality Measures*

1. Traditional.

2. Knupp, 2000.
3. P. Knupp, Algebraic Mesh Quality Metrics for Unstructured Initial Meshes, submitted for publication.
4. SDRC/IDEAS Simulation: Finite Element Modeling--User's Guide

## Metrics for Wedge Elements

The metrics used for wedge elements in CUBIT are summarized in the following table:

| Function Name | Dimension | Full Range | Acceptable Range |
|---|---|---|---|
| Aspect Ratio | L^0 | 1 to inf | 1 to 3 |
| Element Volume | L^3 | -inf to inf | None |
| Condition No | L^0 | 1 to inf | 1 to 3 |
| Jacobian | L^3 | -inf to inf | None |
| Scaled Jacobian | L^0 | -1 to 1 | 0.2 to 1 |
| Shape | L^0 | 0 to 1 | 0.2 to 1 |
| Distortion | L^0 | -1 to 1 | 0.6 to 1 |

*Wedge Quality Definitions*

With a few exceptions, as noted below, Cubit supports quality metric calculations for linear wedge elements only. When calculating quality metrics, that only support linear elements, for a higher order wedge element, Cubit will only use the corner nodes of the element.
**Aspect Ratio**: Maximum edge length ratios at the wedge center
**Element Volume**: Calculated by dividing the wedge into 11 tetrahedron and summing the volume of each.
**Condition No.**: Condition number of the Jacobian matrix at any corner
**Jacobian**: Minimum pointwise volume at any corner. Cubit also supports Jacobian calculations for Wedge21elements.
**Scaled Jacobian**: For linear elements the minimum Jacobian divided by the lengths of 3 edge vectors
**Shape**: 3/Mean Ratio of weighted Jacobian Matrix
**Distortion**: {min(|J|)/actual volume}*parent volume

## Metrics supporting higher-order element types

The following tables details the quality metrics that support some higher order element types:

*Edges*

| Function Name | Higher order types supported |
|---|---|
| Length | All |

*Triangles*

| Function Name | Higher order types supported |
|---|---|
| Distortion | tri6 |
| Element Area | All |
| Normalized Inradius | tri6 |

*Quadrilateral*

| Function Name | Higher order types supported |
|---|---|
| Distortion | quad8 |
| Element Area | All |

*Tetrahedron*

| Function Name | Higher order types supported |
|---|---|
| Distortion | tetra10 |
| Element Volume | All |
| Inradius | tetra10 |
| Jacobian | tetra15 |
| Normalized Inradius | tetra10 |
| Mass Increase Ratio | tetra10 |
| Mean Ratio | tetra10 |
| Timestep | tetra10 |

*Hexahedron*

| Function Name | Higher order types supported |
|---|---|
| Distortion | hex20 |
| Element Volume | All |
| Jacobian | hex27 |

## Measuring Number of Tets Through the Thickness

The ability to check the number of tets through the thickness is given with the following command.

**Quality Surface <surf1_id> <surf2_id> num_thru_thickness**

Finding Intersecting Mesh

The **find mesh intersection** capability finds intersecting mesh between blocks, bodies, volumes, or surfaces. This command is useful for identifying cases where the geometry does not intersect but the mesh does. The command can find intersecting 2-dimensional mesh by specifying a list of surfaces, or 3-dimensional mesh by specifying blocks, bodies, or volumes. Surfaces that have mesh between them that *intersects* within a tolerance of 1e-6 are located. Finding surfaces with intersecting mesh is done using the command:

**Find Mesh Intersection {Block|Body|Surface|Volume} <id_list> [with {Block|Body|Surface|Volume} <id_list>] [low <value=0.0001>] [high <value>] [exhaustive] [worst <num_worst>] [draw] [log] [group<'name'>]**

# *Finding Intersecting 2D Mesh*

To find intersections between 2-dimensional mesh surfaces must be specified. If intersections are found, the surfaces containing the intersecting mesh are drawn (Figure 1) and the put into a group named 'surf_intersect', unless the user has specified another name using the **group <'name'>** option. Also, the ids of the intersecting surface pairs are printed to the terminal,

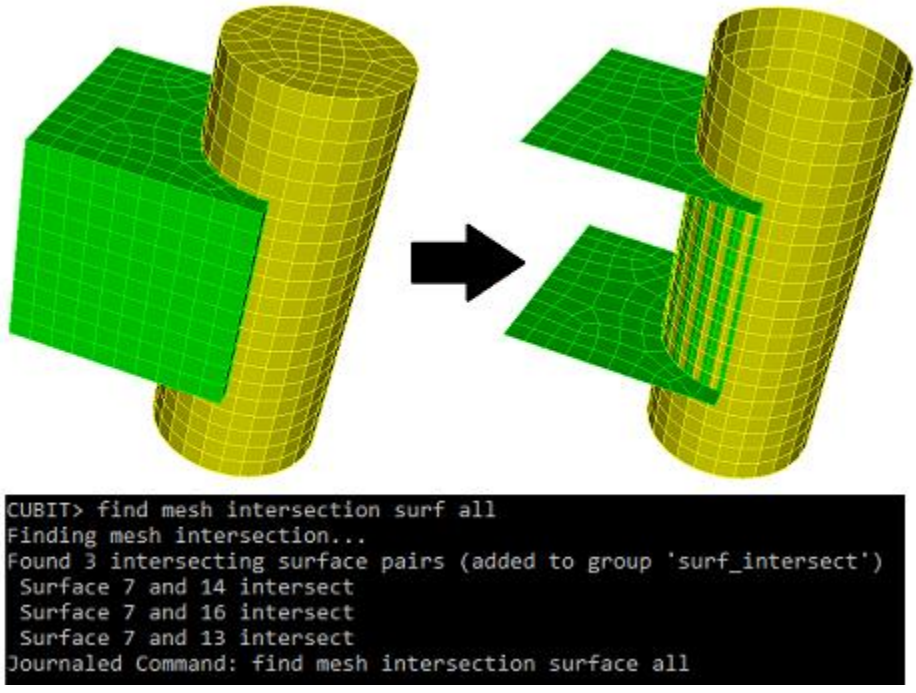



**Figure 1. "Find mesh intersection surface all"**

The **draw** option will draw the surfaces and intersecting mesh in wire frame mode, allowing the user to see exactly where on the surface the mesh intersection is (Figure 2).

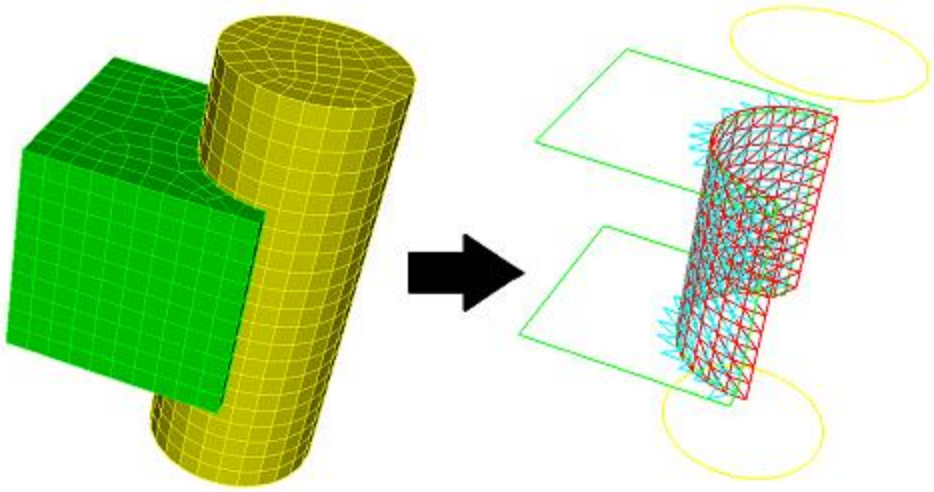


**Figure 2. "Find mesh intersection surface all draw"**

*Facetted Representation*

Detecting mesh intersections between surfaces works entirely off of the mesh, converting the mesh into triangular facets. (The facetted representation is what you see in a shaded view in the graphics). For example, a quad is split into two triangles. Higher order 2D elements are split into multiple triangles.

*Drawing Mesh Intersection*

The command below will draw the mesh intersection for only a pair of surfaces. The surfaces and intersecting mesh are drawn in wire frame mode, allowing the user to see exactly where on the surface the mesh intersection is.

**Draw Surface <id> <id> mesh intersection [add] [include_volume]**

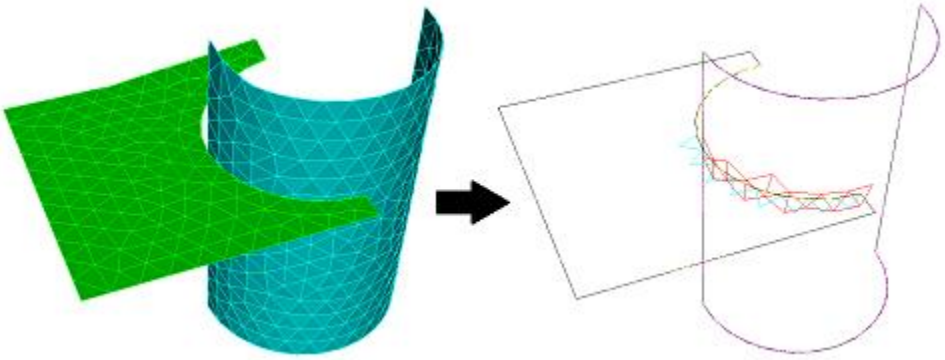
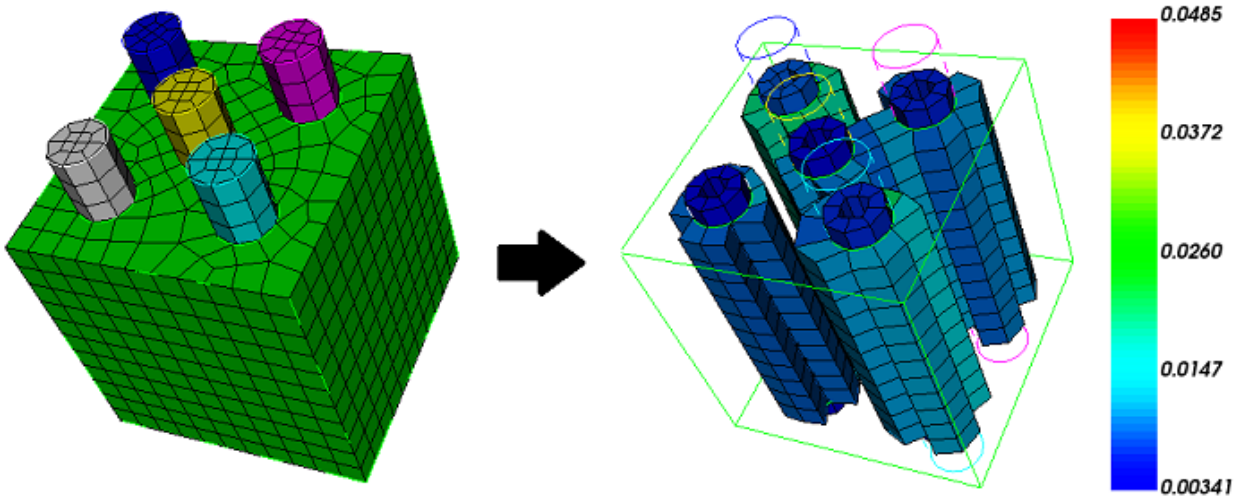


**Figure 3. Draw Mesh Intersection**

# *Finding Intersecting 3D Mesh*

With 3-dimensional entities mesh element intersections can be located by specifying entities: blocks, bodies, or volumes. If intersections are found the intersecting elements are put into a group named 'mesh_intersect' unless the user has specified another name using the **group <'name'>** option. Data is printed to the terminal detailing the intersections. The largest intersection value is reported for each pair of intersecting entities (blocks, volumes, or bodies). This value is the fraction of an element's volume (which element belongs to the entity in the first column) that intersects elements belonging to the entity in the second column. See Figure 5 below. The information printed in columns from left to right is:

- the entities that have intersecting mesh
- the other entities with which that entity's mesh intersects
- the highest intersection value of an element belonging to the entity in column one
- the id of that element
- the number of elements of the entity in column one intersecting other elements of the entity in column two




```
CUBIT> find mesh intersection block  all
   Block     Intersecting with Block   % Worst Intersection   Element Id   #Elements
   -------   -----------------------   --------------------   ----------   ---------
   Block 5   Block 1                   0.048493               2006         104
   Block 4   Block 1                   0.033280               1754         104
   Block 3   Block 1                   0.030450               1538         104
   Block 1   Block 4                   0.024319               701          88
             Block 5                   0.021979               462          88
             Block 6                   0.012961               685          88
             Block 3                   0.012175               896          88
             Block 2                   0.011416               1228         88
   Block 2   Block 1                   0.020682               1430         104
   Block 6   Block 1                   0.020682               2198         104
Found 960 intersecting intersecting elements (added to group 'mesh_intersect')
```

**Figure 5. "Find mesh intersection block all draw"**

If the **with** option is used, the user specifies additional entities. The additional entities will not be reported in the first column of the output. This allows the user to focus on entities of interest without outputting too much data to the terminal. See Figure 6 below.

```
CUBIT> find mesh intersection block 1 with block all
   Block     Intersecting with Block   % Worst Intersection   Element Id   #Elements
   -------   -----------------------   --------------------   ----------   ---------
   Block 1   Block 4                   0.024319               701          88
             Block 5                   0.021979               462          88
             Block 6                   0.012961               685          88
             Block 3                   0.012175               896          88
             Block 2                   0.011416               1228         88
Found 440 intersecting intersecting elements (added to group 'mesh_intersect@A')
```

**Figure 6. "Find mesh intersection block 1 with block all"**

The **low** and **high** options set how much cumulative intersection should be detected. A low value of 0.1 would ignore elements that do not intersect more than 10% of their volume. Similarly, a high value of 0.5 would discard elements that intersect more than 50% of their volume. Both low and high can be used simultaneously. The default for the **low** value is 0.0001

The **exhaustive** option examines all elements for intersection. The default is to only examine elements with nodes on the boundary of the specified entities, anticipating that the intersections will occur mostly at boundaries.

The **worst** parameter limits the printout to the 'n' worst entities with elements of the highest intersections. The intersection fraction reported here is the cumulative intersection an element has with elements of all other entities in the check.

The **draw** option draws the intersecting elements using a color spectrum, red corresponding to high intersection and green to low. The color is according to cumulative intersection, as described in the **worst** option.

If the **log** option is specified, the output from the command will also be sent to a file named "mesh_intersection01.txt", with the number used in the file name incremented as needed. This becomes useful when you have hundreds of volumes with intersections.

**Note:**

- The shape of higher-order mesh elements is considered when computing intersection.
- 3D mesh intersection detection works on free mesh, as long as it has been placed into blocks.
- An intersection fraction greater than 100% is possible, when multiple elements in different entities intersect an element by more than its volume
- For removing mesh intersection, click here.

# mesh modification

## Mesh Modification


- Mesh Smoothing
- Mesh Refinement
- Mesh Scaling
- Mesh Pillowing
- Mesh Coarsening
- Mesh Cleanup
- Mesh Intersection Removal
- Node and Nodeset Repositioning
- Collapsing Mesh Edges
- Align Mesh
- Creating and Merging Mesh Elements
- Matching Tetrahedral Meshes
- Remeshing


After meshing is completed, it may be desirable to change features of the mesh without remeshing the whole volume. Mesh modification methods include tools for improving mesh quality, repositioning mesh elements, or changing mesh density. These methods can be applied on the whole model, or on small sections of the model without requiring remeshing the geometry, and without modifying the underlying geometry.

## mesh smoothing

*Adjust Boundary Orthogonal*

**Applies to**: Surface Meshes
**Summary**: This smoother creates a near orthogonal grid and optionally will make an orthogonal grid if the geometry permits.
**Syntax**:

**Adjust Boundary [Orthogonal] {Surface|Group} <id_range> [Iterations <val>] [snap_to_normal [curve <id>] [fixed curve <id>]]**

**Discussion**:
Adjust Boundary Orthogonal iteratively applies the centroidal area pull algorithm with free boundary nodes. This approximates the affects of an elliptical smoothing algorithm. This algorithm works best with mapped meshes which have an element aspect ratio close to 1. The **snap_to_normal** option is not allowed for non-mapped meshes.

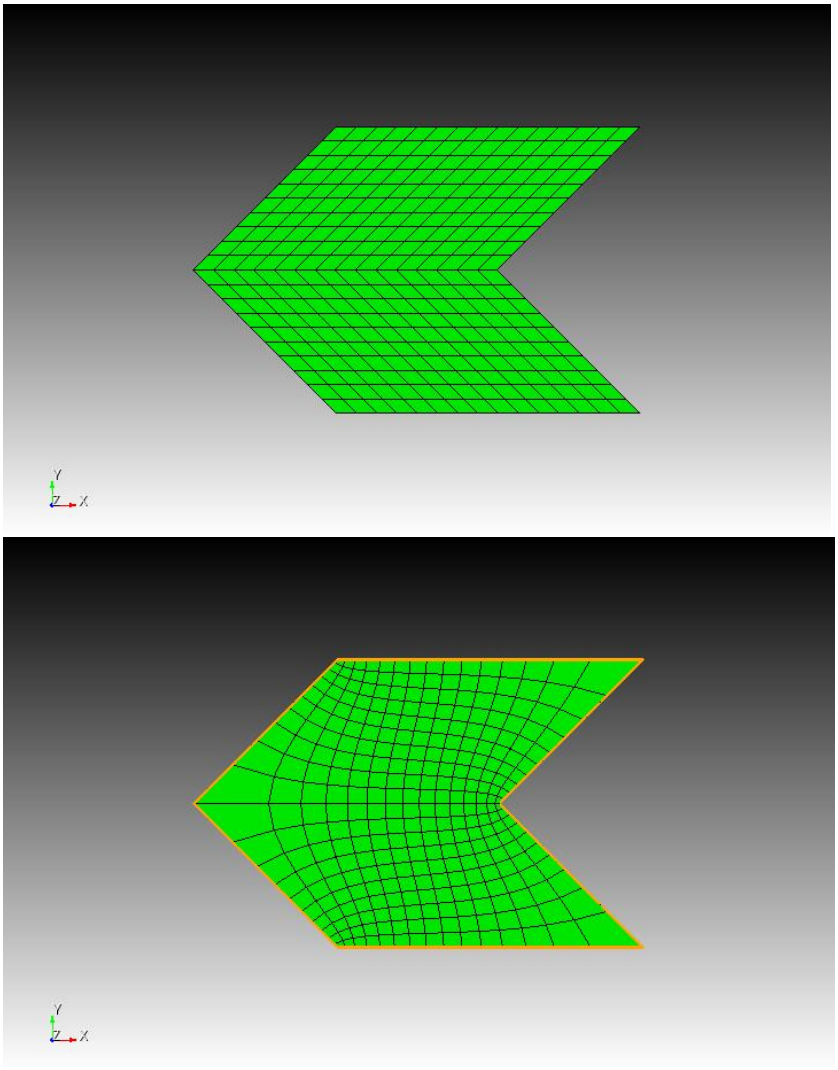


**Figure 1. The affect of the "adjust boundary orthogonal surface 1" on a chevron shape. Note that the nodes are pulled into the acute angles and the edges at the boundary are pulled into a position that is closer to perpendicular at the boundary.**

With some geometries with a mapped mesh it is possible to draw a line that is orthogonal to a boundary curve along the entire u or v direction of the mesh. In these cases, this command optionally allows the user to specify the option **snap_to_normal**. Nodal lines will be created normal to the first curve this is found that will allow perpendicular element edges to span the mesh. The user may optionally specify a curve that is used as the perpendicular basis for projecting the edges.
An edge may also be set as fixed so that a subsequent adjust boundary orthogonal will not affect that edge. If both **snap_to_normal** and **fixed** are set, the curve ids **MUST** be identical.

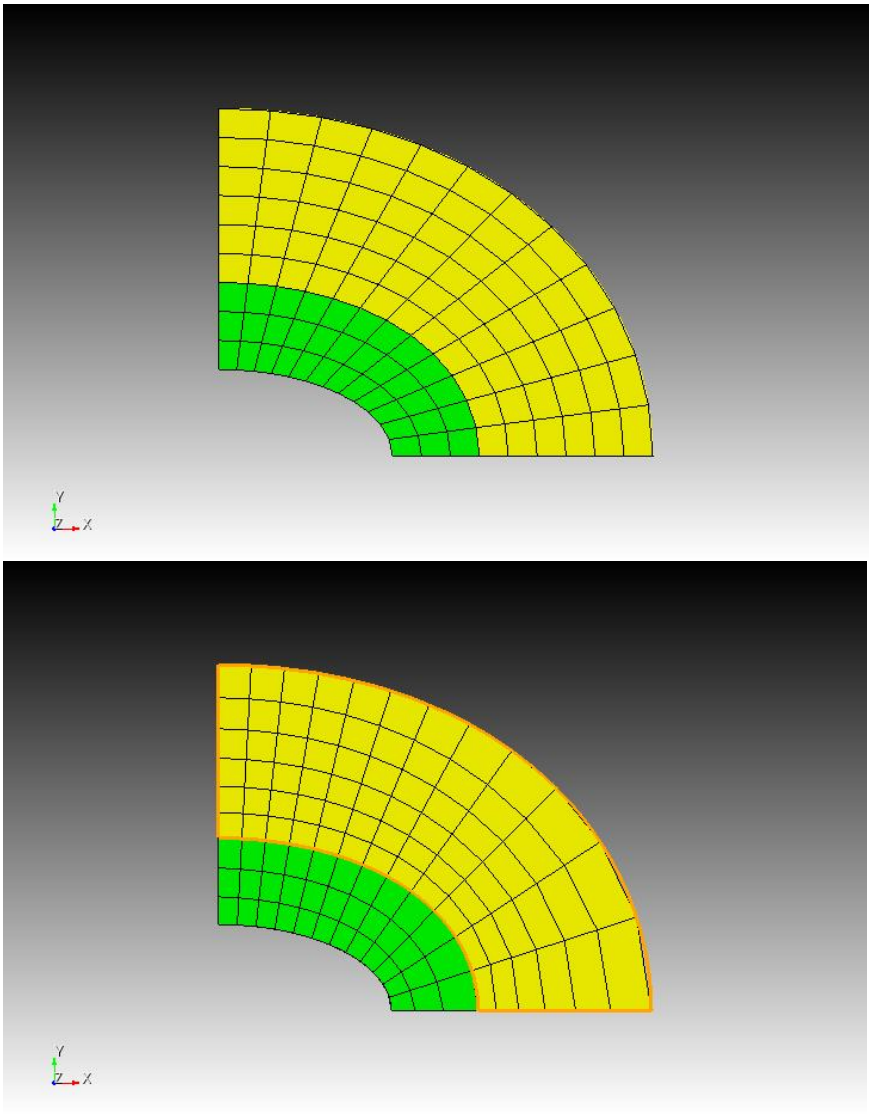


**Figure 2. The affect of adjust boundary orthogonal with the snap to normal curve option is shown. The resulting mesh is orthogonal to the given boundary and projects straight through the mesh.**

The following is an example of how to use this command to create the desired grid in Cubit. Note that to get the desired orthogonal grid the user must adjust the surfaces one at a time.

```
reset
create surface ellipse major radius 2 minor radius 1 zplane
imprint volume 1 with position 0 1 0
create curve offset curve 2 distance 1 extended
create curve offset curve 4 distance 2 extended
create surface skin curve 2 4
create surface skin curve 4 5
delete surface 1
```

merge all surface all scheme map mesh surf all adjust boundary orthogonal surface 2 snap_to_normal curve 6 adjust boundary orthogonal surface 3 snap_to_normal curve 4 fixed curve 4

*Centroid Area Pull*

**Applies to**: Surface Meshes
**Summary**: Attempts to create elements of equal area
**Syntax**:

**Surface <range> Smooth Scheme Centroid Area Pull [Free]**

**Discussion**:
This smooth scheme attempts to create elements of equal area. Each node is pulled toward the centroids of adjacent elements by forces proportional to the respective element areas (Jones, 74).

*Condition Number*

**Applies to**: Triangular or Quadrilateral Surface Meshes, Tetrahedral or Hexahedral Volume Meshes. Does not apply to Mixed Element Meshes.
**Summary**: Optimizes the mesh condition number to produce well-shaped elements.
**Syntax**:

**Surface <surface_id_range> Smooth Scheme Condition Number [beta <double=2.0>] [cpu <double=10>]**

**Related Commands:**
Untangle
**Discussion**:
The condition number smoother is designed to be the most robust smoother in Cubit because it guarantees that if the initial mesh is non-inverted then the smoothed mesh will also be non-inverted. The price exacted for this capability is that this smoother is not as fast as some of the other smoothers.
Condition Number measures the distance of an element from the set of degenerate (non-convex or inverted) elements. Optimization of the condition number increases this distance and improves the shape quality of the elements. Condition number optimization requires that the given mesh contain no negative Jacobians. If the mesh contains negative Jacobians and this command is issued, Cubit automatically calls the Untangle smoother and attempts to remove the negative Jacobians. If successful, condition number smoothing occurs next; the resulting mesh should have no negative Jacobians. If untangling is unsuccessful, condition number smoothing is not performed.
There is no "fixed/free" option with this command; boundary nodes are always held fixed.

The command above only sets the smoothing scheme; to actually smooth the mesh one must subsequently issue the command "smooth surface <surface_id_range>" or "smooth volume <volume_id_range>".
Stopping Criteria: Smoothing will proceed until the objective function has been minimized or until one of two user input stopping criteria are satisfied. To input your own stopping criterion use the optional parameters 'beta' and 'cpu' in the command above. The value of beta is compared at each iteration to the maximum condition number in the mesh. If the maximum condition number is less than the value of beta, the iteration halts. In Cubit condition number ranges from 1.0 to infinity, with 1.0 being a perfectly shaped element. Thus the smaller the maximum condition number, the better the mesh shape quality. The default value of the beta parameter is 2.0. The value supplied for the "cpu" stopping criterion tells the code how many minutes to spend trying to optimize the mesh. The default value is 10 minutes. Optimization may also be halted by using "control-C" on your keyboard.
To view a detailed report of the smoothing in progress issue the command "set debug 91 on" prior to smoothing the surfaces or volumes. You will get a synopsis of whether or not untangling is needed first and whether the stopping criteria have been satisfied. In addition the following printout information is given for each iteration of the conjugate gradient numerical optimization:

**Iteration=n, Evals=m, Fcn=value1, dfmax=value2, time=value3 ave_cond=value4, max_cond=value5, min_jsc=value6**

**n** is the iteration count, **m** is the number of objective function evaluations performed per iteration, **value1** is the value of the objective function (this usually decreases monotonically), **value2** is the norm of the gradient (does not always decrease monotonically), and **value3** is the cumulative cpu time (in seconds) spent up to the current iteration. The minimum possible value of the objective function is zero but this is attained only for a perfect mesh. **ave_cond**, **max_cond**, and **min_jsc** are the average and maximum condition number, and the minimum scaled jacobian. **ave_cond** generally decreases monotonically because it is directly related to **value1**.

*Edge Length*

**Applies to**: Surfaces
**Summary**: This smoother tries to make all edge lengths equal
**Syntax**:

**Surface <range> Smooth Scheme Edge Length**

**Discussion**:
Edge Length smoothing in Cubit is provided by MESQUITE, a mesh optimization toolkit by Argonne National Laboratory and Sandia National Laboratories. (See Brewer, et al. 2003 for more details on the MESQUITE toolkit.) This smooth scheme may be useful for lengthening the shortest edge length in paved meshes.

Interior node positions are adjusted in an optimization loop where the optimal element has an ideal shape (square) and has an area equal to the average element area of the input mesh.
NOTE: This smoother should be avoided when the mesh contains high aspect-ratio elements that the user wants to keep.
Because this smoother essentially tries to make all the edge lengths equal, it is designed to work well on meshes whose elements have aspect ratios close to 1. The farther from 1 the aspect ratio is, the less applicable this smoother will be.

*Equipotential*

**Applies to**: Volume Meshes
**Summary**: Attempts to equalize the volume of elements attached to each node
**Syntax**:

```
Volume <range> Smooth Scheme Equipotential [Free]
```

**Discussion**:
This smoother is a variation of the Equipotential (Jones, 74) algorithm that has been extended to manage non-regular grids (Tipton, 90). This method tends to equalize element volumes as it adjusts nodal locations. The advantage of the equipotential method is its tendency to "pull in" badly shaped meshes. This capability is not without cost: the equipotential method may take longer to converge or may be divergent. To impose an equipotential smooth on a volume, each element must be smoothed in every iteration--a typically expensive computation. While a Laplacian method can complete smoothing operations with only local nodal calculations, the equipotential method requires complete domain information to operate.

*Laplacian*

**Applies to**: Curve, Surface, and Volume meshes
**Summary**: Tries to make equal edge lengths
**Syntax**:

```
{Surface|Volume} <range> Smooth Scheme Laplacian [Free] [Global]
```

**Discussion**:
The length-weighted Laplacian smoothing approach calculates an average element edge length around the mesh node being smoothed to weight the magnitude of the allowed node movement (Jones, 74). Therefore this smoother is highly sensitive to element edge lengths and tends to average these lengths to form better shaped elements. However, similar to the mapping transformations, the length-weighted Laplacian formulation has difficulty with highly concave regions.

Currently, the stopping criterion for curve smoothing is 0.005, i.e., nodes are no longer moved when smoothing moves the node less than 0.005 * the minimum edge length. The maximum number of smoothing iterations is the maximum of 100 and the number of nodes in the curve mesh. Neither of these parameters can currently be set by the user.
Using the **global** keyword when smoothing a group of surfaces will allow smoothing of mesh on shared curves to improve the quality of elements on both surfaces sharing that curve.

*Mean Ratio*

**Applies to**: Triangular or Quadrilateral Surface Meshes, Tetrahedral or Hexahedral Volume Meshes. Does not apply to Mixed Element Meshes.
**Summary**: Moves interior mesh nodes to optimize the average mean ratio metric value of the mesh.
**Syntax**:

**Surface <surface_id_range> Smooth Scheme Mean Ratio [cpu <double=10>]**

**Volume <volume_id_range> Smooth Scheme Mean Ratio [cpu <double=10>]**

**Discussion**:
CUBIT includes a mean ratio smoother provided by MESQUITE, a mesh optimization toolkit by Argonne National Laboratory and Sandia National Laboratories. (See Brewer, et al. 2003 for more details on the MESQUITE toolkit.) This smoother is similar in purpose to the Condition Number smoother. However, the Mean Ratio smoother uses a second order optimization method, and therefore it will often reach a near-optimal mesh more quickly than the Condition Number smoother. The Mean Ratio smoother requires the initial mesh to be untangled, but the smoother is guaranteed to not tangle the mesh. If the user attempts to call the Mean Ratio smoother on a tangled mesh, an untangler will first attempt to untangle the mesh before calling the Mean Ratio smoother.
The Mean Ratio smoother's optimization process terminates when one of the following three criteria is met:

1. The mesh is "close" to an optimal mesh configuration.
2. The maximum allotted time has been exceeded.
3. The user interrupts the smoothing process.

The user has control over the second and the third criteria only. For criterion 2, the default is for the smoother to terminate after ten minutes even if a near-optimal mesh has not been reached. The user can change this time bound by specifying the optional "cpu" argument in the command listed above. This argument takes a single, positive number that represents the time (in minutes) that will be used as a time bound. If the user wishes to terminate the process early, criteria three allows the user to "interrupt" (for example, on some platforms, by pressing CTRL-C) the process. If the process is terminated early,

the mesh will not revert to the original node positions; CUBIT will instead keep the partially optimized mesh.

*Mesh Smoothing*

- Centroid Area Pull
- Equipotential
- Laplacian
- Smart Laplacian
- Condition Number
- Mean Ratio
- Winslow
- Untangle
- Edge Length

**Related Topics**

- Smoothing mesh-based geometry
- Smoothing free meshes

After generating the mesh, it is sometimes necessary to modify that mesh, either by changing the positions of the nodes or by removing the mesh altogether. CUBIT contains a variety of mesh smoothing algorithms for this purpose. Node positions can also be fixed, either by specific node or by geometry entity, to restrict the application of smoothing to non-fixed nodes.
Mesh smoothing in CUBIT operates in a similar fashion to mesh generation, i.e. it is a two-step process whereby a smooth scheme is chosen and set, then a smooth command performs the actual smoothing. Like meshing algorithms, there is a variety of smoothing algorithms available, some of which apply to multiple geometry entity types and some which only apply to one specific type (these algorithms are described below.) To smooth the mesh on a geometry entity, the user must perform the following steps:

1. Set the smooth scheme for the object using the following command:

   **{Curve|Surface|Volume} <range> Smooth Scheme <scheme>**

where **<scheme>** is any acceptable smooth scheme described in this section. Also set any scheme-specific information, using the smooth scheme setting commands described below.

2. Smooth the object, using the command:

   **Smooth Curve <range>**

   **Smooth Surface <range> [Global]**

**Smooth {Body|Volume|Group} <range>**

Groups of entities may be smoothed, by smoothing a group or a body.
If a Body is specified, the volumes in that Body are smoothed. If a Group is specified, only the volume meshes within these groups are smoothed - no smoothing of the surface meshes is performed.

#### Global Smoothing

When smoothing a set of surfaces, the keyword **global** can be added to the smooth command such as

**Smooth Surface <range> [Global]**

If the smoothing algorithm for two neighboring surfaces are both allowed to move boundary nodes, then appending the "global" keyword will often result in a higher quality mesh near the curve(s) shared by those two surfaces.

#### Focused Smoothing on Groups of Mesh Entities

Meshed entities such as hexes or tris can be smoothed individually or in groups by specifying the entities in a list.

**Smooth {Hex|Tet} <range>[Scheme {Equipotential|Laplacian|Random}]**

**Smooth {Face|Tri} <range>[Scheme {Laplacian|Centroid|Winslow}] [Target Surface <id>]**

**Smooth Edge <id_range> [Scheme Laplacian] [Target Surface <id>]**

The **Smooth Edge** command allows the user to smooth individual edges owned by a curve. Specifying a target curve allows the user to move the edges on a meshed curve to a different curve. The target curve or surface does not necessarily need to be the owning curve or surface of the nodes. For example, if given two curves (A and B) and curve A was meshed, the target smoothing could be used to move all of the edges of curve A onto curve B. The smooth scheme option for the edge smoothing is currently limited only to the laplacian scheme.
The **Smooth Face|Tri** command is used to smooth individual faces or triangles. The target option is similar to the curve target option above. Faces or Tris can be smoothed to a surface that is not necessarily the owning surface; in fact, the faces or tris do not even have to be attached to any surface. This makes this option especially helpful for smoothing free meshes. Specifying a smooth scheme allows for relaxation based surface smoothers (i.e. centroid area pull, laplacian, winslow) to be utilized during targeted smoothing. It is not currently enabled for optimization based smoothing schemes.

#### Smooth Tolerance

Smoothing algorithms move nodes in an attempt to improve the quality of the mesh elements. Most of these algorithms are iterative, and the algorithm terminates when some

criterion is met. Specifically, for the Laplacian and Equipotential style smoothers, smoothing is terminated either by satisfying a smoothing tolerance or by performing the maximum number of smoothing iterations. For these smoothers, the smooth tolerance may be set by the user:

**[Set] Smooth Tolerance <tol>**

The value **<tol>** tells the smoother to stop when node movement is less than **tol * local_minimum_edge_length**.
The default value for tol is 0.05. The maximum number of iterations may be set by the user. For volumes, the smooth tolerance and iterations may also be set by
(Note: The above command affects all smoother that respect tolerance.)

**Volume Smooth Tolerance <tol>**

**Volume Smooth Iterations <iters>**

(Note: The above two commands only affect the volume smoothers.)

#### Boundary Mesh Smoothing

Where used in the smooth schemes below, the Free keyword permits the nodes lying on the bounding entities to "float" along those entities; without this keyword, boundary nodes remain fixed.
Nodal positions may be fixed so that no smoothing scheme, either implicit or explicit, will move them, with the following command:

**{Curve|Surface|Volume} <range> Node Position {Fixed|Free}**

**Node <range> Position {Fixed|Free}**

The following command does not fix nodal positions, but does fix the connectivity of the mesh, preventing certain volume schemes from changing the bounding mesh:

**{Curve|Surface|Volume} Mesh {Fixed|Free}**

The additional following scheme is available for research purposes and can be used only after issuing a **'set developer on'** command.

- Randomize

*Smart Laplacian*

**Applies to**: Surface and Volume meshes

**Summary**: Tries to make equal edge lengths while ensuring no degradation in element shape
**Syntax**:

**{Surface|Volume} <range> Smooth Scheme Smart Laplacian**

**Discussion**:
The Smart Laplacian smoothing approach is a variation on the standard Laplacian algorithm. The algorithm iteratively loops over the mesh and updates nodes based on the location of their neighbors. First, a patch of elements is formed around a given node. The quality of this patch is assessed to determine the quality of the worst shaped element. Then a new candidate node position is calculated as the average of the neighboring nodes. The quality of the patch is assessed again using the candidate node position. If there has been no degradation in the quality of the elements in the patch, the candidate node position is accepted; otherwise, the candidate node position is rejected and the node is returned to its previous position.
The Smart Laplacian smoother is intended to provide a reliable smoother that is nearly as fast as the Length-Weighted Laplacian smoother. Due to the dual goals of this smoother, making equal edge length and improving element shape, it will not always be able to make progress. However, it is often useful as a quick alternative to the more time-consuming optimization methods like Mean Ratio or Condition Number. When this smoother fails to make significant progress, the optimization methods can be tried.
The Smart Laplacian Smoother uses the Mean Ratio quality measure to assess element shape. This smoother is ensuring no degradation in the minimum Mean Ratio. The Mean Ratio smoother is optimizing the same metric, but it is attempting to improve the average Mean Ratio quality.

*Untangle*

**Applies to**: Triangular or Quadrilateral Surface Meshes Tetrahedral or Hexahedral Volume Meshes. Does not apply to Mixed Element Meshes.
**Summary**: Removes as many negative Jacobians from the mesh as possible by minimizing a certain objective function.
**Syntax**:

**Surface <surface_id_range> Smooth Scheme Untangle [beta <double=0.02>] [cpu <double=10>]**

**Volume <volume_id_range> Smooth Scheme Untangle [beta <double=0.02>] [cpu <double=10>]**

**Related Commands:**
**Condition Number**
**Discussion**:

The Untangle 'smoother' is designed to eliminate negative Jacobians from a given mesh by moving nodes to appropriate locations. If a mesh node is not involved in causing a negative Jacobian it will not be moved. If a mesh has no negative Jacobians, the Untangler will not move any of the nodes. This smoother is not magic: if an untangled mesh does not exist for the given mesh topology, the untangler will not untangle the mesh. Instead, it will do the best it can and exit gracefully. An untangled mesh produced by this smoother will often have poor shape quality; in that case it is recommended that untangling be followed by condition number smoothing. The untangle smoother is automatically called by the condition number smoother.
There is no "fixed/free" option with this command; boundary nodes are always held fixed. As a result, users should be aware that the volume untangler cannot succeed if the volume contains a surface mesh which contains a negative Jacobian. In that case, one must first remove the surface mesh negative Jacobians by invoking the surface Untangler and then invoke the volume Untangler.
The command above only sets the smoothing scheme; to actually smooth the mesh one must subsequently issue the command "smooth surface <surface_id_range>" or "smooth volume <volume_id_range>".
**Stopping Criteria**: Untangling will proceed until the objective function has been minimized or the optional user input "cpu" has been satisfied. The latter stopping criterion tells the code how many minutes to spend trying to untangle the mesh. The default value is 10 minutes. Optimization may also be halted by using "control-C" on your keyboard.
**Beta Parameter**: An optional user input parameter "beta" plays a role in determining the optimal mesh. Optimization proceeds until the minimum scaled Jacobian of the mesh is (roughly) greater than beta. To remove negative Jacobians one would need beta=0 (however, as a safety margin, we choose beta=0.02 as the default). To further improve the scaled Jacobian of the mesh, input a larger value of "beta". If a mesh with all scaled Jacobians greater than "beta" does not exist, optimization will continue until the cpu time stopping criterion has been met. Therefore, it is best not to use "beta" values too large (say, greater than 0.2) without also decreasing the cpu time limit.
To view a detailed report of the smoothing in progress issue the command "set debug 91 on" prior to smoothing the surfaces or volumes. You will get a synopsis of whether or not untangling is needed and whether the stopping criteria are satisfied. In addition the following printout information is given for each iteration of the conjugate gradient numerical optimization:

**Iteration=n, Evals=m, Fcn=value1, dfmax=value2, time=value3 min_jsc=value4**

**n** is the iteration count, **m** is the number of objective function evaluations performed per iteration, **value1** is the value of the objective function (this usually decreases monotonically), **value2** is the norm of the gradient (does not always decrease monotonically), and **value3** is the cumulative cpu time (in seconds) spent up to the current iteration. The minimum possible value of the objective function is zero; this value is attained only when the minimum scaled Jacobian of the mesh exceeds "beta". The **minimum scaled jacobian** is also reported.

*Winslow*

**Applies to**: Surface meshes
**Summary**: Elliptic smoothing technique for structured and unstructured surface meshes
**Syntax**:

**Surface <range> Smooth Scheme Winslow [Free]**

**Discussion**:
Winslow elliptic smoothing (Knupp, 98) is based on solving Laplaces equation with the independent and dependent variables interchanged. The method is widely used in conjunction with the mapping and submapping methods to give smooth meshes with positive Jacobians, even on non-convex two-dimensional regions. The method has been extended in CUBIT to work on unstructured meshes.

## Align Mesh

At times it is desirable to have identical meshes on two different surfaces or curves. The align mesh command will attempt to assign correspondence between nodes on surfaces or curves and move the nodes on one surface or curve to match the configuration on the other. The command syntax is:

**Align Mesh Surface <id> [CloseTo] Surface <id> [Tolerance <tol>]**

**Align Mesh Curve <id> [CloseTo] Curve <id> [Tolerance <tol>]**

These two commands align the mesh on the first entity with that of the second entity. This means that nodes on the first entity will be moved to the closest location possible to their corresponding nodes on the second entity. This is done without regard to mesh quality, so it is possible to invert elements with this command.

**Align Mesh Node <id> [CloseTo] Node <id> [Tolerance <tol>]**

This command aligns the first node with the second node, within the limits of the geometric entities that own the nodes. This is also done without respect for element quality.
And example of this is given as follows:

**brick x 10**
**volume 1 copy move 11**
**surface all except 10 6 vis off transparent**
**graphics perspective off**
**at 5.552503 3.832384 0.134127**
**from 34.651051 3.640138 -0.193121**
**up 0.006514 0.999945 -0.008172 mesh surface all**
**surface 6 smooth scheme randomize free**
**smooth surface 6**
**node 432 move 0 0 -0.2**

**align mesh node 944 node 432**
**node 432 move 0 0 0.4**
**align mesh curve 23 closeto curve 12**
**align mesh surf 10 closeto surf 6**

## Collapsing Mesh

CUBIT currently offers several options for modifying an existing finite element mesh. Triangle and tetrahedral meshes sometimes contain low quality elements that can be removed by the following operations.

*Collapsing Mesh Using Quality Metrics*

The following first two commands collapse triangles, while the third collapses tetrahedra. Collapses are performed only if the collapse operation does not cause worse quality in the surrounding, surviving mesh; otherwise the collapse is not performed. The collapse operation removes the element by consolidating two neighbor nodes of the specified mesh entity into one, to either node location or their midpoint. The specified metric is used to determine quality, the default being Scaled Jacobian. The **Interior** option of the Collapse Tet command limits the collapse to the interior of the volume mesh, so that the triangle surface mesh is not modified.

**Collapse Edge <ids> [SCALED JACOBIAN|Aspect Ratio|Shape|Shape and Size]**

**Collapse Tri <ids> [SCALED JACOBIAN|Aspect Ratio|Shape|Shape and Size]**

**Collapse Tet <ids> [Interior] [Altitude|Aspect Ratio|Aspect Ratio Gam|Distortion|Inradius|Jacobian|Normalized Inradius|Node Distance|SCALED JACOBIAN|Shape|Shape and Size|Timestep]**

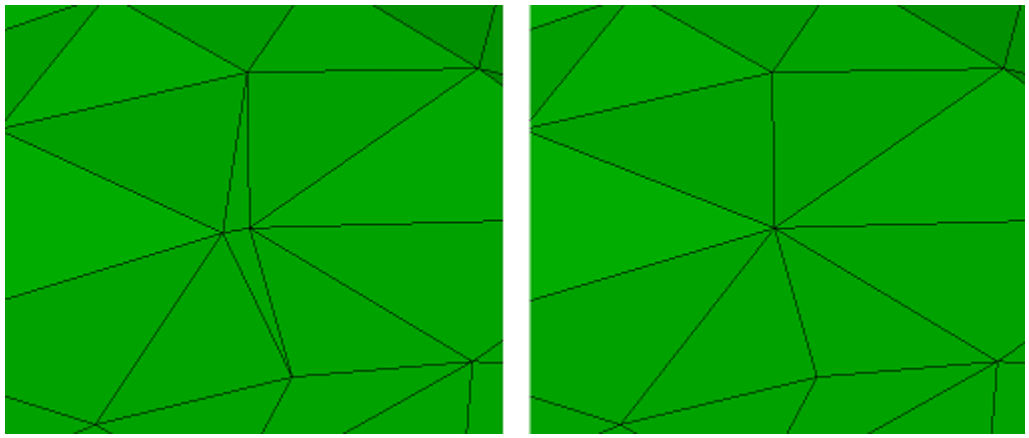


**Figure 1. Before and After Triangle/Edge Collapse**

*Collapsing Mesh Manually*

The following command collapses triangles containing the given edge, with no regard for quality.

**Meshedit Collapse edge <id> [keep node <id>] [compress_ids]**

This command only works on triangle surface meshes. If volumetric elements, or quads, are attached to the edge, the command fails. The **keep node** option control which node to collapse to. The option **compress_ids** compresses holes in the mesh id space caused by the collapse.
Combining nodes to collapse tetrahedra can be done with the following command. The first node specified is merged into (deleted) the second node. If the first node cannot be moved due to geometry constraints (it's owned by a vertex), the operation fails. This command is behind a developer flag so **set dev on** must be issued before use.

**Merge Node <id> <id>**

*Removing Overconstrianed Tets*

Cubit offers commands to remove overconstrained tetrahedra - tets containing two triangles on the same surface and all four corner nodes on surfaces, curves, or vertices. These tets are not amenable to smoothing leaving removal as the only viable solution to improve quality.

**Remove Overconstrained Tet <ids>**

The tets are deleted and the two back sides become surface triangles. If the tets have mid-edge nodes, the back mid-edge nodes are snapped to the surface. If this removal operation generates worse quality (aspect ratio) than in the tet removed, the operation is not performed. Figure 2 shows before and after image of removing overconstrained tets.

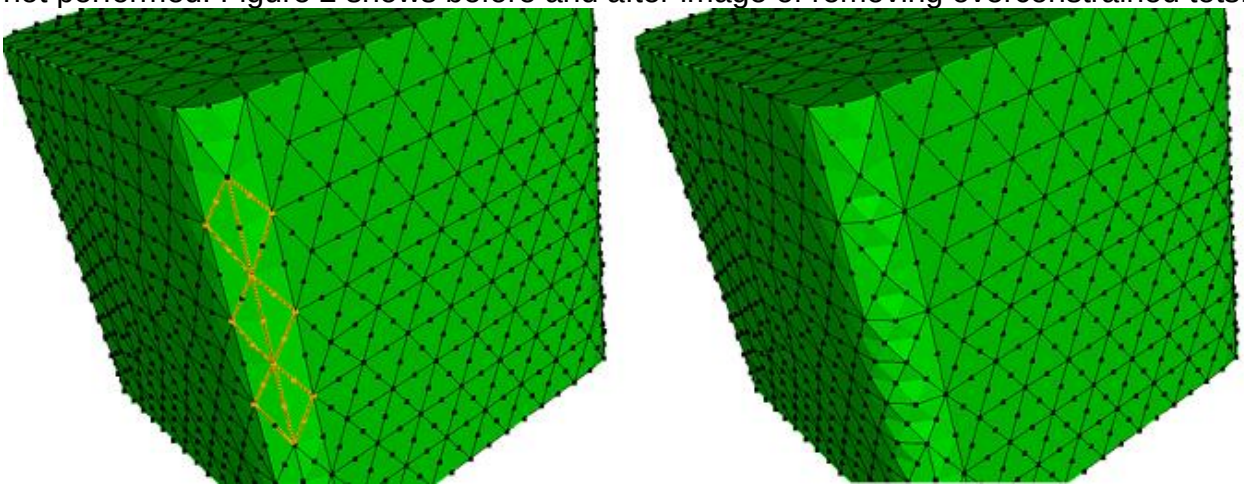


**Figure 2. Before and After Removing Overconstrained Tets**

## Creating and Merging Mesh Elements

The following forms of the create and merge commands operate on meshed entities only. They allow low-level editing of meshes to make minor corrections to a mostly correct mesh. They are not designed for major modifications to existing meshes. Because Cubit's display routines were not designed with these type of operations in mind, these commands may cause the current display of the affected entities to take an unexpected form. An appropriate drawing command can be used to return the display to the desired view.

The delete commands for deleting individual elements are still under development, but they may be used after setting a developer flag.

*Creating Mesh Elements*

The create command uses existing mesh nodes to create new mesh entities.

#### Creating Hex and Tet Elements

**Create {Hex|Tet} Node <range> [Owner Volume <id>]**

Using the nodes specified, this form of the command creates a new hex or tet that will be owned by the specified volume. For a hex, 8 nodes are required. The order in which the nodes are specified is very important. They should describe two opposing faces of the hex; the normal of the first face should point into the hex and the normal of the second face should point out of the hex. For example, to create the hex shown in Figure 1 below, the following command would be entered:

**create hex node 1,2,3,4,5,6,7,8 owner volume 1**

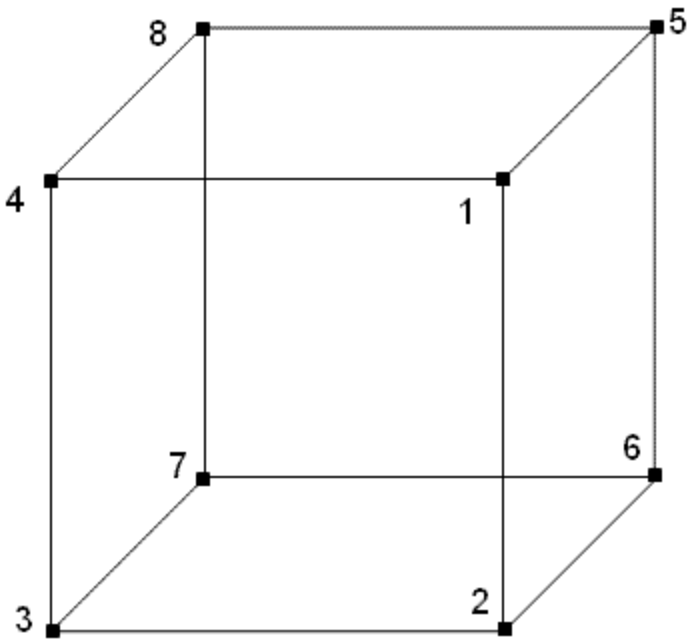



**Figure 1. Node Numbering for the Create Hex command**

To create a tet, 4 nodes are specified. The base is specified as a tri with the normal point toward the fourth node using the right hand rule. To create the tet shown in Figure 2, the following command would be entered:

```
create tet node 1,2,3,4 owner volume 1
```

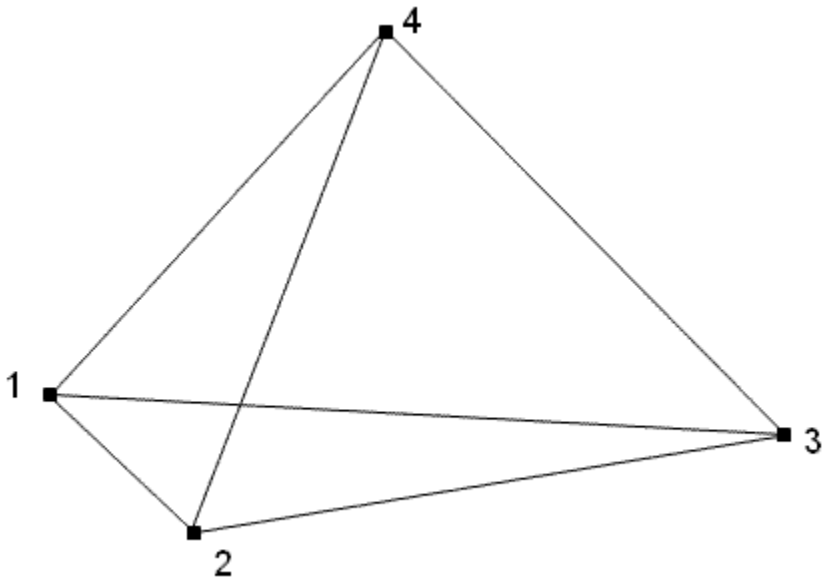



**Figure 2. Node ordering for Create Tet Command**

#### Creating Wedge Elements

**Create Wedge Node <range> [Owner Volume <id>]**

To create a wedge, 6 nodes are specified. The base is specified as a tri with the normal pointing inward using the right hand rule. To create the wedge shown in Figure 3, the following command would be entered:

**create wedge node 1,2,3,4,5,6 owner volume 1**

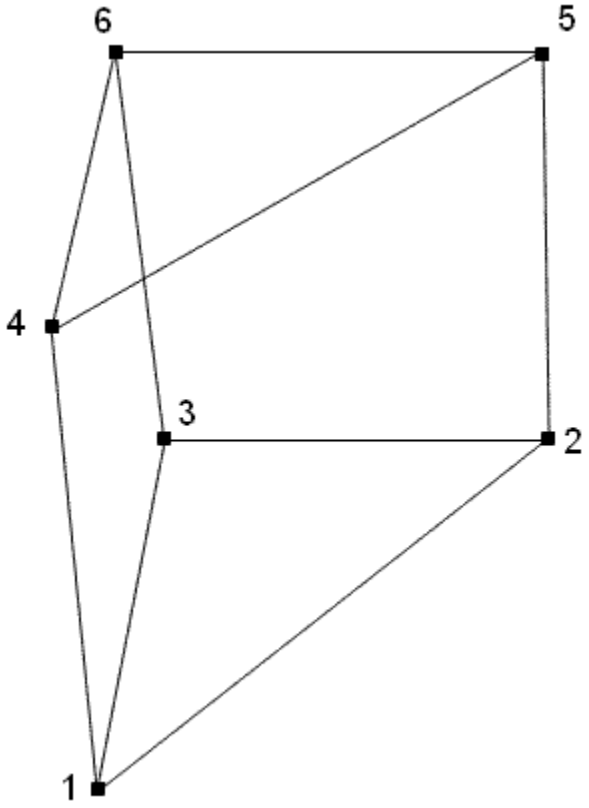



**Figure 3. Node ordering for Create Wedge Command**

#### Creating Face and Tri Elements

**Create {Face|Tri} Node <range> [Owner {Volume|Surface} <id>]**

The next form of the command creates a face or tri that will be owned by the specified volume or surface. Four nodes are specified for a face, three nodes for a tri. The nodes should be specified in the order needed to produce a face or tri with the normal in the desired direction using the right hand rule.

#### Creating Edge Elements

**Create Edge Node <range> [Owner {Volume|Surface|Curve} <id>]**

This form of the command creates an edge that will be owned by the specified volume, surface, or curve. Two nodes must be specified; order is unimportant.

#### Creating Nodes

**Create Node Location <x> <y> <z> Owner {Volume|Surface|Curve|Vertex} <id>**

The last form of the command creates a node at the specified location that will be owned by the specified volume, surface, curve, or vertex. The location is specified by three absolute values that represent the position of the node in 3D space.

*Merging Nodes*

The merge node command is used to join two mesh entities one node at a time. It should be used with care because merging nodes of different meshed entities may have unpredictable results. The syntax is:

**Merge Node <id1> <id2>**

The merge node command replaces the node specified as id1 with the node id2. The command is equivalent to deleting node id1 and creating node id2 in the same location. The resultant merged node takes on the characteristics of the replaced node such as position and owner. This may include some or all of the higher level mesh entities related to the merged node.
Caution should be taken when using the merge node command because other commands involving the related meshed entities may not work properly following the merge.

## Mesh Cleanup

- Tetrahedral Mesh Cleanup
- Hexahedral Mesh Cleanup

Once a mesh has been created or imported, Cubit has tools to both manually and automatically improve the quality of a mesh. Mesh Cleanup is the name for the automatic tools which automatically find bad elements and fixing them by both recomputing node locations (i.e. smoothing) AND redefining the local element connectivity. To automatically cleanup a mesh, use the following command:

**Cleanup {Volume|Block} <id_range> [angle <value=150>]**

This command will cleanup either a tet or a hex mesh as described below.

# Cleaning Up a Tetrahedral Mesh

An alternative to the remesh command for tetrahedral meshes is the cleanup command. For this command the existing mesh is validated and "optimized" by the tetmesher, instead of being deleted and replaced with a different mesh.
To cleanup a tetrahedral volume mesh use the following command:

**Cleanup {Volume|Block} <id_range>**

A second variation of the Cleanup command allows remeshing of tetrahedra that are either part of a free mesh (not owned by a volume) or are a subset of the tetrahedra in the volume. The command is:
**Cleanup Tet <id_range> [Free]**
For example, the command
**cleanup tet all free**
will gather all tetrahedra in a free mesh or single volume, generate a triangle boundary surface, and "optimize" the mesh, ignoring any volume or blocks. Without the optional **free** keyword, the tets will be processed volume by volume or block by block retaining the boundary between adjacent volumes or blocks.
Also, the command
**cleanup tet 200 to 300**
will gather the tetrahedra in the range [200, 300], generate a triangle boundary surface, and "optimize" the mesh. If the tetrahedra in the range are disjointed, i.e., multiple, independent sets, this operation may fail. It is best to specify a contiguous set of elements.
**Note:** Cubit will issue an error if the tetrahedra are owned by more than one volume or mesh container.

# Cleaning Up a Hexahedral Mesh

The command to cleanup a hex mesh is:

**Cleanup Volume <id_range> [angle <value=150>]**

Hexahedral mesh cleanup is newer to Cubit and currently only a single type of bad element is found and fixed. The hex mesh quality configuration that is currently implemented is when a column of hex elements is on the boundary of a volume, the hexes in the column each have 2 adjacent quad faces on the boundary, and the dihedral angle between those 2 faces is greater than the specified **angle** tolerance. This situation is illustrated in Figure 1 where the red column of hexahedra has a good angle on the source surface, which flattens out to 180 degree angle on the target creating inverted hex elements. The **angle** parameter determines how large the angle can get before being cleaned up.

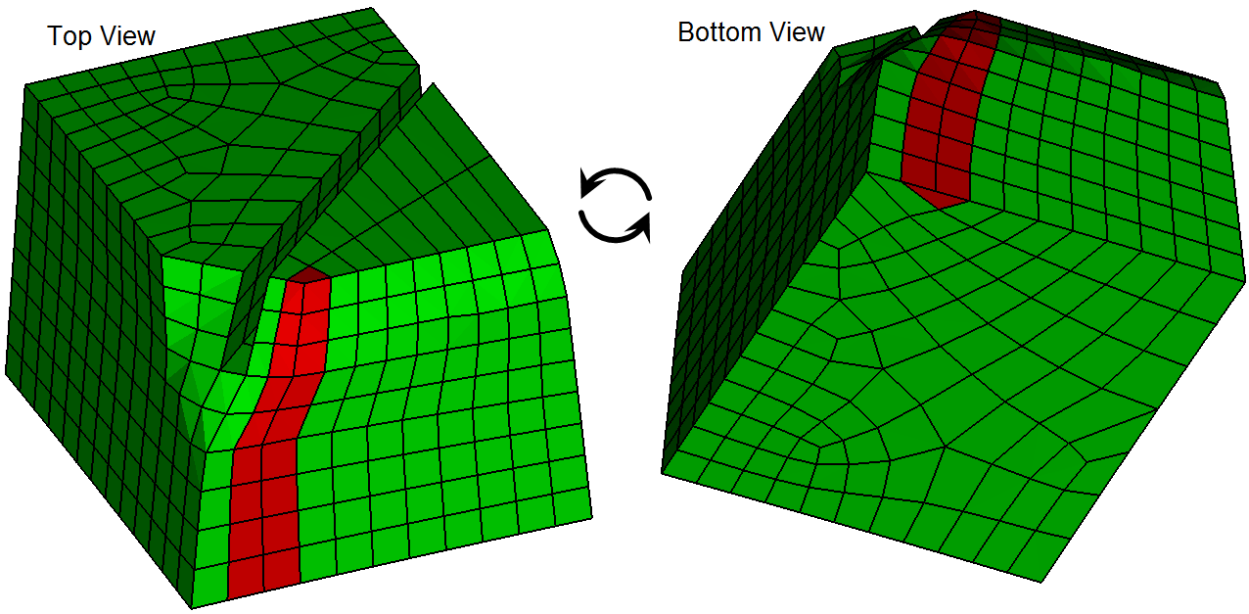



**Figure 1. Example of hexahedral mesh with case handled by hex mesh cleanup.**

Figure 2 illustrates the result of hex mesh cleanup. Internally, Cubit finds the column of hexahedra with the bad elements, as well as an adjacent column of hexahedra, and then automatically performs some hex column operations followed by smoothing to improve the quality of the elements locally.

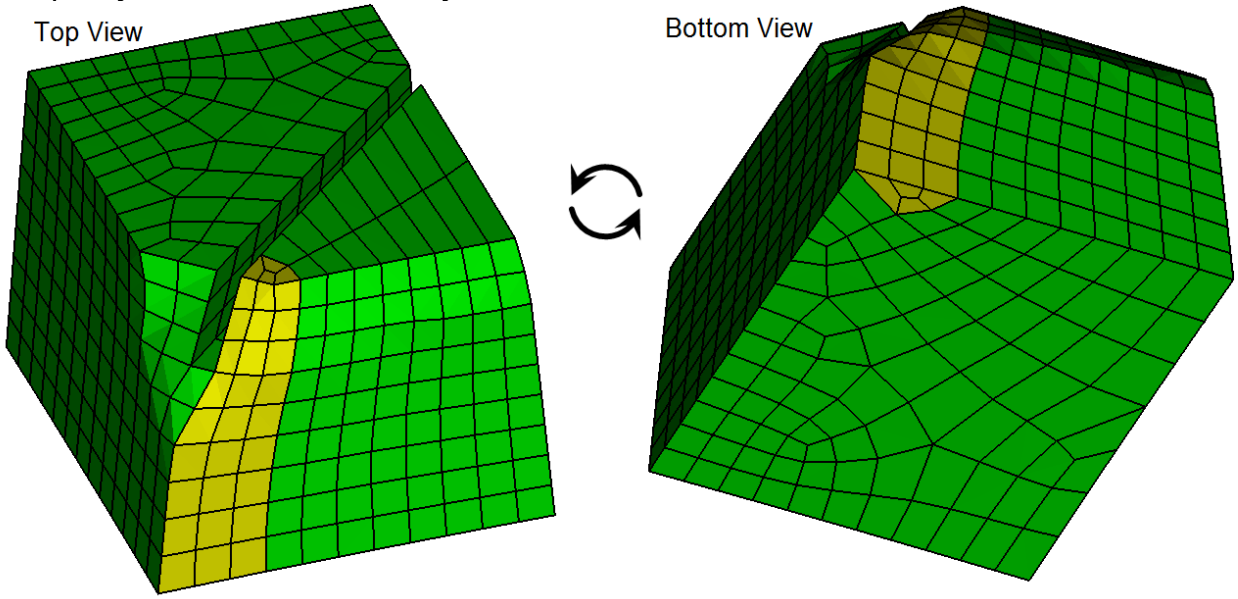



**Figure 2. The mesh from Figure 1 after hex mesh cleanup.**

## Remeshing

Mesh generation is frequently an iterative process of meshing, deleting the mesh, and remeshing. The remesh command is a convenient tool to bypass the mesh deletion process when used to remesh a volume. You may also use the remesh command to replace a localized set of deformed triangles or tetrahedra after analysis. Thus, remeshing can become part of an optimization loop.
Use the following command to remesh hexahedra:

```
Remesh Volume <range>
```

Use the following command to remesh tetrahedra:

```
Remesh {Volume|Block|Tet} <range> [FIXED|free]
```

or to remesh a range of tets or tris based upon quality criteria:

```
Remesh Tet <id_range> | [quality <tet_metric> [less than|greater than] <value>
...] [inflate <value>][size <value>][FIXED|free][preview]
```

```
Remesh Tri <id_range> | [quality <tri_metric> [less than|greater than] <value>
...] [inflate <value>][preview]
```

#### Remeshing a Swept Volume Mesh

The remesh command can be useful when using the sweep scheme. When a sweep scheme is applied to the volume, it will delete the target surface mesh on a volume with one of the sweeping schemes and then remesh the volume. It is useful when changing between sweep smooth options as in the following example below.

```
volume 1 scheme sweep
mesh volume 1
```

At this stage, the user may discover that poor quality elements may have been generated. The user could then do the following:

```
volume 1 sweep smooth winslow
remesh volume 1
```

At this point, the volume is remeshed using the sweep smooth winslow option.

#### Remeshing Tetrahedra

When used for tetrahedra, the **Remesh** command generates a new tetrahedral mesh after deleting the existing mesh described by the list of tetrahedra, volumes, or blocks. When remeshing a list of tetrahedra, the smallest set of tets possible is replaced, which often means a partial remeshing of volumes, surfaces and/or curves. This set will always include the input list of tetrahedra but may include more.
Each tetrahedron may only be in one volume or block, but the list of tetrahedra may span volumes or blocks. Each block is treated individually if multiple blocks are specified.
The default **FIXED** option will ensure that any triangle or edge in the tetrahedron list to be remeshed that lie on geometric surfaces or curves will not be affected by the remesh operation. In contrast, the **free** option allows edges and triangles on curves and surfaces

to be removed and remeshed. Use the **FIXED** option when it is important to maintain the boundary mesh configuration fixed, otherwise the free option will remesh the portions of curves and surfaces in the remesh region.
The **Remesh** command can be used to selectively remove and remesh a small portion of tetrahedron in the mesh that have been identified as poor quality. This can be an effective tool for improving mesh quality on a deformed mesh following an analysis without the need to regenerate the full mesh.
The **quality** option will identify those tetrahedra from the full model and apply the remeshing opertaion only to those tetrahedra. Any of the standard quality metrics for tetrahedra may be used as the **<tet_metric>**. These include: **Aspect Ratio Bet, Aspect Ratio Gam, Element Volume, Condition No., Jacobian, Scaled Jacobian, Shape, Relative Size, Shape And Size, Distortion, Allmetrics, Algebraic** and **Traditional.** The metric specification is used in conjunction with a **less than** or **greater than** specification and a threshold value. For example, the syntax below would remesh all tetrahedra in the mesh who's scaled jacobian metric was less than 0.2.

```
remesh tet quality Scaled Jacobian less than 0.2 inflate 1 free
```

The **inflate** option can be used to expand the set of tets selected by the **quality** metric criteria. The **<value>** input following the **inflate** option is the number of tet layers surrounding the poor quality tets that will be included in the remesh region. Usually a value of 1 is sufficient to allow the tet mesher to generate better quality elements, however 2 or greater will remesh a larger portion of the mesh. A value of 0 is generally not recommended as it usually does not provide enough space for the tet mesher to improve element quality. The **inflate** option can also be used independently from the remesh command. See the **Inflate** command described below.
This command also allows for multiple quality criteria. For example, the following command would use both aspect ratio and scaled jacobian as criteria for remeshing. Any number of quality criteria may be included in the command syntax:

```
remesh tet quality Scaled Jacobian less than 0.2 quality Aspect Ratio Bet
greater than 4 inflate 1 free
```

The **preview** option will display the tetrahedra selected by the **quality** criteria and **inflate** options without actually performing the remeshing operation.
Sizing functions may be used with tet remeshing. See Mesh Adaptivity and Sizing Functions and Exodus II-based Field Function for more information.
The **size** parameter may be used to control the size of the tets created in the remesh.

#### Inflating a set of Tets

In cases where a set of tets are to be remeshed, it is useful to be able to expand the set to include additional surrounding tets. This is to allow the mesher more freedom to place good quality elements, but also to ensure a valid shape in which the mesher has to work. The Inflate command starts with a given set of tetrahedra and will expand the set based on the number of user defined layers as well as manifold criteria. The result will be added to the curent group, or a new group can be created. The following describes the syntax and arguments to this command:

```
Inflate {group <id>|tet <ids>} {manifold|layer <value>}[{add|create <"name">}]
[draw]
```

**group<id>|tet<range>:** input to this command can be with a group name, group id, or a range of tets. The group must contain at least 1 tet. The tets need not be contiguous.
**manifold**: This option will add tets to the set where the boundary or skin of the tets meet at a single edge or node. This ensures that a complete valid manifold definition of the boundary of the set of tetrahedra can be defined. This is important for the tetrahedral mesh generator which requires a manifold boundary definition. both **layer** and **manifold** can be used in the same command.
**layer <value>:** This option will add the number of layers of tets indicated by **value** to the set. A layer is defined by all tets connected by at least a node to the skin of the existing set. This option alone does not guarantee a valid manifold definition. Use both the **layer** and **manifold** in the same command options to ensure a manifold definition.
**add|create<"name">:** The **add** option will add tets in the inflated region to the input group. An input group must be specified for this to be a valid option. The **create** option will create a new group and add all tets (including the input), to a new group specified by **<"name">**. If neither **add** nor **create** are specified, a new default group named "**inflated_tets**" will be created. If a group of that name already exists, it will be added to.
**draw:** The **draw** option will display both the input set of tets and the inflated tets in the graphics window. The input tets will be displayed in green and the inflated tets will be displayed in red.
**Examples:**
Generate a simple tet mesh. For tets with ids 1 to 10, define a 1 layer buffer and ensure it maintains a manifold boundary. The result will be placed in a new group called "**inflated_tets**" and displayed in the graphics window.

```
brick x 10
vol 1 scheme tetmesh
mesh vol 1
inflate tet 1 to 10 layer 1 draw
```

Create a group called "bad_tets"containing all tets in volume 1 with quality metric (scaled Jacobian) less than 0.2. Expand that group by one layer and remesh it.

```
group 'bad_tets' equals qual vol 1 scaled high 0.2
inflate bad_tets layer 1 add
remesh tet in bad_tets
```

#### Remeshing Triangles

Remeshing triangles works in many of the same ways as remeshing tets, generating a new triangle mesh after deleting the existing mesh described by the list of triangles. When remeshing a list of triangles, the smallest set possible is replaced, which often means a partial remeshing of surfaces only. This set will always include the input list of triangles but may include more to ensure the set is non-manifold. Some important differences between remeshing triangles vs tets are:

- Triangle remeshing does not support the **free** option. So if the triangles include mesh edges that lie on geometric curves, these edges will not be affected by the remesh operation.
- Triangle remeshing uses the **trimesh** scheme to do the remeshing, whereas tet remeshing with the **free** option uses scheme **triadvance**.

- A **size** parameter allows for control of the size of the triangles generated in the remesh.

## Edge Swapping

The edge swap command allows a user to target a specific edge between two triangles (similar functionality for quads and tets has not been included) and change the connectivity of the triangles. Multiple edges can be swapped simultaneously. The input order of the edges is the order in which the swaps will be performed.

Typically, the edge swap command is used to specifically repair local mesh connectivity.

**Swap Edge <id_list>**

The following images show the before and after views of a model where the highlighted edge is swapped. The edge in each image is the same edge.

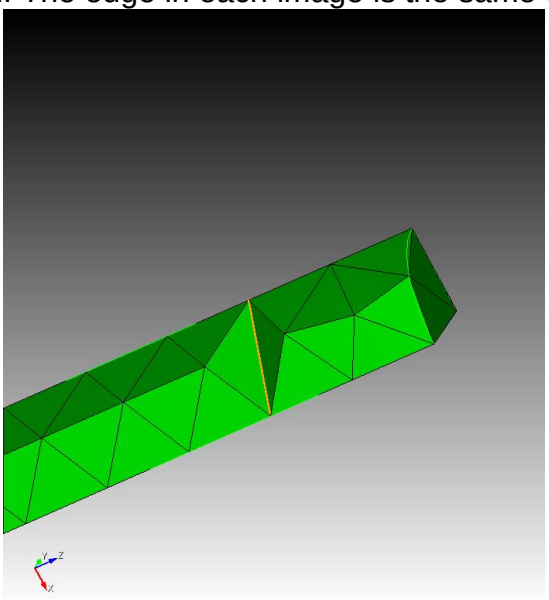


Image 1 - Before edge swapping

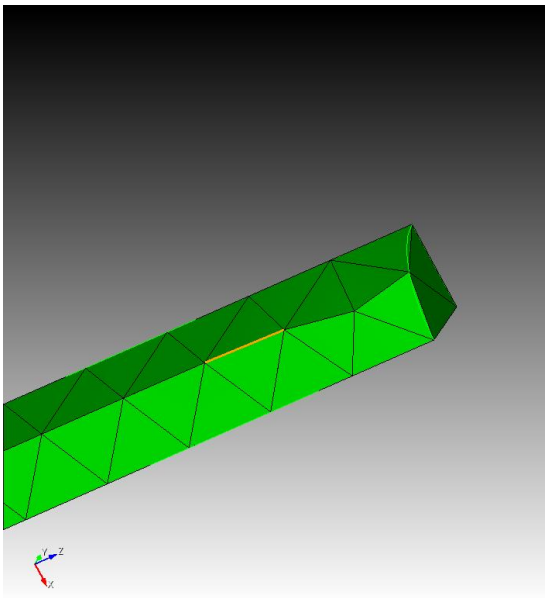


Image 2 - After edge swapping

## Matching Tetrahedral Meshes

The intended use of this function is for importing two exodus or genesis files that have non-conforming mesh where they touch and modifying the meshes locally to make them conforming. The result is a single mesh that is stitched together at the locally modified region. This functionality is currently only available for tetrahedral meshes.

Tetrahedral mesh matching will work on free mesh only. The interface where the two meshes will be matched need not be planar. A single target sideset and one or more source sidesets should be provided. The source sideset should be completely enclosed in the target sideset so that the boundaries of the two sidesets do not intersect. The two meshes need not touch exactly at the sidesets but the closer the meshes are to touching the better the results will be. Small gaps or overlaps will generally be allowed. Both of the meshes involved in the matching should be contained in defined blocks prior to issuing the command.

The syntax for the command is:

**Meshmatch tet sideset <id_list> onto sideset <id>**

The one or more sidesets specified *before* the '**onto**' keyword are the source sideset(s). The sideset *after* the '**onto**' keyword is the target sideset.

## Mesh Coarsening

*Hexahedral Coarsening*

CUBIT provides a limited number of options for coarsening hexahedral meshes. The options currently available for hex coarsening rely on the hex sheet extraction process described in Mesh Refinement page. Removing a sheet from a hexahedral mesh essentially means that a complete layer of hexes will be removed and the adjacent layers expanded to take its place.

#### Extracting a Single Hex Sheet

The following command can be used to extract a single hex sheet.

**Extract sheet { Edge <id> | Node <id_1> <id_2> }**

The edge or node pair are used to define the sheet that will be extracted. Figure 3 below shows an example of extracting a hex sheet. In this example the hex sheet is specified by the node pair highlighted in the images. Note that the entire layer of hexes between the highlighted nodes has been removed and the neighboring layers have been expanded to take its place.

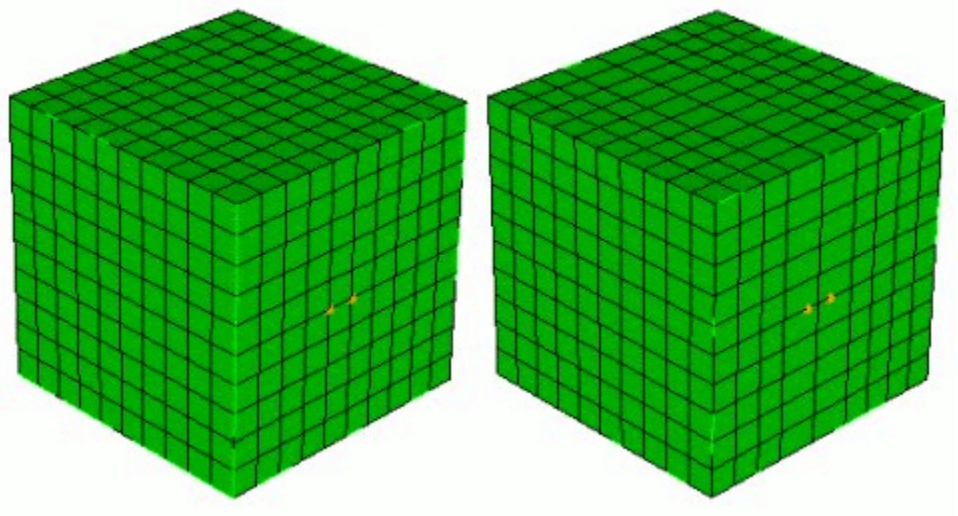


**Figure 3. Example of Hex Sheet Extraction**

**Note**: Also see the Mesh Refinement section for a description of hex sheet drawing.

#### Extracting multiple sheets along a curve

Another option for extracting hex sheets can be done by specifying a curve at which to perform the sheet extraction operations. In this case, multiple layers of hexes can be removed by specifying a curve perpendicular to the hex layers. The command for coarsening perpendicular to a curve is as follows:

**Coarsen Mesh Curve <id> Factor <value> [NO_SMOOTH|smooth]**

**Coarsen Mesh Curve <id> Remove {<num_edges>|edge <id_ranges>} [NO_SMOOTH|smooth]**

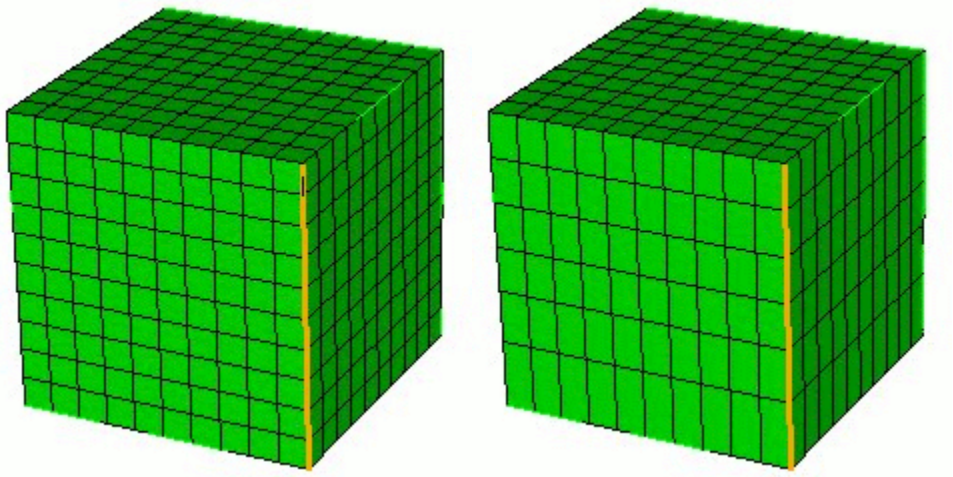


**Figure 4. Coarsening a mesh by extracting sheets perpendicular to a curve**

The first option uses the **Factor** argument. The factor argument controls how much larger the edges will be on the curve. For example, Figure 4 shows the coarsen mesh curve command used with a factor of 2. In this case, the command attempts to make the mesh edges approximately twice the length relative to their original length along the curve.
The second option uses the **Remove** argument. With this option, a specified number of layers may be removed from the mesh. This may be accomplished by indicating an exact number, or by providing a list of edge IDs that correspond to the layers that will be removed.
The **NO_SMOOTH|smooth** option allows the user to improve the element quality after the sheet extraction process by smoothing the remaining nodes. The default for both of these commands is to not smooth. Smoothing may also be accomplished after sheet extraction by using the smooth volume command.

#### Uniform hex coarsening

By applying the coarsen mesh curve command multiple times to curves that are orthogonal in the model, the effect of uniform coarsening of the mesh may be achieved.

## Mesh Refinement

- Global Mesh Refinement
- Refining at a Geometric or Mesh Feature
- Hexahedral Refinement Using Sheet Insertion
- Local Refinement of Tets, Triangles, and Edges
- Parallel Refinement
- Uniform Mesh Refinement

CUBIT provides several methods for *conformally* refining an existing mesh. Conformal mesh refinement does not leave hanging nodes in the mesh after refinement operations, rather conformal mesh refinement provides transition elements to the existing mesh. Both local and global mesh refinement operations are provided.

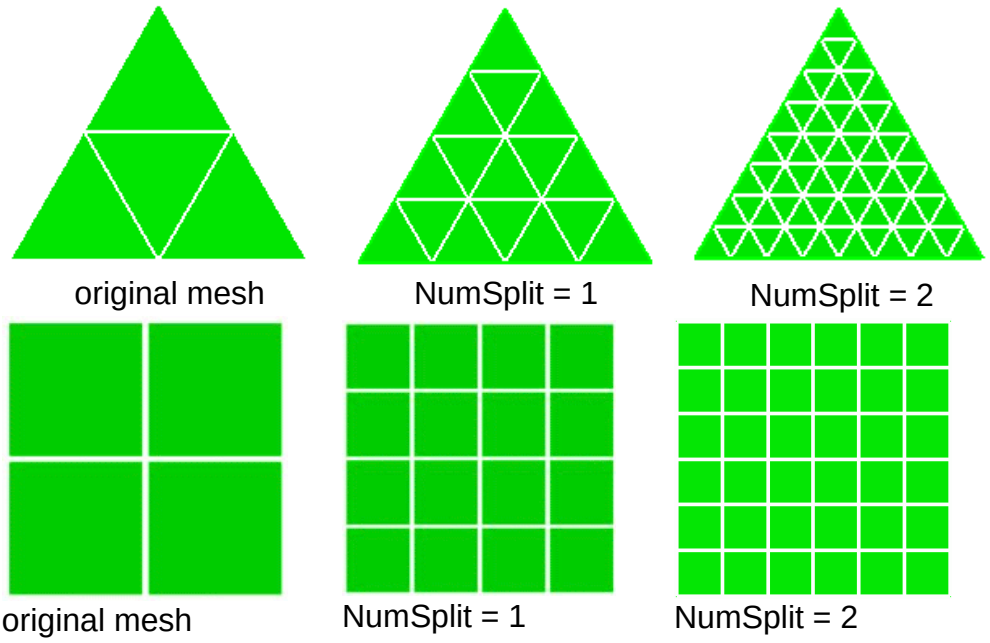
*Global Mesh Refinement*

The Refine Surface and Refine Volume commands provide capability for globally refining an entire surface or volume mesh. Global refinement will only be used if the entire body is included in the command. Otherwise, the command will be interpreted as local refinement (see below.) This distinction can be important because the global refinement algorithm divides each element into fewer sub-elements than local refinement. The command syntax is:

**Refine Volume <range>numsplit<int>**

**Refine Surface <range>numsplit<int>**

The numsplit option specifies how many times to subdivide an element. A value of 1 will split every triangle and quadrilateral into four pieces, and every tetrahedron and hexahedron into eight pieces. Examples of global refinement on each element are shown below.



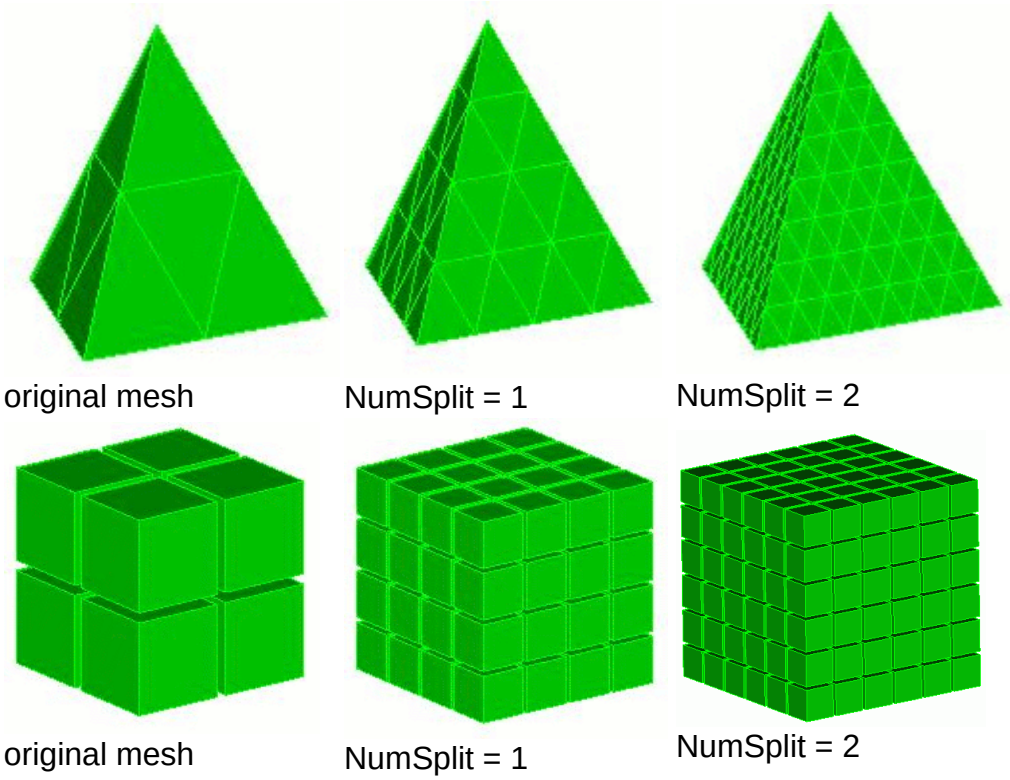



**Figure 1. Example of uniform refinement for each of the mesh entities**

*Refining at a Geometric or Mesh Feature*

CUBIT also provides methods for local refinement around geometric or mesh features. Individual elements or groups of elements can be refined in this manner using the following syntax.

**Refine {Node|Edge|Tri|Face|Tet|Hex} <range>**
**[NumSplit<int = 1>|Size <double> [Bias <double>]]**
**[Depth <int>|Radius <double>] [Sizing_Function]**
**[Smooth]**

**Refine {Vertex|Curve|Surface} <range>**
**[NumSplit<int = 1>|Size <double> [Bias <double>]]**
**[Depth <int>|Radius <double>] [Sizing_Function]**
**[Smooth]**

To use these commands, first select mesh or geometric entities at which you would like to perform refinement. Refinement will be applied to all mesh entities associated with or within proximity of the entities. The all keyword may be used to uniformly refine all elements in the model
The following is a description of refinement options.
**NumSplit**

Defines the number of times the refinement operation will be applied to the elements in the refinement region. For uniform or global refinement, where all elements in the model are to be refined, A NumSplit value of 1 will split each triangle and quadrilateral into four elements, and each tetrahedron and hexahedraon into eight elements. A numsplit of 2 would result in 9 and 27 elements respectively. For uniform refinement, the total number of elements obeys the following:

1. **NE = NI * (NumSplit+1)^Dim**

where **NE** is the final number of elements, **NI** is the initial number of elements and **Dim** is 2 or 3 for 2D and 3D elements respectively.
In cases where only a portion of the elements are selected for refinement, the elements at the boundary between the refined and non-refined elements will be split to accommodate a transition in element size. The transition pattern will vary depending on the local features and surrounding elements. For non-uniform refinement of hexahedron, for a numsplit of 1, each element in the uniform refinement zone will be subdivided into 27 elements rather than 8. This affords greater flexibility in transitioning between the refined and unrefined elements.

**Size, Bias**

The Size and Bias options are useful when a specific element size is desired at a known location. This might be used for locally refining around a vertex or curve. The Bias argument can be used with the Size option to define the rate at which the element sizes will change to meet the existing element sizes on the model. Figure 2 shows an example of using the Size and Bias options around a vertex. Valid input values for Bias are greater than 1.0 and represent the maximum change in element size from one element to the next. Since refinement is a discrete operation, the Size and Bias options can only approximate the desired input values. This may cause apparent discontinuities in the element sizes. Using the default smooth option can lessen this effect. It should also be noted that the Size option is exclusive of the NumSplit option. Either NumSplit or Size can be specified, but not both.

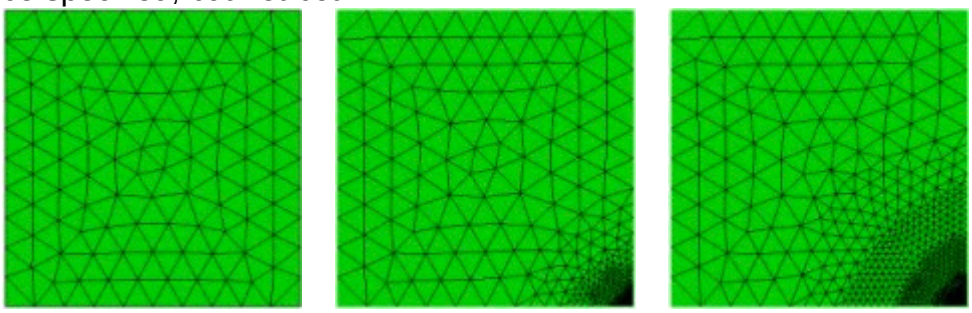



**Figure 2. Example of using the Size and Bias options at a Vertex.**

**Depth**

The Depth option permits the user to specify how many elements away from the specified entity will also be refined. Default Depth is 1. Figure 3 shows an example of using the depth option when refining at a node.

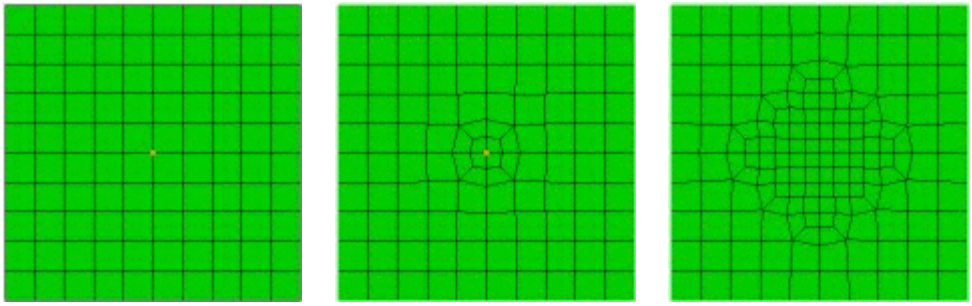



**Figure 3. Example of using the Depth option at a node to control how far from the node to propagate the refinement.**

**Radius**

Instead of specifying the number of elements to describe how far to propagate the refinement, a real Radius may be entered. The effects of the Radius are similar to that shown in Figure 3, except that the elements whose centroid fall within the specified Radius will be refined. Transition elements are inserted outside of this region to transition to the existing elements.

**Sizing Function**

Refinement may also be controlled by a sizing function. CUBIT uses sizing functions to control the local density of a mesh. Various options for setting up a sizing function are provided, including importing scalar field data from an exodus file. In order to use this option, a sizing function must first be specified on the surface or volume on which the refinement will be applied. See Adaptive Meshing for a description of how to define a sizing function.

**Smooth**

The default mode for refinement operations is to NOT perform smoothing after splitting the elements. In many cases, it may be necessary to perform smoothing on the model to improve quality. The smooth option provides this capability.

**Controlling Regularity of Triangle Refinement**

The default behavior of triangle refinement is to attempt to maximize element quality using the basic one->four template. This can sometimes result in an irregular pattern, where one or more edges are swapped. To enforce regularity of the triangle refinement pattern, regardless of quality, the folowing setting may be used.

1. **Set Triangle Refine Regular {on|OFF}**

*Hexahedral Refinement Using Sheet Insertion*

Several tools for refining a hexahedral mesh using sheet insertion and deletion are available in CUBIT.

- Refining at a Geometric Feature
- Refining along a Path
- Refining a Hex Sheet

- Directional Refinement
- Hex Sheet Drawing

#### Refining at a Geometric Feature

The following commands offer additional controls on refinement with respect to one or more geometric features of the model.
An existing hexahedral mesh can be refined at a geometric feature using the following command:

**Refine Mesh Volume <id> Feature {Surface | Curve | Vertex | Node} <id_range> Interval <integer>**

This command refines the mesh around a given feature by adding sheets of hexes. These sheets can be generalized as planes for surfaces, cylinders for curves, and spheres for vertices. The **interval** keyword specifies the number of intervals away from the feature to insert the new sheet of hexes. For this command a single sheet of hexes is inserted into the hexahedral mesh.
Figure 4 shows an example of this command where the feature at which refinement is to be performed is a curve. In this case the interval chosen was, 2. This indicated that the elements 2 intervals away from the curve would be refined.

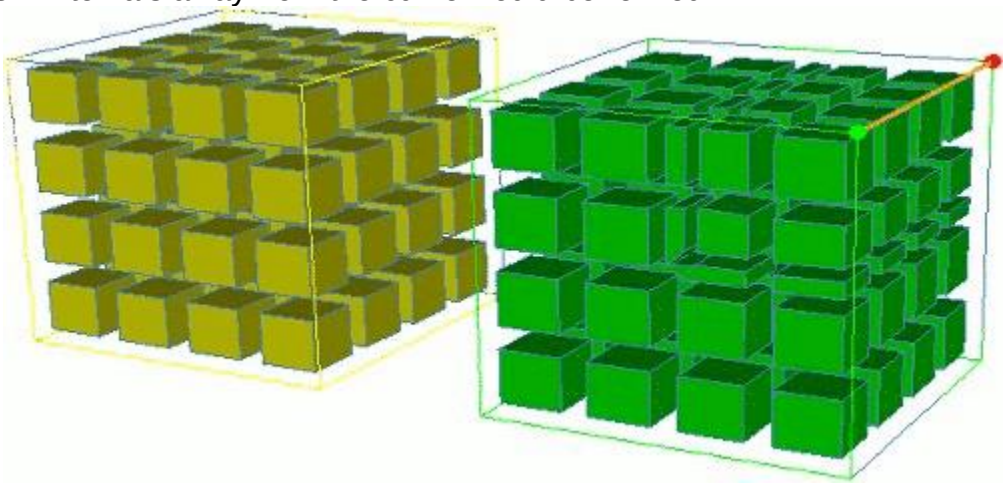


**Figure 4. Example of Refinement at a curve**

#### Refining along a path

Hexahedral meshes can be refined from a specific node and along a propagated path using the following command

**Refine Mesh Start Node <id> Direction Edge <id> End Node <id> [Smooth]**

Figure 5 shows a swept mesh and its cross section. The cross section view on the left shows a path that has been propagated through the mesh between the start node and end node. This path is then projected along a chain of edges in the direction given by the

direction edge as shown in Figure 5. The start node and end node must be on the same sweep layer. This refinement procedure also requires the volume's meshing scheme to be set to sweep. If the smooth keyword is given the mesh will be smoothed after the refinement step is complete.

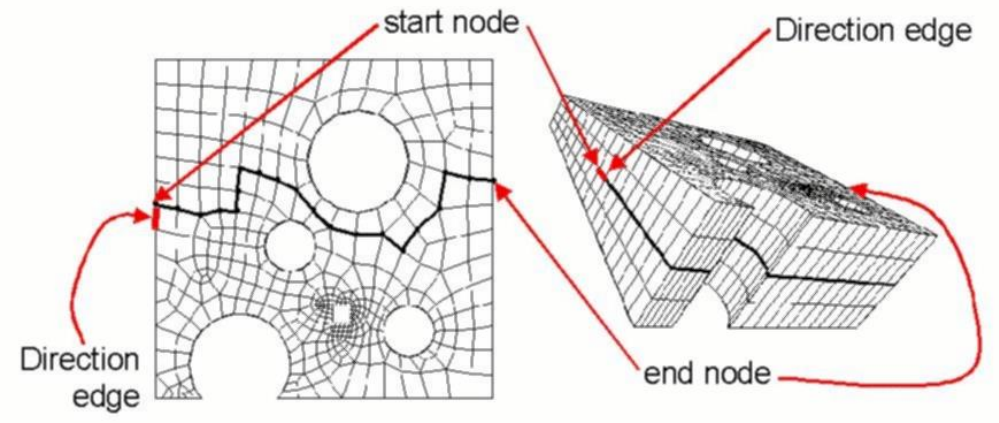



**Figure 5. Refining a Mesh Along a Path**

#### Refining a Hex Sheet

The following command can be used to refine the elements in one or more hex sheets:

**Refine Mesh Sheet [Intersect] { Node <id_1> <id_2> | Edge <id_range> } { Factor <double> | Greater_than <size> } [Smooth] [in volume <id_range> [depth <num_layers]]**

The **node** and **edge** keywords are used to define the hex sheet(s) to be refined. If the node option is chosen, only one node pair can be entered (see Figure 6). If the edge option is chosen, one or more edges can be entered (see Figure 7).

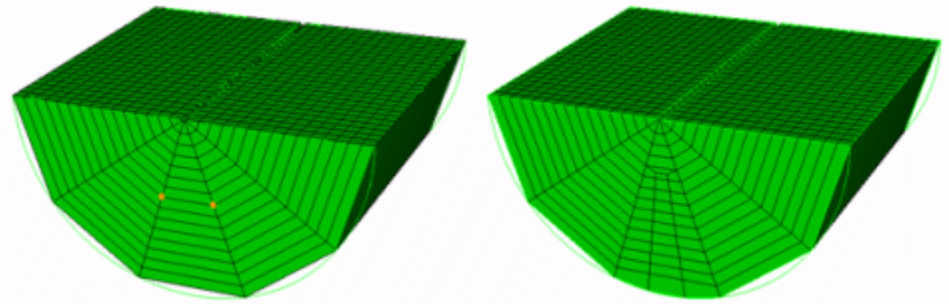


**Figure 6. Refine mesh sheet node 796 782 greater_than 6**

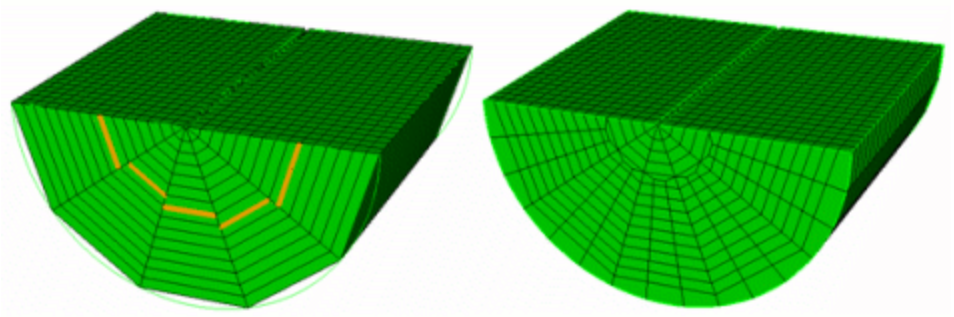


**Figure 7. Refine mesh sheet edge 1584 1564 1533 1502 1471 greater_than 6**

The **factor** and **greater_than** keywords are used to specify the refinement criteria for the selected hex sheet(s). If the factor keyword is used, the length of the smallest edge in the hex sheet is determined and any edge in the hex sheet with a length greater than the smallest length multiplied by the factor is refined. If the greater_than keyword is used, any edge in the hex sheet with a length greater than the specified size is refined.
The **intersect** keyword is optional. It is used to more easily define multiple hex sheets to be refined. If the intersect keyword is entered, the node and edge keywords are used to define a chord rather than a sheet (a chord is the two-dimensional equivalent of the three-dimensional sheet). The chord will be limited to the surface(s) associated with the nodes or edge entered, and all sheets intersecting the chord will be selected for refinement (see Figure 8). When the node keyword is used with the intersect option, the nodes must define an edge on the surface of the mesh.

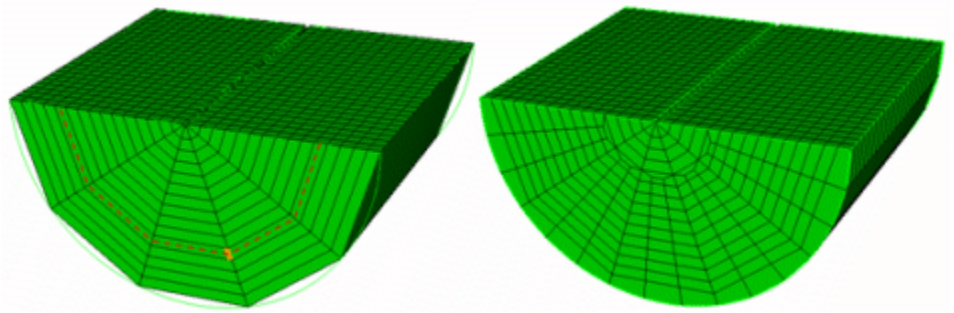


**Figure 8. Refine mesh sheet intersect edge 1499 greater_than 6**

The **smooth** keyword is also optional. When the smooth keyword is entered, the elements that have been refined are smoothed in an attempt to improve element quality. Figure 9 shows the same command as Figure 8 with the addition of the smooth keyword. Smoothing may or may not be beneficial, depending on the situation.

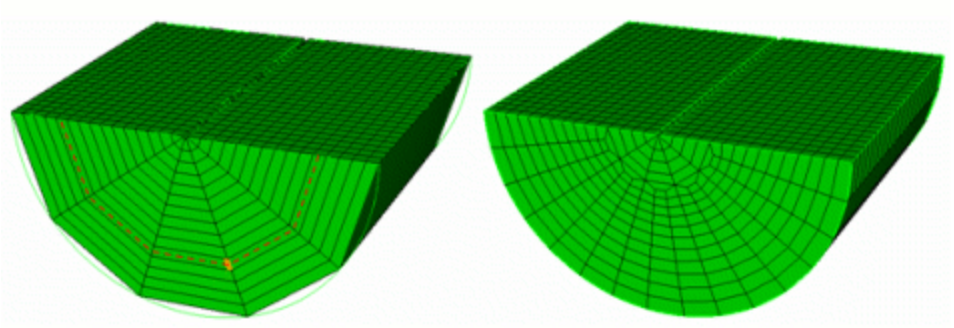


**Figure 9. Refine mesh sheet intersect edge 1499 greater_than 6 smooth**

#### Directional Refinement

Mesh sheet refinement can also be used to refine a mesh in a particular direction. This can help control anisotropy. The following command can be used as a short cut for specifying what sheets should be used in refinement.

> **Refine Volumes <id_range> using {Plane <options> | Surface <id_range> | Curve <id_range> } [ Depth <num_layers> ] [ Smooth ]**

The volumes specified indicate which hexes can be refined. A transition layer will be made out of hexes surrounding the indicated volumes. If the **depth** option is used, additional layers of hexes around the specified volumes will be included in the refinement region. Behind the **using** option, if the **plane** option is employed, all the edges in the volume which are parallel to the plane (to a small tolerance) are used to specify the sheets to refine. If the **surface** or **curve** option is employed instead, all the edges in the surfaces or curves will be used.
For example, Figure 10 and 11 shows directional refinement using the plane option. The command used to convert the mesh in Figure 10 to Figure 11 is:

```
refine vol 2 using plane xplane depth 1
```

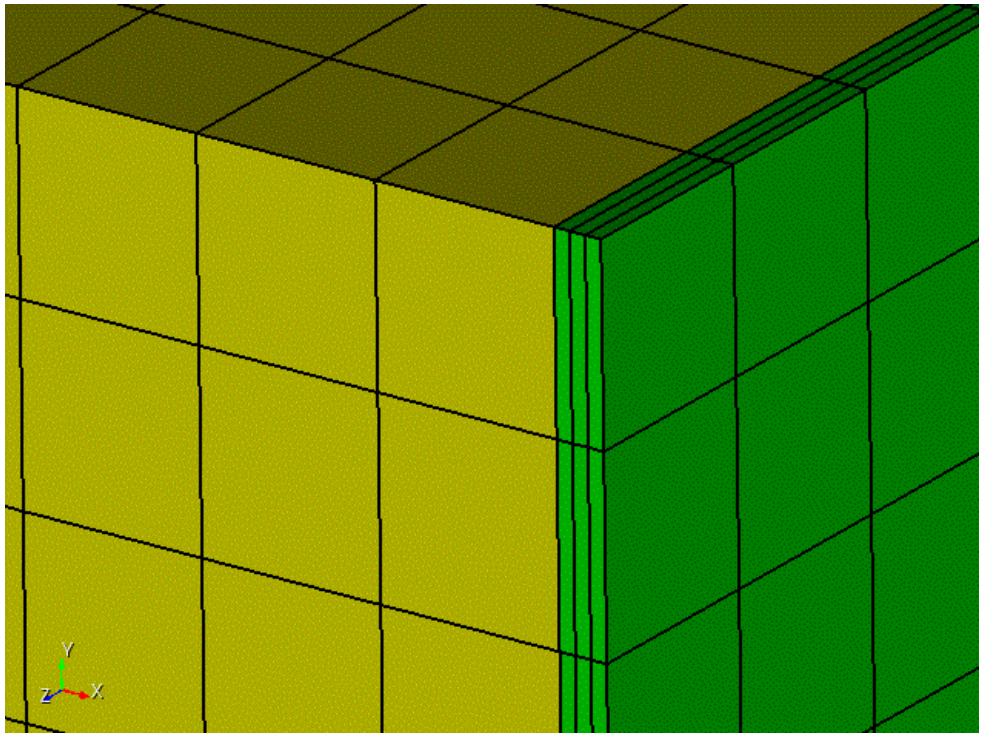



**Figure 10. Starting mesh**

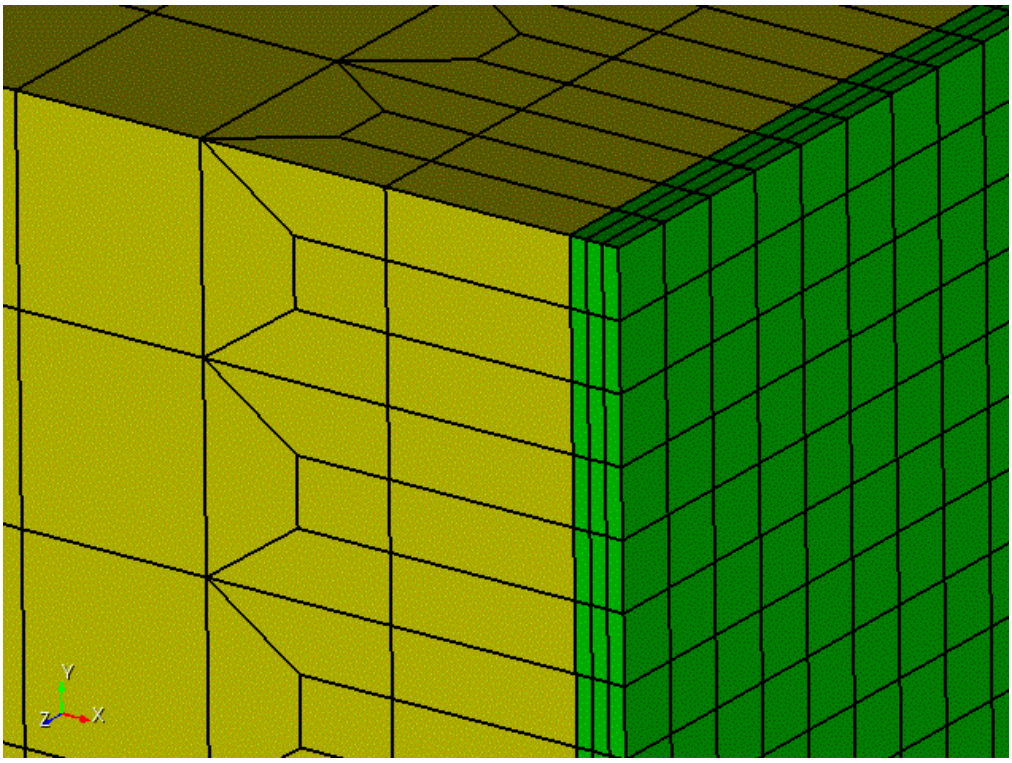



**Figure 11. Directional result of refinement resulting from using the plane option on the refinement command.**

Directional refinement can be used iteratively to reduce or create anisotropy of any level. This is done by applying the direction refinement command iteratively. A second iteration of directional refinement can be applied by issuing the same command again. To improve element quality, however, it is often recommended to perform refinement parallel to the plane before subsequent iterations. For example, taking the mesh in Figure 11 as input, the following commands will generate the mesh in Figure 12.

```
refine mesh sheet edge ( at 4.5 5 5 ordinal 1 ) factor 0
```

```
refine vol 2 using plane xplane depth 1
```

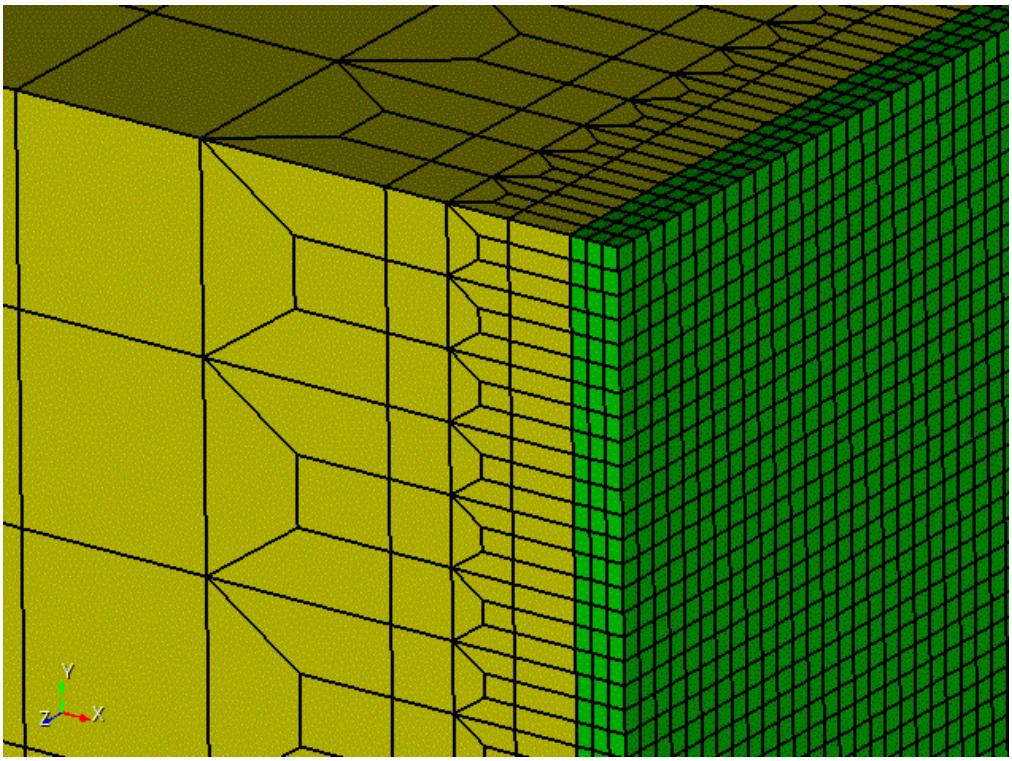



**Figure 12. A 2nd iteration of direction refinement is applied.**

#### Hex Sheet Drawing

Since refinement of hex meshes generally occurs by inserting hex sheets, tools have been provided to draw a specified sheet or group of sheets.
This command draws a sheet of hexes that is defined by the edge or node pair.

```
Draw Sheet {Edge <id> |Node <id_1> <id_2>}[Mesh [List]] [Color
<color_name>] [Gradient]
```

The following command draws the three sheets that intersect to define the given hex. These sheets are drawn green, yellow, and red. To draw a specific sheet, list its color in the command.

**Draw Sheet Hex <id> [Green][Yellow][Red][Mesh [List]] [Gradient]**

The 'gradient' keyword for both commands draws the sheet in gradient shading according to the distance between opposite hex faces that are parallel to the sheet.
The 'mesh' keyword will draw the hexes in the hex sheet. If the 'list' keyword is also given, the ids of the hexes in the sheet will be listed.

*Local Refinement of Tets, Triangles, and Edges*

Local refinement of tets, triangles, and edges is available by refining individual entities or by refining to guarantee a user-specified number of tests through the thickness:

- Single Entity Refinement
- 'N' Tets Through the Thickness Refinement

#### Single Entity Refinement

Local refinement of tets, triangles, and edges is available. When refining triangles a node is inserted at the enter of the triangle and three new triangles are connected to this node. The original triangle is deleted. The command to refine triangles is:

**Refine Local Tri <tri_id_list>**

When refining an edge, a node splits the original edge between two triangles and four new triangles are created and connected to the new node. The command to refine an edge is:

**Refine Local Edge <edge_id>**

When refining a tet edge, the tet edge is split by a node and then all tets attached to the original edge are split into two through a triangle that goes through the new node. All other adjacent nodes and edges are unmodified by the operation. Note that on the interior of the mesh tet edges are not represented explicitly so the command takes two nodes as input to define the edge. The command to refine a tet edge is:

**Refine Tet_edge Node <node1_id> <node2_id>**

#### 'N' Tets Through the Thickness Refinement

Cubit provides a capability to guarantee a user-specified number of tets through the thickness. This functionality is intended to work on an existing tet mesh using mesh refinement. The user specifies the geometry or mesh defining the thin region and also the number of desired tets through the thickness and the refinement algorithm will run until it meets this criteria. The number of tets through the thickness in this context is interpreted as the number of mesh edges through the thickness and the algorithm will continue to do

refinement until there are no mesh edge paths through the thin region that contain fewer mesh edges than the number specified by the user. The command for doing this is:

> **Refine min_through_thickness <val> source {surface|node|tri|nodeset|sideset|block} <id_range> target {surface|node|tri|nodeset|sideset|block} <id_range> [anisotropic] [single_iteration] [dont_fill_in_gaps]**

The various options are described below.
**anisotropic**: When this option is specified in the command the algorithm will only attempt to refine the edges that go roughly normal to the source and target entities. This will give an anisotropic result. The meshes on the source and target will generally not be affected when this option is used. When this option is not specified the refinement algorithm will be isotropic in nature and will propagate much more. However, it will tend to have better transitioning from the refined region to the non-refined regions.
**dont_fill_in_gaps**: When this option is specified in the command the algorithm will NOT try to grow the regions that will be refined. When the regions are grown it helps to avoid leaving small pockets of mesh that are not refined (splotchiness). This has effect only on isotropic refinement (when the "anisotropic" option is NOT used).
**single_iteration**: When this option is specified in the command the algorithm will only run for one iteration even if the min_through_thickness criteria is not met.
A quality command for querying the minimum number of tets through the thickness is found here.
Below is an example using the following commands:
**refine min_through_thickness 4 source surf 1 target surf 2 anisotropic**
**refine min_through_thickness 4 source surf 7 target surf 13 3 14 anisotropic**
Surfaces 1 and 2 are the two surfaces on opposite sides of the thin region in the green volume and surfaces 7 13 3 14 are the surfaces on opposite sides of the thin region in the yellow volume.

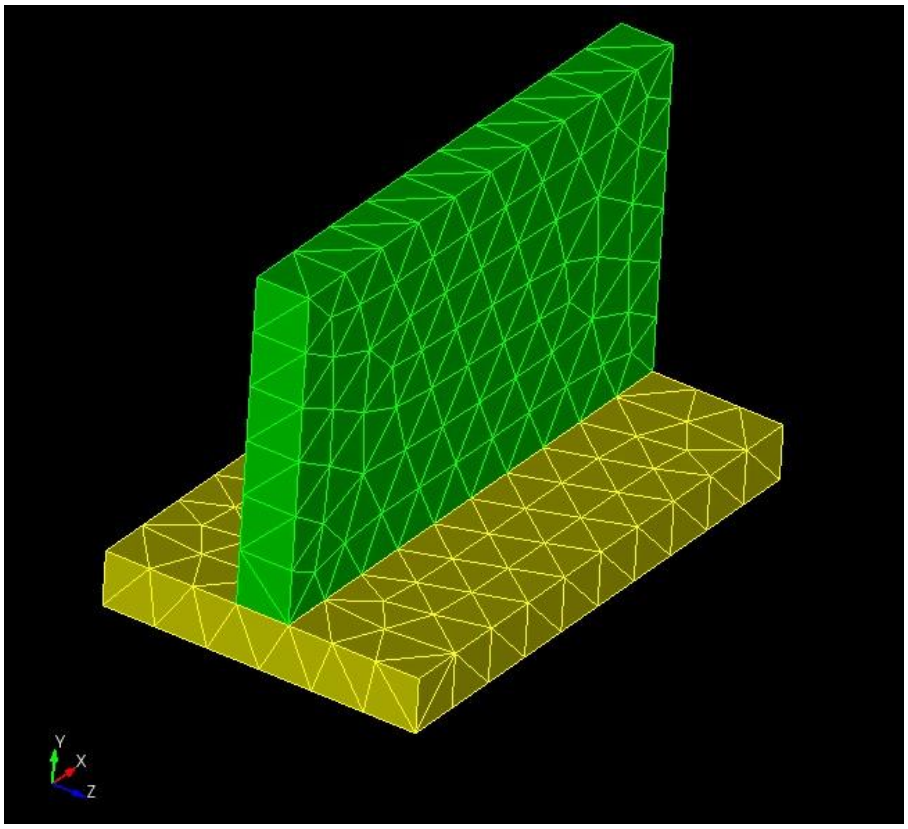



**Figure 13. Before N through the thickness refinement.**

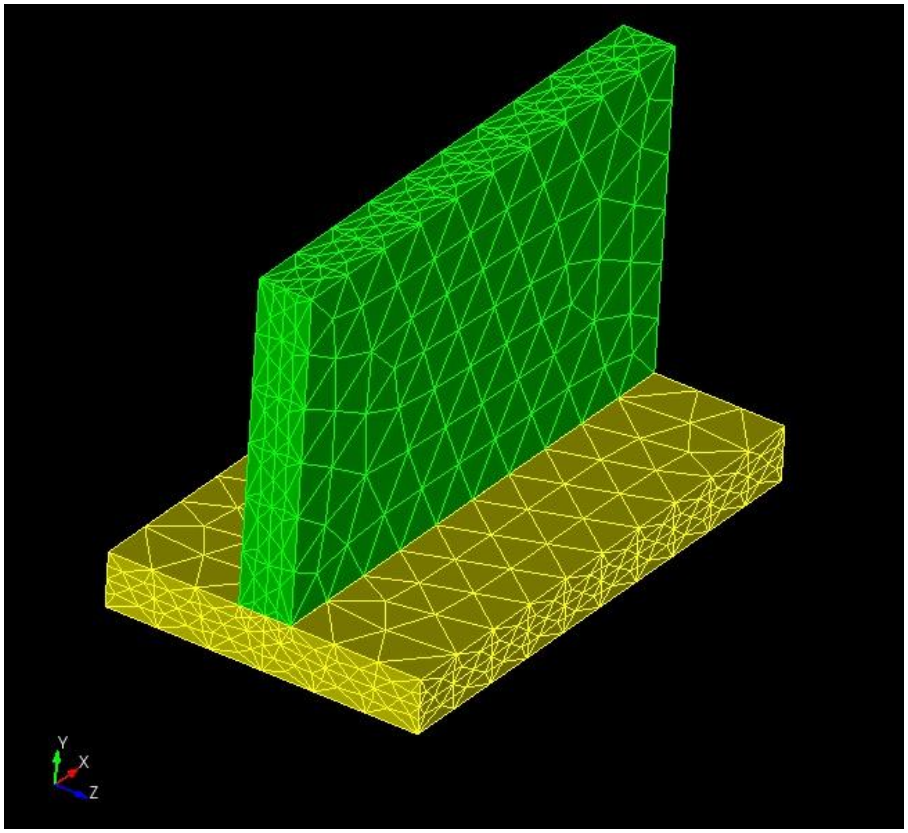



**Figure 14. After N through the thickness refinement.**

*Parallel Refinement*

Global mesh refinement can be used to increase global mesh density with a single command. If an extremely large mesh is desired, one approach is to generate a coarse mesh with the desired relative mesh gradations, and then perform global mesh refinement to scale the number of elements up across the model. Depending on the amount of refinement requested, this can exceed the memory limits of Cubit running on a single processor. Global parallel mesh refinement allows refinement to go beyond the memory limits of a single processor. The resulting mesh size is only limited by the number of processors you have available to perform the refinement. The command syntax is:

> **Refine Parallel [Fileroot <'root filename'>] [Overwrite] [No_geom] [No_execute] [Processors <int>] [Numsplit <int>] [Version <'Sierra version'>]**

This command causes Cubit to write two files to disk. First, Cubit writes an Exodus file named <root filename>.in.e which contains the mesh elements in the current Cubit session. Second, Cubit writes an OpenNURBS 3dm file http://www.opennurbs.org. named .3dm, which contains a definition of the geometry from the current Cubit Session. The **Fileroot** argument specifies the full path and root of the files that will be written. Additional blocks are written to the Exodus file to correspond to the geometry entities in

the 3dm file. The **Overwrite** argument specifies if existing files on disk with the same names should be overwritten or not.
When the mesh is refined in STK_Adapt, the new nodes created during refinement will be projected to the geometry definitions from the OpenNURBS file. If the **No_geom** argument is specified, only the Exodus file is written, and new nodes will be placed by evaluating the shape functions of the elements being evaluated.
The exported Exodus and OpenNURBS files are prepared specifically for input into the Sierra STK_Adapt program. By default, Cubit spawns STK_Adapt in the background after exporting the files. If the **No_execute** argument is specified, the Cubit command exports the files, but does not spawn STK_Adapt. The user can then move those files to a large parallel machine to perform the STK_Adapt refinement.
If **No_execute** is not specified, then Cubit will spawn Sierra STK_Adapt in the background to perform the refinement. The **Processors** argument specifies the number of processors to use for the STK_Adapt run. The **Numsplit** argument specifies how many times the global refinement should be performed. If **Numsplit** = 1, then each element edge is split into 2 sub-edges. If **Numsplit** = 2, then each element is split into 4 sub-edges, etc. The optional **Version** argument allows the user to specify which version of STK_Adapt should be run. Possible values for **Version** include "head", "4.22.0", etc.
Refine parallel command creates groups to visualize the association between mesh entities (edge, tri, and quads ) and geometric entities (curves & surfaces). There are three types of groups that exist for each mesh entity type edge, tri, and quad. First group contains unique 1-1 map between mesh entity and geometric entity. Note that issuing "Debug 212" command before calling the refine parallel command, will create separate group for each geometric entity containing unique mesh entities. Next group contains mesh entities that point to multiple geometric entities. And the final group contains mesh entities not associated with any geometric entity.
After the Refine Parallel command finishes, the mesh in Cubit does not change, normally because the resulting mesh would be too big to store in Cubit on a single processor. Instead, the refined mesh is written to disk in a series of Exodus files, one per processor, using the **Fileroot** argument as the root of the Exodus file names. For example, if **Fileroot** is *"somemesh"* and **Processors** is 8, STK_Adapt will write out eight Exodus files named *somemesh.e.8.0*, *somemesh.e.8.1*, â€¦, *somemesh.e.8.7*. These files can be kept distributed for an analysis run, or united using the Sierra EPU command. In this example, Cubit would have written out a file called *somemesh.in.e*, which contains all of the sideset, nodeset, and block definitions defined in the Cubit session. All of these sidesets, nodesets and blocks are transferred to the refined exodus files (*somemesh.e.8.0*, etc.) for use in the subsequent analysis. The *somemesh.e.** files will also contain several other blocks which correspond to geometric entities defined in *somemesh.3dm* to enable the mesh to be refined to the CAD geometry, and should be ignored by downstream applications.
Sierra STK_Adapt must be in the PATH on the computer Cubit is running on. If Sierra STK_Adapt cannot be found, Cubit returns an error and no refinement is performed. Information on how to download and build Sierra STK_Adapt can be found at http://trilinos.sandia.gov/packages/stk/.

*Uniform Mesh Refinement*

The Uniform Mesh Refinement (UMR) tool refines a mesh stored in the Exodus format uniformly, splitting every element in the mesh into a number of sub-elements, and writes the fine mesh to a new Exodus file. The resulting elements have roughly half the edge length of the original mesh. The algorithm uses an efficient streaming pipeline with low memory requirements. On recent CPUs with an SSD (solid state drive), it will write a mesh up to 4 billion elements or nodes in only a few minutes, and millions of elements or nodes in seconds or less.
In the alpha release of UMR, only blocks and sidesets of tetrahedra (4 node) and triangles (3 node) are supported. No projection of new nodes to geometry or smoothing is performed. Each TET element results in 8 new TET elements, each TRI results in 4 new TRI elements. Resulting blocks and sidesets in the output mesh will have the same IDs as the input mesh.
In its current form, UMR is run as an executable available on the LAN. It has the following options as displayed with the '-h' option:
Usage: ./build/extra/extra [OPTIONS] [INPUT] [OUTPUT]

| Positionals | |
|---|---|
| INPUT FILE</> | Input mesh file (coarse mesh). |
| OUTPUT FILE</> | Output mesh file (refined mesh). |

| Options: | |
|---|---|
| -h,--help | Print this help message and exit |
| -g,--geometry FILE | Snap boundary to geometry given in STEP format |
| -a,--associate | Output the geometry association to a .exo file |
| -b,--boundary | Output the boundary mesh as OBJ file |
| -m,--metrics | Print element quality in input mesh and exit |
| -j,--jobs N:UINT in [0 - 30] | Number of concurrent jobs. 0 means maximum number of logical cores. (default = 0). |
| -v,--verbose N:INT in [0-3] | Write increasingly more output (default =1) |
| -q,--quiet | Suppress output (same as --verbose=0) |
| -d,--debug N:INT in [0 - 5] | Write more debug output to log files (default = 0). |
| -V,--version | Prints version information |

Project home page: https://cee-gitlab.sandia.gov/meshing/extra

## Mesh Scaling

Cubit supports the scaling of hexahedra and tetrahedra meshes. Mesh Scaling allows a series of meshes to be built [typically] for solution convergence studies or other purposes. Each mesh has progressively larger or smaller elements. For example, if the input mesh has 10,000 hexahedra, scaling with a multiplier of 2.0 will result in a mesh of about 20,000 hexahedra, with approximately the same element orientation and size gradations as the original, as seen in Figure 1. Additional meshes can be built by scaling the original mesh with multipliers of 4, 6, 8, etc. Scaling by a value less than 1.0 will produce a mesh with fewer elements. Scaling by a negative value is not allowed. Convergence studies can be

performed with much less computational cost than if traditional global refinement is used, because the element increase at each step of the series can be smaller.

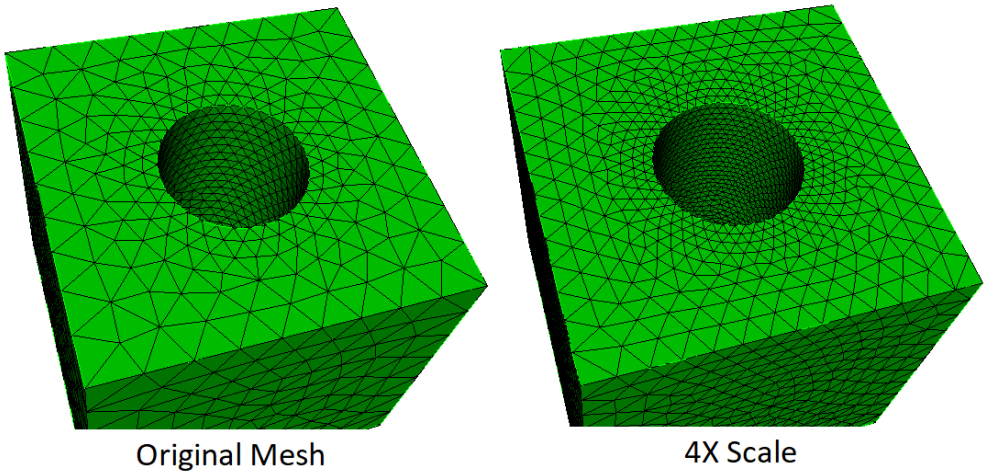



**Figure 1.**

**Command Syntax**

**scale mesh [volume <ids>]**
**[multiplier <value, default=2.0>]**
**[minimum <value, default=1>]**
**[{SWEPT_BLOCKS | legacy | maintain_structure}]**
**[feature_angle <value>]**
**[force_structured in {[volume <ids>] [surface <ids>]}]**
**[thin_gap_intervals <value, default=2>] [fix_all_gaps]**
**[max_aspect_ratio <value> in volume <ids>]**
**[max_feature_length <value, default=30>]**
**[smooth_volume {ON|off}]**

**Command Options**

**scale mesh [volume <ids>]**

Specifying a list of volumes is optional. By default, all volumes will be scaled. For Hex Mesh Scaling, the specified volumes, together with any volumes merged with them, will be scaled. Tet Mesh Scaling only scales the specified volumes, preserving the shared mesh boundary with any non-participating volumes. This approach allows individual assembly components to be scaled, without scaling the entire model.

**[multiplier <value, default=2.0>]**

The target number of output elements is the number of input elements times the **multiplier** parameter. The default value is 2.0. For example: the mesh in Figure 2a has 3025 hex elements. After scaling by 2.0, the mesh in Figure 2b has 6804 hexes. Note the locations of new nodes are projected to lie on the associated CAD geometry, if any.

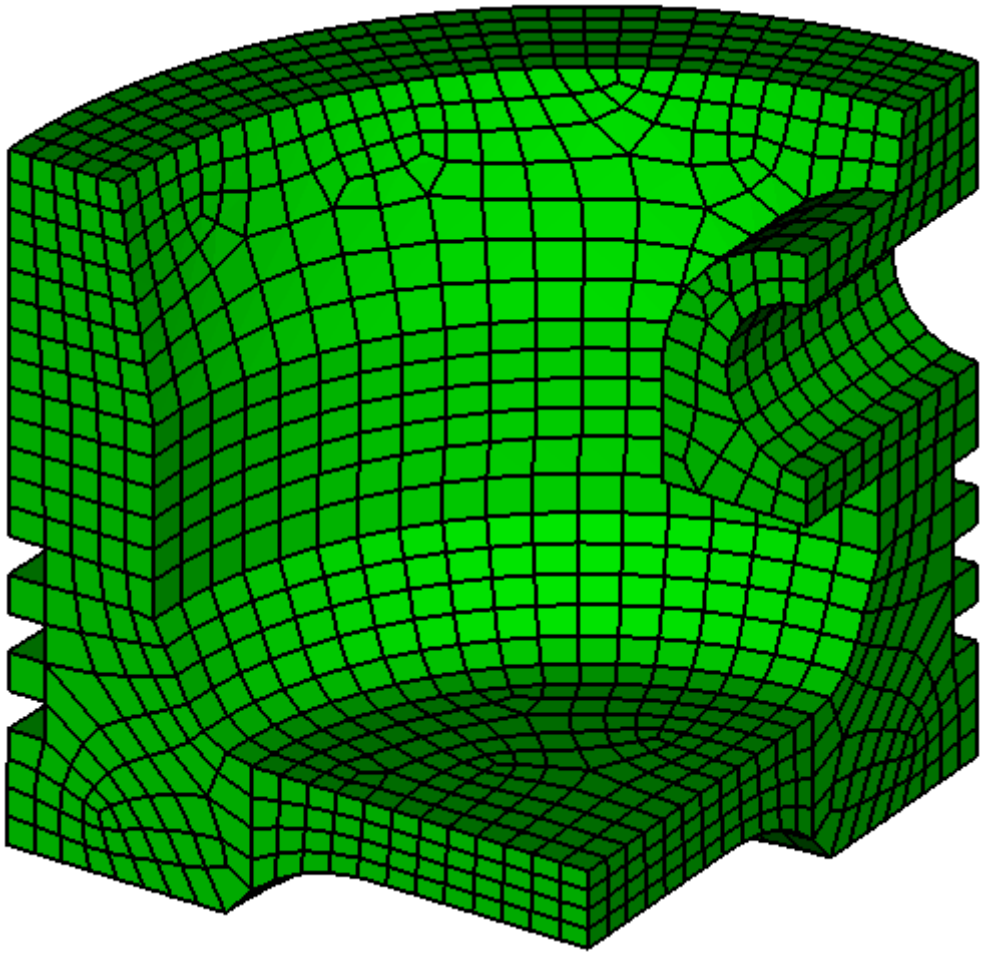


**Figure 2a. Input mesh of 3025 hex elements.**

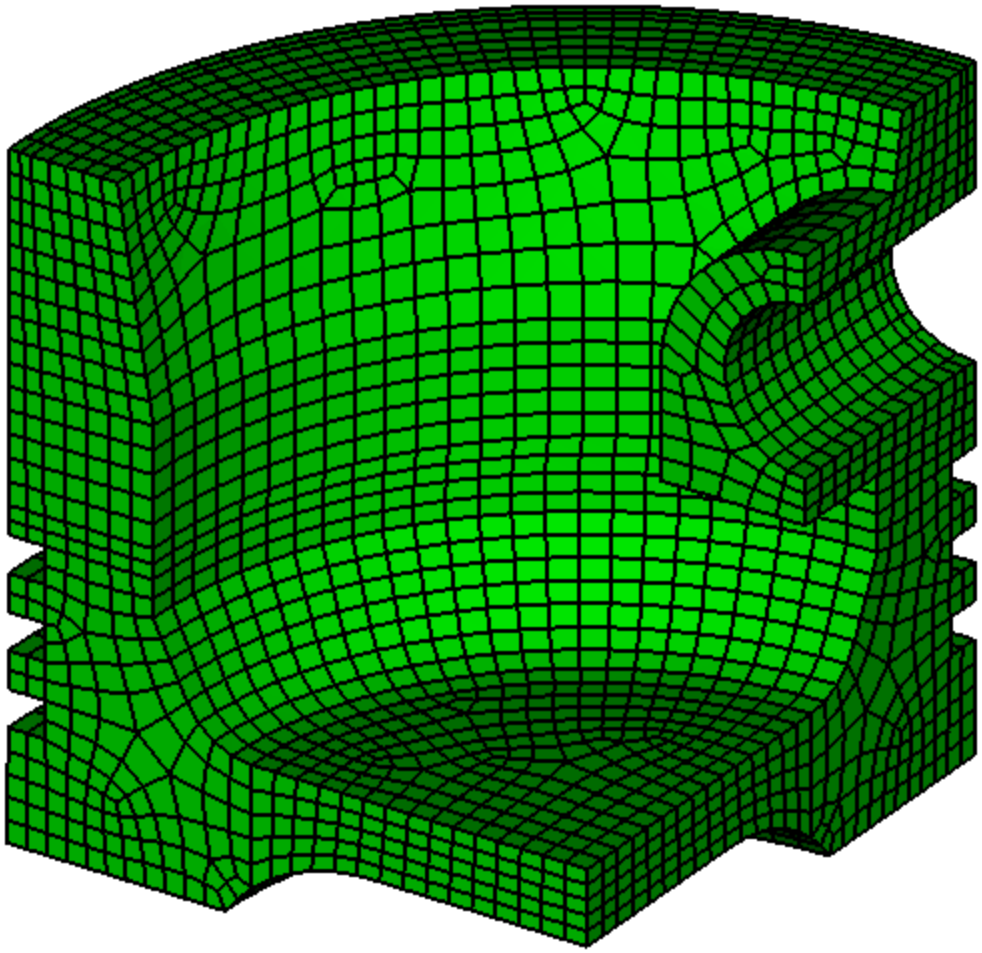


**Figure 2b. Output mesh of 6804 hexes,
after scaling with a multiplier of 2.0.**

*Tet Mesh Scaling*

Tetmesh scaling woks by using the original tet mesh as a background sizing mesh, with a size on each node to achieve the desired scale factor. The only command parameter relevant to tet mesh scaling is the **multiplier** parameter. All others are used for Hex Mesh Scaling.

*Hex Mesh Scaling*

Hex Mesh Scaling is more flexible than template-based global refinement methods, because it does not require that every element is refined, or refined in the same way. Instead, Hex Mesh Scaling decomposes the entire mesh into larger "blocks" of hexes, and then refines the blocks. In this way, Mesh Scaling supports increasing the element count by small multiplicative factors, e.g. 1.5, that are impossible with template based refinement. However, like template refinement, it can ensure that every location of the mesh is refined. These features can be useful for solution verification.

A traditional template-based refinement replaces each hexahedron with a 2x2x2 structured grid of hexahedra, increasing the element count by a factor of 8X. In contrast, Hex Mesh Scaling refines "blocks" of elements (not to be confused with Exodus element blocks). The block decomposition subdivides the entire mesh into structured (mapped) and swept blocks. A block may contain many elements, but is not allowed to cross geometric boundaries, boundary conditions, and loading constraints. For example, a block cannot have a curve or nodeset in its interior, nor hexes from multiple Exodus blocks. Blocks may be structured or logical sweeps. A structured block is restricted to be a grid of MxNxO hexes, so, its extent is limited by any surface nodes that do not have exactly four edges, etc. Hex Mesh Scaling remeshes the entire model conforming to the block decomposition, using the original mesh as a sizing function, multiplied by the scale factor.

#### Hex Scale Mesh Command Options

##### [minimum <value, default=1>]

The **minimum** parameter provides further control over the level of refinement. It is the minimum number of intervals added to each block-curve. Specifying **minimum 1**, which is the default, will guarantee that at least one interval is added to every element block in all 3 directions, which guarantees every part of the domain is scaled by at least a little bit. This can be "turned off" by specifying **minimum 0**.

Refinement and Coarsening Levels

The multiplier determines the target number of output hexes. There is no guarantee it will be achieved exactly. For **minimum 0** and small multipliers, there is no guarantee that every block will be refined in all 3 directions. This is because the target number of elements may be reached first. This may lead to unevenly distributed refinement, with jumps in adjacent element sizes.

An uneven distribution may also result if adjacent blocks have significantly different MxNxO intervals; this is common with the **legacy** option. For example, for a 1x10x12 block adjacent to a 6x10x12 block, Mesh Scaling could output 2x11x13 and 7x11x13 blocks. The M value of the first block has doubled, 2/1, while the M value of the second block has only increased by 7/6. Thus, the user may observe a jump in the lengths of adjacent edges.

To coarsen a mesh, specify a multiplier less than one. For example, a multiplier of 0.9 will attempt to decrease the element count by 10%. Each block side will have its intervals *decreased* by the minimum value. A block must have at least one interval, so how far the mesh can be coarsened is limited by the distance between mesh irregularities, geometry, boundary condition and loading constraints.

Solution Verification

Mesh Scaling is useful for solution verification, as it can easily generate a series of similar meshes of increasing mesh density. For best results, generate each mesh in the series by scaling the original mesh, rather than scaling the previous mesh of the series. It is suggested that each mesh uses a **multiplier** at least 2X larger, and a **minimum** at least one more, than the prior mesh. Small multipliers alone are unlikely to produce sufficient changes for solution verification. The **minimum** is especially useful for ensuring changes

in regions that are initially coarse. A good set of (**multiplier, minimum**) parameters follows:

**(2X, 1) (4X, 2) (8X, 3) (16X, 4), (prior *2, prior +1), etc.**

A large **minimum** can cause quality problems and generate too many elements. For example, the **minimum** in the parameter series **(2X, 1) (3X, 2) (4X, 3) (5X, 4) (6X, 5) (7X, 6) (8X, 7) etc.** would likely be too aggressive. It would produce many more elements than the specified multiplier, potentially causing poor element quality or even mesh scaling failure. For a slowly increasing set of multipliers, a less aggressive minimum series is recommended, such as

**(2X, 1) (3X, 1) (4X, 2) (5X, 2) (6X, 2) (7X, 2) (8X, 3) etc..**

**[{SWEPT_BLOCKS | legacy | maintain_structure}]**

There are three major block decomposition variations to choose from. To understand their differences, one must first understand the two types of blocks: "structured" and "swept" blocks. A structured block is a MxNxO structured grid. A swept block is a single-source to single-target sweep of some subset of a single volume. That is, it is composed of a single surface of quad elements, projected some number of layers to form hexes.
The block decomposition options are **swept_blocks** (default), **maintain_structure** and **legacy**. For **legacy**, only structured blocks are used. For **swept_blocks** and **maintain_structure**, the decomposition constructs large swept blocks wherever logical sweeps can be identified, and structured blocks otherwise. The main difference is that **swept_blocks** remeshes the source surfaces of swept blocks from scratch. In contrast, **maintain_structure** partition each swept block into structured sub-blocks, and remeshes by selectively refining those sub-blocks. Thus **swept_blocks** may change the number and relative location of irregular nodes, whereas **maintain_structure** keeps them the same.
Typically, **swept_blocks** and **maintain_structure** provide smoother, more evenly distributed refinements compared to **legacy**. This is because with swept blocks, there are typically significantly fewer blocks in the decomposition. Having fewer blocks increases the likelihood that each block will receive at least some refinement before the multiplier is reached.
However, **legacy** and **maintain_structure** provide element orientations and structure closer to the original mesh than **swept_blocks**. This is because structured blocks maintain the irregular nodes.
Often **maintain_structure** provides both element orientations closer to the original mesh and a smoother, more evenly distributed refinement. Its structured blocks preserve orientations and structure. Its swept blocks provide the freedom to distribute changes, and smooth the mesh, across its structured sub-blocks.

**[force_structured in {[volume <ids>] [surface <ids>]}]**

In some cases, the user may want to use swept blocks in only some parts of the model. The original mesh may have small regions with carefully constructed meshes. Using

**swept_blocks** can destroy these constructions, replacing them with pave-and-sweep meshes. These constructions can be preserved by specifying **force_structured** for the surfaces and volumes containing them.
For example, see Figures 3, 4 and 5. In Figure 3 surface 108 was meshed with great care to ensure a structured mesh around the holes, while surface 34 was meshed with paving. Since surface 108 has irregular nodes, it appears to Mesh Scaling as the source surface of a swept block. Figure 4 illustrates the resulting mesh from the command **"scale mesh multi 2"**. Notice the structured meshes around the holes have been replaced with a standard paved mesh. Figure 5 illustrates preserving the structured holes of surface 108 while allowing surface 34 to be repaved. The command was **"scale mesh multi 2 force_structured in surface 108"**. A different mesh would result from the command **scale mesh multi 2 force_structured in volume 1** , because this would also preserve the irregular nodes in surface 34, resulting in more blocks and a less smooth mesh.

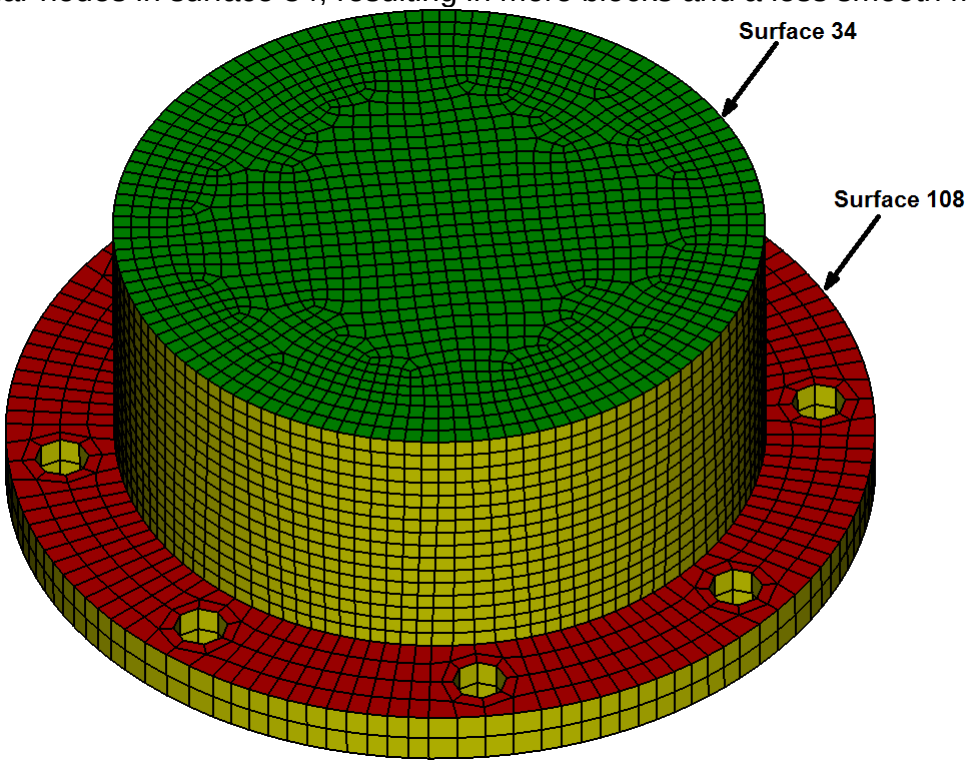



**Figure 3. Input mesh. Surface 108's mesh has desired structure and irregular nodes. Surface 34 contains a paved mesh.**

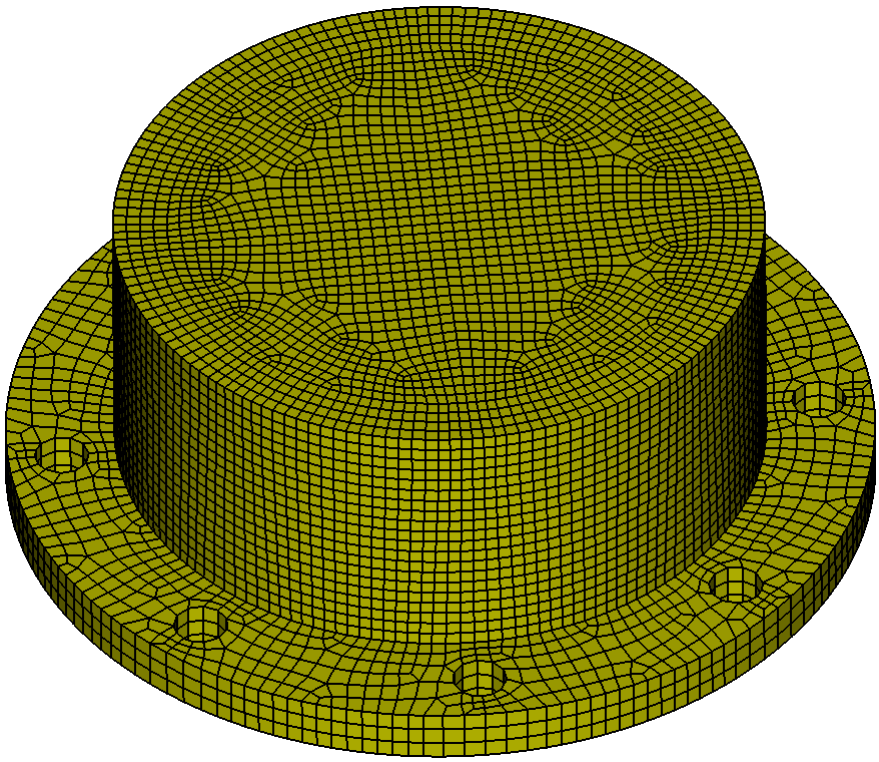


**Figure 4. Output mesh from the command "Scale Mesh Multi 2". The mesh on surface 108 is replaced with a paved mesh.**

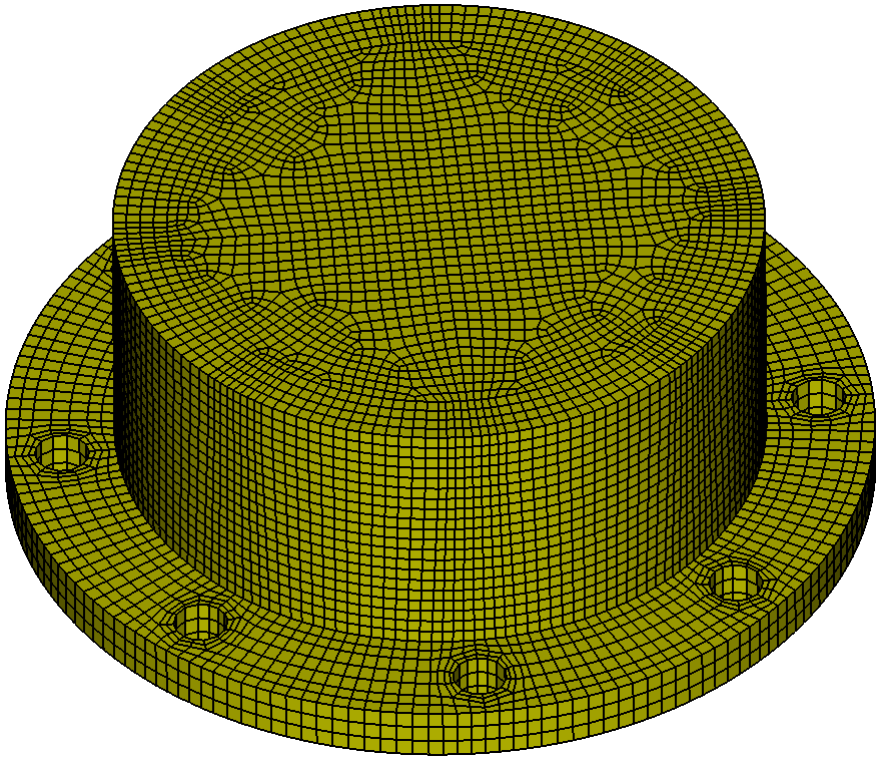


**Figure 5. Output mesh from the command "Scale Mesh Multi 2 force_structured in Surface 108". The desired features are maintained, while swept blocks are used in unimportant regions.**

**[thin_gap_intervals <value, default=2>] [fix_all_gaps] [max_feature_length]**

For the **maintain_structure** option, the **thin_gap_intervals** parameter determines how thin gaps are defined. If two disjoint curves of a surface come close together, the space between them is considered a "thin gap" if the number of intervals across that space is at most **thin_gap_intervals**. Mesh scaling gives high priority to adding intervals within thin gaps. By default, the position of *some* nodes within thin gaps are fixed to help reduce skew. The **fix_all_gaps** option fixes *all* nodes in thin gaps. Features, e.g. curves and gaps, longer than **max_feature_length** intervals will be split into multiple features. This results in fixing additional nodes along the feature to help reduce skew.
In general, **maintain_structure** should result in a smoother scaled mesh than the other algorithms, however, sometimes it can introduce some skew in the scaled elements. If skew is introduced by scaling using **maintain_structure**, try either increasing the **thin_gap_intervals** parameters, specifying **fix_all_gaps**, decreasing the **max_feature_length** parameter, or all three.

**[feature_angle <value>] [max_aspect_ratio <value> in volume <ids>]**

The **feature_angle** and **max_aspect_ratio** options affect the formation of swept blocks. These are alpha, experimental commands.

**[smooth_volume {ON|off}]**

If **smooth_volume** is on, then the volume mesh is smoothed as a post-process if it has poor quality elements, and smaller minimum quality than the original mesh. By default, smooth_volume is on.

## Block Repositioning

A capability to reposition blocks is provided. This capability will retain all the current connectivity of the nodes involved. Unlike the Nodeset Move command, this command works for blocks containing free mesh (mesh not owned by geometry.)

**Block <id_range> Move <delta_x><delta_y><delta_z>**

## Node and Nodeset Repositioning

A capability to reposition nodesets and individual nodes is provided. This capability will retain all the current connectivity of the nodes involved, but it cannot guarantee that the new locations of the moved nodes do not form intersections with previously existing mesh or geometry. This capability is provided to allow the user maximum control over the mesh model being constructed, and by giving this control the user can possible create mesh that is self-intersecting. The user should be careful that the nodes being relocated will not form such intersections.

The user can reposition nodes appearing in the same nodeset using the **NodeSet Move** command. Moves can be specified using either a relative displacement or an absolute position. The command to reposition nodes in a nodeset is:

**Nodeset <nodeset_list> Move <delta_x> <delta_y> <delta_z>**

**Nodeset <nodeset_list> Move To <x_pos> <y_pos> <z_pos>**

The first form of the command specifies a relative movement of the nodes by the specified distances and the second form of the command specifies absolute movement to the specified position. The third form of the command specifies a displacement with respect to a specified surface normal.
Individual nodes can be repositioned using the **Node Move** command. Moves are specified as relative displacements. The command syntax is:

**Node <range> Move <delta_x> <delta_y> <delta_z>**

**Node <range> Move {[X <val>] [Y <val>] [Z <val>]}**

**Node <range> Move Normal to Surface <id> distance <val>**

**Node <range> Move Closest Surface <id> distance <val>**

Nodes can also be repositioned using a location or direction specification. See Location, Direction, and Axis Specification for details on the location and direction specification. The command syntax is:

**Node <range> Move Location <options>**

**Node <range> Move Direction <options>**

See also Transforming Mesh Coordinates.

## Mesh Pillowing

Mesh pillowing is a mesh refinement technique that inserts a layer or 'pillow' of elements around the boundary of an enclosed mesh. It can be used to improve mesh quality while preserving the outer boundary of the selected element set. Mesh Pillowing can be used to quickly perform a number of meshing tasks, such as inserting a uniform boundary layer a specified distance from an outer boundary, or inserting a ring of elements around a hole.

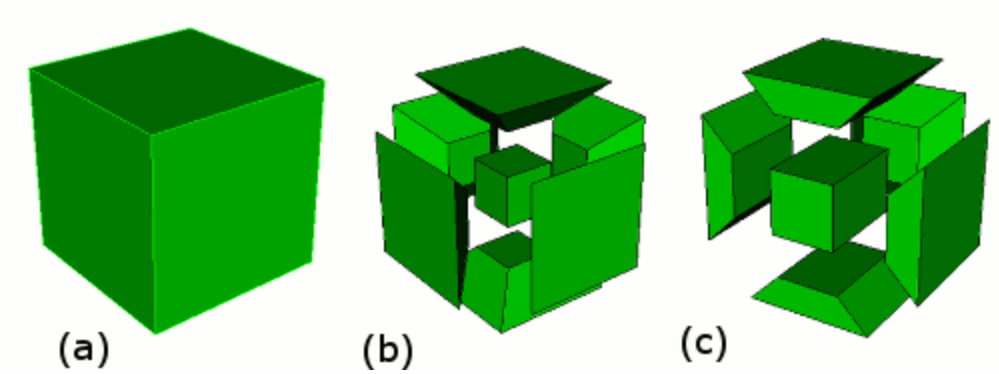



**Figure 1: A single hex before (a) and after (b) a pillow operation. The far right (c) depicts a pillow operation with the front surface designated as a 'through' surface.**

During a typical pillow operation, the user selects a set of elements, called a 'shrink set', to define what elements will be operated on. All elements on the outer boundary of the shrink set are then shrunk towards the center of the set. New elements are then created to fill the gap between the original boundary and the shrunk boundary. The newly created elements form the pillow around the selected shrink set. Figure 1a and 1b show an example of a pillow operation performed on a single hex. Geometry surfaces, or mesh element faces can be specified as **through** surfaces for the pillowing operation. This means that the pillow will extend through the selected surfaces, and no new elements will be created along them. Figure 1c shows the effect of pillowing a single hex with one surface selected as a through surface.

In some cases a shrink set may not be valid due to the geometry of a specific region. As the exterior nodes of the shrink set move towards the middle they must be able to maintain appropriate geometric associations. Nodes on vertices must move along curves, nodes on curves must move along surfaces. If there are multiple curves or surfaces along which an exterior node might travel, then the ownership is ambiguous and the pillowing will fail.

Using the optional **distance** keyword with a specified value allows manual control of the distance that each boundary element is shrunk towards the center of the shrink set. If no distance value is specified, an appropriate value is calculated for each element. If a distance value is specified, all newly created nodes will have their position fixed by default. This allows the user to smooth the mesh without altering the node positions of the newly created hexes. If the optional **unfix_nodes** keyword is used, this default behavior is changed, and any smooth operations will alter the newly created node locations. By default, a smooth operation is automatically performed following any pillow operation unless the optional **no_smooth** keyword is used.

Similar analogous commands are available for creating a pillow around a set of two dimensional faces.

**Syntax:**

```
Pillow Hex <ids> [ Through { [Surface <ids>][Face <ids>][Tri <ids>] } ] [
Distance <value> ] [ Unfix_nodes ] [ No_smooth ]
```

**Pillow Face <ids> [ Through Curve <ids> ] [ Distance <value> ] [ Unfix_nodes ] [No_smooth]**

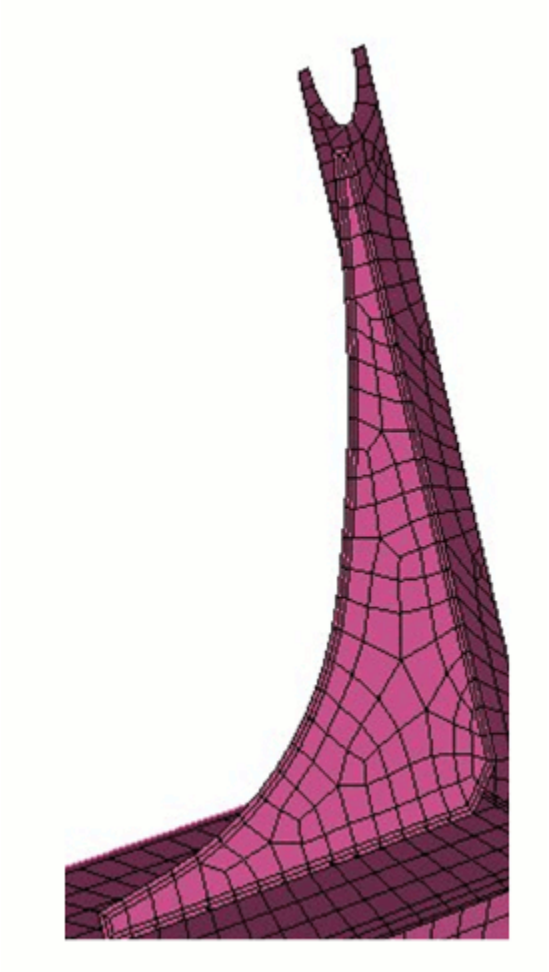


**Figure 2: Example model using pillow operations to create ordered nodes a specified distance around the boundary of a mesh.**

## Mesh Column Operations

Column operations allow users direct control over the mesh connectivity while maintaining full-geometric associativity. Often, hex meshing schemes such as sweeping and mapping result in mesh topology forced into unnatural shapes, such as a square shaped source surface mesh getting swept into a circular target surface. Often forcing meshes into shapes like this results in poor element quality because of non-optimal element angles. The Column commands allow users to directly modify mesh topology to make minor tweaks to a mesh improving element quality. Column operations are almost always followed by smoothing to enable element quality improvement.

Cubit provides tools to perform insertion, deletion, swapping, grouping, and drawing of hex columns.

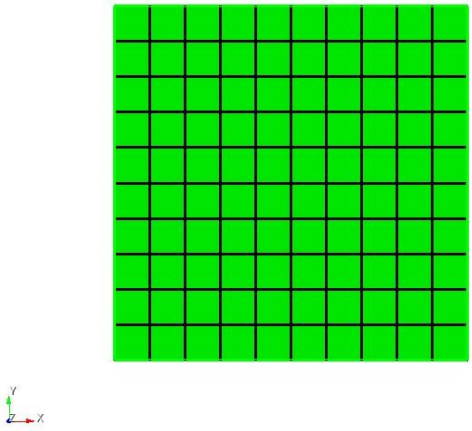
- Column Insertion
- Column Deletion
- Column Swapping
- Column Groups
- Drawing Columns


*Column Insertion*

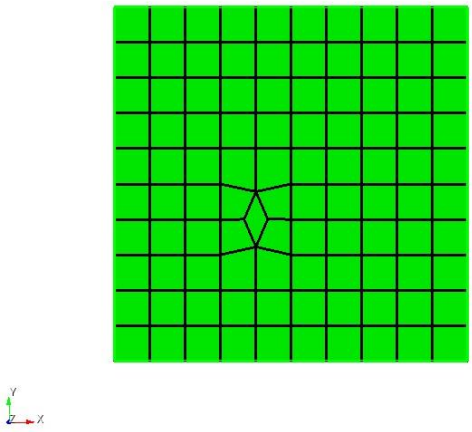
A single column can be inserted into the mesh by using the following command:

**column open node <center node id> <orientation node ids>**

For example, given the following meshed brick:



we issue the command, **column open node 89 88 90 ,** to get this result:

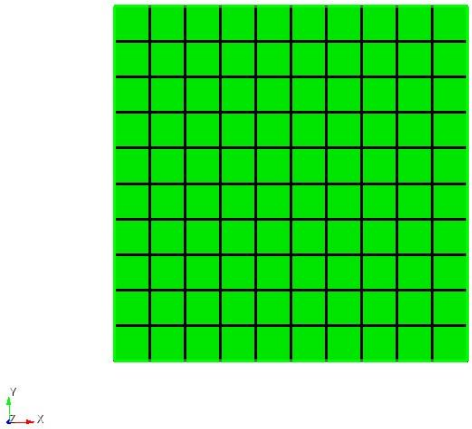


*Column Deletion*

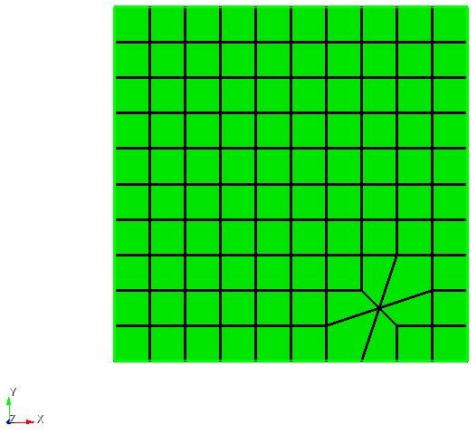
Columns can be removed with neighboring columns being joined together using **collapse** commands. Collapse commands are of two types: interior and boundary.
For interior node collapse, the two nodes which are opposite on a face are combined together. The column associated with the face is removed. Use the following command:

**column collapse node <opposite node ids>**

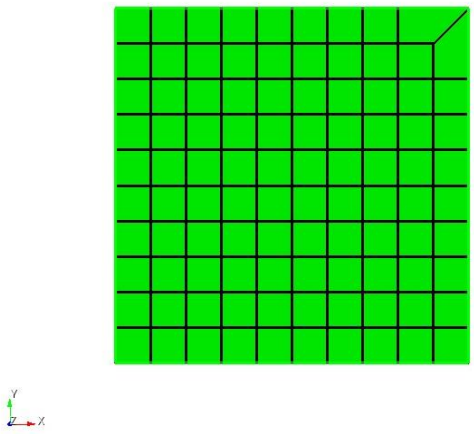
For example, given the following meshed brick:



we issue the command, **column collapse node 51 59**, to get this result:



The column collapse command can be used with boundary nodes. For example, we issue the command, **column collapse boundary node 13 2 11**, to get this result:

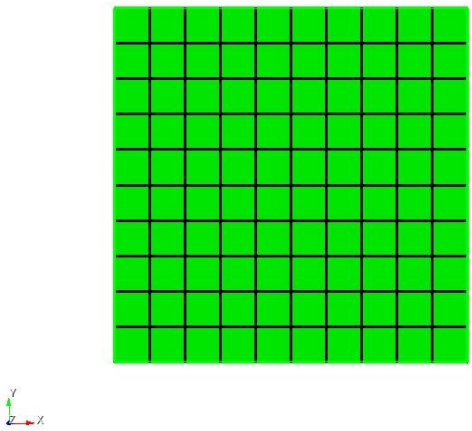


*Column Swapping*

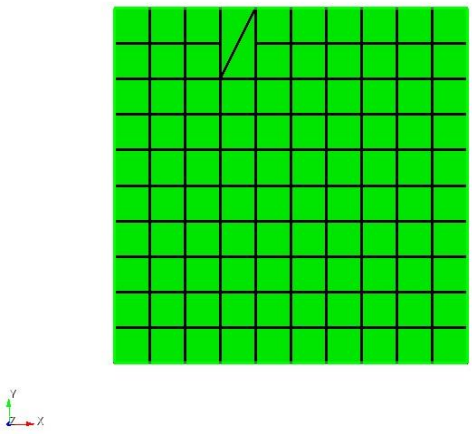
Faces between two hex columns can be swapped using the following command:

**column swap node <old edge node ids> <new edge node ids>**

For example, given the following meshed brick:



we issue the command, **column swap node 103 94 102 18**, to get this result:



*Column Groups*

A group consisting of hexes that comprise a column can be created using the following command:

**column { face <id> | edge <id1> <id2> | hex <id1> <id2> } group**

*Drawing Columns*

Columns can be drawn using the following command:

**draw column { face <id> | edge <id1> <id2> | hex <id1> <id2>}**

## Removing Intersecting Mesh

In addition to finding mesh intersection , Cubit® can also remove mesh intersection by slightly repositioning the nodes of intersecting elements. Nodes are moved along surface normals just enough so that the intersection is zero or undetectable. The command syntax is:

**Remove Mesh Intersection {Block|Body|Volume} <id_list> [with {Block|Body|Volume} <id_list>] [low <value=0.0001>] [high <value>] [detail]**

The **low** and **high** options set how much cumulative intersection should be detected. A low value of 0.1 would ignore elements that do not intersect more than 10% of their volume. Similarly, a high value of 0.5 would discard elements that intersect more than 50% of their volume. Both low and high can be used simultaneously. The default for the **low** value is 0.0001

The **detail** option provides information, printing the ten nodes moved the largest distances, providing users an idea of the greatest changes in the mesh.

```
Largest node movements:
Node 165 moved a distance 0.003889
Node 170 moved a distance 0.003886
Node 1091 moved a distance 0.003861
Node 163 moved a distance 0.003832
Node 1100 moved a distance 0.003794
Node 1090 moved a distance 0.003739
Node 1101 moved a distance 0.003728
Node 117 moved a distance 0.003701
Node 164 moved a distance 0.003672
Node 121 moved a distance 0.003483
```

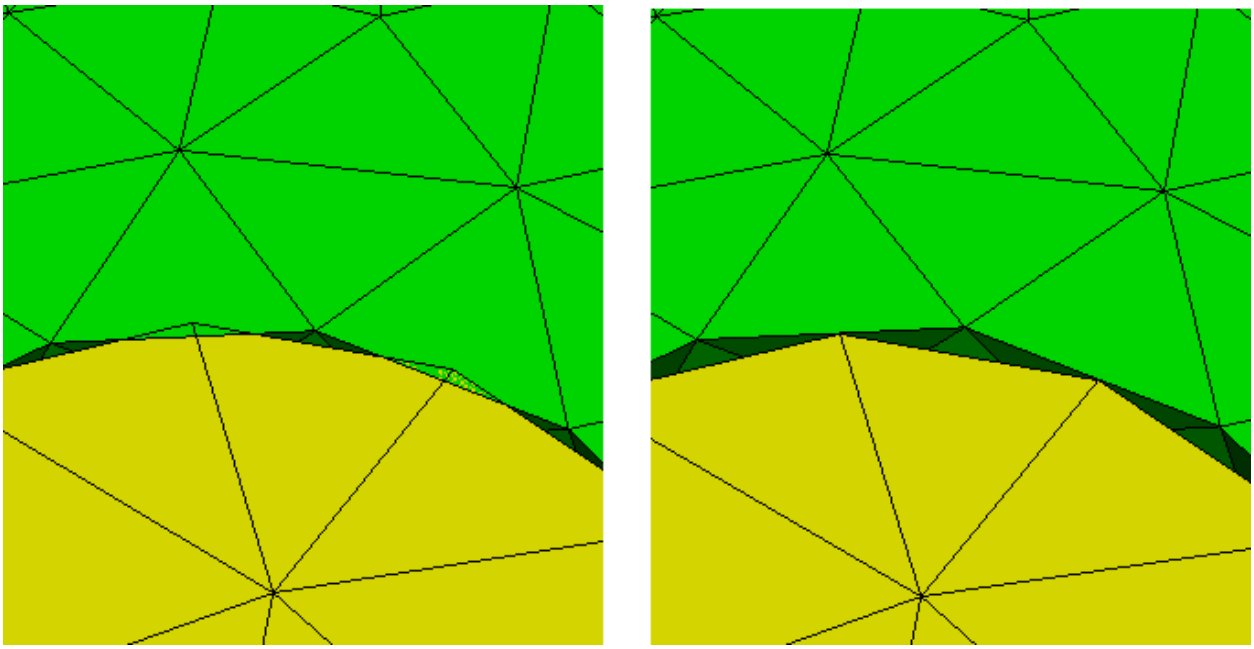


**Figure 1. "Before and after mesh intersection removal"**

**Note:**

- Elements that intersect more than 50% of their volume will not be modified, as element quality could degrade substantially. Users will be notified of these elements through a printed warning.
- The shape of higher-order mesh elements is considered when computing intersection.

# mesh import

## Importing a Mesh

- Importing 2D Exodus II Files
- Importing Exodus II Files
- Importing Patran Files

- Importing I-DEAS Files
- Importing Abaqus Files
- Importing Nastran Files
- Importing Fluent Files

ExodusII finite element data files can be imported into CUBIT. Several options for importing the mesh are available, (including mesh transformations):

- Importing a free mesh without geometry.
- Importing a free mesh and associating the mesh with ACIS-based geometry currently residing in CUBIT.
- Importing a 2D mesh and constructing ACIS-based Geometry
- Importing a mesh and constructing Mesh-Based Geometry from dihedral angles and boundary conditions.
- Importing a preview mesh.

## Importing Fluent Files

The command to import a mesh from a fluent format file is:

**Import Fluent [Mesh Geometry] '<input_filename>' [Feature Angle <angle>] [nobcs]**

Including the keyword **Mesh Geometry** will instruct CUBIT to create mesh-based geometry. This will provide the user with the ability to remesh geometric entities. If the user does not import with the Mesh Geometry flag, he will have to tell CUBIT to draw the mesh after the import is done in order to view it.
The **Feature Angle** is used when building the surface topology to determine when to split a surface into two surfaces. If the angle between two neighboring element normals is less than Feature Angle, then the two elements will be placed on separate surfaces. If the keyword Feature Angle is not supplied, the default 135 degrees is used. For a description of importing mesh geometry see Importing Exodus II Files.
The keyword **nobcs** can be included if boundary conditions are not to be imported.
It should be noted that CUBIT sometimes cannot successfully generate mesh-based geometry for complex models. If this occurs, import the mesh without the Mesh Geometry flag, and draw the mesh to view it.

## Importing 2D Exodus Files

CUBIT has a limited capability to create ACIS Geometry from 2D ExodusII finite element mesh files. (For a more general capability, see the Import Mesh Geometry command, which will create Mesh-Based Geometry).
To import a 2D Exodus II file and create ACIS geometry, the following command can be used:

**Import Free Mesh '<filename>' {Time <t> | Step <step#> | Last}**

CUBIT can create ACIS geometry from 2D Exodus II data files (4, 8, or 9 node QUAD or SHELL element types) that do not have enclosed voids (holes surrounded by mesh) and which were originally generated with CUBIT and exported to ExodusII with the Nodeset Associativity option set to on. The Nodeset Associativity command records the topology of the geometry into special nodesets which allow CUBIT to reconstruct a new solid model from the mesh even after it has been deformed. The new solid model of the deformed geometry can be remeshed with standard techniques or meshed with a sizing function that can also be imported into CUBIT from the same ExodusII file. CUBIT's implementation of the paving and triadvance algorithms can generate a mesh following a sizing function to capture a gradient of any variable (element or nodal) present in the ExodusII file.
In order for this feature to be effective, the following commands must be issued when the mesh is exported and later imported:

```
nodeset associativity on

set associativity complete on
```

The first command ensures that the geometry will be correctly recovered from the mesh, while the second ensures that boundary condition and material IDs will be recovered.

## Importing Abaqus Files

The command to import a mesh from an Abaqus format file is:

```
Import Abaqus [Mesh Geometry] '<input_filename>' [Feature Angle <angle>]
[Nobcs]
```

Including the keyword **Mesh Geometry** will instruct CUBIT to create mesh-based geometry. This will provide the user with the ability to remesh geometric entities. If the user does not import with the Mesh Geometry flag, he will have to tell CUBIT to draw the mesh after the import is done in order to view it.
The **Feature Angle** is used when building the surface topology to determine when to split a surface into two surfaces. If the angle between two neighboring element normals is less than Feature Angle, then the two elements will be placed on separate surfaces. If the keyword Feature Angle is not supplied, the default 135 degrees is used. For a description of importing mesh geometry see Importing Exodus II Files.
The keyword **nobcs** can be included if boundary conditions are not to be imported.
The Abaqus importer can import the following Abaqus file formats: flat file, part-independent, and part-dependent.
It should be noted that CUBIT sometimes cannot successfully generate mesh-based geometry for complex models. If this occurs, import the mesh without the Mesh Geometry flag, and draw the mesh to view it.
To list Abaqus cards supported by Cubit:

**List Abaqus Import Cards**

This command will list out all supported Abaqus cards that CUBIT can interpret.

**Table 1. Supported Element Types**

| | 1st Order | 2nd Order |
|---|---|---|
| **Triangle** | S3<br>CAX3<br>CPE3 | STRI65<br>CAX6<br>CPE6 |
| **Quadrilateral** | S4<br>CAX4<br>CPE4 | S8<br>CAX8<br>CPE8 |
| **Tetrahedron** | C3D4 | C3D10 |
| **Hexahedron** | C3D8 | C3D20 |
| **Line Element** | B21<br>B31<br>T2D2<br>T3D2<br>SPRINGA<br>SPRING1<br>SPRING2 | B22<br>B32<br>T2D3<br>T3D3 |

See http://www.simulia.com/ for more information on the ABAQUS file format.

## Importing Exodus II Files

- Importing a Free Mesh without Geometry
- Importing a Free Mesh onto Existing Geometry
- Creating Mesh-based Geometry on Import
- Importing a Lite Mesh

The commands to import meshes from an Exodus II format file are:

**Import Mesh '<exodusII_filename>' No_Geom**
**[Block <block_ids>] [block_name <names>]**
**[{genesis_collision_fail|COMBINE_GENESIS_IDS|combine_genesis_names|unique_genesis_ids}] [block_offset <value>] [sideset_offset <value>]**
**[nodeset_offset <value>] [{node_id_collision_fail|UNIQUE_NODE_IDS}]**
**[{element_id_collision_fail|UNIQUE_ELEMENT_IDS}] [node_offset <value>]**
**[element_offset <value>]] [nodal_var{<string>|all_nodal_vars}]**
**[element_var<string>|all_element_vars}] [group_name '<free_elements>']**
**[[Time <time>|Step <step>|Last] [Scale <value>]]**

**Import Mesh '<exodusII_filename>'**
**[Block <block_ids>] [block_name <names>]**
**[{genesis_collision_fail|COMBINE_GENESIS_IDS|combine_genesis_names|unique_genesis_ids}] [block_offset <value>] [sideset_offset <value>]**

**[nodeset_offset <value>] [{node_id_collision_fail|UNIQUE_NODE_IDS}] [{element_id_collision_fail|UNIQUE_ELEMENT_IDS}] [node_offset <value>] [element_offset <value>]] [{Group|Body|Volume|Surface|Curve|Vertex} <id_range> | Preview]**

**Import Mesh Geometry '<exodusII_filename>' [Block <id_range>|ALL] [block_name <names>] [{genesis_collision_fail|COMBINE_GENESIS_IDS|combine_genesis_names|unique_genesis_ids}] [block_offset <value>] [sideset_offset <value>] [nodeset_offset <value>] [{node_id_collision_fail|UNIQUE_NODE_IDS}] [{element_id_collision_fail|UNIQUE_ELEMENT_IDS}] [node_offset <value>] [element_offset <value>]] [nodal_var{<string>|all_nodal_vars}] [element_var<string>|all_element_vars}] [Use [NODESET|no_nodeset] [SIDESET|no_sideset] [Feature_Angle <angle>] [Surface_Feature_Angle <angle>] [LINEAR|Gradient|Quadratic|Spline|Acis] [Deformed {Time <time>|Step <step>|Last} [Scale <value>] ] [MERGE|No_Merge] [Merge_nodes <tolerance>] [{midnode_correction|NO_MIDNODE_CORRECTION}]**

**Import Mesh '<exodusII_filename>' Lite [{genesis_collision_fail|COMBINE_GENESIS_IDS|combine_genesis_names|unique_genesis_ids}] [block_offset <value>] [sideset_offset <value>] [nodeset_offset <value>] [{node_id_collision_fail|UNIQUE_NODE_IDS}] [{element_id_collision_fail|UNIQUE_ELEMENT_IDS}] [node_offset <value>] [element_offset <value>]]**

Related Commands:

**Import Mesh Geometry (options)**

**Import Free Mesh (2D)**

**Delete Mesh Preview**

**Export [ Genesis | Mesh ] '<filename>'**

**List Import Mesh NodeSet Associativity**

**List [Export Mesh] NodeSet Associativity**

**[Set] Import Mesh NodeSet Associativity [ON|off]**

**[Set] [Export Mesh] NodeSet Associativity [on|OFF]**

**Transforming Mesh Coordinates**

**Set Import Mesh [Vertex] [Curve] [Surface] Tolerance <distance>**

**Set Import Mesh NodeSet Order [On|Off]**

**List Import Mesh NodeSet Order**

*Common Options*

#### Specifying a Portion of the Mesh to be Imported

The **Block** option in the **Import Mesh** command indicates that only the specified element block should be imported from the Exodus II file. The blocks are specified by ID. The **block_name** option can be used to import only the named blocks. Multiple blocks can be imported by making a comma separated list of names. The **block** and **block_name** options can be used together to specify some blocks by ID and others by name. If the **block** or **block_name** options are not specified, then the entire mesh file is read. These options are not yet supported for lite imports, which currently imports the entire mesh.

#### Combine Genesis IDs Option

The **combine_genesis_ids** option, is used to combine blocks where the IDs in the session and the file being imported are identical. This can occur when importing into an active session where Cubit IDs have already been assigned. The default behavior is to combine genesis entities based on IDs. If two entities have differnet IDs, but the same names, they will not be combined, and the import will fail.

#### Combine Genesis Names and Nodes IDs Options

The **combine_genesis_names** option, is used to combine blocks where the names in the session and the file being imported are identical. This can occur when importing into an active session where names have already been assigned. If two entities have different names, but identical IDs, they will not be combined, and the import will fail.
The **combine_node_ids** option is only available in the 'lite' form of the 'import mesh' command. This option combines nodes, where the ids in the session and the file being imported are identical, regardless of location (coincidence).

#### Unique Genesis IDs Option

The **unique_genesis_ids** option is used to renumber genesis entities from the genesis file in the case that ID overlap exists when importing into Cubit. The incoming genesis entities are kept unique and are not combined with genesis entities already in the session. In case of colliding IDs, a report displayed in the command window showing the original and new IDs. This renumbering can occur when importing into an active session where Cubit IDs have already been assigned. If an entity being imported has the same name as one already in the session, the entity being imported will be renamed with a number suffix '_N'.

#### Genesis Collision Fail Option

The **genesis_collision_fail** option allows the user to prevent the genesis file import if any incoming genesis entity IDs or names are already used by genesis entities in the session. This can occur when importing into an active session where Cubit IDs have already been assigned.

#### Block_Offset, Sideset_Offset, Nodeset_Offset Options

The **block_offset**, **sideset_offset**, and **nodeset_offset** options may be used to modify the IDs of genesis entities being imported. The provided value will be added to the ID in the file.

#### Unique Node and Element IDs Option

The **unique_node_ids**, and **unique_element_ids** options are used to automatically renumber nodes and elements from the genesis file in the case that ID overlap exists when importing into Cubit. If there are no overlaps, the IDs in the file will be preserved. This can occur when importing into an active session where Cubit IDs have already been assigned.

#### Element and Node ID Collision Fail Option

The **node_id_collision_fail**, and **element_id_collision_fail** options allows one to prevent the genesis file import if any incoming node or element IDs are already used by mesh entities in the session. This can occur when importing into an active session where Cubit IDs have already been assigned.

#### Node_Offset and Element_Offset Options

The **node_offset**, and **element_offset** options may be used to modify the IDs of nodes and elements being imported. The provided value will be added to the ID in the file.

#### Node/Element Variables

The **nodal_var** and **element_var** options allow nodal and element variable information respectively to be imported from the Exodus file. To import a specific varible, a name can be specified. Using the ***_all** option imports all variable information. By default, no variable information is imported.

*Importing a Free Mesh Without Geometry*

The command to import a free mesh from an Exodus II format file without mesh-based geometry is:

```
Import Mesh '<exodusII_filename>' No_Geom [group_name '<free_elements>']
[[Time <time>|Step <step>|Last] [Scale <value>]]
```

When a free Exodus II mesh is imported into Cubit, it contains no geometric or topological information. Previously, the user could either associate that mesh with existing geometry, or build mesh-based geometry to fit the mesh. A third option, as of Cubit 11.1, allows the user to retain the disassociated mesh as a free mesh inside Cubit.
A free mesh may be modified as described in the Free Mesh section of the documentation. This includes limited access to smoothing, renumbering, transformations, refinement, mesh quality, and other mesh centric operations.
When an Exodus II File is imported as a free mesh, Cubit will automatically create a group called "free_elements" to contain the free mesh elements. The 'group_name' option can be used to give the group a different name.
Deformation information can be read in via the **Time/Step/Last** and **Scale** parameters.
**Note:** The **Import Mesh <filename> No_Geom** command is not to be confused with the **Import Free Mesh** command which applies only to 2D Exodus II Files.The term "Free Mesh" in both places of the documentation refers to the same thing - a mesh without geometry. However, in the case of all other import mesh commands, the imported free mesh ends up associated with geometry. The **Import Mesh <filename> No_Geom** is the only way to import a free mesh that remains disassociated from geometry.

*Importing a Mesh Onto Existing Geometry*

The command to import a free mesh from an Exodus II format file and associate it with existing geometry is:

```
Import Mesh '<exodusII_filename>'
[{Group|Body|Volume|Surface|Curve|Vertex} <id_range> | Preview]
```

The user can import a mesh from an Exodus II file and associate the mesh with matching geometry. The resulting mesh may then be manipulated normally. For example, the mesh may be smoothed or portions of it deleted and remeshed. The user can save their work by exporting the geometry and mesh, and then restore the geometry and mesh later. In some cases, saving and restoring can be faster or more reliable than replaying journal files.
Saving and importing a mesh may be useful for teams working on creating a conforming mesh of a large assembly so that they can pass information to one another. For example, a team member can export the mesh on the surfaces between two parts, and another team member import the mesh for use on an adjoining part of the assembly.
As of cubit version 7.0, any higher order elements, block definitions, nodesets, and sidesets are retained on import.

#### Importing a Mesh with Nodeset Associativity

Meshes can be imported into Cubit that contain nodeset associativity data used for defining finite element boundary conditions. If an exported Cubit mesh is going to be imported back onto the same geometry, then before exporting the user should issue the following command:

```
set export mesh nodeset associativity on
```

This causes extra nodeset data to be written, which associates every node to a geometric entity, resulting in an import which is more reliable. When importing, if the user does not want to use the nodeset associativity data that exists in a file, then before importing the following command should be used:

**set import mesh nodeset associativity off**

The user may wish to turn geometry associativity off if, for example, the geometry is no longer identical as a result of curves being composited, or Cubit names changed due to a ACIS version changes.

#### Importing a Mesh onto Modified Geometry

Although there are some exceptions, Cubit requires that the mesh be imported onto the same geometry from which it was exported.
Since merge information is not stored with the ACIS representation, care should be taken that the geometry is merged the same way on export and import of the mesh. If not, importing the mesh one block at a time in successive commands may increase the chance of a successful import, at the cost of more memory and time.
Between exporting and importing a mesh, the geometry may be modified slightly by compositing entities. Mesh import will, however not be successful if entities are partitioned or a body is webcut. In some cases mesh import may be successful on modified geometry if the new vertices match up exactly with nodes of the mesh, and the new curves match up exactly with edge chains of the mesh. Unless this criteria is met, associating the mesh with the geometry will be unsuccessful.

#### Mesh Import Tolerance

To change the tolerance with which imported mesh must line up with geometry issue the command:

**Set Import Mesh [Vertex] [Curve] [Surface] Tolerance <distance>**

#### Specifying a Portion of the Mesh to be Imported

The **Block** option in the **Import Mesh** command indicates that only the specified element block should be imported from the Exodus II file. In the same manner, the **Volume** and other geometry options provide a way to import the nodes and element on the indicated geometry. If neither a **block** nor a **geometry entity** is specified, then the entire mesh file is read.
If a **block** is specified without specifying a **geometry entity**, associativity or proximity is used to determine which volume the block elements should be associated with. If a **block** and a **volume** are specified, the block elements are associated with the specified volume, provided they actually match. If a **volume** is specified without a **block**, associativity data is used to find a block corresponding to the given volume.

#### Nodeset Ordering

If the Import mesh NodeSet Order flag is on, the nodesets will be read in a manner which allows them to be associated with existing geometry. This means the nodesets are assumed to be in ascending order. If the flag is set to false, the goemetry nodesets in imported mesh files are assumed to be in random order. This value is on by default, and should not need to be changed by the user.

*Creating Mesh-Based Geometry on Import*

Cubit's mesh generation tools require an underlying geometry representation. In most cases, the ACIS solid modeling engine, compiled with Cubit, is used to represent the geometry. However, in some cases, an ACIS representation is not available, and a previously developed finite element mesh is the only available representation of the model. In order to utilize Cubit's mesh generation tools, the **import mesh geometry** command provides an option for creating geometry directly from the finite element mesh. The **import mesh geometry** command will create a new volume for every block defined in the Exodus II file. It will also create curves, surfaces and vertices at appropriate locations on the model based on dihedral angles (also called feature angles) and assigned nodesets and/or sidesets. The mesh used to construct the geometry will be owned by the new geometric entities. This means that the mesh can be deleted, remeshed, or smoothed using any of Cubit's meshing tools by simply using the new geometry definition. Cubit will assign appropriate intervals to the new curves as well as determine an acceptable meshing scheme for surfaces and volumes.
The command to import a finite element mesh from an ExodusII format file and generate geometry from the mesh is:

**Import Mesh Geometry '<exodusII_filename>'**
**[Block <id_range>|ALL] [Use [NODESET|no_nodeset] [SIDESET|no_sideset]**
**[Feature_Angle <angle>] [Surface_Feature_Angle <angle>]**
**[LINEAR|Gradient|Quadratic|Spline|Acis] [Deformed {Time <time>|Step**
**<step>|Last} [Scale <value>] ] [MERGE|No_Merge] [Merge_nodes <tolerance>]**

#### File Name

Type the name of file to import in single quotation marks. The file must reside in the current directory. For information on changing the current directory, see Cubit environment commands. To list all the files in the current directory, type **ls** at the command prompt.

#### Blocks

Use this option to select the specific blocks to be imported from the Exodus II file. If no blocks are entered, then all blocks will be read and imported from the file. Standard ID parsing can also be used in this argument to select a range of blocks. For example "**1 to 5**" or "**1, 5 to 10 except 6**".
Each unique block selected to be imported will define a new body in the geometric model. Figure 1 shows a simple example of the geometry generated from the 3D finite element mesh.

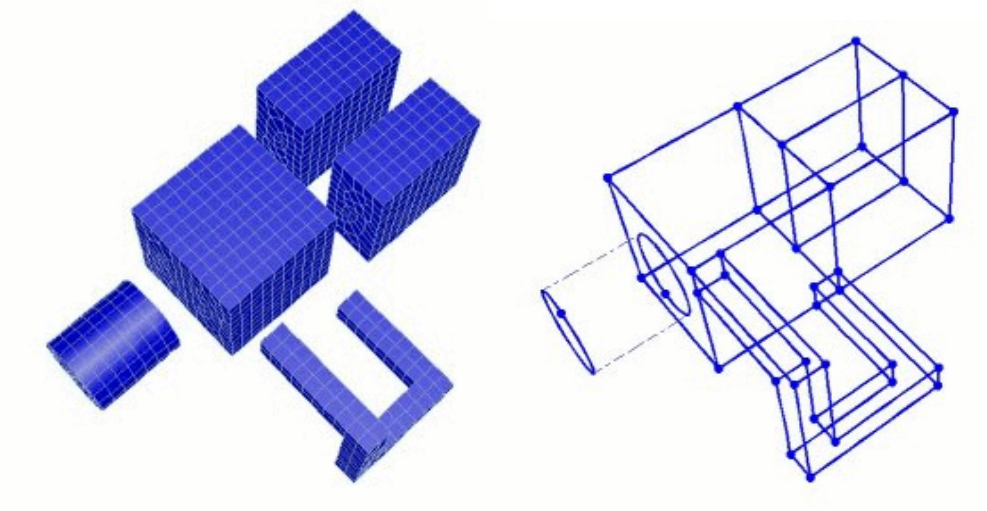


**Figure 1. Example of mesh based geometry (right) created from a finite element mesh (left)**

Blocks may be composed of 1D, 2D or 3D elements. For blocks composed of 2D elements (i.e. QUAD4, SHELL etc.), a sheet body will be created. One dimensional elements (i.e.. BEAM, TRUSS, etc.) will define curves. Where a block may be composed of more than one disconnected sets of elements, one body will be created for each continuous region of elements assigned to the same block. Where possible, the ID of the new body will be the same as the block ID. Since IDs must be unique, if a body ID is already in use, the next available ID will automatically assigned by the program.

#### Nodesets/Sidesets

Use the **nodeset** and **sideset** options to use any nodeset and sideset information in the Exodus II file in constructing geometry. Recall that nodesets and sidesets are generic boundary condition data assigned to nodes, edges or faces of the finite elements. It is useful to group mesh entities belonging to unique boundary conditions into geometric entities. This permits the user to remesh a particular region of the model without having to reassign boundary conditions.
If the **nodeset** and **sideset** arguments are given, geometric entities will be generated for each unique set of nodes, edges or element faces assigned to a nodeset or sideset. The default is to use any nodeset and sideset information available in the file. Figure 2 shows an example of how nodeset and sideset information might be used to generate geometry.

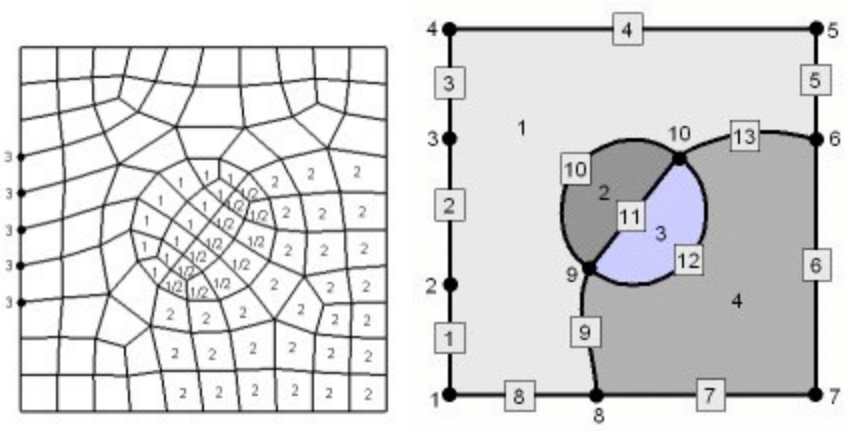


**Figure 2. Example of geometry created from mesh entities assigned to nodesets (3) and sidesets (1 and 2).**

Upon import, nodesets and sidesets are automatically created with the appropriate geometric entities assigned to them. The IDs of the new geometric entities, if generated from boundary condition data, will be the same as the nodeset and sideset IDs. Where doing so would conflict with existing geometric IDs, the program will automatically select the next available ID.

#### Feature Angle

Use this option to specify the angle at which surfaces will be split by a curve or where curves will be split by a vertex. 180 degrees will generate a surface for every element face, while 0 degrees will define a single, unbroken surface from the shell of the mesh. The default angle is 135 degrees.

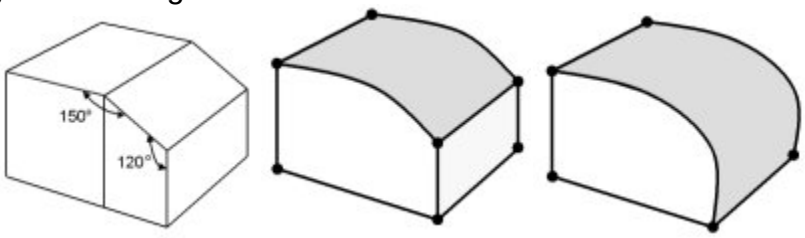



**Figure 3. Example use of Feature Angle**

Figure 3 shows an example of the use of different feature angles. On the left is a simple two-element hex mesh. Specifying a feature angle greater than 120 degrees would create the geometry in the center image. Using a feature angle less than 120 degrees and greater than 90 degrees would define the geometry on the right.
It is possible to independently control the feature angle for surfaces and curves. If Surface_Feature_Angle is specified, it controls the angle at which surfaces will be split by a curve, while Feature_Angle controls the angle at which curves will be split by a vertex. If Surface_Feature_Angle is not specified, Feature_Angle will control the angle for both surfaces and curves.

**Smooth Curves and Surfaces**

This argument allows the option of using a higher-order approximation of the surface when remeshing/refining the resulting geometry. Default is to use the original mesh faces themselves as the curve and surface geometry representation. If the finite element model to be imported is to represent geometry with curved surfaces, it may be useful to select this option. If selected, it will use a 4th order B-Spline approximation to the surface [Walton,96]. Figure 4 shows the effect of the smooth curve and surface option.

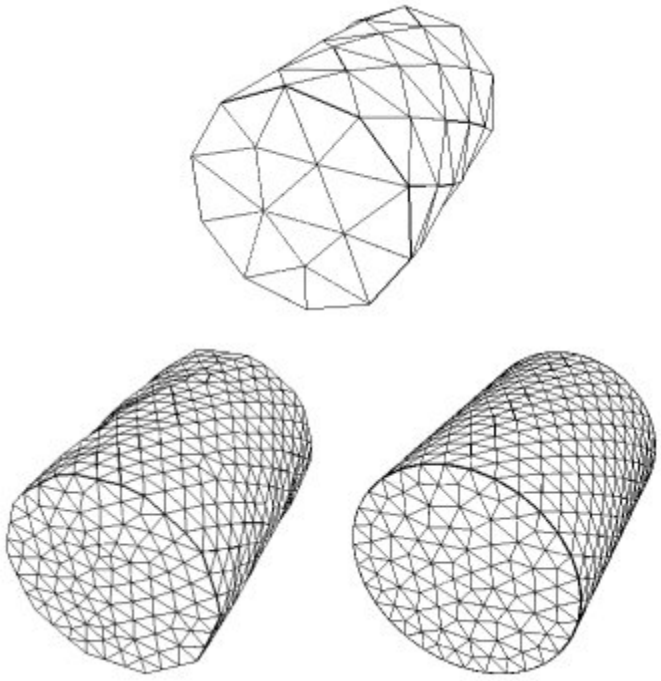


**Figure 4. Effect of Smooth Curve and Surface Option for remeshing of mesh-based geometry**

In this figure the top image is the original finite element mesh imported into Cubit. In this example both models have been remeshed with the same element size. The difference is that the figure on the right uses the smooth curve and surface option. While this option can improve the surface representation, it should be noted that memory requirements and meshing times can sometimes be affected.
If importing the Exodus II file using the command line, other options for surface representations are also available.

**[LINEAR|Gradient|Quadratic|Spline|Acis]**

The method used from the GUI is either **Linear** or **Spline**. The **Gradient** and **Quadratic** methods are still somewhat experimental and may not be as general purpose as the **Spline** representation. The **Acis** option will attempt to create ACIS geometry from the mesh. This option is an alpha feature and can only be used if developer commands have been turned on. For more detail see: Acis Geometry From Mesh

#### Apply Deformations

The **deformed** option permits the user to import time-dependant deformation information from the Exodus file. For this option, any vector data in the Exodus II file is assumed to be deformation information. If selected, deformations will be applied to the nodes upon import. Enter a specific time step value, integer step, or the last time available in the file. If time-dependant data is available in the Exodus II file, selecting the down arrow in the edit field will display the available time steps in the file. Default time is the last time step.

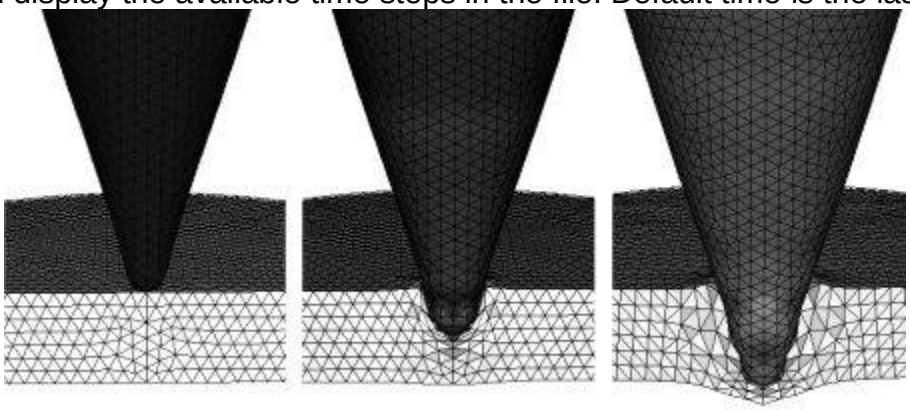


**Figure 5. Example of remeshing of a deformed finite element mesh**

Figure 5 shows an example of using Mesh-Based Geometry for a large deformation analysis. In this case, the analysis [Attaway et. al.,98] began and continued until mesh quality became unacceptable. At that point, the mesh was imported into Cubit and geometry re-created from the computed deformations. The finite element mesh could then be removed, remeshed or improved and written back to an Exodus II file. After remapping [Wellman,99] the appropriate analysis variables back to the mesh, the analysis could then be restarted. This process was repeated multiple times until the desired results were achieved.
**Note**: Care should be taken when using large deformations, as inverted elements (negative Jacobians) may produce unpredictable results with the resulting geometric representation.
Also available is an optional **scale** factor. This applies the indicated scale to all deformations. Default is 1.0.

#### Merge

This option allows the user to either merge or not merge the resulting volumes. The default option is to merge adjacent volumes. This results in non-manifold topology, where neighboring volumes share common surfaces. Using the no_merge option, adjacent volumes will generate distinct/separate surfaces.

#### Merge Nodes

The **merge_nodes** option will allow the user to specify a different tolerance for merging nodes on import. The default value is 1e-6.

**Note**: Care should be taken when setting import merge tolerances. Setting a tolerance too low will not merge adjacent nodes. Setting the tolerance too high can produce undesirable results, and severely tangle the mesh.

#### Midnode Correction

When importing a mesh in Cubit, the software performs quality checks on the elements. By default, if any elements fall below the specified quality threshold, Cubit notifies the user. This quality assessment applies to imported mesh-based geometry that uses high-order tetrahedral elements (tet10).
For TET10 elements, the **node constraint smart** command evaluates element quality using a designated quality metric, typically the default **normalized inradius**. If the quality of an element falls below a defined threshold (default: **0.15**) and straightening the edge of the TET10 would improve the quality, Cubit automatically performs midnode correction. This correction can also be enabled during the import process.
By default, the midnode correction option is set to **no_midnode_correction**, meaning that the mesh will be imported without modifying midnode locations. However, users have the flexibility to choose the **midnode_correction** option when importing mesh-based geometry. Enabling this option allows Cubit to straighten high-order edges if the mesh quality metric falls below the designated threshold and if it results in improved quality.

*Importing a Lite Mesh*

The command to import a lite mesh from an Exodus II format file is:

```
Import Mesh '<exodusII_filename>' lite
```

When an Exodus II mesh is imported into Cubit using the lite option, it contains no geometric or topological information.
The lite mesh import may be an option for users wanting to quickly view the mesh without gaining all the abilities to modify the mesh.
More information on how a lite mesh may be viewed or modified is described in the Lite Mesh section of the documentation.

### Importing I-DEAS Files

The command to import a mesh from an I-DEAS format file is:

```
Import Ideas [Mesh Geometry] '<input_filename>' [Feature Angle <angle>]
[Nobcs]
```

Including the keyword **Mesh Geometry** will instruct CUBIT to create mesh-based geometry. This will provide the user with the ability to remesh geometric entities. If the user does not import with the Mesh Geometry flag, he will have to tell CUBIT to draw the mesh after the import is done in order to view it.
The **Feature Angle** is used when building the surface topology to determine when to split a surface into two surfaces. If the angle between two neighboring element normals is less than Feature Angle, then the two elements will be placed on separate surfaces. If the

keyword Feature Angle is not supplied, the default 135 degrees is used. For a description of importing mesh geometry see Importing Exodus II Files.
The keyword **nobcs** can be included if boundary conditions are not to be imported.
It should be noted that CUBIT sometimes cannot successfully generate mesh-based geometry for complex models. If this occurs, import the mesh without the Mesh Geometry flag, and draw the mesh to view it.
To see more information on the I-DEAS file format, visit their website at www.siemens.com.

## Importing Nastran Files

The command to import a mesh from an Nastran format file is:

**Import Nastran [Mesh Geometry] '<input_filename>' [Feature Angle <angle>] [Nobcs]**

Including the keyword **Mesh Geometry** will instruct CUBIT to create mesh-based geometry. This will provide the user with the ability to remesh geometric entities. If the user does not import with the Mesh Geometry flag, he will have to tell CUBIT to draw the mesh after the import is done in order to view it.
The **Feature Angle** is used when building the surface topology to determine when to split a surface into two surfaces. If the angle between two neighboring element normals is less than Feature Angle, then the two elements will be placed on separate surfaces. If the keyword Feature Angle is not supplied, the default 135 degrees is used. For a description of importing mesh geometry see Importing Exodus II Files.
The keyword **nobcs** can be included if boundary conditions are not to be imported.
It should be noted that CUBIT sometimes cannot successfully generate mesh-based geometry for complex models. If this occurs, import the mesh without the Mesh Geometry flag, and draw the mesh to view it.
See http://en.wikipedia.org/wiki/Nastran for more information on the NASTRAN file format.

## Importing Patran Files

The command to import a mesh from an Patran format file is:

**Import Patran '<neutral_filename>'**

**Import Patran Mesh Geometry '<neutral_filename>' [Use [Feature_Angle <angle>] [Linear|Gradient|Quadratic|Spline] ]**

See Importing Exodus II Files for a description of the import options.
For more information on the Patran file format, see their website at www.mscsoftware.com.

# adaptivity and sizing functions

## Geometry Adaptive Sizing for TriMesh and TetMesh Schemes

The **TriMesh** and **TetMesh** schemes in Cubit are based upon third party libraries known as MeshGems that are developed and distributed by Distene. They are robust and fast triangle and tet meshing algorithms that have built in capabilities for adaptively controlling the mesh size based upon feature sizes. In most cases the sizing controls provided as part of the **scheme** command are sufficient to control mesh sizes. As such, the sizing functions described in this section cannot be used with the the MeshGems triangle and tet meshing algorithms. If a sizing function is assigned to a volume or surface, and the **TriMesh** or **TetMesh** scheme is selected, rather than using the MeshGems algorithm for meshing the surfaces, it will automatically revert to using the **TriAdvance** scheme. Any settings defined with the **TriMesh** or **TetMesh** scheme will be ignored and the sizing function will be used to determine local mesh sizes.

When using the **TriMesh** and **TetMesh** schemes, recommended practice is to mesh all surfaces and volumes simultaneously. This provides the greatest flexibility to the algorithms to determine feature sizes and their effect on neighboring surfaces and volumes. The default settings for **TriMesh** and **TetMesh** schemes will automatically provide geometry adaptive mesh sizing. These default settings can however be adjusted by using the settings on the **scheme** command. The scheme settings are described in the **TetMesh** and **TriMesh** sections of the documentation.

## Mesh Adaptivity and Sizing Functions

CUBIT provides several options for controlling the density of a mesh by adapting to various geometric, analysis, or user-defined properties. Interval sizes are defined automatically, explicitly, or through *sizing functions*. The sizing functions can be based on the physical features of the model, a previous analysis solution, or a user-specified bias. Adaptivity can apply to meshing either curves or surfaces.

*Adaptive Curve Meshing*

CUBIT provides several ways to adaptively mesh curves. Three curve meshing schemes are provided for this purpose. They include the following schemes:

- Curvature
- FeatureSize

The first two schemes use characteristics of the geometric model to define element sizes. The third scheme uses a field function typically defined from a previous analysis solution. FeatureSize is an alpha feature and should be used with caution.

*Adaptive Surface Meshing*

Adaptive surface meshing in CUBIT produces a function following mesh which sizes elements based on the value of the driving function at the spatial location at which the element is to be placed. Adaptive surface meshing is performed using the paving,

triadvance or tridelaunay algorithms in combination with an appropriate sizing function. The types of sizing functions that can be used are

- Bias Sizing
- Constant Sizing
- Curvature Sizing
- Linear Sizing
- Interval Sizing
- Inverse Sizing
- Super Sizing
- Test Sizing
- Exodus-based field function
- Geometry Adaptive (Skeleton Sizing)
- Geometry Adaptive for TriMesh and TetMesh Schemes

Super sizing and test sizing functions are alpha features and should be used with caution. The procedure for adaptively meshing a surface is to designate paving, triadvance or tridelaunay as the mesh scheme for that surface, assign sizing function types, and mesh the surface.
The command syntax of these commands is:

**Surface < id > Scheme {Pave|TriAdvance|TriDelaunay}**

then

**Import Sizing Function '<exodusII_filename>' Block <block_id> Variable '<variable_name>' Time <time> [Deformed]**

**Surface <id> Sizing Function [Type] Exodus [Min <min_value> Max <max_value>]**

or

**Surface <id> Sizing Function [Type] {Constant|Curvature|Interval|Inverse|Linear|Super|Test|None}] [Neighbor [<max_neighbors>]]**

**(See note below regarding 'Neighbor' parameter)**
or

**Surface <id> Sizing Function [Type] Bias Start Curve <id_range> {Finish Curve <id_range>| Factor <val>}**

then

**Mesh Surface <id>**

*Adaptive Volume Meshing*

Adaptive volume meshing in CUBIT produces a function following mesh that sizes elements based on the value of the driving function at the spatial location at which the element is to be placed. Adaptive volume meshing is performed using the tetmesh scheme in combination with an appropriate sizing function. The types of sizing functions that can be used are constant, test, geometry adaptive and geometry adaptive (skeleton sizing). Test sizing is an alpha feature and should be used with caution. Other sizing functions will be added in future versions of Cubit.
The procedure for adaptively meshing a volume is to designate **tetmesh** as the mesh scheme for that volume, assign sizing function types, and mesh the volume.
The command syntax of these commands is:

**Volume <id> scheme tetmesh**
**Volume <id> Sizing Function [Type] {Constant|Test|None}**
**Mesh Surface <id>**

The following sections describe details of the various volume sizing methods.

- Constant Sizing
- Test Sizing
- Geometry Adaptive (Skeleton Sizing)
- Geometry Adaptive for TriMesh and TetMesh Schemes

**Note regarding 'Neighbor' parameter:**
The **maximum neighbors** is the number of points used by the sizing function to compute the size at the requested point. If the number of neighbors is zero, all of the points on the boundary are used in the size calculation. If the number of neighbors is some other number, only that number of closest points are used in the calculation.

## Bias Sizing Function

**Syntax**:

**Surface <id> Sizing Function Type Bias Start Curve <id_range>**
**{Finish Curve <id_range>| Factor <val>}**

**Synopsis:**
The **Bias** sizing function for surfaces is similar to biasing curves. Indeed, setting a bias sizing function for a surface will bias the boundary curves, as well as control paving to follow the bias inside the surface. You first specify the size of a couple of bounding curves (the start curves), then specify the bias sizing function for the surface.
**Discussion:**
Recall that for biasing curves, you specify the start and end vertex. For the bias sizing function, you specify the start curves, from which to bias away. The sizes of these curves should already be set before setting the surface sizing function since their average size is taken to be the starting size (almost). If the start curve sizes change, then you should set the surface sizing function again.

You can either supply a geometric factor, or the set of finish curves whose sizes you want to match at that distance. A geometric factor. It automatically sizes and biases or dualbiases the non-start curves, including any finish curves. These curves need not be perpendicular to the starting curves. The interval count and scheme are soft-set, so they won't be changed if they are already hard-set. If the size of the start curves or finish curves are changed, then the sizing function command should be re-issued.
The sizing function value at a point is defined in terms of the straight-line distance from the point to the closest starting curve. So, it works best if all the starting curves have the same size, and the surface is relatively flat. But, starting curves need not be parallel to one another. Similarly, the non-start curves need not have any particular orientation wrt the start curves.
The bias sizing function was designed to easily set the sizes of a sequence of adjoining surfaces: assign a size to the curve you want to bias away from, then set the bias sizing function of the first surface, with its finish curves being the start curve of the second surface, etc. See the last example below.

**Examples:**

Here are some example journal files and resulting pictures:

```
# bias_sz_fn_demo.jou
brick x 100 y 10 z 10
color vol 1 red
surface 1 scheme pave
surface all except 1 visibility off
# label curve interval
# graph text 2
display
# mesh 1
curve 4 size 2
surface 1 sizing function type bias start curve 4 factor 1.3
mesh surface 1
# see figure 1
```

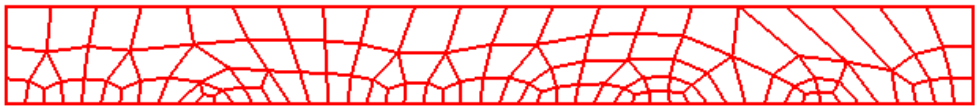


**Figure 1. Surface with bias sizing function factor > 1.**

```
# mesh 2
delete mesh
surface 1 sizing function type bias start curve 4 factor {1/1.1}
mesh surface 1
# see figure 2
```

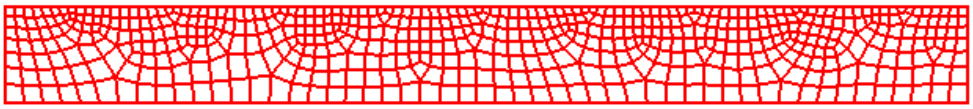


**Figure 2. Surface with bias sizing function factor < 1**

```
# mesh 3
reset
cyl rad 6 z 1
cyl rad 4 z 1
sub 2 from 1
section body 1 yplane
section body 1 xplane
surf all except 19 vis off
color vol 1 red
display
# finish curve mesh
surf 19 scheme qtri base scheme pave
surface 19 size 0.7
curve 26 size 0.07
surface 19 sizing function type bias start curve 26 finish curve 25
mesh surface 19
pause
# see figure 3
```

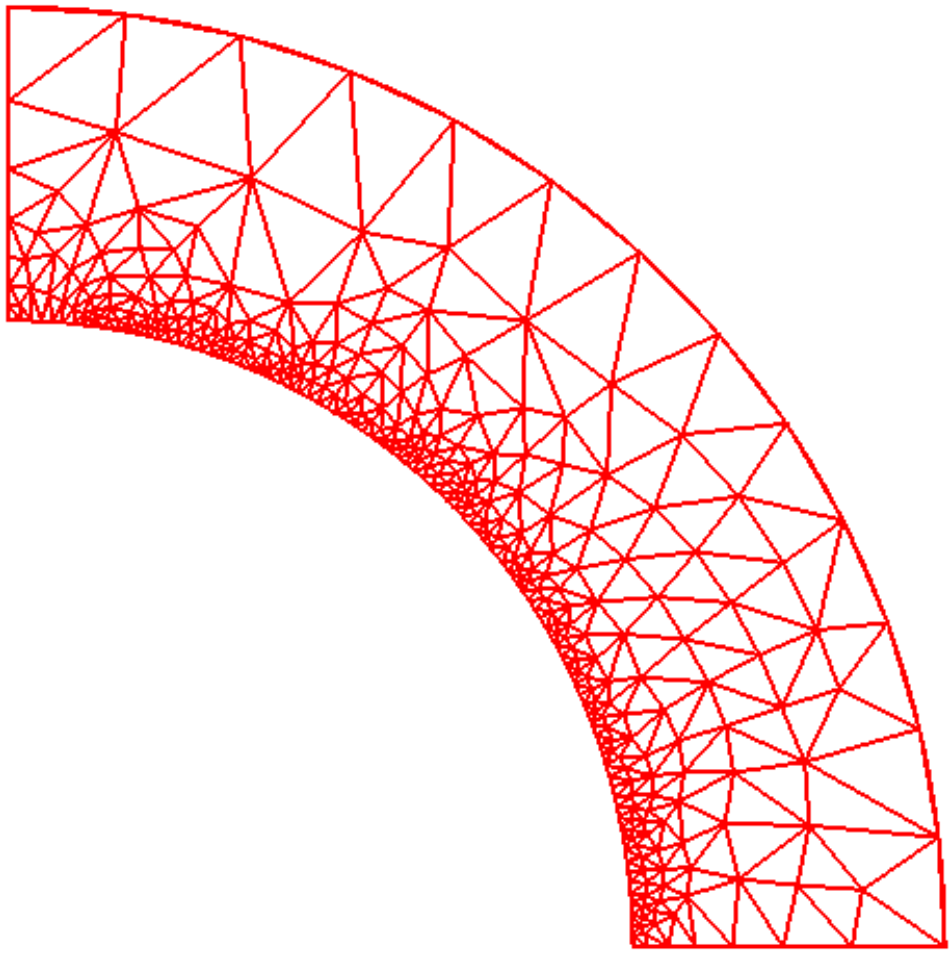


**Figure 3. Surface with bias sizing function start and finish curve. Scheme qtri, base scheme pave.**

```
# dual bias mesh
delete mesh
curve 25 26 size 0.02
curve 25 26 scheme equal
surface 19 sizing function type bias start curve 26 25 factor 1.3
mesh surface 19
zoom curve 12
pause
# see figure 4
```

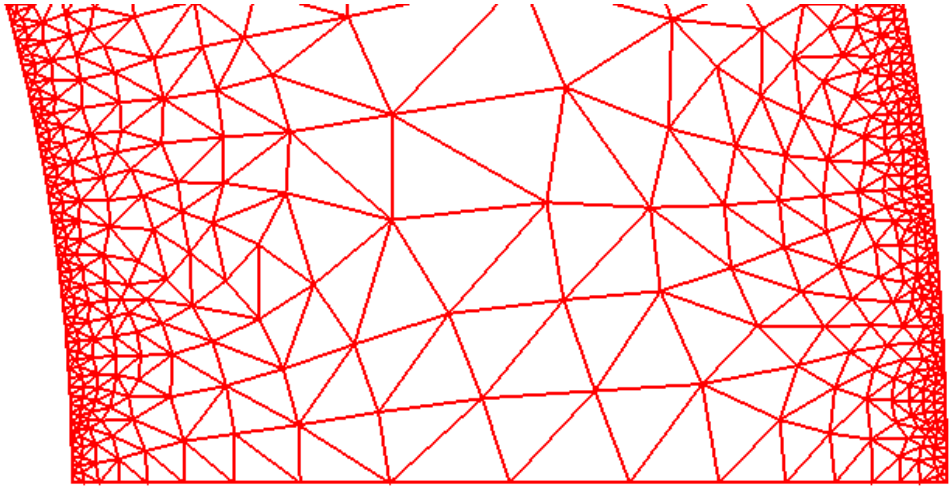


**Figure 4. Close up of surface with dual bias sizing function start and finish curve. Scheme qtri, base scheme pave.**

```
# funny face
reset
prism sides 5 z 1 radius 1
cylinder radius 0.1 z 1
body 2 move -0.4 0 0
subtract 2 from 1
cylinder radius 0.1 z 1
body 3 move 0.2 0 0
subtract 3 from 1
prism sides 6 radius 0.2 z 1
body 4 move 0 -0.4 0
subtract 4 from 1
surface all except 34 visibility off
color vol 1 red
display
surface 34 scheme pave
curve 23 19 size 0.01
surface 34 sizing function type bias start curve 19 23 factor 1.3
mesh surface 34
# see figure 5
```

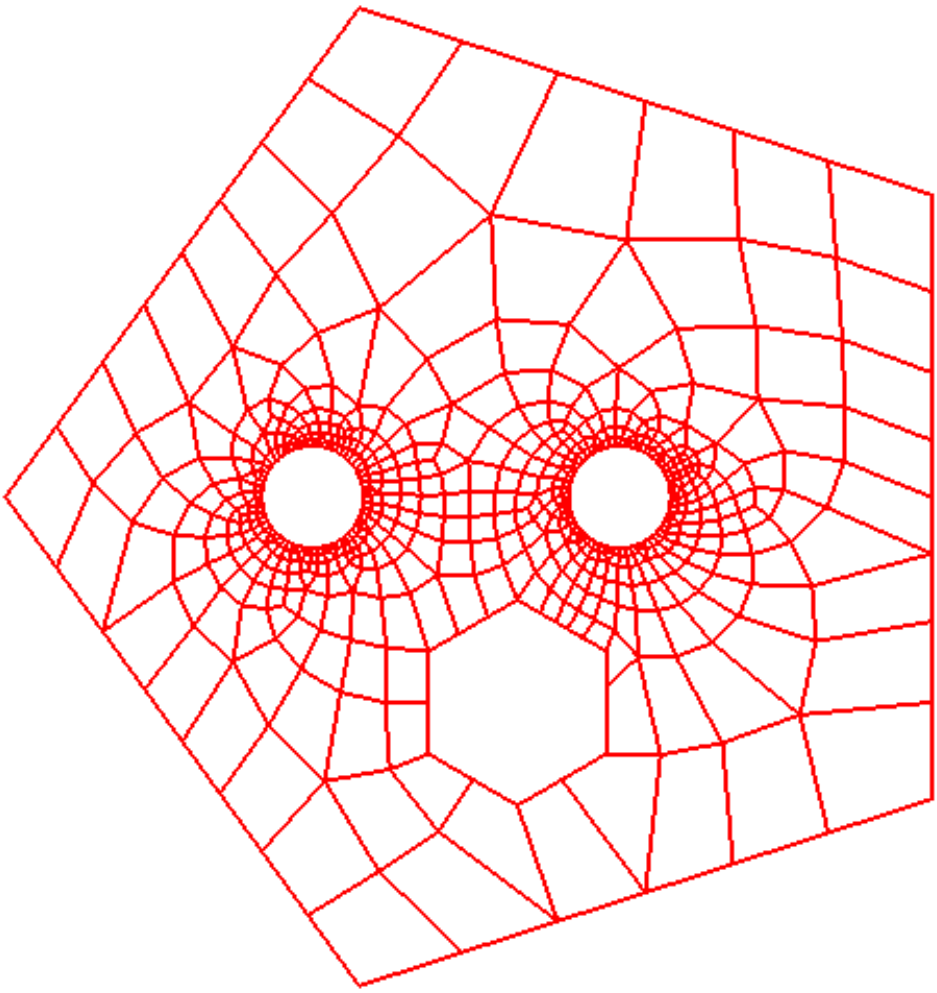


**Figure 5. Bias away from two round holes.**

```
# bias surface chain
reset
cylinder radius 1 z 1
cylinder radius 0.2 z 1
cylinder radius 0.4 z 1
cylinder radius 0.8 z 1
imprint body all
delete body 2 3 4
section body 1 xplane
section body 1 yplane
surface all except 42 43 44 45 vis off
color volume 1 red
surface all scheme pave
curve 55 interval 36
surface 43 sizing function type bias start curve 55 factor 1.3
surface 44 sizing function type bias start curve 57 factor 1.3
```

```
# curve 57 had its size determined by a prior bias sizing function
surface 45 sizing function type bias start curve 58 factor 1.3
surface 42 sizing function type bias start curve 55 factor 1.3
mesh surface 42 43 44 45
display
highlight curve in surface 42 43 44 45
# see figure 6
```

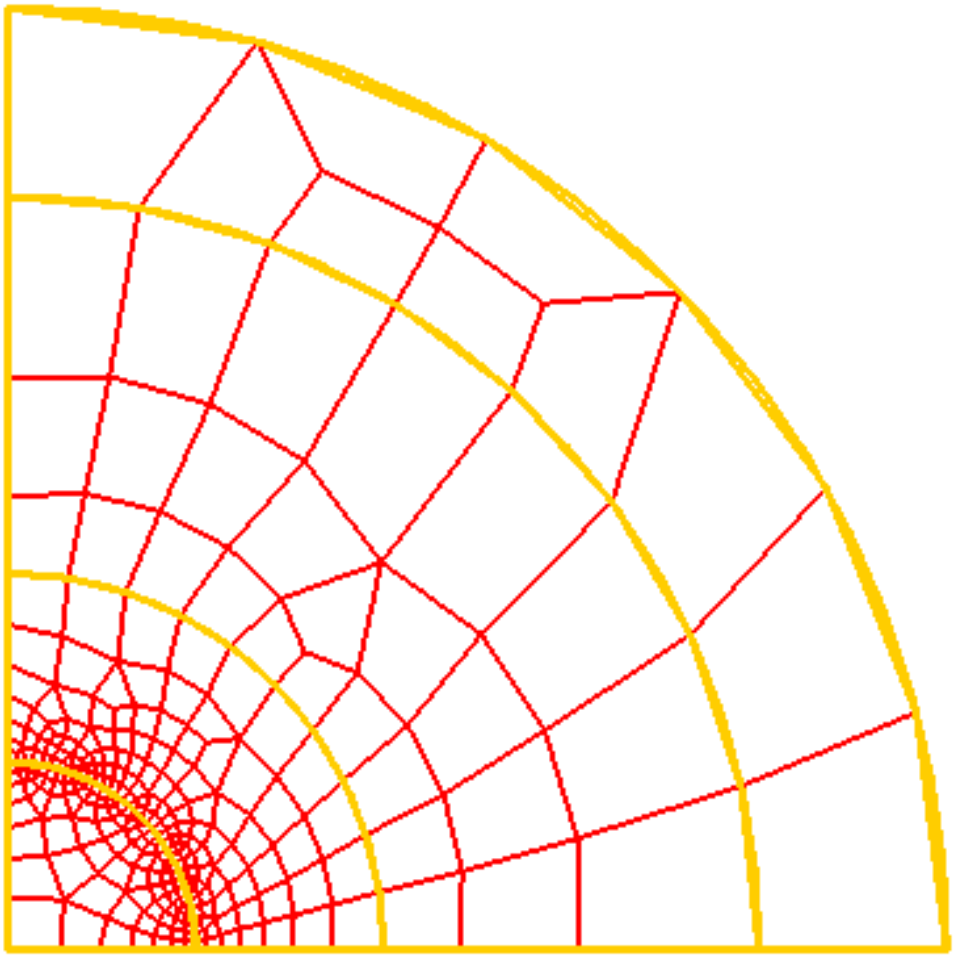


**Figure 6. A chain of biased surfaces. Only one curve's intervals were explicitly set.**

## Constant Sizing Function

**Syntax:**

```
Surface <id> Sizing Function [Type] Constant

Volume <id> Sizing Function [Type] Constant
```

**Synopsis:**

The **Constant** sizing function specifies that a constant element size be used over the interior of the surface or volume. The value used as the constant size is the interval size that has been set for the entity. For example, the following commands will cause the mesh size to be smaller on the interior than on the surface's bounding curves.

```
reset
brick x 10
surface 1 scheme pave
curve in surface 1 interval 5
surface 1 size 0.5
surface 1 sizing function constant
mesh surface 1
```

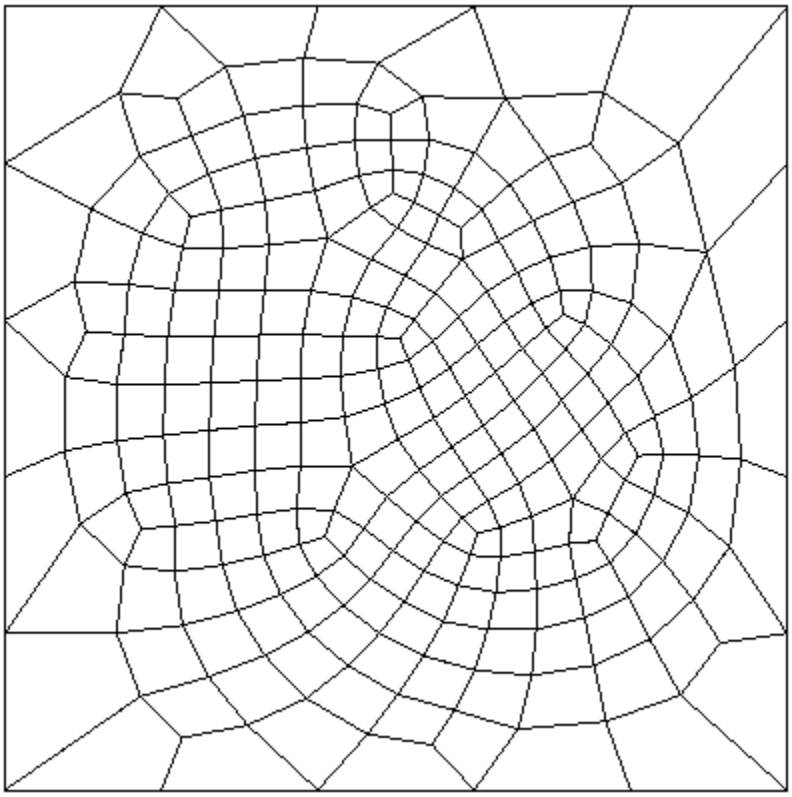


**Figure 1. Constant Sizing Function**

## Curvature Sizing Function

The **Curvature** sizing function determines element size based on the curvature evaluation of a surface at the current location. Two surface curvature values (taken perpendicular to each other) are compared at the location of interest, and the largest is used as the sizing function for the mesh. Figure 1 shows a solid with a highly deformed surface which displays rapid change of surface curvature at several locations.

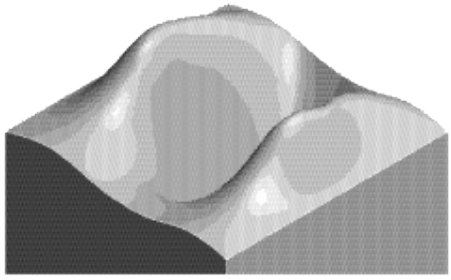


**Figure 1. NURB solid with high surface curvature change**

Figure 2 depicts a normal paved mesh of this surface using a common size on all bounding curves and no sizing function in the interior. The total number of quadrilateral shell elements for this case is 1988. Figure 3 shows a mesh which was generated with the curvature sizing function option. The mesh is graded denser in the regions of quickly changing curvature, such as at the tops of the hills and at the bottom of the valley. Due to the intense interrogation of the underlying geometric modeler which the curvature method relies on, this option can be very computationally expensive.

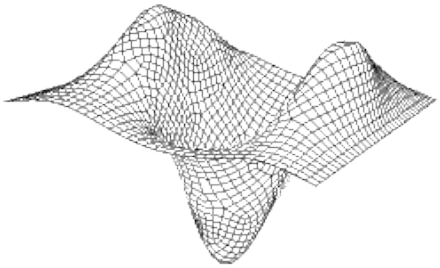


**Figure 2. NURB mesh with no interior sizing function**

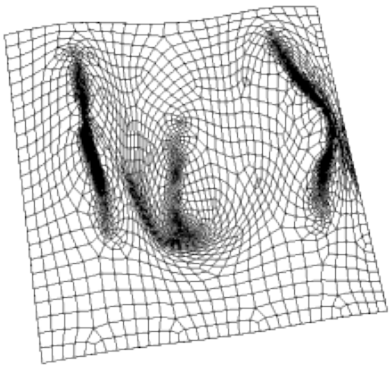


**Figure 3. NURB mesh with curvature sizing function**

## Exodus II-based Field Function

The ability to specify the size of elements based on a general field function is also available in CUBIT. With this capability, the desired element size can be determined using

a field variable read from a time-dependent variable in an Exodus II file. Both quadrilateral and triangle elements are supported for surfaces, and both tetrahedral and hexahedral elements are supported for volumes.
A field function is a time-dependent variable in an Exodus II file. Either node-based or element-based variables may be used. Currently, field functions are imported from element and node-based Exodus II data. The mesh block containing the corresponding elements must be imported along with the field function data.
Exodus variable-based adaptive meshing is accomplished in CUBIT in several steps:

1. Surface mesh scheme set to Pave or TriMesh, and/or volume mesh scheme set to Tetmesh.
2. An Exodus mesh and variable for that mesh is read into CUBIT.
3. The surface or volume sizing function type is designated and the Exodus variable is mapped to give normalized and localized size measures.
4. The geometry is meshed

Importing a field function, and normalizing that function are done in two separate steps to allow renormalization. The following command is used to read in a mesh for the field function:

**Import Sizing Function '<exodusII_filename>' Block <block_id> Variable `<variable_name>' [Time <time_val> | Step <step> | Last] [Deformed]**

The **block_id** is the element block to be read, which can be a single block id or the word **all**. The **variable_name** is an Exodus time-dependent variable name (either element-based or nodal-based) which values are used to drive the mesh size. The timestep for the time-dependent variable can be specified as a **time** with a value, as a **step** with an index or **Last** to use the last timestep in the file. The **Deformed** keyword indicates whether to read the deformed field function mesh, which should align with the geometry being meshed and needs to be accounted for in the field function data. (For information on creating deformed geometry from EXODUSII data, see Importing 2D EXODUSII Files and Importing EXODUSII Files) .
Note that when a field function is read in, the mesh is stored in the background, and therefore the geometry is not considered meshed. Also note that if deformation is not being modeled, the geometry should be in the same state as it was when that mesh was written (see Importing a Mesh for more details on importing meshes).
Once the field function has been read in, it can be normalized before being used to generate a mesh. The normalization parameters (**Min_size** and **Max_size**) are specified in the same command that is used to specify the sizing function type for the surface or volume. The syntax of these commands are:

**Surface <id> Sizing Function Type Exodus [Min_size <min_val> Max_size <max_val> Log_Map Inverse_Map Scale_Mesh_Multiplier <value>]**

**Volume <id> Sizing Function Type Exodus [Min_size <min_val> Max_size <max_val> Log_Map Inverse_Map Scale_Mesh_Multiplier <value>]**

If normalization parameters are specified, the field function is normalized so that its range falls between the minimum and maximum values input. If an element-based variable is used for the sizing function each node is assigned a value that is the average of variables on all connected elements. Nodal variables are used directly.
The **Log_Map** option maps the range so that it is logarithmic, base 10.
**Inverse_Map** flips the mapping so that smaller and larger values in the mapping generate larger and smaller elements respectively. See figure 1 below.

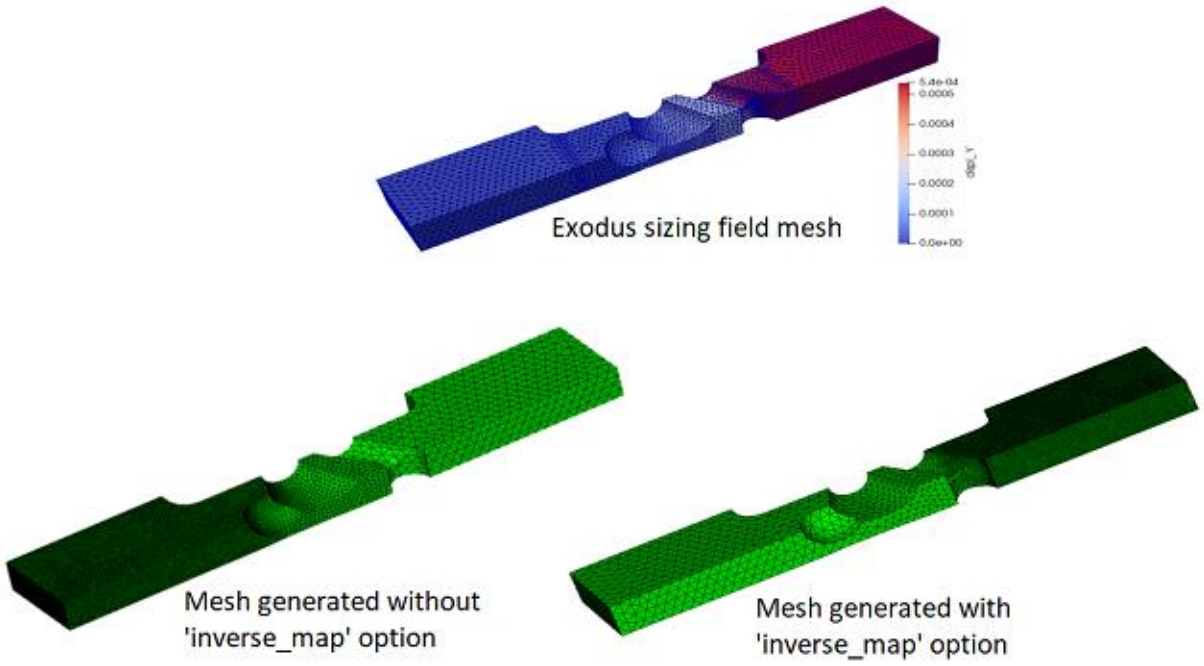



**Figure 1. Effect of 'inverse_map' option**

**Scale_Mesh_Multiplier** scales the size that the field data ultimately yields.
After the sizing function normalization, the geometry may be meshed using the normal meshing command.
For example, the left image in Figure 2 depicts a plastic strain metric which was generated by PRONTO-3D [Taylor, 89] a transient solid dynamics solver, and recorded into an ExodusII data file. When the file is read back into CUBIT, the paving algorithm is driven by the function values at the original node locations, resulting in an adaptively generated mesh [Attaway, 93]. The right image in Figure 1 depicts the resulting mesh from this plastic strain objective function.

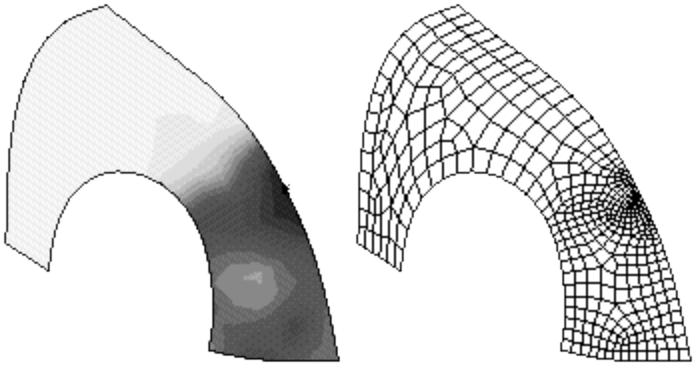


**Figure 2. Plastic strain metric and the adaptively generated mesh**

*Surface/Curve Meshing with Exodus II - based Field Functions*

While adaptively meshing a surface using a field function, the curves will be meshed using the Exodus II information. To override this, curves may have their meshing scheme set to equal or some other desired scheme. While adaptively meshing a volume using a field function, the surfaces and curves will be meshed using the Exodus II information. To override this for a surface, one can set the sizing function to "none" for that surface.

## Geometry Adaptive Sizing Function (Skeleton Sizing)

The **Geometry Adaptive Sizing Function**, also referred to as the **Skeleton Sizing Function** (Quadros 2005; Quadros 2004; Quadros 2004(2)), automatically generates a mesh sizing function based upon geometric properties of the model. This sizing scheme attempts to create a sizing function that allows unstructured meshing schemes to generate a mesh with the following properties:

- The sizes of the mesh elements vary smoothly throughout the mesh
- The mesh elements resolve the geometry to a sufficient degree
- The mesh elements do not over-resolve the geometry.

The geometry adaptive sizing function can be used to create sizing information for surfaces, solids, and assemblies.
This sizing function uses geometric properties to influence mesh size. The scheme calculates or estimates:

- 3D-proximity (thickness though the volume)
- 2D-proximity (thickness across a surface)
- 1D-proximity (curve length)
- Surface curvature
- Curve curvature.

These properties are then used to calculate a sizing function throughout the geometric entity (or entities). Regions of relatively high complexity will have a fine mesh size, while regions of relatively low complexity will have a coarse mesh size. For example, generally,

a high-curvature region on a surface will have a finer mesh size than a low-curvature region on that surface

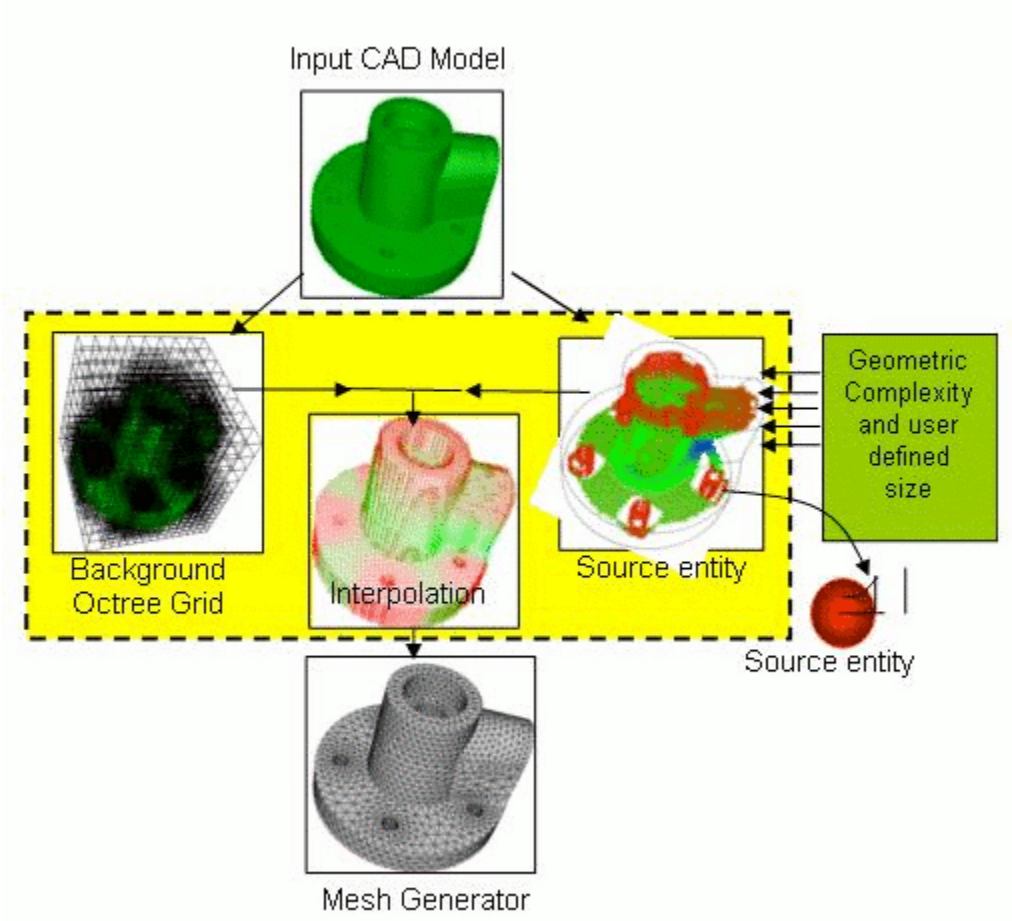



**Figure 1: Overview of Computational Framework**

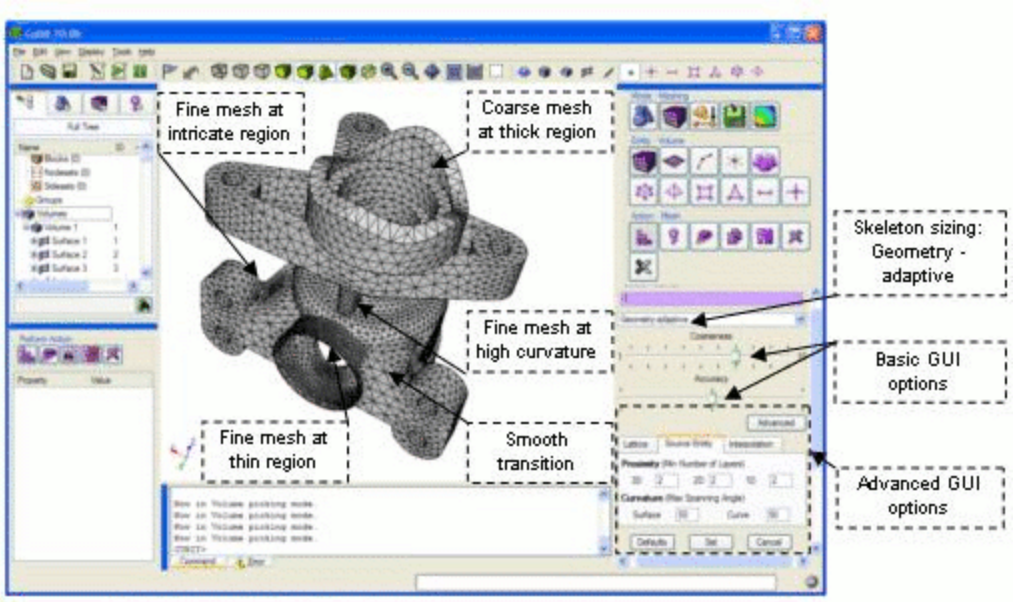



**Figure 2: Skeleton Sizing Function example in the GUI**

*Skeleton Sizing Behaviors*

Skeleton sizing can be specified on single or multiple surface(s)/volume(s) at a time from the GUI (Meshing Control Panel) or the command-line. The following describes how specifying sizing on entities can change skeleton sizing's behavior:
*Single surfaces/volumes* – If skeleton sizing is applied to surfaces/volumes one at a time, each entity's sizing is not influenced by the others. On the command-line, issue a separate command for each entity. In the GUI, specify only one surface or volume before selecting "Apply Size".
*Multiple surfaces* – If skeleton sizing is applied on multiple surfaces together, then geometric features of a particular surface may affect its neighboring surfaces.
*Multiple volumes (assembly sizing)* – Skeleton sizing can be applied to assembly models so that geometric features of a volume may influence its neighbors. Volumes should first be imprinted and merged before they are specified together for skeleton sizing.

*Command Line Syntax*

Skeleton sizing on surfaces:

**Surface <surface_id_range> Sizing Function Skeleton**
**{[scale <1 to 10 = 7>] [time_accuracy_level <1 to 3 = 2>]**
**[min_depth <3 to 8 = 5>] [max_depth <4 to 9 = 7>] [facet_extract_ang <1 to 30 = 10>]**
**[min_num_layers_2d < 1 to N = 1>] [min_num_layers_1d < 1 to N = 1>]**
**[max_span_ang_surf <5.0 to 75.0 = 45.0 degrees>]**
**[max_span_ang_curve <5.0 to 75.0 = 45.0 degrees>]**
**[min_size <float>] [max_size <float>] [max_gradient <float=1.5>]}**

Skeleton sizing on volumes:

**Volume <range> Sizing Function Skeleton**
**{[scale <1 to 10 = 7>] [time_accuracy_level <1 to 3 = 2>]**
**[min_depth <3 to 8 = 5>] [max_depth <4 to 9 = 7>] [facet_extract_ang <1 to 30 = 10>]**
**[min_num_layers_3d < 1 to N = 1>] [min_num_layers_2d < 1 to N = 1>]**
**[min_num_layers_1d < 1 to N = 1>]**
**[max_span_ang_surf <5.0 to 75.0 = 45.0 degrees>]**
**[max_span_ang_curve <5.0 to 75.0 = 45.0 degrees>]**
**[min_size <float>] [max_size <float>] [max_gradient <float=1.5>]}**

The options are explained below:

*Basic Arguments*

- **max_size** (default=auto): The value for max_size is calculated automatically by default. Users can specify any positive real number based on the dimensions of the model to control the max size of the elements. If the skeleton sizing function creates large elements, than this argument can be used to control the maximum element size.
- **min_size**(default=auto): The value for min_size is calculated automatically by default. Users can specify any positive real number based on dimension of the model to specify the minimum size of the elements.
- **max_gradient** (1.0 to 3.0, default 1.5): The transition in element size is controlled using this parameter. Larger values of max_gradient result in fewer elements, but also lead to more abrupt transitions in size and possibly poorer quality elements.

**Scaling and Accuracy Arguments:**

- **scale** (1 to 10, default 7): The overall size of the elements is controlled by this argument. A coarser mesh can be generated by increasing the value of scale up to 10.0. To get a finer mesh, decrease the value of the scale (minimum value = 1).
- **time_accuracy_level** (1 to 3, default 2): This controls the computational time and accuracy level by adjusting various internal parameters of the skeleton sizing function. Users should try levels in increasing order. Level 1 takes the shortest time to compute the skeleton sizing function and Level 3 takes the longest time to compute the skeleton sizing function. However, Level 1 is less accurate than Level 2 and Level 3.

*Advanced Arguments*

**Lattice Arguments:**

The skeleton sizing function is generated and stored on a background octree grid whose cells are subdivided based on the graphics facets of the model. The level of subdivision

of the background grid affects how well the sizing function captures the geometric complexity of features. Reasonable defaults have been selected for the following two refinement (subdivision) parameters, but these may be overridden for use with simple (decrease parameters) or more complex (increase parameters) models.

- **min_depth** (default auto): min_depth controls the maximum cell dimension of the background octree grid. The higher the value of min_depth, the smaller the dimension of the maximum-sized cell. Computational time increases with increasing min_depth. By default the min_depth is calculated based on the geometric complexity of the input model and mesh size specified on sub-entities.
- **max_depth** (default auto): max_depth controls the minimum cell dimension. If the object contains very fine features then increasing the value of max_depth is suggested. The maximum depth has been limited to 9. By default the max_depth is calculated based on the geometric complexity of the input model and mesh size specified on sub-entities.
- **facet_extract_ang** (default 10 degree): facet_extract_ang is used to control the faceted representation of NURBS model. This option gives control of the accuracy of a faceted approximation of the model used to compute the adaptive sizing. For models with high curvature regions, decreasing the tolerance will give a better approximation of the geometry and avoid the creation of random dense meshes. Note that increasing this angle too much can generate invalid facets over curved regions, while decreasing the angle too much can cause signficant slowdowns in sizing calculations.

#### Source Entity Arguments

- **min_num_layers_3d** (Any value greater than 1, default 1): This parameter ensures that a minimum specified number of layers exist across the thickness of the volume. This parameter could be useful in generating meshes for mold flow simulation.
- **min_num_layers_2d** (Any value greater than 1, default 1): This parameter ensures that a minimum specified number of layers exist across the thickness of a surface.
- **min_num_layers_1d** (Any positive integer value, default 1): This ensures that any curve contains a minimum specified number of intervals.
- **max_span_ang_curve** (Range 5.0 to 75.0, default 45.0): Maximum spanning angle is a parameter that controls the mesh size at curved regions of curves. It is defined as the angle subtended by the normals at the end nodes of the mesh edge in the curved region of a curve. When a finer mesh is needed at curved regions of curves, then max_span_ang_curve should be decreased.
- **max_span_ang_surf** (Range 5.0 to 75.0, default 45.0 deg): Maximum spanning angle is a parameter that controls the mesh size at curved regions of surfaces. It is the angle subtended by the normals at the end nodes of the mesh edge in a curved region of a surface. When a finer mesh is needed at curved regions of surfaces, then max_span_ang_surf should be decreased.

**Note:** These arguments override the basic arguments. For example, time accuracy level 1 internally sets min_depth = 4 and max_depth = 6, and when min_depth is set to 4 and max_depth is set to 7 in the advanced options (recommended for models with fine features), then advanced options override the basic options. In the command-line, to override the depths set by a time_accuracy_level, specify min_depth and max_depth after it.

*Adding User Specified Sizing Sources*

Skeleton sizing function gives an option to manually add sizing sources on geometric entiies such as vertices, curves, and surfaces. These sizing sources control the size and scope (region of influence via num_layers) at geometric entities. The below command gives the syntax for adding sizing sources. Please note that the below command for adding sizing sources should be called after issuing the above given skeleton sizing command. First, the skeleton sizing command automatically generates sizing sources based on the geometric factors such as proximity, surface curvature, curve length, etc. Issuing the below command creates sizing sources in addition to the automatically generated sizing sources. Finally, when the meshing command is called, the mesh sizing function is calculated using all the sizing sources.

**Volume <vol_id_range> Sizing Function Skeleton add size_source {vertex|curve|surface} <id_range> size <double> num_layers <int>**

***Skeleton with Other Sizing Controls***

**Skeleton sizing function produces a smooth sizing function when called with other sizing controls available in Cubit. Skeleton sizing function behaves as SOFT firmness level. Skeleton sizing function always respects interval count specified on the curves. Skeleton sizing function respects interval size on curves and surfaces only if it is specified after calling the skeleton sizing function.**

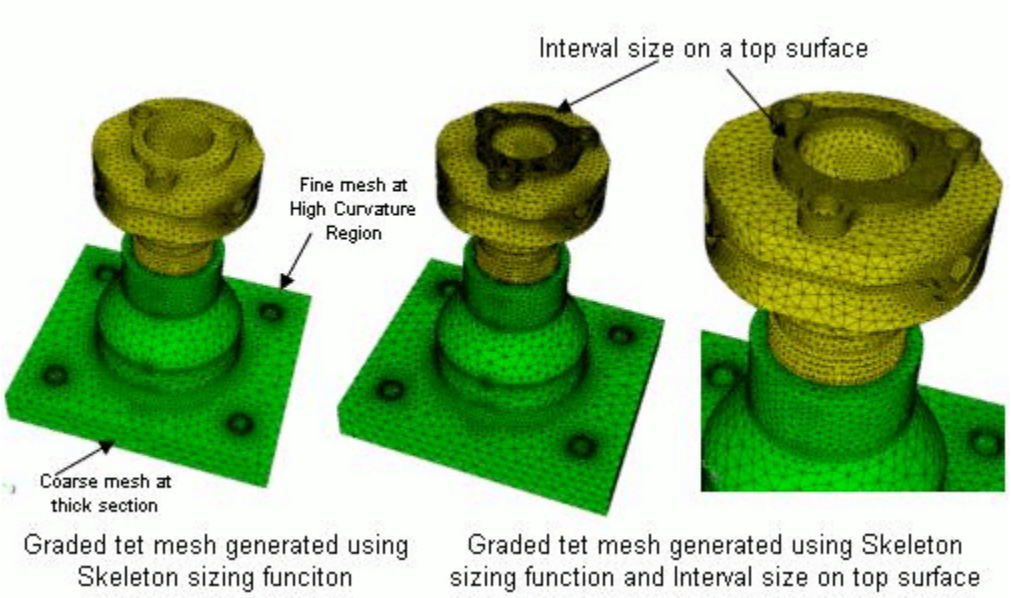


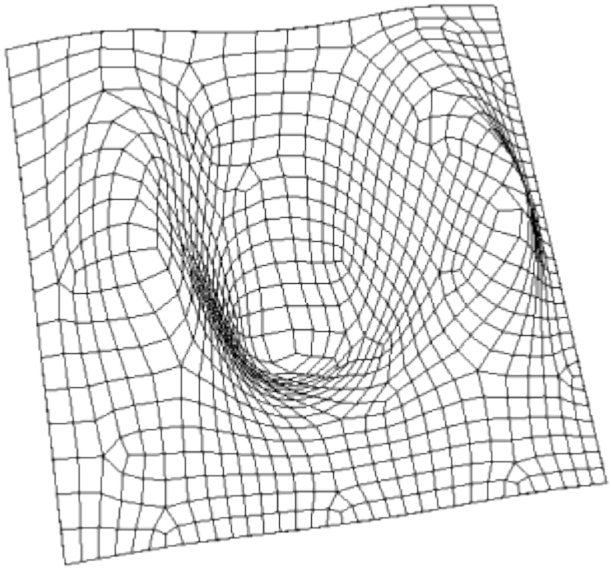
**Figure 3: Skeleton sizing function with other sizing controls**

***Limitations***

- **Currently, the skeleton sizing function is primarily intended for use with ACIS models. Skeleton sizing may be used on facet-based models (STL, facet, and MBG format) models, but results are not guaranteed. Sizing function generation with other geometry engines in Cubit is not guaranteed or supported in Cubit 10.1.**
- **The skeleton sizing function has mainly been tested with trimesh and tetmesh schemes. In general, structured or semi-structured meshing schemes do not have enough flexibility to utilize the skeleton sizing function. It is recommended that the skeleton sizing be used only with unstructured meshing schemes. However, if using skeleton sizing in conjunction with the pave scheme for surfaces, decreasing the max_gradient and scale arguments is suggested.**
- **For sizing function generation of assemblies in Cubit 10.1, at least time_accuracy_level 2 is generally recommended. This helps ensure that the geometric complexity of small features is captured. For example, "volume all sizing function skeleton time_accuracy_level 2"**

## Interval Sizing Function

The **Interval** sizing function is similar to the Linear function, but bases edge length at a location on the squared lengths of edges bounding the surface weighted by their inverse distance from the current location. An example is shown below.



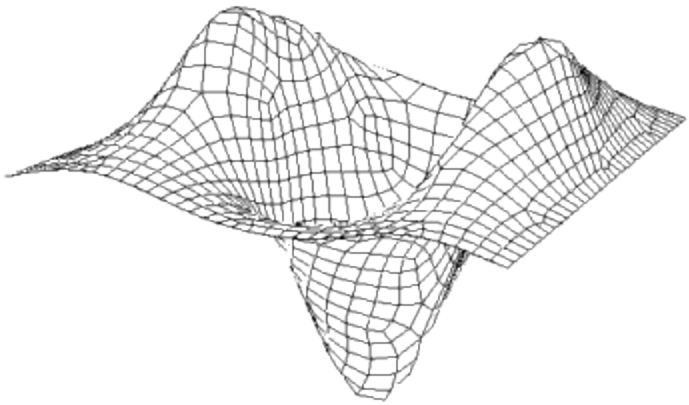
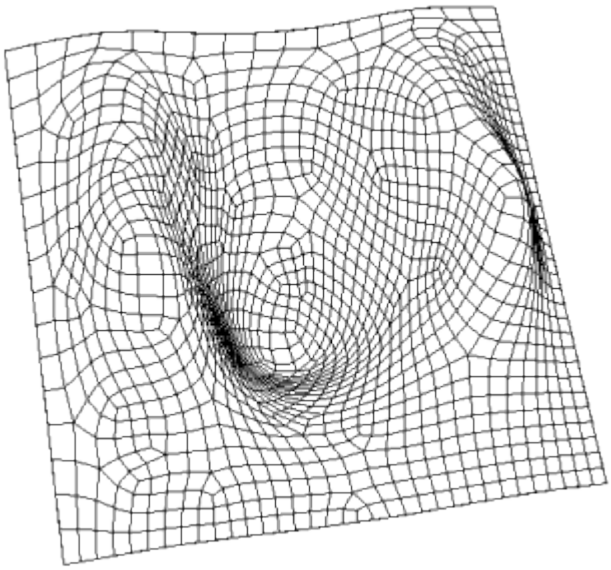


**Figure 1. NURB mesh with interval sizing function, 34 by 16 density**

## Inverse Sizing Function

The **Inverse** sizing function is also similar to the Linear function, but this method bases edge length at a location on the inverse lengths of edges bounding the surface weighted by their inverse distance from the current location (see Figure 1). The difference between the three linear sizing functions (Linear, Interval, Inverse) is sometimes subtle, but is driven by the geometry being meshed since the influence of these functions is strongly controlled by the number, positioning, and mesh density of the bounding curves relative to the interior surface area.



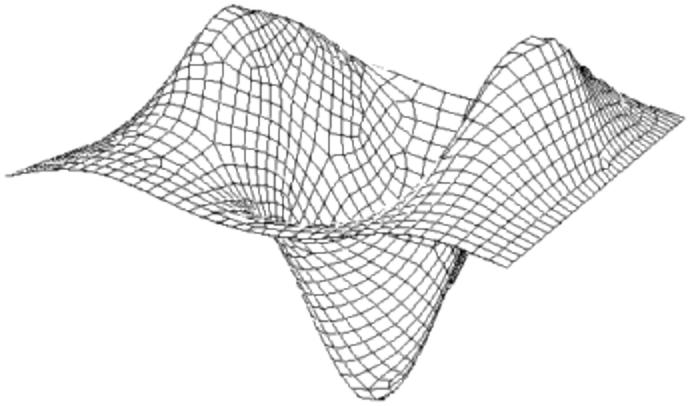


**Figure 1. NURB mesh with inverse sizing function, 34 by 16 density**

## Linear Sizing Function

The **Linear** class of sizing functions determines element size based on a weighted average of edge lengths for mesh edges bounding the surface being meshed. There are several variants of this class of sizing function. The Linear function bases edge length at a location on the lengths of edges bounding the surface weighted by their inverse distance from the current location. The result of this weighting is a more gradual change in mesh density during a transition between dense and coarse mesh. Figure 1 shows the same NURB surface mesh but with intervals of 34 on two curves and intervals of 16 on the remaining two bounding curves and no sizing function. It can be observed that the mesh progresses more rapidly inward from the coarser meshed curves, which locates the transition region much closer to the finer meshed curves. To combat this, the Linear function weights the sizing of new elements such that these transitions occur slower. Figure 2 displays two views of the same NURB geometry with the same bounding curve mesh density using the linear sizing function.

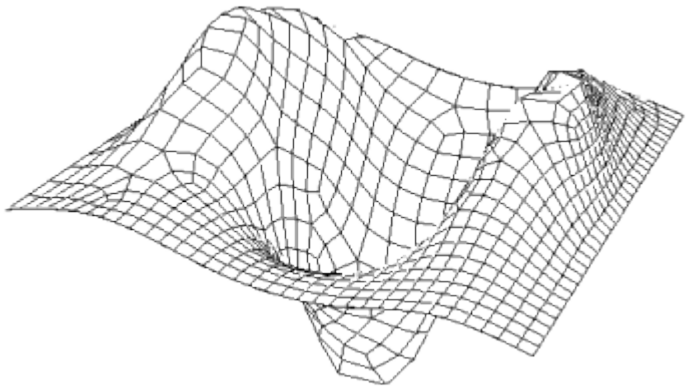


**Figure 1. NURB mesh with no sizing function, 34 by 16 density**

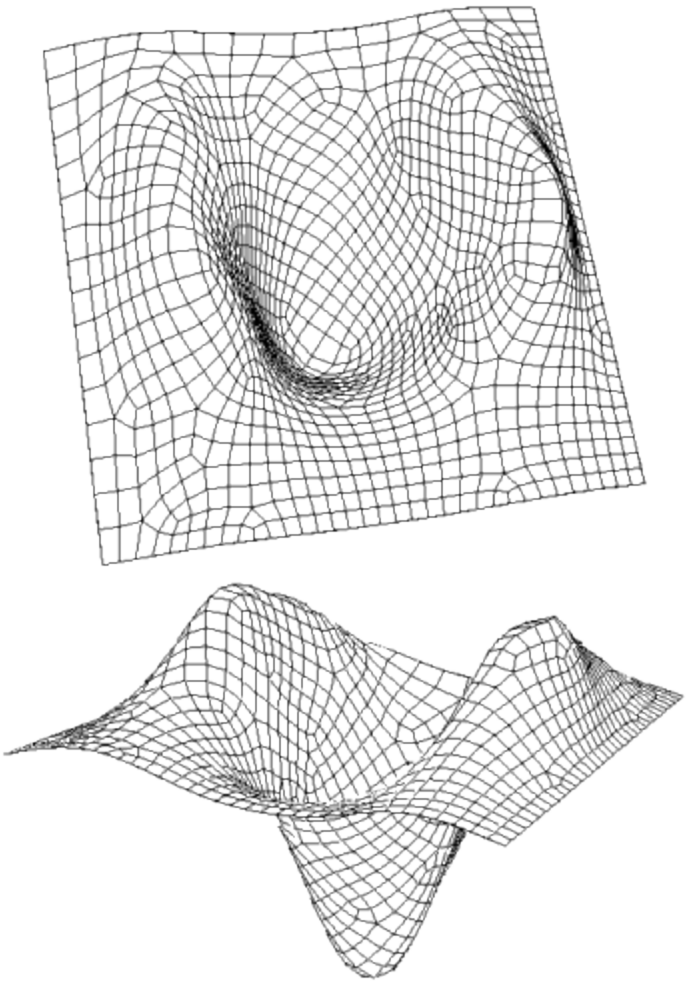


**Figure 2. NURB mesh with linear sizing function, 34 by 16 density**

# Free Meshes

A free mesh is a mesh that is not associated with any underlying geometric entities. A free mesh contains only mesh elements (hexahedra, triangles, edges, nodes, etc), and not volumes, surfaces, etc. Since there is no underlying geometry, operations on free meshes are limited. The following operations can be performed on free meshes in some capacity:

- Creating a free mesh
- Creating mesh-based geometry to fit a free mesh
- Mesh merging
- Mesh transformations
- Mesh smoothing
- Mesh quality operations
- Mesh refinement
- Cleaning up a free mesh

- Assigning boundary conditions
- Skinning a free mesh
- Mesh deletion
- Bottom-up element creation
- Exporting a free mesh

## Creating a free mesh

A free mesh can be created in three ways.

1. Importing a mesh into Cubit using the **Import Mesh [No_Geom]** command. This option is discussed in detail in Importing Exodus II Files.
2. Disassociating an existing mesh from its geometry
3. Creating a mesh with the geometry-tolerant mesh scheme

*Disassociating a mesh from its geometry*

The command to disassociate a mesh from existing geometry is:

```
Disassociate Mesh [From] {Volume|Surface|Curve|Vertex} <id_range>
```

For example:

```
brick x 10
mesh volume all
disassociate mesh from volume 1
delete volume 1
```

When a mesh is disassociated from its geometry, a group called 'disassociate elements' is created to contain the free mesh.

## Creating Mesh-Based Geometry to fit a Free Mesh

It is possible to create underlying mesh-based geometry to own a free mesh. It is similar in functionality to the Import Mesh Geometry command, but it does not require the extra import/export step. For example, a user would be able to read in a free mesh, fix any mesh problems, and then create the mesh-based geometry without having to write the mesh to a file first. The command syntax is:

```
Create Mesh Geometry {Hex|Tet|Face|Tri|Block} <range> [no_nodeset]
[no_sideset] [exclude block <range>] [Feature_Angle <angle=135>] [Acis]
[Keep]
```

The command also applies to any subset of the mesh. For example, you can create mesh geometry for a group of hexes or element blocks.

If the **keep** option is specified, the mesh will be duplicated so you will have two copies of the mesh: The original mesh and the new mesh that is owned by the new MBG geometry. If the keep option is not specified, the existing mesh will be reused, and duplicate elements will not be created. Elements will now be owned by the new MBG geometry. The command will check for mesh ownership and will issue a warning. Use the keep option if the mesh is already owned. The keep option is not specified by default.

By default, genesis entities will be used as criteria for building the new MBG geometry. The **no_nodeset** and **no_sideset** options can be specified to prevent this. Any genesis entities defined on the free mesh are transferred to the new MBG geometry. When the **keep** option is used copies of genesis entities are made on the new MBG geometry.
The **exclude block** option excludes specified blocks from feature angle detection. Any volumes created from these excluded blocks will have only one surface.
The **Acis** option will attempt to create ACIS geometry from the mesh. This option is an alpha feature and can only be used if developer commands have been turned on. For more detail see: Acis Geometry From Mesh

## Merging a free mesh

To merge two free meshes, the equivalence command may be used. The command syntax is:

**Equivalence Node <range> [Tolerance <value>] [Preview]**

All nodes in the given range that are within the specified tolerance will be merged. The merged and unchanged (unmerged) nodes are put into groups. The maximum distance between merged nodes, and the minimum distance between unmerged nodes, are listed for the user because if these values are close together it can indicate a problem with the Tolerance. Nodes within tolerance and part of the same element will not be merged. This prevents collapsing elements into degenerate forms. With the Preview option, the nodes aren't actually merged, but the ones that would have merged are drawn and grouped. For example:

```
br x 10
volume 1 copy move x 10
mesh volume all
disassociate mesh from volume 1 2
delete volume 1 2
equivalence node all tolerance 0.05
## merges all nodes that are within 0.05 of each other
```

## Free Mesh Transformation Operations

Mesh transformations for free meshes are achieved through the use of the group transformation commands, given in Basic Group Operations. All members of a free mesh are automatically assigned to a group. These groups can then be modified using group operations. The following command sequence illustrates how transformations might be applied to a free mesh.

```
brick x 10
mesh volume 1
disassociate mesh from volume 1
delete volume 1
group disassociated_elements move x 10
group disassociated_elements rotate 15 about x
group disassociated_elements scale 0.25
group disassociated_elements reflect 1 1 0
group 'node_group' add node 1 to 121
```

**group node_group move z 5**
##The moved nodes do not also move the attached geometry, as one might expect.
If a group is composed of mesh entities, these commands will only operate on the nodes in the group. All nodes of the group will be moved, scaled, rotated, or reflected as specified. If there are no nodes in the group, Cubit will return an error. Including all nodes in the group will transform the whole model. Including only a subset of nodes will transform those nodes and their enclosed elements, but it will not transform the whole mesh.
Disassociated mesh elements cannot be copied using the Group copy commands. To create a copy they must be exported and reimported. Alternatively, they can be associated with mesh-based geometry, and then copied using the typical copy commands.

*Extruding Mesh Elements*

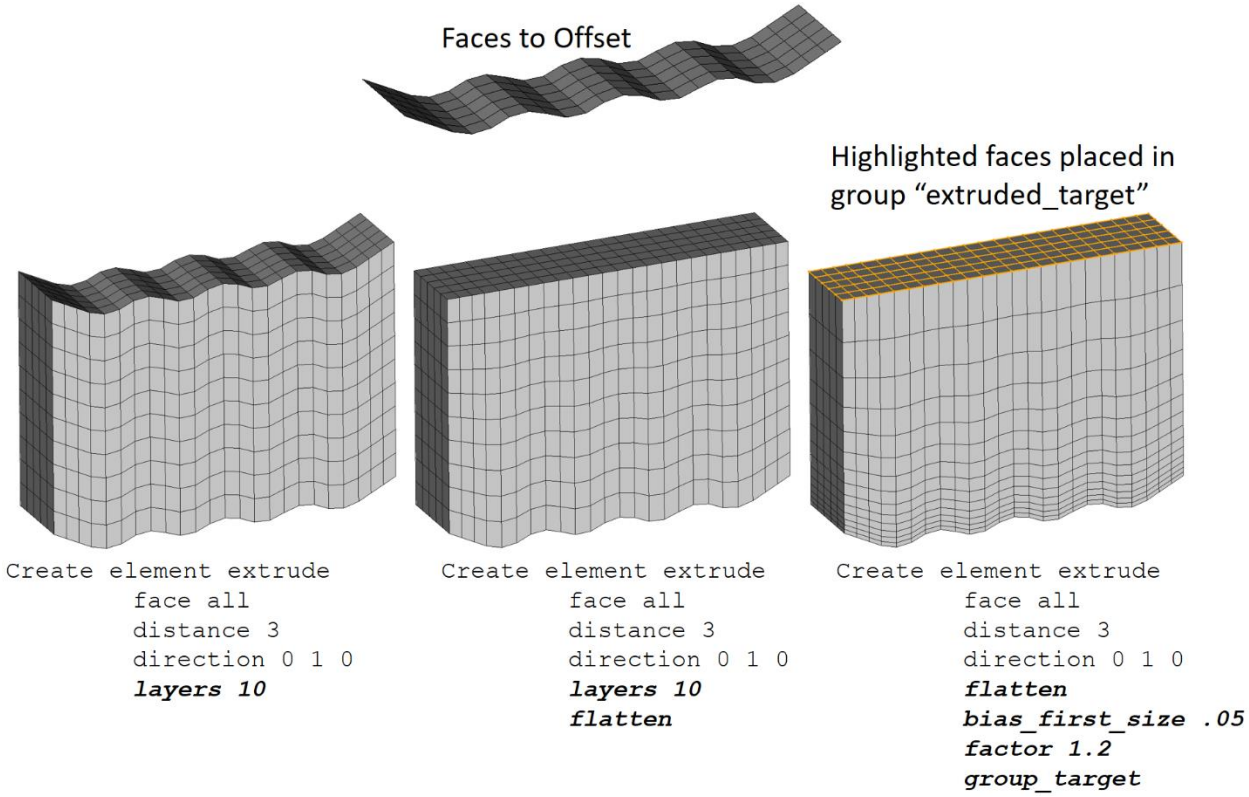
Mesh elements can be extruded to create new elements from existing nodes, edges, faces or triangles. There are two forms of the extrude command as follows:

**Create Element Extrude {Node|Edge|Face|Tri} <element_list> Direction <options> [Distance <value>] {Layers <num_layers | {bias_first_size <value> factor <value>}} [Twist <angle> Axis <axis_options>] [flatten] [group_target]**

**Create Element Extrude {Node|Edge|Face|Tri} <element_list> Along Curve <curve_list> [Layers <num_layers]**

In the first form of the command, a direction and **distance** are specified to define the extrusion. To define node spacing along the extrusion, the command takes either the **layers** or **bias_first_size** and **factor** parameters, but not both. Specifying a value for the **layers** option determines how many evenly sized elements will be created in the given distance. The **bias_first_size** and **factor** parameters define a bias that will be used along the extrusion distance. When specifying entities to be extruded which are either non-planar, or not orthogonal to the extrusion direction, the **flatten** parameter results in the extrusion terminating on a single plane orthogonal to the extrusion direction. The optional **group_target** parameter places all of the faces, tris, edges, or nodes at the end of the extrusion into a group named "extruded_target" which can be conveniently used to define a subsequent extrusion with a different set of extrusion parameters. **Twist** can also be specified and requires an angle of twist and a twist axis.

The figure below illustrates the new **flatten**, **bias_first_size**, **factor**, and **group_target** keywords. At the top is a set of non-planar faces to offset. On the bottom-left, each node on all of the faces are extruded the same amout resulting in a target of the extrusion with identical curvatuve to the input faces. In the bottom-middle, the **flatten** keyword is used, resulting in the extrusion terminating on a single plane. In the bottom-right, the **bias_first_size**, **factor**, and **group_target** keywords are used, resulting in a biased extrusion, and the creation of a group named "extruded_target", which contains the faces on the end of the extrusion.




In the second form, a curve is specified, along with the input entities will be extruded. Extruding along a curve supports the **layers** parameter, but does not currently support the **distance**, **bias_first_size**, **factor**, **flatten**, **twist** or **group_target** parameters.

```
#Extrude a face in a given direction:
create node location 0 0 0
create node location 1 0 0
create node location 1 1 0
create node location 0 1 0
create face node 1 to 4
create element extrude face 1 direction 0 0 1 distance 3 layers 3
create element extrude face 1 direction 0 0 1 distance 3 layers 3 twist 90 axis direction 0 0 1 origin 0 0 0
#Sweep face along curve
create node location 0 0 0
create node location 1 0 0
create node location 1 1 0
create node location 0 1 0
create face node 1 to 4
create vertex location position 0 0 0
create vertex location position 0 .2 1
create vertex location position 0 1 2
create vertex location position 0 3 2
create vertex location position 0 4 1
create vertex location position 0 5 0
```

**create curve spline vertex 1 2 3 4 5**
**create element extrude face 1 layers 5 along curve 1**

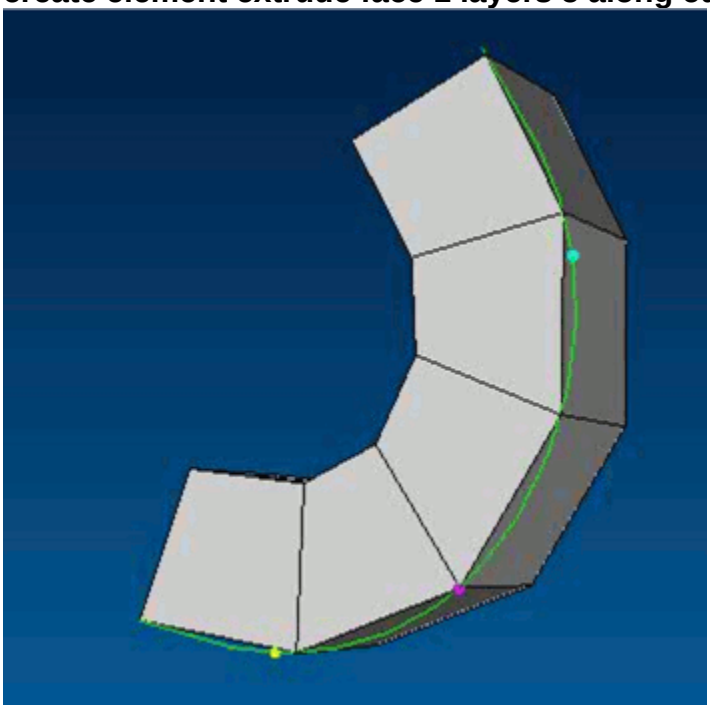


**Figure 1. Extruding mesh elements along a spline**

*Offsetting Mesh Elements*

Faces and triangle elements can be used to create hexahedral and wedge elements from an offset command. The default offest direction is normal to the selected face. The **Oppposite_normal** option will use the reverse direction. The **layers** parameter determines how many elements will be created in the given direction.

**Create Element Offset {Face|Tri} <element_list> [Normal_to|Opposite_normal] {Distance <value>] [Layers <num_layers>]**

#Create wedge and hex elements from face and tri elements via offset
**create node location 0 0 0**
**create node location 1 0 0**
**create node location 1 1 0**
**create node location 0 1 0**
**create node location 2 0 1**
**create node location 2 1 1**
**create node location 1 2 0**
**create face node 1 to 4**
**create face node 3 2 5 6**
**create tri node 7 4 3**

**create tri node 7 3 6**
**create element offset face all tri all distance 3 layers 3 opposite_normal**

*Revolving Mesh Elements*

Elements can be created by revolving an existing element around a given axis. The **Attempt_fix** parameter will try to fix any poorly formed hex elements by collapsing them into wedge elements. **Angle** determines the amount of rotation around the axis. The **Layers** option determines how many elements will be created in the given rotation. The **quadratic** option will result in the creation of quadratic elements. For example, it can result in HEX20 or TET10 elements being created. To further specify the element type, for the created elements, one may put them into a block.

**Create Element Revolve {Edge|Face|Tri} <element_list> Axis <axis_options> Angle <angle> [Layers <num_layers>] [Attempt_fix] [quadratic]**

#Revolve 2 faces around the Y-axis and collapse inner hexes to wedges
**create node location 0 0 0**
**create node location 1 0 0**
**create node location 1 1 0**
**create node location 0 1 0**
**create node location 2 0 0**
**create node location 2 1 0**
**create face node 1 2 3 4**
**create face node 2 5 6 3**
**create element revolve face 1 2 axis direction 0 1 0 angle 180 layers 4 attempt_fix**
**block 1 hex all**
**block 2 wedge all**
**block 1 element type hex8**
**block 2 element type wedge6**

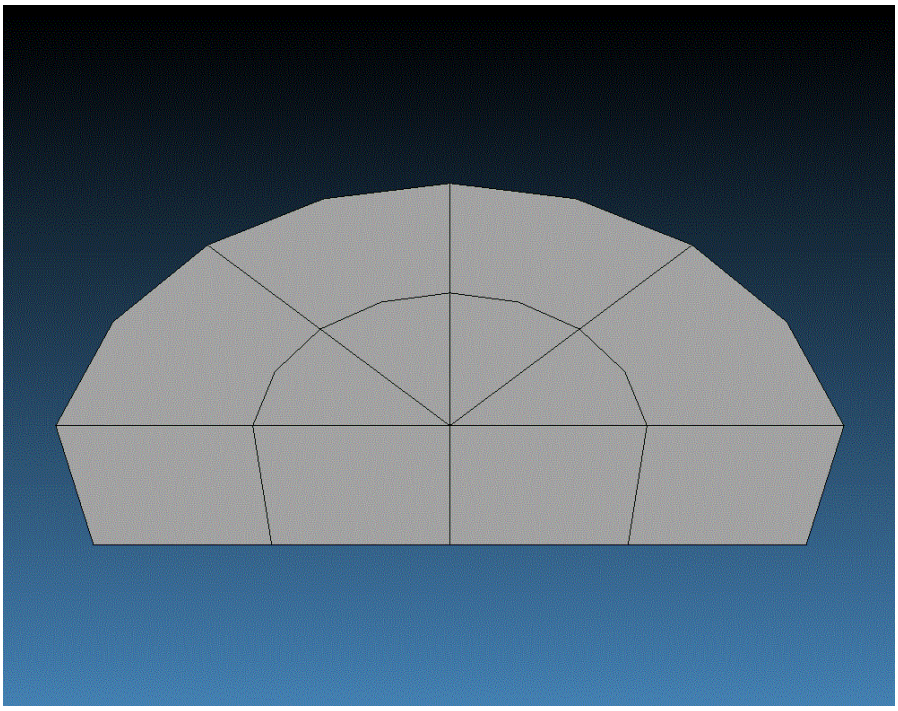


**Figure 2. Revolving free mesh elements to create hex and wedge elements**

## Smoothing a free mesh

Interior nodes can be smoothed using commands such as *smooth hex all*, or *smooth tet all in block 100*. These commands will smooth only the interior node on the elements used in the command. The nodes on the boundary will remain unchanged. To smooth nodes on a boundary, the target smoothing option can be used. Targeted smoothing allows the user to smooth a group of mesh elements to a surface or curve that is not their owner. Targeted smoothing is discussed under Mesh Smoothing. The following sequence of commands illustrate the capability of smoothing a free mesh to a target surface.

```
sphere rad 25
webcut vol 1 plane xplane offset 18
delete vol 2
webcut volume 1 plane yplane offset 8
webcut volume 1 plane yplane offset -8
delete vol 1 3
surf 16 copy
delete vol 4
surf 18 scheme pave
surf 18 size 2
mesh surf 18
disassociate mesh surf 18 ##Mesh and geometry overlap
refine face 1 radius 3
set developer on ## Smoothing free mesh is a developer command
smooth face all scheme laplacian
##Smoothed mesh is away from surface
```

**smooth face all scheme laplacian target surface 18**
##Smoothed mesh is aligned with surface

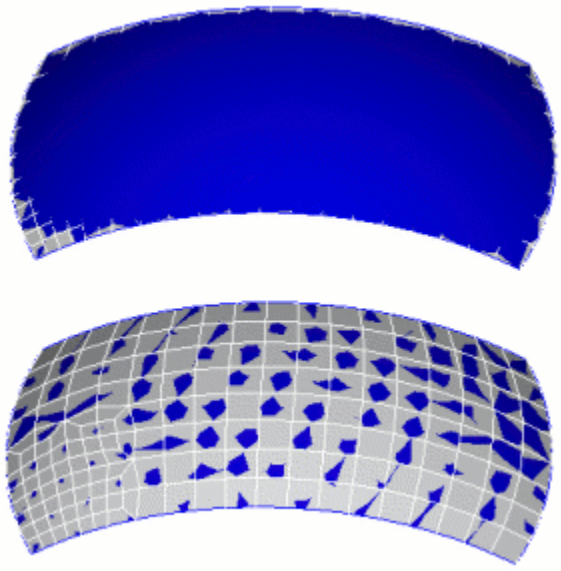


**Figure 3. Smoothing without a target (above) and smoothing to a target surface (below).**

## Mesh quality on a free mesh

The mesh quality checks for a free mesh are the same as for other geometry-based meshes. The difference is in how you specify elements in the command. Instead of specifying volumes or surfaces you would specify groups of hexes, faces, tris, or tets. Examples are given below:

**quality hex all**
**quality face all scaled jacobian**
**quality tet 1 to 100 draw mesh**

## Mesh refinement on a free mesh

Refinement for a free mesh is limited to refinement of mesh elements. Refinement may be accomplished by specifying groups of mesh elements which to refine using the regular refinement options. For boundary elements, the refinement scheme will use averaging methods to determine node placement, in the absence of a boundary geometry to define node placement.

## Cleaning up a free mesh

A free tet mesh may be cleaned up using the Cleanup Tet command. For example

**cleanup tet all**
#cleans up all tets
**cleanup tet 1 to 1000**
#cleans up all tets in the range [1,1000]

It is best to specify contiguous sets of elements for this command.

## Assigning boundary conditions

Assigning boundary conditions on free meshes can be accomplished by explicitly specifying mesh elements, by creating a sideset or block from the skin of a group of

elements, or by creating groups based on feature angle using the seed method. Once the group is created it is easy to assign it to a nodeset or sideset.
Cubit will respect block, nodeset, and sideset data that is associated with an imported free mesh, or disassociated mesh. The following command sequence illustrates how the group seed operation could be used for assigning boundary conditions on free meshes.

```
##Creating blocks, nodesets and sidesets on free meshes
cylinder radius 3 z 12
volume 1 size 0.5
mesh volume 1
disassociate mesh from volume 1
delete volume 1
group 'mygroup1' add seed face 752 feature_angle 45
##Groups all faces on the cylindrical surface
group 'mygroup2' add seed face 752 feature_angle 45 divergence
##Groups only faces within 45 degrees of seed face
sideset 1 group mygroup1
sideset 2 group mygroup2
block 1 hex all
draw sideset 1
draw sideset 2
draw block 1
```

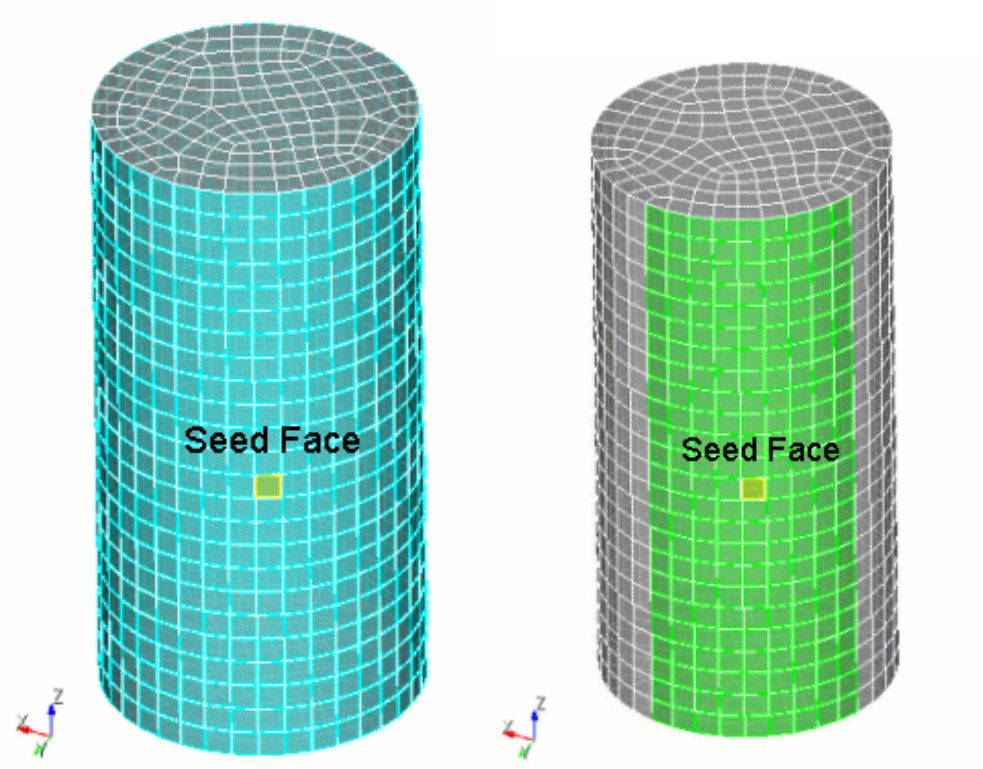



**Figure 4. Grouping faces on free meshes using the seed method. The feature angle method is used on the left with a feature angle of 45 degrees. On the right is the result if using the divergence method.**

Even though boundary conditions can be defined directly only on geometry entities, these geometry-based BCs will be maintained on the free mesh following the disassociate command. The following command line sequence illustrates this capability.

```
##Respecting blocks, nodesets and sidesets in mesh elements after disassociation
brick x 10
mesh vol 1
sideset 1 surface 1
nodeset 1 curve 1
block 1 volume 1
disassociate mesh from volume 1
draw sideset 1
draw nodeset 1
draw block 1
```

## Skinning a free mesh

The skin command takes a list of mesh elements and returns the triangles and faces on the boundary of that group. The group of elements returned from the command can be assigned to either a group, sideset, or block. Free meshes can be skinned by specifying either a list of hexahedra, a list of tetrahedra, or a list of blocks.

## Deleting free mesh elements

Typically meshes are deleted by specifying owning geometry. For free meshes, the meshes cannot be deleted in this fashion. Instead, the mesh may be deleted using the **Delete mesh** command. The syntax is:

```
Delete Mesh
```

This command will delete all mesh entities in the entire model. To specify groups of elements for deletion, you can use the individual deletion commands. The command to delete a group of free mesh elements is:

```
Delete {Node|Hex|Tet|Face|Tri} <id_range> [No_propagate]
```

When deleting elements, the default behavior will be that the child mesh entities will be deleted when they become orphaned. For example, when a hex is deleted, if its faces, edges and vertices are no longer used by adjacent hex elements, then they will also be deleted. The **no_propagate** option will leave any child mesh entities regardless if they become orphaned.

## Bottom-up element creation

Bottom-up mesh element creation methods are available for free meshes. The difference between element creation methods for free meshes versus associated meshes is that the free meshes commands do not have a command option to associate the elements with an owning body. Otherwise the commands are identical to mesh element creation commands for associated meshes. The command syntax for free meshes is:

```
Create Node <x> <y> <z>

Create {Hex|Tet|Tri|Face|Edge} Node <id_range>
```

### Exporting free meshes

Free meshes can be exported as ExodusII files. All elements belonging to any block are exported. Any elements not belonging to a block will not be exported (i.e. Cubit will not assign default blocks).

# Mesh Deletion

Meshing a complex model often involves iteration between setting mesh parameters, meshing, and checking mesh quality. This often requires removing mesh, for only an entity or for an entity and all its lower order geometry, or sometimes for the entire model. The command to remove all existing mesh entities from the model is:

```
Delete Mesh
```

The command for deleting mesh on a specific entity is:

```
Delete Mesh {geom_list} [Propagate]
```

These commands automatically cause deletion of mesh on higher dimensional entities owning the target geometry.
If the Propagate keyword is used, mesh on lower order entities is deleted as well, but only if that mesh is not used by another higher order entity. For example, if two surfaces (surfaces 1 and 2) sharing a single curve are meshed, and the command "delete mesh surface 1 propagate" is entered, the mesh on surface 1 is deleted, as well as the mesh on all the curves bounding surface 1 except the curve shared by surface 2. In some cases, the capability to delete individual mesh faces on a surface is needed. Deleting a mesh face involves closing a face by merging two mesh nodes indicated in the input. The syntax for this command is:

```
Delete Face <face_id> Node <node_id> [Node <diagonal_node_id>]
```

This command is provided primarily for developers' use, but also provides the user fine control over surface meshes. At the present time, this command works only with faces appearing on geometric surfaces and should be used before any hex meshing is performed on any volume containing the face to be deleted.

### Automatic Mesh Deletion

Cubit will automatically delete the mesh from a geometry that is about to be modified by a geometry modification command. To change this behavior, so that Cubit will issue an error instead of automatically deleting the mesh, use the following command.

```
Set Mesh Autodelete [ON|Off]
```

# Mesh Validity

After a mesh is generated, it is checked to ensure that the mesh has valid connectivity. If an invalid mesh is formed, then CUBIT automatically deletes it. This default behavior can be changed with the following command:

**Set Keep Invalid Mesh [on|off]**

The current behavior can be viewed with the following command:

**List Keep Invalid Mesh**

The Jacobian quality metric is also computed automatically to check quality after a mesh is generated. If the quality is poor, a warning is printed to the terminal.

# Skinning a Mesh

The Skin command takes a range of hexahedra, tetrahedra, blocks, or volumes and generates a collection of triangles or quadrilaterals on the exterior of the volumetric elements. This is the skin mesh.

**Skin {Block|Volume} <range> [Individual] [Nomake]**

**Skin {Element|Hex|Tet|Wedge|Pyramid|Face|Tri|Block|Volume} <range> [Nomake]**

**Skin {Element|Hex|Tet|Wedge|Pyramid|Face|Tri|Block|Volume} <range> [Make {Block|Sideset [<id>] |Group [<name>|<id>]}**

**Skin {Element|Hex|Tet|Wedge|Pyramid|Face|Tri|Block|Volume} <range> {Add|Replace} {Block|Sideset [<id>] |Group [<name>|<id>]}**

The **Individual** keyword tells Cubit to skin Blocks or Volumes, one by one independently of each other, even if they share merged surfaces.
The **Nomake** keyword tells Cubit to not create any kind of grouping of the mesh faces resulting from the skinning operation.
If the **Make** option and its arguments are present, then the specified object (block, sideset or group) receives the skin mesh. The command fails if an object with the optional identifier already exists. If the object identifier is omitted, the identifier is set to the next object of that type. The skin mesh is stored in the next available sideset if the Make option is missing.
Another command form has two options, **Add** and **Replace**. Each option has a required, associated identifier. If the identifier is missing or invalid, the command fails. The Add option appends the skin mesh to the object. The Replace option removes any existing mesh from the object before adding the skin mesh.
The skin mesh will respect the merged volumes. If two adjacent volumes are merged, the skin mesh will not include the merged surface. If the volumes are not merged, each volume will generate a separate skin surface. If volumes are not merged, they are treated

separately. The skin command will also respect any number of interior voids. All surface elements will be oriented forward with respect to the originating volumes.
The primary use for the skin command is to generate surface meshes of quads or tris for sidesets and remeshing.
For **Face** and **Tri** elements (2d elements), the skin is a set of edges (1d elements.) The skin for 3d elements is a set of 2d elements.

# Lite Meshes

Cubit has the ability to represent mesh using either a lightweight or heavy representation. The lightweight representation option is new for Cubit 15.3, and can be referred to as lite. The heavy representation is useful for supporting all the various mesh manipulation operations available in Cubit. While still under development, the lite representation option is intended to be a quick way to display larger meshes while supporting a smaller subset of mesh manipulation operations.
The following are supported operations with lite mesh:

- Creating a lite mesh
- Graphics
- Information
- Modifications to lite mesh
- Exporting a lite mesh
- Equivalencing of nodes

The following items are not yet supported:

- Information and listing of individual hexes, tets, wedges, pyramids, faces, tris and edges
- Modifying the content of blocks, sidesets and nodesets
- Modifying the element type of a block
- Renumbering blocks, sidesets and nodesets
- Topology checks on blocks
- Cleanup on blocks
- Skinning on blocks
- Quality calculations on elements
- Moving blocks or nodesets
- Deleting individual blocks, nodesets and sidesets
- Creating rebar elements
- Smoothing elements
- Refining the mesh
- Merging nodes

## Creating a lite mesh

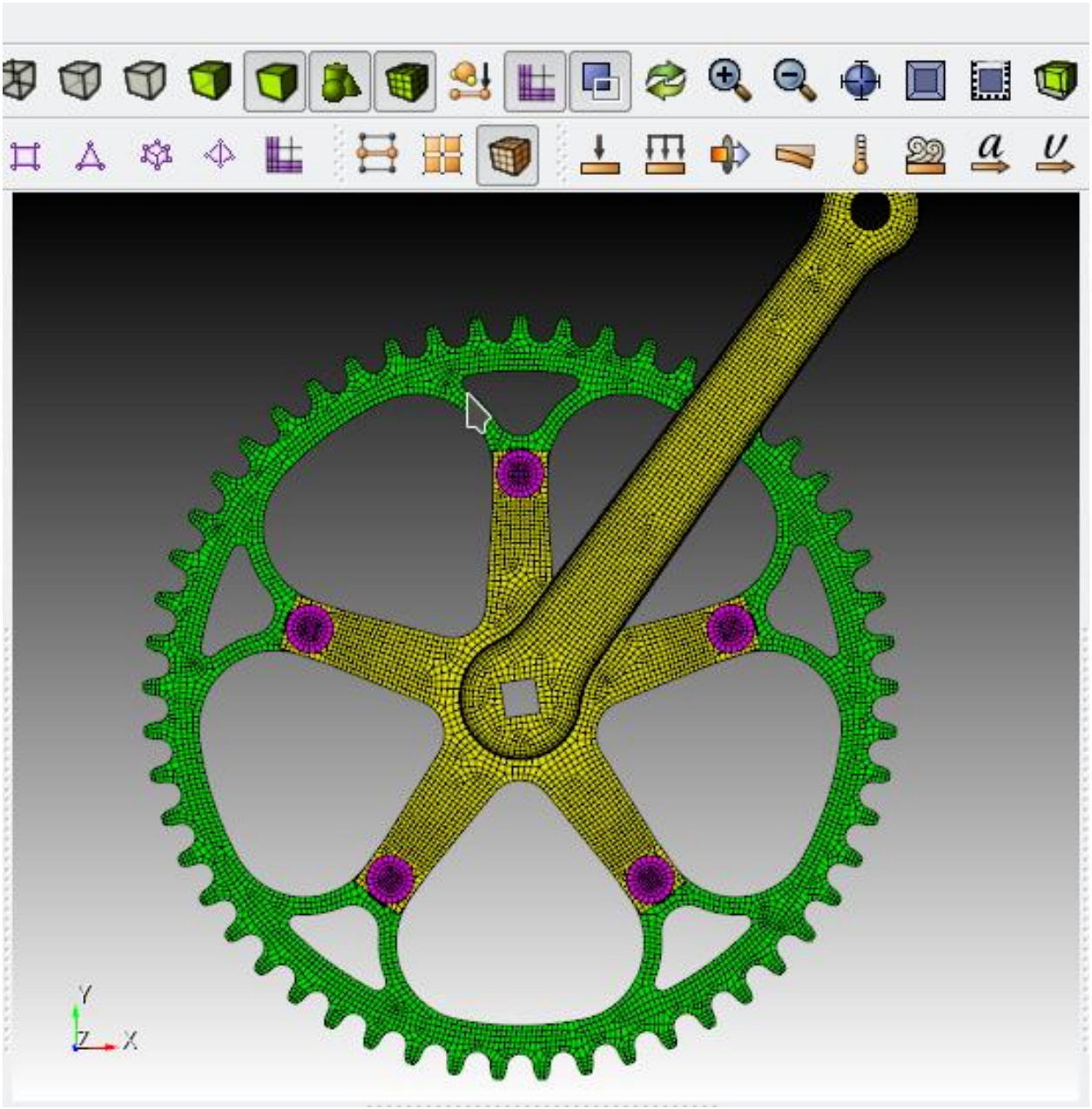
To use the lite mesh representation, one may import a mesh file using the 'lite' option. There is not currently another way to create lite mesh other than importing from a file. The command to import a lite mesh is:

**Import mesh "<filename>" lite**

Additional options for lite import can be found under the Import Mesh Lite command.

## Graphics

Generally, the graphical features for lite meshes is supported at the same level as for heavy meshes, including the ability to draw, pick, select, highlight, zoom among other operations. The coloring of the mesh is based on blocks, and may be adjusted by the user. Toggling visibility of all sidesets and nodesets can be done by clicking the Display Boundary Conditions toolbar button or with the **bc visibility {on|off}** command. Toggling visibilty of all blocks can be done by clicking the Display Mesh toolbar button or with the **mesh visibility {on|off}** command.



The draw, zoom and select commands work on blocks, sidesets and nodesets. Also, selecting those genesis entities in the graphics window will result in them being highlighted in both the graphics window and in the tree. Selecting of nodes and elements has not yet been implemented for meshes imported in lite mode.

### Information

There are several ways to view information about the lite mesh. The tree and the property page can show information about the blocks, sidesets and nodesets. Also, the **list** command can print information about individual blocks, sidesets, and nodesets. The **list element** command will print out the ID space used by elements. The **list node** command will show the ID space used by nodes. Listing of individual elements and nodes is not yet supported.

### Modification to lite mesh

Some modifications to genesis entities are supported. Blocks, sidesets and nodesets may have names assigned to them. Blocks may have their attributes modified, and materials may be assigned to blocks. Not supported is the ability to modify the contents of blocks, nodesets and sidesets.

### Exporting a lite mesh

Exporting a lightweight mesh to an Exodus file is supported. This includes writing out blocks, nodesets, sidesets, element ids, node ids, etc... Not all Exodus data is read in, and if there is some data not recognized by Cubit, it will not be exported. Field data is an example of Exodus data not recognized by Cubit, nor exported. Importing multiple Exodus files and exporting a single file is supported. Importing a single Exodus file and exporting a portion of it is supported. Distribution factors are preserved when reading/writing in lite mode.

### Equivalencing of nodes

Equivalencing of nodes in a lightweight mesh is supported using this command.

**Equivalence Node <range> [Tolerance <value>] [Preview]**

All nodes given and within tolerance of each other will be merged. More information on equivalencing nodes can be found here

# Finite Element Model

## Finite Element Model


- Global Element IDs
- Exodus Boundary Conditions
- Non-Exodus Boundary Conditions
- Exporting the Finite Element Model


This chapter describes the techniques used to complete the definition of the finite element model. The definitions of the basic items in an Exodus database are briefly presented, followed by a description of the commands a user would typically enter to produce a customized finite element problem description, and how to export the finite element model.

## Global Element IDs

### Cubit Mesh Entity ID Spaces

All mesh entities have an ID associated with them which is unique within the corresponding mesh entity ID space. For example, a hex will have an id which can be used in commands such as "list hex 17". However, this ID is only unique amoungst the hexahedra in the Cubit session. There could also be a tet, quad, tri, edge, node, etc. with ID 17.

### Global Element IDs

Whenever a hex, tet, quad, tri, etc. gets put into an element block, it is assigned another ID which is called the *Global Element ID*. The *Global Element ID* is unique amoungst all element which have been put into any block. Starting in Cubit 14.0, it is exported to the Exodus file format so that downstream analysis applications can map elements back to the corresponding Cubit hex, quad, etc. In Cubit 14.0, Global Element are not supported by the other exporters, but will be in future releases.

*Interacting with GlobalElement IDs*

After a hex, quad, etc. is placed into an element block, you can see which ID was assigned to it with the **list** command. For example:

```
reset
bri                                    x                                    10
mesh                                  vol                                  all
block                      1                      hex                      all
list hex 1
```

The resulting output will contain the following:

```
CUBIT>                     list                      hex                      1
Hex                                                                          1
Global Element ID = 1
```

In this simple example, the Hex ID is the same as the Global Element ID, but this will not always be true.

These Global Element IDs are exported as the global id in the Exodus file. If during an analysis run, a particular element needs to be identified back in the Cubit session, they can be found with any of the following commands:

```
List element <id_range>

Draw element <id_range>

Highlight element <id_range>

List element <id_range>
```

Users can control the assigned *Global Element ID* with the renumber command.

# export

## Export Mesh and Its Geometry Association

Cubit offers the option to export a complete finite element mesh, along with its association to an ACIS geometry model. This is useful if a 3rd party application is going to be used to modify the mesh after exporting from Cubit, and you want the geometry available to project to during the modification operations. The command is:

```
Export m2g '<fileroot>' [{opennurbs|ACIS}] [overwrite]
```

The **fileroot** argument to this command is not a complete filename, rather, it is a full path and filename without the file extension. The **export m2g** command will write out:

- **fileroot.XYZ:** A geometry file containing the geometric model, where XYZ is the 3 letter file extension of the requested geometry output.
- **fileroot.exo:** The full mesh in the exodus format.
- **fileroot.m2g:** An ascii file defining how the mesh in fileroot.exo is associated to fileroot.sat.

The **export m2g** command can export either an ACIS (*.sat) file or an OpenNurbs (*.3dm) file depending on the supported geometry types of the downstream 3rd party application in use. The default is to export an ACIS file.
An example usage of the **export m2g** command is the Sierra mesh_scale command. Sierra mesh_scale is a batch program which performs the same mesh scaling as can be performed in Cubit with the **scale mesh** command. When generating new nodes on the boundary of the mesh, Sierra mesh_scale projects the locations of the new nodes to the ACIS geometry model by leveraging the information exported by the **export m2g** command.

## Exporting Sierra Files

Sierra input decks can be exported from Cubit.  This capability was added in response to a need to translate Abaqus input decks to Sierra input decks by importing the Abaqus deck into CUBIT and then immediately exporting the Sierra deck. Therefore, it is

assumed that most of the input deck information has been created outside of CUBIT and that the user will not interact with it in CUBIT .

The Sierra input deck writer is simply another export format and as a result it can be used for any currently defined mesh and input deck info defined in Cubit.

The Sierra input deck exporter relies on some of the mesh-specific information that is generated when exporting the Genesis mesh. Therefore, you should **export the Genesis mesh before exporting the** Sierra **input deck.**

## Defining PARAMS for NASTRAN

**List Nastran Exporter Params**

**Set Nastran Exporter Params Add '<param_string>'**

**Set Nastran Exporter Params Remove '<param_string>'**

**Set Nastran Exporter Params Clear**

Nastran uses “PARAMS” to define additional instructions and settings in its Bulk Data file. Any string can be defined as a Nastran Exporter Param, and it will be exported to the Nastran file as “PARAM, <string>”.

## Exporting ABAQUS

Mesh can be exported from CUBIT in the ABAQUS format. The command to export to ABAQUS is:

```
Export Abaqus <’filename’> [Block <id_list>] [Sideset <id_list>] [Nodeset
<id_list>] [BCSet <id_list>] [Group <id_list>] {[Instance Block <id_list>
[Source_csys <id>] Target_csys <id_list>] | [Instance_per_block]} [partial]
[Overwrite] [Everything]
```

The ABAQUS file written by CUBIT will contain a single part, or mulitple parts. Multiple parts can be defined by using the **instance** option. Additionally, a single block/part can be instanced multiple times by giving multiple target coordinate systems. A bolt mesh used several times is an example where one might want multiple instances. To instance a block, a source coordinate system and a target coordinate system (where the mesh will be translated and rotated to) need to be defined. If no source coordinate system is given in the command, the default (global) coordinate system is used. The default (global) coordinate system can be referenced specifcally using '0' for the id. The instance keyword can be used as many times as needed. For example, **block 1** can be instanced 3 times using 2 defined coordinate systems and the global coordinate system using:

```
create coordinate frame origin location 2 0 0 tag 'R'
```

```
create coordinate frame origin location -2 0 0 tag 'R'

Export Abaqus "myfile.inp" Instance Block 1 Target_csys 0 1 2
```

To enable automatic instancing, with the global coordinate system, based on defined blocks, the **Instance_per_block** option can be used.
Materials are supported with ABAQUS export as well. See documentation on materials for more information.

```
create material "Steel-200" property_group "CUBIT-ABAQUS"

modify material "Steel-200" scalar_properties "DENSITY" 7.8240

block 1 add material "Steel-200"
```

Additionally, Groups may be used to define additional node and element sets. For example:

```
group 'set-material-statistic-200' add tet in block 1 node in block 1
```

**Note**: By default, the Abaqus exporter writes 6 decimal places. The command "**set Abaqus precision <n>**" can be used to change the number of decimal places written.

## Exporting an Exodus II File

After defining the element blocks, nodesets and sidesets for a model, the model can be written to the Exodus II file using the command:

```
Export [Genesis|Mesh] '<filename>' [dimension {2|3}] [Block <id_list>]
[no_ids] [Qualityfile] [XML '<filename>'] [Overwrite]
```

**Note:** The ordering of options in this command is important. Misordering the options can cause the command to ignore some options.
The **Genesis** or **Mesh** arguments are optional and both indicate that an Exodus II format will be written. The filename can be any valid filename. Where a full path is not specified, the file will be written in the current working directory.
The **dimension** argument is also optional. Most element types have an inherent dimensionality associated with them. For example, a truss or beam element is inherently 2D while a hex or tetra element is 3D. Without this argument, only the x-y location of the nodal coordinates of 2D elements are written to the Exodus II file. Using the argument *dimension 3* in this example permits the full 3D coordinates to be written.
The optional **Block** argument may also be added to the **Export** command. Without this argument all blocks defined in the current model will be exported to the Exodus II file. This argument permits the user to specify only a subset of all the blocks in the model. The **<id_list>** may be any valid set of integers corresponding to the block ids in the current model.

**no_ids** prevents the writing of node and element id maps in the Exodus file. This can be useful if users do not want Cubit ids in downstream workflows.
The **Qualityfile** option exports, in addition to the mesh, a text file containing a printout of the element quality using the 'Allmetrics' option. The name of this file is the base name of the mesh file (file extension removed) and "_quality.txt" added.
The **XML** optional argument may also be added to the **Export** command. When this argument is included and assembly data exists in the model, an XML file is written which describes the relationship between block IDs in the Exodus II file and parts in the assembly. See the Parts, Assemblies and Metadata section for details.

*Element and Node ID Maps*

Element ID map and node ID map are always written to the Exodus II file. The IDs written to the node ID map are the node IDs used to refer to nodes at the Cubit command line. The IDs written to the element ID map are the Global Element IDs which are assigned to the hex, tet, quad, etc. when they are added to an element block. The node and element ID maps can be used when a particular element or node is refered to in a downstream application and the corresponding node or element in Cubit must be found. Some analysis and post-processing applications consider these maps to be optional, while others ignore the maps even if they are present. See the Exodus manual for more information on element and node ID maps.

*Exporting a Parallel Mesh for pCAMAL*

**Export Parallel "<filename>" [Block <id_list>] [Overwrite] [Processor <number>]**

The **Export Parallel** command is used to output an ExodusII file with the boundary mesh or shell for sweepable volumes that were meshed with set parallel meshing enabled. The options are the same as those for the "export genesis" command except for the addition of the processor option.
The processor option allows the user to specify the number of processors that will be used to mesh the volume with the pCAMAL option. This same option exists in the pCAMAL application and is more often used there since the number of available processors is known then rather than when the output file is created in Cubit.
If the processor option is given, Cubit attempts to balance the number of sweepable volumes to run on n processors by converting many-to-one sweeps to one-to-one sweeps, subdividing the sweep volume along its sweep direction, or partitioning the source surface of a one-to-one sweep if the number of source quads is much larger than the number of layers.

*Converting an Exodus II file to ASCII*

The Exodus II file format is binary. It is frequently necessary to view the contents of the Exodus II file as plain text. A publicly available tool known as **ncdump** can be used to

view the contents of an Exodus II file. **ncdump** is part of the **netCDF** library and is currently available from Unidata at the following URL:
http://www.unidata.ucar.edu/
On a UNIX platform, typical use of the **ncdump** utility is:

**ncdump filename.e > filename.txt**

In this format, the **ncdump** utility will take the Exodus II file, **filename.e**, and dump the contents to an ASCII file **filename.txt**
Another option for converting between binary and ASCII formats of Exodus II files is a utility known as **exotxt**. Exotxt is part of the SEACAS tool suite. Contact the Sandia CUBIT development team for a copy of this utility.
Note that the 'stock' ncdump utility should work for most meshes; however, Sandia increases some of the dimensions in order to handle larger meshes (more element blocks, boundary conditions, variables). The dimensions we increase in netcdf.h are:
**NC_MAX_DIMS** (max dimensions per file) from 100 to 65536
**NC_MAX_VARS** (max variables per file) from 2000 to 524288

*Controlling Exodus II Output Precision*

By default, exodus files are written with double precision numbers. It may be useful to change this for large meshes to decrease output file size. This can be done using the following command:

**Set Exodus Single Precision [On|Off]**

This command toggles the Exodus output file between single precision (floats) and double precision.

*Large Exodus Format*

The **Set Large Exodus** command enables the large exodus file setting to create a model that can store individual datasets larger than 2 gigabytes. This modifies the internal storage used by ExodusII and also puts the underlying netcdf file into the "64-bit offset" mode.

**Set Large Exodus [ON|Off]**

*Exodus NetCDF4/HDF5 Format*

The **Set Exodus NetCDF** command enables the exodus NetCDF4/HDF5 file setting to create a model that can store even larger files with unlimited dimensions. This modifies the internal storage used by ExodusII to an HDF5 based file. This setting overrides the **Set Large Exodus** setting.

**Set Exodus NetCDF4 [On|OFF]**

*Exporting Geometry Association with the Exodus Mesh*

Optionally, you can also export the associated ACIS geometry and the correspondence between the mesh and the geometry by using the **export m2g** command.

## Exporting the Finite Element Model

For information on exporting an Exodus File, see Exporting Exodus II Files. Custom translators are available to translate between the Exodus II format and a limited number of other analysis code formats. Contact the cubit development team for a current list of supported translator formats. For information on the GDF format, see Exporting GDF Files. The general syntax for the various exporters is as follows. The specific exporter commands are listed below.

**Export {Abaqus [Explicit]* [Partial]* | CGNS | Nastran | Ideas | Patran | LSDyna | Fluent} <'filename'> [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}***] [Overwrite] [Everything] [NX]****

**Export {Sierra | VRML} <'filename'> [Overwrite]**

* **Explicit** and **Partial** keywords only available with Abaqus Exporter
** **NX** keyword only available with I-DEAS Exporter
***The **dimension** argument is also optional. Most element types have an inherent dimensionality associated with them. For example, a truss or beam element is inherently 2D while a hex or tetra element is 3D. Without this argument, only the x-y location of the nodal coordinates of 2D elements are written to the Exodus II file. Using the argument dimension 3, in this example, permits the full 3D coordinates to be written.

The Abaqus Exporter has a few additional keywords available. See the last paragraph below for an explanation of those keywords:

> **Export Abaqus <'filename'> [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}] [nodefile <'filename'>] [elementfile <'filename'>] [flat] [overwrite] [everything]**

If no blocks are exported, Cubit will export all nodes and elements in the model. If one or more blocks are entered in the command, only those blocks will be exported. Similarly, if no BCSets are entered in the command, Cubit will export all boundary conditions as a single BCSet. If one or more BCSets are entered into the command, only those BCSets will be exported. Use the overwrite flag to overwrite an existing file.

By default, Cubit will reassign node and element IDs based on which block they are in. If the everything keyword is present, Cubit will export all nodes and elements in the model, whether they are in a block or not.

The I-DEAS Universal file can be read into Siemen's NX application if the file is generated using the **NX** keyword. This is because extra information must be written to an I-DEAS Universal file in order for NX to be able to read it.

There are a few keywords specifically for the Abaqus exporter. **Flat** can be used if the user desires Cubit to write out the model as a "flat file." Abaqus refers to files a "flat files"

when they do not use the *PART/*INSTANCE structure. All nodes and elements will be defined at the global level. The keywords **elementfile** and **nodefile** can be used to instruct Cubit to export the nodes and/or elements to a separate file.
If the **Explicit** keyword is used with Abaqus, Cubit will write an Abaqus Explicit deck. The one Explicit-only feature that Cubit supports is **Fixed Mass Scaling**.
If the **Partial** keyword is used with Abaqus, Cubit will write a partial Abaqus deck. Cubit will output the mesh as defined by the Abaqus keywords **PART, NODE, ELEMENT, NSET, ELSET, and SURFACE**. Everything else is ignored. Use the Abaqus keyword **INCLUDE** to include this file in a master Abaqus deck for analysis.
Specific Exporter Commands:

**Export Abaqus [explicit] '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [group <id_list>] [instance block <id_list> [source_csys <id_list>] target_csys <id_list> [preview]] [dimension {2|3}] [overwrite] [everything] [partial]**

**Set Abaqus Precision <n=6>**

Note: This command can be used to control the number of decimal places written to the Abaqus file.

**Export CGNS '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}] [overwrite] [everything]**

**Export Nastran '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}] [overwrite] [everything]**

**Export Ideas '<filename>' [NX] [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}] [overwrite] [everything]**

**Export Patran '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [overwrite] [everything][dimension {2|3}]**

**Export Lsdyna '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [overwrite]**

**Export Fluent '<filename>' [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>] [dimension {2|3}] [overwrite] [everything]**

Note: The following command is for exporting mesh geometry, (.**msh format**.)

**Export Fluent '<filename>' [Surface <id_list>|Volume <id_list>] [Overwrite]**

**Export Sierra <'filename'> [Overwrite]**

**Export VRML <'filename'> [Overwrite]**

[Additional Information on building Cubit models for CFD]
[Exporting ABAQUS]
[Defining PARAMS for NASTRAN]

*Supported element types*

| ExodusII | CGNS | Abaqus | Nastran | I-DEAS UNV | Patran | LS- |
|---|---|---|---|---|---|---|
| SPHERE | | | | | | ELEME |
| SPRING | | SPRINGA/SPRING1/SPRING2 | CBUSH1D** | | | |
| BAR | | B21** | CROD** | 121** | Bar2 | |
| BAR2 | | B21** | CROD** | 121** | Bar2 | |
| BAR3 | | B22** | CROD** | 121** | | |
| BEAM | | B31 | CROD** | 21** | Bar2 | ELEME |
| BEAM2 | | B31 | CROD** | 21** | Bar2 | ELEME |
| BEAM3 | | B32 | CROD** | 24** | | |
| TRUSS | | T3D2/T3D2T*,** | CROD** | 121** | Bar2 | ELEME |
| TRUSS2 | | T3D2/T3D2T*,** | CROD** | 121** | Bar2 | ELEME |
| TRUSS3 | | T3D2/T3D2T*,** | CROD** | 121** | | |
| QUAD | QUAD_4 | CPE4R/CPE4RT* | CQUAD4 | 54 | Quad4 | ELEMEN |
| QUAD4 | QUAD_4 | CPE4R/CPE4RT* | CQUAD4 | 54 | Quad4 | ELEMEN |
| QUAD5 | | | | | Quad5 | |
| QUAD8 | | CPE8R/CPE8RT* | CQUAD8 | 55 | Quad8 | |
| QUAD9 | | S9R5 | CQUAD | 55 | Quad9 | |
| SHELL | | S4R/S4RT* | CQUAD4 | 94*** | Quad4 | ELEMEN |
| SHELL4 | | S4R/S4RT* | CQUAD4 | 94*** | Quad4 | ELEMEN |
| SHELL8 | | S8R/S8RT* | CQUAD8 | 95*** | Quad8 | |
| SHELL9 | | S9R5 | | 95*** | Quad9 | |
| TRI | TRI_3 | CPS3/CPS3T* | CTRIA3 | 51 | Tri3 | ELEMEN |
| TRI3 | TRI_3 | CPS3/CPS3T* | CTRIA3 | 51 | Tri3 | ELEMEN |
| TRI6 | | CPS6/CPS6T* | CTRIA6 | 52 | Tri6 | |
| TRI7 | | | | 52 | Tri7 | |
| TRISHELL | | STRI3 | CTRIA3 | 91 | Tri3 | ELEMEN |
| TRISHELL3 | | STRI3 | CTRIA3 | 91 | Tri3 | ELEMEN |
| TRISHELL6 | | STRI65 | CTRI6 | 92 | Tri6 | |
| TRISHELL7 | | | | 92 | Tri7 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| HEX | HEXA_8 | C3D8R/C3D8RT* | CHEXA | 115 | Hex8 | ELEMEN |
| HEX8 | HEXA_8 | C3D8R/C3D8RT* | CHEXA | 115 | Hex8 | ELEMEN |
| HEX9 | | | | | | |
| HEX20 | | C3D20R/C3D20RT* | CHEXA | 116 | Hex20 | |
| HEX27 | | | CHEXA | 116 | | |
| TETRA | TETRA_4 | C3D4/C3D4T* | CTETRA | 111 | Tet4 | ELEMEN |
| TETRA4 | TETRA_4 | C3D4/C3D4T* | CTETRA | 111 | Tet4 | ELEMEN |
| TETRA8 | | | | | | |
| TETRA10 | | C3D10/C3D10MT* | CTETRA | 118 | Tet10 | ELEMEN |
| TETRA14 | | | | | | |
| WEDGE | | C3D6/C3D6T* | CPENTA | 112 | | ELEMEN |
| HEXSHELL | | | | | | |
| PYRAMID | | | CPYRAM | 115† | | |
| PYRAMID5 | | | CPYRAM | 115† | | |
| PYRAMID8 | | | CPYRAM | 116† | | |
| PYRAMID13 | | | CPYRAM | 116† | | |
| PYRAMID18 | | | CPYRAM | 116† | | |
| PERELEMENT_TOPOLOGY_XXX | | | | | | |

*Thermal element
**Check to make sure the element's properties are correct after exporting
***Also exports lofting factor for shell elements (IDEAS)
† The element type will be HEX but the number of nodes will be the number of nodes in the pyramid.

*Supported boundary conditions types*

| Cubit Element Type | ExodusII | CGNS | Abaqus | Nastran | I-DEAS UNV | Patran | LS-DYNA | Fluent |
|---|---|---|---|---|---|---|---|---|
| BC Set | | | *STEP | SUBCASE | 2428 | | | |
| Displacement | | | | SPC | 791 | 08 | | |
| Temperature | | | *BOUNDARY | TEMP | 791 | 10 | | |
| Force | | | *CLOAD | FORCE/MOMENT | 790 | 07 | | |
| Pressure | | | *DSLOAD | PLOAD4 | 790 | 6 | | |
| Convection | | | *SFILM *** | CONV | 790 | 17 | | |
| Heat Flux | | | *DSFLUX | QHBDY | 790 | 16 | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Contact | | | *CONTACT | | 2471 | | | |
| Materials | | | *MATERIAL | MAT1_, MAT4_ | 1716 | 03 | | |
| **CFD Boundary Conditions** | | | | | | | | |
| Interior | | | | | | | | 2 |
| Wall | | | | | | | | 3 |
| Inlet Pressure | | | | | | | | 4 |
| Inlet Massflow | | | | | | | | 20 |
| Inlet Velocity | | | | | | | | 10 |
| Outlet Pressure | | | | | | | | 5 |
| Far-field Pressure | | | | | | | | 9 |
| Symmetry | | | | | | | | 7 |

*** Does not allow separate temperatures for top and bottom of shell elements. Values will be averaged.

## Exporting Fluent Grid Files

Geometry can be exported from Cubit to the Fluent **.msh** format. This format can be used to exchange grid information between **.msh** compatible programs including Fluent, GAMBIT, and TGrid. The command used to export the mesh geometry is:

**Export Fluent '<filename>' [Surface <id_list>|Volume <id_list>] [Overwrite]**

The filename should be enclosed in either single or double quotes. By convention, the file extension .msh is applied to grid files. The extension should be included in the filename section. Other file extensions such as .cas may be used, but they cannot be guaranteed to be compatible with either GAMBIT or TGrid.
In order to guarantee that the grid file will be compatible with Fluent, all bodies must be merged (See Geometry Merging). Several types of Fluent boundary condition zones are now implemented in Cubit. They are:

- axis
- exhaust fan
- fan
- inlet vent
- intake fan
- interface
- interior
- mass flow inlet

- outflow
- outlet vent
- periodic
- periodic shadow
- porous jump
- pressure far field
- pressure inlet
- pressure outlet
- radiator
- symmetry
- velocity inlet
- wall

Boundary condition zones created in two different ways. The first way involves user-defined mesh groups consisting only of quads (3D), triangles (3D), or element edges (2D) (See Geometry Groups). The second way involves sidesets. Specifying a boundary condition consists of selecting a user-defined mesh group or a sideset, or a surface. Selecting a surface automatically assigns the boundary condition to the sideset associated with that surface. The boundary condition type is specified and is either given a name or an id (See Using CFD Boundary Conditions). Groups or sidesets of mixed type (e.g. hexes and faces) will not be exported. All surfaces not set to one of the first seven boundary condition types are automatically set to type 'wall'. The various parameters for each of the boundary condition types must be set within either Fluent or GAMBIT.
Cell zones are automatically created for 3D meshes containing blocks. Blocks must contain entire and continuous volumes in order to create a valid grid. In 2D models, the cell zones are created from sidesets containing only quads or tris. In order to create a valid grid, these sidesets must contain whole, continuous surfaces. All cell zones are by default set to type 'fluid.'
If no entities are specified, the entire model is exported. In order to export selected entities, the types 'volume' and 'surface' can be specified. In 2D cases, use 'surface' while in the 3D case use 'volume.'

## Transforming Mesh Coordinates

A mesh can be scaled and transformed to a new location as it is written to or read from an Exodus file. To transform a mesh during import or export use the following command:

```
Transform Mesh {Input|Output}
[Scale <xyz_factor>]
[Scale <x_factor> <y_factor> <z_factor>]]
[Scale {X|Y|Z} <factor>]
[Translate <dx> [<dy> [<dz>]]]
[Translate {X|Y|Z} <distance>]
[Rotate <degrees> about {X|Y|Z}]
[Reset]
```

This command may be repeated any number of times using any number of options. Transform commands are cumulative, added to the effect of previous transforms. If more than one transformation is entered in the same command, transformations are applied in the order they appear in the command.
To clear a transformation matrix, use the **Reset** option:

```
Transform Mesh {Input|Output} Reset
```

Mesh input and output transformations are also cleared when you reset the entire model using the **Reset** command.
Transforming a mesh during output **does not** change the position of the mesh within CUBIT. It only changes the nodal positions written to the Exodus file. Nodal positions may be changed within CUBIT by transforming the body that contains the mesh. See Geometry Transforms for information on how to apply transformations to a Body.
Transforming a mesh during input **does** change the position of the mesh with CUBIT. The file being read is not modified.
Transformations applied during mesh input are independent of transformations applied during mesh output.
The following example generates a simple mesh, writes the mesh with its coordinates scaled by a factor of 2, and then re-imports that mesh, restoring the scaling to what it originally was in CUBIT.

```
brick x 10
volume 1 interval 4
mesh vol 1
transform mesh output scale 2
export mesh 'temp.exo'
delete mesh
transform mesh input scale .5
import mesh 'temp.exo'
```

See Geometry Transforms for information on how to apply transformations to a Body.
See Nodeset and Nodeset Repositioning
See Importing a Mesh
See Mesh Based Geometry

## Coordinate Frames

CUBIT allows the user to define coordinate systems (frames) that are written to a mesh file. These coordinate frames are generally used as reference coordinate systems during analysis. In CUBIT, the user may define multiple exodus coordinate frames. When created, a coordinate frame is assigned an id. Coordinate frames can be created using x-y-z coordinates, nodes or vertices with the following commands:

```
Create Coordinate Frame
<xval> <yval> <zval>//origin
<xval> <yval> <zval> //z-axis
<xval> <yval> <zval> //xz-plane
[tag { 'R' | 'C' | 'S' } ]
```

**Create Coordinate Frame Node**
**<node_origin_id>**
**<node_zaxis_id>**
**<node_xzplane_id>**
**[tag { 'R' | 'C' | 'S' } ]**

**Create Coordinate Frame Vertex**
**<vertex_origin_id>**
**<vertex_zaxis_id>**
**<vertex_xzplane_id>**
**[tag { 'R' | 'C' | 'S' } ]**

Using the 'tag' option specifies the type of coordinate frame, i.e., rectangular (R), cylindrical (C) or spherical (S). The default coordinate frame type is rectangular. Exodus coordinate frames may also be listed and deleted using the commands below:

**List Coordinate Frame [ids] [ <frame_id>]**

**Delete Coordinate Frame [ids] [ <frame_id>| all]**

Any coordinate frames that exist at the time the exodus file is exported will be written out in the exodus file.
These coordinate frames can also be referenced when exporting an Abaqus file.

### Exporting GDF Files

The Geometric Data Format (GDF) is an export format used by WAMIT. The file format consists of quadrilateral and triangular elements.

**Export GDF '<filename>' [entity_list] [Block <id_list>] [ulen <value=1.0>] [gravity <value=9.80665>] [isx <value=0>] [isy <value=0>] [Overwrite]**

All elements to be exported should be part of a block. All blocks are exported unless otherwise specified in the command. The standard GDF parameters can be specified on export.

## exodus

### Element Block Specification

- Creating Blocks
- Assigning a Name or Description to an Element Block
- Defining the Element Type
- Node Constraints for High Order Elements
- Default Element Blocks
- Duplicate Block Elements
- Assigning Attributes
- Displaying Blocks

- Deleting Blocks
- Renumbering Element Blocks
- Automatically Assigning Mesh Edges to a Block (Rebar)
- Creating Spider Blocks
- Creating Beam Blocks
- Creating Spring Blocks
- Creating Sphere Blocks
- 2d Elements
- Mixed Element Output
- Adding Materials to a Block
- SUPERELEMENT_TOPOLOGY_XXX Support


Element blocks are the method CUBIT uses to group related sets of elements into a single entity. Each element in an element block must have the same basic and specific element type.
The preferred method for defining blocks is to use geometric entities such as volumes, surfaces or curves. Blocks can also be defined using mesh entities. If a block is defined at a geometric entity, each of the elements *owned* by the geometry are automatically assigned to the block. Deleting or remeshing the geometry automatically changes the set of elements grouped into the block. If mesh entities are used to specify a block, deleting the mesh will also delete the elements from the block.
Some important notes regarding Element Blocks are as follows:

- Multiple volumes, surfaces, and curves can be contained in a single element block
- A volume, surface, or curve can only be in one element block
- Element Block id's are arbitrary and user-defined. They do not need to be in any contiguous sequence of integers.
- Element Blocks can be assigned a single floating point number, referred to as the block Attribute; this number is used to represent the length or thickness of Bar and Shell elements, respectively. The attribute defaults to 1.0 if not specified.

*Creating Element Blocks*

Element blocks are defined with the following Block commands.

**Block <block_id> [ADD|Remove] {Vertex | Node} <range>**

**Block <block_id> [ADD|Remove] {Curve | Edge} <range>**

**Block <block_id> [ADD|Remove] {Surface | Face | Tri} <range>**

**Block <block_id> [ADD|Remove] {Volume | Hex | Tet | Pyramid | Wedge} <range>**

**Block <block_id> [ADD|Remove] Group <range>**

These commands define blocks based on a list of geometric or mesh entities. A block can only hold entities of the same dimensionality. For example, a block defined to hold vertices and nodes cannot also hold hexes. The above commands reflect this restriction. This restriction also applies when adding entities using groups. When creating a block using a group containing entities of different dimensionality the behavior is undefined.
Adding geometric entities to a block effectivily adds all mesh entities of the same dimensionality contained in the geometric entity to the block. For example, adding a volume to a block adds all hexes, tets, pyramids and wedges contained in the volume to the block. Removing geometry entities works in the same manner. Thus the following commands:

```
Block 1 add volume 1

Block 1 remove hex 1
```

Creates block 1 containing all of the hexes, tets, pyramids and wedges in volume 1 except for hex 1.
When a mesh entity, or a meshed geometric entity is put into a block, it is assigned a *Global Element ID* which is exported to the exodus file for tracking during analysis.

*Assigning a Name or Description to an Element Block*

The following commands can be used to assign a name or description to an element block. Assigning a name to a block can be more intuitive than using traditional integer IDs, and the name and description are preserved in DART metadata-enabled applications (like SIMBA). This command is also available for nodesets and sidesets.

```
Block<ids> Name "<new_name>"

Block<ids> Description "<description>"
```

*Defining the Element Type*

Each block must have a specific element type associated with it. To assign an element type to a block, use the following command:

```
Block <block_id_range> Element Type <type>
```

Available element types are defined by the Exodus II file format specification (Schoof, 95). CUBIT supports the following element types:
**Nodes**: SPHERE SPRING
**Curves**: BAR BAR2 BAR3 BEAM BEAM2 BEAM3 TRUSS TRUSS2 TRUSS3 SPRING
**Surfaces**: QUAD QUAD4 QUAD5 QUAD8 QUAD9 SHELL SHELL4 SHELL8 SHELL9 HEXSHELL TRI TRI3 TRI6 TRI7 TRISHELL TRISHELL3 TRISHELL6 TRISHELL7

**Volumes**: HEX HEX8 HEX9 HEX20 HEX27 TETRA TETRA4 TETRA8 TETRA10 TETRA14 TETRA15 PYRAMID PYRAMID5 PYRAMID13 PYRAMID18 WEDGE WEDGE6 WEDGE15 WEDGE16 WEDGE20 WEDGE21

If the element type is not assigned for an element block, it will be assigned a default type depending on which type of geometry entity is contained in the block. The default values used for element type are:

**Volume**: 8-node hexahedral elements (HEX8) will be generated for hex meshes. TETRA4 will be generated for tet meshes.
**Surface**: 4-node shell elements (SHELL4) will be generated for quad meshes and TRISHELL3 for tri meshes.
**Curve**: 2-node bar elements (BAR2) will be generated.
**Node**: 1-node elements (SPHERE) will be generated.

*Node Constraints for High Order Elements*

Higher order nodes are moved to curved geometry by default. To change this, use the following command:

**set Node Constraint {on|off|SMART [tet quality {distortion|NORMALIZED INRADIUS}][threshold <value=0.15>]]}**

**On** means higher order mid-nodes snap to curved geometry. **Off** means the mid-nodes retain their positions. **"smart"** means higher order mid-nodes will only snap to geometry if they do not cause quality problems after being moved. Nodes that cannot be moved without causing quality problems are placed at the average location of the element nodes: for edges, this means on the line containing the edge; for 2d elements, this usually means on the plane containing the element.
When the **smart** option is used, the **tet quality** and **threshold** options can also be used. **Tet quality** indicates the quality metric that will be used for determining whether mid-nodes will be projected or straightened. This option is currently only valid for high order tets (TETRA10) and tris (TRI6). **Normalized Inradius** or **Distortion** metrics may be selected as criteria for projections. The **threshold** value indicates the quality value at which mid-nodes will not be projected. For example, if **Normalized Inradius** falls below the threshold value, the element edge will be straightened. Those with metrics above the threshold will be projected.

*Default Element Blocks*

When exporting an ExodusII file, if the user has not specified any Element Blocks, by default element blocks will be written for any meshed volumes. This default behavior can be changed, to write surface, volume, or no meshes by default. This option can be set using the command

**Set Default Block [ON|off|Volume|Surface|Curve]**

Default behavior, **ON**, is for the blocks to automatically be written based on their owning geometry. When the **OFF** setting is used, only the mesh contained in blocks created by the user will be exported. Mesh not in an element block at export time, will not be exported. The export will still succeed and no error will be thrown. If **Volume** is specified, only elements contained in volumes will have default blocks specified. Similarly, the **Surface** or **Curve** argument indicates that only surfaces or curves containing elements will use default blocks, respectively.
When default blocks are used, the IDs for the resulting blocks will be the ID of the owning geometry.

*Duplicate Block Elements*

By default, any given element cannot be included in more than one block. However, when using the following command, an element may be included in more than one block. Please note, since material properties are assigned to blocks, using this command to allow duplicate block elements may result in an element being assigned to multiple materials.

**Set Duplicate Block Elements {on|OFF}**

Cubit stores only a single Global Element ID (GID) for each element. If an element is placed into more than one block, when the model is exported to Exodus, new additional GIDs will be assigned to the element for each additional block that an element is in. These additional GIDs are exported to the exodus file, but Cubit currently only stores and tracks the first GID assigned.

*Assigning Attributes to Blocks*

It may be necessary to associate attributes with a specific element block. Attributes are generally integer or floating point values that represent some physical property in the region occupied by the block, such as material properties or shell thickness. To assign the number of attributes for an element block, use the following command:

**Block <id_range> Attribute Count <0-20>**

CUBIT will store up to 20 attributes per block. Specify the maximum number of attributes to be stored on the block with this command. Once this command has been executed, individual attributes may be set using the following command:

**Block <id_range> Attribute Index <index> <value>**

The index is an integer from 1 to the maximum count specified in the Block Attribute Count command. The value may be any valid floating point number.
To assign a value to all attributes of an element block, use the command:

**Block <block_id_range> Attribute <value>**

*Displaying Element Blocks*

Blocks can be viewed individually with CUBIT by employing the following command:

**Draw Block <block_id_range> [Color <color_spec>] [add] [thickness [offset [scale <val>] | include_normal]]**

For blocks that are of type SHELL and TRISHELL or one of its variants including the [thickness] keyword and parameters will result in the blocks being color-coded by shell thickness with a corresponding color bar. Blocks can be drawn with their specified thickness, so they visually have a thickness. This thickness can also be scaled in the draw command. Arrows defining the shell normal direction will be displayed as well as a legend showing the thickness values.
Block colors can also be changed using the following command:

**Color Block <block_id_range> {color|Default}**

*Deleting Element Blocks*

All Nodesets, Sidesets and Blocks may be deleted from the model using the following command:

**Reset Genesis**

To remove only Blocks, the following may be used:

**Reset Block**

To remove a specific block, use:

**Delete Block <block_id_range>**

*Renumbering Element Blocks*

The block renumber command gives the user the ability to renumber blocks to fit the user's needs. The command is:

**Block <id_range> renumber start_id <id> [uniqueids]**

The **id_range** must include existing entities or the command will fail.
The **start_id** plus the number of entities must specify a new id space that does not overlap with the existing block ids. In other words, if the current block numbers are 100, 105, 106, and 109, a **start_id** of 102 would suggest new block numbers of 102, 103, 104, and 105. This would cause an id space conflict and the command will fail.
If the user specifies the **uniqueids** option, then the new entity id space must not conflict with the existing id space of all blocks, nodesets, and sidesets.

Example:
Assume:
block ids: 100, 105, 106, 109

**block all renumber start 20**
**block 20 renumber start 24**
After commands:
block ids: 21, 22, 23, 24
To renumber the elements within a block, see the renumber command

*Automatically Assigning Mesh Edges to a Block (Rebar)*

After a mesh has been defined within a volume, it may be useful to use the existing mesh edges as the basis for an element block. Such an element block might be composed of bars or truss type elements that might propagate through a solid medium such as rebar placed in reinforced concrete. Although the **Block <id> Edge <range>** command could be used for this task, it would prove extremely tedious defining the individual edges to add to the block. To make this process easier, the following command can be used:

**Rebar Start <x> <y> <z> Direction <x> <y> <z> [Length <value>] Block <id> [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss|truss2|truss3}]**

The **Rebar** command allows the user to specify a starting location for a set of edges and an initial direction. The program will find the closest existing node in the mesh to **Start <x> <y> <z>** and begin propagating through the mesh in the specified **Direction <x> <y> <z>**, adding edges to the block as it propagates through the mesh. The edge that is attached to the last node and is within a fixed 30 degrees of the specified direction is added to the block. The Propagation of the edges continues until either the optional **Length** value is reached or an edge does not meet the **Direction** criteria. Also required with this command is a **block** ID. An **Element Type** can also be specified.
Similarly, you can use the following command which will use the 30 degree cone described above to gather edges from a surface into a single block using the Cartesian x, y, and/or z vectors.

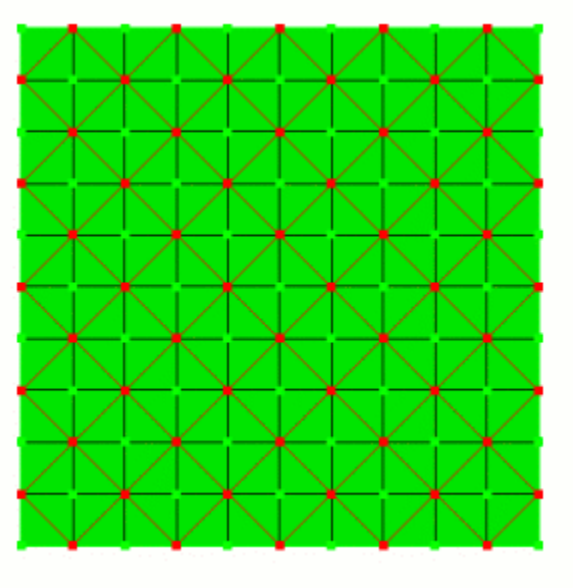
**Rebar Surface <range> [x] [y] [z] Block <id> [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss|truss2|truss3}] [Propagate]**

**Diagonal and Orthogonal Rebar Blocks**

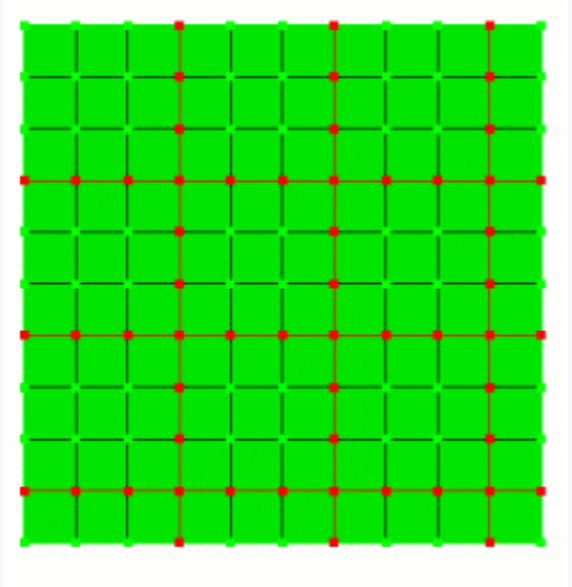
Another method for generating rebar blocks include the Diagonal/Orthogonal option. This command can only be used on surfaces that have been meshed with the mapping scheme. This command will create a block of edges from the mapped mesh by starting in one corner and gathering edges orthogonally, or creating new edges diagonally based on the option specified, using the parametric coordinate system dictated by the mapping

scheme on the surface. The spacing option dictates how many edges are skipped over before starting the next set of rebar edges.

**Rebar Surface <range> {Diagonal|Orthogonal} [Spacing <int>] [Block <id> [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss}]**



**CUBIT> rebar surf 1 diagonal spacing 2 block 2**

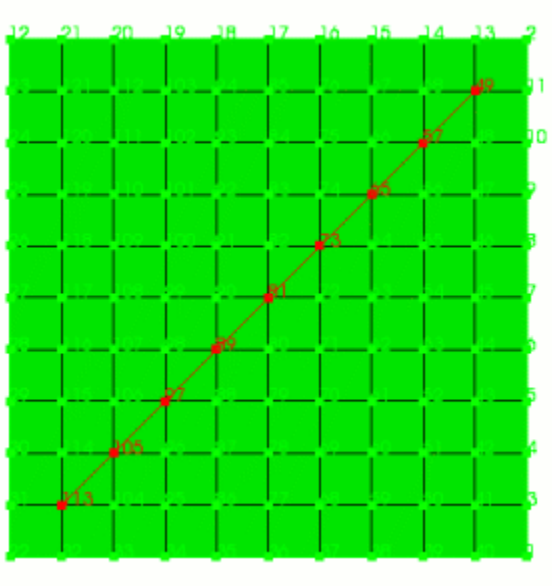


**CUBIT> rebar surf 1 orthogonal spacing 3 block 3**

**Specifying a set of nodes**

A final rebar option allows the user to create or group rebar edges into a specified block using nodes. Edges are created, or gathered, using the ordered list of nodes specified in the command.

**Rebar Node <range> [Target Block <id>] [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss}]**



**CUBIT> rebar node 113 105 97 89 81 73 65 57 49 target block 1**

A related command for creating curve geometry directly from mesh edges is the Create Curve from Mesh command. See Curve creation for more details.

*Creating Spider Blocks*

The block creation tool also allows the user to create a special block of bar elements that can be used as part of the boundary specification. This command creates beam type elements directly without creating any underlying geometry.
The command for creating this type of block is:

**Block <id> Joint [Vertex <id> | Node <id> ] Spider {Surface|Curve|Vertex|Face|Tri|Node} <range> [preview] [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss|truss2|truss3}]**

The **joint node** is the starting location of the bar elements and the **spider** location is the terminating location of the bar elements. You can specify the joint node as either a node or a vertex. Optionally, if no joint node is specified, a joint node will automatically be created at the centroid of the nodes on the specified terminating location. You can specify the terminating location as either a node, vertex, geometric surface or the face of a mesh entity.

Some analysis codes refer to these bar elements as tied contacts or rigid bar elements. They can be used to tie models together or to enforce specific kinds of boundary conditions. For example, in the figure below a block of beam elements is used to tie a node at the center of the circle to every node on the edge of the circle. This arrangement can be used to enforce circularity but still allow for displacement of the entire circle. This may occur if there are additional structures above the cylinder that are being excluded from the current finite element model. The beam elements were created by a series of commands of the form

**block 10 joint node 1 spider node 2**

The **preview** option can be included to draw the location of the beam blocks on the screen without actually executing the command.

If geometry (surfaces, curves, or vertices) is specified to define the spider, the spider will be 'tied' to that geometry, meaning:

- if the geometry is meshed, the edges will be created
- if the geometry is deleted, the mesh will be deleted
- if the geometry is translated, the edges will be translated.

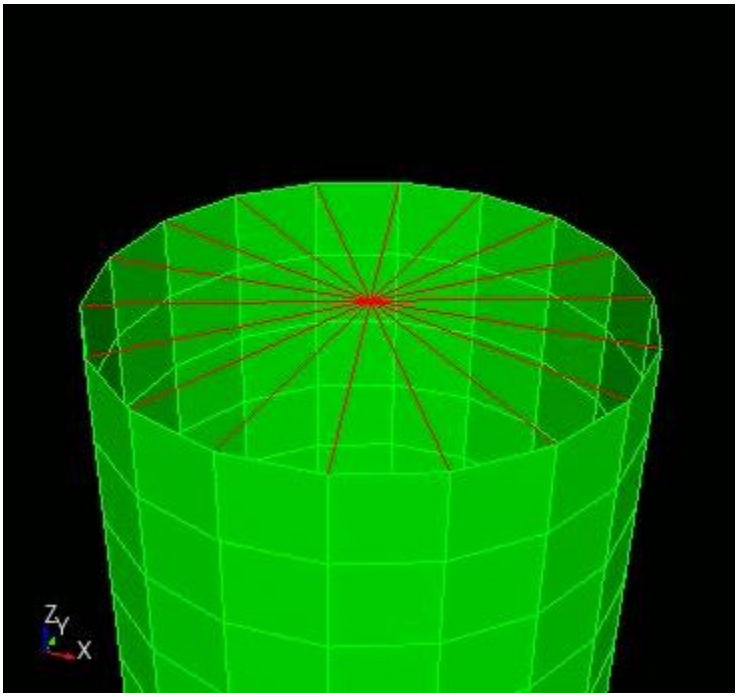



**Figure 1. Beam elements created with the Spider command**

*Creating Beam Blocks*

Properties for blocks that are beam types (beam, beam2, beam3) have additional commands to define a cross-sectional area. The following command can be used to change the type of cross-sectional area of a beam block:

**Block <id> beam_type {CIRCLE|box|rectangle|pipe|ibeam|general}**

The dimensions are set by listing them after the keyword **beam_dimensions**:

**Block <id> beam_dimensions <values>**

The order in which the values need to be specified are described in the chart below.
If the solver used is to integrate over the section during the simulation, turn **section_integration** on using the following command:

**Block <id> section_integration {ON|off}**

The beam normal vector is a vector normal to the plane of motion and tangent to the first bending axis. This vector can be set using the following command:

**Block <id> beam_normal <x><y><z>**

| Section Profile | Order to Specify Dimensions |
|---|---|
| Circle | Radius |
| Pipe | Outer radius, wall thickness |
| Rectangle | Width, height |
| Box | Total width, total height, thickness (right), thickness (top), thickness (left), thickness (bottom) |
| I-Beam | Distance to bending axis (from bottom), total height, bottom width, top width, thickness (bottom), thickness (top), thickness (web) |
| General | Area, Ixx, Ixy, Iyy, Polar moment of inertia (J) |

*Creating Spring Blocks*

Spring blocks that will be exported to Abaqus can contain additional properties related to Abaqus springs. Users can specify the spring type, stiffness, and DOFs associated with Abaqus springs. The spring type mapping to Abaqus elements is in the following table.

| CUBIT Block Spring Type | Abaqus Element Type |
|---|---|
| Node_to_node | SPRINGA |
| Node_to_node | SPRING1 |
| Node_to_ground_fixed | SPRING2 |

The spring type is set using the **spring_type** keyword. In order to use this command, the block must already have an element type of “SPRING.” If a DOF is associated with a spring, the **spring_dof_1** keyword is used to specify the DOF on the first node and **spring_dof_2** is used to specify the DOF on the second node (SPRING2 only).

**Block <id> [spring_type {NODE_TO_NODE | node_to_node_fixed_axis | node_to_ground}] [stiffness <k>] [spring_dof_1 <n>] [spring_dof_2 <n>]**

*Creating Sphere Blocks*

Sphere elements are created in CUBIT by inserting either nodes or vertices into a block.

**Block <id> {node|vertex} <id_range>**

The command above causes CUBIT to internally create a sphere element and associate it to the inserted node, or to the node associated to the inserted vertex.

```
Example:
brick x 10
vol all size 5
mesh vol all
create vertex 0 0 10
#{sph_vtx_id=Id("vertex")}
mesh vertex {sph_vtx_id}
#{sph_nd=Id("node")}
block 1 volume 1
block 2 vertex {sph_vtx_id}
block 3 joint node {sph_nd} spider surf 1
locate sphere all
```

The example commands above will generate the model illustrated in the figure below.

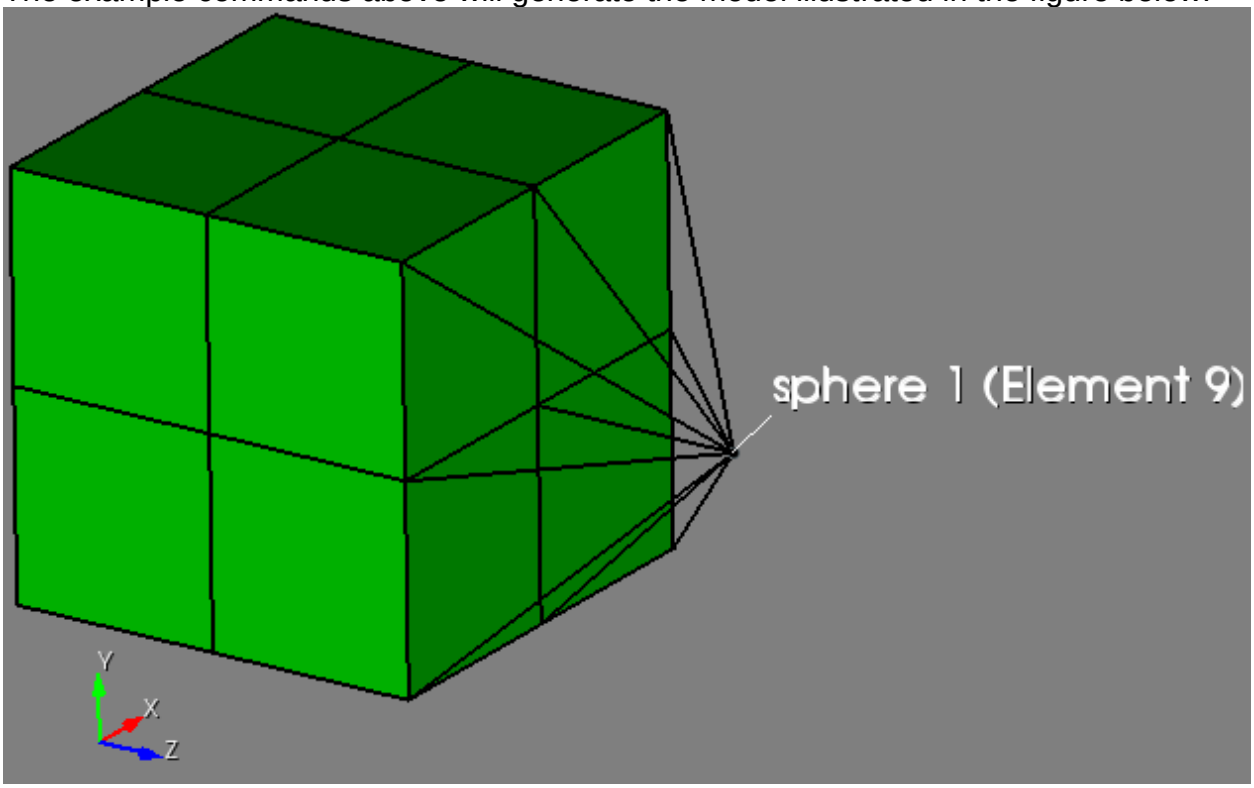



**Figure 2. A sphere element created and connected to a solid mesh with 2d elements.**

You can interact with sphere elements in Cubit with the commands below:

**locate sphere <id_range>**

**draw sphere <id_range>**

**highlight sphere <id_range>**

**list sphere ids**

**list sphere <id_range>**

*2D Elements*

CUBIT is a 3d mesh generator by default. Element types, by default, are respectively TRISHELL and SHELL for triangle and quad elements. If a 2d mesh is desired, blocks types must be explicitly set to TRI or QUAD.

```
Example:
create brick x 10
surface 1 scheme trimesh
mesh surface 1
block 1 surface 1
block 1 element type tri
export mesh "mymesh.exo"
```

**Sideset 1** will be based on the TRI and QUAD elements in **blocks 1 and 2**, with the side numbering referring to the edges of the triangles and quads.

*Mixed Element Output*

The **Set Block Mixed Output** command controls the behavior of blocks containing different element types when exporting in a file format that doesn't support blocks with mixed element types. If DEGENERATE, all elements will be exported in one block, but tets and pyramids will be written as degenerate hexes, and triangles will be written as degenerate quads. If OFFSET (set by default), then new element blocks will be created separating the types. Hex and Quad blocks retain the block id, whereas tets, triangle, pyramids and wedges get put into other blocks. The ids of the other blocks are based on the block id plus the offset for that type. Those values are set using the offset commands.

```
Set Block Mixed Element Output { OFFSET | Degenerate }

Set Block Triangle Offset <value>

Set Block Tetrahedron Offset <value>

Set Block Pyramid Offset <value>
```

*Adding Materials to a Block*

```
Block <id> Material <id|'name'>
```

If a material is assigned to an element block, the material properties will be associated with the block's elements when the mesh is exported. If no material is assigned to a block, a default material will be used during export.

*SUPERELEMENT_TOPOLOGY_XXX Support*

Cubit has basic support for Superelements. Importing mesh with Superelements is supported using either the lite, no_geom or geometry options. When using the geometry option to construct Mesh-Based geometry, superelements do not have geometry created, but continue to exist as free elements. Superelements are visually represented as a point cloud since there is no connecting topology. Cubit does not have the ability to create these elements. Importing and exporting these elements is supported.

## Exodus II File Specification

*Exodus II Manual*

The full Exodus II manual is available from the web.

*Element Block Definition Examples*

#### Multiple Element Blocks

Multiple element blocks are often used when generating a finite element mesh. For example, if the finite element model consists of a block which has a thin shell encasing the volume mesh, the following block commands would be used:

```
Block 100 Volume 1
Block 100 Element Type Hex8
Block 200 Surface 1 To 6
Block 200 Element Type Shell4
Block 200 Attribute 0.01
Mesh Volume 1
Export Genesis `block.g'
```

This sequence of commands defines two element blocks (100 and 200). Element block 100 is composed of 8-node hexahedral elements and element block 200 is composed of 4-node shell elements on the surface of the block. The "thickness" of the shell elements is 0.01. The finite element code which reads the Genesis file (block.g) would refer to these blocks using the element block IDs 100 and 200. Note that the second line and the fourth line of the example are not required since both commands represent the default element type for the respective element blocks.

#### Surface Mesh Only

If a mesh containing only the surface of the block is desired, the first two lines of the example would be omitted and the Mesh Volume 1 line would be changed to, for example **Mesh Surface 1 To 6**.

#### Two-dimensional Mesh

CUBIT also provides the capability of writing two-dimensional Genesis databases similar to FASTQ. The user must first assign the appropriate surfaces in the model to an element block. Then a Quad* type element may be specified for the element block. For example

```
Block 1 Surface 1 To 4
Block 1 Element Type Quad4
```

In this case, it is important for users to note that a two-dimensional Genesis database will result. In writing a two-dimensional Genesis database, CUBIT ignores all z-coordinate data. Therefore, the user must ensure that the Element Block is assigned to a planar surface lying in a plane parallel to the x-y plane. Currently, the Quad* element types are the only supported two-dimensional elements. Two-dimensional shell elements will be added in the near future if required.

## Exodus II Model Title

CUBIT will automatically generate a default title for the Genesis database. The default title has the form:

```
cubit(genesis_filename): date: time
```

The title can be changed using the command:

```
Title '<title_string>'
```

## Defining Materials and Media Types

Materials can be defined in CUBIT and assigned to element blocks. If an element block is exported without a material assigned to it, a default material (with properties for common steel) will be exported for it.

```
Create Material [id] [Name <'name'>] [Elastic_modulus <value>]
[Poisson_ratio <value>] [Shear_modulus <value>] [Density <value>]
[Specific_heat <value>] [Conductivity <value>] [User constants <value ...>]
[DepVar <value>]

Modify Material <id_list|'name'|all> [Name <'name'>] [Elastic_modulus
<value>] [Poisson_ratio <value>] [Shear_modulus <value>] [Density <value>]
[Specific_heat <value>] [Conductivity <value>] [User constants <value ...>]
[DepVar <value>]

Create Media [id] [Name <'name'>] [Fluid|Porous|Solid]

Modify Media <id_list|'name'|all> [Name <'name'>] [Fluid|Porous|Solid]
```

Materials can be created with any number of the following material properties:

- Elastic modulus
- Poisson Ratio
- Density
- Specific Heat
- Conductivity
- Shear Modulus (must satisfy E = 2G(1+v) )
- User Constants
- DepVar (Only written to Abaqus file)

Media types include:

- Fluid
- Porous
- Solid

Any properties that are not initialized by the user will have a default value of 0.
Materials and media types can be listed and deleted using the following commands:

**List Material <id_list|'name'|all>**

**Delete material <id_list|'name'|all>**

**List Media <id_list|'name'|all>**

**Delete Media <id_list|'name'|all>**

Materials and media can be added to an existing block using the following command:

**Block <id> Material <id|'name'>**

**Block <id> Media <id|'name'>**

*Custom Material Commands*

The Cubit SDK allows custom material properties to be defined using the MaterialInterface. The following versions of the material commands allow users to create materials with custom properties that have already been defined using the SDK.

**Create {material|media} 'material_name' property_group 'group_name' [id <requested_id>] [description 'string']**

**Modify {material|media} 'material_name' [property_group 'group_name'] [id <requested_id>] [description 'string'] [rename 'new_name'] [scalar_properties ('property_name' <property_value>)...]**

**[vector_property 'property_name' <val1> <val2>...]**
**[matrix_property 'property_name' <val1> <val2>...]**
**[clear properties 'property_name1' 'property_name2'...]**

A scalar property has a single value associated with it. The scalar properties defined by Cubit are:

- "MODULUS" (The modulus of elasticity)
- "SHEAR_MODULUS"
- "POISSON" (Poisson's ratio)
- "DENSITY"
- "SPECIFIC_HEAT"
- "CONDUCTIVITY" (Thermal conductivity, not electrical)
- "THERMAL_EXPANSION"
- "YIELD_STRENGTH"
- "ULTIMATE_STRENGTH"
- "ULTIMATE_STRAIN"
- "DEPVAR" (Property required for Abaqus export)
- "CFD_MEDIA_TYPE" (0 = fluid; 1 = porous; 2 = solid)

Vector properties are given in the command as a list of values. The vector properties defined in Cubit are:

- "USER_CONSTANTS" (User-defined material constants)

Matrix properties are also given as a list of values. The number of columns in the matrix is defined by the specific property, and Cubit automatically divides the given values into rows and columns based on the column count. For example, if a matrix property has 2 columns, the value list "2 33.1 3 18.9" is interpreted as the matrix:

| | |
|---|---|
| 2 | 33.1 |
| 3 | 18.9 |

Cubit defines the following matrix properties:

- "ELASTIC_MODULUS_VS_TEMPERATURE" (2 columns)
- "POISSONS_RATIO_VS_TEMPERATURE" (2 columns)
- "DENSITY_VS_TEMPERATURE" (2 columns)
- "YIELD_STRESS_VS_STRAIN_VS_TEMPERATURE" (3 columns)
- "SPECIFIC_HEAT_VS_TEMPERATURE" (2 columns)
- "CONDUCTIVITY_VS_TEMPERATURE" (2 columns)

A property group is a collection of material properties. Its main purpose is to help define what properties a material should have, even if a value is not given for the property. Cubit defines the following property groups:

- "CUBIT-FEA" (generic FEA material)
- "CUBIT-ABAQUS" (material specific to Abaqus export)

- "CUBIT-CFD" (generic CFD media)

**Table 1. Property groups defined in CUBIT**

| Property | "CUBIT-FEA" | "CUBIT-ABAQUS" | "CUBIT-CFD" |
|---|---|---|---|
| "CFD_MEDIA_TYPE" | | | x |
| "MODULUS" | x | x | |
| "SHEAR_MODULUS" | x | x | |
| "POISSON" | x | x | |
| "DENSITY" | x | x | |
| "SPECIFIC_HEAT" | x | x | |
| "CONDUCTIVITY" | x | x | |
| "THERMAL_EXPANSION" | x | x | |
| "YIELD_STRENGTH" | x | x | |
| "ULTIMATE_STRENGTH" | x | x | |
| "ULTIMATE_STRAIN" | x | x | |
| "DEPVAR" | | x | |
| "USER_CONSTANTS" | x | x | |
| "ELASTIC_MODULUS_VS_TEMPERATURE" | | x | |
| "POISSONS_RATIO_VS_TEMPERATURE" | | x | |
| "DENSITY_VS_TEMPERATURE" | | x | |
| "YIELD_STRESS_VS_STRAIN_VS_TEMPERATURE" | | x | |
| "SPECIFIC_HEAT_VS_TEMPERATURE" | | x | |
| "CONDUCTIVITY_VS_TEMPERATURE" | | x | |

## Exodus Boundary Conditions

Sandia's finite element analysis codes have been written to transfer mesh definition data in the ExodusII file format (citation Schoof, 95). The ExodusII database exported during a CUBIT session is sometimes referred to as a Genesis database file; this term is used to refer to a subset of an Exodus file containing the problem definition only, i.e., no analysis results are included in the database.

The ExodusII database contains mechanisms for grouping elements into Element Blocks, which are used to define material types of elements. ExodusII also allows the definition of groups of nodes and element sides in Nodesets and Sidesets, respectively; these are useful for defining boundary and initial conditions. Using Element Blocks, Nodesets and Sidesets allows the grouping of elements, nodes and sides for use in defining boundary conditions, without storing analysis code-specific boundary condition types. This allows CUBIT to generate meshes for many different types of finite element codes.

*Element Blocks*

Element Blocks (also referred to as simply, Blocks) are a logical grouping of elements all having the same basic geometry and number of nodes. All elements within an Element Block are required to have the same element type. Access to an Element Block is accomplished through a user-specified integer Block ID. Typically, Element Blocks can also be assigned material properties to associate material properties with a group of elements.

*Nodesets*

Nodesets are a logical grouping of nodes accessed through a user-specified Nodeset ID. Nodesets provide a means to reference a group of nodes with a single ID. They are typically used to specify load or boundary conditions on portions of the CUBIT model or to identify a group of nodes for a special output request in the finite element analysis code.

*Sidesets*

Sidesets are another mechanism by which constraints may be applied to the model. Sidesets represent a grouping of element sides and are also referenced using an integer Sideset ID. They are typically used in situations where a constraint must be associated with element sides to satisfactorily represent the physics (for example, a contact surface or a pressure.

*Element Types*

The basic elements used to discretize geometry were described in the mesh generation chapter. Within each basic element type, several specific element types are available. These specific element types vary by the number of nodes used to define the element, and result in different orders of accuracy of the element. The element types available for each basic element type defined in CUBIT are summarized in the following table.

**Table 1. Element Types Defined in CUBIT**

| Basic Element Type | Specific Element Type | Notes |
|---|---|---|
| Node | SPHERE | |
| Edge | BAR, BAR2, BAR3, BEAM, BEAM2, BEAM3, TRUSS, TRUSS2, TRUSS3, SPRING | By default, Bars have 2 DOF's per node; Beams, trusses and springs have 3 |
| Triangle | TRI, TRI3, TRI6, TRI7, TRISHELL, TRISHELL3, TRISHELL6, TRISHELL7 | Tri element nodal coordinates |

| | | |
|---|---|---|
| | | are always 3D. |
| Quadrilateral | QUAD, QUAD4, QUAD8, QUAD9; SHELL, SHELL4, SHELL8, SHELL9, HEXSHELL | By default, quad element nodal coordinates are 2D, i.e., only x and y coordinates. Shell element nodal coordinates are 3D. |
| Tetrahedron | TETRA, TETRA4, TETRA8, TETRA10, TETRA14, TETRA15 | TETRA8 contains vertex nodes and mid-face nodes. |
| Hexahedron | HEX, HEX8, HEX9, HEX20, HEX27 | |
| Pyramid | PYRAMID, PYRAMID5, PYRAMID8, PYRAMID13, PYRAMID18 | A PYRAMID8 is output as a degenerate HEX. |
| Wedge | WEDGE, WEDGE6, WEDGE15, WEDGE16, WEDGE20, WEDGE21 | |
| Superelement | SUPERELEMENT_TOPOLOGY_XXX | An element composed of an variable number of nodes. |

For a description of the node and side numbering conventions for each specific element type, see the Appendix. Element types can be set for individual Element Blocks, either before or after meshing has been performed.

## Nodeset and Sideset Specification

- Creating Nodesets and Sidesets
- Assigning Names and Descriptions to Nodesets and Sidesets
- Grouping Faces on a Surface into a Sideset
- Deleting Nodesets and Sidesets
- Renumbering Nodesets and Sidesets
- Transfer Nodesets and Sidesets
- Displaying Nodesets and Sidesets
- Nodeset Associativity Data

- Equation-Controlled Distribution Factors
- Nodesets/Sidesets/Blocks Behavior with Geometric Entity Copy

Boundary conditions such as constraints and loads are applied to the finite element model using *nodesets or sidesets*, also known as Genesis entities. Rather than attempting to maintain specific boundary condition information, such as load, temperature, constraint, etc., Genesis entities are the generic vehicle for the user to set up boundary conditions on the model. Nodes, elements and element faces are instead grouped together and assigned unique IDs. Node, element and face IDs assigned to Genesis entities can then be written to the Exodus II mesh file. Once imported to the intended analysis application, the nodeset and sideset IDs can be appropriately interpreted as specific physical boundary conditions.
The preferred method for creating Genesis entities is to assign vertices, curves, surfaces or volumes to a specific nodeset or sideset ID. Any mesh entity *owned* by the geometric entity in a nodeset or sideset is automatically assigned to the same nodeset or sideset. This allows greatest flexibility in generating and updating the finite element mesh. For example, if a surface belongs to a specific sideset, remeshing the surface will automatically delete any old faces from the sideset and add the faces of the new mesh.
In some cases, the geometric model does not provide enough resolution to define the desired boundary conditions. In this case, the model may be partitioned using CUBIT's virtual geometry features. Where this may not be feasible, mesh entities can also be added directly to the desired nodeset or sideset. Where individual mesh entities have been added to nodesets or sidesets, deleting the mesh will also remove these elements from the Genesis entity. If the geometry is remeshed, the new mesh entities must also be added once again to the nodesets or sidesets.
Nodesets can be created from groups of nodes categorized by their owning volumes, surfaces, curves or vertex. Individual nodes may also be added to a nodeset. Nodes can belong to more than one nodeset.
Sidesets can be created from groups of element sides or faces categorized by their owning surfaces or curves or by their individual face IDs. Element sides and faces can also belong to more than one sideset.

*Creating Nodesets and Sidesets*

Nodesets and Sidesets are created in CUBIT by assigning the appropriate geometry or mesh entities in the model to a nodeset or sideset ID. The following commands can be used:

**Nodeset <nodeset_id> [ADD|Remove] {Curve | Surface | Volume | Vertex | Node} <range>**

**Sideset <sideset_id> [ADD|Remove] Group <id_range>**

**Sideset <sideset_id> [ADD|Remove] {Curve|Surface|Edge|Face|Tri} <id_range>**

**Sideset <sideset_id> [Add] Edge <id_range> [wrt {{Tri|Face} <id_range> | all } ]**

**Sideset <sideset_id> [Add] Face <id_range> [wrt {Hex <id_range> | all} ]**

**Sideset <sideset_id> [Add] Tri <id_range> [wrt {Tet <id_range> | all} ]**

**Sideset <sideset_id> [Add] Surface <id_range> [wrt {{Volume|Surface} <id_range> | all} ] [FORWARD|Reverse|Both]**

**Sideset <sideset_id> [Add] Curve <id_range> [wrt {Surface <id_range> | all} ]**

Like element blocks, Nodesets and Sidesets are given arbitrary, user-defined ID numbers. If there are no user-defined Nodesets or Sidesets, none are written to the Exodus II file.
With Sidesets, direction is often important. For surfaces, the direction may be specified using the **Forward**, **Reverse**, or **Both** options. The **Forward** option will write a sideset in relation to hexes in the surface's forward volume, which is the volume that the surface's normal points away from. The **Reverse** option will write a sideset in relation to hexes in the surface's reverse volume, which is the volume that the surface's normal points into. The **Both** option will allow sidesets to be written in relation to the hexes that lie in volumes on both sides of the surface. The default is **Forward**. The user can additionally specify the volume from which the hexes should be taken in relation to by using the **wrt Volume** option.
Direction is equally important for curves in Sidesets. The **wrt Surface** option allows the user to indicate which surface's faces will be included in the Sideset. The **wrt All** option will include all faces attached to the curve. The default is wrt All.

**Useful hint:**

When creating nodesets and sidesets it is often userful to use the Extended Command Line Entity Specification. Here is an example that creates a nodeset which includes all the nodes on the exterior of the geometry:

```
# Create the geometry
Create brick x 10
Create cylinder height 10 radius 2
Move volume 2 z 10
# Merge the geometry
Merge volume all
# Mesh the geometry
Mesh volume all
# Create a nodeset that includes only those nodes
# located on the exterior of the geometry
Nodeset 1 add surface in volume all with not is_merged
```

The following commands remove nodes from the nodeset that belong to a surface. Continuing from the previous example:

```
# Remove surface 2 from the nodeset
Nodeset 1 remove surface 2
# Remove nodes from the nodeset
# that belong to the curves that bound surface 2
Nodeset 1 remove node in curve in surface 2
```

Nodes can also be added or removed based upon their coordinates. Here is an example that removes all the nodes with a z coordinate equal to 15. Continuing from the previous example:

```
# Remove the nodes with a z coordinate equal to 15
Nodeset 1 remove node in surface all with z_coord = 15
```

*Assigning Names and Descriptions to Nodesets and Sidesets*

Nodesets and sidesets can be assigned names and descriptions. Using names and descriptions is often more intuitive than using traditional integer IDs. When exporting a mesh as a DART artifact, names and descriptions are included in the metadata, making them available to DART metadata-enabled applications such as SIMBA. To give a name or description to nodeset or sideset, use one of the following commands:

```
{Nodeset|Sideset} <ids> Name "<new_name>"

{Nodeset|Sideset} <ids> Description "<description>"
```

This command can also be used to define names and descriptions for Element Blocks.

*Grouping Faces on a Surface into a Sideset*

A sideset can be created from a subset of the faces on a given surface by using one of the following commands:

```
SideSet <sideset_id> Surface <id_range> Patch Maximum <x> <y> <z>
Minimum <x> <y> <z>

SideSet <sideset_id> Surface <id_range> Patch Center <x> <y> <z> Radius
<value> [Filter] [Partition]

SideSet <sideset_id> Surface <id_range> Patch Center <x> <y> <z>
Outer_radius <value> Inner_radius <value> [Filter] [Partition]
```

**SideSet <sideset_id> Surface <id_range> Patch Cylinder <axis_specification> Radius <rad> [Filter] [Partition]**

**SideSet <sideset_id> Surface <id_range> Patch Cylinder <axis_specification> Outer_radius <rad> Inner_radius <rad> [Filter] [Partition]**

These commands place only the faces meeting the specified criteria into the sideset.

- Using the **maximum** and **minimum** options will include all faces on the surface whose centroid falls within the axis-aligned box defined by the maximum and minimum points.
- Using the **center** and **radius** options will include all faces on the surface whose centroid falls within the sphere defined by center and radius.
- Using the **center**, **outer_radius**, and **inner_radius** options will include all faces on the surface whose centroid falls within the sphere defined by center and outer_radius, but excluding those faces whose centroid falls within the sphere defined by center and inner_radius. In other words, a face will be included if the distance between the face and the center point is between inner_radius and outer_radius.
- Using the **cylinder** option will include all faces whose centroid falls within a cylinder of infinite length with the given axis and radius. The axis is specified as described in Specifying an Axis.
- Using the optional inner_radius will exclude those faces whose centroid is closer to the axis than the specified inner_radius.

Normally, these commands place the individual elements into the sideset. If the mesh on the surface is deleted, the elements will be removed from the sideset. If the surface is then remeshed, new elements will NOT automatically be added to the sideset. This is usually the intended behavior.
If the **filter** option is included, only a single connected set of elements is added to the sideset. If the shape of the surface is such that multiple disconnected sets of elements fall within the specified spherical or cylindrical region, the filter option will limit the faces added to the sideset to the one set closest to center.
Using the **partition** option changes this behavior. The partition option causes the surface to be split, based on the faces included in the patch. The newly created patch surface will be added to the sideset instead of the individual elements. If the mesh is deleted and a new mesh is generated, the new mesh on the patch surface will automatically be included in the sideset, just as occurs with other geometric entities assigned to sidesets.
Note that the sideset patch commands work with both triangular and quadrilateral faces.

#### Grouping elements in voids and enclosures

The **sideset start enclosure** command creates sidesets of monotonically increasing ID numbers containing the elements comprising the watertight skin of the input elements. When there's a 'void' in the middle of the elements, a region devoid of elements, though

still enclosed by elements, this enclosed region will also have a sideset defined on the skin of the enclosed region.

**Sideset Start <id> Enclosure {Volume|Hex|Tet} <range>**

The start id is the id of the sideset at which to start. The ID numbers will increase monotonically unless there is a conflicting ID number. The command will add as many sidesets as there are fully continuous regions or tris or faces in the input group. This function can be particularly helpful for calculations for radiation enclosures.

*Deleting Nodesets and Sidesets*

All Nodesets, Sidesets and Blocks may be deleted from the model using the following command:

**Reset Genesis**

To remove only nodesets or sidesets, the following may be used:

**Reset Nodeset**

**Reset Sideset**

To remove a specific nodeset or sideset, use:

**Delete Nodeset <nodeset_id_range>**

**Delete Sideset <sideset_id_range>**

*Renumbering Nodesets and Sidesets*

The nodeset and sideset renumber commands give the user the ability to renumber these entities to fit the user's needs. The command is:

**{Nodeset|Sideset} <id_range> renumber start_id <id> [uniqueids]**

Example:
Assume:
sideset ids: 1, 2, 4, 6, 10

**sideset all renumber start 30**
After renumbering:
sideset ids: 30, 31, 32, 33, 34

The **id_range** must specify existing nodesets or sidesets, respectively, or the command will fail.
The new ids to be assigned cannot contain the id of an existing nodeset (when renumbering nodesets), or an existing sideset (when renumbering sidesets).
For example, given sidesets with ids 100, 105, 106, and 109, the command

**sideset all renumber start_id 102**

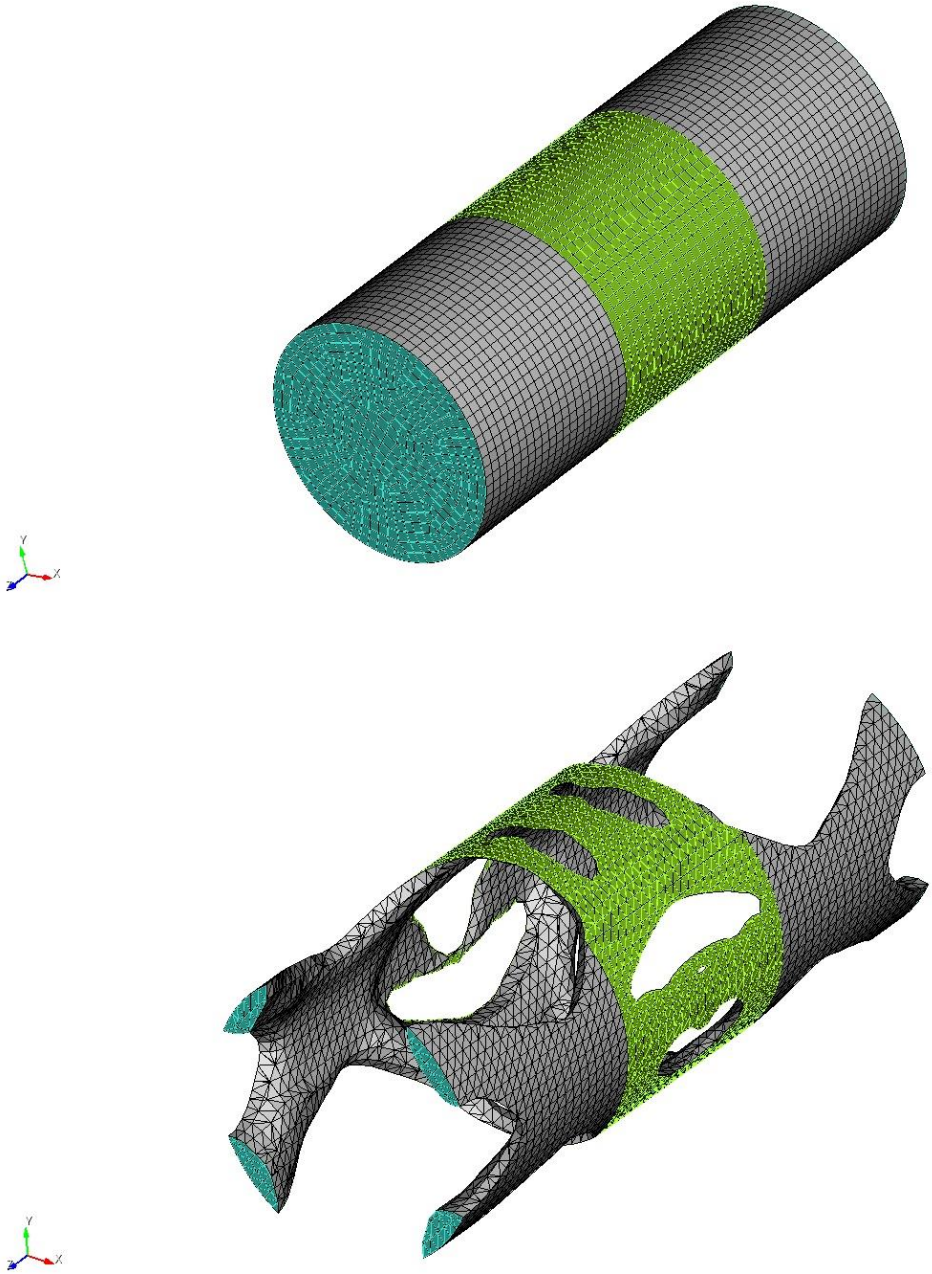
would attempt to renumber the sideset ids to 102, 103, 104, and 105. Since sideset 105 already exists, the command will fail.
When the **uniqueids** option is specified, the new ids to be assigned cannot contain the id of an existing nodeset OR an existing sideset OR an existing block. For example, given sidesets with ids 100, 105, 106, and 109, and given blocks with ids 201, 202, and 203, the command

**sideset all renumber start_id 200 uniqueids**

would attempt to renumber the sideset ids to 200, 201, 202, and 203. While this does not conflict with existing sideset ids, it does conflict with the existing block ids and so the command will fail.

*Tranfer Nodesets and Sidesets*

The capability shown in the below figure automates the transfer of sidesets and nodesets from the source mesh (e.g. design domain model) to the target mesh (e.g. topology optimized shape). Transfer of sidesets and nodesets was a user-intensive, error-prone, bottleneck as users were required to go through the painful step of manually selecting elements to define equivalent sidesets on the target mesh.



The commands to automatically transfer nodesets and sidesets are given below:

**transfer nodeset <ids> onto {tri <ids> |face <ids> |tet <ids> |hex <ids> } [tolerance <value> ]**

**transfer sideset <ids> onto {tri <ids> |face <ids> |tet <ids> |hex <ids> } [tolerance <value> ] [log]**

**tri <ids> and face <ids> :** if the target mesh is a surface mesh, use tri or face option.

**tet <ids> and hex <ids>:** this option can be used and the boundary skin mesh of the volumetric mesh will be extracted automatically to associate skin elements and nodes to the sidesets and nodesets, respectively.
**tolerance <value>:** The tolerance option must be used only if the default smart local tolerance doesn't give the desired results.
**log :** The log option outputs the area of sidesets in source and target meshes and the ratio of the areas. The below table shows an example output. For example, the area_ratio can be used to adjust the pressure boundary conditions to maintain same forces.
**Log output of transfer sideset command**

| sideset_id | old_area ID | new_area | area_ratio ID |
|---|---|---|---|
| 1 | 56.4843 | 4.8099 | 0.09 |
| 2 | 94.2209 | 68.4337 | 0.73 |

The below Python script Illustrates transfer of sidesets and nodesets from a source mesh to a target mesh using the existing and the new commands

```
#!python
import cubit
cubit.init(['cubit','-nojournal'])
# STEP 1: import the source mesh as mesh based geometry (MBG); delete mesh and blocks
cubit.cmd('import mesh geometry "design_domain_mesh.exo" feature_angle 135.00 merge ')
cubit.cmd('del mesh')
cubit.cmd('del block all')
# STEP 2: import the target mesh as a free mesh
cubit.cmd('import mesh "optimized_mesh.exo" no_geom')
# STEP 3: transfer sideset and nodeset from the source mesh to the target mesh
cubit.cmd('transfer sideset all onto tet all')
cubit.cmd('transfer nodeset all onto tet all')
# STEP 4: export optimized mesh with sidesets
cubit.cmd('export mesh \'optimized_mesh_with_bc.exo\' block all sideset all nodeset all')
```

*Displaying Nodesets and Sidesets*

Nodesets and Sidesets can be viewed individually through CUBIT by employing the following commands:

**Draw NodeSet <nodeset_id_range> [Color <color_spec>] [add]**

**Draw SideSet <sideset_id_range> [Color <color_spec>] [add]**

Nodeset and Sideset colors can also be changed using the following commands:

**Color NodeSet <nodeset_id_range> {color|Default}**

**Color SideSet <sideset_id_range> {color|Default}**

*Nodeset Associativity Data*

Nodesets can be used to store geometry associativity data in the Exodus II file. This data can be used to associate the corresponding mesh to an existing geometry in a subsequent CUBIT session. This functionality can be used either to associate a previously-generated mesh with a geometry (See Importing an Exodus II File), or to associate a field function with a geometry for adaptive surface meshing (See Adaptive Meshing).
The commands to control and list whether associativity data is written or read from an Exodus II files are the following:

**List Import Mesh NodeSet Associativity**

**List [Export Mesh] NodeSet Associativity**

**List [Export Mesh] NodeSet Associativity Complete**

**set Import Mesh NodeSet Associativity [ON|off]**

**[set] [Export Mesh] NodeSet Associativity [on|OFF]**

**[set] [Export Mesh] NodeSet Associativity Complete [On|OFF]**

Associativity data is stored in the Exodus II file in two locations. First, a nodeset is written for each piece of geometry (vertices, curves, etc) containing the nodes owned for that geometry. Then, the name of each geometry entity is associated with the corresponding nodeset by writing a property name and designating the corresponding nodeset as having that property. Nodeset numbers used for associativity nodesets are determined by adding a fixed base number (depending on the order of the geometric entity) to the geometric entity id number. The base numbers for various orders of geometric entities are shown in the following table. For example, nodes owned by curve number 26 would be stored in associativity nodeset 40026.

**Table 1. Nodeset ID base numbers for geometric entities**

| Geometric Entity | Base Nodeset ID |
|---|---|
| Vertex | 50000 |
| Curve | 40000 |
| Surface | 30000 |
| Volume | 20000 |

Instead of storing just the nodes owned by a particular entity, nodes for lower order entities are also stored. For example, the associativity nodeset for a surface would contain all nodes owned by that surface as well as the nodes on the bounding curves and vertices.

*Equation-Controlled Distribution Factors*

By default, distribution factors on nodesets or sidesets are written with a constant value of "1" at each node. It is also possible to vary the distribution factor for each node in a nodeset or sideset, using an equation to control the value of the distribution factor at each node. To do so, an equation must first be defined using the command:

**Create Equation "<expression>" name "<name>"**

where **expression** is any mathematical expression which evaluates to a single number, and **name** is the name by which this equation will be known. The expression is written using aprepro syntax, with a few differences from the use of APREPRO in its usual context.

1. The expression as a whole is not wrapped in curly braces "{" and"}".
2. The expression may include any of the following pre-defined variables:

**x** - The x-coordinate of the current node
**y** - The y-coordinate of the current node
**z** - The z-coordinate of the current node
**n** - The CUBIT ID of the current node. This is the ID of the node in CUBIT, which may not be the same as the node's ID in the Exodus II file.

For example, to define an equation which varies from -10 to 10 based on the sine of the node's x_coordinate:

**Create Equation "10*sin(x)" Name "my_equation"**

Once an equation has been defined, it can be applied to a nodeset or sideset:

**Nodeset <id> Distribution Equation "<equation_name>"**
**Sideset <id> Distribution Equation "<equation_name>"**

For example, to apply the equation created earlier to nodeset 10:

**Nodeset 10 Distribution Equation "my_equation"**

When nodeset 10 is written to an Exodus II file, "my_equation" will be evaluated once for each node in the nodeset, with the values of x, y, z, and n set to appropriate values for the node. The result is used as the distribution factor for that node.

Here is a complete example that writes out the distribution factors 0.0, 0.5, and 1.0 for the 3 nodes on the curve:

```
# Create a straight line from (0,0,0) to (1,0,0)
create vertex 0 0 0
create vertex 1 0 0
create curve vertex 1 2
# Mesh with 3 nodes
curve 1 interval 2
mesh curve 1
# Create a block and a nodeset
block 1 curve 1
nodeset 1 curve 1
# Define an equation and apply it to the nodeset
```

```
create equation "x" name "simple_eq"
nodeset 1 distribution equation "simple_eq"
# Write the mesh
export mesh "temp.g" overwrite
```

Here is another complete example that varies the distribution factors for sideset 20 from zero to 1, depending on the node's x-coordinate. The sideset is applied to sides of HEX20 elements, so each element side has 8 different distribution factors.

```
# Mesh a cube
brick x 10
mesh volume 1
# Create a block of 20-noded hexes
block 1 volume 1
block 1 element type hex20
# Apply a sideset to be used for a variable pressure
sideset 20 surface 1
# Define an equation and apply it to the sideset
create equation "(x+5)/10" name "zero_to_one"
sideset 20 distribution equation "zero_to_one"
# Write the mesh
export mesh "temp.g" overwrite
```

Note that distribution equations only affect Exodus II output. Equations are currently ignored for other mesh file types.
See APREPRO in the appendix.

*Nodesets/Sidesets/Blocks Behavior with Geometric Entity Copy*

The below commands can be used to set the behavior of bc (boundary contitions of nodesets/sidesets/blocks) propagation when a geometric entity is copied. The default **OFF** option specifies that any bc containing the copied geometry and/or mesh of the copied geometry will not be contained in a bc. The **use_original** option will add the new, copied geometry and/or mesh into the existing bc. The **on** option will create new bcs, containing the copied geometry and/or mesh, mirroring the original bcs.

**set copy_nodeset_on_geometry_copy [on | OFF| use_original]**
**set copy_sideset_on_geometry_copy [on | OFF| use_original]**
**set copy_block_on_geometry_copy [on | OFF| use_original]**

# non exodus

CUBIT Initial Conditions

In CUBIT, initial conditions can be applied to nodesets. CUBIT supports the following types of initial conditions: displacement, velocity,

acceleration, temperature, and generic field. For now, initial conditions are only supported by CUBIT's Abaqus exporter. The commands to create an initial condition are:

**Create initialcondition [id] type temperature [name <'name'>] [{add|on} nodeset <entity_list>] [value <val>]**

**Create initialcondition [id] type displacement [name <'name'>] [{add|on} nodeset <entity_list>] [dof {1|2|3|4|5|6} {value <value>|off}]**

**Create initialcondition [id] type velocity [name <'name'>] [{add|on} nodeset <entity_list>] [dof {1|2|3|4|5|6} {value <value>|off}]**

**Create initialcondition [id] type acceleration [name <'name'>] [{add|on} nodeset <entity_list>] [dof {1|2|3|4|5|6} {value <value>|off}]**

**Create initialcondition [id] type field [name <'name'>] [{add|on} nodeset <entity_list>] [variable <n> value <val>**

For most of the initial conditions, only two pieces of data are required: a list of nodesets this IC is applied to, and an initial value. Optionally, a name can be specified for the initial condition. To modify an initial condition, replace the word “create” with the word “modify.” If modifying an IC, the IC’s ID must be passed in so CUBIT knows which IC you are modifying. Example:

Modify initialcondition 3 value 1.23

Use this command to list the information about a set of initial conditions:

List initialcondition <id_list>

Use this command to delete a set of initial conditions:

Delete initialcondition <id_list>

## Using Constraints

Constraints couple the motion of a set of nodes to the motion of a reference node. Rigid bodies and kinematic constraints do exactly this for blocks and sidesets, respectively. A distributing constraint allows users to average the constrained motion of a sideset by using weight factors to control force transmission (to be specified outside of CUBIT). A tie constraint can be used to tie the elements of one sideset to the elements of another. Currently, only the Abaqus Exporter supports this type of constraint.

Note that as of CUBIT 13.0, constraints are supported by the Abaqus Importer/Exporter only. Contact the CUBIT support team if support in additional file formats is needed.

To create a constraint, use one of the following commands:

**Create Constraint {Kinematic|Distributing} [name '<name>'] [vertex|node] <id> sideset <id>**

**Create Constraint Rigidbody [name '<name>'] [vertex|node] <id> block <id>**

**Create Constraint Tie [name '<name>'] master sideset <id> slave sideset <id>**

A constraint's name, dependent object, and independent object can be changed using the following commands:

**Modify Constraint <id|name> [name '<name>'] [vertex|node] <id> sideset <id>**

**Modify Constraint <id|name> [name '<name>'] [vertex|node] <id> block <id>**

**Modify Constraint <id|name> [master sideset <id>] [slave sideset <id>]**

Constraints can be listed or deleted using the following commands:

**List Constraint <id_range>**

**Delete Constraint <id_range>**

## Cubit Boundary Conditions

- Sets
- Restraints
- Loads
- Contacts
- CFD Boundary Conditions
- Miscellaneous commands
- CUBIT Initial Conditions

In CUBIT, boundary conditions are applied to sidesets or nodesets. Sidesets and nodesets can contain geometry or mesh. This means that models can be remeshed without worrying about losing boundary condition data if the boundary condition is applied to a geometry-based sideset/nodeset.
The sideset/nodeset used by a boundary condition will be visible to the user, and the user can modify the sideset/nodeset separately from the boundary condition. Sidesets/nodesets can be assigned to (or removed from) a boundary condition at any time.
Boundary conditions are broken into four groups: Restraints, loads, contact, and cfd. Each restraint that is created will belong to a restraint set, each load will belong to a load set, and each contact definition will belong to a contact set. A boundary condition set consists of any number of restraints, contact pairs, and loads. CFD boundary conditions do not belong to boundary condition sets.
**Table 1: Overview of boundary condition entities available in Cubit**

| Entity | Description and scope |
|---|---|
| Acceleration | Creates an acceleration boundary condition (acts on a body, volume, surface, curve, or vertex) |
| Velocity | Creates a velocity boundary condition (acts on a body, volume, surface, curve, or vertex) |
| Boundary Condition Set | Creates a BC set (contains restraint, load and contact sets) |
| Contact Region | Creates a contact region between two surfaces or two curves |
| Contact Pair | Creates a contact pair between two previously defined contact regions |
| Displacement | Creates a displacement boundary condition (acts on a body, volume, surface, curve or vertex) |
| Temperature | Create a temperature boundary condition (acts on a surface, curve or vertex) |
| Force | Creates a force boundary condition (acts on a surface, curve or vertex) |
| Pressure | Creates a pressure boundary condition (acts on a surface or curve) |
| Heat flux | Creates a heat flux boundary condition (acts on a surface or curve) |
| Inlet Velocity | Creates an inlet velocity boundary condition (acts on a surface) |
| Inlet Pressure | Creates an inlet pressure boundary condition (acts on a surface) |
| Inlet Massflow | Creates an inlet massflow boundary condition (acts on a surface) |
| Outlet Pressure | Creates an outlet pressure boundary condition (acts on a surface) |
| Farfield Pressure | Creates a farfield pressure boundary condition (acts on a surface) |
| Symmetry | Creates a symmetry boundary condition (acts on a surface) |

## CFD Boundary Conditions

CUBIT can export models to the Fluent mesh format and CNGS unstructured mesh format for CFD analysis. CUBIT also supports defining the below CFD boundary conditions. Only the region on which the BC acts can be defined in CUBIT. The data associated with each boundary condition (pressure, velocity, mass values, etc.) is not defined within CUBIT and must be assigned using a CFD model editor, such as Fluent.

**create cfd_bc [id] {axis|exhaustfan|fan|inletvent|intakefan|interface|interior|massflowinlet|outflow|outletvent|periodic|periodicshadow|porousjump|pressurefarfield|pressureinlet|pressureoutlet|radiator|supersonicinflow|supersonicoutflow|symmetry|velocityinlet|wall} [name <'name'>] [{add|on} {sideset|surface} <entity_list>**

The following shows the commands for modifying, deleting, and lising CFD boundary conditions.

**modify cfd_bc [id] [name <'name'>] [{add|remove} {sideset|surface} <entity_list>**

**delete cfd_bc {<ids>|'<string>'}**

**list cfd_bc {<ids>}**

## Using Contact Surfaces

- Contact Region
- Contact Pair
- Auto-Contact Tool

*The Contact Region*

To define contact between two entities, Cubit requires each entity to be defined as a separate **contact region**. Each region can be made up of multiple 1D or 2D entities.

**Create Contact Region [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>]**

**Modify Contact Region {id_list|'name'|All} [Name <'name'>] [{Add|Remove} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>]**

*The Contact Pair*

> **create contact pair [id] [name <'name'>] [master contact region <id|'name'>] [slave contact region <id|'name'>] [friction <value>] [tolerance <value>] [tied {on|OFF}] [General <on|OFF> [Exterior <on|OFF>]]**
>
> **modify contact pair {id_list|'name'|all} [name <'name'>] [master contact region <id|'name'>] [slave contact region <id|'name'>] [friction <value>] [tolerance <value>] [tied {on|OFF}] [General <on|OFF> [Exterior <on|OFF>]]**

A contact pair is composed of two contact regions. One region will be the 'master' surface, and the other will be the 'slave.' 2D contact regions can not be mixed with 1D

contact regions. The friction coefficient can also be included. The **tolerance** keyword is currently unused. Use the **tied** keyword to specify that the contact is to define tied contact between the two contact regions, essentially “gluing” the parts together. **Currently, this option is only available when using the Abaqus Exporter**.
The **General** keyword can be used to specify general (i.e. global) contact without specifying surfaces/curves to use as contact pairs. Currently, this keyword is only valid when exporting to Abaqus. If the **Exterior** keyword is used with the **General** keyword, then Abaqus will consider all exterior surfaces when determining contact regions. If the Exterior keyword is omitted, then the user must provide a master contact region and/or a slave contact region.

*Auto-Contact Tool*

With the auto-contact tool, Cubit can search for contact pairs and automatically set up all of the necessary contact regions and contact pairs.

**Create Contact Autoselect [{Volume|Surface|Curve} <ids>] [Master Volume <id>] [Maxgap <value>] [Curve_Contact]**

The optional geometry list can be used to limit Cubit’s search to only a subset of entities. If this list is omitted, all bodies in the model will be searched. The optional **master volume** keyword can be used to tell Cubit which volume should be used as the master contact region. If this keyword is omitted, the user will not have control over which volume is the master region. The **maxgap** keyword can be used to control how Cubit searches for contact regions. This value is used as the maximum amount of gap that can exist between two surfaces and be identified as a contact region. If this keyword is omitted, the geometry tolerance is used. The **curve_contact** keyword can be used to indicate the model requires curve contact as opposed to surface contact.

## Using Loads

- Force
- Pressure
- Heat Flux
- Convection

*Forces*

**Create Force [id] [Name <'name'>] [ {Add|On} {Nodeset|Surface|Curve|Vertex|Face|Tri|Edge|Node} <entity_list>] [Force Value <val>] [Moment Value <val>] [Direction { direction_options}]**

**Create Force [id] [Name <'name'>] [ {Add|On} {Nodeset|Surface|Curve|Vertex|Face|Tri|Edge|Node} <entity_list>] [ Vector <val> <val> <val> <val> <val> <val>]**

**Modify Force {id_list|'name'|all} [Name <'name'>] [ {Add|Remove} {Nodeset|Surface|Curve|Vertex|Face|Tri|Edge|Node} <entity_list>] [Force Value <val>] [Moment Value <val>] [Direction { direction_options}]**

**Modify Force {id_list|'name'|all} [Name <'name'>] [ {Add|Remove} {Nodeset|Surface|Curve|Vertex|Face|Tri|Edge|Node} <entity_list>] [ Vector <val> <val> <val> <val> <val> <val>]**

A CUBIT user has the ability to create forces on 0D, 1D, and 2D entities. A force can be created using the direction syntax (see Specifying Direction). If the vector keyword is used, the first three values are the force components, and the last three values are the moment components.
The use of the **force** and **moment** keywords specify the type of load. If both a force and a moment are to be applied, first create one of them, then modify it to add the other. Note that every instance of a **force** or **moment** keyword must have an accompanying **value** keyword.
Regarding **force** and **moment** keywords, the following detail may be helpful:
A user may create a force and moment at the same time, but can only specify a direction once. If the force and moment have the same unit vector, it will be successful. If a users wants to create a force in the direction 1,2,3 and a moment in the direction 1,0,0, the user will have to create one, then add the other by modifying it.

*Using Pressure*

**Create Pressure [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Magnitude <value>] [TOP|Bottom] [PRESSURE|Totalforce]**

**Modify Pressure {id_list|'name'|all} [Name <'name'>] [{Add|Remove} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Magnitude <value>] [TOP|Bottom] [PRESSURE|Totalforce]**

Cubit users can create pressure boundary conditions on 1D and 2D entities. Positive surface pressures acting on solid elements are defined as pointing into the face of the elements. Pressures are always normal to the face. For shells and independent surfaces, a ‘left-hand-rule’ is employed. Point your left thumb at the surface in question. If the direction your fingers curl matches the direction of ascending vertex numbering, the direction of the pressure vectors will match the direction of your thumb.

**Value**

The **value** variable is the magnitude of the pressure boundary condition. If the user leaves this value blank, CUBIT will assign the pressure magnitude to zero (possibly a trivial case) and issue a warning. Typing a negative value will not flip the direction of the pressure arrows on the display; instead, the pressure magnitude will be negative.

#### Pressure and Total Force

The pressure and totalforce keywords are used to modify the pressure boundary condition. The pressure keyword is the default. All pressures applied with this keyword present (or with both of these keywords absent from the command string) are pure pressures. If the user enters the totalforce keyword, the pressure magnitude is divided by the area of the surface the pressure is acting on (or the length of the curve, for a curve pressure). In effect, the user is entering a force that is treated and exported as a pressure.

#### Top and Bottom

The **top** keyword (default) indicates the pressure will occur on the top of a shell element. Specifying **bottom** will cause the pressure to be applied to the bottom of the element.

*Using Heat Flux*

**Create Heatflux [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Value <value>]**

**Create Heatflux [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Face|Tri} <entity_list>] [Top <value> Bottom <value>]**

**Modify Heatflux {id_list|'name'|All} [Name <'name'>] [{Add|Remove} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Value <value>]**

**Modify Heatflux {id_list|'name'|All} [Name <'name'>] [{Add|Remove} {Sideset|Surface|Face|Tri} <entity_list>] [Top <value> Bottom <value>]**

A CUBIT user may apply heat flux boundary conditions to 1D and 2D entities, including thin-shell elements.

#### Top and Bottom Values

Heat fluxes can be applied to thin-shell elements as well. The same rules apply to thin-shell heat fluxes as to thin-shell temperatures: thin-shell heat fluxes can only be applied to surfaces and heat flux boundary conditions cannot contain regular and thin-shell heat flux values (see journal below). However, thin-shell heat flux commands do not contain gradient or middle keyword options. Only top and bottom keywords are supported.

*Using Convection*

**Create Convection [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Surrounding {<value>| Top <value> Bottom <value>} Coefficient {<value>| Top <value> Bottom <value>}]**

**Modify Convection [id] [Name <'name'>] [{Add|On} {Sideset|Surface|Curve|Face|Tri|Edge} <entity_list>] [Surrounding {<value>| Top <value> Bottom <value>} Coefficient {<value>| Top <value> Bottom <value>}]**

A Cubit user can apply convection boundary conditions to 1D and 2D entities. Convection is a transport of thermal energy that is proportional to the difference between the surface temperature and the temperature of the surroundings.

#### Surrounding

The surrounding keyword specifies the temperature surrounding the entity with the convection boundary condition.

#### Coefficient

The coefficient keyword is a convection coefficient, in units of energy per length times time times temperature (i.e., [energy]/([length]*[time]*[temperature]) ).

## Miscellaneous Boundary Condition Commands

- Delete
- List
- Draw
- Highlight

*Delete*

The **BC delete** keyword combination is used to delete boundary conditions. The current list of all entities that can be deleted using this command were shown in Table 1. Cubit currently has no ‘undo’ command to ‘undelete’ a boundary condition deletion.

**Delete {bc_type} [<id-range>|All]**

**Delete Boundary Conditions**

Every set (and boundary condition within them) can be deleted at once by typing **delete boundary conditions**. This command will delete all boundary conditions from your model.

*List*

The **List** keyword combination is used to list boundary conditions. The current list of all entities that can be listed using this command was shown in Table 1. Cubit's parser can evaluate boundary conditions given the entities they act on. For example, "List pressure in surface 1" will list all pressures that act on Surface 1.

**List {bc_type} [<id-range>]**

**List Boundary Conditions**

Every set (and boundary condition within them) may be listed at once by typing **list boundary conditions**. CUBIT will list the number of sets and individual boundary conditions in your model. This command will list the total number of each type of set and boundary condition, including boundary conditions that are not a part of a BC set.

*Draw*

**Draw {bc_type} {<id-range>|all}[Add]**

The **draw** keyphrase allows a CUBIT user to draw any type of boundary condition. This command will clear the graphics window of every part of the model except for the selected boundary condition. Using the **add** keyword will permit multiple boundary conditions to be drawn at the same time. Any combination of boundary conditions and entities that were valid for delete and list are also valid for draw.

*Highlight*

**Highlight {bc_type} {<id-range>|All}**

The **highlight** keyphrase allows a CUBIT user to highlight any boundary condition. Highlighting a boundary condition will turn it bright orange and the vectors defining it will thicken. The highlight command is similar to the draw command.

## Using Restraints

- Displacement
- Acceleration
- Velocity
- Temperature

*Displacements/Accelerations/Velocities*

A CUBIT user has the ability to create displacement boundary conditions on most geometric entities found within Cubit.

**Create Displacement [id] [Name <'name'>] [{Add|On} {Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node}**

**<entity_list>] [DOF {All|{[1][2][3][4][5][6]}} Fix <value>]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

**Modify Displacement {id_list|'name'|all} [name <'name'>] [{Add|Remove}**
**{Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node}**
**<entity_list>] [DOF {All|{[1][2][3][4][5][6]}} {Fix <value>|Free}]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

**Create Acceleration [id] [Name <'name'>] [{Add|On}**
**{Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>]**
**[DOF {All|{[1][2][3][4][5][6]}} Fix <value>]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

**Modify Acceleration {id_list|'name'|all} [name <'name'>] [{Add|Remove}**
**{Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>]**
**[DOF {All|{[1][2][3][4][5][6]}} {Fix <value>|Free}]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

**Create Velocity [id] [Name <'name'>] [{Add|On}**
**{Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>]**
**[DOF {All|{[1][2][3][4][5][6]}} Fix <value>]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

**Modify Velocity {id_list|'name'|all} [name <'name'>] [{Add|Remove}**
**{Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>]**
**[DOF {All|{[1][2][3][4][5][6]}} {Fix <value>|Free}]**
**[SmallestCombine|Average|LargestCombine|OVERWRITE]**

A number of required and optional keywords make the BC create displacement command one of the more complicated of the boundary condition commands. These keywords will be examined individually in detail.

**Degrees of Freedom**

The **dof** keyword is the heart of this command. It specifies how to constrain the entity in question. The keyword is an abbreviation for ‘degree of freedom’. Typing the optional keyword all tells CUBIT that the entered command will encompass all six degrees of freedom. The degrees of freedom (1 - 6) are defined below in Table 2.

**Table 2: CUBIT definitions of the six degrees of freedom.**

| DOF | Physical analog |
|---|---|
| 1 | x-translation |
| 2 | y-translation |
| 3 | z-translation |
| 4 | x-rotation |

| | |
|---|---|
| 5 | y-rotation |
| 6 | z-rotation |

CUBIT will allow displacement commands to be applied upon between one and all six of the degrees of freedom. The degrees of freedom do not need to be entered in any order. The command strings ‘ 1 2 3 4 5 6 ‘ ‘2 6 1 4 3 5’ and ‘all’ will end with the same result.

#### Fixed or Free

The **fix** and **free** keywords tell CUBIT whether an entity’s displacement defined by the dof keyword is to be enforced with a finite value or not. If the displacement is fixed, the entity will be constrained in the pre-specified degrees of freedom. A decimal number entered after the fix keyword will be the value of the enforced degree(s) of freedom. CUBIT allows the user to leave this value blank if the enforced displacement is to be zero, for convenience. However, entering ‘0’ is still permitted. If a user wishes to remove a displacement from an entity, he or she should just delete it rather than trying to set all of the degrees of freedom to free.

#### Displacement Combinations

The **SmallestCombine**, **Average** and **LargestCombine** keywords deal with displacement combinations. These keywords only apply when a user is modifying an existing displacement boundary condition.
The **SmallestCombine** keyword will compare the existing displacement values with the current (residing on the command line) displacement values. The keyword will modify the displacement to the match the displacements dictated by the boundary condition that has the smallest absolute value. If the boundary condition with the smallest absolute value is the existing value, the displacement boundary condition will be unchanged. If the current boundary condition has a smaller absolute value than the existing displacement, the displacement boundary condition will be changed to incorporate the new values.
The **Average** keyword will average the existing displacement values with the current (residing on the command line) displacement values. Note that these averages are not continually updated (i.e., they are not weighted). If a user created a displacement boundary condition and constrained a degree of freedom to 10.0 and then constrained the same degree of freedom to 20.0 with the Average keyword, the new displacement value would be 15.0. But if a user constrained the same degree of freedom to 30.0, while using the Average keyword, the new displacement value would be 22.5 ([15+30]/2), not 20.0 ([10+20+30]/3).
The **LargestCombine** keyword will compare the existing displacement values with the current (residing on the command line) displacement values. The keyword will modify the displacement to the match the displacements dictated by the boundary condition that has the largest absolute value. If the boundary condition with the largest absolute value is the existing value, the displacement boundary condition will be unchanged. If the current boundary condition has a larger absolute value than the existing displacement, the displacement boundary condition will be changed to incorporate the new values.
When none of these keywords are specified, CUBIT will combine displacements in its default mode, Overwrite. The Overwrite keyword overwrites the entity’s previous

displacement boundary condition(s) with the displacement values specified in the command.

### *Temperature*

CUBIT can create temperature boundary conditions on most geometric and mesh entities. The temperature boundary condition can also be applied to thin-shell elements.

**Create Temperature [id] [Name <'name'>] [{Add|On} {Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>] [Value <val>]**

**Create Temperature [id] [Name <'name'>] [{Add|On} {Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>] [{ Top <val> Bottom <val> | [Middle <val>] [Gradient <val>] } ]**

**Modify Temperature {id_list|'name'|all} [name <'name'>] [{Add|Remove} {Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>] [Value <val>]**

**Modify Temperature {id_list|'name'|all} [name <'name'>] [{Add|Remove} {Nodeset|Volume|Surface|Curve|Vertex|Hex|Tet|Face|Tri|Edge|Node} <entity_list>] [{ Top <val> Bottom <val> | [Middle <val>] [Gradient <val>] } ]**

The **value** keyword defines the amplitude (temperature). The other command options are discussed below

#### Top, Gradient, Middle, Bottom

The above keywords are only used for thin-shell elements (i.e., 2D entities). The valid combinations are limited to: top and bottom, middle and gradient, only gradient or only middle. It should be noted that temperature boundary conditions cannot contain regular and thin-shell temperature values.

## Boundary Condition Sets

**Create bcset [id] [name <'name'>] [After bcset <id>] [{Add|Remove} {bc_type} <id-range | <with name 'name'> >] [analysistype {STATIC|heat|dynamic|modal}] [modal_max_frequency <value>]**

**Modify bcset {id_list|'name'|all} [name <'name'>] [After bcset <id>] [{Add|Remove} {bc_type} <id-range | <with name 'name'> >] [analysistype {STATIC|heat|dynamic|modal}]**

*** *ABAQUS Parameters* ***

**Modify bcset {id_list|'name'|all} [max_step_increments <value>] [nonlinear_geometry <on|OFF>][perturbation <on|OFF>][stabilize <on|OFF>] [steadystate <on|OFF>][modal_max_frequency <value>]**

**Modify bcset {id_list|'name'|all} [initial_step_size <value>] [step_period <value>][min_step_size <value>] [max_step_size <value>][min_step_temperature_change <value>]**

**Modify bcset {id_list|'name'|all} [mass_scaling <on|OFF>] [mass_scaling_dt <value>][mass_scaling_factor <value>] [mass_scaling_type <'uniform'|'BELOW_MIN'|'set_equal_dt'>]**

**Modify bcset {id_list|'name'|all} [restart <on|OFF>][restart_overlay <on|OFF>] [{restart_frequency|restart_num_intervals} <value>]**

**Modify bcset {id_list|'name'|all} [output_field <on|OFF>] [output_field_frequency <value>] [output_history <on|OFF>] [output_history_frequency <value>]**

**Modify bcset {id_list|'name'|all} [el_file <on|OFF>][el_file_frequency <value>] [node_file <on|OFF>][node_file_frequency <value>]**

**Modify bcset {id_list|'name'|all} [el_print <on|OFF>][el_print_frequency <value>] [node_print <on|OFF>][node_print_frequency <value>]**

**** NASTRAN Parameters ****

**Modify bcset {id_list|'name'|all} {displacement_output <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> }}**

**Modify bcset {id_list|'name'|all} {oload <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> }}**

**Modify bcset {id_list|'name'|all} {mpcforces <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> }}**

**Modify bcset {id_list|'name'|all} {spcforces <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> }}**

**Modify bcset {id_list|'name'|all} {stress <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> } {CENTER|cubic|sgage|corner} {VONMISES|maxs}}**

**Modify bcset {id_list|'name'|all} {element_strain_energy <on|OFF> {PLOT|print|punch|punchprint} {group <ALL|none|<id>> } {AVERAGE|amplitude|peak}}**

CUBIT can create BC sets, which is a group of previously defined loads, restraints and contact pairs. A BCSet is used to define a load case (analysis step) when writing out 3rd party analysis decks. A BCSet can be a static analysis set, a thermal analysis set, a modal analysis set, or a dynamic analysis set by specifying the **analysistype**. The **After** keyword can be used to define the order that the BCSets will be written when the model is exported.
Several solver-specific parameters can be set for a BCSet. For **ABAQUS**, parameters associated with *STEP, *STATIC, *DYNAMIC, *FREQUENCY, *HEAT TRANSFER, *MASS SCALING, *RESTART, *OUTPUT, *EL FILE, *NODE FILE, *EL PRINT, and *NODE PRINT can be modified. For **Nastran**, output requests can be defined for Displacement, Reaction Loads, MPC Forces, SPC Forces, Stress, and Element Strain Energy.

# Boundary Layer Meshing

## Boundary Layer Meshing

Boundary layer meshing is best accessed via the GUI.
To create a boundary layer:

1. On the Command Panel, click on **Mesh** and then **Boundary Layer**.
2. Click on the **Create** action button.
3. Select **System Assigned ID** or manually enter an ID.
4. On the **Settings** tab, enter the appropriate size for the first row in the boundary layer.
5. Enter a value for the **Growth Factor**.
6. Specify the **Number of Layer**.
7. Optionally select **Internal Continuity**.

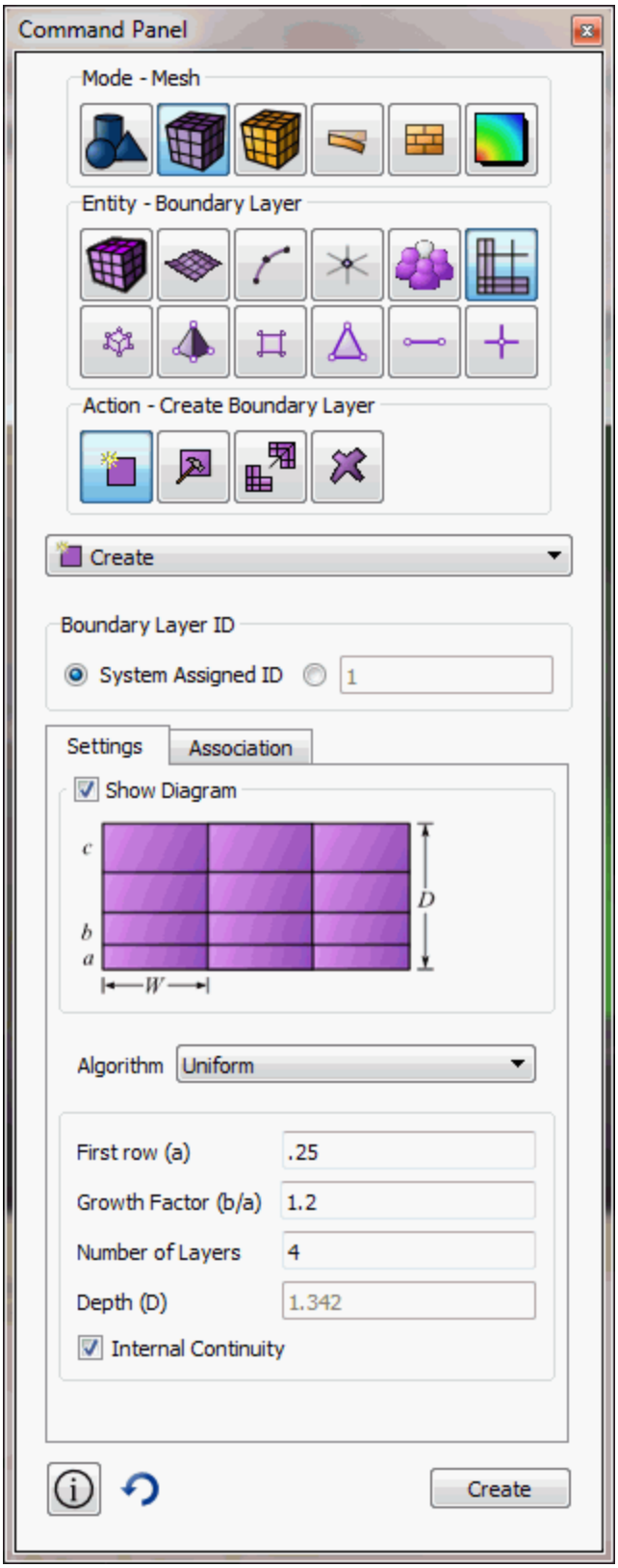



**Figure 1 - Settings Panel**

**First row(a)** -- the height of the first layer in the boundary layer
**Growth Factor(b/a)** -- the factor by which each layer grows
**Number of Layers** -- the number of layers that make up a boundary layer
**Internal Continuity** -- continuity flag for boundary layers. If on, all intersections are a side type.

8. On the **Association** tab, select **Curve** for 2D boundary layers or **Surface** for 3D boundary layers.

   For 2D boundary layers:

   1. In the **Curve ID(s)** field, enter the curve ID(s) where the boundary layer(s) begins.
   2. In the **Surface ID** field, enter the surface ID that contains the boundary layer(s).
   3. Click **Add** to add the curves to the boundary layer(s).
   4. Click **Create** to create the boundary layer(s).

   For 3D boundary layers:

   5. In the **Surface ID(s)** field, enter the surface ID(s) where the boundary layer(s) begins.
   6. In the **Volume ID** field, enter the volume ID that contains the boundary layer(s).
   7. Click **Add** to add the curves to the boundary layer(s).
   8. Click **Create** to create the boundary layer(s).

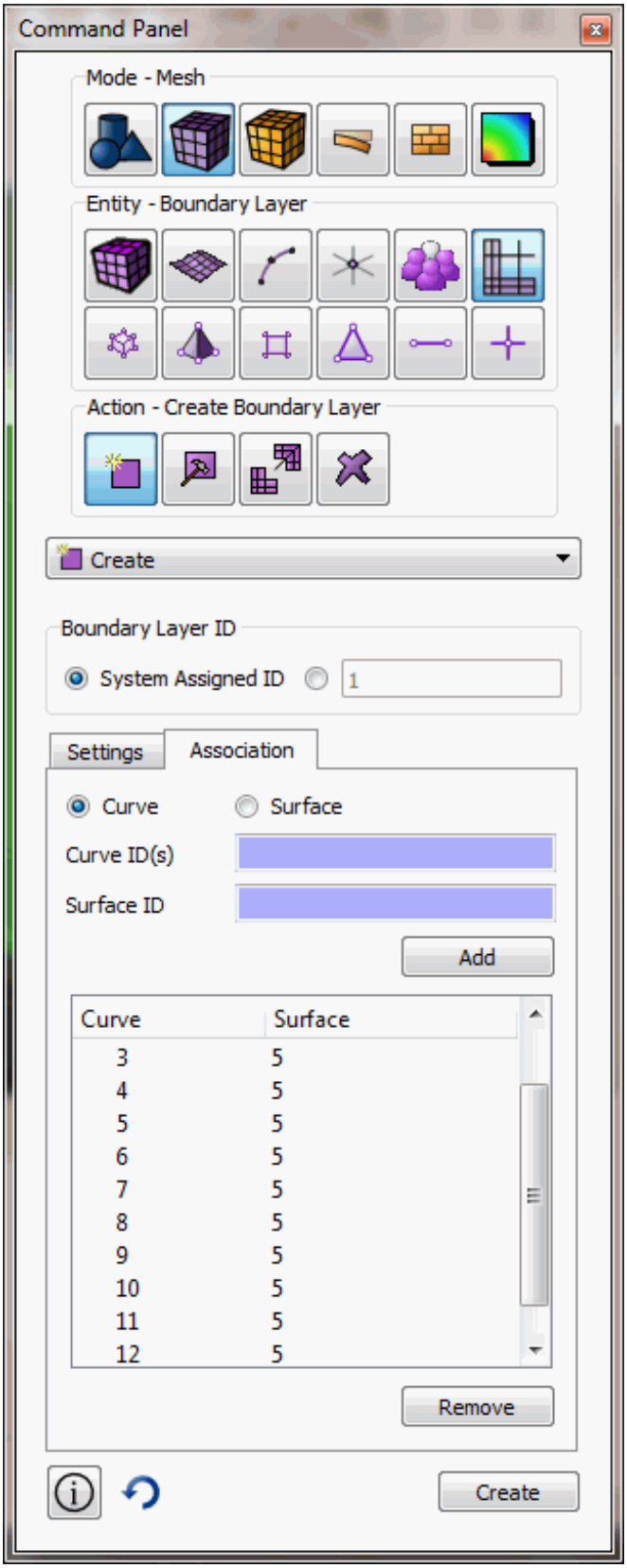



**Figure 2 - Association Panel**

For 2d boundary layers, a curve/surface pair is given to create a boundary layer starting from a curve and growing out on the given surface.
For 3d boundary layers, a surface/volume pair is given to create a boundary layer starting from a surface and growing out on the given volume.

*Intersection Types*

In some cases, the user may want to adjust the intersection types. This could be because the automatic intersection type is not desired, or because it is not workable due to ambiguity.
The four intersection types are:

- end - suitable for angles between 0 and 135 degrees.
- side - suitable for angles between 135 and 225
- corner - suitable for angles between 225 and 315
- reversal - suitable for angles 315 to 360

These intersection types may be set on a vertex/surface basis and on a curve/volume basis.

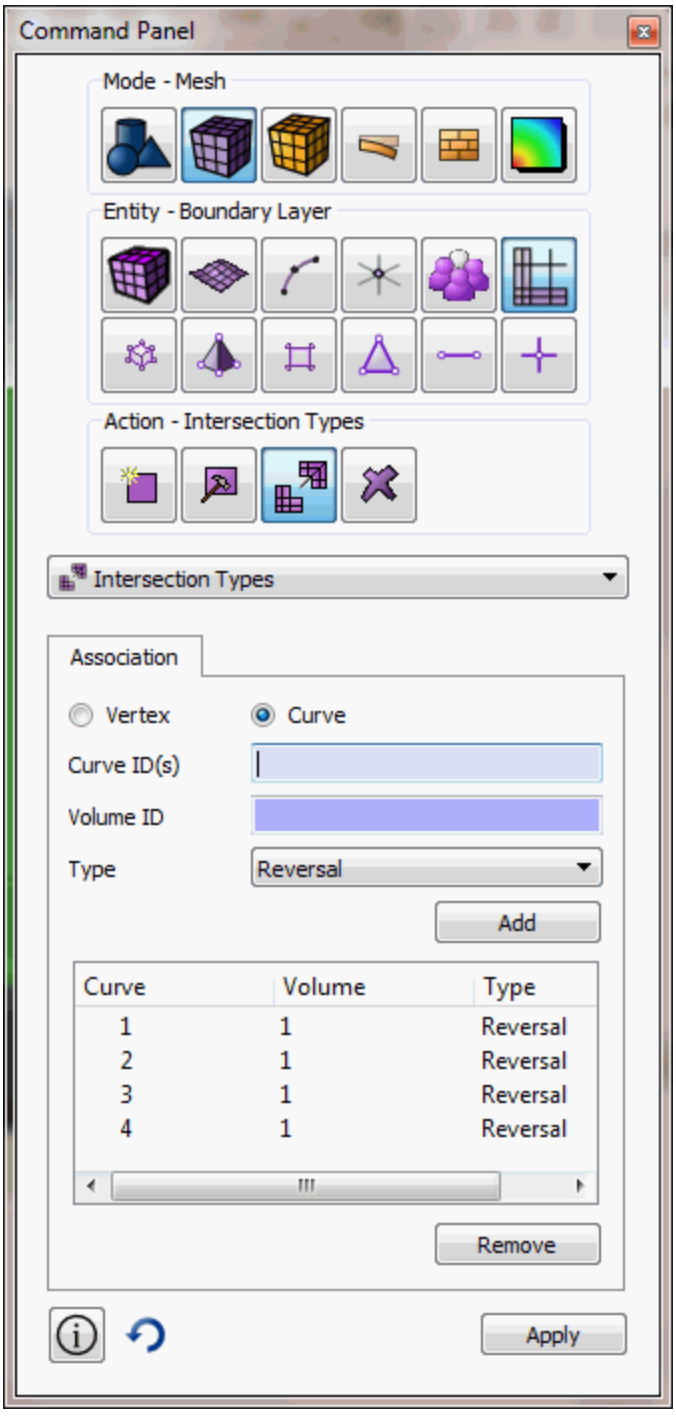



**Figure 3 - Intersections Type Panel**

*Current Limitations*

Not all combinations of intersection types and topology are supported for 3d cases. An end, corner, or reversal may not span multiple curves in a single volume. A possible workaround is to composite the curves to make a single curve.

Not all meshing schemes may be used in combination with boundary layers. In cases where it is not supported, the boundary layer will be ignored in mesh generation. It is supported with the following schemes:

- trimesh
- pave
- map
- submap
- sweep
- tetmesh.

*Underlying Trelis Commands*

**Create Boundary_layer <id>**

**Delete Boundary_layer <id>**

**Modify Boundary_layer <id> add Curve <id_range> Surface <id>**

**Modify Boundary_layer <id> remove Curve <id_range> Surface <id>**

**Modify Boundary_layer <id> add Surface <id_range> Volume <id>**

**Modify Boundary_layer <id> remove Surface <id_range> Volume <id>**

**"*** Only three of the four parameters should be specified ***

"*** (Height, Growth, Layer, or Depth) ***

**Modify Boundary_layer <id> uniform Height <double> Growth <double> Layers <double> Depth <double>**

**Modify Boundary_layer <id> continuity {yes | no}**

**set boundary_layer intersection volume <id> curve <ids> type {end, side, corner, reversal, default}**

**set boundary_layer intersection surface <id> vertex <ids> type {end, side, corner, reversal, default}**

*Sample Journal Files*

**Example 1**

reset
create surface rectangle width 10 height 3

```
create surface circle radius 5 zplane
surf 2 move z -10
create volume loft surface 1 2
delete surf 1 2
create boundary_layer 1
modify boundary_layer 1 uniform height 0.1 growth 1.2 layers 4
modify boundary_layer 1 add surface 4 volume 3 surface 5 volume 3 surface 6 volume 3
surface 7 volume 3
set boundary_layer intersection volume 3 curve 10 type side
set boundary_layer intersection volume 3 curve 12 type side
set boundary_layer intersection volume 3 curve 14 type side
set boundary_layer intersection volume 3 curve 16 type side
mesh vol 3
```

**Example 2**

```
reset
create surface rectangle width 2
cylinder radius 0.1 z 0.1
cylinder radius 0.02 z 0.1
section volume 2 xplane reverse
section volume 3 xplane
volume 3 move x 0.5
create volume loft surface 12 8
unite volume 2 3 4
volume 2 copy
volume 5 scale 0.3 0.3 1
volume 5 rotate -10 about z
volume 5 move x 0.55 y -0.1
volume 2 5 move x -0.25
imprint all
delete vol 2 5
surf all scheme trimesh
group "profile" add curve in surface 30 29
create boundary_layer 1
modify boundary_layer 1 uniform height 0.002 growth 1.2 layers 6
modify boundary_layer 1 add curve in group with name "profile" surface 28
curve in group with name "profile" size 0.02
surface 28 size 0.3
surface 28 sizing function linear neighbor 2
mesh surface 28
```

# Step-by-Step Tutorials

## Step-By-Step Tutorials

The purpose of this chapter is to demonstrate the capabilities of CUBIT for finite element mesh generation as well as provide a brief tutorial on the use of the software package. This chapter is designed to demonstrate step-by-step instructions for generating a simple mesh on a perforated brick.

The following activity demonstrates the basics of using CUBIT to generate and mesh a geometry. By following these steps, you will become familiar with the basics of the command-line and GUI interfaces without stopping for detailed explanations. All the commands introduced in this tutorial are documented in subsequent chapters on this manual.

Here are a few tips for the examples in the tutorial:

- Focus on the instructions preceded with "Step" numbers. These walk you through a series of explicit activities that describe how to complete the task.
- Refer to the screen shots and other pictures that show what you should see on your own monitor as you progress through the tutorial.
- In this tutorial, command line options will look like this:

```
cubit> <Your commands go here>
```

If you do not have the Graphical User Interface (GUI) version of CUBIT, follow the steps in the right column below, otherwise, proceed through the steps on the left:

| GUI | CL | |
|---|---|---|
| Overview | Overview | |
| Step 1 | Step 1 | Beginning Execution |
| Step 2 | Step 2 | Creating the Brick |
| Step 3 | Step 3 | Creating the Cylinder |
| Step 4 | Step 4 | Adjusting the Graphics Display |
| Step 5 | Step 5 | Forming the Hole |
| Step 6 | Step 6 | Setting Interval Sizes |
| Step 7 | Step 7 | Surface Meshing |
| Step 8 | Step 8 | Volume Meshing |
| Step 9 | Step 9 | Inspecting the Model |
| Step 10 | Step 10 | Defining Boundary Conditions |
| Step 11 | Step 11 | Exporting the Mesh |

### Additional Tutorials

ITEM Tutorial - A tutorial on the new ITEM wizard.

Power Tools GUI Tutorial - A tutorial on geometry decomposition and cleanup using the Power Tools on the new CUBIT GUI.

Decomposition Tutorial - A series of webcutting hints and suggestions for creating sweepable volumes on various models.
Geometry Cleanup Process Flow - A flowchart on geometry cleanup and defeaturing.

# item

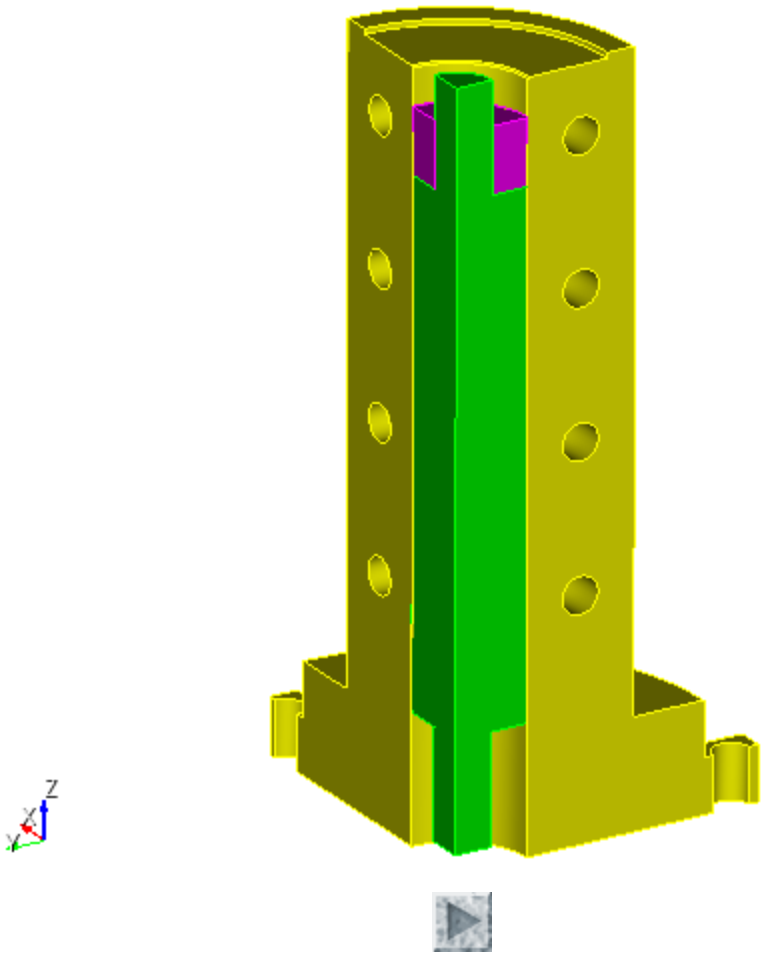
## ITEM Tutorial

*Overview*

This tutorial will demonstrate the use of the Immersive Topology Environment for Meshing (ITEM) to create a finite element mesh. ITEM is a wizard-like environment that guides a user through a typical mesh generation process from import to export. Each page in the ITEM workflow is linked to other pages, and one can easily move around in the environment by clicking on links on each page. Most of the pages contain diagnostic tools that search the model for specific geometry or mesh-related issues. Clicking on a entity in the ITEM output window will then generate specific command suggestions to resolve the problem. The following topics are included in this tutorial:

- Importing a geometry
- Creating the finite element model
- Removing small features
- Using merge tolerance to find and fix small misalignments
- Decomposing a model
- Generating a mesh
- Validating the mesh
- Creating boundary conditions
- Exporting the mesh

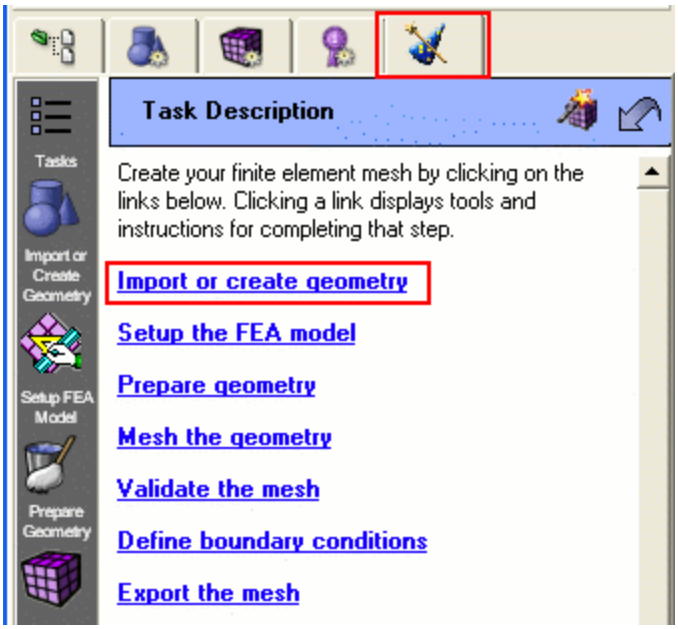
The model that will be used is shown below.




## ITEM Tutorial Step 1

*Step 1: Import Geometry*

- Click on **ITEM tab** on the power tools panel on the left hand side of the screen.
- Click on **Import or create geometry**




- Click on **Import a CAD model**
- Click on **Acis**
- Browse for the "item_tutorial.sat" file and import. The file may be in the Cubit directory/folder under 'tutorials.'
- Leave the default options selected on the import dialog box.
- Click **Done**
- Click **Done**

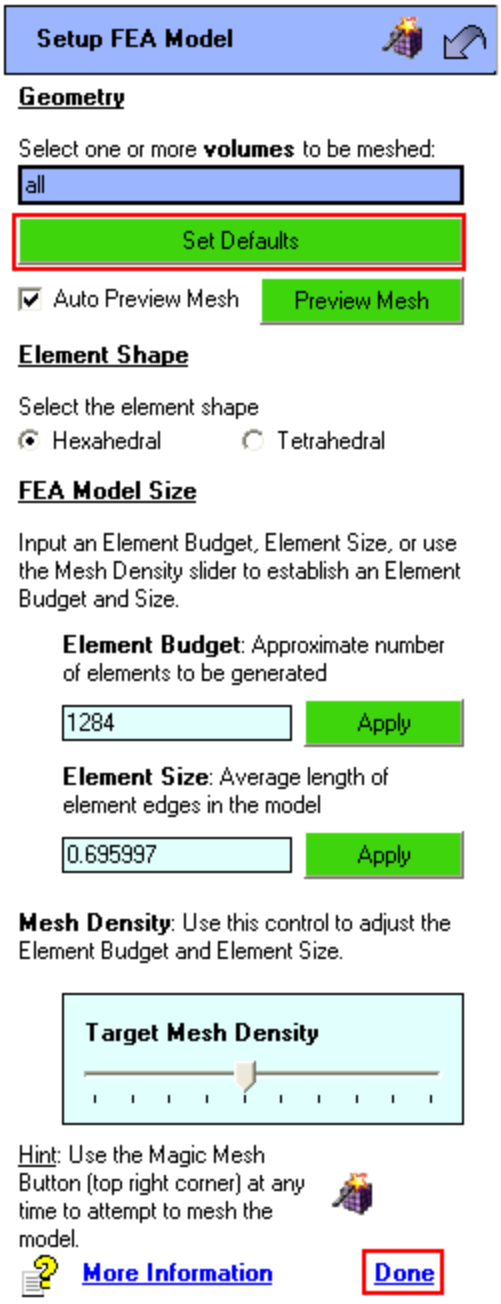
In most cases, clicking the Done button also acts like a "Back" button. Clicking Done will return the user to the previous page while preserving any changes made on that page.



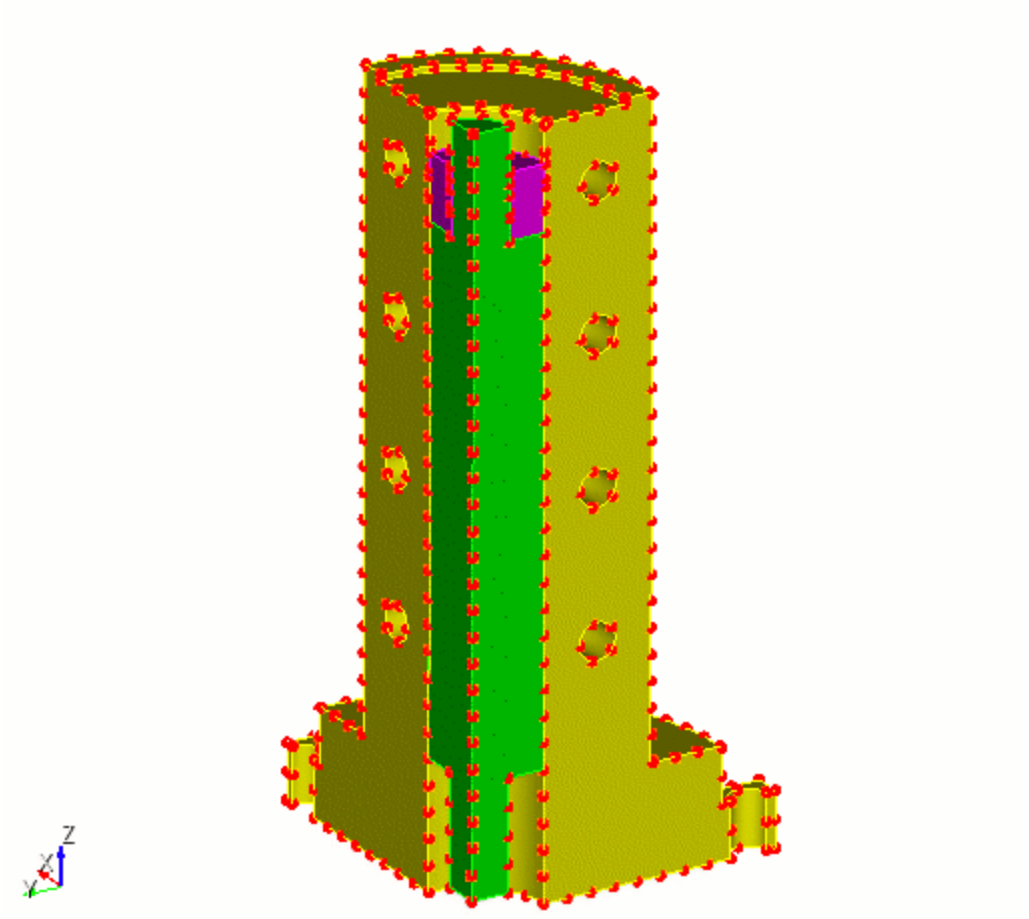
## ITEM Tutorial Step 2

*Step 2: Setup The FEA Model*

- Click on **Setup the FEA model**
- Click on **Set Defaults** (this determines a default mesh size and populates the fields accordingly-it also previews the mesh if the **Auto Preview Mesh** checkbox is checked)




- Examine the preview mesh




- Click **Done** to return to the previous page.
- Click **Done** again to return to the main ITEM task page



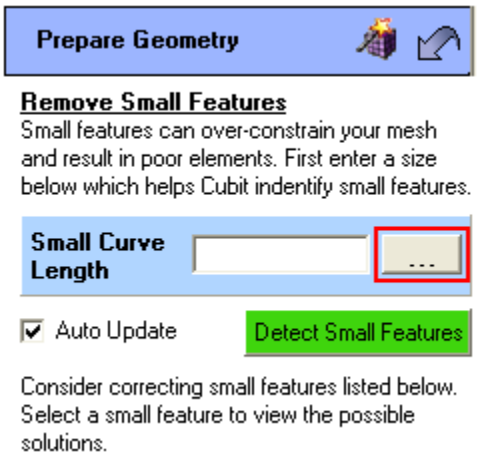
## ITEM Tutorial Step 3

*Step 3: Remove Small Features*

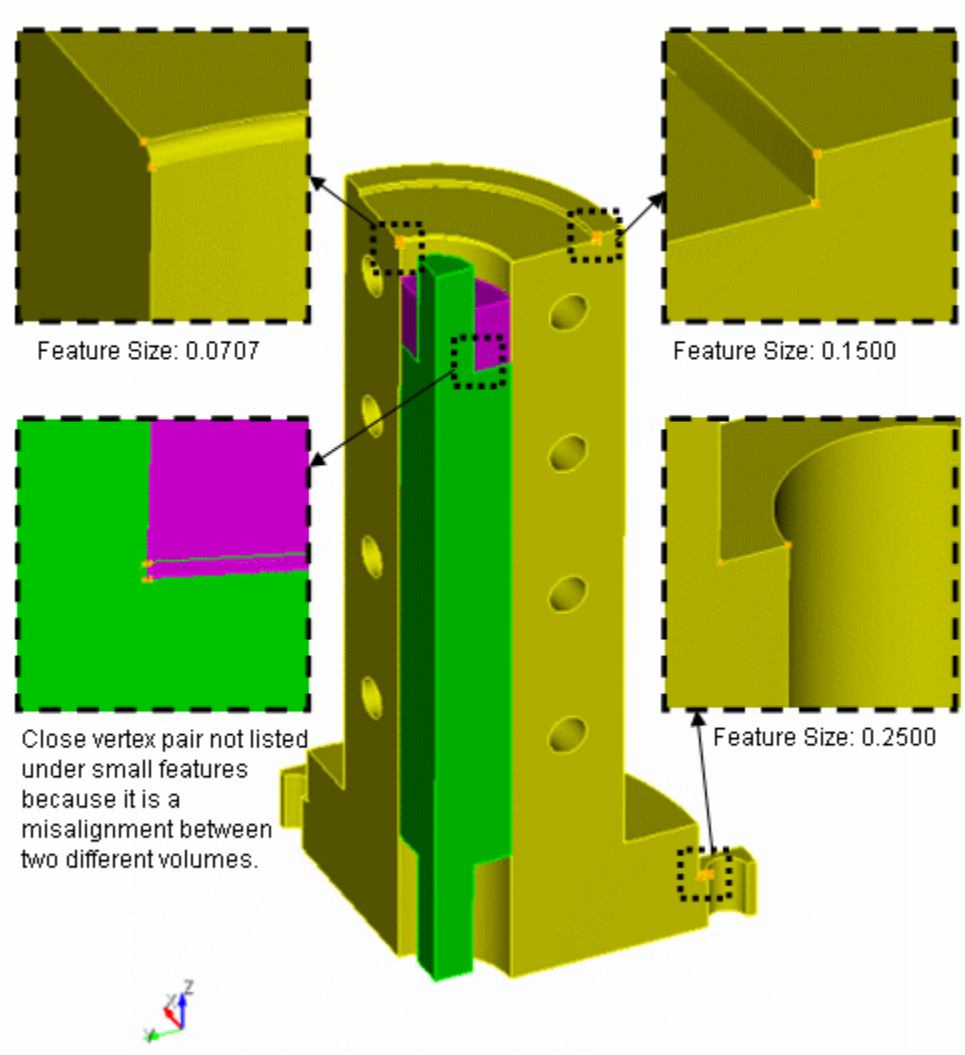
- Click **Prepare Geometry**
- Click **Run Checked Diagnostics**: The red exclamation point indicates a problem area that may need to be addressed




- Click on **Remove small features**
- Click on the button with three dots (...) next to the small curve length input box. This will open the small feature size panel. The small feature size panel is a tool to help the user find an appropriate small feature size. The smallest feature size is the smallest detail in the model that needs to be resolved in the mesh. It is also used for several calculations and diagnostic tools in the geometry ITEM panels. Any feature that is smaller than the smallest feature size will be flagged as a small curve or surface, and will need to be removed.

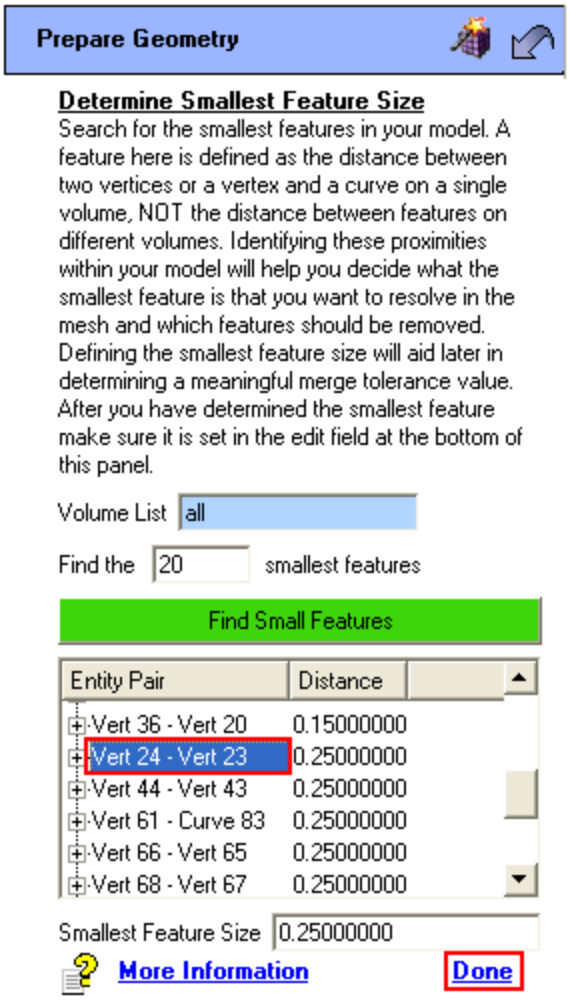


- Click on **Find small features**
- Change the Number to **20**
- Click on **Find Small features** again. One important thing to note is that the smallest features will only include vertex-vertex and vertex-curve pairs on the same volume. The small gaps and misalignments between volumes will be addressed during the imprinting and merging step.

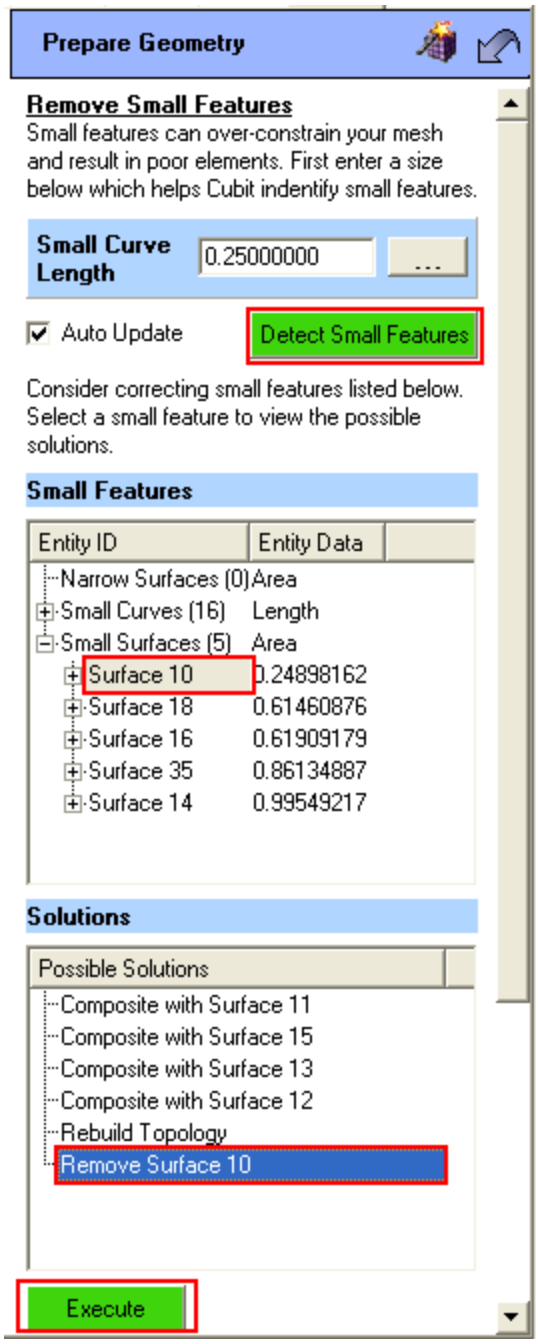



In addition to setting the small feature size, the smallest feature size panel of itself is a useful tool for visualizing and grouping small features. Sometimes it is useful just to have a list of the smallest features and a means of quickly visualizing and grouping them.

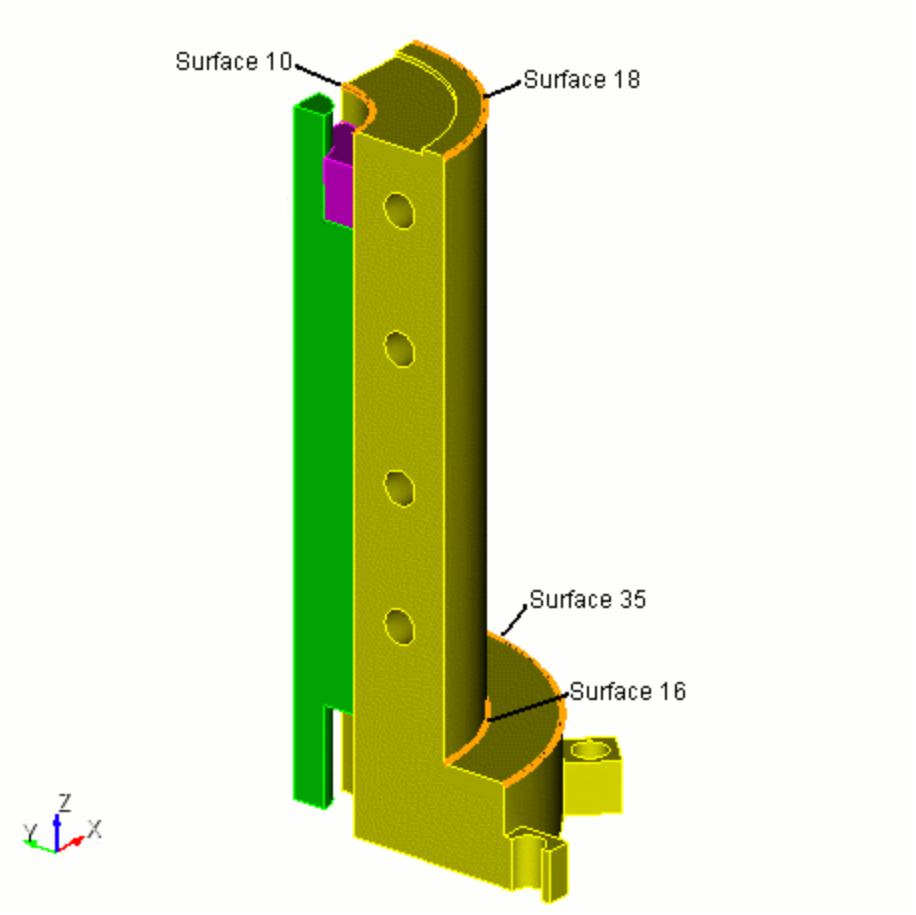
- Scan through the vertex-vertex pairs on the list. Right-click on any of the entities on the list and select **Zoom to Pair** or **Fly-in** to zoom in on the small feature.
- Locate pair **Vertex 75 and 74**
- Clicking on the pair should populated the small feature size input box and the bottom of the window.
- Click Done



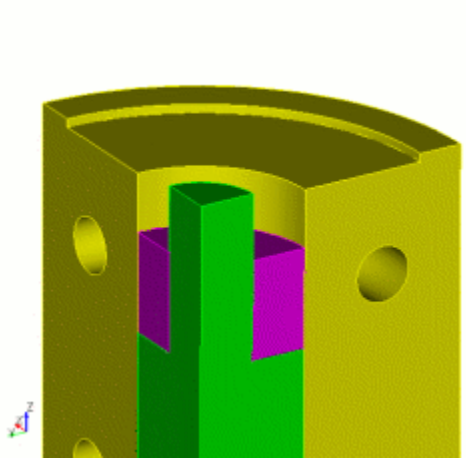

- You should now be on the **Remove Small Features** page. Click **Detect small features**. There should be 4 small surfaces on the list.
- Click on the plus sign to open the **Small Surfaces** group. The first four surfaces are small filleted surfaces that are easily removed.
- Click on **Surface 10**
- Click on **Remove Surface** from the Solutions window
- Click **Execute**

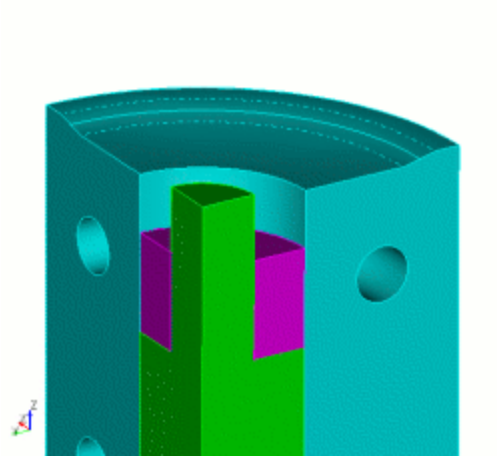



- Repeat for **Surface 18**, **Surface 16**, **Surface 35**




- On the 4th small surface execute the **Rebuild Topology** solution. The rebuild topology option will remove curves or surfaces from a model and reconstruct the geometry using real geometry operations. In this case, it replaces the sharp lip surface with a gradual surface, as shown in the following image.





- The rest of the small curves in the model are at the small feature size limit of 0.25. To remove these from the list, either change small feature size to a smaller value like 0.24, Or click **Mark ALL as Okay**
- Click **Done** to return to the main Prepare Geometry page

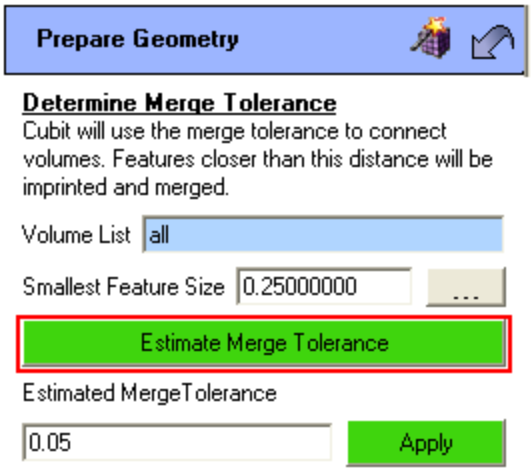


## ITEM Tutorial Step 4

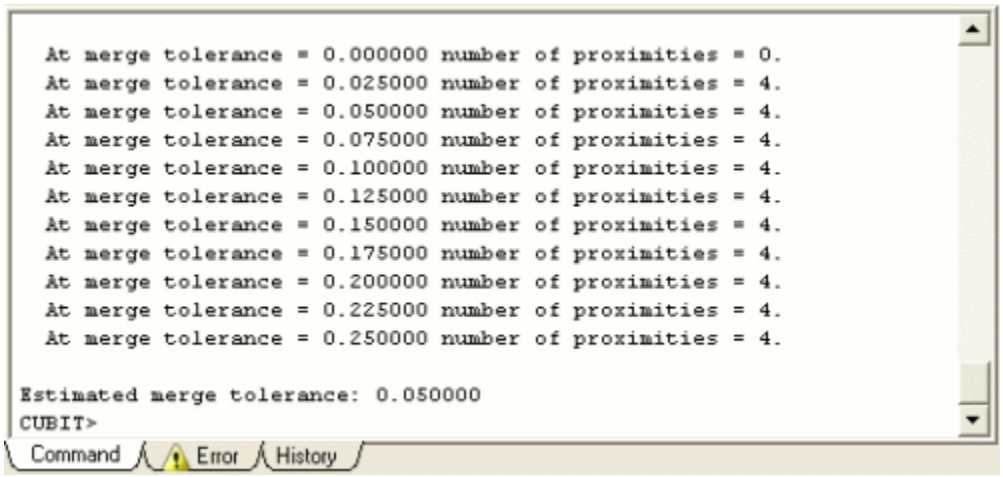
*Step 4: Connect Volumes*

The next step in the mesh generation process is to merge all shared curves and surfaces. This is necessary so that adjacent volumes can shared boundary meshes. For most geometries, this step presents no major complications. But in many cases, misalignments, tolerance problems, or other cleanup operations can prevent proper merging. The ITEM panel is designed to guide users through imprint/merge problems.

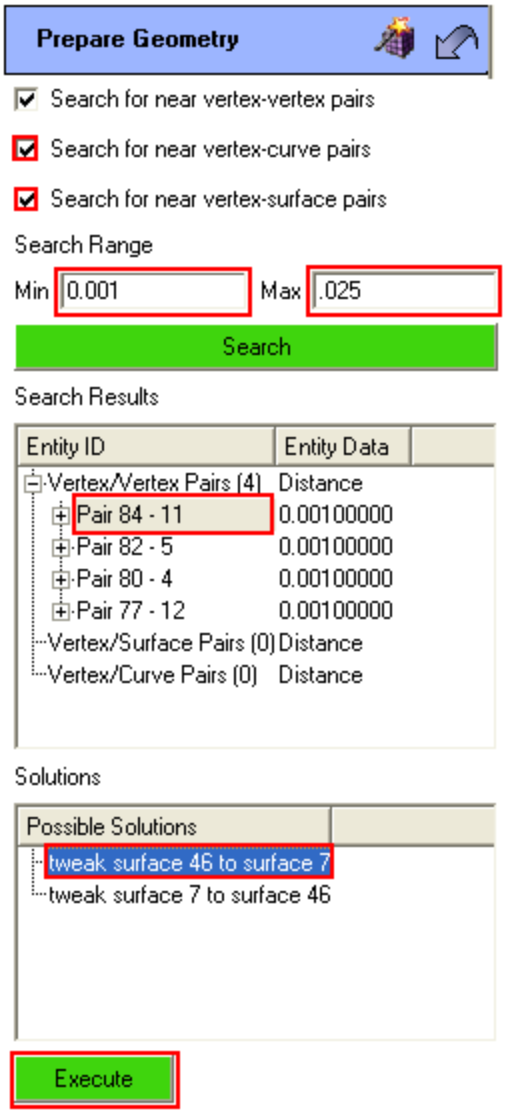
- Click **Connect Volumes**
- Click **Imprint and merge**
- Click on the button with three small dots **(...)** next to the Merge Tolerance field
- Click **Estimate Merge Tolerance**. The merge tolerance panel is used to help the user find an appropriate merge tolerance. In addition to determining a proper tolerance for merging, the merge tolerance can also be used as a diagnostic tool to find small misalignments, as will be demonstrated below. Many of these can be resolved prior to imprinting and merging.




- Check vertex-curve and vertex-surface pairs
- Look in the ouput of the command line workspace. A proximity is nearly coincident entities that would be merged at the given tolerance. From the given list, you can tell that there are 4 entities that would be merged at a merge tolerance of 0.025 which would not be merged if the merge tolerance were 0. This means that those entities are less than 0.025 apart.




- Change the search parameter to very small number like Min=0.001 and search again.
- Four **Vertex-Vertex Pairs** appear
- Open **Vertex/Vertex** pairs group
- Right click on first pair and choose **Label Pair** to view
- Click on the first solution (**tweak surface 46 to surface 7**) and click **Execute**




- Click **Done**
- Click **Imprint/Merge button**
- Click **Detect Potential Problems**
- .

No problems should appear on the list, signifying that imprinting and merging has most likely been successful. There are several diagnostic tools on this page that help to determine if imprint/merging has been successful. These include:

- **Overlapping Surfaces**- Surfaces that overlap, but are not merged.
- **Non-manifold curves**- Two curves that are merged but that don't have any merged surfaces.
- **Non-manifold vertices**- Two vertices that are merged, but do not share any merged curves.

- **Floating volumes**- Volumes that are not connected to any other volumes (meaning they are not merged)

All of these diagnostics could be run at any time but the results are most meaningful after an imprint/merge operation.

- Click **Done**
- Click **Done**

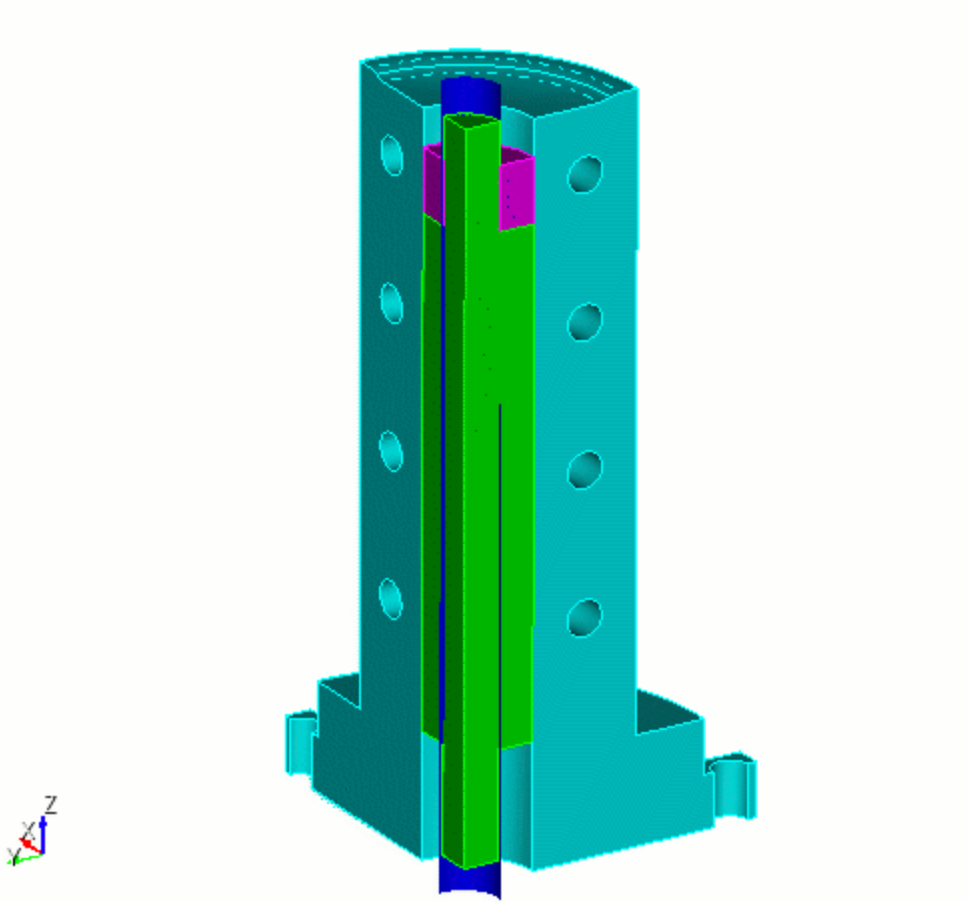


## ITEM Tutorial Step 5

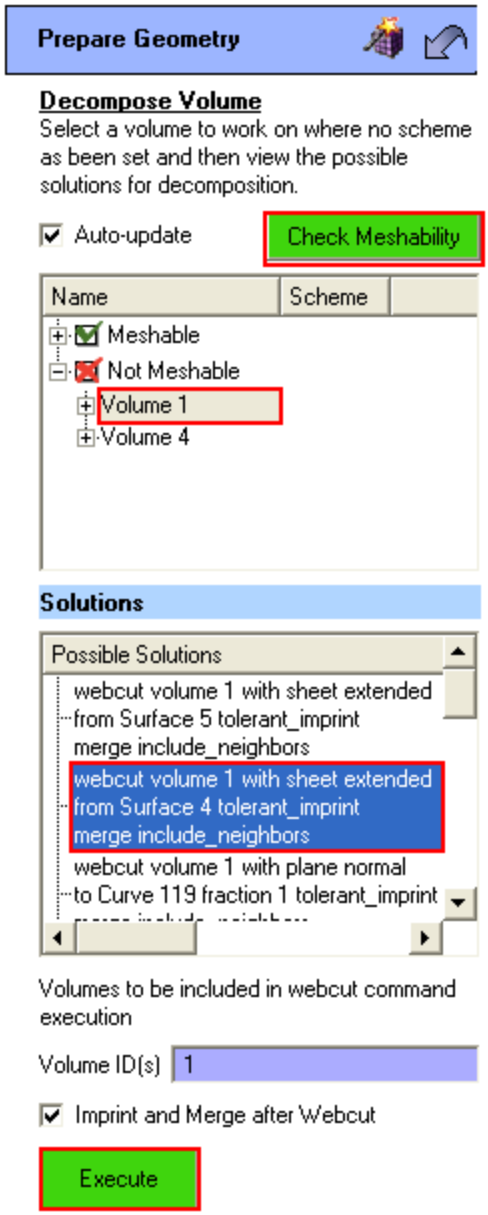
*Step 5: Build a Meshable Topology*

The next step in the mesh generation process can be one of the most challenging. Building a meshable topology involves decomposing an assembly into meshable parts. For sweeping, this means decomposing it into volumes composed of many-to-one and one-to-one sweepable parts. Each decomposed volume is further constrained because it needs to be able to share boundary meshes on merged surfaces. Since the number of possible decomposition strategies are numerous, it is not yet possible to automatically decompose most models. Instead, the ITEM framework seeks to provide possible decomposition options to the user, which they can be easily executed (and if necessary, quickly undone).

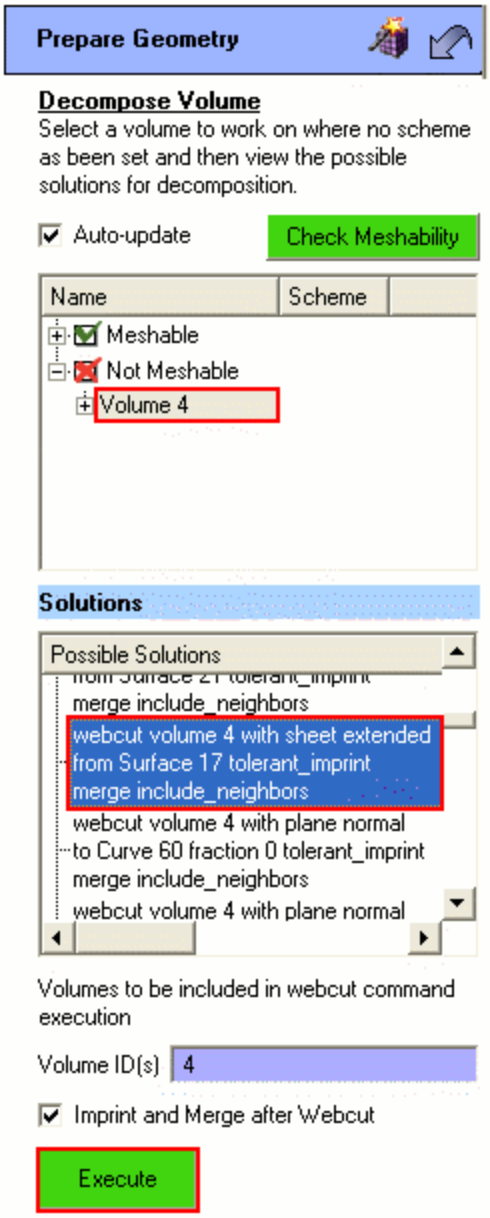
- Click **Build meshable topology**
- Click **Decompose volume**
- Click **Check Meshability**
- Examine **Volume 3**. There is 1 source and 1 target surface and it is sweepable.
- Examine **Volume 1**. There are two source surfaces and two target surfaces. It needs to be webcut cylindrically through middle

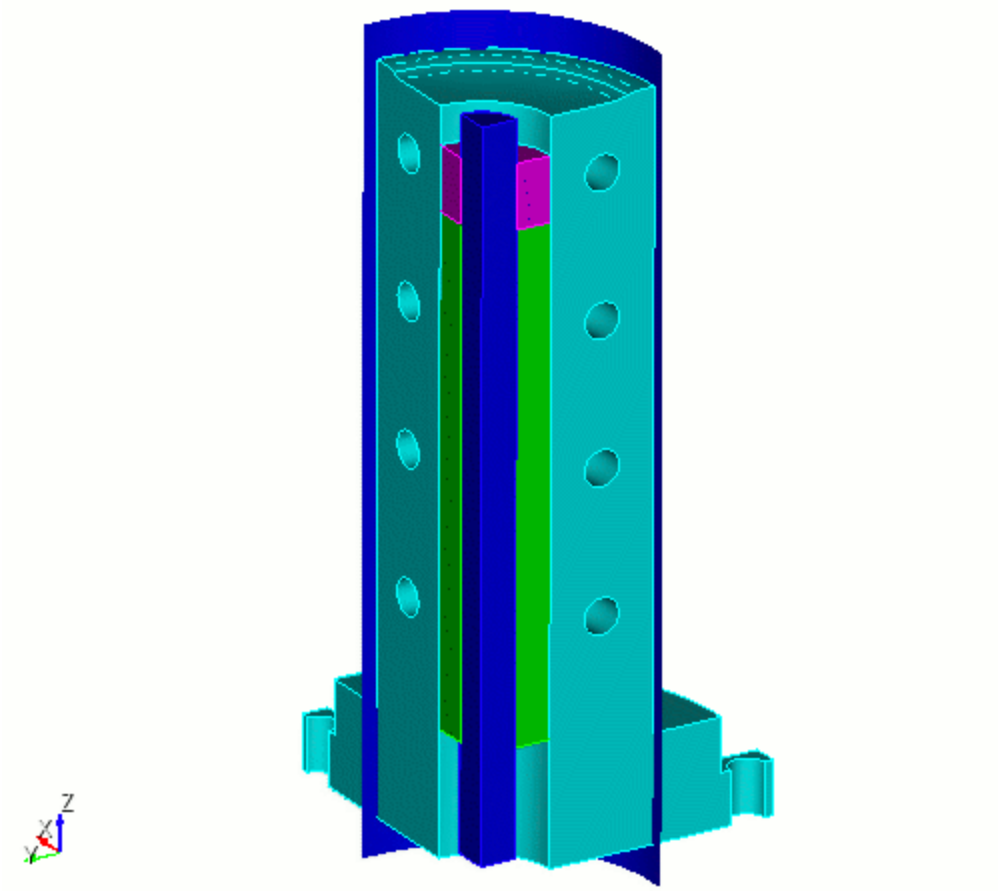



- Click on **Volume 1**
- Choose cylindrical webcut and execute making sure the imprint and merge after webcut is checked




- Examine **Volume 2**. The tall portion should be swept around in the direction of the holes. The bottom portion should be swept from top to bottom. It is many to one sweepable if webcut cylindrically around inner cylinder.
- Choose cylindrical webcut that cuts off thicker part at bottom of **Volume 4.**


Prepare Geometry
Decompose Volume
Select a volume to work on where no scheme as been set and then view the possible solutions for decomposition.
Auto-update
Check Meshability
Name
Scheme
Meshable
Not Meshable
Volume 4
Solutions
Possible Solutions
merge include_neighbors
webcut volume 4 with sheet extended from Surface 17 tolerant_imprint merge include_neighbors
webcut volume 4 with plane normal to Curve 60 fraction 0 tolerant_imprint merge include_neighbors
webcut volume 4 with plane normal
Volumes to be included in webcut command execution
Volume ID(s)
4
Imprint and Merge after Webcut
Execute



- Click **Done**.
- Click **Done** after running checked diagnostics again to verify.
- Click **Done**.

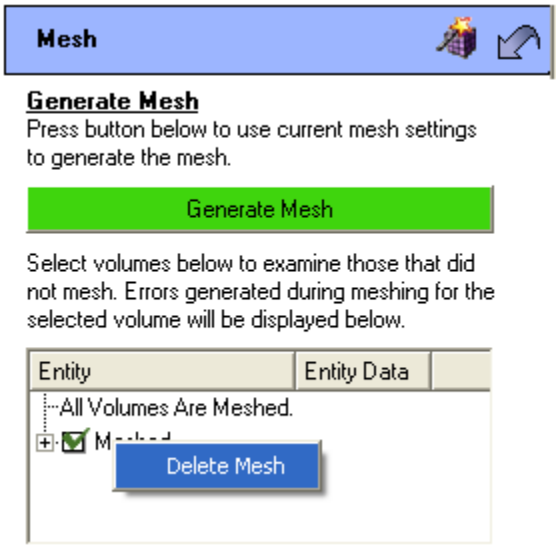
## ITEM Tutorial Step 6

*Step 6: Meshing the Geometry*

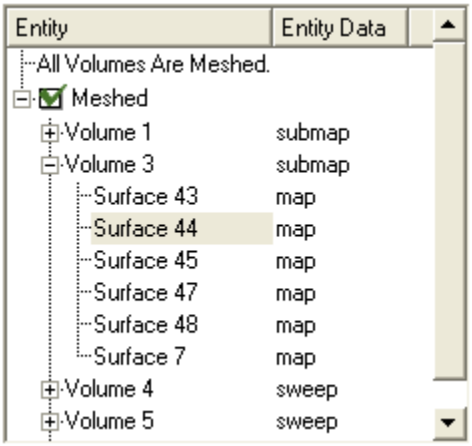
The actual mesh generation process is usually quite iterative. Rare is the case where meshing succeeds perfectly on the first try, even when all volumes are "meshable". Even if it does succeed, it is usually constrained by areas of poor quality elements. ITEM was designed to help users navigate the iterative mesh generation process. When meshing fails, the mesh generation panel helps to explain common error messages and suggest possible strategies for getting a model to mesh.

- Click **Mesh the geometry**
- Click **Generate Mesh**. Meshing succeeds in this case, but on closer inspection, it appears as though it has forced a mapped mesh on two surfaces that would be better as paved surfaces.

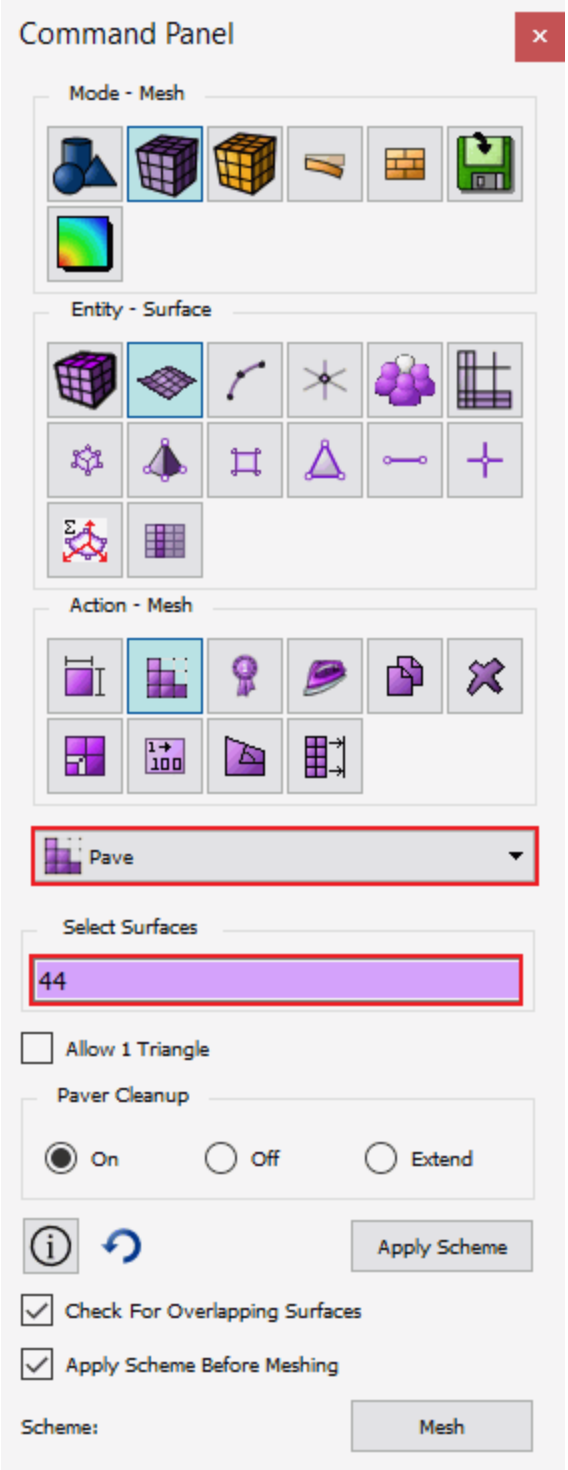
- Right click on **Meshed Volumes** and select **Delete Mesh**



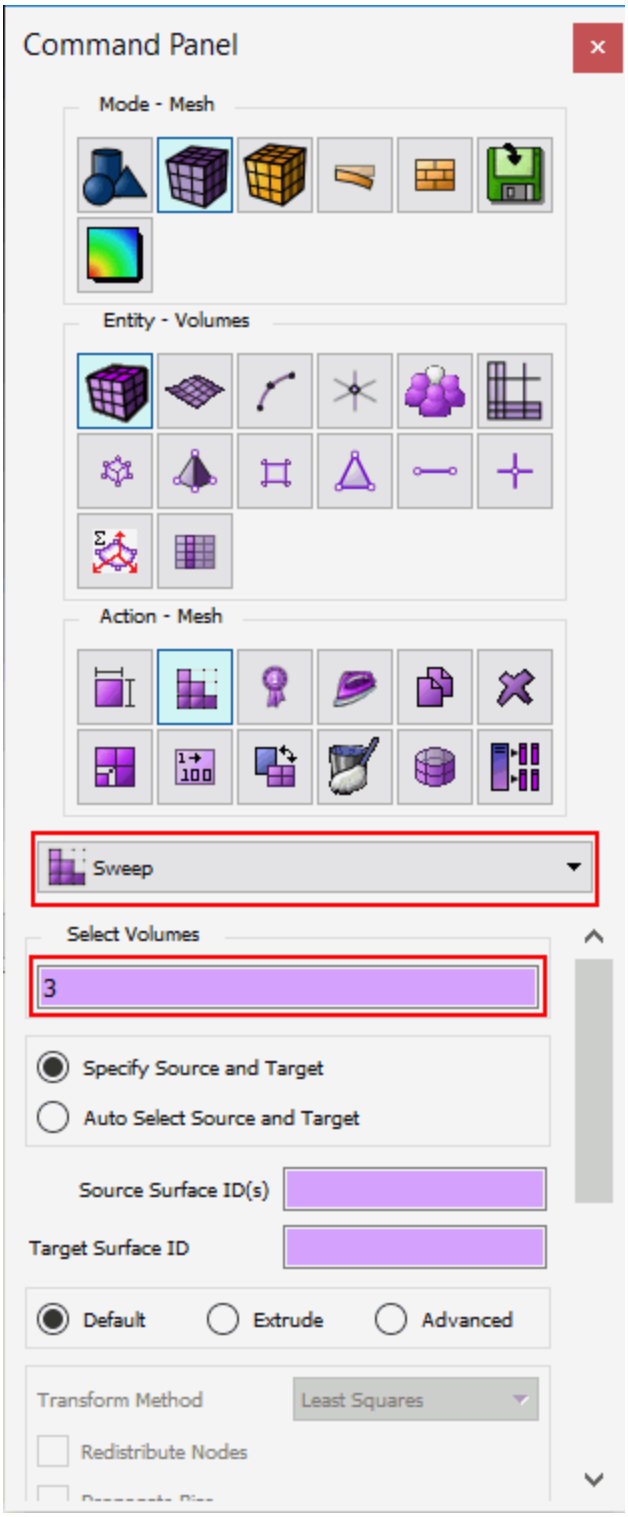

- Click on **Volume 3-Surface 44** - Notice the scheme is set to map




- Click on the **Meshing-Surface** command panel and set the scheme to pave for **Surface 44**



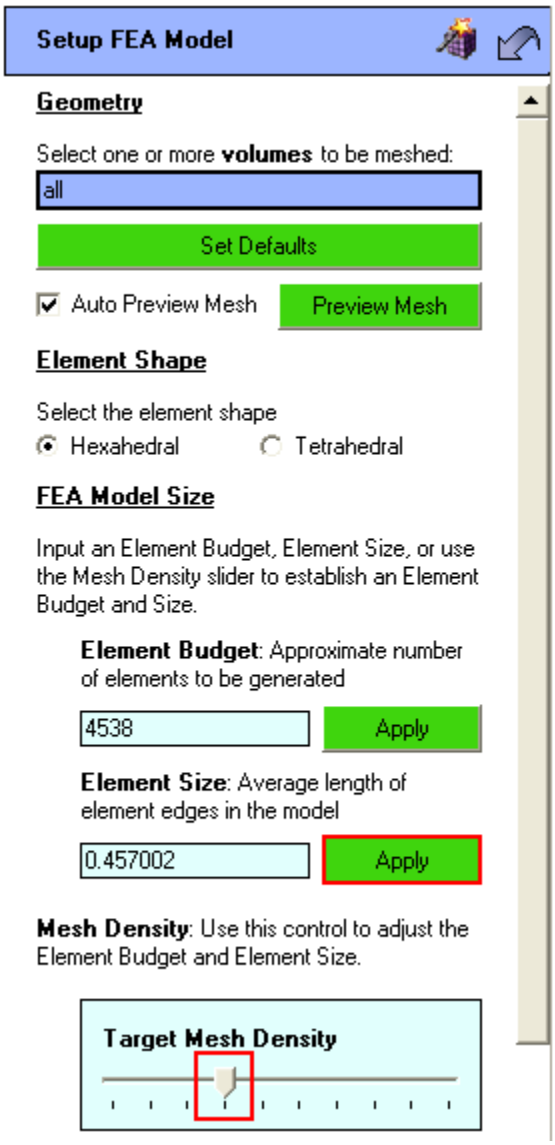

- Repeat for **Surface 20**
- Go to command panel for **Meshing-Volume** and select **Volume 3**. Change the scheme to **Sweep**.



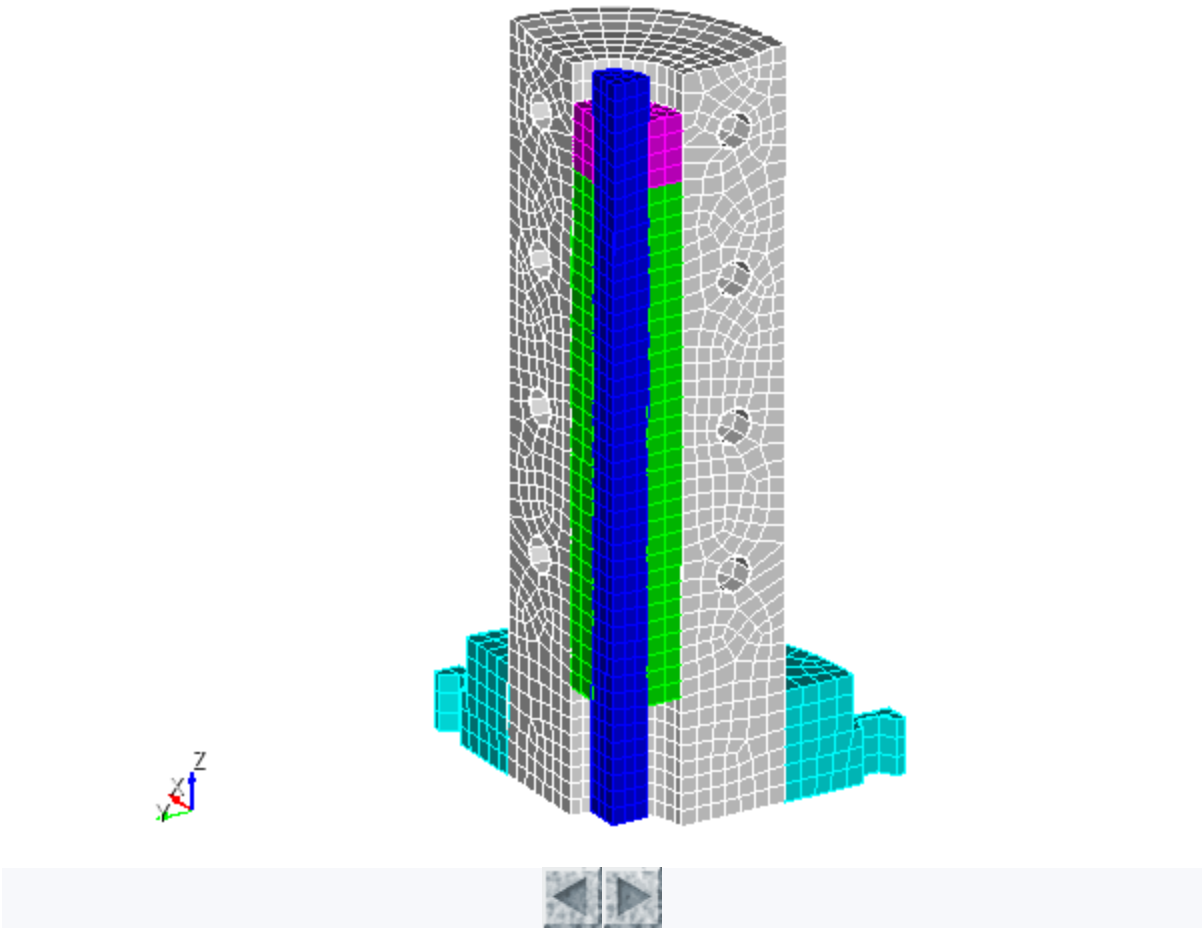

The mesh density didn't adequately capture the mesh features. To decrease the mesh size, return to the setup panel.

- Back on the ITEM Panel, click on **Setup the FEA model** button on the left panel.

- Change the mesh size by moving the target mesh density slider one position to the left.
- Click **Apply** next to the **Element Size**



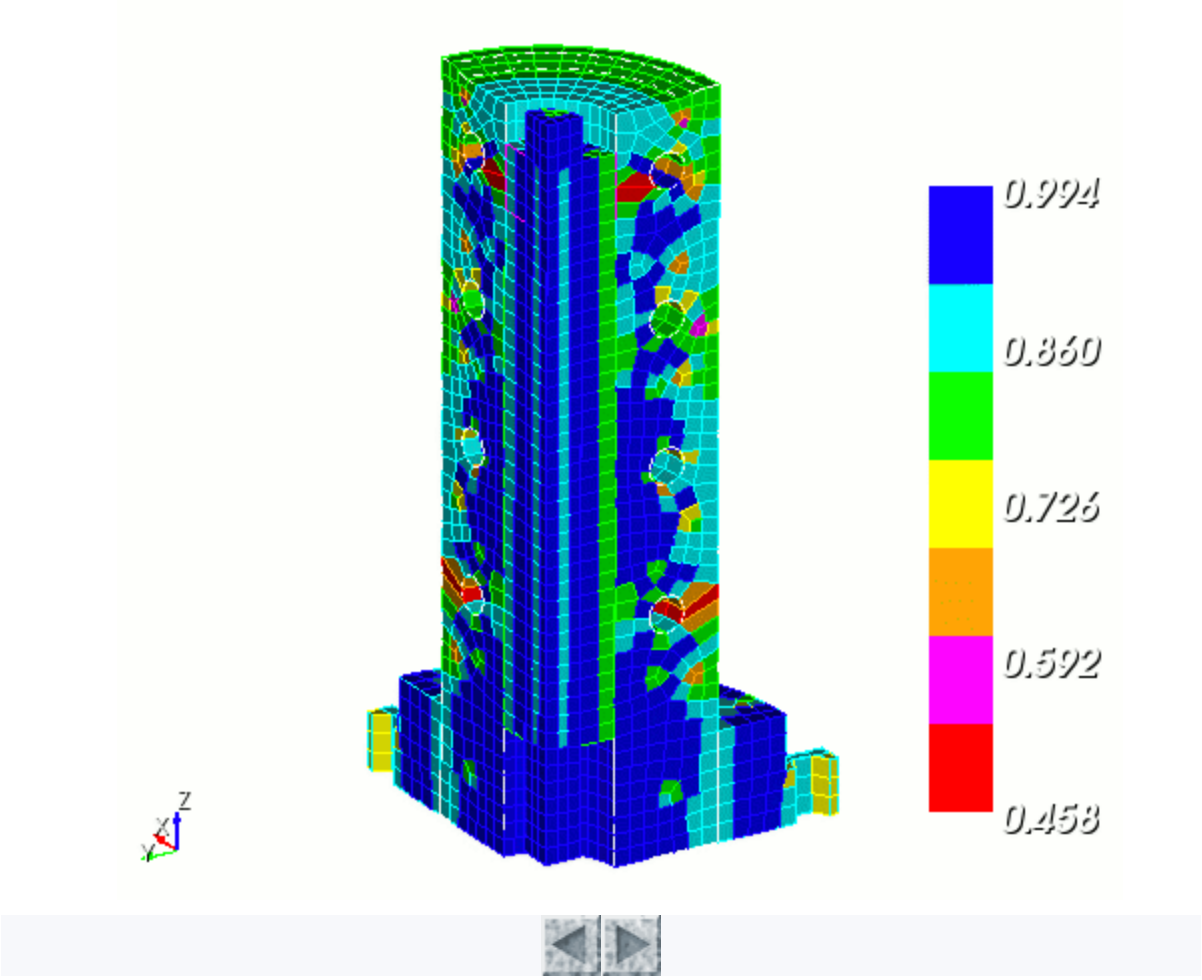

- Click on **Done**
- Click on **Mesh the geometry**
- Click on **Generate Mesh**




ITEM Tutorial Step 7

*Step 7: Validate Mesh*

- Click on **Validate Mesh**
- Click on **Check mesh quality**
- Click on **Analyze Quality** - No bad elements found

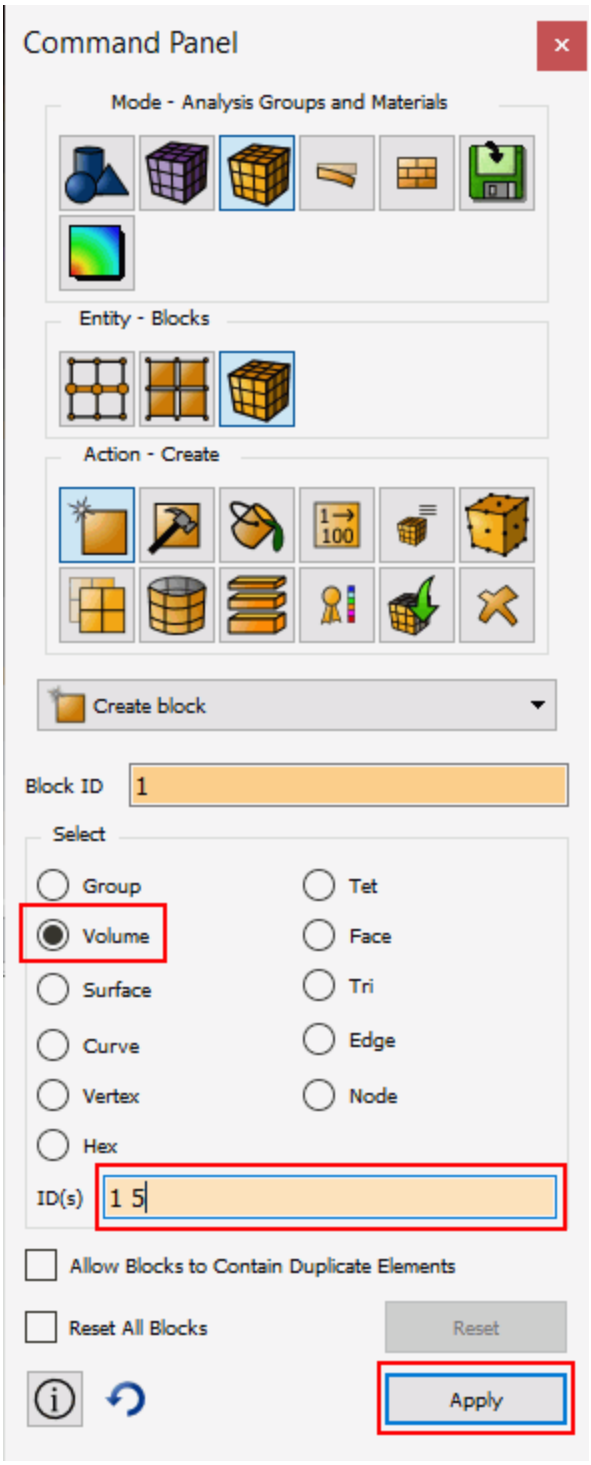



ITEM Tutorial Step 8

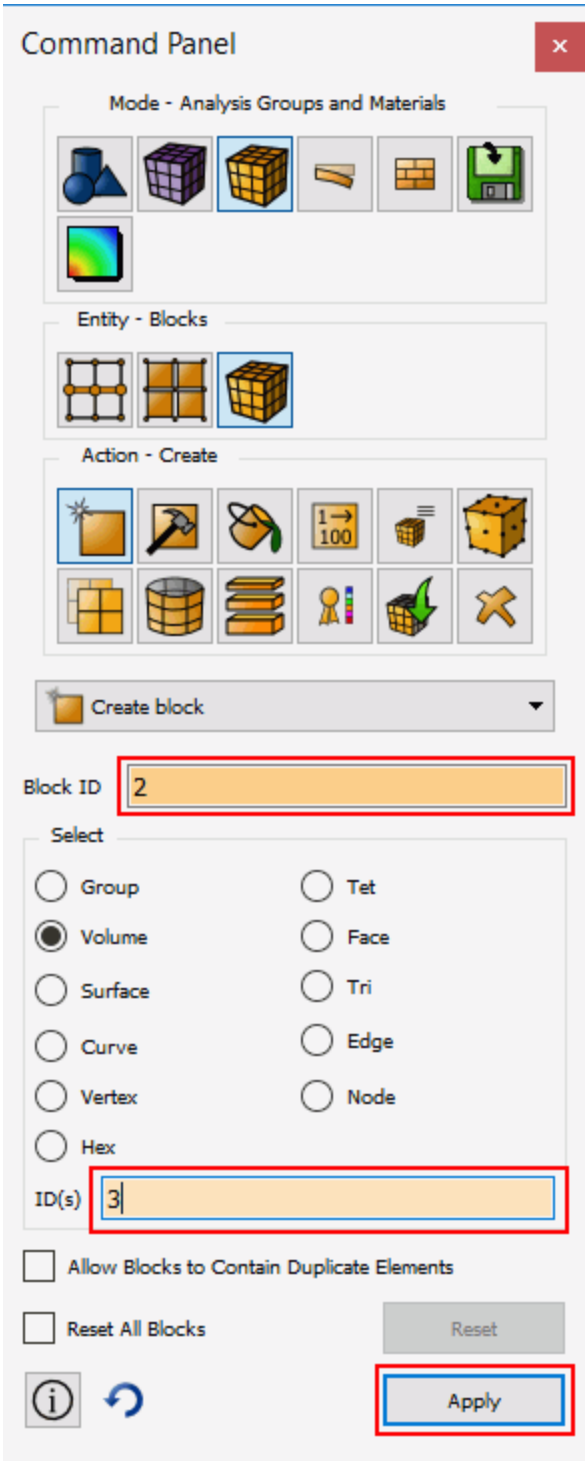
*Step 8: Define Boundary Conditions*

Exodus boundary conditions are specified as generic blocks, nodesets, and sidesets. Clicking on a boundary condition type on the ITEM panel will open the corresponding command panel.

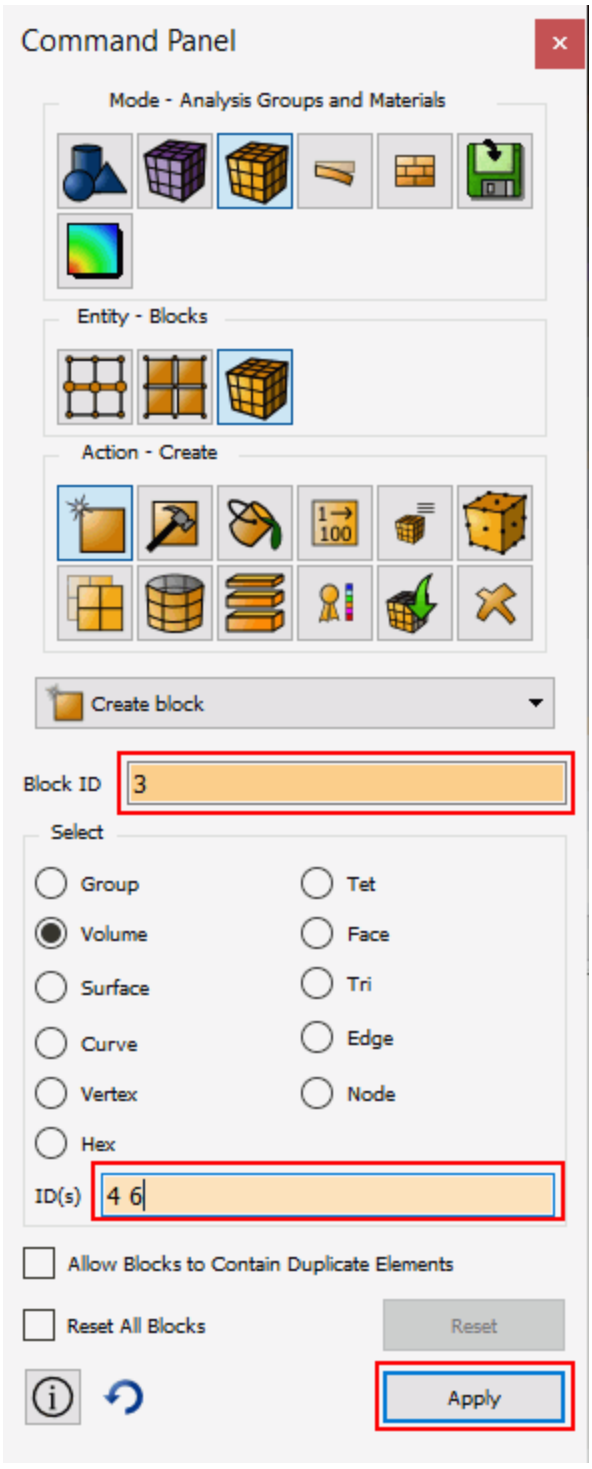
- Click on **Define boundary conditions** on the ITEM panel
- Click on **Define Block**. This opens the Material Properties->Block Panel and automatically assigns a default block id of 1.
- Set selection type to **Volume**
- Select **Volumes 1 5**
- Click **Apply**



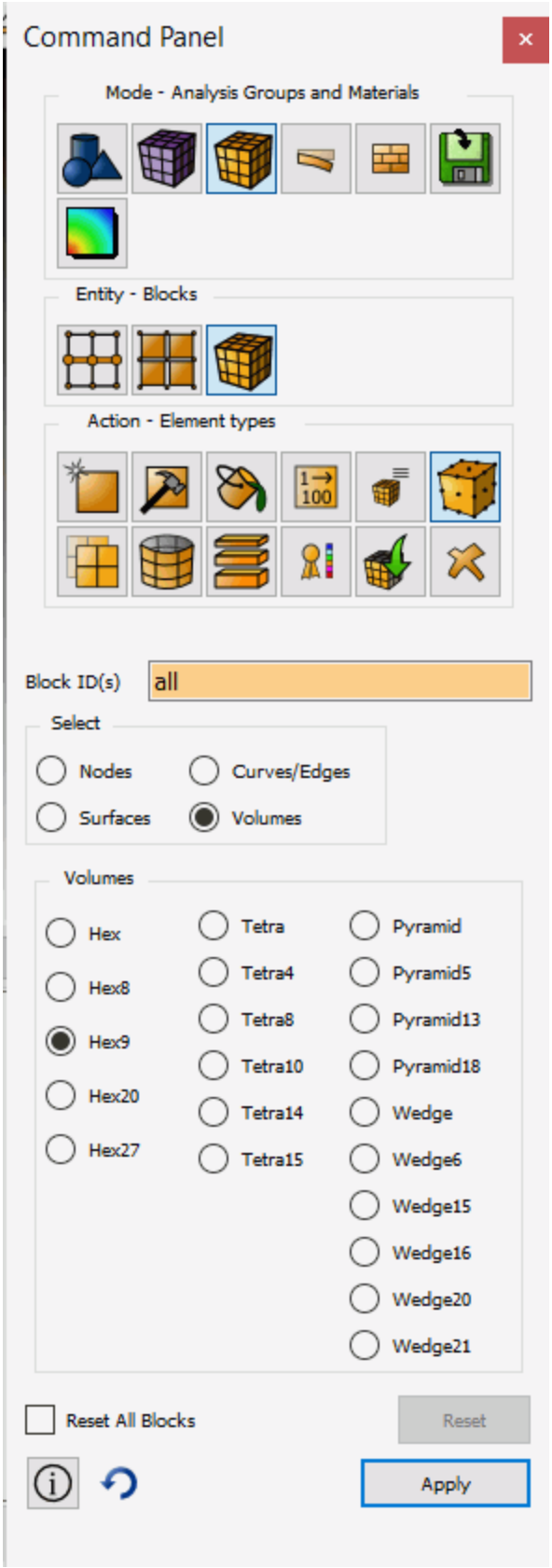

- Click on **Define Block** on the ITEM Panel. It should automatically increment the Block ID for you.
- Select **Volume 3**
- Click **Apply**




- Click **Create Block** on the ITEM Panel to increment block ID
- Select **Volumes 4 6**
- Click **Apply**




- Change to the **Element Type** panel by selecting the element type button
- Change the **Element Type** to **Hex9**
- Click **Apply**


Command Panel
Mode - Analysis Groups and Materials
Entity - Blocks
Action - Element types
Block ID(s)
all
Select
Nodes
Curves/Edges
Surfaces
Volumes
Volumes
Hex
Hex8
Hex9
Hex20
Hex27
Tetra
Tetra4
Tetra8
Tetra10
Tetra14
Tetra15
Pyramid
Pyramid5
Pyramid13
Pyramid18
Wedge
Wedge6
Wedge15
Wedge16
Wedge20
Wedge21
Reset All Blocks
Reset
Apply

- On the ITEM panel click **Done**

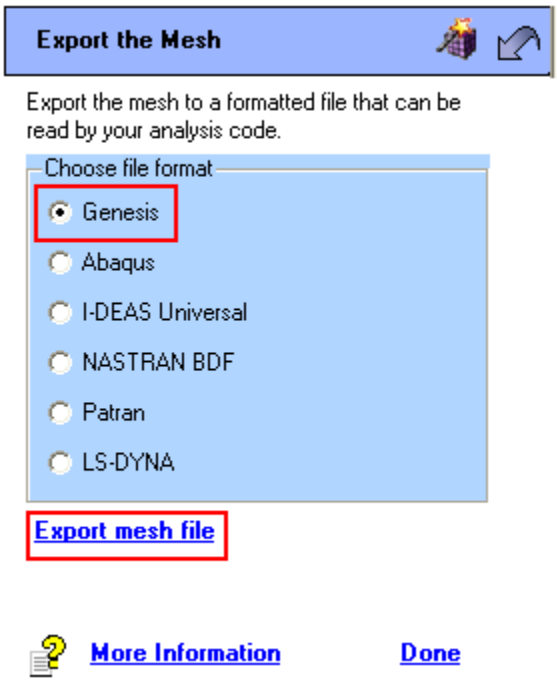


## ITEM Tutorial Step 9

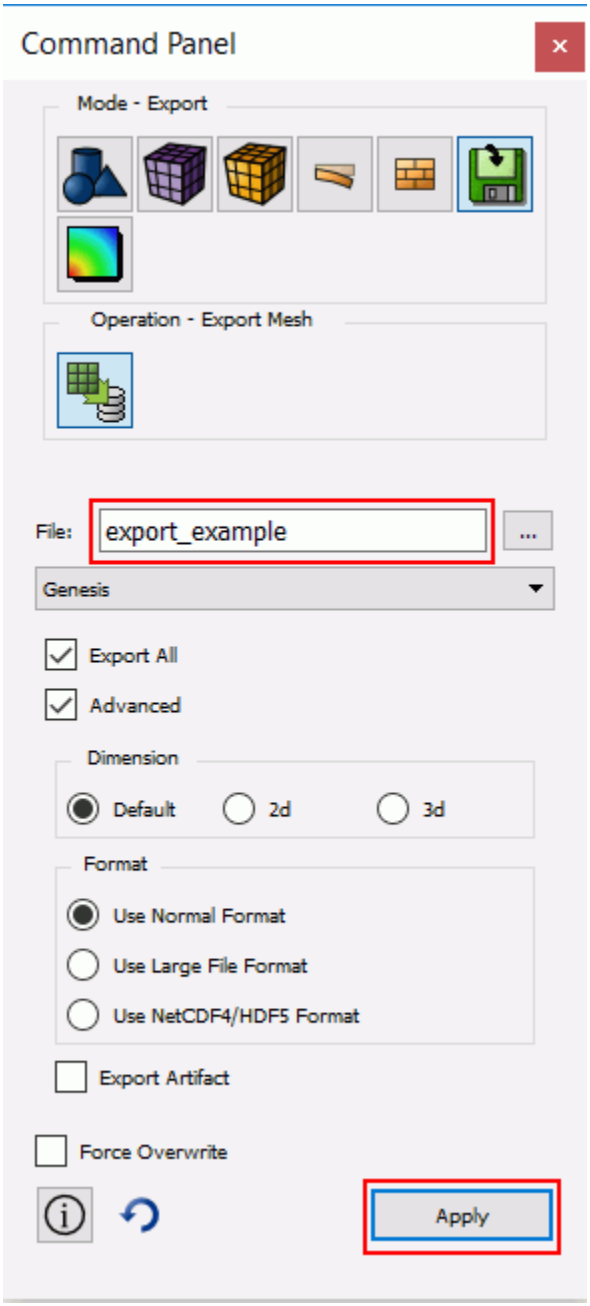
*Step 9: Export the Exodus Model*

Cubit primarily supports the Exodus format for mesh export. But there are also limited export abilities for other formats as well. For a list of export capabilities see Exporting the Finite Element Model

- Click on the **Export the Mesh** link from the main ITEM task page
- Set the **export type** to **Genesis**. This opens the export mesh dialog box on the command panels.




- Assign a filename
- Click **Apply** (all blocks will exported by default).




- Congratulations on completing the ITEM tutorial. Click on the arrow to return to the main tutorial page.



# power tools

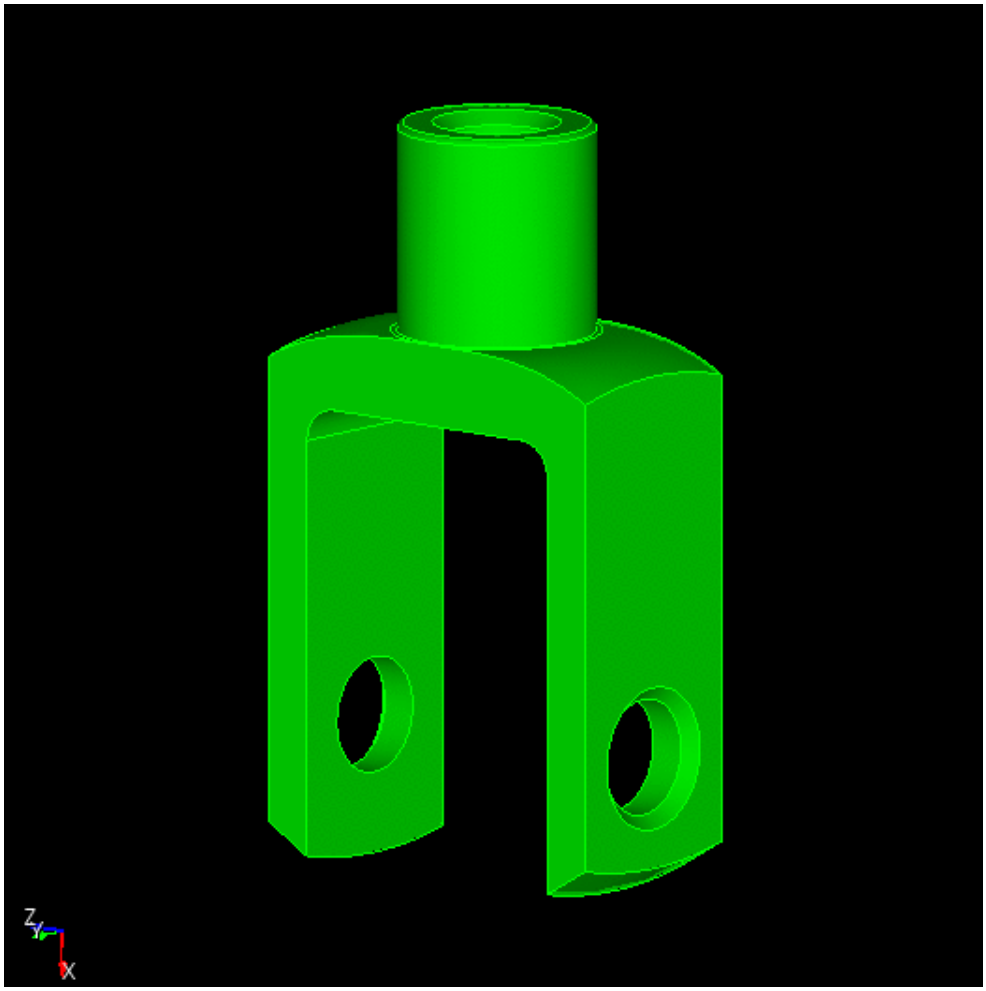
Power Tools GUI Tutorial

# *Overview*

This tutorial demonstrates using the Power Tools on the CUBIT GUI for geometry decomposition and cleanup. The following features will be covered:

- Importing Geometry
- Analyzing Geometry
- Geometry Power Tools
- Webcutting
- Imprint/Merge
- Mesh Power Tools
- Meshing

Each of these steps is described in detail in the following sections. For this tutorial you will need to have a basic understanding of the CUBIT GUI functionality, including how to select entities, maneuver in the graphics window, operate the Control Panel, and use toolbars. If you have not already done so, we recommend completing the Basic Tutorial first. The following image shows the geometry that will be used for this tutorial.
**NOTE**: Many of the steps in this tutorial include operations on specific entities which are identified by ID. When the solid modeling kernal is updated in Cubit the ID space may change. As such, you may not be able to rely on the IDs specified in this tutorial. Please look at the associated graphics to determine which entity/ID is being referred to.

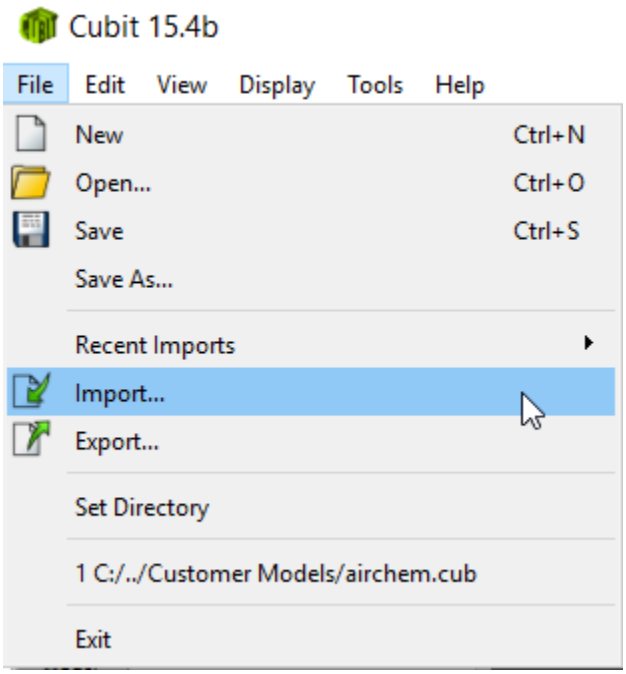



Power Tools GUI Tutorial Step 1

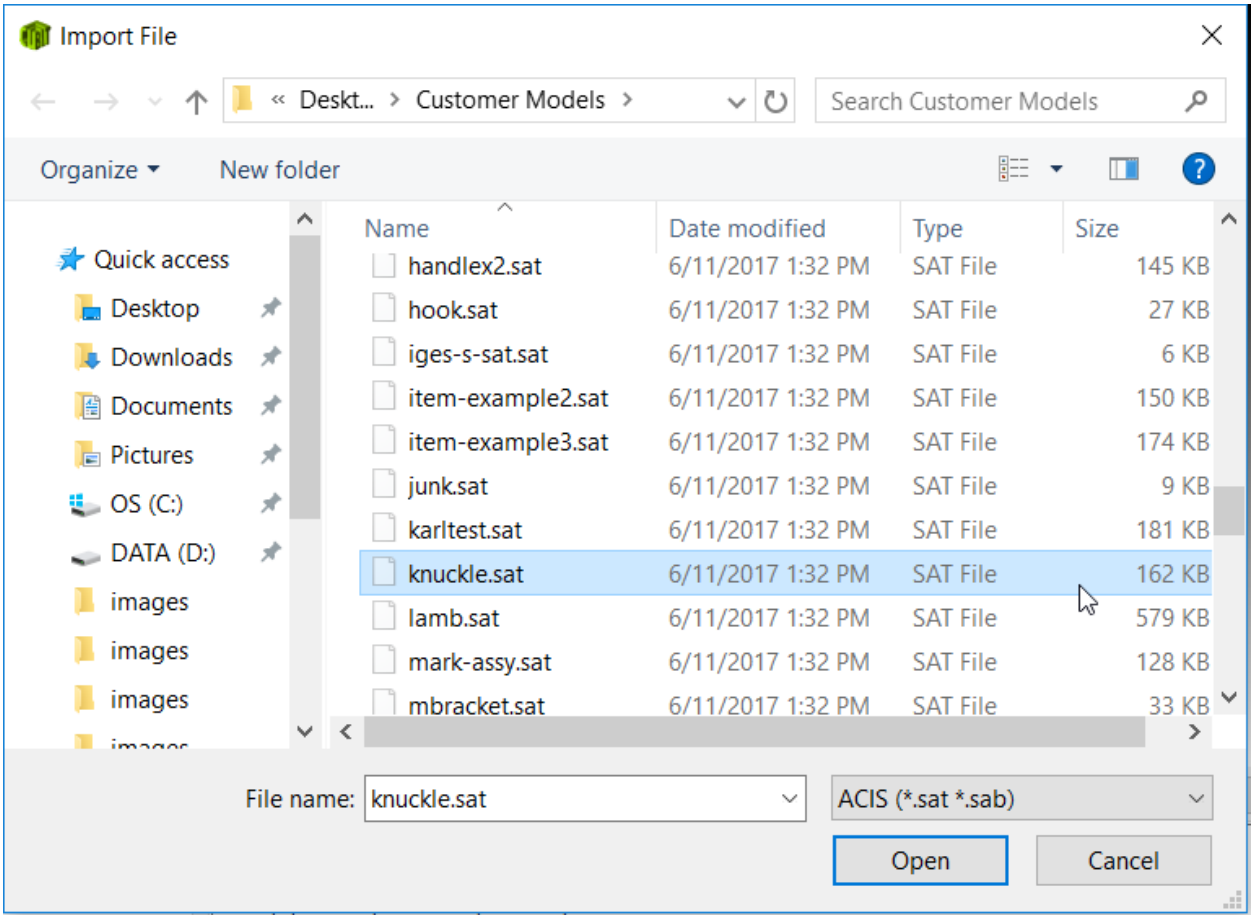
# *Step 1: Import the Geometry*

Begin by opening a new session of CUBIT. To complete this tutorial, you will need to download the ACIS file that contains the geometry definition.

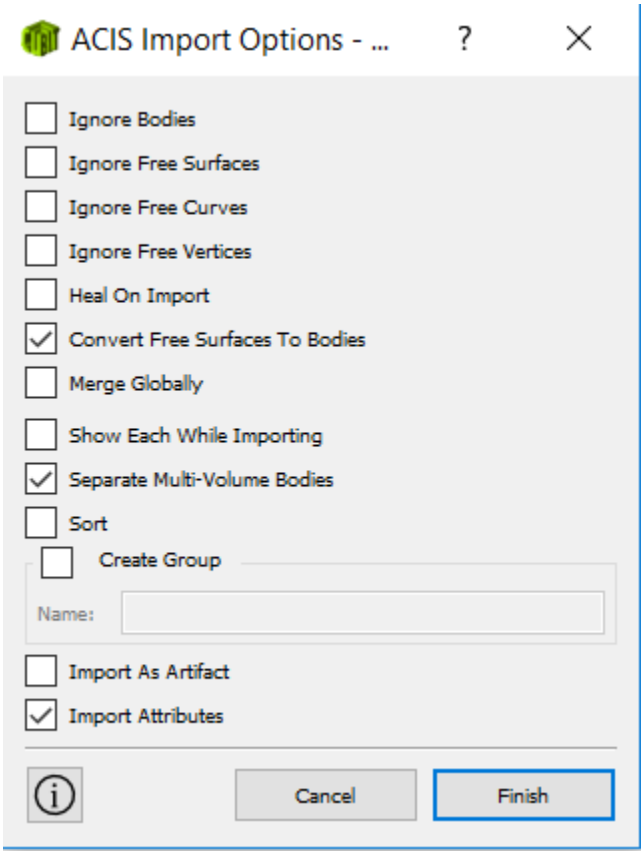
- Download geometry file knuckle.sat (**Note**: This link will not work from within Cubit. You will need to access this documentation from the cubit web site, or locate the file on your computer. It is included in the distribution of CUBIT under components\cubit\help\step_by_step_tutorials\power_tools)
- Select the **Import** option from **File** menu



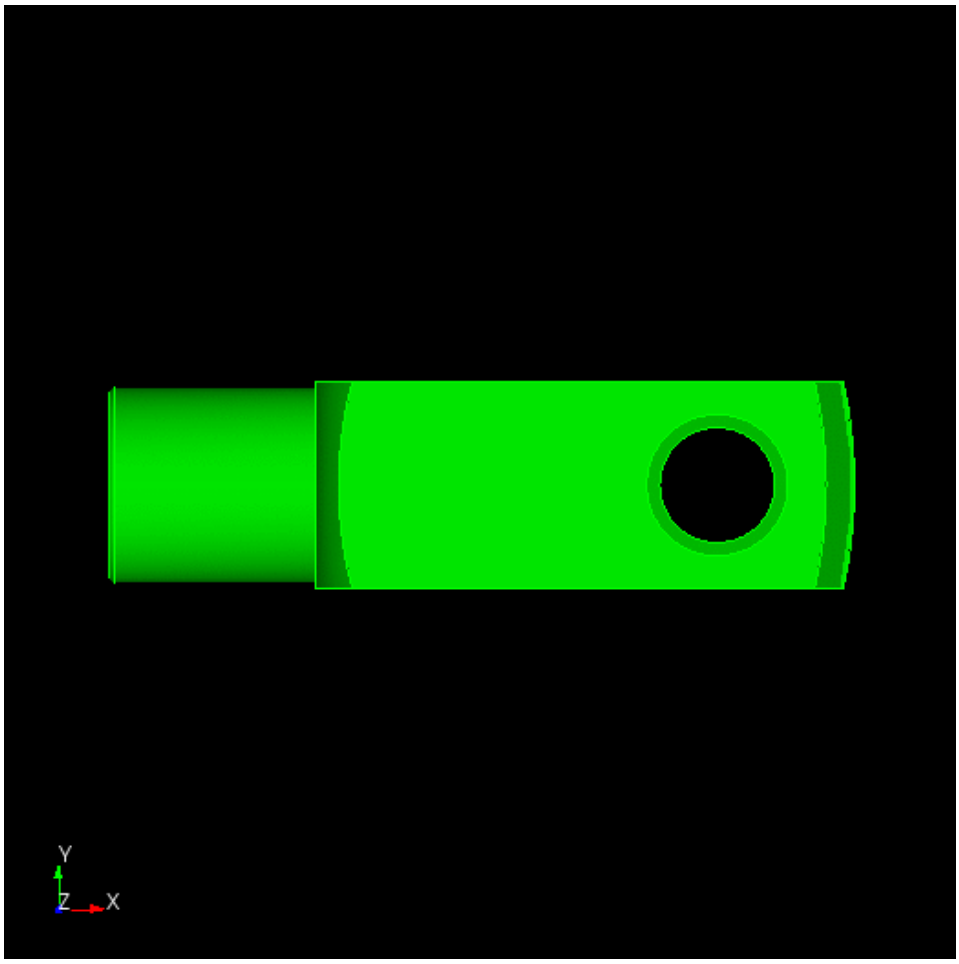

- The following dialog box will appear. Open the file by clicking on the name and selecting **Open**. If you do not see the file, make sure that you are in the right directory, and that the file type is set to ACIS.



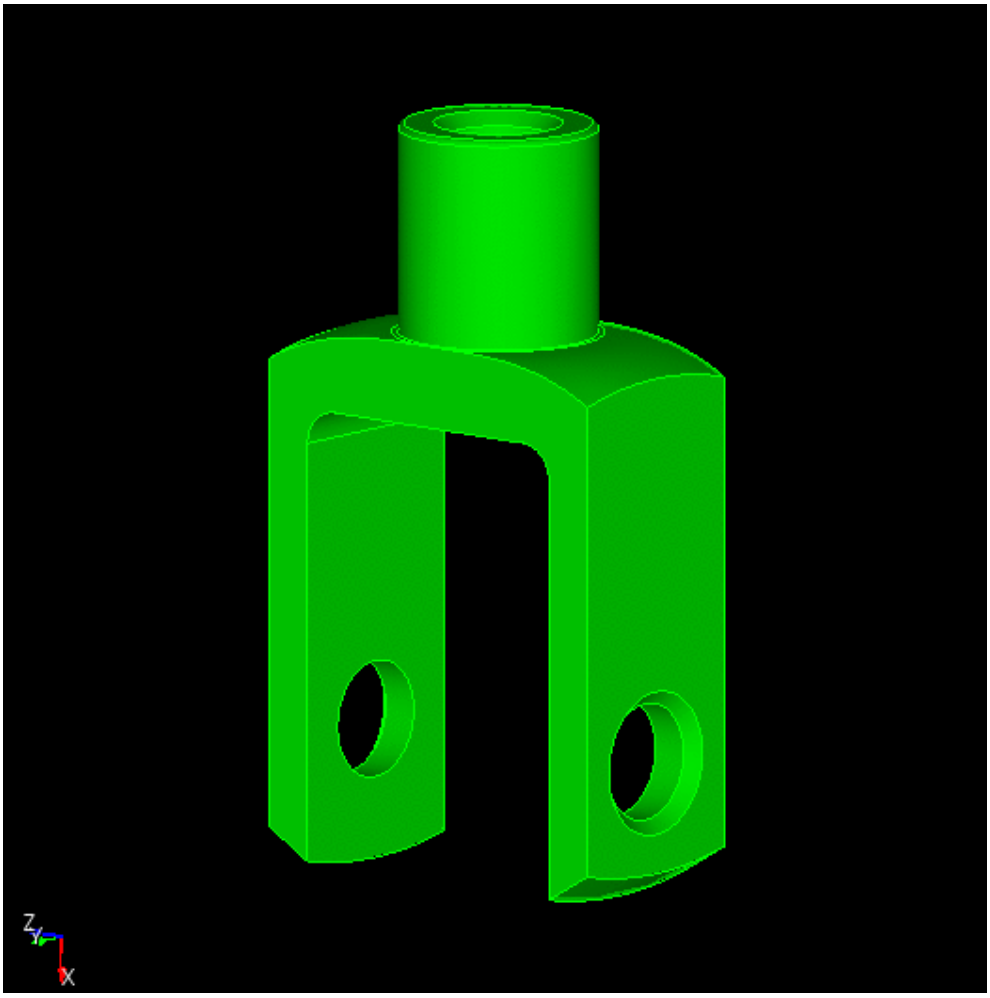

- Leave all of the import settings on their default settings and select **Finish**




Your graphics window should now appear as follows:



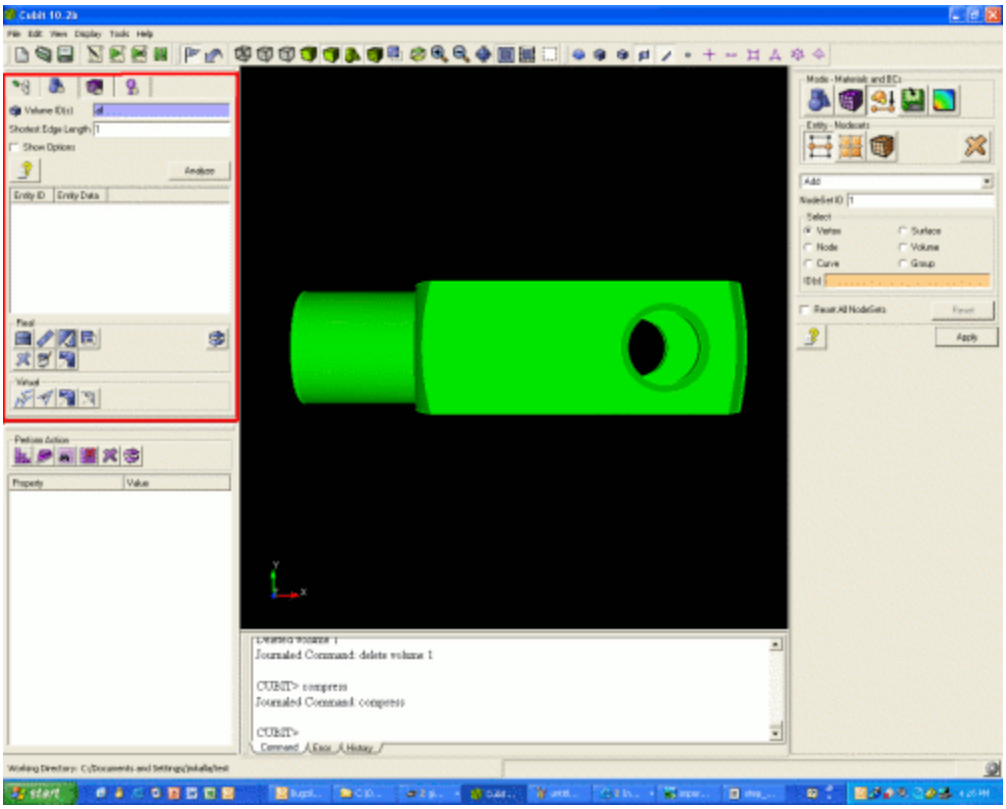

- Use the mouse to rotate the image in the graphics window to get a better perspective. For help with using the mouse in the graphics window, see Mouse Based Zoom, Pan and Rotate .




Power Tools GUI Tutorial Step 2

# *Step 2: Analyze the Geometry*

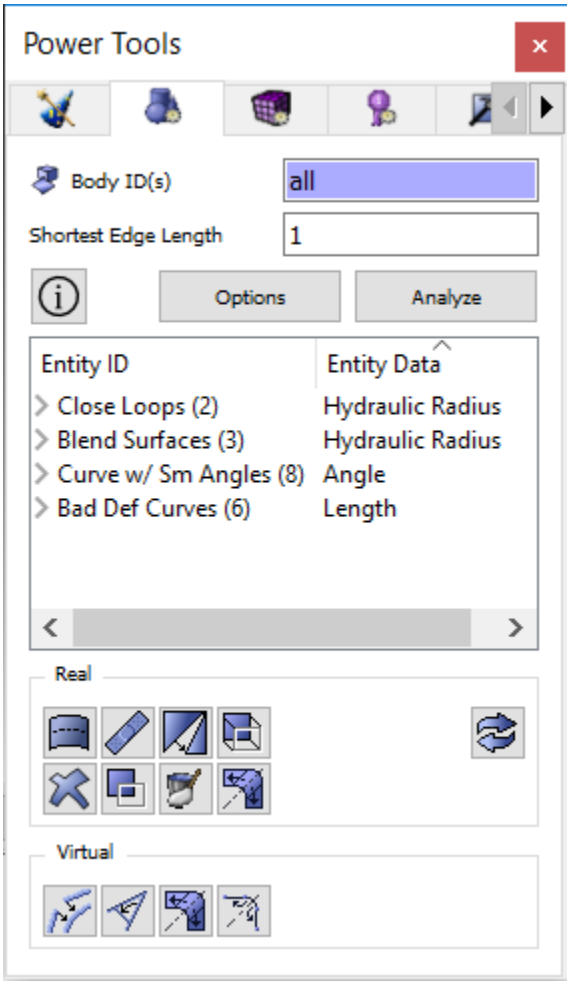
The Geometry Power Tools are located in the Entity Tree Window under the blue geometry tab. This menu provides access to many of the geometry analysis and clean-up tools in CUBIT.



Many geometries that are imported from other solid modeling software contain inconsistencies or small gaps that can cause meshing to fail. These problems are the result of differences in tolerances, file transfer loss, or inherent limitations in the parent system. In other instances, the geometry has no inconsistencies, but may be unsuitable for meshing because of topology such as small angles, overlap, or features smaller than the desired meshing size. The geometry analysis tool will analyze the volumes and return a list of suspected problems. To see a list of analysis options, click the "Show Options" box below the Analyze button.

Many of these problems can be fixed using the tools on the Power Tools menu. These include Split Surface, Heal, Tweak, Remove, Merge, Composite, Collapse Angle, Collapse Curve, and Collapse Surface. Many of these tools will be demonstrated in this tutorial.

- Open the **geometry repair** tab in the Entity Tree window
- Type all in the **Volumes to Analyze** field
- Set the **Shortest Edge Length** to 1
- Press **Analyze**



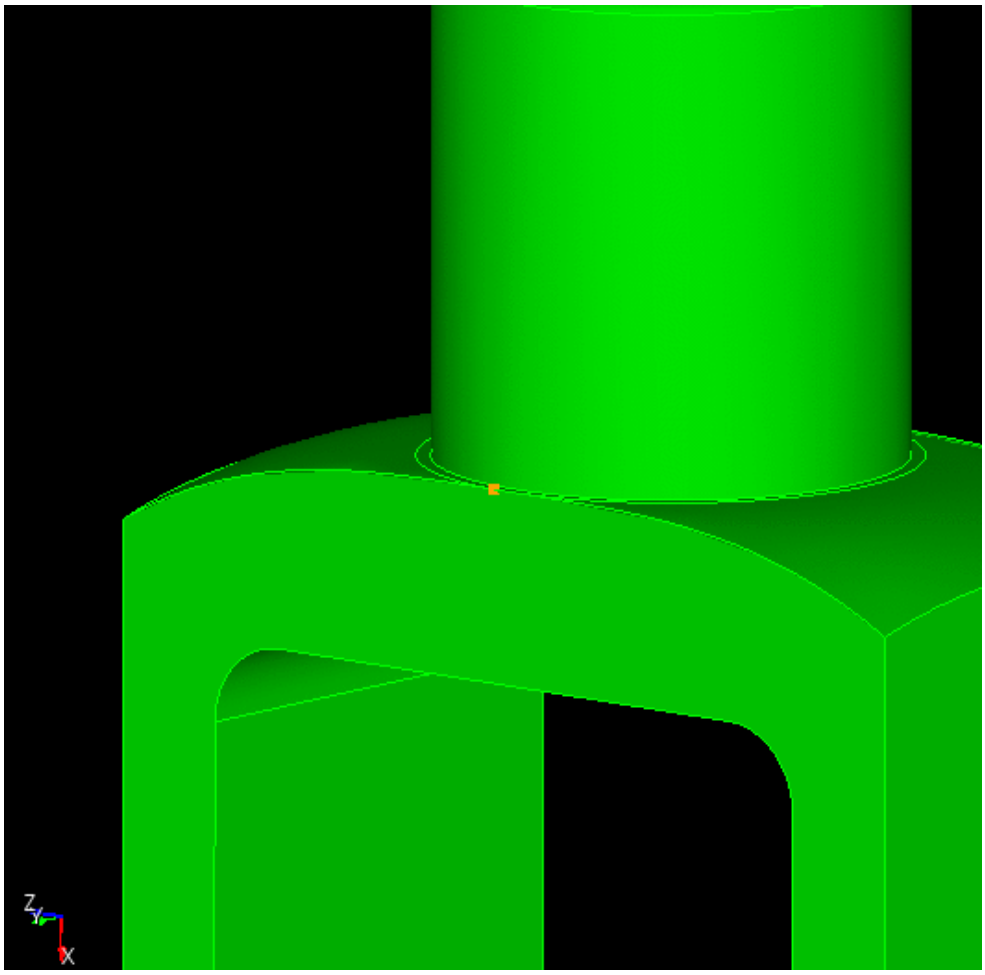

After the Analyze Button is pushed, display area will appear as shown above. There are four suspected problems with this geometry: Curves with Small Angles, Blend Surfaces, Close Loops, and Badly Defined Geometry. The numbers in parentheses indicate the number of occurrences of this problem in the model. Clicking on the + sign by each label will list the CUBIT entities by ID with this problem. Clicking on the + sign by each entity will cause that entity's children or parents to be listed (depending on the entity and the type of geometry test). See documentation on Geometry Repair for more information about the display window. Clicking on the name of an entity will highlight that entity in the graphics window.

- Select **Vertex 45** under Curves with Small Angles

Observe that this vertex is highlighted in the graphics window.

- Right click and select **Zoom To** from the list of options

The graphics window should look like this:




- Right Click on Vertex 45 and select **Reset Zoom** from the list of options

The image should now be reset to the previous graphics state.
You can experiment with some of the other options in the top half of the right click menu. They are:

- **Fly-in** - Animated zoom feature
- **Locate** - Labels entity
- **Draw** - Draw this entity by itself
- **Draw with Neighbors** - Draw this entity with all adjacent curves and surfaces
- **Clear Highlights** - Clear all highlighted entities
- **Reset Graphics** - Refresh graphics screen

The graphics window may also be reset by pressing the reset graphics button  on the menu.

Power Tools GUI Tutorial Step 3

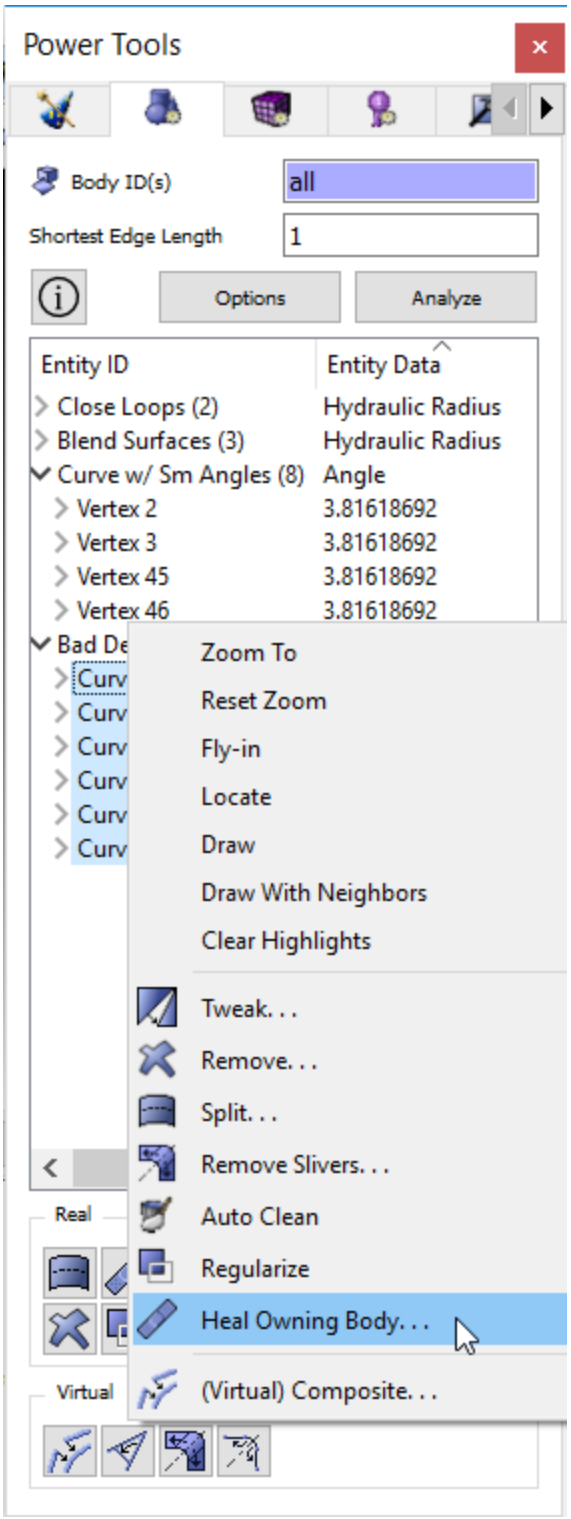
# *Step 3: Healing the Geometry*

The first step to improving any geometry is to look for badly defined geometry and to fix it using the Autoheal tool in CUBIT. The Geometry Analysis tool may detect these inconsistencies, but only if such a function exists in the parent software. It is always a good idea to run the Autoheal on imported geometry. In this example, the Power Tools has located some badly defined curves. This step will show you how to use the geometry repair tool to fix these curves.

- **Highlight** all of the badly defined curves by holding down the Shift key while selecting
- Right click and select **Heal Owning Body** from the list of options

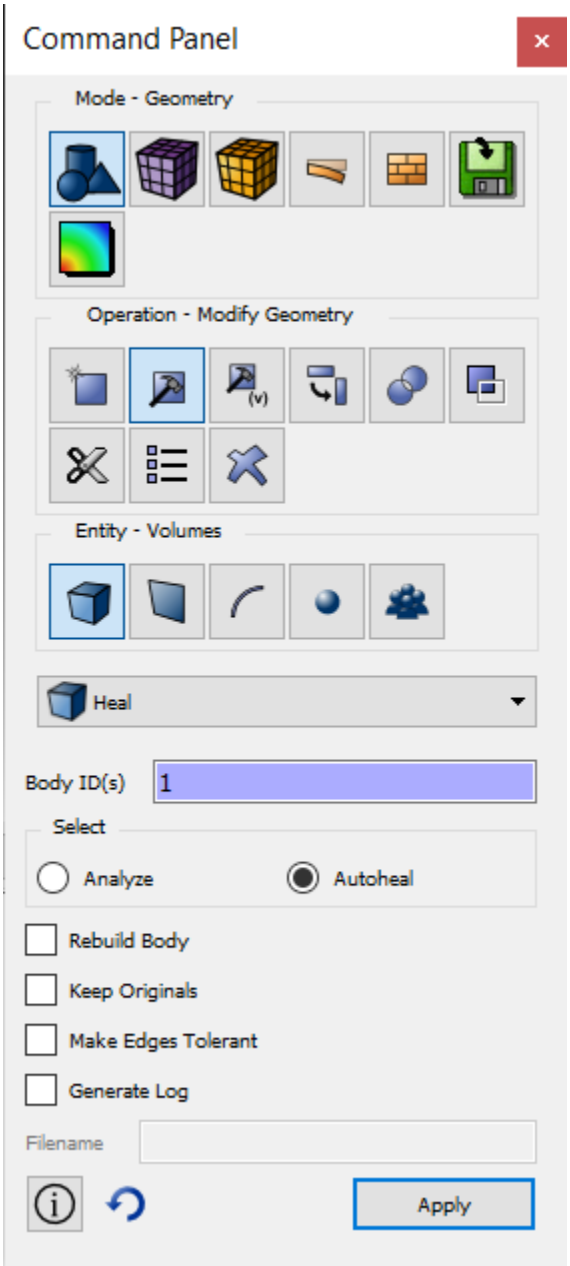
**OR**

- Click the  button

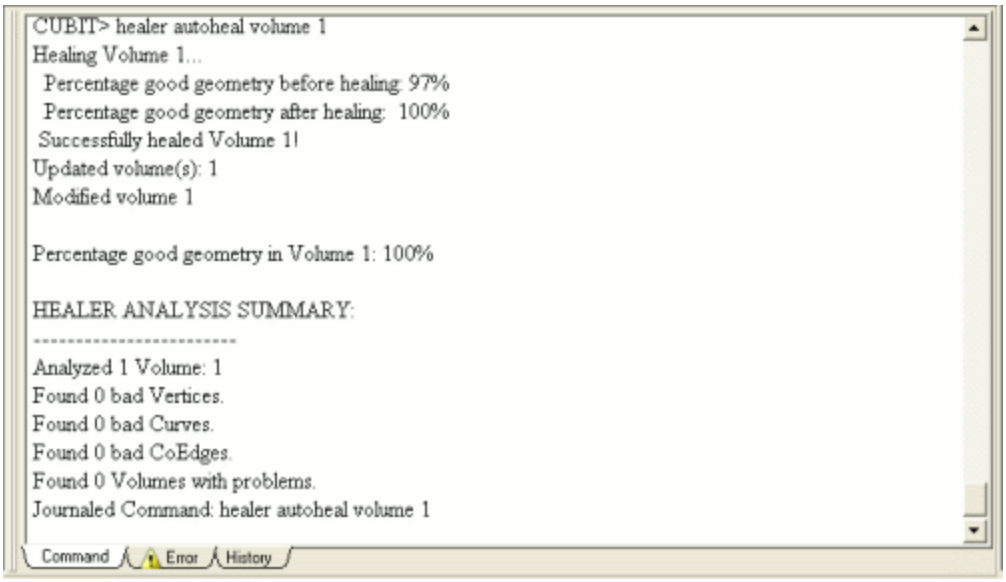



The Geometry Repair Tool does not execute any geometry clean-up commands directly, but directs you to the place on the Control Panel where this function can be executed. The following menu will appear on the Control Panel. Notice that the id of the owning body has already been pasted into the input window.

- Select the **Autoheal** button
- Press **Apply**




The output window on the CUBIT GUI should appear with the following message. You may have to scroll to see the whole thing. The percentage before and after healing are 97% to 100%. Healing has been successful.




Run the geometry analysis test again to guarantee that all bad geometry has been removed.

- Press the **Analyze** Button in the Geometry Repair window

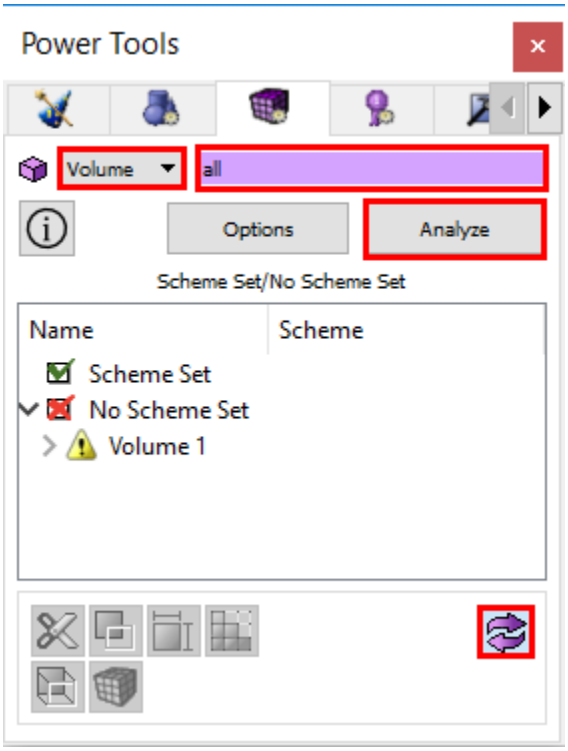


Power Tools GUI Tutorial Step 4

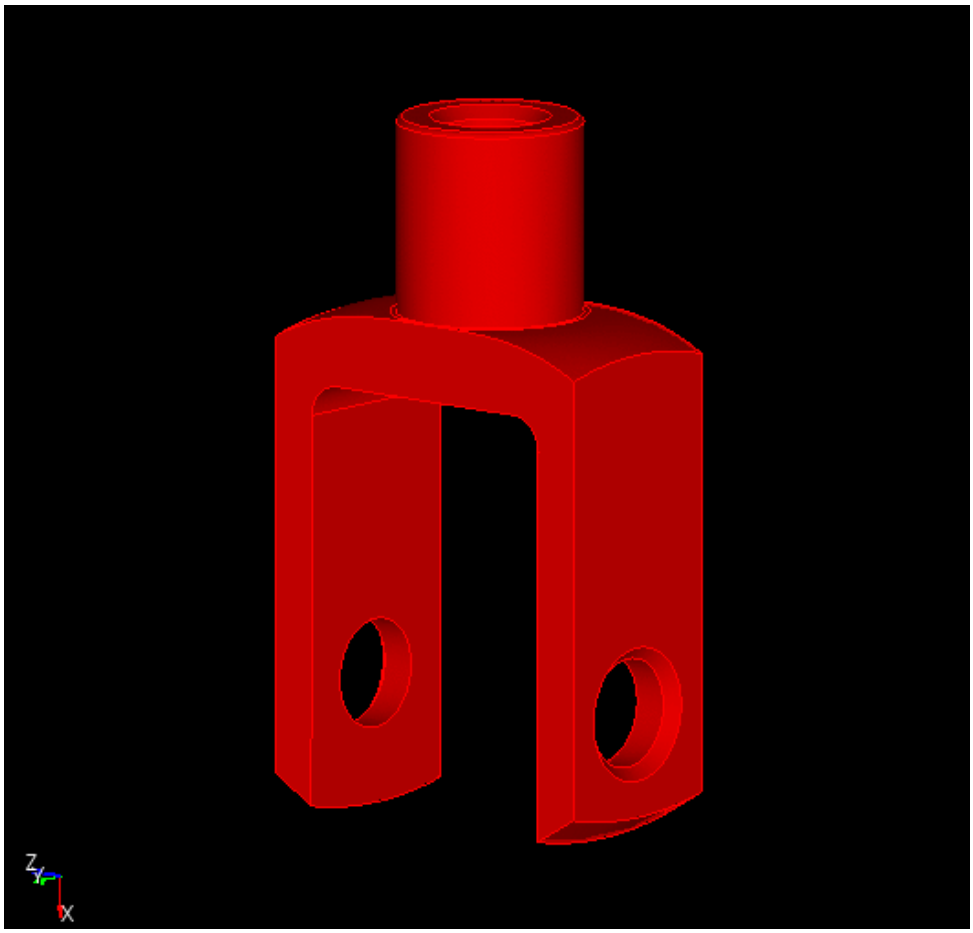
# *Step 4: Mesh Power Tools*

The Mesh Power Tool provides an easy and graphical way to determine if volumes are meshable. This tool will employ the AutoScheme feature in CUBIT to select and assign schemes to meshable volumes. If a volume is not currently meshable, it will be flagged and highlighted. Use the Mesh Power Tool to determine if the volume is currently meshable.

- Click on the purple Mesh Tools tab in the Power Tools window.
- Select **Volume** as the entity type in the pull-down menu (It may already be selected)
- Enter **all** in the input window
- Press **Analyze**




Volume 1 will appear under the "No Scheme Set" heading.
The graphics window should look like this with Volume 1 highlighted in red. Using this graphics feature, all volumes that are meshable will be highlighted in green, and all volumes that are not currently meshable will be highlighted in red.




- Toggle the **Graphics Button** (located in the bottom right corner of the tool) off so that Volume 1 is shown in green again



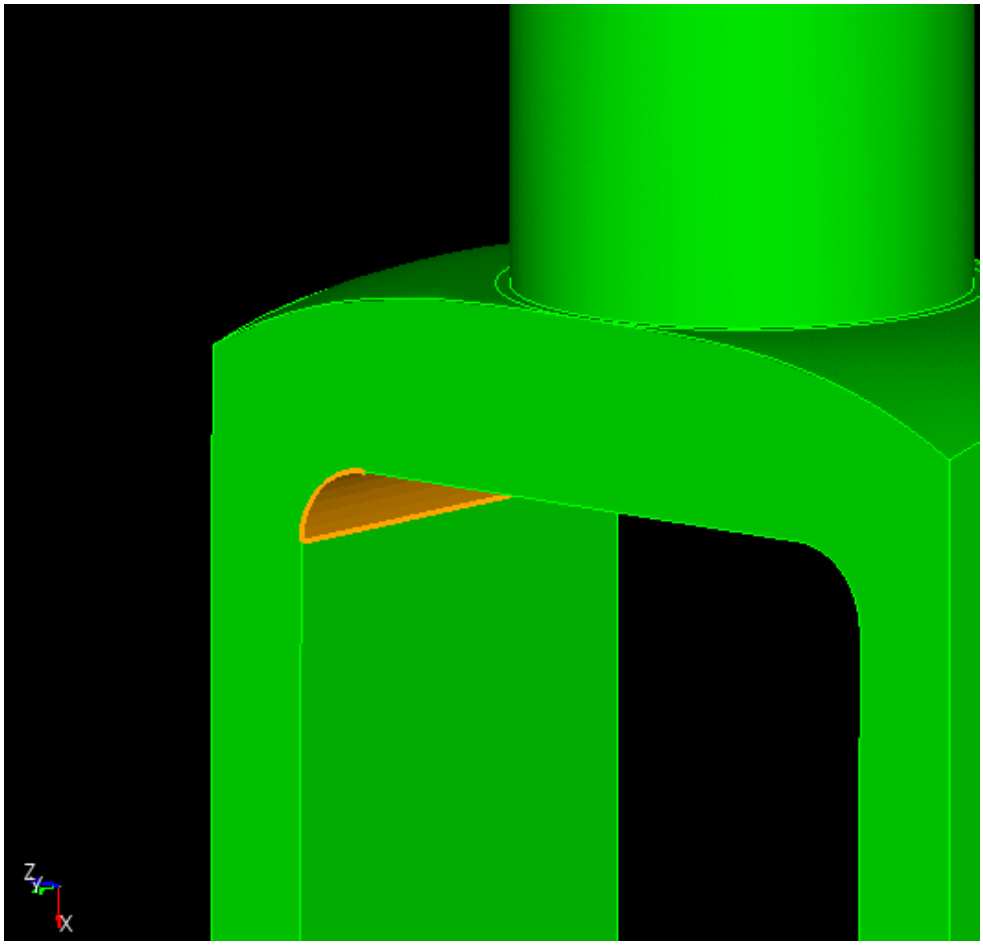
Power Tools GUI Tutorial Step 5

# *Step 5: Splitting Filleted Surfaces*

The previous step determined that the volume was not currently meshable, and that further decomposition was required. This decomposition can be performed using the tools in the Geometry Repair power tools. A good place to start is with blend surfaces.
A blend surface is a transitional surface that connects two orthogonal planes, also known as a fillet. Blend surfaces can be problematic in meshing because there is no clear transition between the two orthogonal surfaces, making sweeping or mapping algorithms difficult. The Split Surface function divides these blend surfaces (or any surface) into two distinct surfaces.

- Select **Surface 22** from the list of blend surfaces
- Right click and select **Zoom To** from the list of options

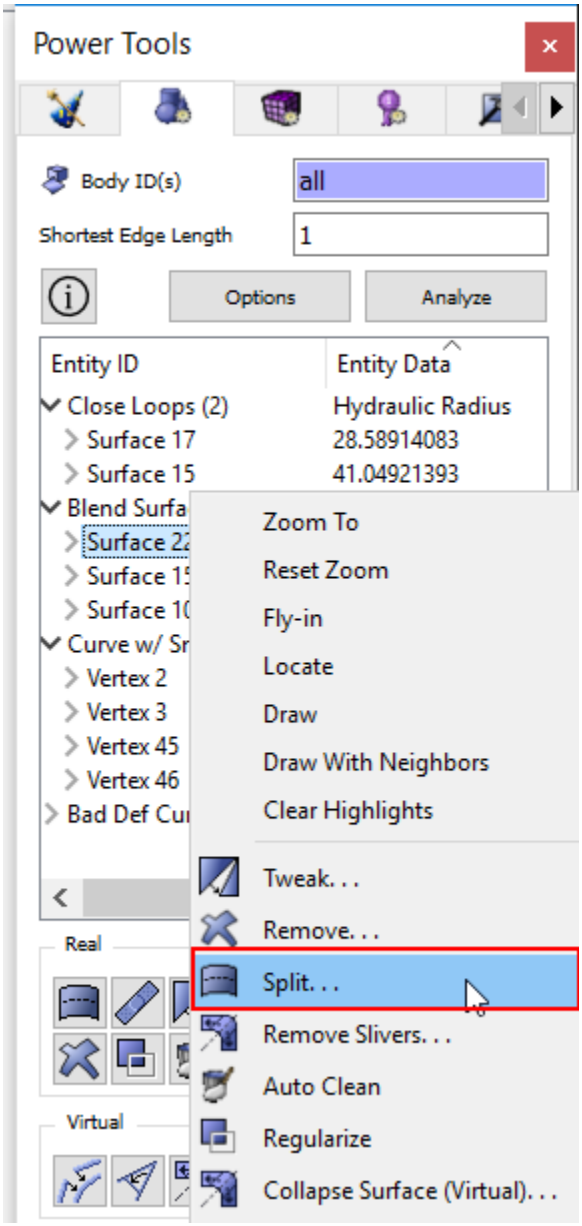
The graphics window should look like this:



- Right click with Surface 22 highlighted and select the **Split** button

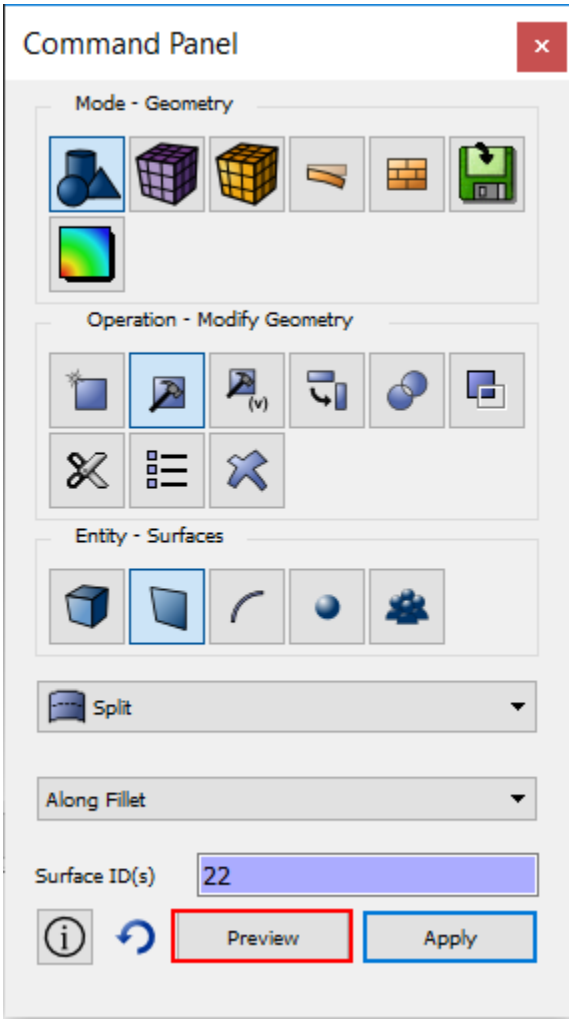
**OR**

- Click the  button on the tool panel

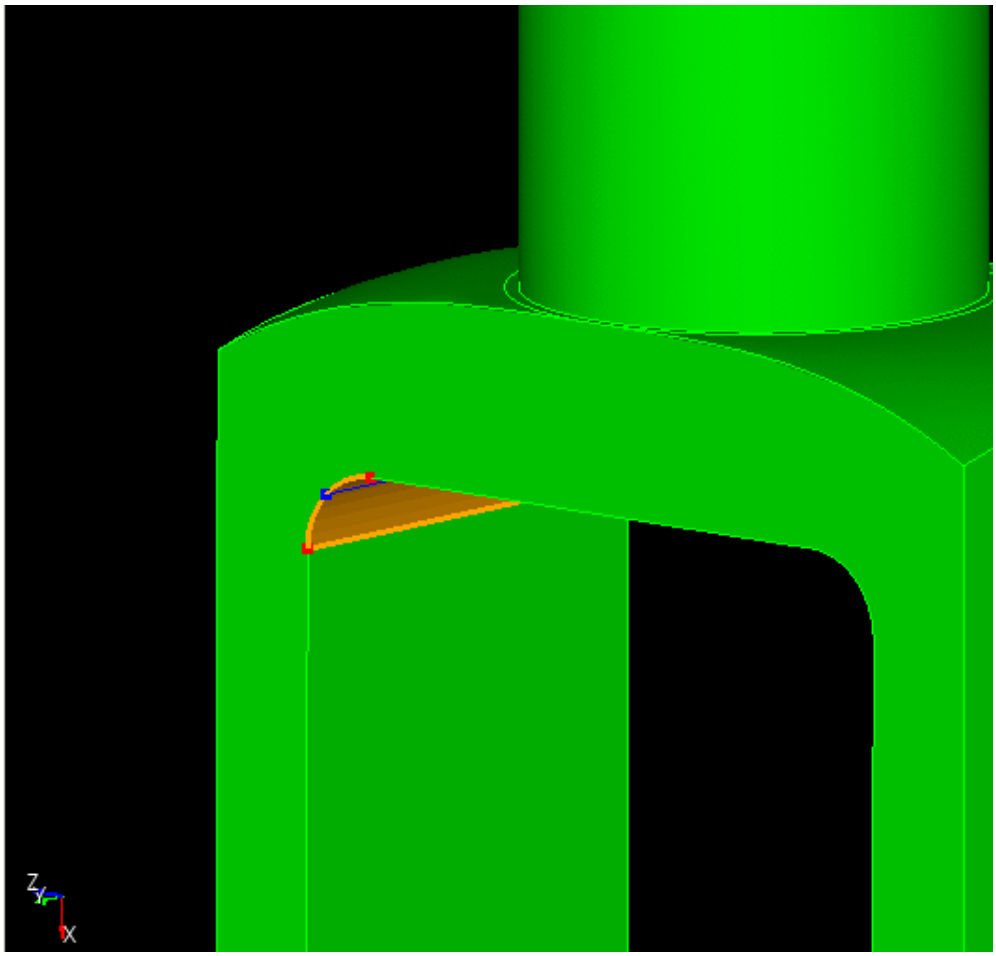



The Geometry-Surface-Modify-Split Menu will appear on the Control Panel. Make sure the Surface id is input in the window.

- Press the **Preview** Button




The blue line shows where the surface will be split.




- Press the **Apply** Button

The surface should now appear split.

- Repeat these steps with the opposite blend surface (ID 10)

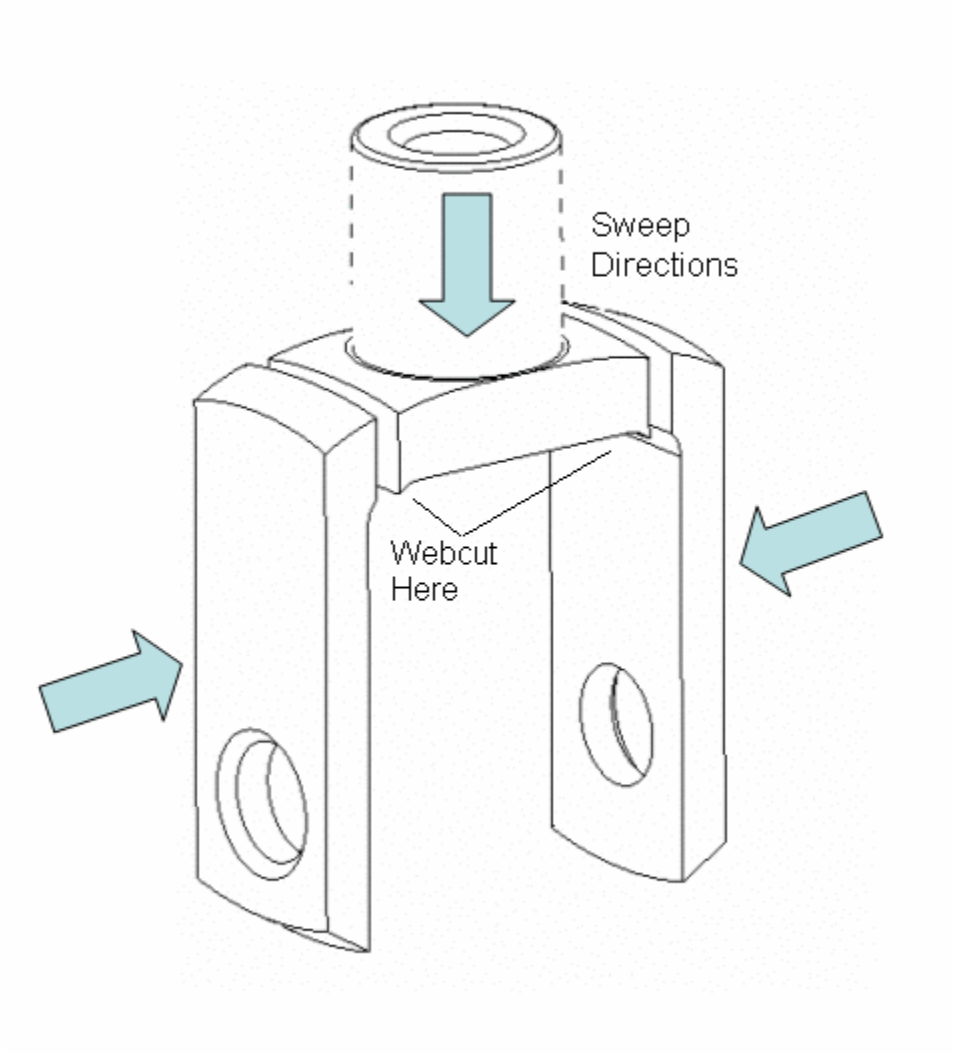


Power Tools GUI Tutorial Step 6

# *Step 6: Web Cutting*

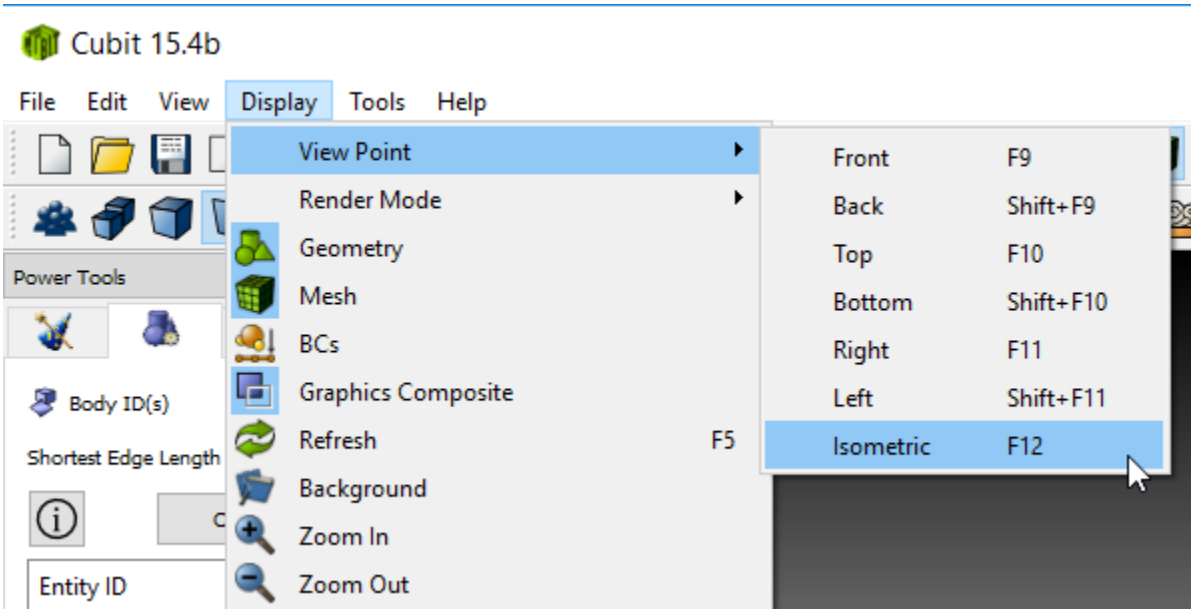
Since the model has several through holes, sweeping is not possible from a single source and target. However, it is possible to divide the model into three sweepable regions. The figure below shows where to divide the model to get it into sweepable regions. These regions coincide with the holes in the model.

Web cutting is this process of dividing volumes into sweepable regions by cutting with a plane. For this exercise, you will use the curves that were just created with the split surface command to cut the volume.



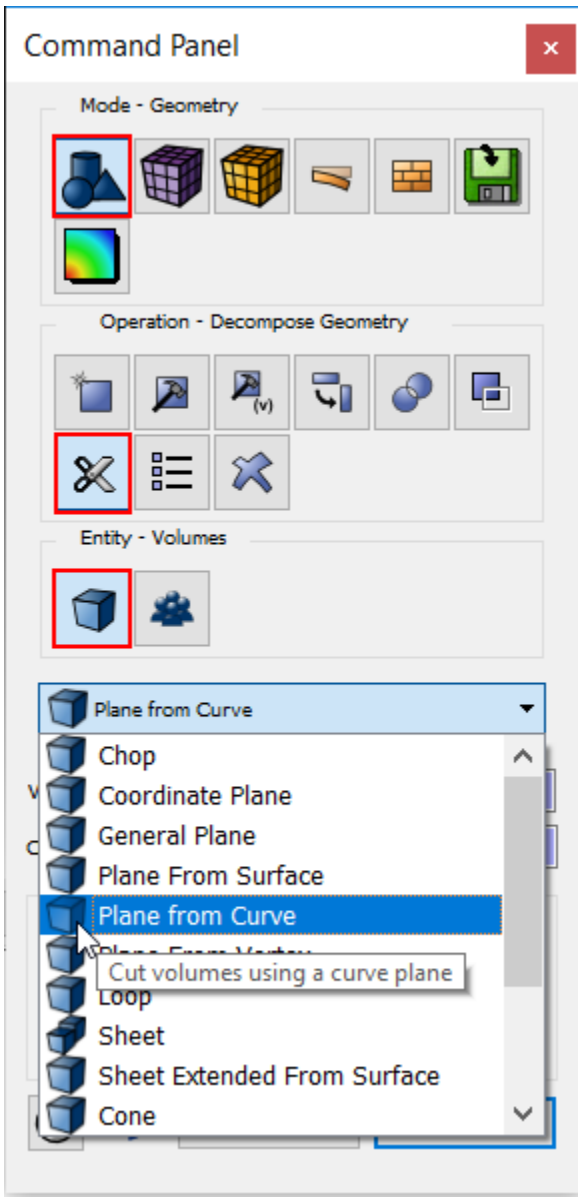

In order to visualize the process more clearly, switch to the isometric view.

- Change the view to isometric in the **Display** menu under **View Point**

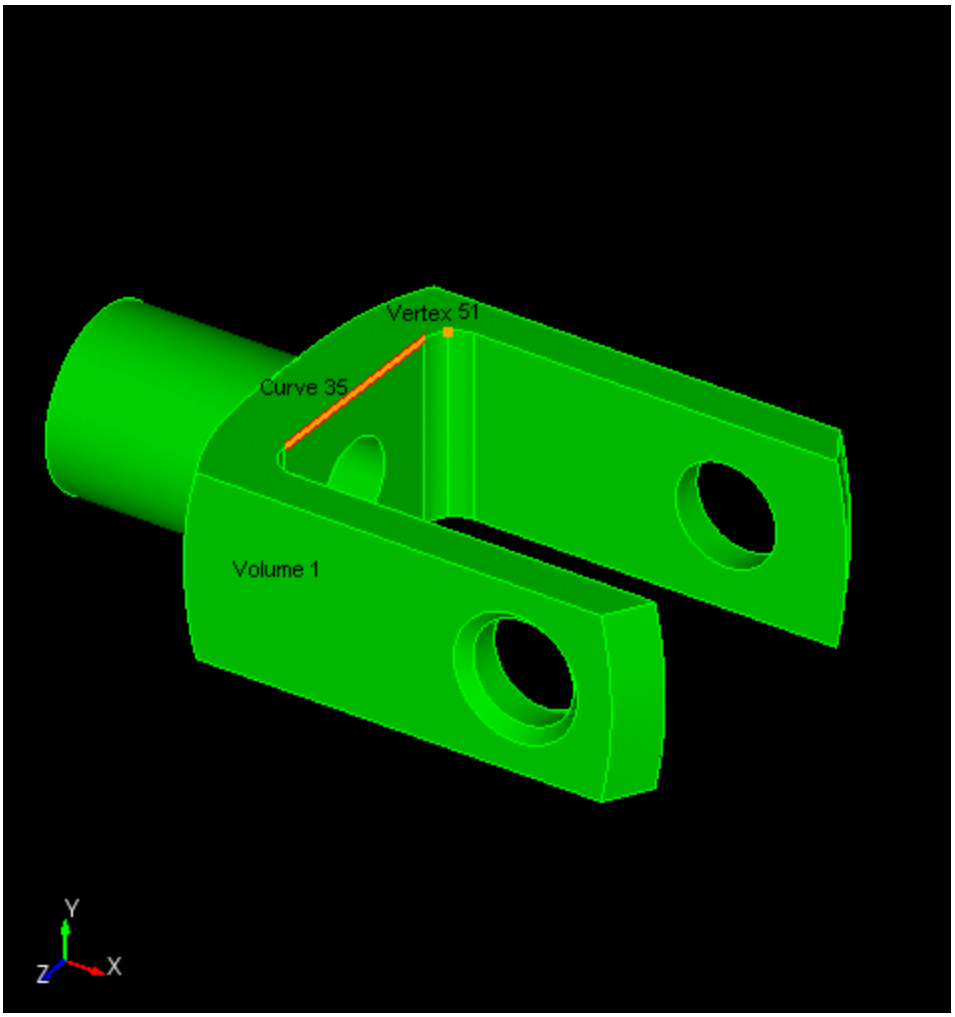



The web cutting menu is located under Geometry-Webcut-Volume on the Control Panel.

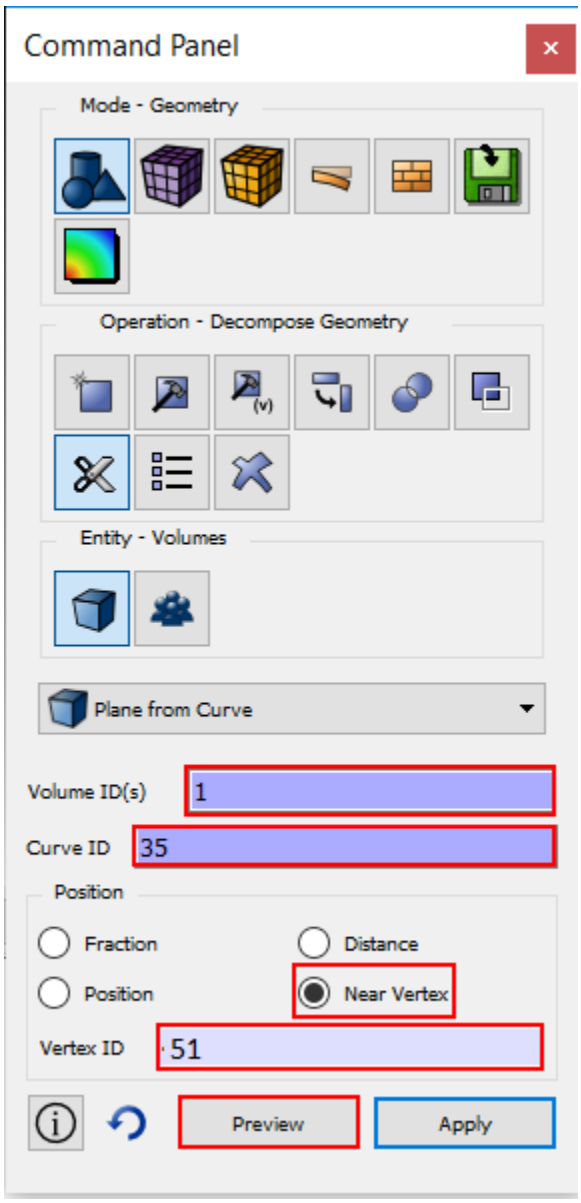
- Click on **Geometry**, then **Webcut**, then **Volume** on the Control Panel
- Select **Plane from Curve** from the list of options



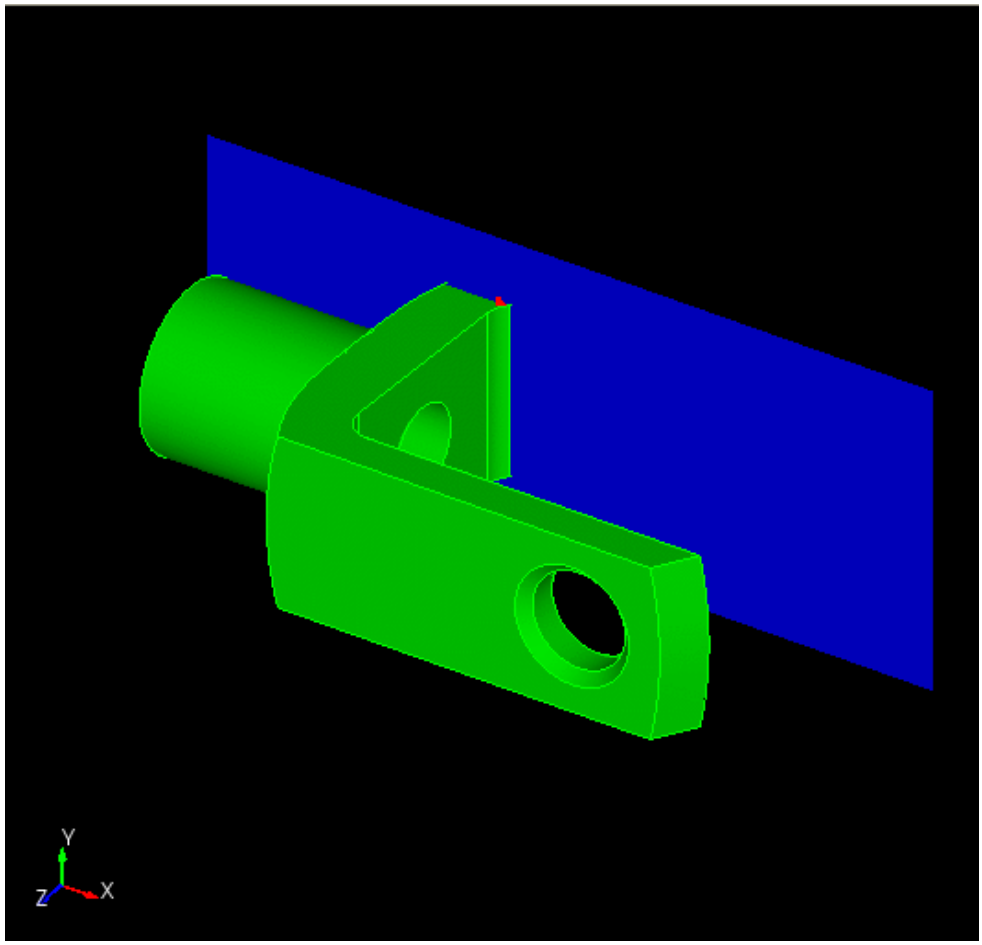

The following image shows the entity ids that will be used to webcut the volume. Select entities with the mouse by clicking on them.




- Enter **Volume 1** by typing it or selecting from the graphics window
- Enter **Curve 35** by typing it or selecting from the graphics window
- Change the **Type** to **Near Vertex**
- Enter **Vertex 51** by typing it or selecting from the graphics window
- Press **Preview**




A blue preview plane should appear in the following position. Check to make sure that your model looks the same.

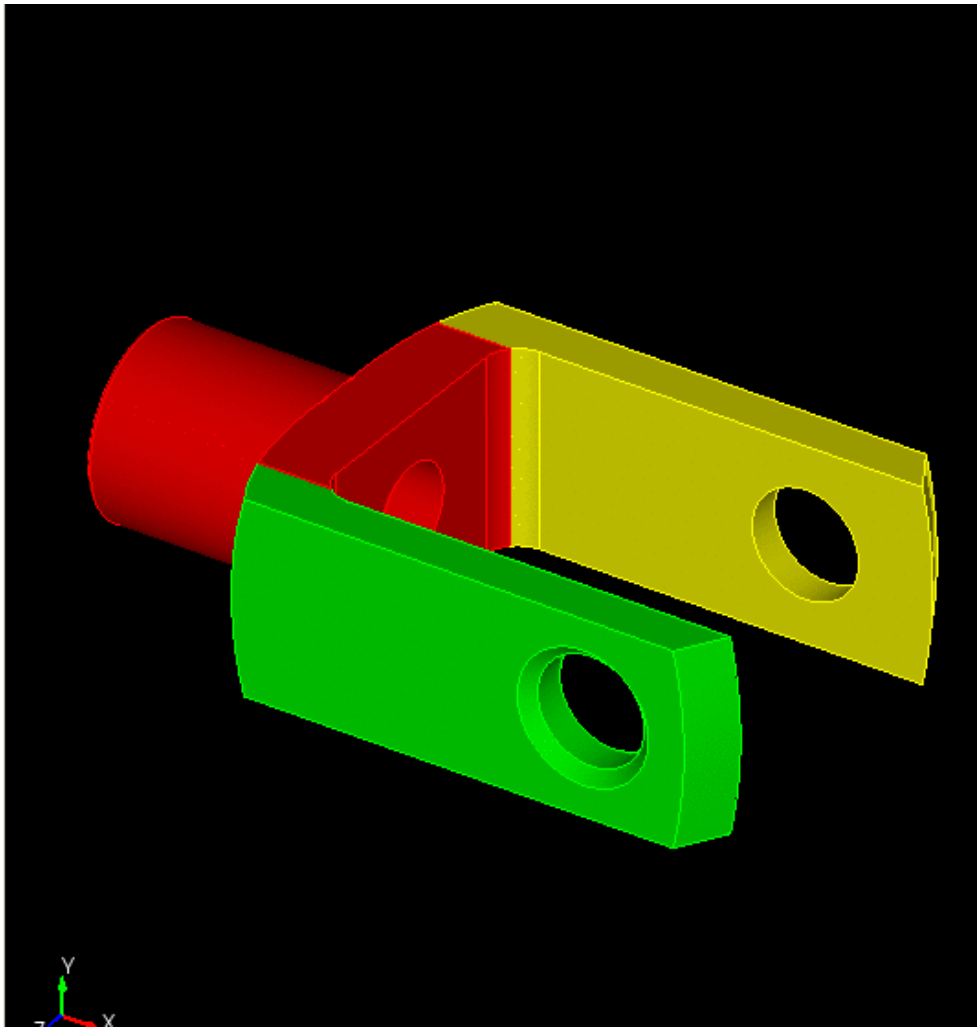



- Press **Apply**

The volume has now been split into two volumes. Volume 2 is shown in yellow. Repeat these steps with the other side of the part. The Volume and Curve ids will remain the same.

- Enter **Vertex 49** in the input window or select from the graphics window
- Press **Preview**, then **Apply**

The final webcut volume should look like this:

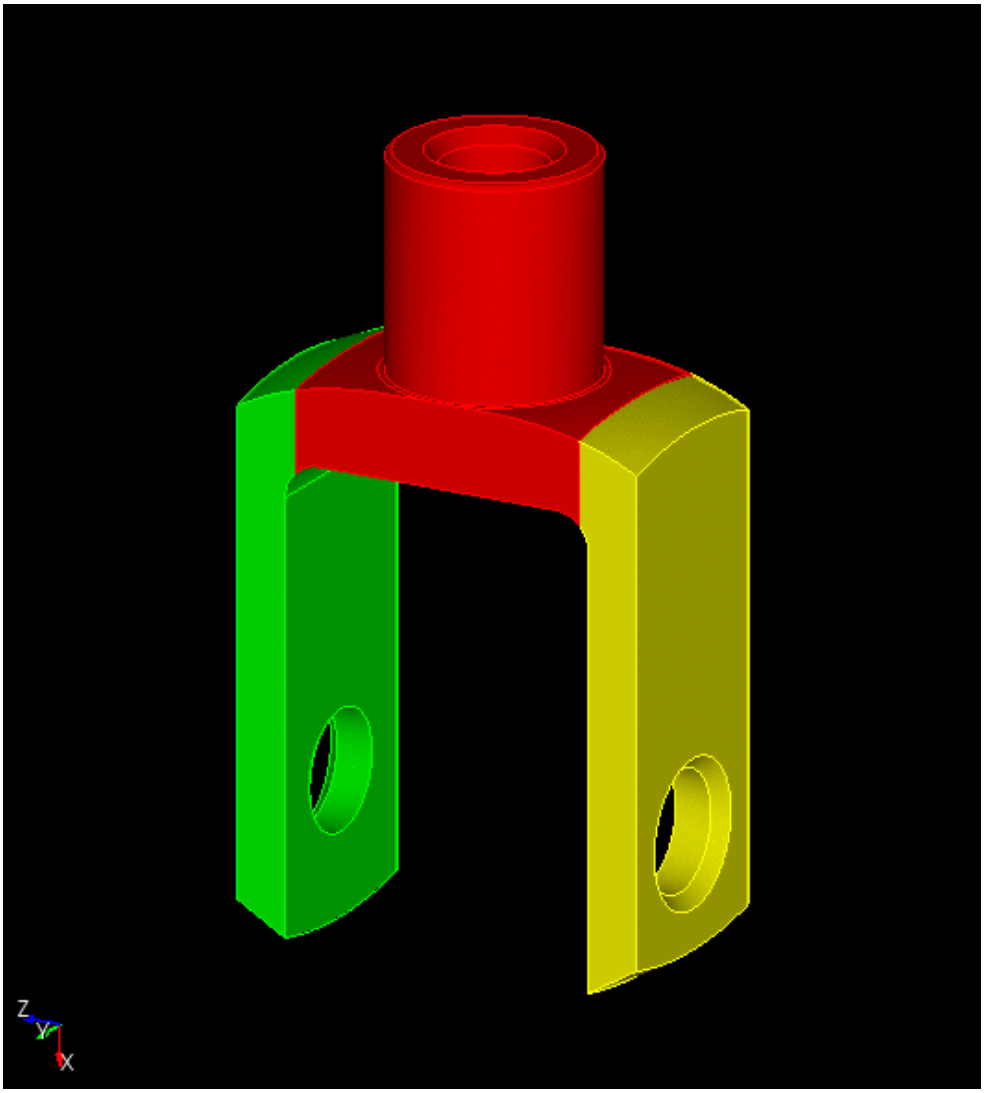



Power Tools GUI Tutorial Step 7

# *Step 7: Removing Small Surfaces*

Some surfaces are too small for analysis and should be removed from the model. In this example, Surface 15 and Surface 17 may fall into that category, assuming that the distance between curves on these surfaces is smaller than the desired final mesh size. You can remove these surfaces by extending adjacent surfaces until they intersect.

- **Rotate** the model to the following orientation



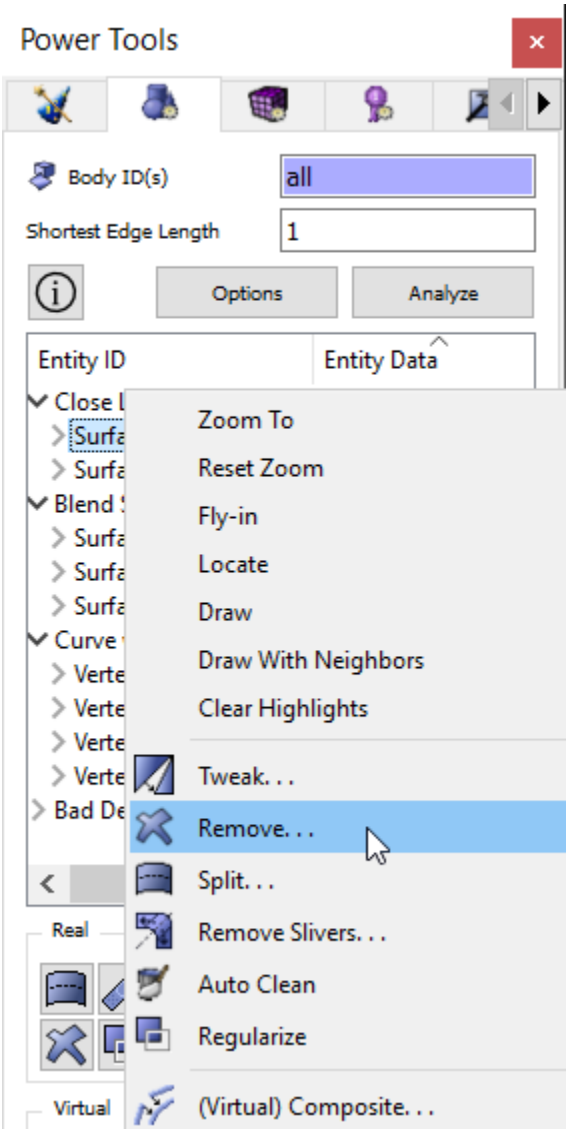

- Press **Analyze** on the Geometry Power Tools menu

You will notice that a new category has appeared labeled Overlapping Surfaces. This is because there are two new surfaces created for each of the webcuts that overlap a surface on the original body. This can be removed using the Imprint/Merge function which will be explained in Step 9.

- **Zoom** to **Surface 17** in the graphics display
- **Right Click** on **Surface 17** in the Geometry Repair window and select **Remove**

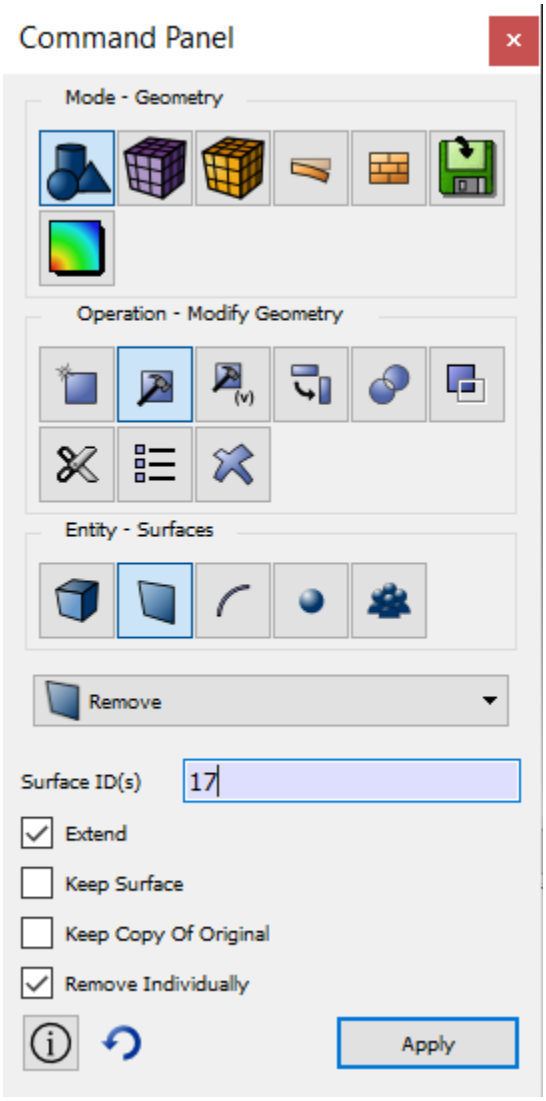
**OR**

- Press the **Remove Button**  on the tool bar

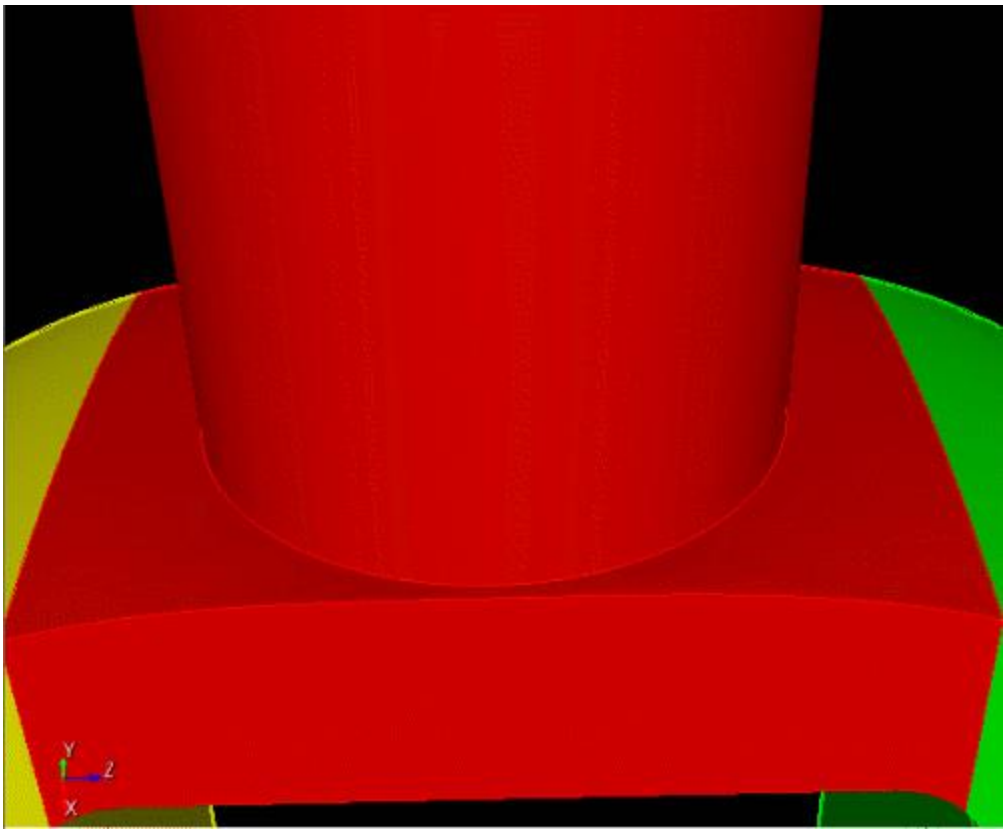



The Control Panel will appear under the Geometry-Surface-Modify- Remove heading. The Surface id should appear in the input window.

- Make sure that **Surface 17** appears in the window and the **Extend** button is checked
- Press **Apply**

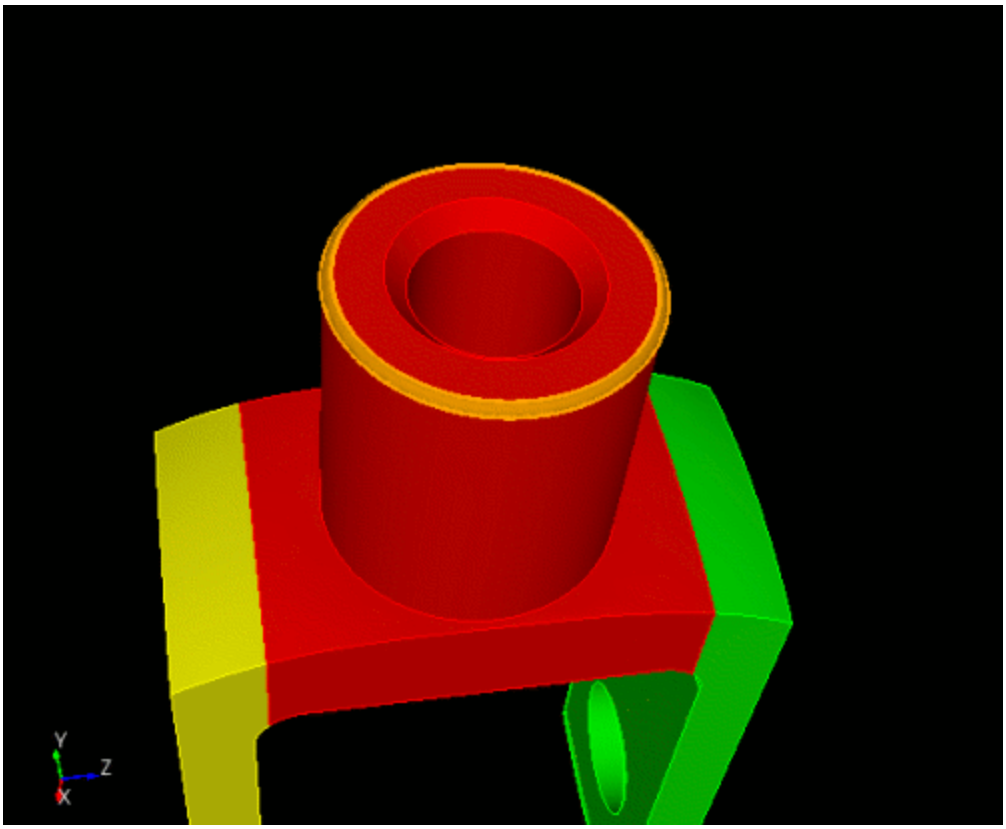



The small surface no longer appears.



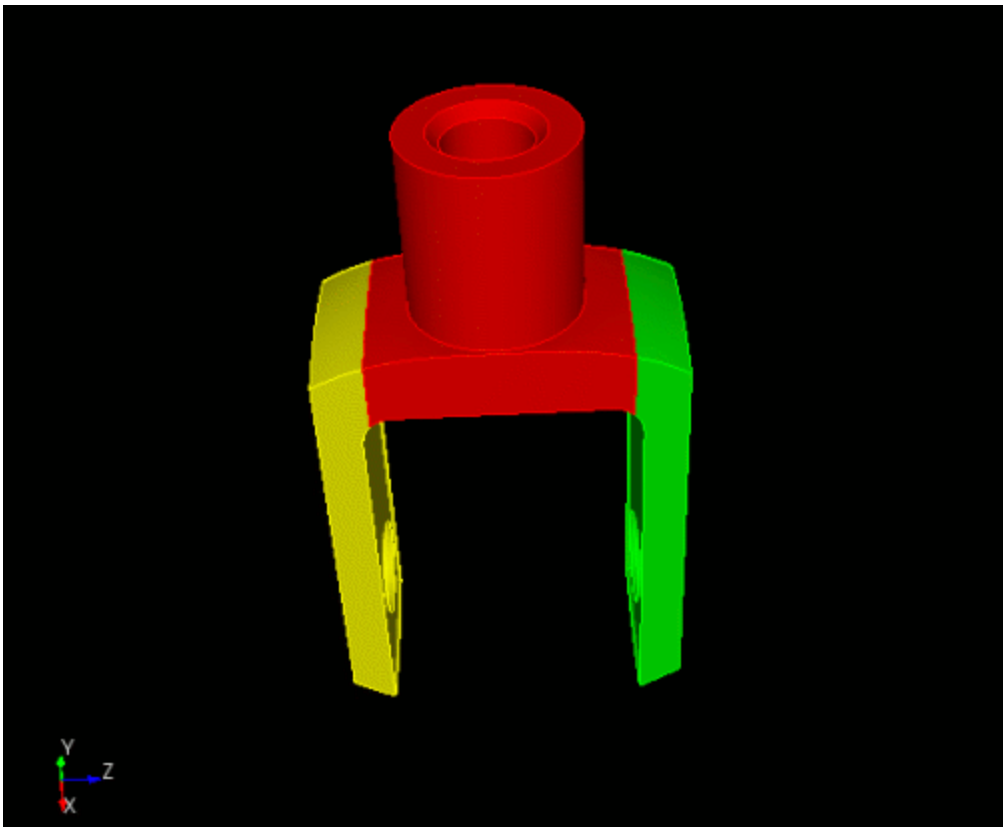

- Highlight **Surface 15** and select the **Remove** option

Surface 15 is shown highlighted in the following image.




- The Geometry-Surface-Modify-Remove option appears on the Control Panel. Make sure that **Surface 15** appears in the input window.
- Press **Apply**

Reset the Zoom to show the entire model.




Power Tools GUI Tutorial Step 8

# *Step 8: Tweaking Surfaces*

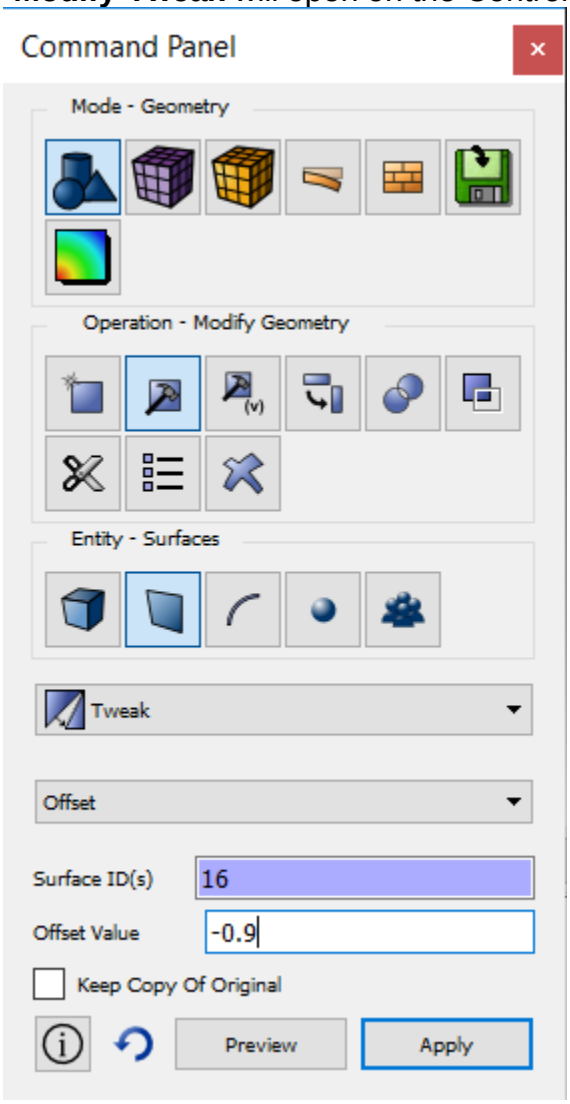
Tweaking is the process of deleting, moving, or offsetting, surfaces and extending or trimming adjacent surfaces to fill in the gaps. Tweaking is useful for eliminating gaps between components, simplifying geometry or changing the dimensions of an entity. Tweaking will be used in this example to decrease the radius of the upper cylinder.
Begin by reanalyzing the geometry.

- Press **Analyze** on the Power Tools menu

There should be 1 entry under the "Close Loops" category for Surface 38. A close loop (pronounced KLOS) is a surface which has two loops that are within some small distance of each other at their closest points. The parameter for distance is the square of the shortest edge length parameter.

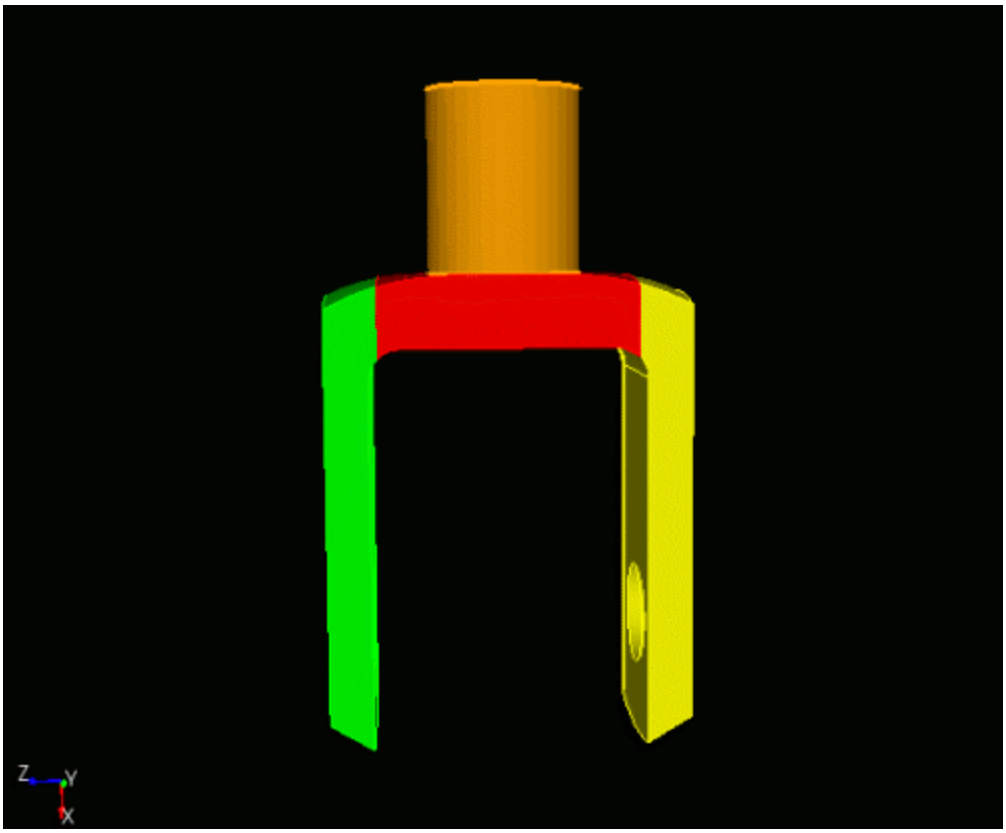
- Press the **Tweak Button**  (since you are not tweaking Surface 41 directly, the surface does not need to be highlighted when you press the tweak button)

The **Geometry-Surface-Modify-Tweak** will open on the Control Panel as shown below.




- Enter **Surface 16** by typing it in at the input line or selecting from the graphics window
- Select the **Offset** option from the pull-down menu

Surface 16 is shown highlighted below.

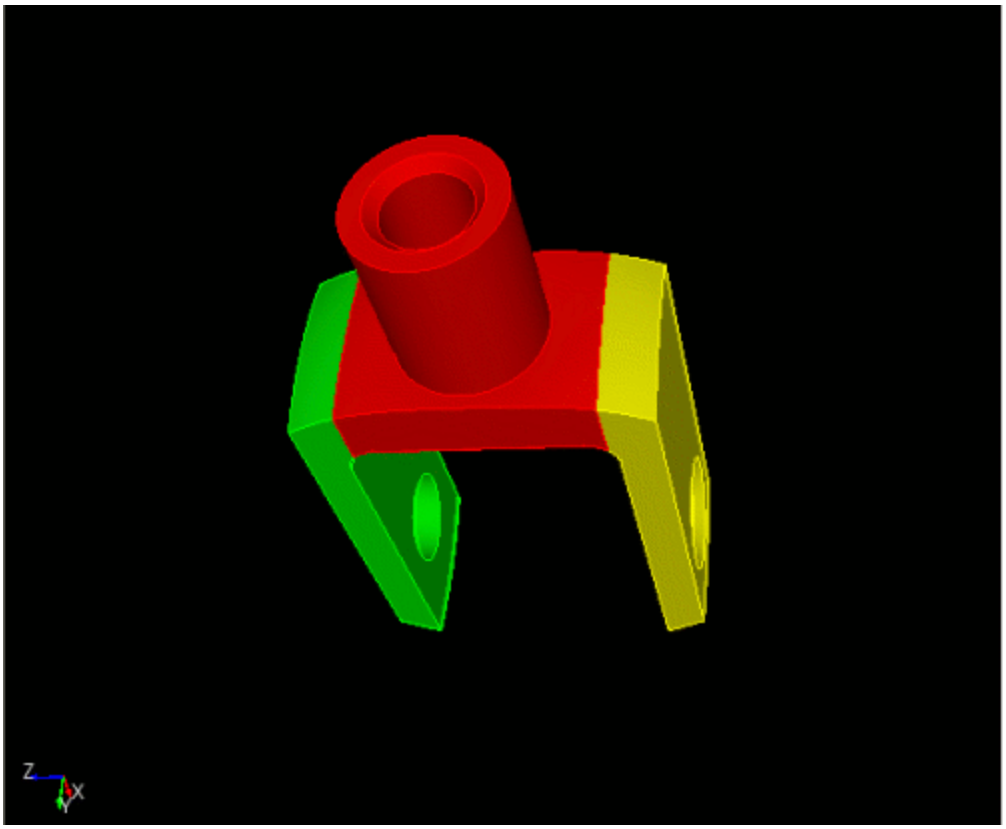



- Enter an Offset **Value** of **-0.9**.

The offset value is a percentage of the current size. Entering -0.9 will decrease the radius by 10 percent.

- Press **Apply**

The graphics window should now look like this. Notice that the radius of the cylinder has shrunk inward, increasing the gap between the edges on Surface 41.

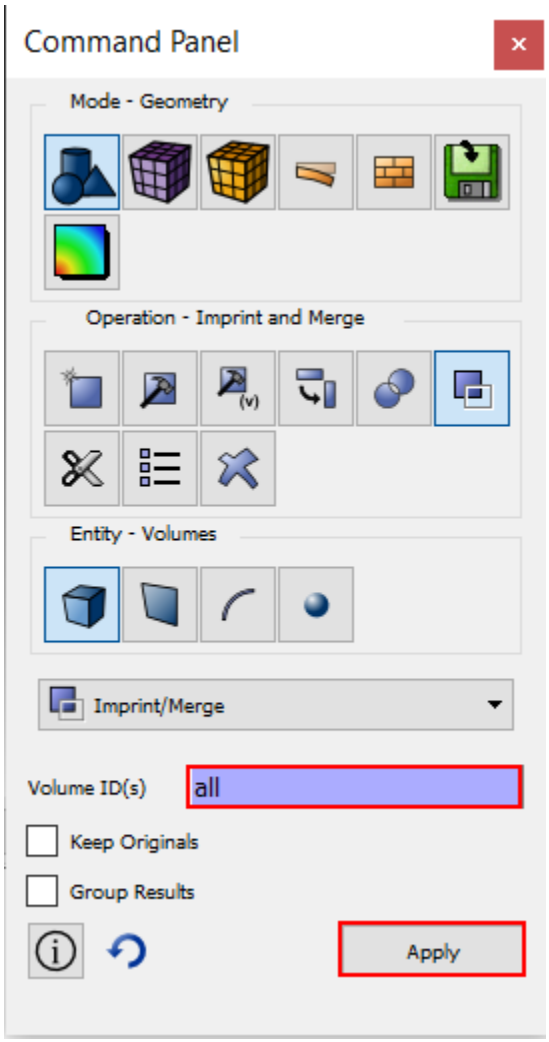



Power Tools GUI Tutorial Step 9

# *Step 9: Imprint/Merge*

Imprinting is the process of projecting curves from one surface onto an overlapping surface. Merging is the process of taking two overlapping surfaces and merging them into one surface shared by two volumes, creating non-manifold geometry. Both imprinting and merging are necessary to make adjacent volumes have identical meshes at their intersection. Imprinting and merging is almost always necessary after webcutting.

- To open the imprint/merge menu, select the **Geometry** icon, then **Imprint and Merge**, then **Volumes** on the Control Panel
- Enter **all** in the input window.
- Press **Apply**




You will not notice any visible changes in the graphics window after imprint/merge operations, but results of the operations will be printed in the output window. Confirm that both surfaces have been merged by reading the output in the graphics window (You may have to scroll to see all of the results)
You can return to the Power Tools menu to see that the Close Loops and Overlapping Surfaces are gone.

- Press **Analyze** in the Power Tools menu

The display window will now read "Nothing Found" to indicate that are no geometry tests that fail.

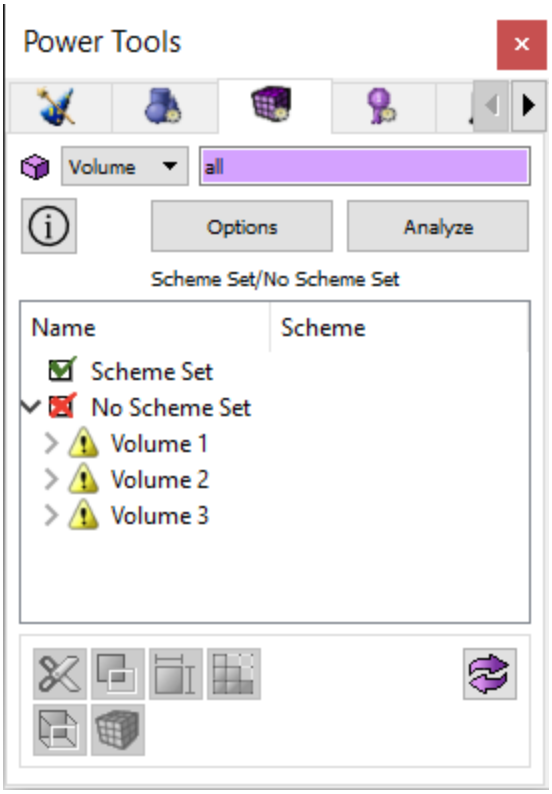


Power Tools GUI Tutorial Step 10

# *Step 10: Compositing Surfaces*

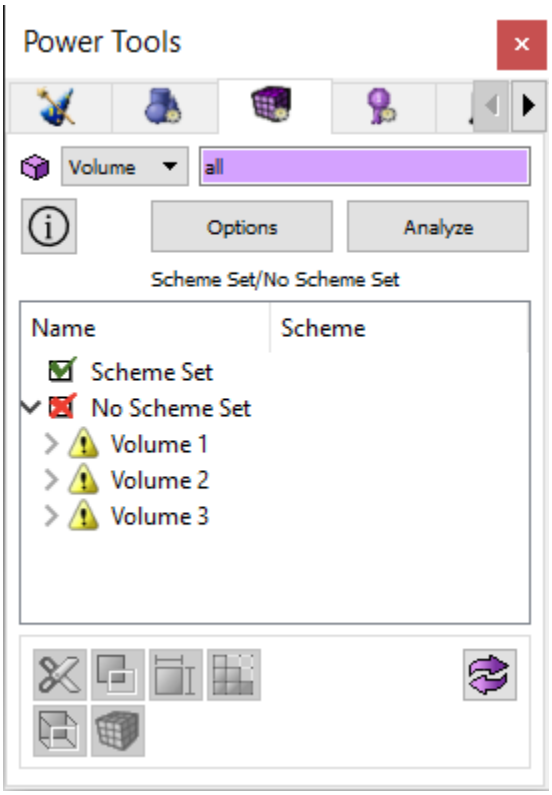
Composite surfaces are adjacent surfaces that have been merged into one surface. Composite surfaces are created using Virtual Geometry, which is a built-in geometry kernel that sits on top of the existing geometry, and does not change the underlying geometry definition. Virtual geometry has the added advantage of being reversible. It can be removed after meshing. The general purpose for using composite surfaces is to deconstrain the mesh. For example, compositing two surfaces will remove the requirement that nodes be placed on the curve between the surfaces. Composite surfaces will be used in this example to facilitate the sweeping algorithm.

- Open the **Mesh Tools** tab
- Enter '**all**' in the input field
- Press **Analyze**
- Toggle the **Reset Graphics** button to show entities in green and red (for meshable and non-meshable volumes)

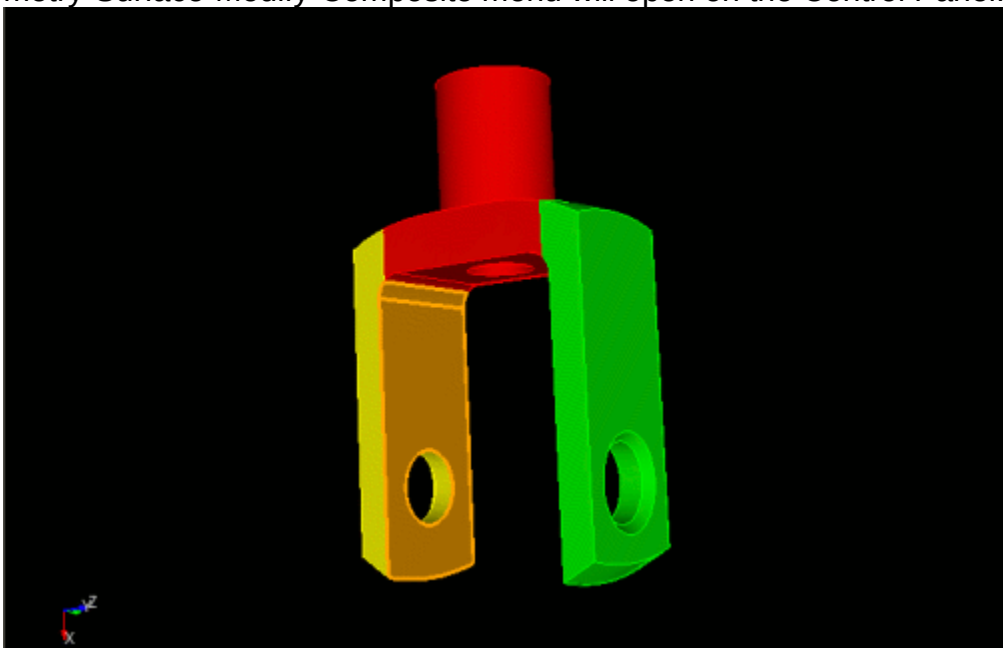



No volumes are listed as automatically meshable. In the graphics window, red indicates that the volume scheme has not been set. Green indicates that the scheme has been set.

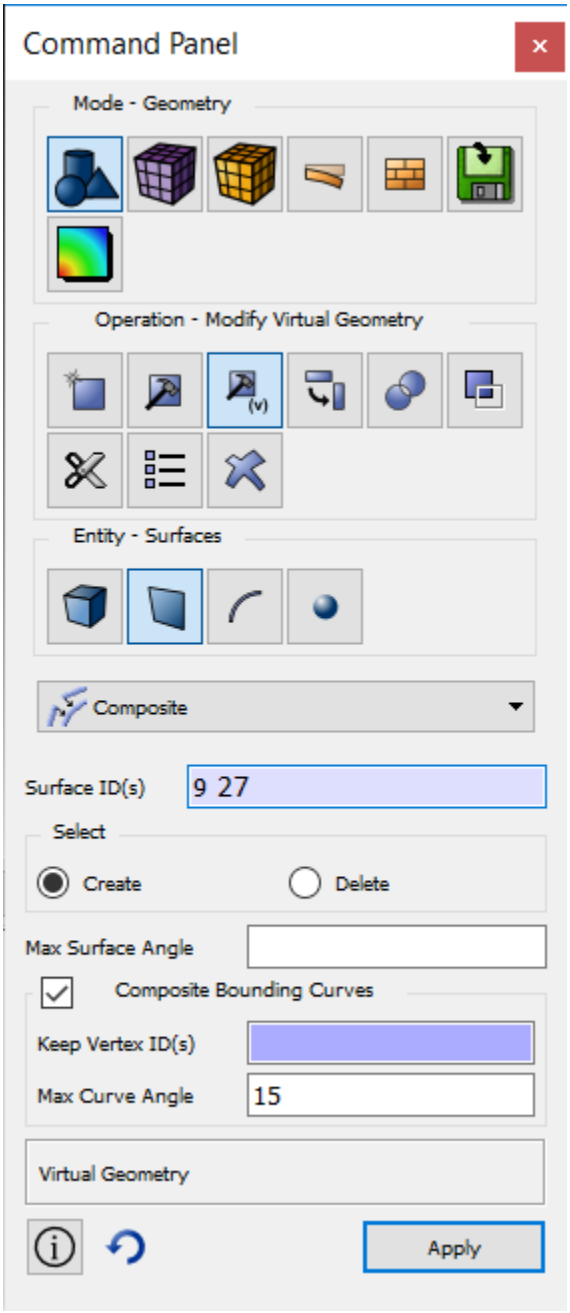
- Toggle the **Reset Graphics** button so it returns to the normal colors
- Open the **Geometry Tools** tab
- Press the **Composite Button**  on the toolbar



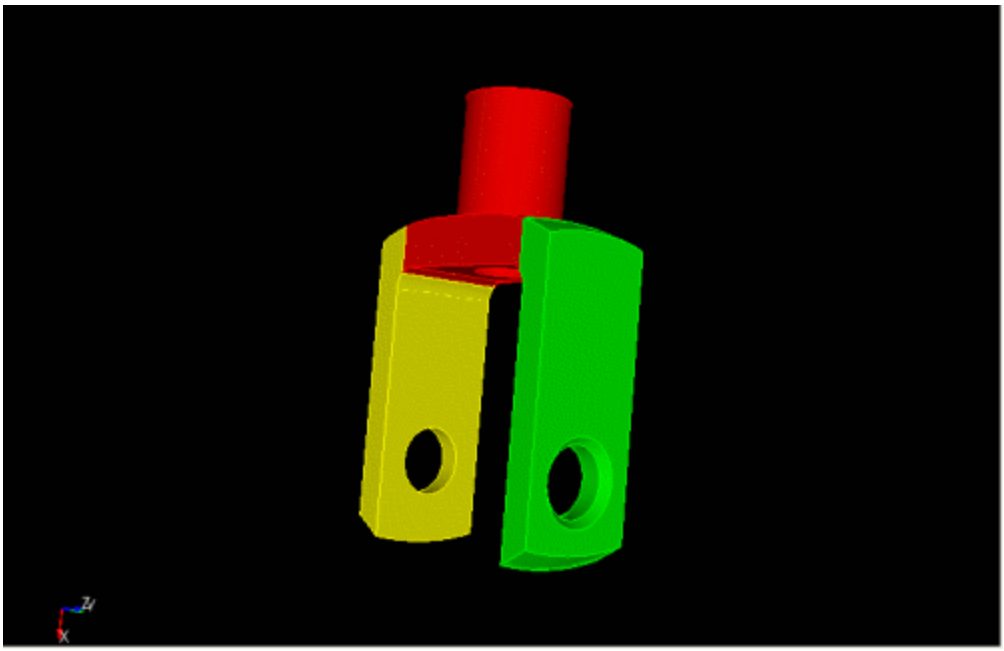

The Geometry-Surface-Modify-Composite menu will open on the Control Panel.



- Select **Surfaces 9** and **27** (shown in the image above) by entering them in at the input line, using CTRL-Click (Windows) in the graphics window, or Command Key-Click (Macintosh) in the graphics window
- Make sure the **Create** button is checked
- Press **Apply**

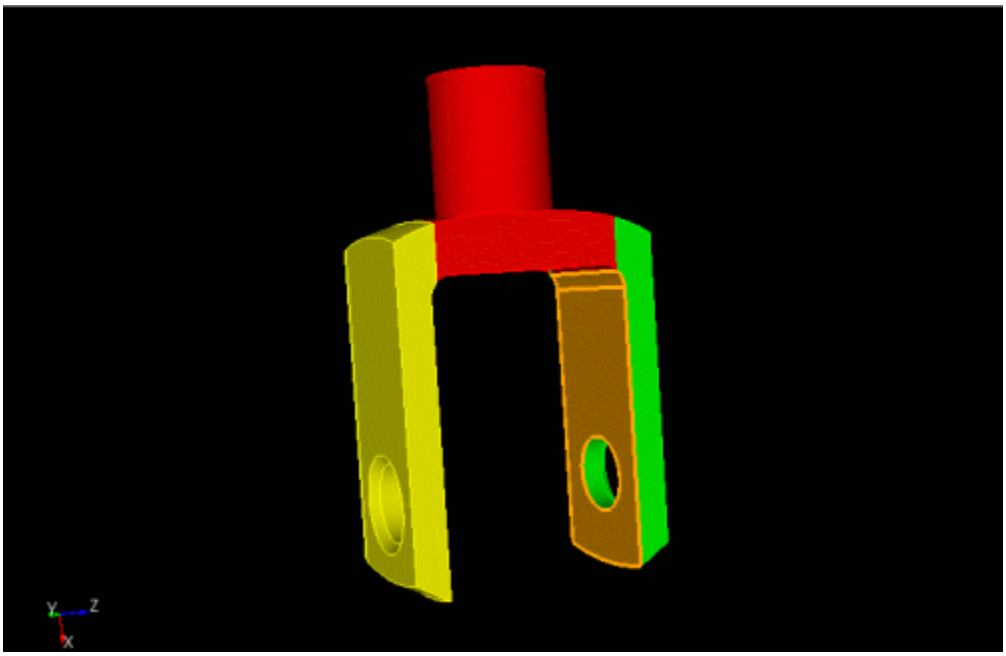



The two surfaces should appear merged.

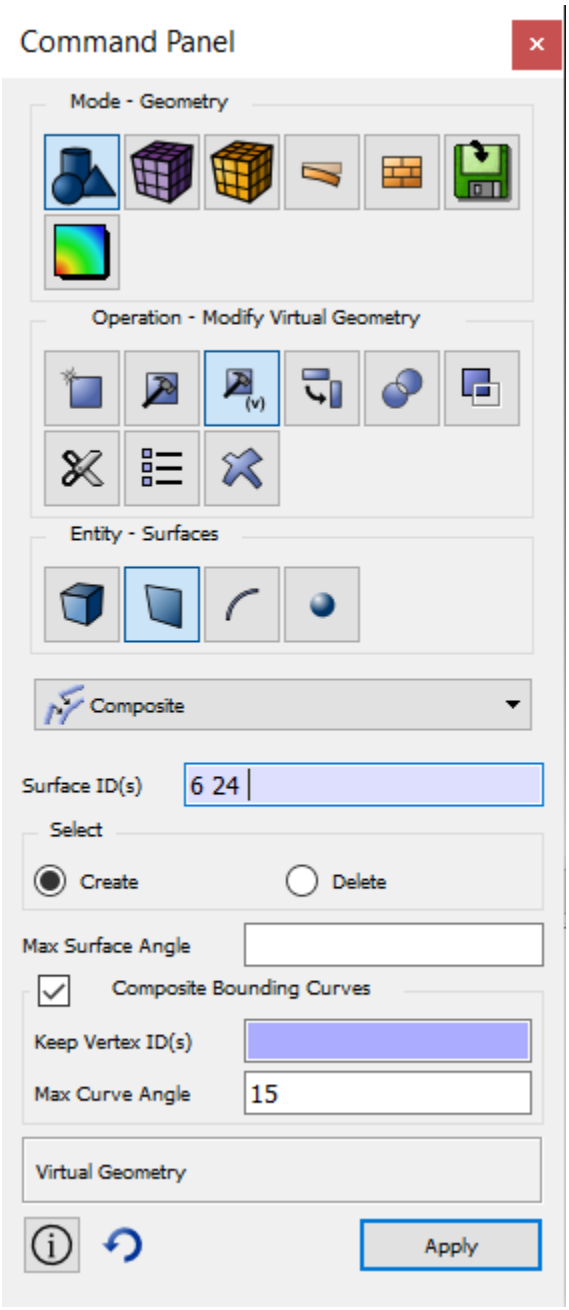


Repeat these steps with the opposite side.

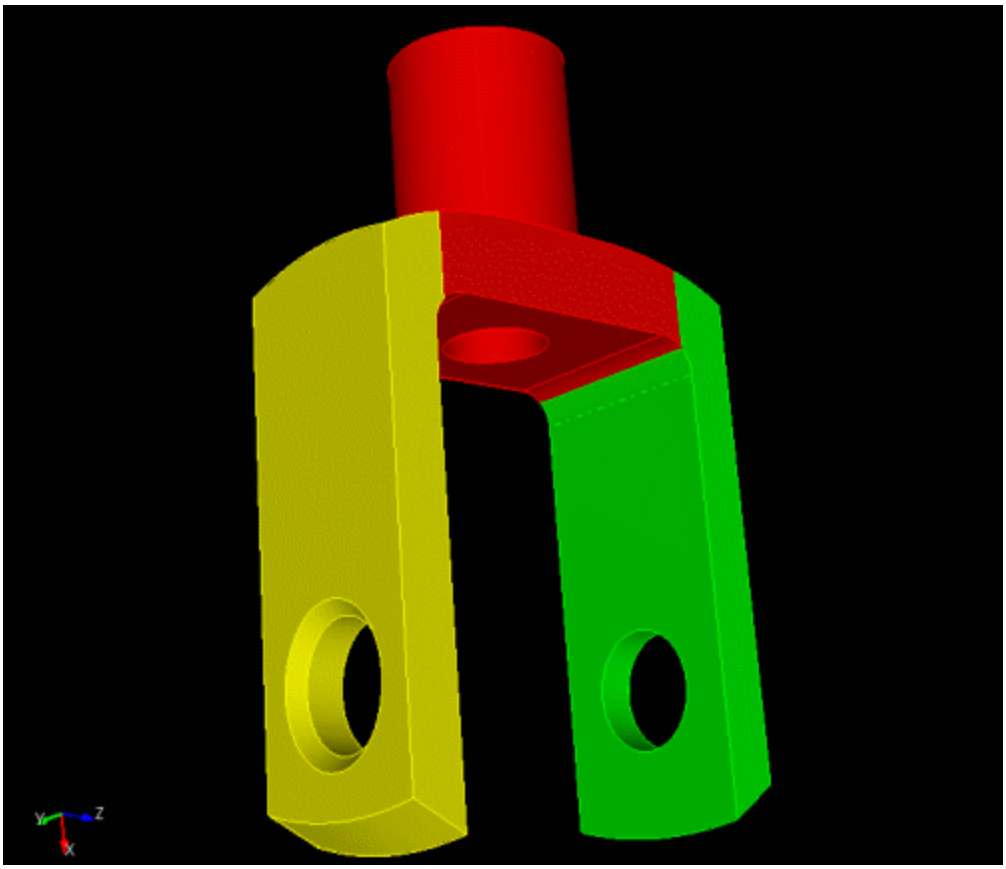
- Rotate the view window so **Surface 6** and **24** are visible



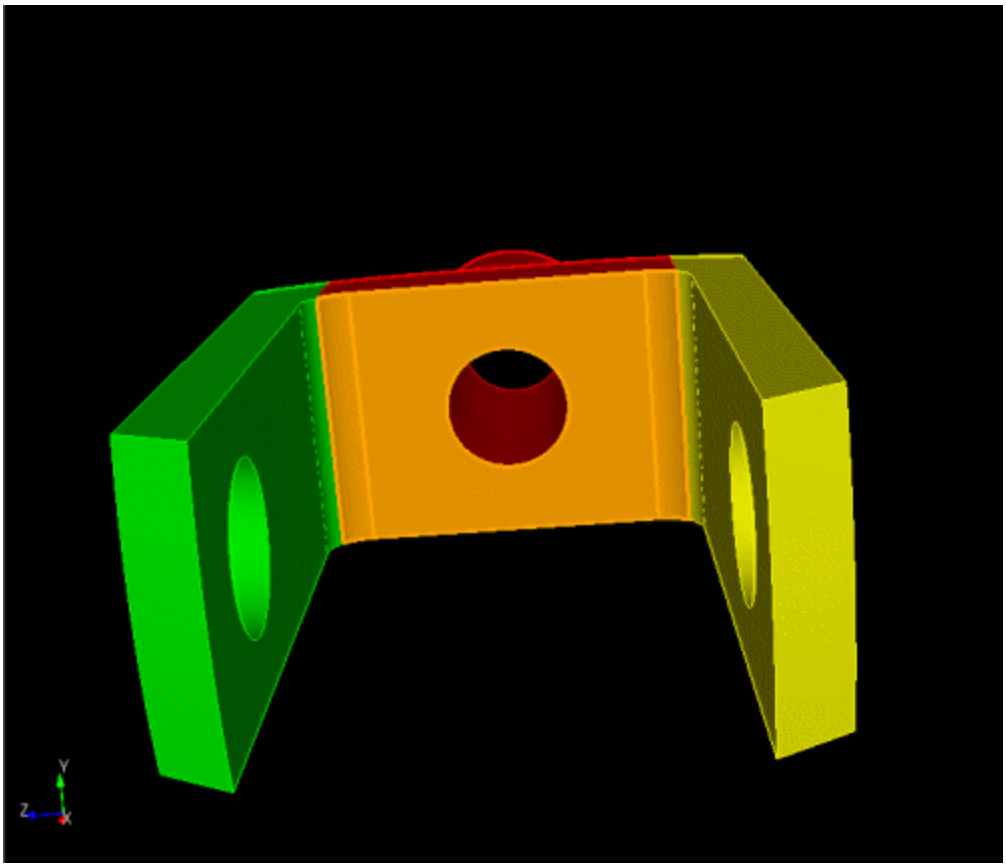
- Select **Surface 6** and **Surface 24** by using CTRL-Click (Windows), Command Key-Click (Macintosh), or entering the ids the input window
- Press Apply



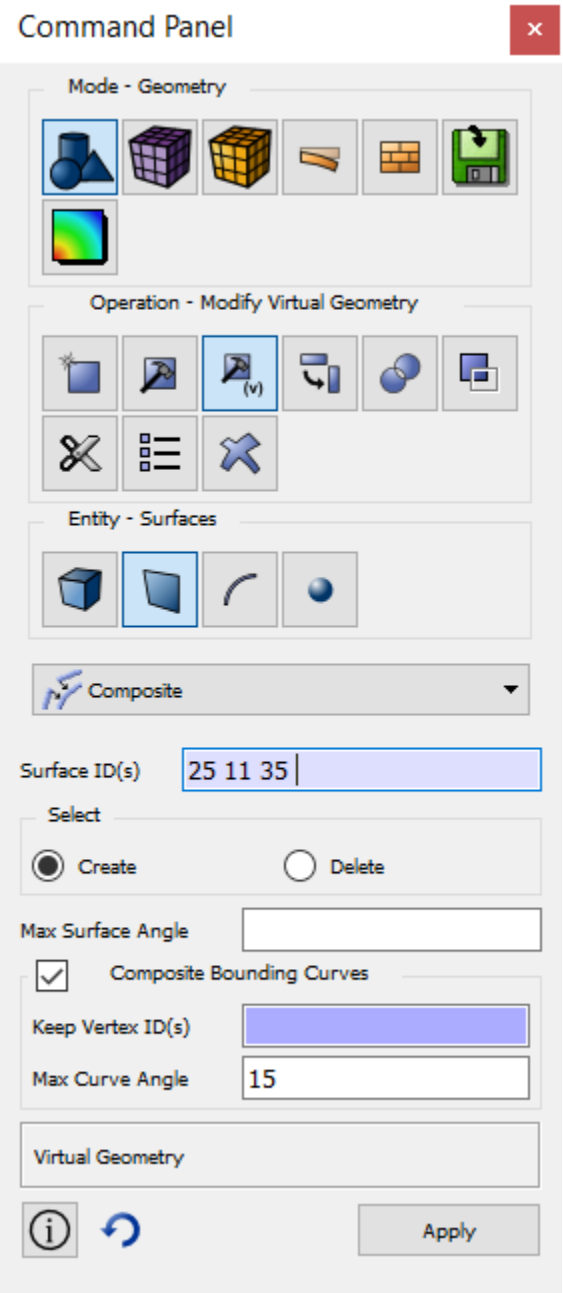

Check to see that the surfaces have been composited and that your graphics window looks like the following image.



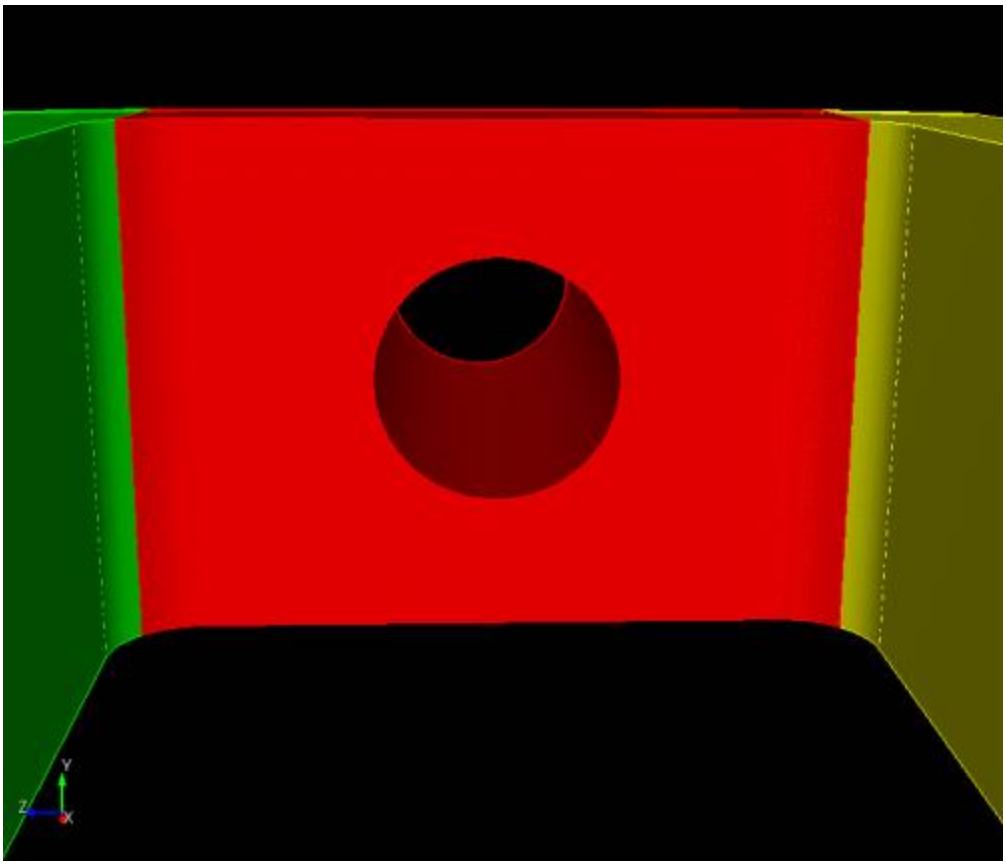
Finally, surfaces 11, 25, and 35 (shown below) need to be composited.




Use the command panel to choose surfaces for the composite command.




Press the **apply** button and check the results in the graphics window.

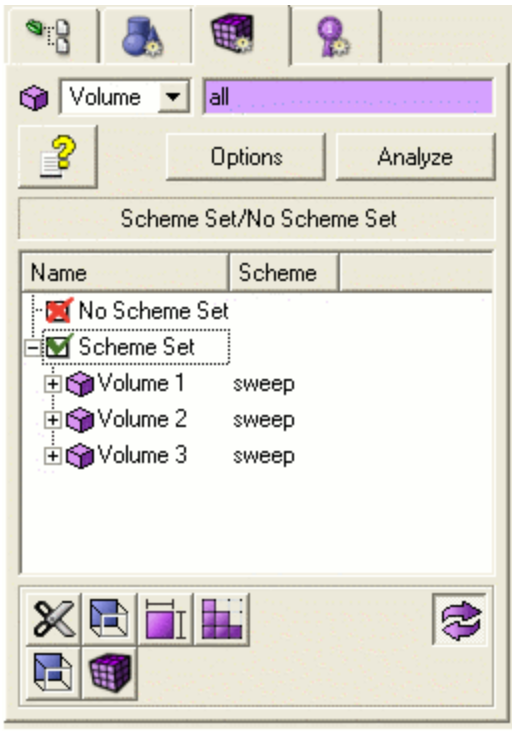


Power Tools GUI Tutorial Step 11

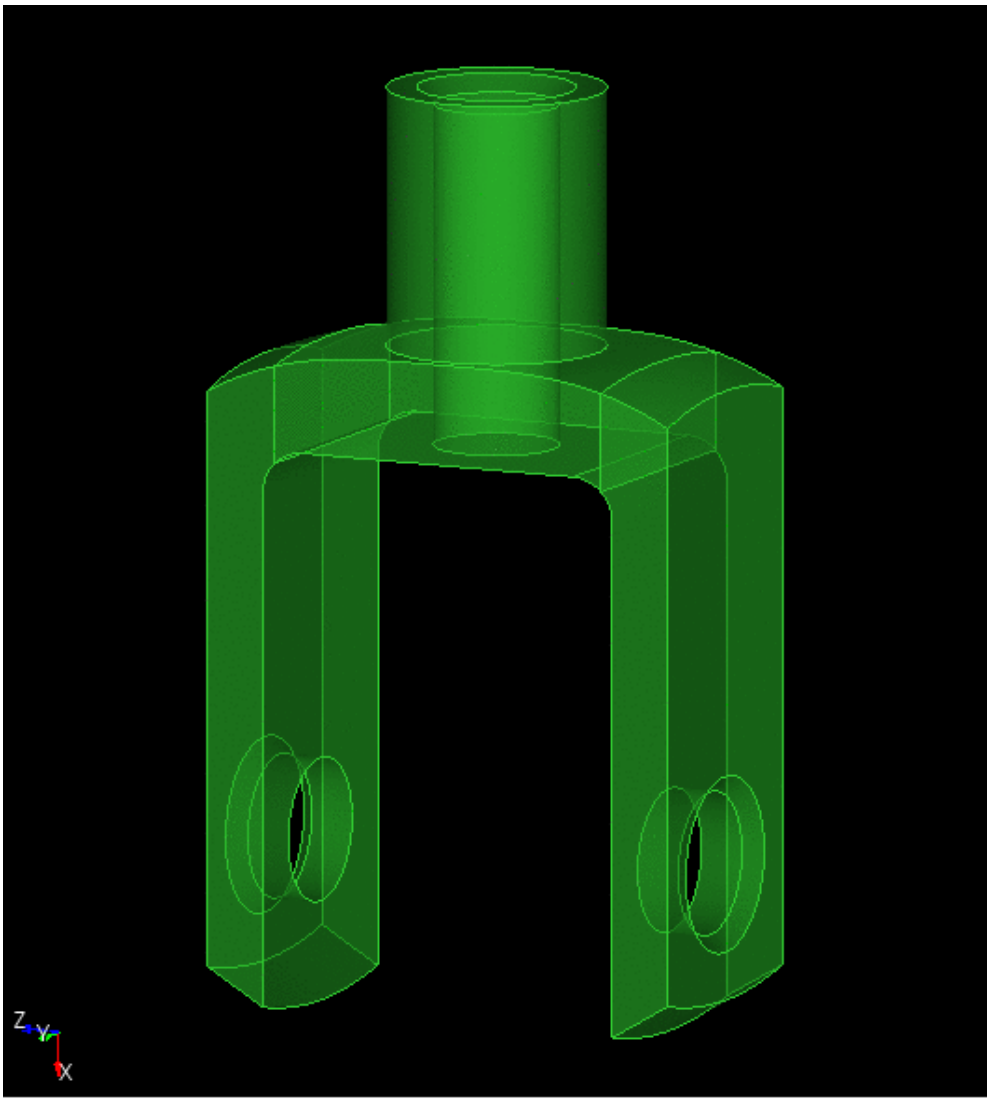
# *Step 11: Meshing the Model*

Use the Mesh Power Tools to apply schemes to the remaining volumes.

- Press the **Mesh Tools** tab in the Power Tools window
- Press **Analyze**



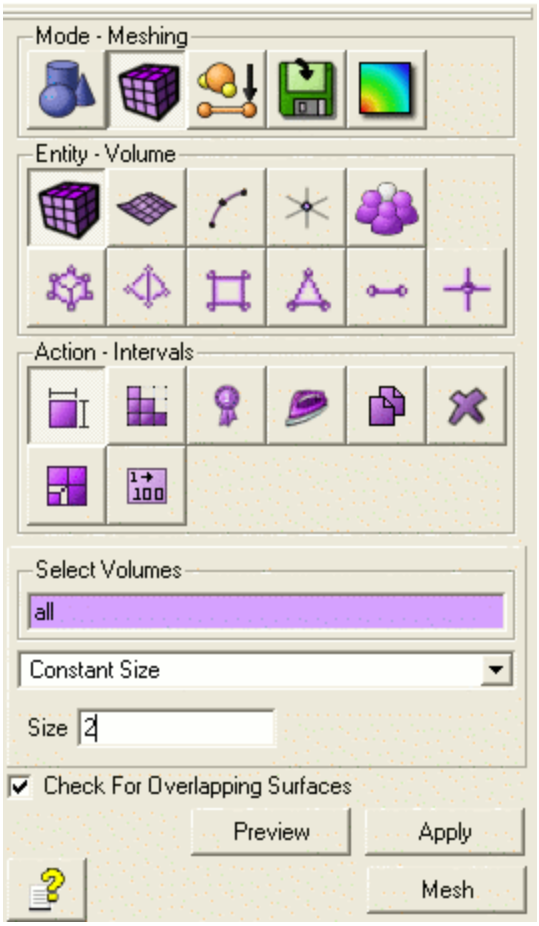

All of the schemes have now been set with a sweeping algorithm. The model is ready to be meshed. All volumes should appear green in the graphics window.



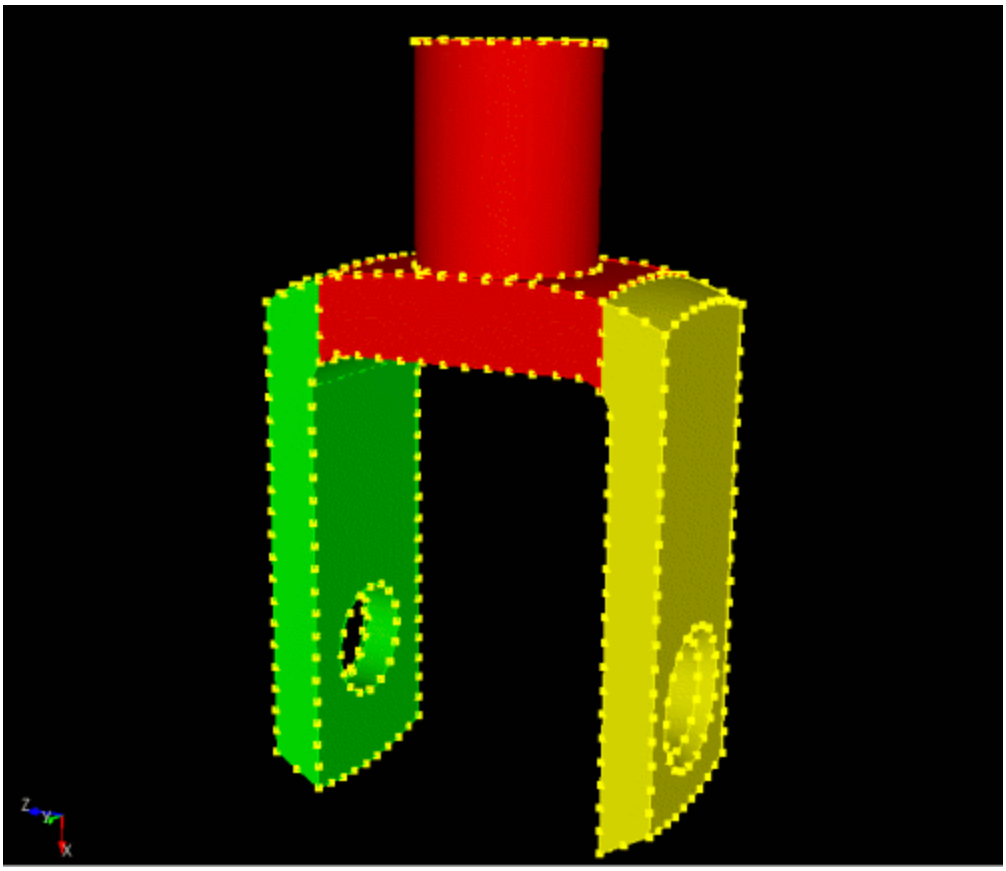

- Toggle the **Reset Graphics** button to return volumes to their original colors

Select **Volume** as the entity, and **Intervals** as the Action.

- Enter **all** in the "Select Volumes" input window
- Select **Constant Size** from the list of sizing options
- Enter **2** for the size
- Press **Apply Size**
- Press **Preview**

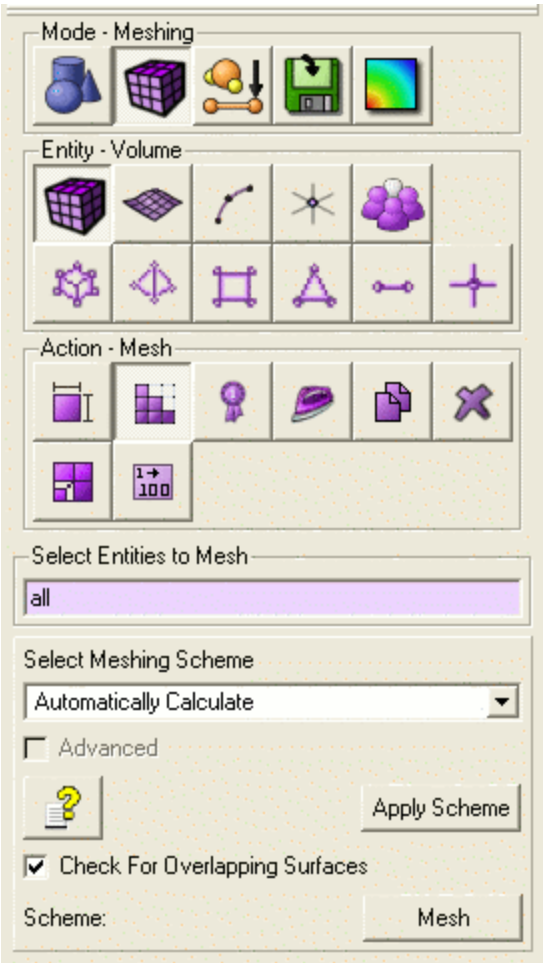



The graphics window should appear as follows, with the mesh size increments highlighted on all of the curves in the model.

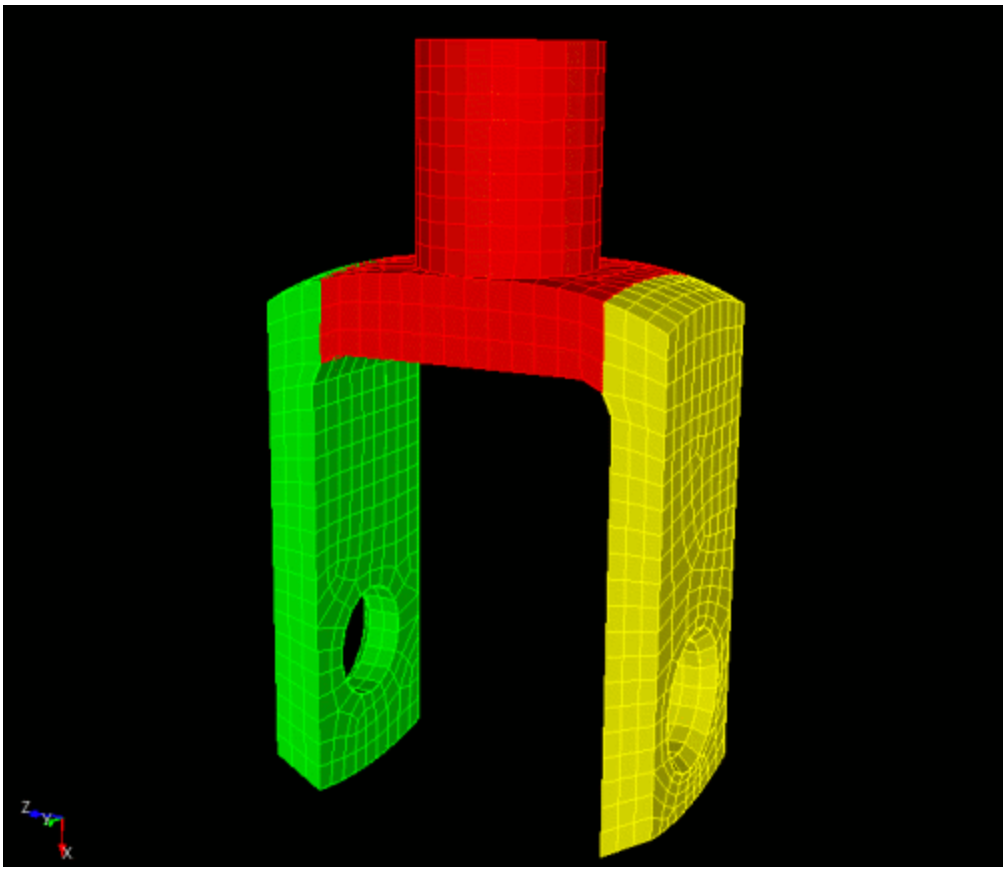


- Go to Mode - Meshing, Entity - Volume, Action - Mesh, and press the **Mesh** Button

There is no need to press the Apply Scheme button since the scheme have already been set in the Meshing Tools.




The final mesh should look like this:



Congratulations! You have just completed the Power Tools Tutorial. Click on the arrow to return to the Tutorial home.

# decomposition

## Decomposition Tutorial

*Creating Sweepable Volumes Through Webcutting*

Most volumes require some measure of decomposition before they can be meshed with a hexahedral meshing scheme. The most common hexahedral meshing tool is the sweeping algorithm. Sweeping is the process of creating a hexahedral mesh by extruding a quadrilateral surface mesh from a source surface onto a topologically similar target surface by way of a linking surface. The surface mesh can be meshed with any surface meshing scheme (i.e. structured or unstructured mesh), but the most common surface meshing scheme for the sweeping algorithm is the pave scheme. In fact, the sweeping algorithm is sometimes called the "pave-sweep" algorithm. Most volumes aren't automatically sweepable, which is why geometry decomposition is so important to the meshing process. Decomposition usually involves a series of webcutting, boolean, and virtual geometry operations that break up a larger model into sweepable regions. Studies have shown that this step in the meshing process is the most time consuming for the analyst. The goals of this tutorial are for the user to learn to:

1. Recognize sweepable volumes
2. Recognize how to decompose a model into sweepable parts
3. Gain proficiency with webcutting and other decomposition techniques
4. Avoid common pitfalls with decomposition and sweeping

*Why use sweeping?*

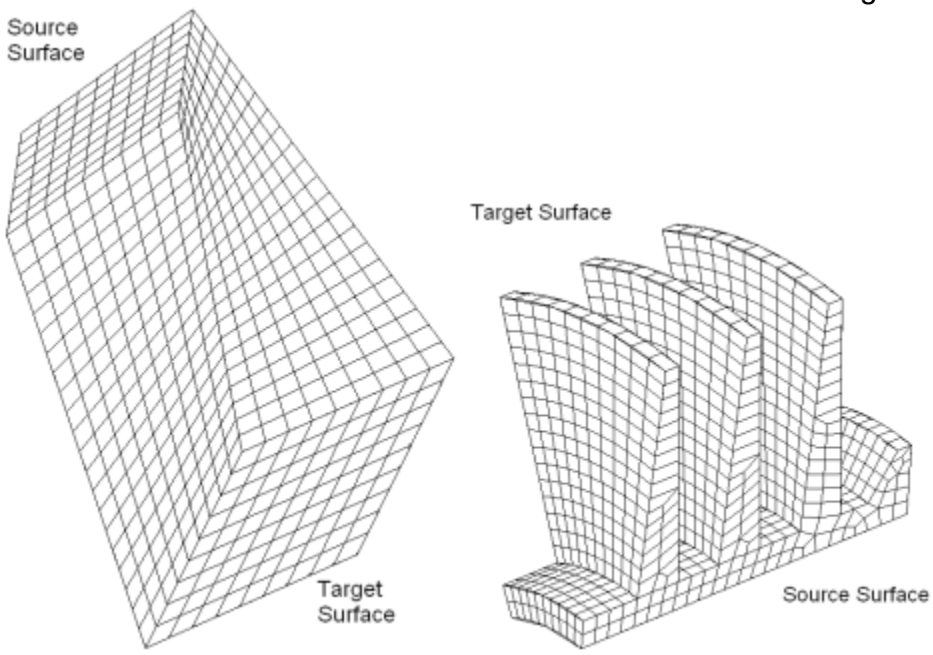
Of all the hexahedral meshing schemes in the Cubit toolkit, sweeping is considered the most reliable at producing high quality elements. Although decomposing a model into sweepable volumes can be time-consuming, and sometimes falls into the realm of trying to fit a square peg into a round hole, the pave-sweep algorithm has a high rate of success, and it sometimes the only way to get a hexahedral mesh on a model.

*What makes a volume sweepable?*

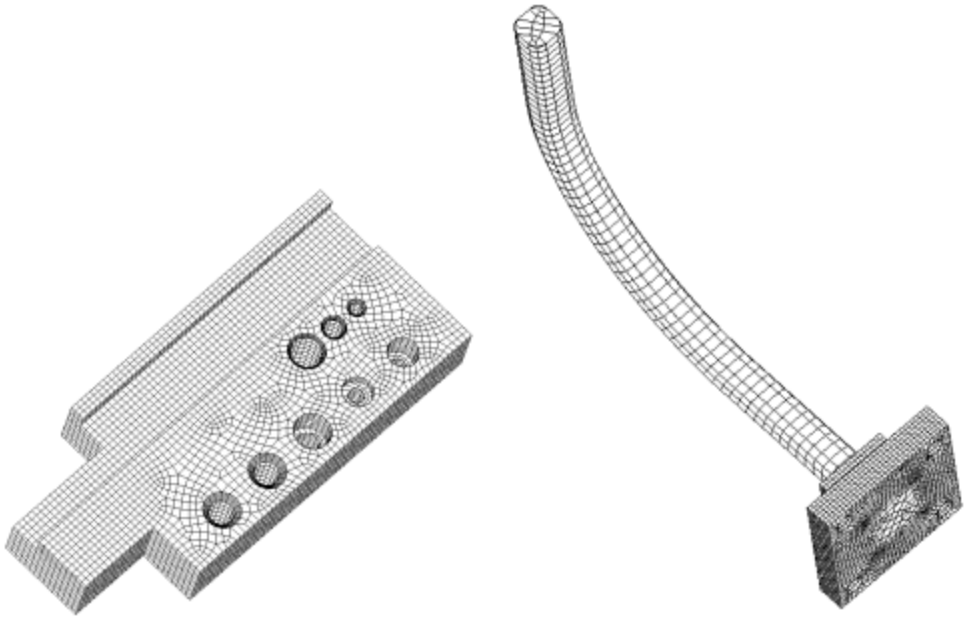
Recognizing sweepable topologies can be an art form. Sweepable volumes can be comprised of many different topologies. We typically classify sweeping problems into three groups, based on the number of source/target surfaces.

#### Basic Sweep Groups

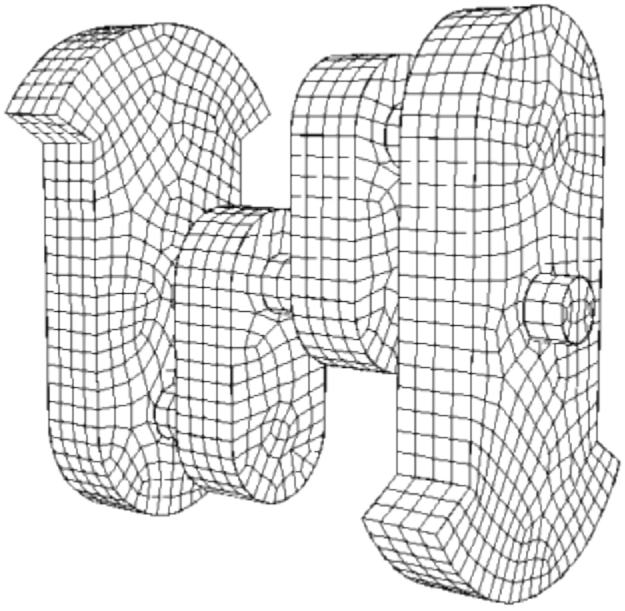
**One-to-one:** A volume with a one source surface and one target surface.




**Many-to-one:** A volume with multiple source surfaces and one target surface



**Multisweep (or Many-to-Many):** A volume with multiple target surfaces



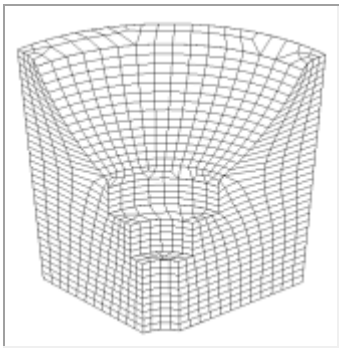
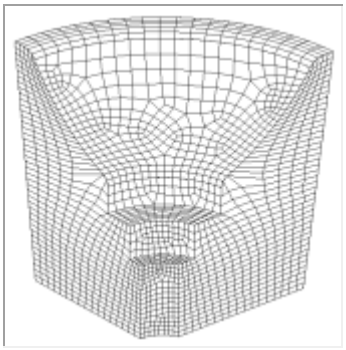
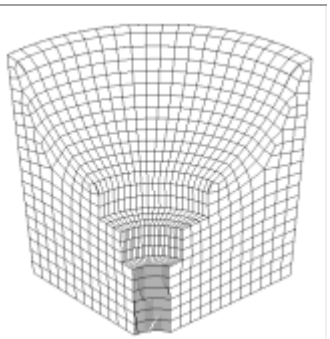
**Points to consider when determining whether a volume is sweepable**

- Swept surface meshes can be extruded through a volume which is rotated or translated. However, if the translation/rotation is severe then the quality of the resulting mesh may be poor.
- A volume with multiple target surfaces and a single source surface can sometimes be inverted and handled as a many-to-one sweepable volume. Otherwise, it is treated as a multisweep problem.
- Imprinting introduces new topology onto surfaces. Sweepable volumes may not be sweepable after imprinting and merging adjacent surfaces

- Multisweep is still under development, and has limitations, so if you are having difficulty with the multisweep algorithm, it is usually a good idea to decompose it into many-to-one or one-to-one sweepable regions.
- Cubit won't always automatically recognize your volume as a sweepable volume, even if it is. Sometimes, you have to give it a list of source/target surfaces explicitly.

**Basic Sweep Paths**

In addition to the different topologies, sweepable volumes can be classified by the sweep direction. These include: top-to-bottom, inside-to-outside, and around (rotational). Be sure to consider all the possibilities for sweep directions when you begin decomposing a model. And keep in mind that sweep paths must be compatible with adjacent volumes. To be compatible, overlapping surfaces must have the same scheme (i.e. both must be a linking surface or a paved surface). The volume below is meshed three different times with the three different sweep directions. Notice the difference in element sizes and orientations between the meshes. See if you can pick out the different source and target surfaces in each example. As an exercise, try to mesh this model with each of the different sweep paths.

|  |  |  |
|---|---|---|
| Top-to-Bottom | Inside-to-Outside | Around (Rotational) |
| Many-to-one | Many-to-one | One-to-one (this is the default sweep direction for this model) |

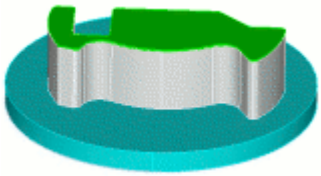
*What are some good strategies for decomposing my model?*

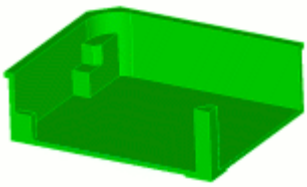
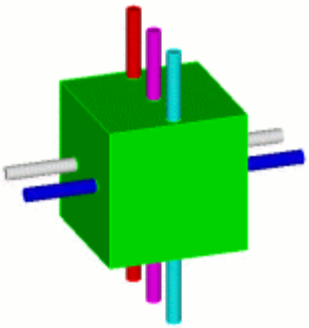
Recognizing when a volume is sweepable is a difficult task of itself, but being able to come up with viable webcutting, compositing, and boolean strategies to make a volume sweepable is even more difficult, and can only be achieved through practice. Here are some general principles to follow when decomposing a model.

1. Select your sweep path
2. Use as few webcuts as possible
3. Set your own source and target surfaces if Cubit does not pick them automatically

4. If one of your volumes does not mesh, or has an undesirable mesh, try changing the order in which you mesh volumes. This will hardset the intervals on the volumes.
5. The Reset Volume command will remove all schemes and interval settings from volumes.
6. If changing the mesh order or resetting the volumes does not work and you continue to get "Matching Intervals Failed" errors, set explicit intervals on some or all curves.
7. Make additional webcuts if necessary.
8. Check for sliver surfaces or curves that may have been introduced during decomposition and remove these through tweaking collapsing, or compositing.
9. Change surface vertex types on mapped or submapped surfaces if you need to force a certain configuration
10. Use partitioning to introduce virtual geometry constraints without affecting the underlying geometry
11. Composite surfaces to remove constraints without affecting the underlying geometry
12. Save your work often. For a complex model, the meshing process can be very iterative. You may need to start over many times until you find an acceptable solution.

The following is a compilation of several different decomposition problems of varying difficulty. If you accessed this help from the Cubit program (as opposed to the web documentation), you will need to browse for the geometry files from within your Cubit installation directory. They should be located in the "/components/cubit/help/step_by_step_tutorials/decomposition" directory of the Cubit installation folder.

| Example | Image | File |
|---|---|---|
| Beginner | | |
| Sweeping through multiple adjacent volumes |  | example01.sat |
| Interlocking rings |  | example02.sat |

| | | |
|---|---|---|
| Webcutting using the "sweep" option |  | example03.sat |
| Using the loft command |  | example04.sat |
| Multiple sweep directions |  | example05.sat |
| Advanced | | |
| Employing symmetry and controlling skew |  | example06.sat |

| | | |
|---|---|---|
| Using virtual geometry |  | example07.sat |
| Sweeping volumes with narrow angles and surfaces |  | example08.sat |

## Example 1. Sweeping multiple adjacent volumes

The following model has several interior volumes which share surfaces. This example may at first seem complex, but it actually requires very little decomposition. The key to this example is that each of the interior volumes is already sweepable, oriented along the same sweep axis, and none of the linking surfaces have additional topology introduced through imprint/merge. In fact, there is only one required webcut to make this model automatically sweepable.

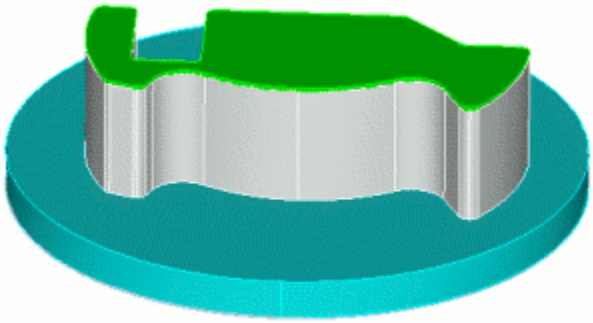


Figure 1. Exterior view

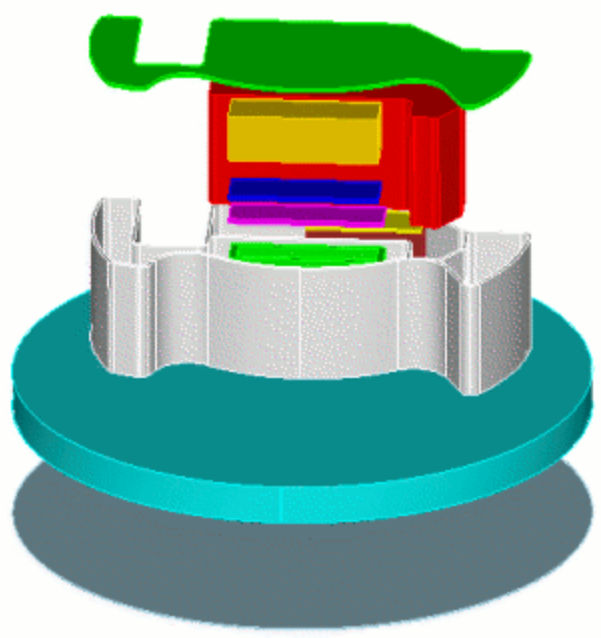
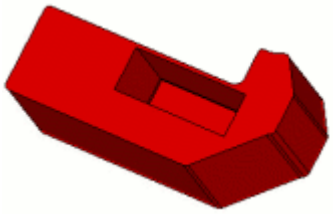
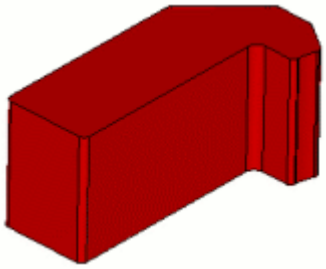
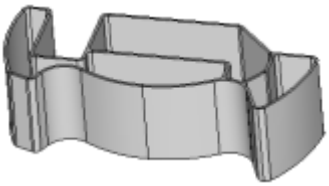
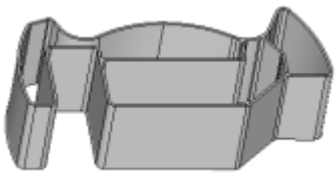
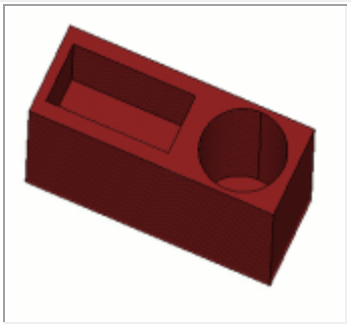
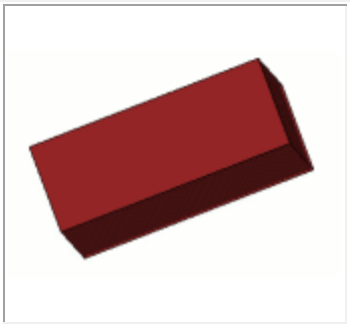
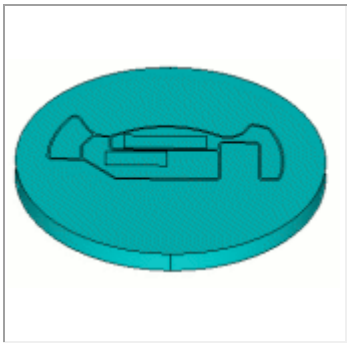
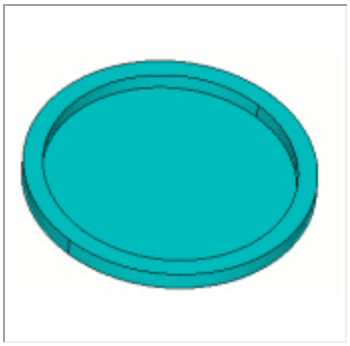


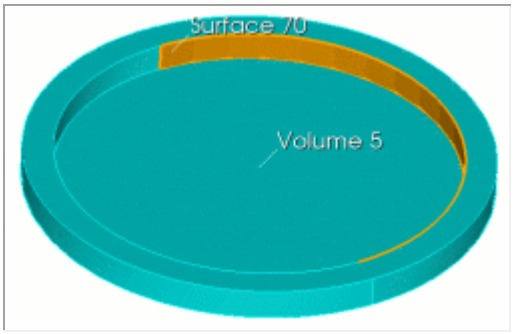
Figure 2. Interior view

We examine several of the volumes below.

| Source Surface(s) | Target Surface(s) | Sweep type |
|---|---|---|
|  |  | **Many-to-one Sweepable** Imprinting and merging adjacent volumes creates additional partitions on the source surface, but the target surface does not contain imprints. |
|  |  | **Many-to-one Sweepable** Multiple source surfaces due to interior void |
|  |  | **One-to-One Sweepable** Source and target surfaces are single surfaces, and there are no imprints on the linking surfaces |

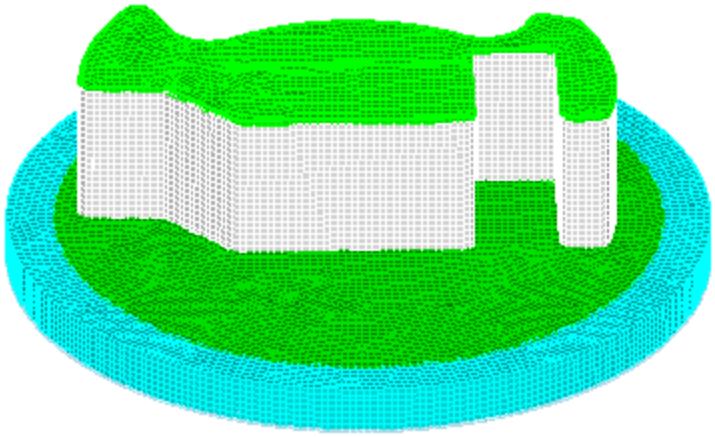
| | | |
|---|---|---|
|  |  | **Many-to-one Sweepable**<br>Interior void causes multiple source surfaces. |
|  |  | **Multisweep**<br>Imprinting causes multiple source surfaces and interior void causes multiple target surfaces. This volume requires decomposition |

*Suggested webcut*

| Webcut | Command |
|---|---|
|  | **CUBIT> webcut volume 5 with sheet extended from surface 70**<br>**CUBIT> imprint all**<br>**CUBIT> merge all**<br>**CUBIT> volume all size 0.15**<br>**CUBIT> volume all scheme auto** |

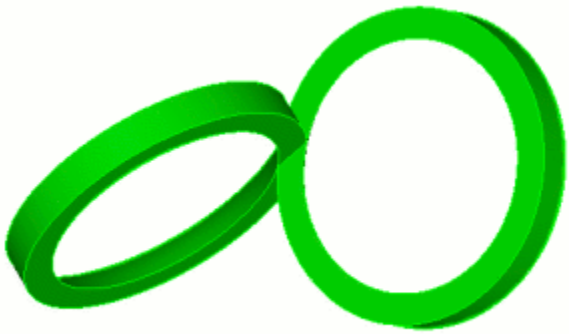
*Final mesh*

The final mesh is created at a size of 0.15 for all volumes.



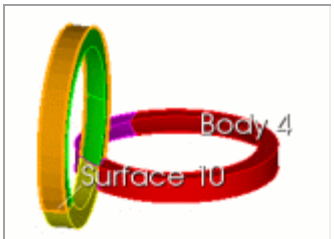
## Example 2. Interlocking rings

The following example is composed of two rings of constant cross-section that can be swept along their axes. The problem here is that the rings overlap, forming a tetrahedral shape which cannot be swept. The key to solving this problem is separating out the region of overlap, explicitly setting the source and target surfaces, and using the tetprimitive scheme on the tetrahedral region.

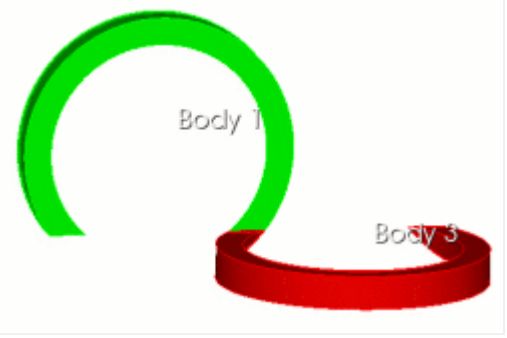
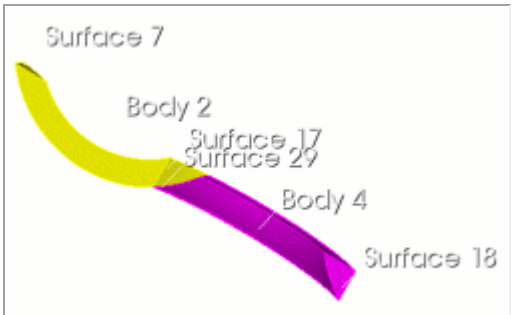


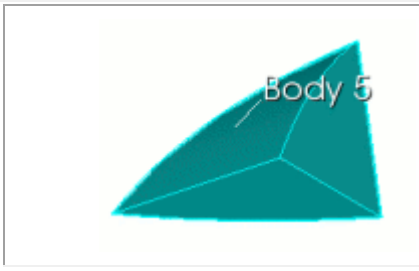
*Suggested webcuts*

| Webcut | Command |
| --- | --- |
|   | `CUBIT> webcut body 1 plane surface 5` |

| | |
|---|---|
|  | **CUBIT> webcut body 2 sheet extended from surface 4** |
|  | **CUBIT> webcut body 3 plane surface 12** |
|  | **CUBIT> webcut body 4 sheet extended from surface 10**<br>**CUBIT> imprint all**<br>**CUBIT> merge all** |

There are five volumes that result from the webcutting. Two of them are automatically sweepable. Two of them must have their schemes set explicitly, and one of them is meshed using the tetprimitive scheme.

| Webcut | Command |
|---|---|
|  | One-to-one Sweepable<br>Source and target are set automatically using autoscheme<br>**CUBIT> volume 1 3 scheme auto** |
|  | One-to-one Sweepable<br>Must have source and target set explicitly<br>**CUBIT> volume 2 scheme sweep source 17 target 7**<br>**CUBIT> volume 4 scheme sweep source 29 target 18** |

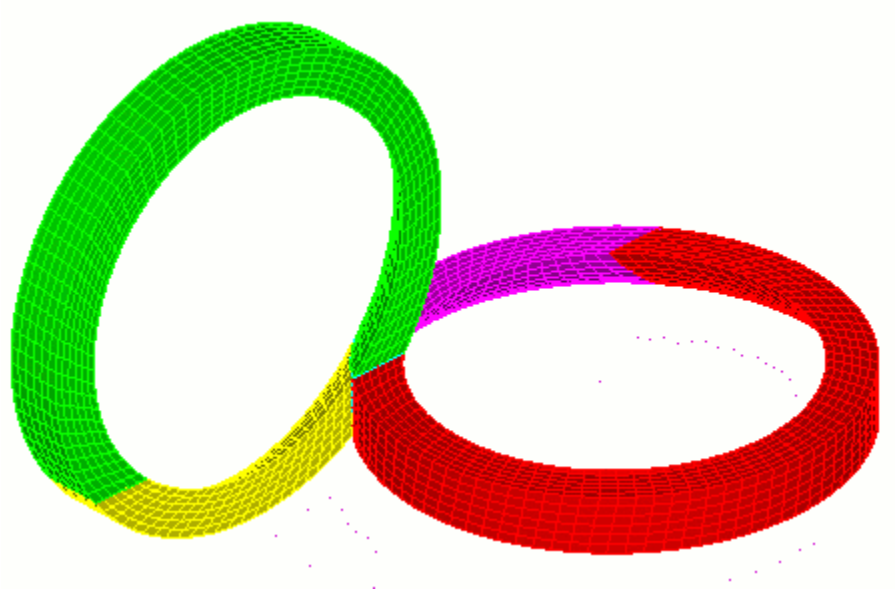



Non-sweepable
Use the tetprimitive scheme

```
CUBIT> curve in volume 5
interval 6
CUBIT> volume 5 scheme
tetprimitive
CUBIT> volume all size 0.5
CUBIT> mesh volume all
```

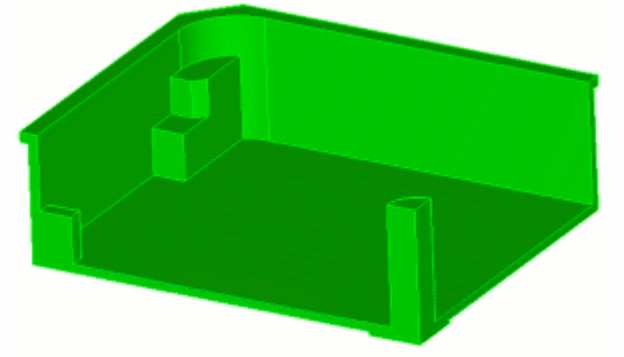
*Final mesh*

The final mesh is created at a size of 0.5 for all volumes.

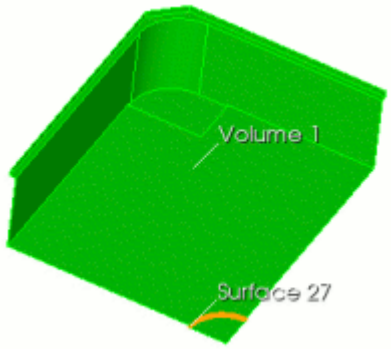
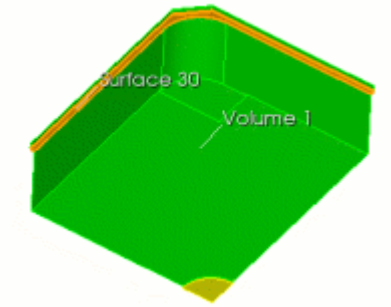
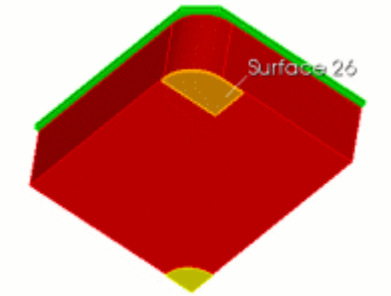


## Example 3. Webcutting using the sweep option

This example introduces additional webcutting options. This example would be a simple many-to-one sweep except for the overhanging lip and the protrusions on the bottom surface. To a beginner user, it may at first seem reasonable to use the bottom surface as a webcutting plane. However, this will not create a many-to-one sweepable volume. Instead, you need to use the protruding surfaces as cutting planes, and extend them through the entire volume.

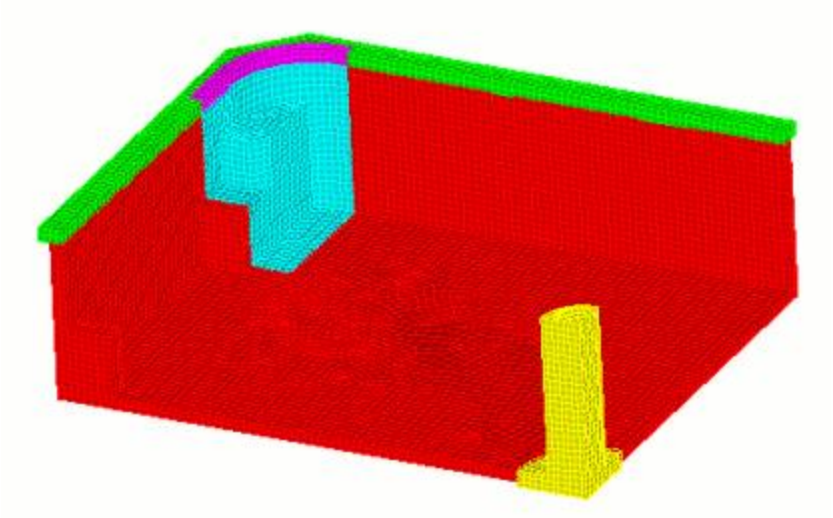


*Suggested webcuts*

| Webcut | Command |
| --- | --- |
|  | **CUBIT> webcut volume 1 with sheet extended from surface 27** |
|  | **CUBIT> webcut volume 1 with plane surface 30** |
|  | **CUBIT> webcut vol all sweep surf 26 vector -1 0 0 through_all**<br>Now Volume 3 (red) has only 1 target surface.<br>**CUBIT> imprint all**<br>**CUBIT> merge all**<br>**CUBIT> volume all size 0.05**<br>**CUBIT> mesh volume all** |

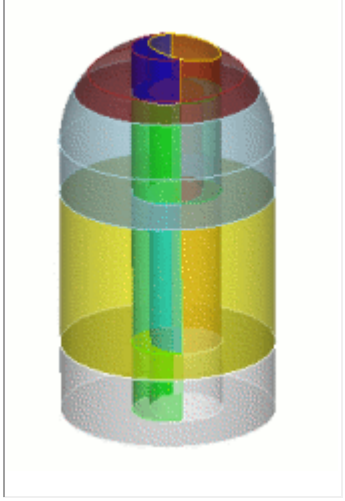
*Final mesh*

The final mesh is created at a size of 0.05 for all volumes.



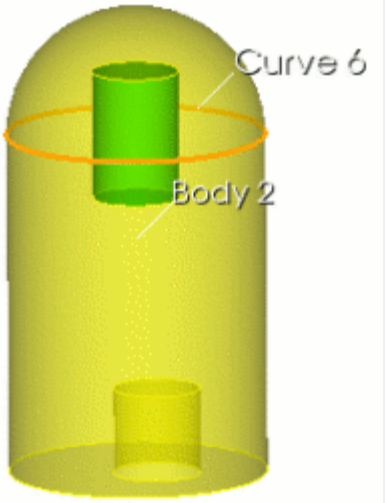
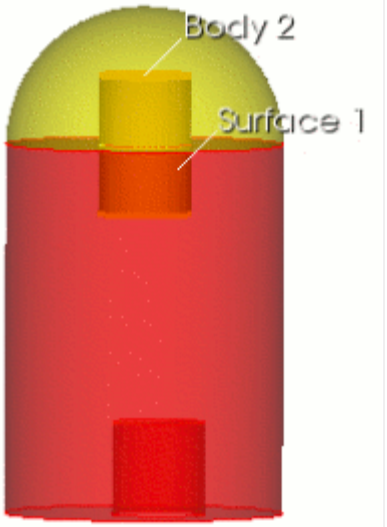
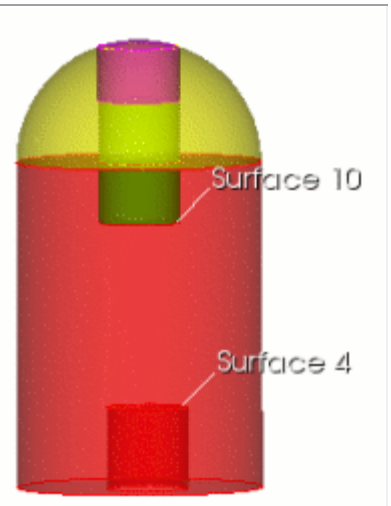
## Example 4. Using the Loft command

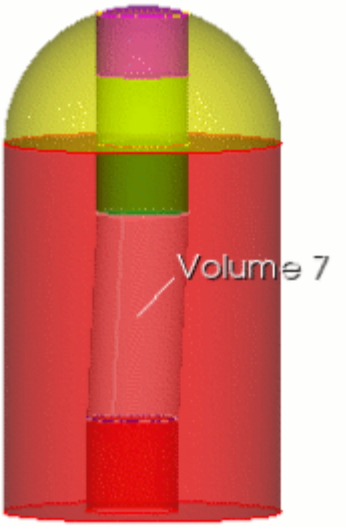
In the next example, the loft command significantly decreases the number of required webcuts. This model also demonstrates using 2 separate sweep paths (top-to-bottom and rotational) on adjacent volumes.

| | | |
|---|---|---|
|  |  |  |
| Original Volume | Webcuts created from sweeping surfaces (not recommended) | Webcuts using loft command (recommended) |

*Suggested webcuts*

| Webcut | Command |
|---|---|

| | |
|---|---|
|  | **CUBIT> webcut body 2 loop curve 6** |
|  | **CUBIT> webcut body 2 sheet extended from surface 1** |
|  | **CUBIT> create surface from surface 10**<br>**CUBIT> create surface from surface 4**<br>**CUBIT> create body loft surface 19 20** |



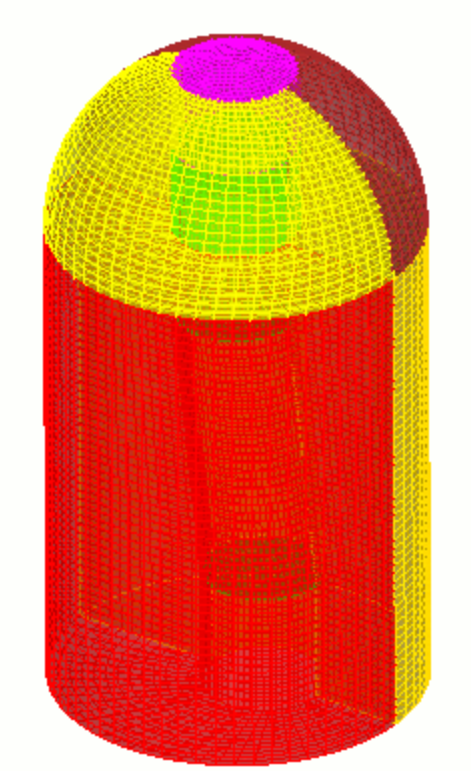

```
CUBIT> webcut body 3 tool body 7
CUBIT> delete body 5 6 7
```




```
CUBIT> webcut body 2 3 plane yplane
CUBIT>          imprint          all
CUBIT>           merge           all
CUBIT>   volume   all   size   0.15
CUBIT> mesh volume all
```

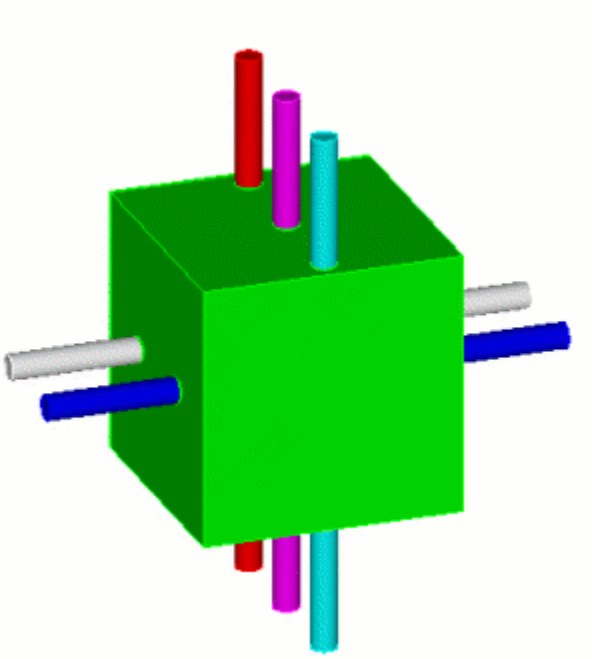
*Final mesh*

The final webcut model consists of a central shaft which can be swept top to bottom, and a surrounding casing which can be swept around. This is possible because the shared surface is a linking surface for both types of sweeps. The final mesh is created with a size of 0.15

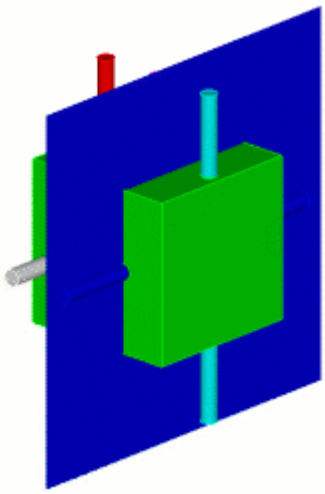
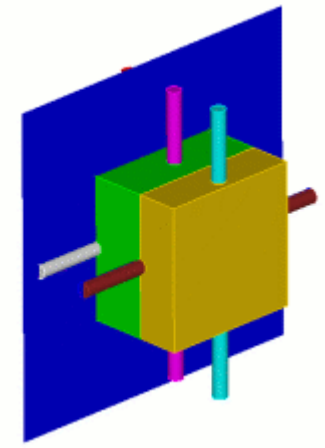


Example 5. Multiple sweep directions

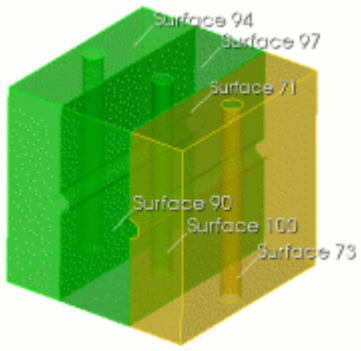
The next example gives another example of using different sweep directions on the same model. The following model shows a brick which is perforated by several cylindrical shafts. The shafts do not intersect each other.



*Suggested webcuts*

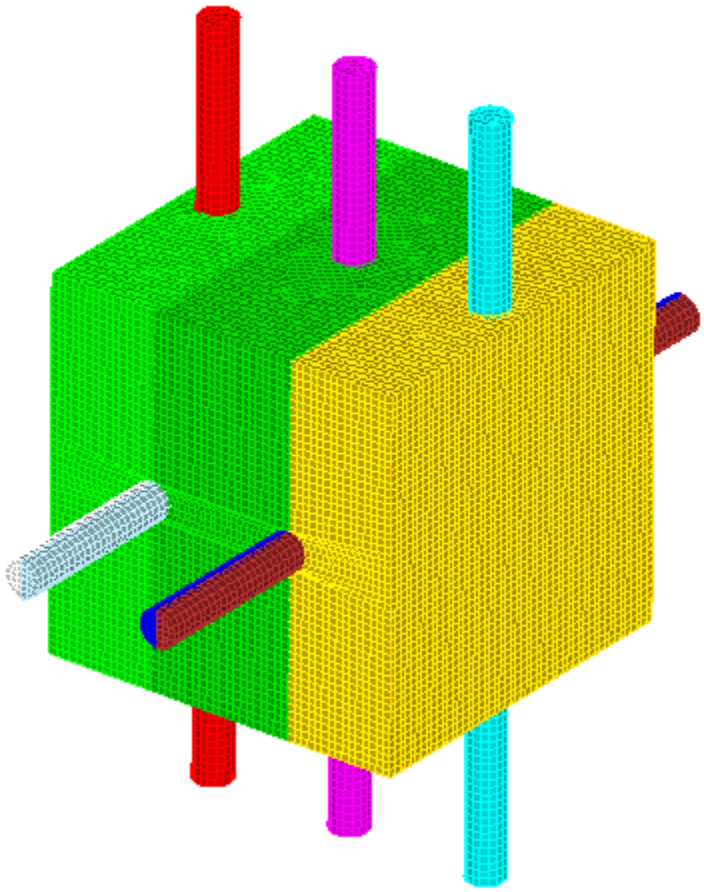
| Webcut | Command |
| --- | --- |
|  | `CUBIT> webcut volume all with plane yplane offset 20` |
|  | `CUBIT> webcut volume all with plane yplane offset -20`<br>`CUBIT>imprint all`<br>`CUBIT>merge all` |

All of the volumes in this model are now one-to-one sweepable. However, the source and target surfaces for the main block portions must be set explicitly

| | |
| --- | --- |
|  | `CUBIT>volume 8 scheme Sweep source surface 94 target surface 90 rotate off`<br>`CUBIT>volume 10 scheme sweep source surface 71 target surface 73 rotate off`<br>`CUBIT>volume 12 scheme Sweep source surface 97 target surface 100 rotate off`<br>`CUBIT>volume all size 2`<br>`CUBIT>mesh volume all` |

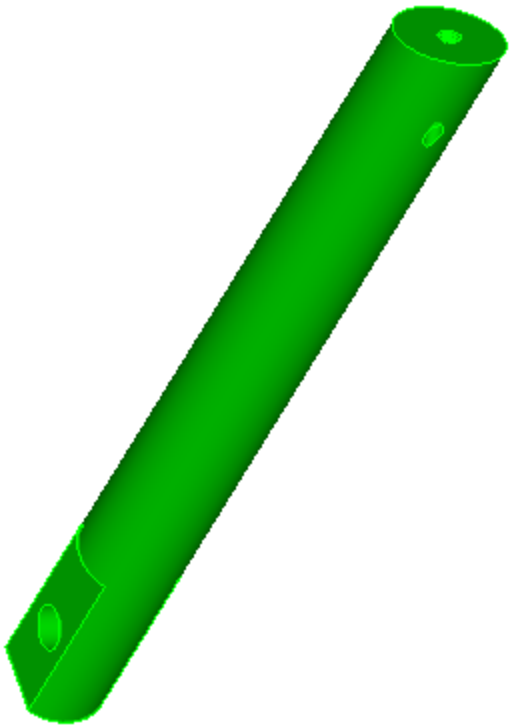
*Final mesh*

In this model it is possible to have different sweep directions since the surfaces which overlap are both linking surfaces. The final mesh is created with a mesh size of 2 and is shown below.



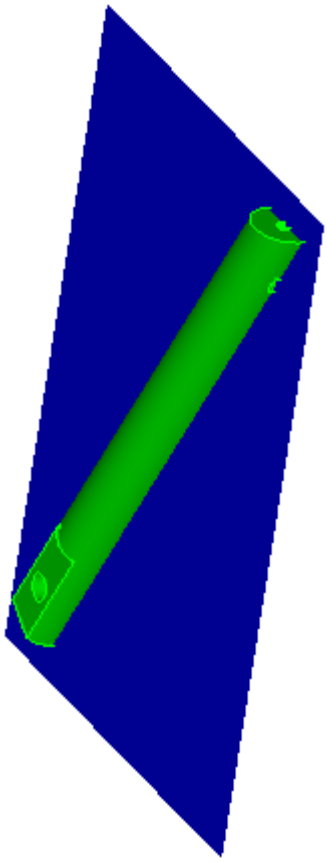
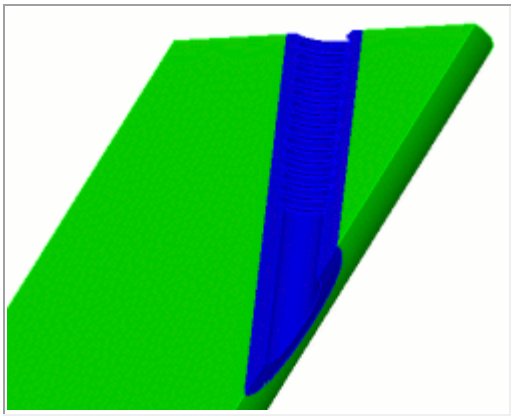
## Example 6. Employing Symmetry

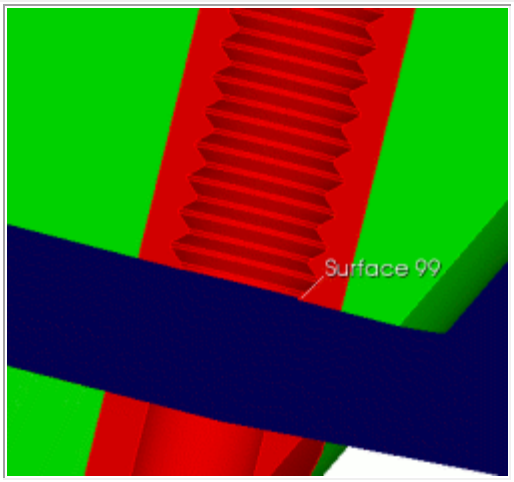
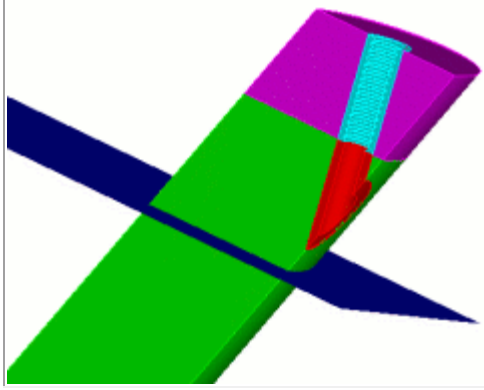
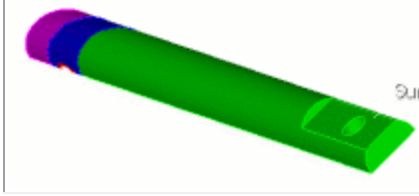
One technique for creating a symmetric mesh on a symmetric model is to mesh only half of the volume, then copy the mesh onto the other half. The following example employs this technique. This model at first appears quite simple, but it actually requires a good deal of webcutting to get a reasonable mesh that is not highly skewed.

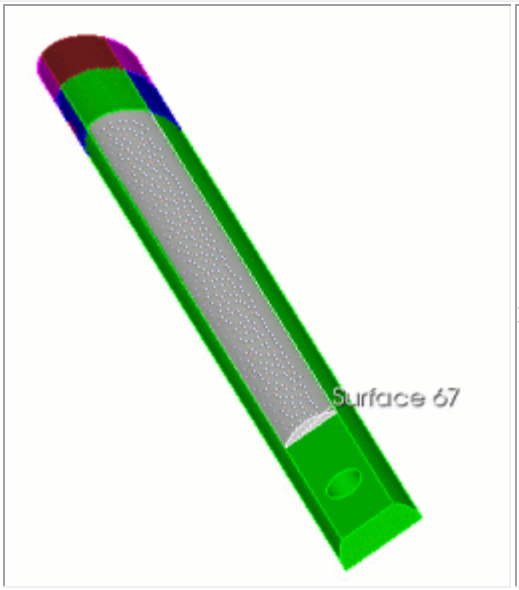


*Suggested webcuts*

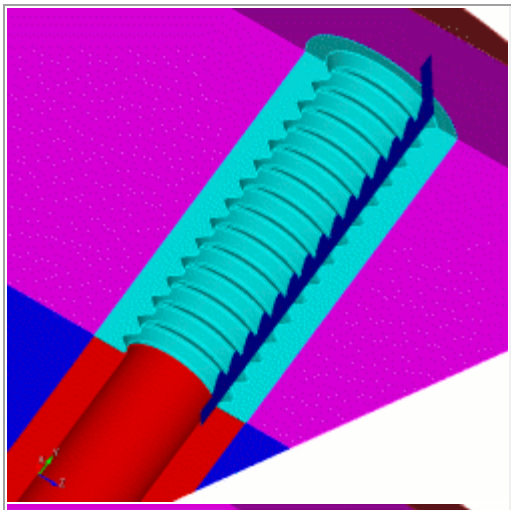
| Webcut | Command |
| --- | --- |

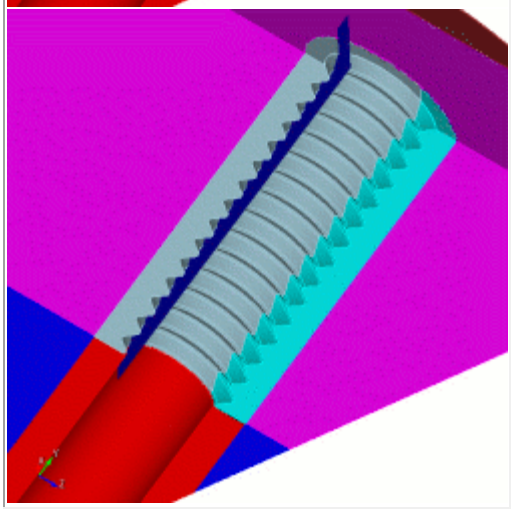
| | |
|---|---|
|  | CUBIT> webcut body 1 with plane xplane offset 0<br>CUBIT> delete body 2 |
|  | CUBIT> webcut body 1 with cylinder radius 2.75 axis y |

| | |
|---|---|
|  | **CUBIT> webcut body 1 3 with plane yplane offset 0** |
|  | **CUBIT> webcut body 1 with plane yplane offset -15** |
|  | **CUBIT> webcut body 1 6 4 with plane surface 64** |

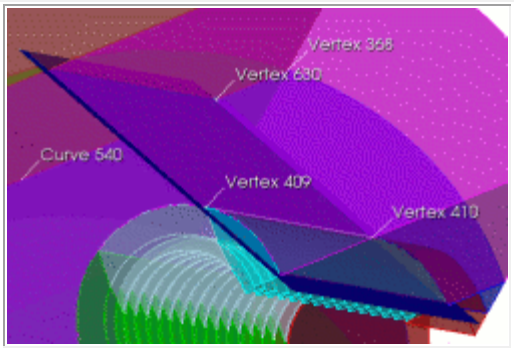



```
CUBIT> webcut body 1 with
plane surface 67
```

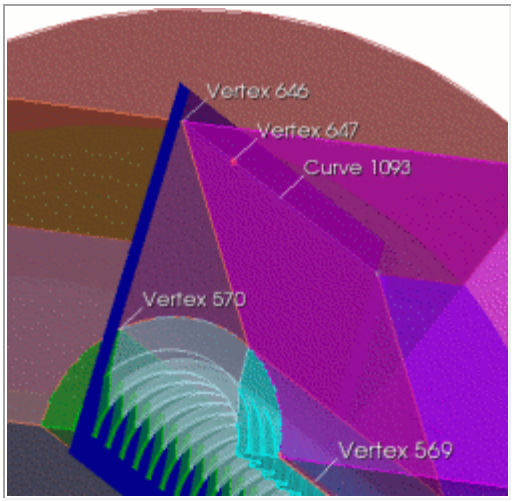




```
CUBIT> webcut body 5 with
plane zplane offset 1.5
CUBIT> webcut body 11 with
plane zplane offset -1.5
```

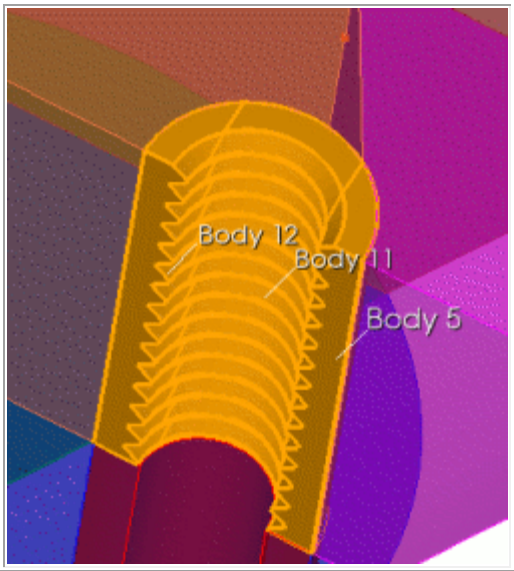



```
CUBIT> create vertex on curve 540 distance 2 from vertex 368
CUBIT> webcut body 4 with plane vertex 409 vertex 410 vertex 630
```

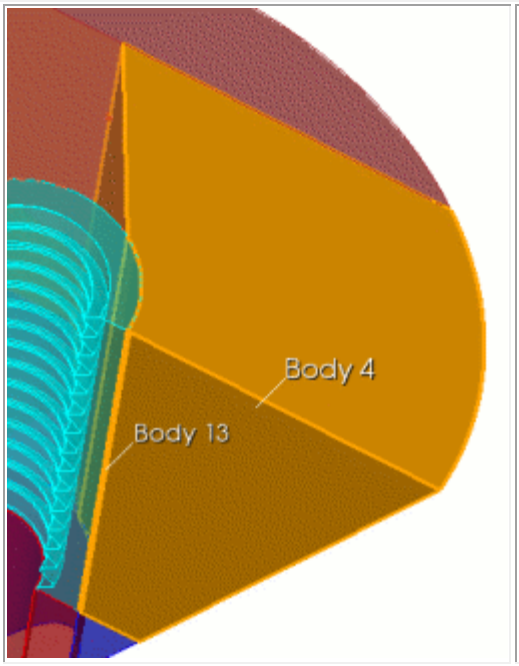



```
CUBIT> create vertex on curve 1093 distance 3 from vertex 646
CUBIT> webcut body 14 with plane vertex 570 vertex 569 vertex 647
```

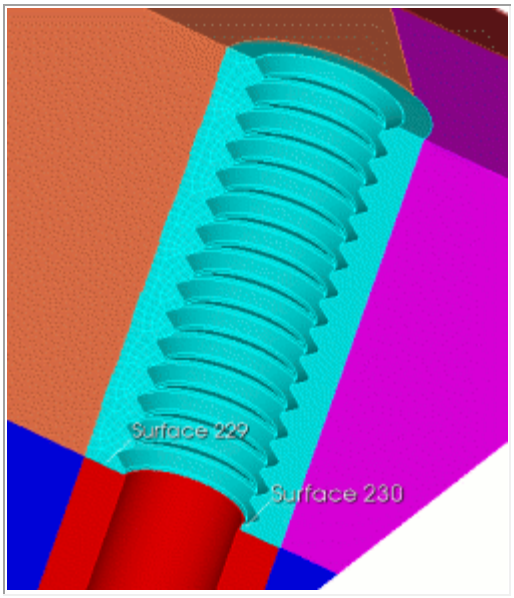
This wedge shape webcut is a method of controlling skew in the final mesh.



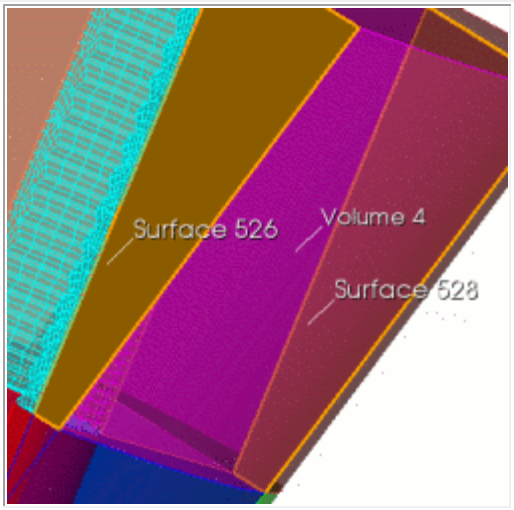
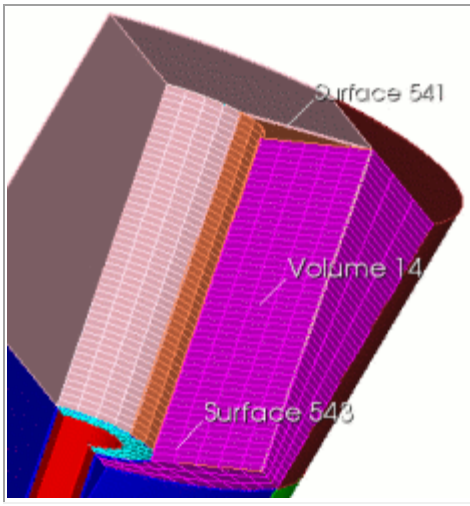
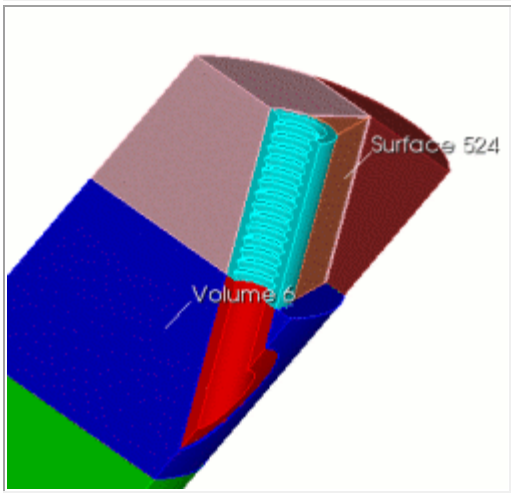

```
CUBIT> unite body 5 11 12
```

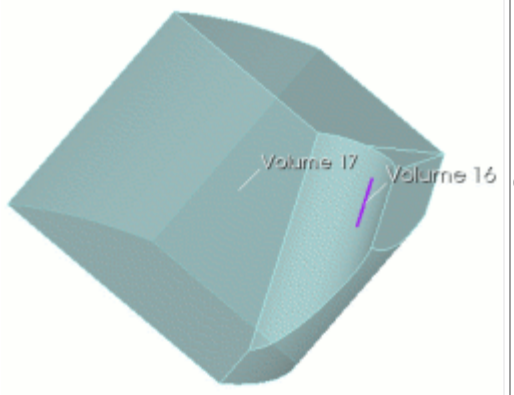
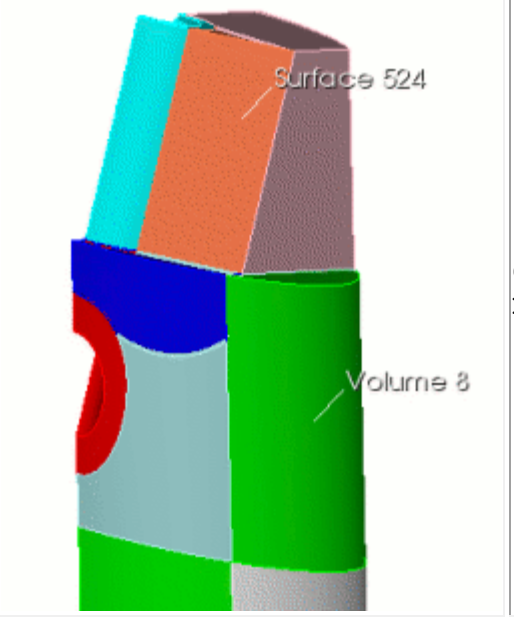


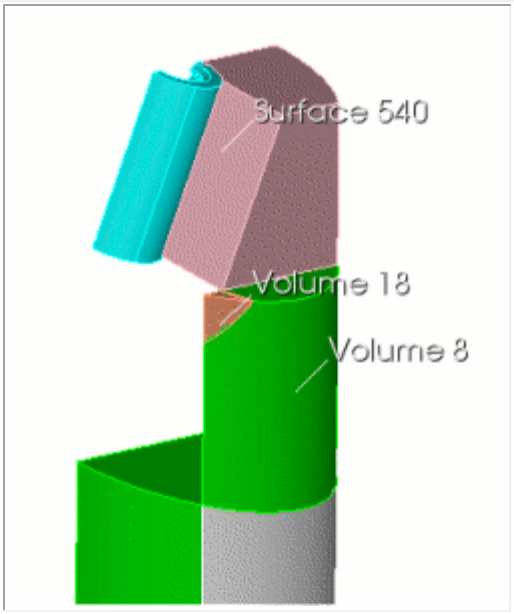
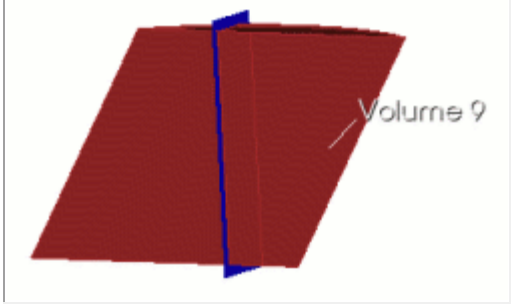
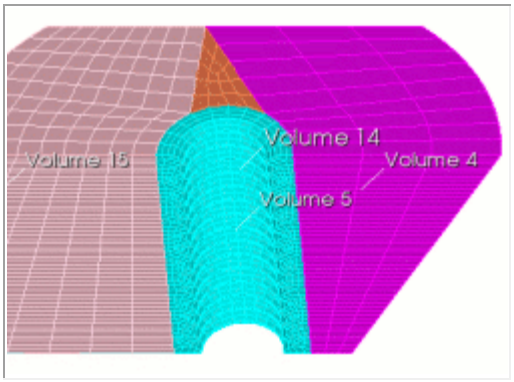
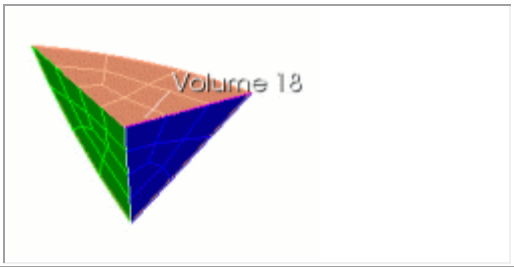

```
CUBIT> unite body 4 13
CUBIT> delete vertex all
CUBIT> imprint all
CUBIT> merge all
CUBIT> vol all size .5
```

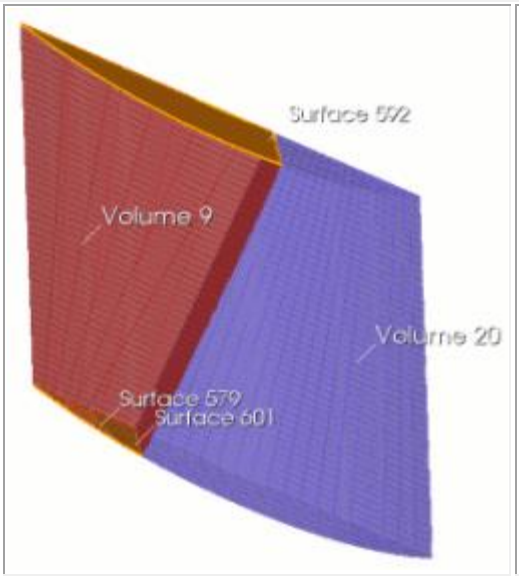
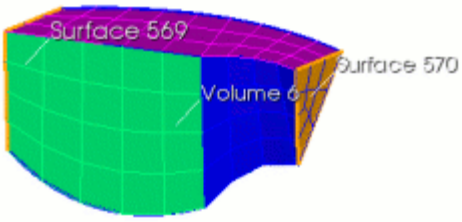
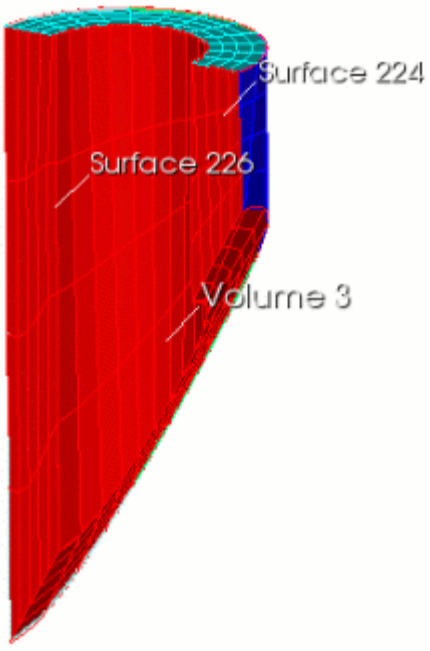


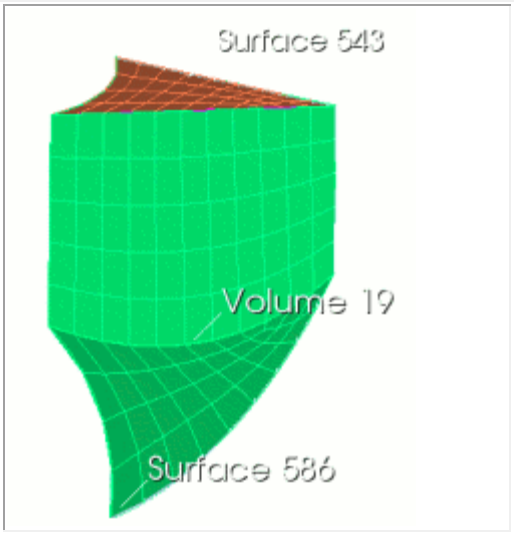
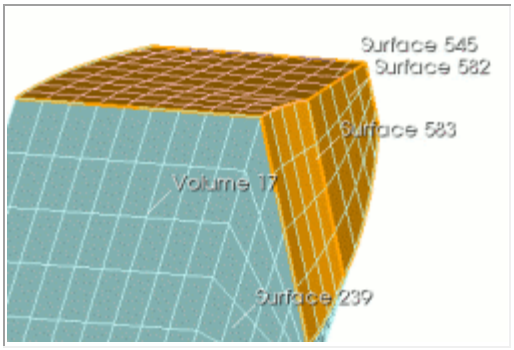
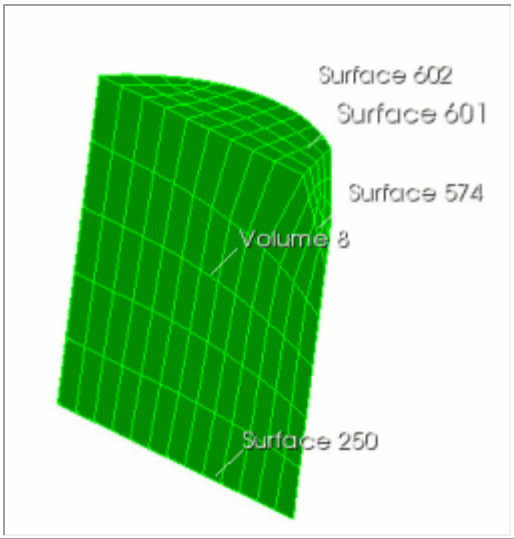

```
CUBIT> surf 229 size .25
CUBIT> mesh surf 229
CUBIT> volume 5 scheme sweep
source 229 target 230
CUBIT> mesh volume 5
```

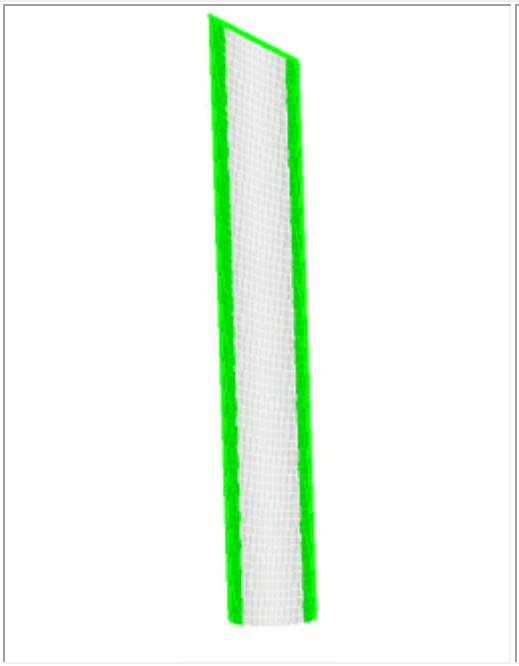
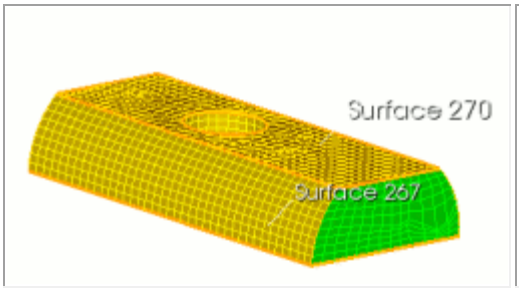
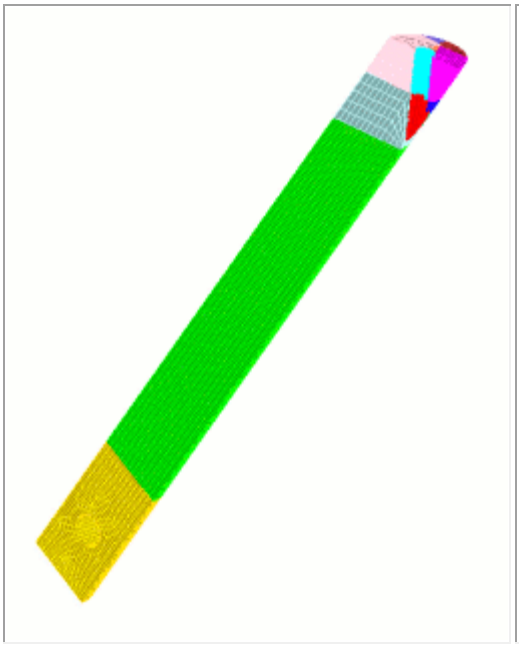
| | |
|---|---|
|  | **CUBIT> volume 4 scheme sweep source surface 526 target 528**<br>**CUBIT> mesh volume 4** |
|  | **CUBIT> volume 14 scheme sweep source 543 target 541**<br>**CUBIT> mesh volume 14**<br>**CUBIT> delete mesh**<br>**CUBIT> unmerge all** |
|  | **CUBIT> webcut body 6 with plane surface 524** |

| | |
|---|---|
|  | **CUBIT> unite body 16 17** |
|  | **CUBIT> webcut body 8 with plane surface 524** |

| | |
|---|---|
|  | **CUBIT> webcut body 18 with plane surface 540** |
|  | **CUBIT> webcut volume 9 with plane zplane offset -3 rotate 5 about x**<br>This is another effort to prevent skew in the final mesh<br>**CUBIT> imprint all**<br>**CUBIT> merge all** |
|  | **CUBIT> mesh volume 5 (swept around)**<br>**CUBIT> mesh volume 4 (mapped)**<br>**CUBIT> mesh volume 14 (swept top to bottom)**<br>**CUBIT> volume 15 scheme map**<br>**CUBIT> curve all in volume 15 size 0.5**<br>**CUBIT> mesh volume 15** |
|  | **CUBIT> volume 18 scheme tetprimitive**<br>**CUBIT> volume 18 interval 3**<br>**CUBIT> mesh volume 18** |

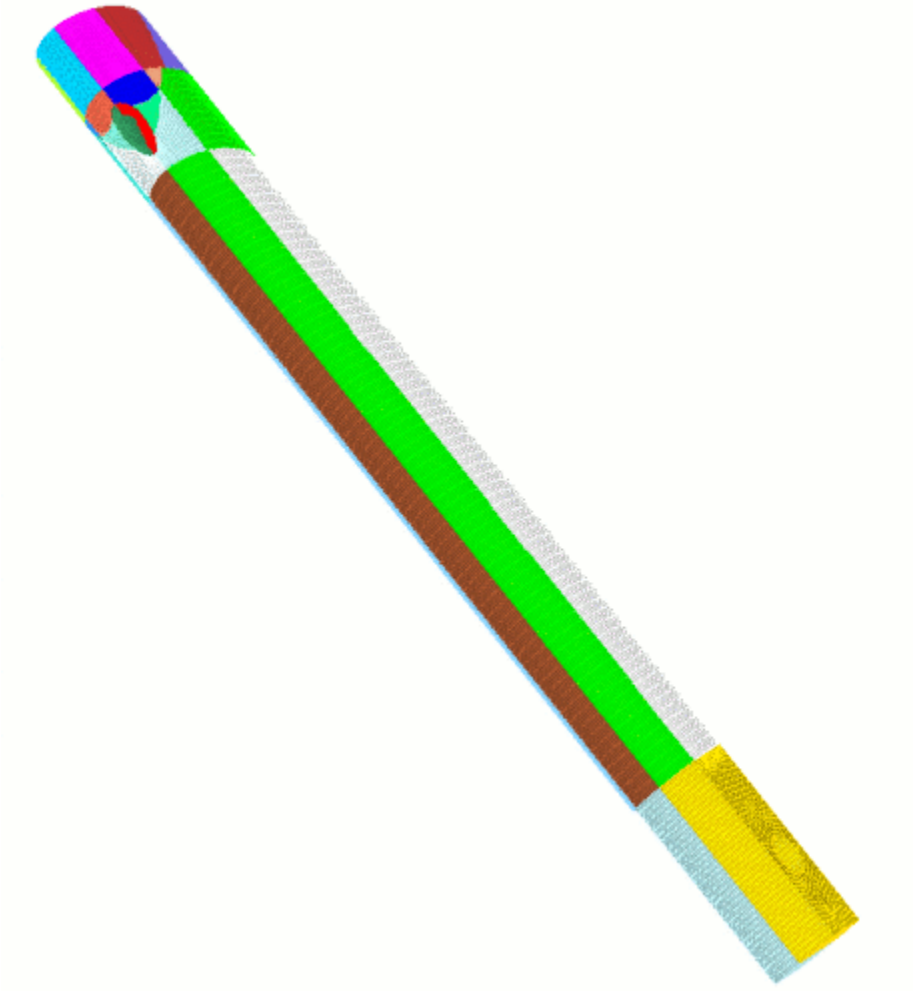
| | |
|---|---|
|  | **CUBIT> volume 9 scheme sweep source surface 579 601 target surface 592 rotate off**<br>**CUBIT> mesh volume 9**<br>**CUBIT> mesh volume 20** |
|  | **CUBIT> volume 6 scheme sweep source 569 target 570**<br>**CUBIT> mesh volume 6** |
|  | **CUBIT> volume 3 scheme sweep source 224 target 226**<br>**CUBIT> surf 224 226 scheme map**<br>**CUBIT> mesh volume 3** |

| | |
|---|---|
|  | **CUBIT> volume 19 scheme sweep source 543 target 586**<br>**CUBIT> mesh volume 19** |
|  | **CUBIT> volume 17 scheme sweep source 545 583 582 target 239**<br>**CUBIT> mesh volume 17** |
|  | **CUBIT> volume 8 scheme sweep source 574 597 601 target 241**<br>**CUBIT> mesh volume 8** |

| | |
|---|---|
|  | `CUBIT> volume 7 1 size 2`<br>`CUBIT> volume 7 1 scheme auto` |
|  | `CUBIT> volume 10 scheme sweep source 270 target 267`<br>`CUBIT> mesh volume 7 1`<br>`CUBIT> mesh volume 10` |
|  | `CUBIT> unmerge all`<br>`CUBIT> body all copy reflect x`<br>`CUBIT> merge all` |

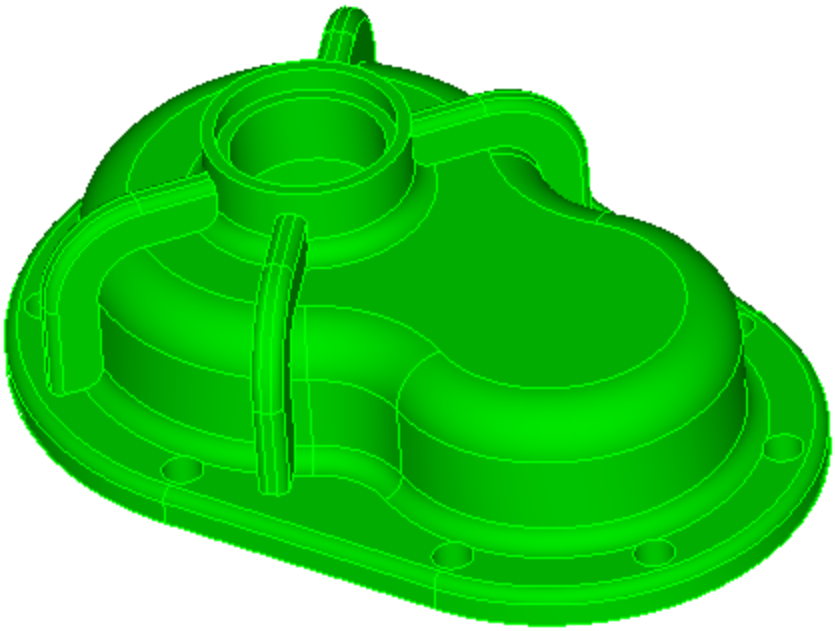
*Final mesh*

The entire mesh is copied and reflected around the x axis during the last step. The advantage of symmetry in this example is that it cuts the decomposition in half, and it also ensures a perfectly symmetrical mesh.

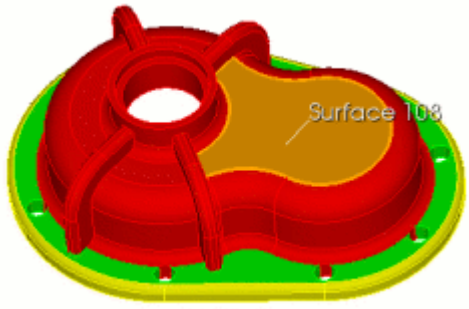


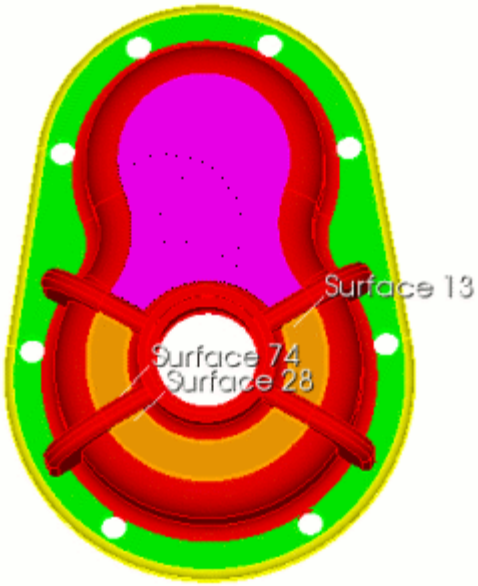
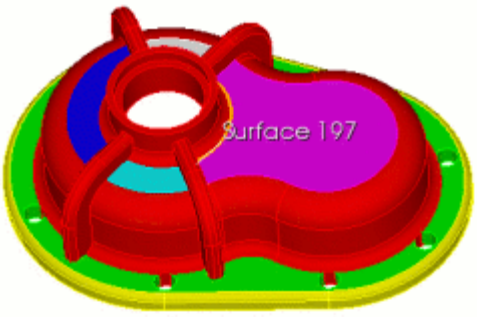
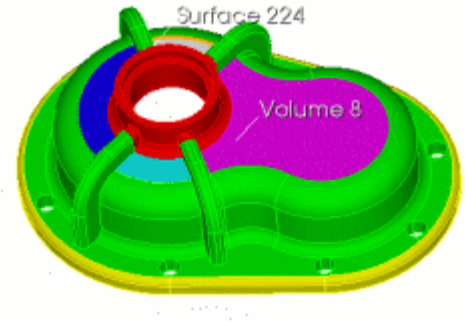
## Example 7. Using virtual geometry in geometry decomposition

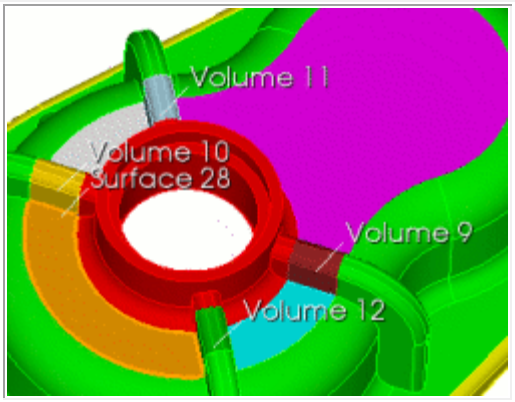
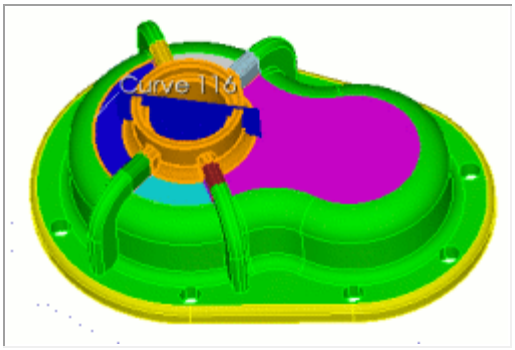
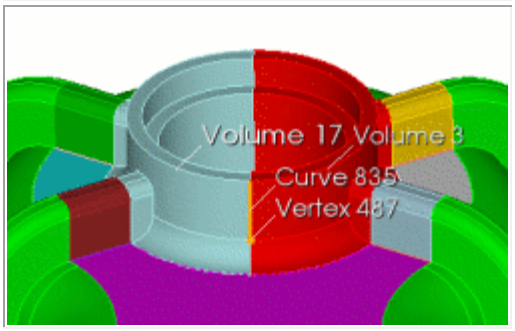
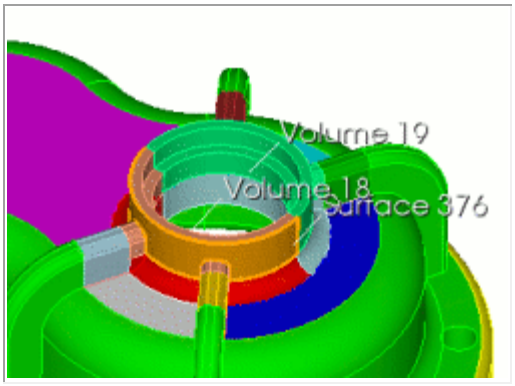
Virtual geometry is used to change the properties of mesh without changing the underlying geometry. The next example uses virtual geometry to remove unwanted sliver curves, and to create a sweepable volume. The composite curve function is used to combine sliver curves that are created from webcutting a slightly curved surface. Then the partition surface command is used to create additional partitions on a surface to ensure sweepability.

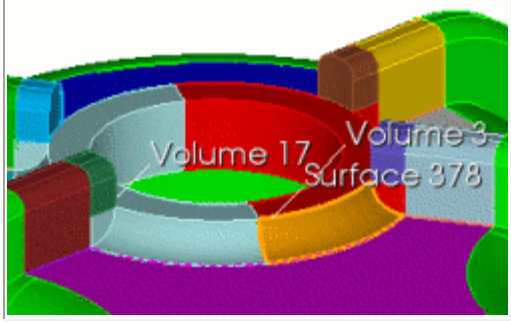


*Suggested webcuts*

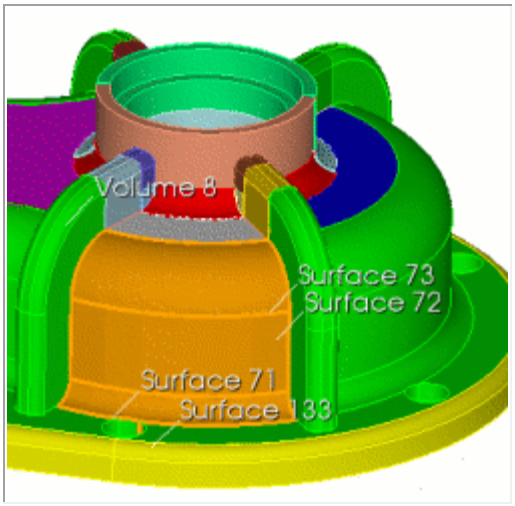
| Webcut | Command |
| --- | --- |
|  | **CUBIT> webcut volume 1 sweep surface 2 vector 0 0 -1 through_all** |
|  | **CUBIT> webcut volume 3 sweep surface 108 vector 0 0 -1 through_all** |

| | |
|---|---|
|  | **CUBIT> webcut volume 3 sweep surface 13 vector 0 0 -1 through_all**<br>**CUBIT> webcut volume 3 sweep surface 28 vector 0 0 -1 through_all**<br>**CUBIT> webcut volume 3 sweep surface 74 vector 0 0 -1 through_all** |
|  | **CUBIT> webcut volume 3 with sheet extended from surface 197** |
|  | **CUBIT> webcut volume 8 with sheet extended from surface 224** |

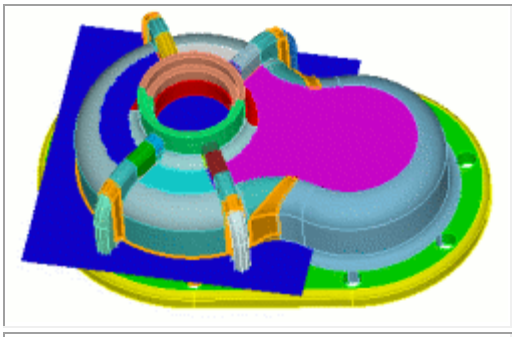
| | |
|---|---|
|  | **CUBIT> webcut volume 11 10 12 9 with plane surface 28** |
|  | **CUBIT> webcut volume 3 with plane normal to curve 116 fraction 0.5** |
|  | **CUBIT> webcut volume 3 17 with plane normal to curve 835 close_to vertex 487** |
|  | **CUBIT> webcut volume 18 19 with sheet extended from surface 376** |



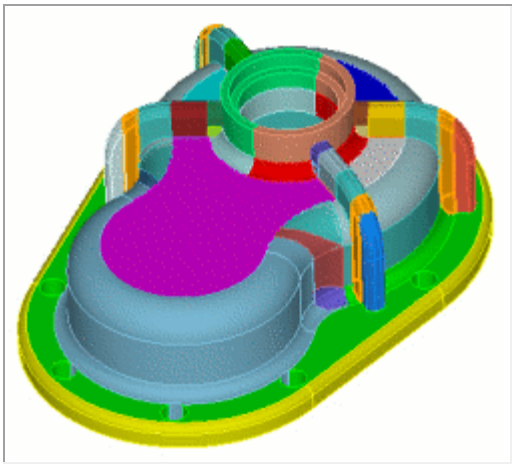

```
CUBIT> webcut volume 3 17 with sheet extended
from surface 378
```



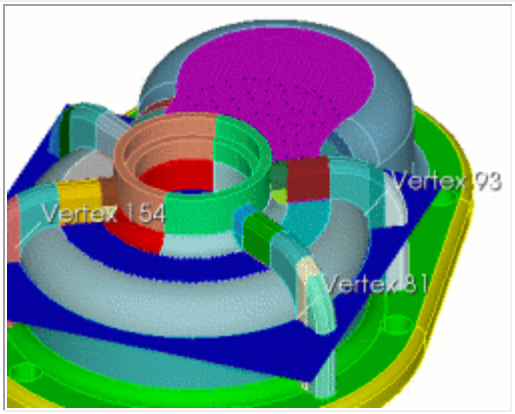

```
CUBIT> webcut volume 8 with sheet extended
from surface 73
CUBIT> webcut volume 8 with sheet extended
from surface 72
CUBIT> webcut volume 8 with sheet extended
from surface 133
CUBIT> webcut volume 8 with sheet extended
from surface 71
```



```
CUBIT> webcut volume 8 with plane vertex 709
vertex 713 vertex 702
```

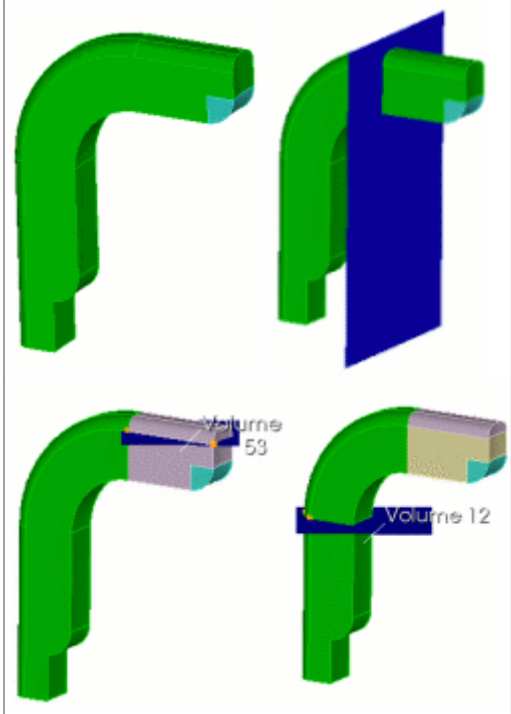


```
CUBIT> unite volume 36 45
CUBIT> unite volume 37 43
CUBIT> unite volume 35 44
CUBIT> unite volume 39 42
```

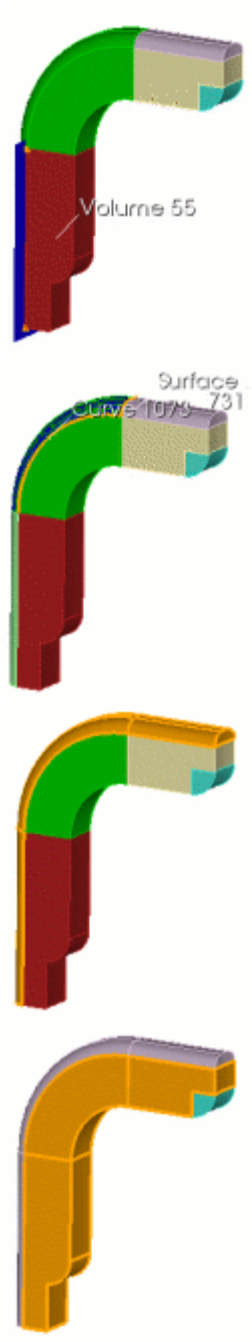



```
CUBIT> webcut volume 29 with plane vertex 81
vertex 93 vertex 154
```

```
CUBIT> unite volume 33 36 50 11
CUBIT> unite volume 10 49 37 31
CUBIT> unite volume 12 52 35 34
CUBIT> unite volume 9 51 39 32
CUBIT> unite volume 9 22
CUBIT> unite volume 12 23
CUBIT> unite volume 20 33
CUBIT> unite volume 21 10
```

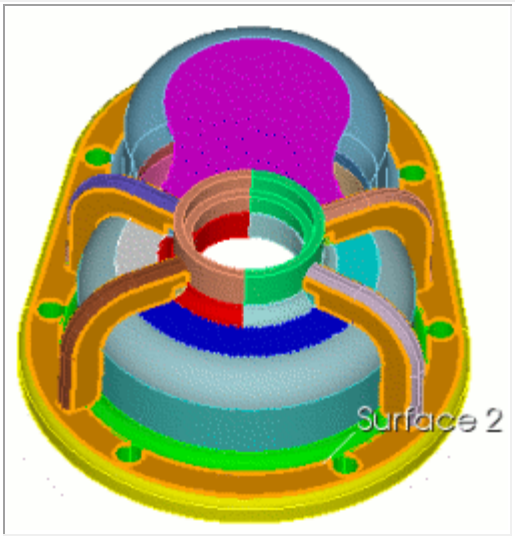
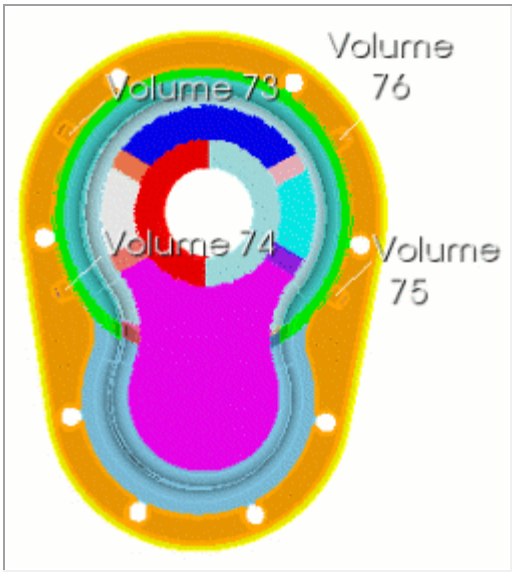
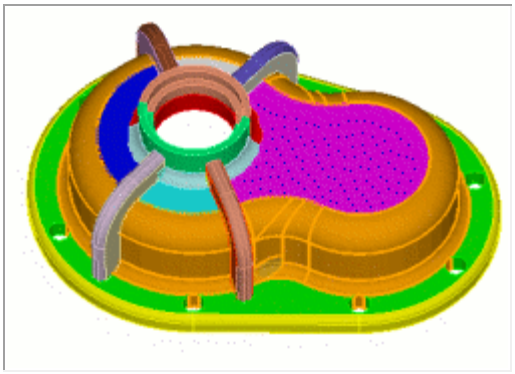


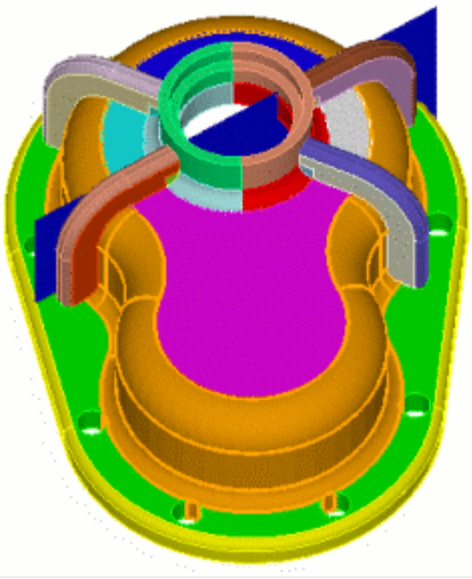
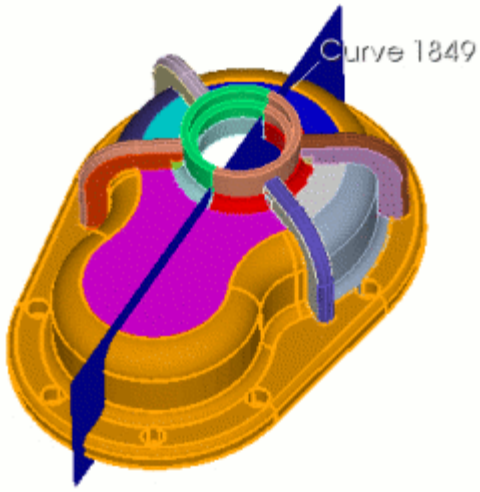
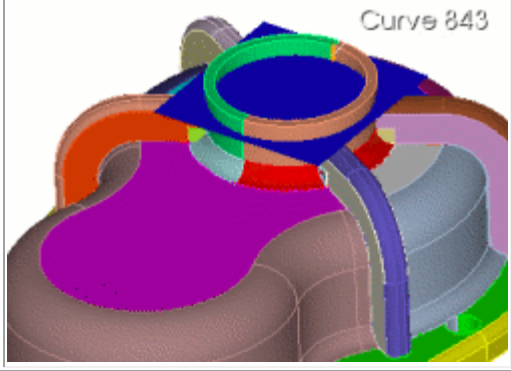

```
CUBIT> webcut volume 12 with plane vertex 86
vertex          71          vertex          76
CUBIT> webcut volume 53 with plane vertex 738
vertex          87          vertex          741
CUBIT> webcut volume 12 with plane vertex 72
vertex          85          vertex          74
CUBIT> webcut volume 55 with plane vertex 754
vertex          205         vertex          208
CUBIT>  webcut  volume  12  sweep  surface  731
along       curve       1073        through_all
CUBIT>     unite     volume     53     57     56
CUBIT> unite volume 54 12 55
CUBIT>  webcut volume 9 with plane vertex 99
vertex          101         vertex          103
CUBIT> webcut volume 58 with plane vertex 769
vertex          98          vertex          772
CUBIT> webcut volume 9 with plane vertex 106
vertex          104         vertex          100
CUBIT> webcut volume 60 with plane vertex 781
vertex          201         vertex          198
CUBIT> webcut volume 9 sweep surface 764 along
curve             1078             through_all
CUBIT>     unite     volume     58     62     60
CUBIT> unite volume 59 9 61
CUBIT> webcut volume 20 with plane vertex 140
vertex          138         vertex          135
CUBIT> webcut volume 63 with plane vertex 139
vertex          137         vertex          134
CUBIT> webcut volume 20 with plane vertex 141
vertex          800         vertex          796
CUBIT> webcut volume 64 with plane vertex 803
vertex          220         vertex          223
CUBIT>  webcut  volume  63  sweep  surface  803
along       curve       1238        through_all
```

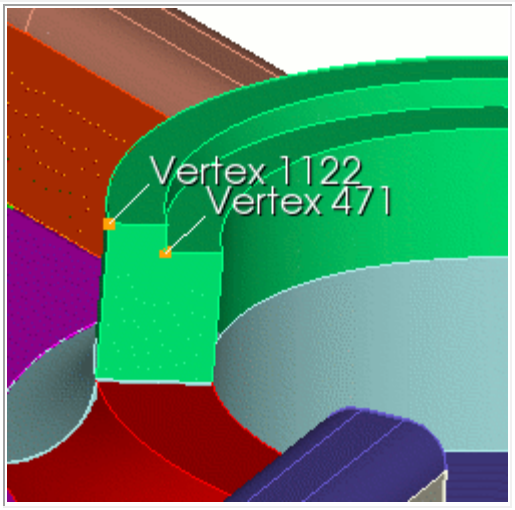
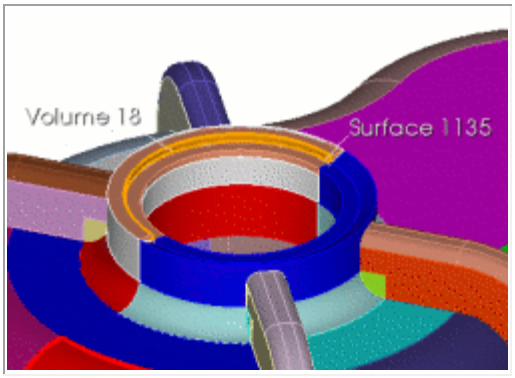
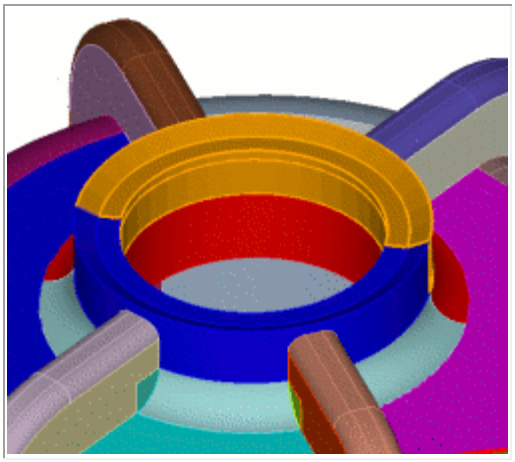
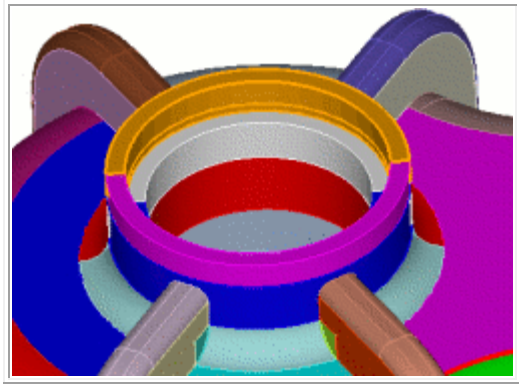


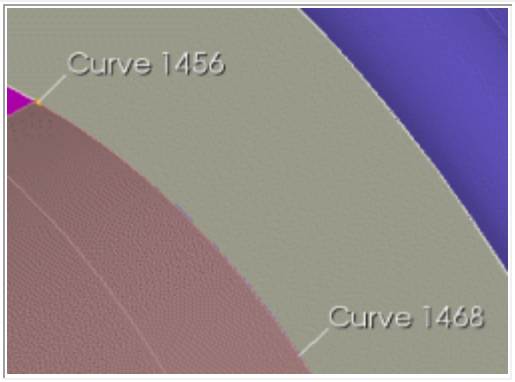

```
CUBIT>    unite    volume    20    67    66
CUBIT> unite volume 65 63 64
CUBIT> webcut volume 21 with plane vertex 165
vertex         163         vertex         160
CUBIT> webcut volume 68 with plane vertex 164
vertex         162         vertex         159
CUBIT> webcut volume 21 with plane vertex 825
vertex         169         vertex         822
CUBIT> webcut volume 69 with plane vertex 830
vertex         216         vertex         213
CUBIT>  webcut  volume  68  sweep  surface  836
along      curve      1069       through_all
CUBIT>    unite    volume    21    72    69
CUBIT> unite volume 70 68 71
```

These are the steps to webcut each of the stiffeners into the configuration shown. It is repeated for each of the stiffeners. This is also the step which creates the sliver curves which must be composited out later.

| | |
|---|---|
|  | **CUBIT> webcut volume 70 65 59 54 with plane surface 2** |
|  | **CUBIT> unite volume 1 76 75 73 74** |
|  | **CUBIT> unite volume 28 47 46 41 48 38 8 30 29 40** |

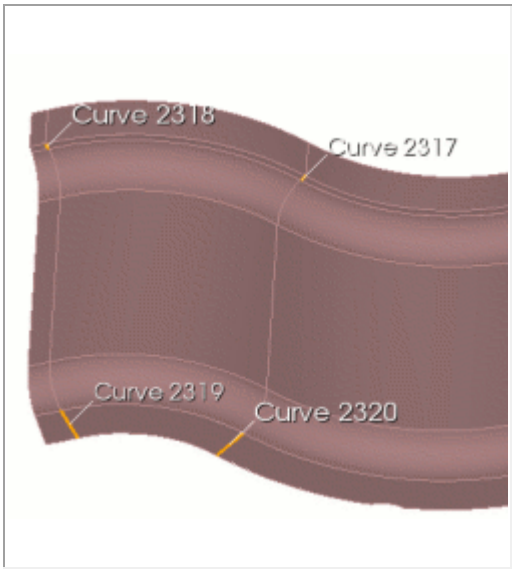
| | |
|---|---|
|  | **CUBIT> webcut volume 28 with plane surface 870**<br>**CUBIT> webcut volume 28 77 with plane surface 871**<br>**CUBIT> webcut volume 28 77 with plane surface 878**<br>**CUBIT> webcut volume 28 77 with plane surface 879** |
|  | **CUBIT> webcut volume 1 81 2 82 with plane normal to curve 1849 fraction 0.5** |
|  | **CUBIT>webcut volume 19 18 with plane normal to curve 843 fraction 0.75** |

| | |
|---|---|
|  | **CUBIT> create curve vertex 1122 vertex 471 on surface 1134** |
|  | **CUBIT> webcut volume 19 sweep curve 2073 along curve 2042 through_all**<br>**CUBIT> webcut volume 18 with sheet extended from surface 1146**<br>**CUBIT> webcut volume 18 with sheet extended from surface 1135** |
|  | **CUBIT> unite volume 91 92**<br>**CUBIT> delete curve 2073** |
|  | **CUBIT> unite volume 89 18**<br>**CUBIT> unite volume 88 19**<br>**CUBIT> imprint all**<br>**CUBIT> merge all** |



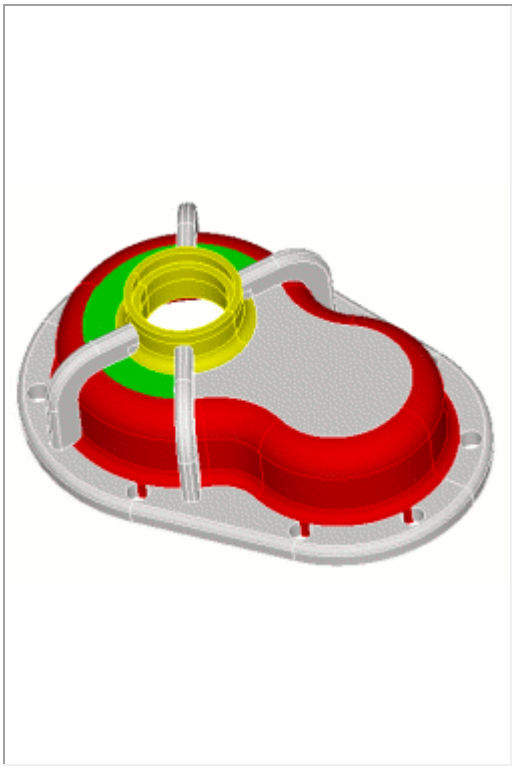

Composite small curves formed from webcuts

```
CUBIT> composite create curve 1456 1468
CUBIT> composite create curve 1459 1467
CUBIT> composite create curve 1499 1511
CUBIT> composite create curve 1502 1510
CUBIT> composite create curve 1371 1379
CUBIT> composite create curve 1370 1381
CUBIT> composite create curve 1423 1413
CUBIT> composite create curve 1422 1414
CUBIT> volume all scheme auto
```




Create the partitioned curves shown using existing vertices

```
CUBIT> partition create surface 1067 vertex 311 175
CUBIT> partition create surface 1067 vertex 174 312
CUBIT> partition create surface 1063 vertex 123 294
CUBIT> partition create surface 1251 vertex 170 226
CUBIT> partition create surface 1082 vertex 195 115
CUBIT> partition create surface 1082 vertex 242 116
CUBIT> partition create surface 1077 vertex 117 309
CUBIT> partition create surface 1255 vertex 118 310
```



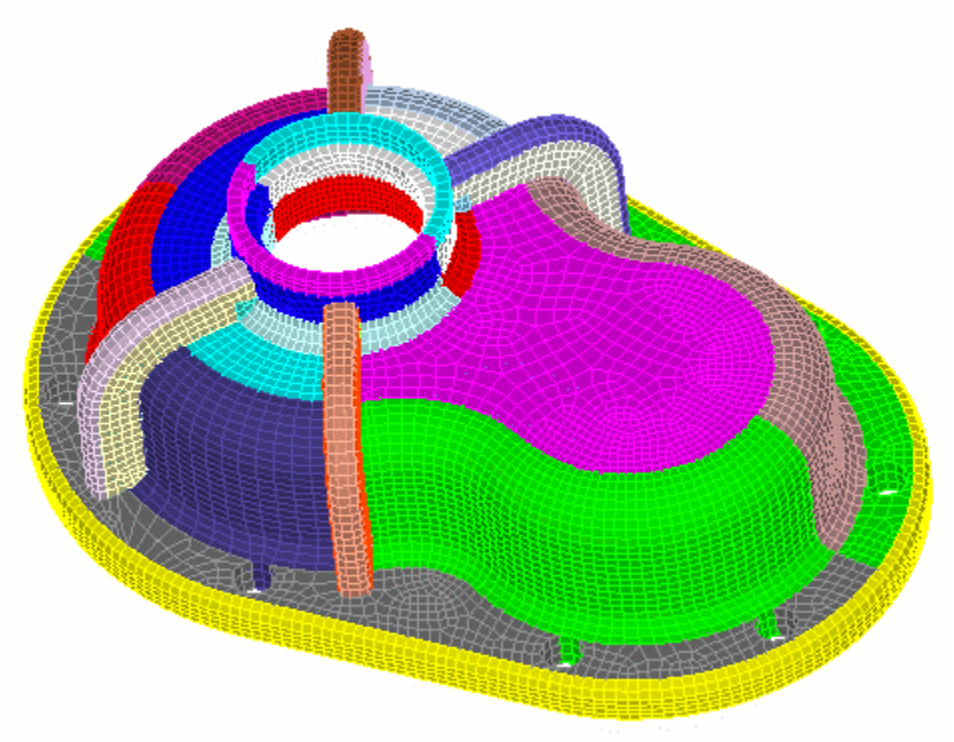
Meshing order is significant in this case. Since meshing a volume will hard set the interval counts on curves and surfaces, you will need to make sure that all of the interval counts are the same on adjacent volumes. Usually the meshing algorithm can handle this interval matching, but sometimes it helps to mesh volumes in a certain order. In this case, the meshing order also significantly changes the quality in the resulting mesh.

```
CUBIT> reset volume all
CUBIT> volume all scheme auto
CUBIT> volume 81 scheme sweep source surface 979 target surface 1061 rotate off
CUBIT> volume 81 sweep smooth auto
CUBIT> volume 85 scheme sweep source surface 1061 target surface 889 rotate off
CUBIT> volume 85 sweep smooth auto
CUBIT> volume all size 0.1
CUBIT> curve 2125 2122 interval 12
CUBIT> mesh vol 5 6 7 13 14 15 16 (COLORED GREEN)
CUBIT> mesh Volume 85 81 77 83 78 82 87 28 80 79 (COLORED RED)
```

| | |
|---|---|
| | CUBIT> mesh vol 88 89 91 90 17 3 (COLORED YELLOW)<br>CUBIT> mesh volume with not is_meshed (COLORED WHITE) |

*Final mesh*

The final mesh is shown below.

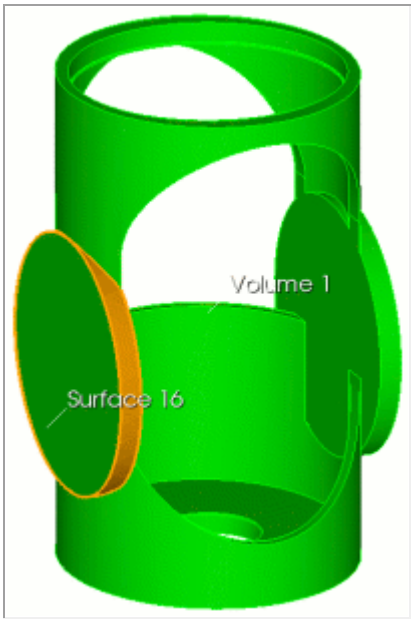


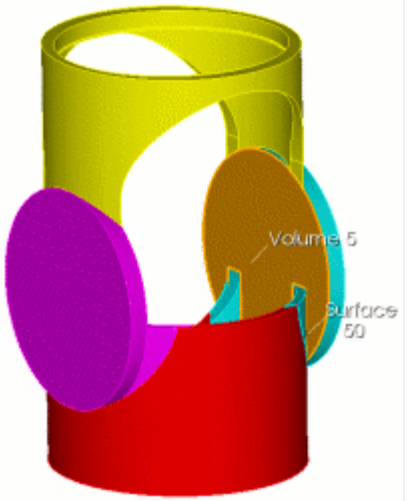
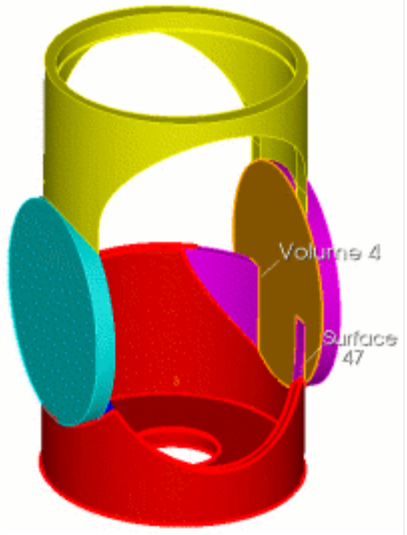
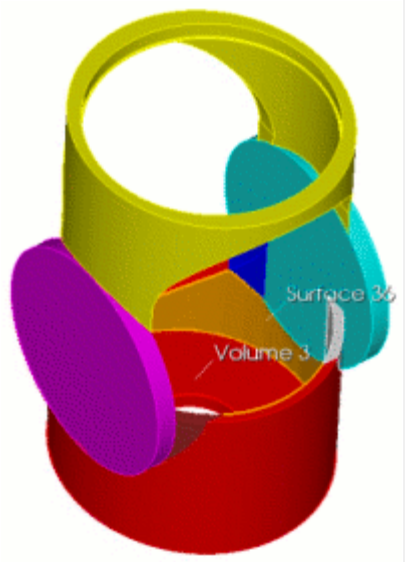
## Example 8. Sweeping volumes with narrow angles and surfaces

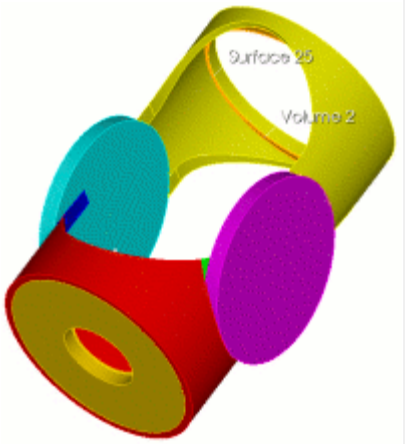
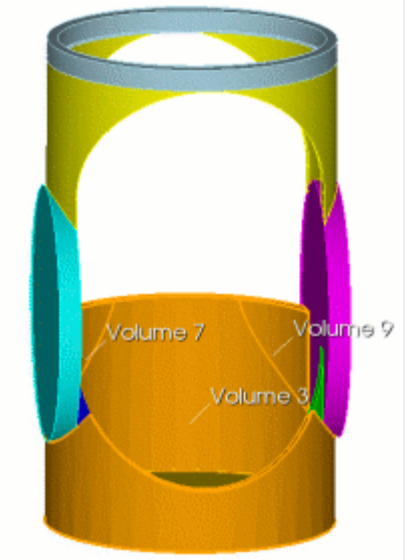
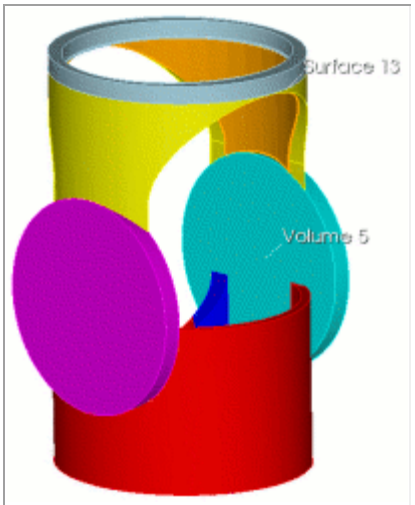
Narrow angles are a challenge for sweeping algorithms. In the next example, a well-placed webcut shaves off the end of the small angle to create an additional surface for the sweeping algorithm.

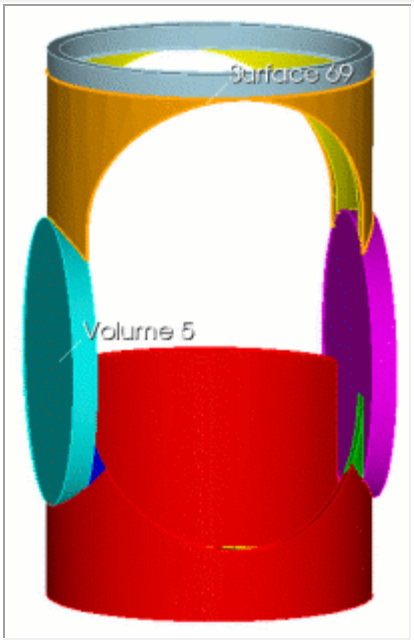
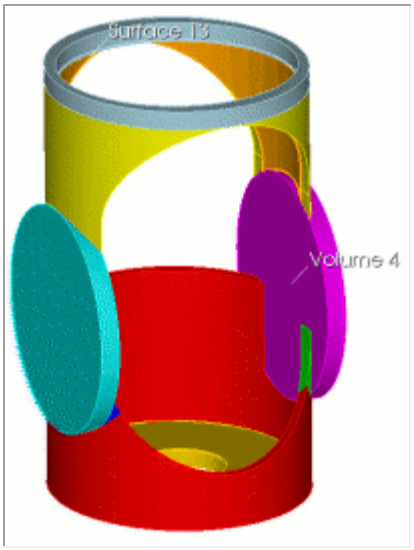


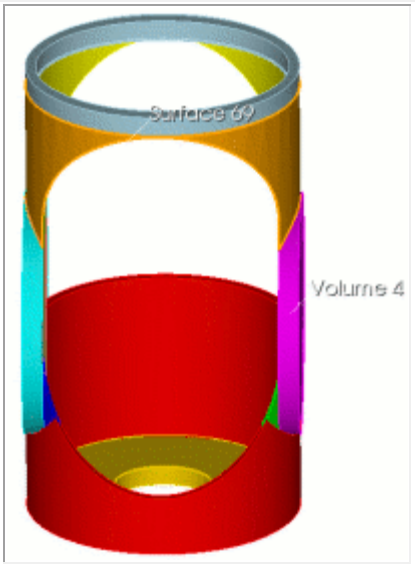
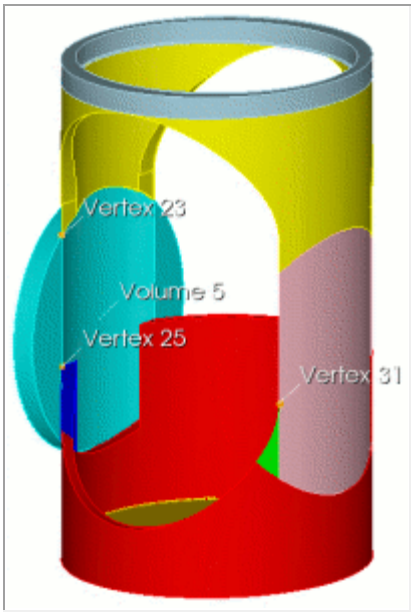
*Suggested webcuts*

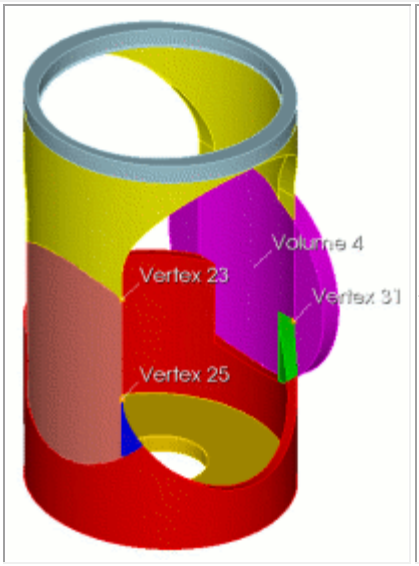
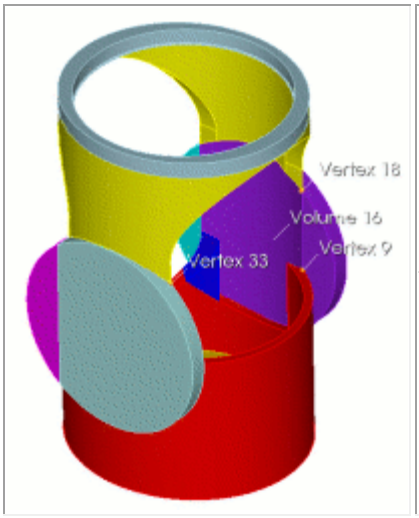
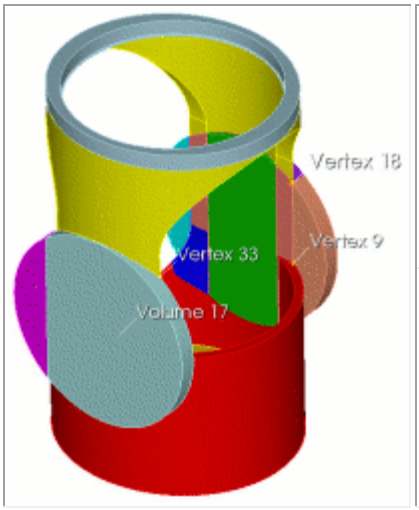
| Webcut | Command |
| --- | --- |
|  | **CUBIT> webcut volume 1 with sheet extended from surface 16** |

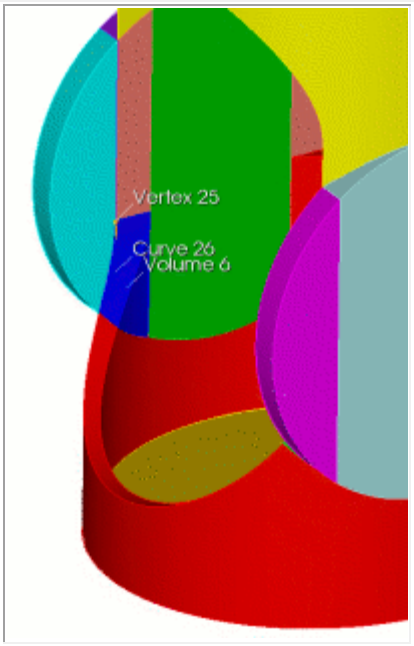
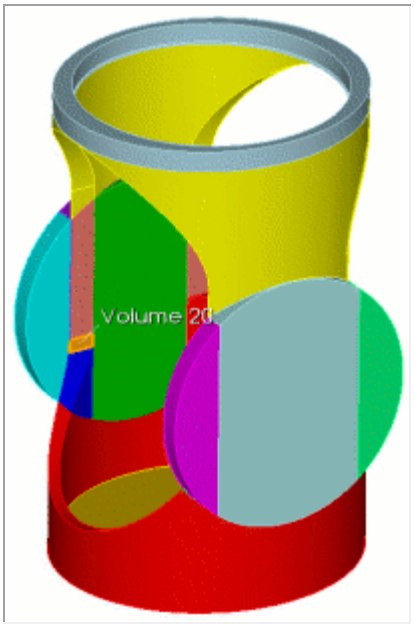
| | |
|---|---|
|  | **CUBIT> webcut volume 5 with plane surface 50** |
|  | **CUBIT> webcut volume 4 with plane surface 47** |
|  | **CUBIT> webcut volume 3 with sheet extended from surface 36** |

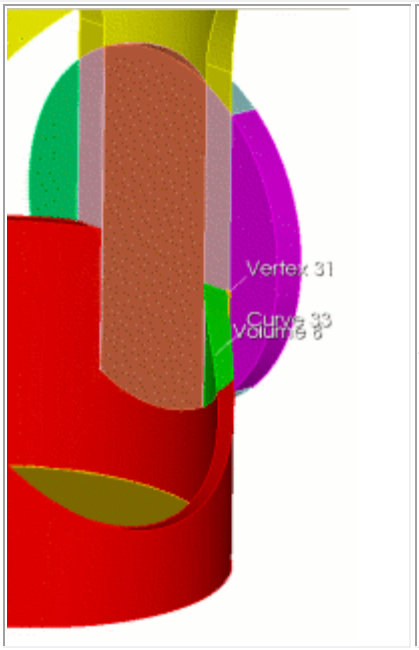
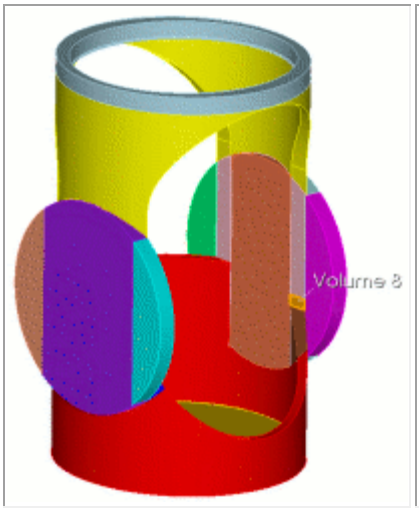
| | |
|---|---|
|  | **CUBIT> webcut volume 2 with plane surface 25** |
|  | **CUBIT> unite volume 3 9 7** |
|  | **CUBIT> webcut volume 5 with sheet extended from surface 13** |

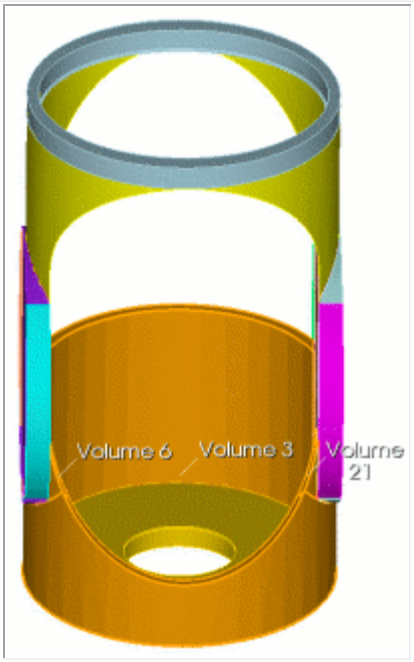
| | |
|---|---|
|  | **CUBIT> webcut volume 5 with sheet extended from surface 69** |
|  | **CUBIT> webcut volume 4 with sheet extended from surface 13** |

| | |
|---|---|
|  | **CUBIT> webcut volume 4 with sheet extended from surface 69** |
|  | **CUBIT> webcut volume 5 with plane vertex 23 vertex 25 vertex 31** |

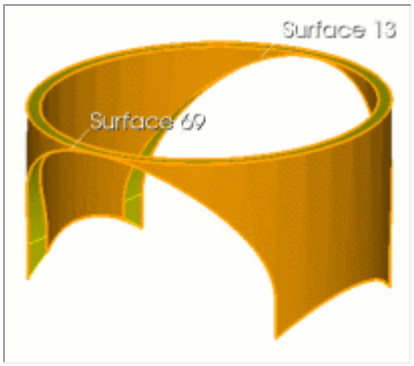
| | |
|---|---|
|  | **CUBIT> webcut volume 4 with plane vertex 23 vertex 25 vertex 31** |
|  | **CUBIT> webcut volume 16 with plane vertex 18 vertex 9 vertex 33** |
|  | **CUBIT> webcut volume 17 with plane vertex 18 vertex 9 vertex 33** |

| | |
|---|---|
|  | **CUBIT> webcut volume 6 with plane normal to curve 26 distance 0.6 from vertex 25** |
|  | **CUBIT> delete volume 20** |

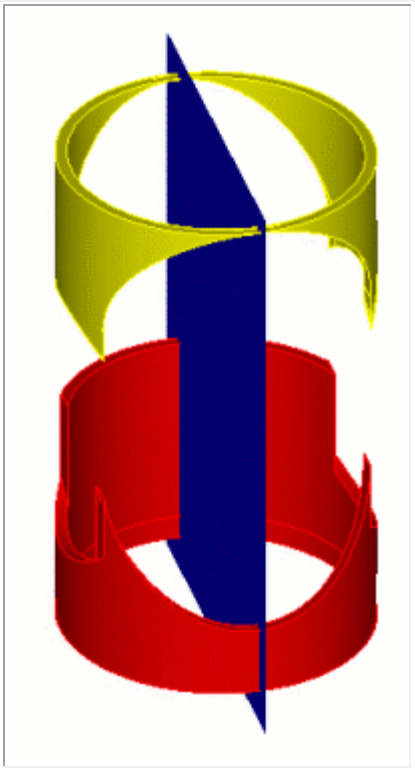
| | |
|---|---|
|  | **CUBIT> webcut volume 8 with plane normal to curve 33 distance 0.6 from vertex 31** |
|  | **CUBIT> delete volume 8** |

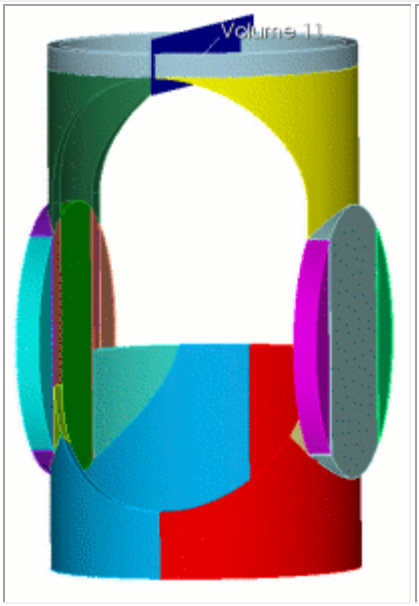



```
CUBIT> unite volume 3 21 6
CUBIT>     imprint     volume     all
CUBIT>      merge      volume     all
CUBIT>    volume    all    size    0.3
CUBIT> volume all scheme auto
```

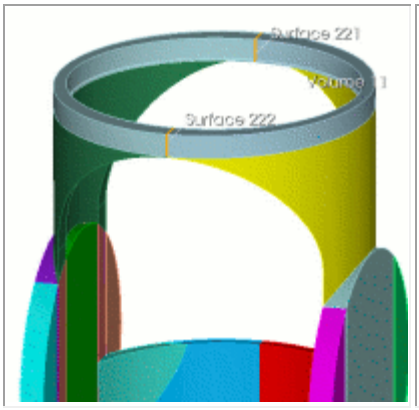



```
CUBIT> volume 2 scheme sweep source
13 target 69
CUBIT> volume 2 sweep smooth auto
CUBIT> unmerge volume all
```

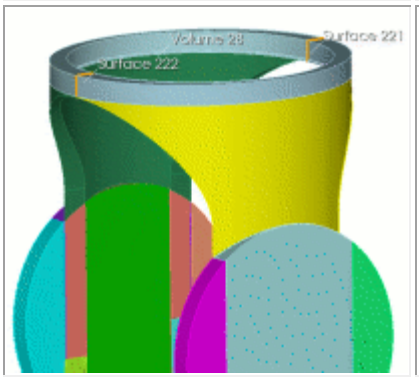
| | |
|---|---|
|  | **CUBIT> webcut volume 2 3 with plane zplane** |
|  | **CUBIT> webcut volume 3 with sheet extended from surface 154** |
|  | **CUBIT> webcut volume 23 with sheet extended from surface 153** |



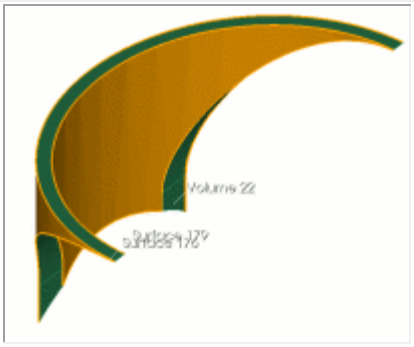
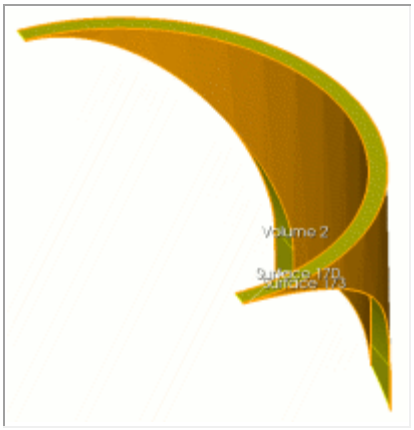
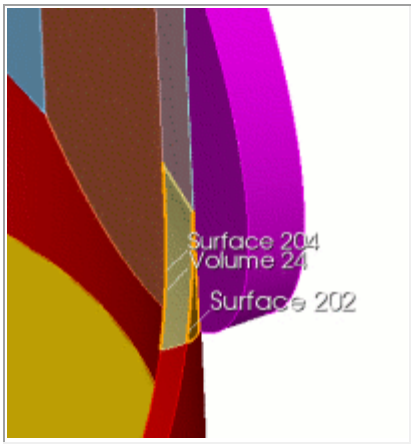

```
CUBIT> webcut volume 11 with plane
zplane noimprint nomerge
CUBIT> imprint volume all
CUBIT> merge volume all
```

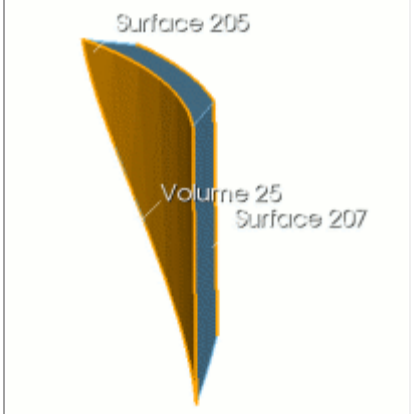



```
CUBIT> volume 11 scheme sweep source
surface 221 target surface 222
rotate off
CUBIT> volume 11 sweep smooth auto
```

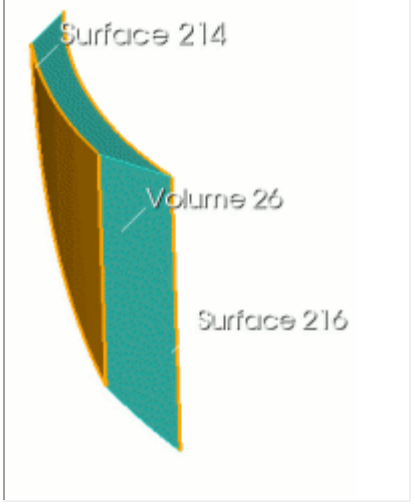



```
CUBIT> volume 28 scheme sweep source
surface 222 target surface 221
rotate off
CUBIT> volume 28 sweep smooth auto
```

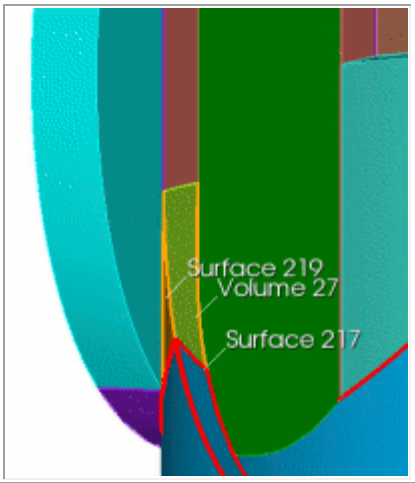
| | |
|---|---|
|  | **CUBIT> volume 22 scheme sweep source surface 176 target surface 179 rotate off**<br>**CUBIT> volume 22 sweep smooth auto** |
|  | **CUBIT> volume 2 scheme sweep source surface 173 target surface 170 rotate off**<br>**CUBIT> volume 2 sweep smooth auto** |
|  | **CUBIT> volume 24 scheme sweep source surface 204 target surface 202 rotate off**<br>**CUBIT> volume 24 sweep smooth auto** |



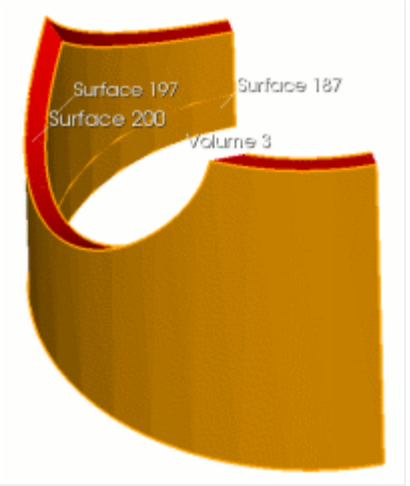
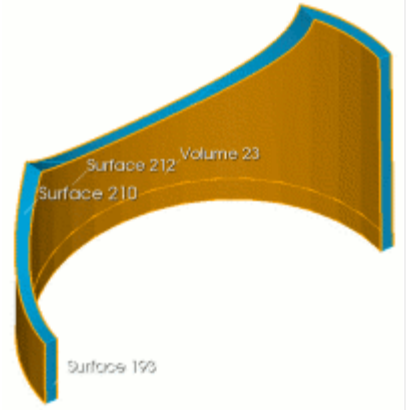

```
CUBIT> volume 25 scheme sweep source
surface  205  target  surface  207
rotate off
CUBIT> volume 25 sweep smooth auto
```




```
CUBIT> volume 26 scheme sweep source
surface  214  target  surface  216
rotate off
CUBIT> volume 26 sweep smooth auto
```

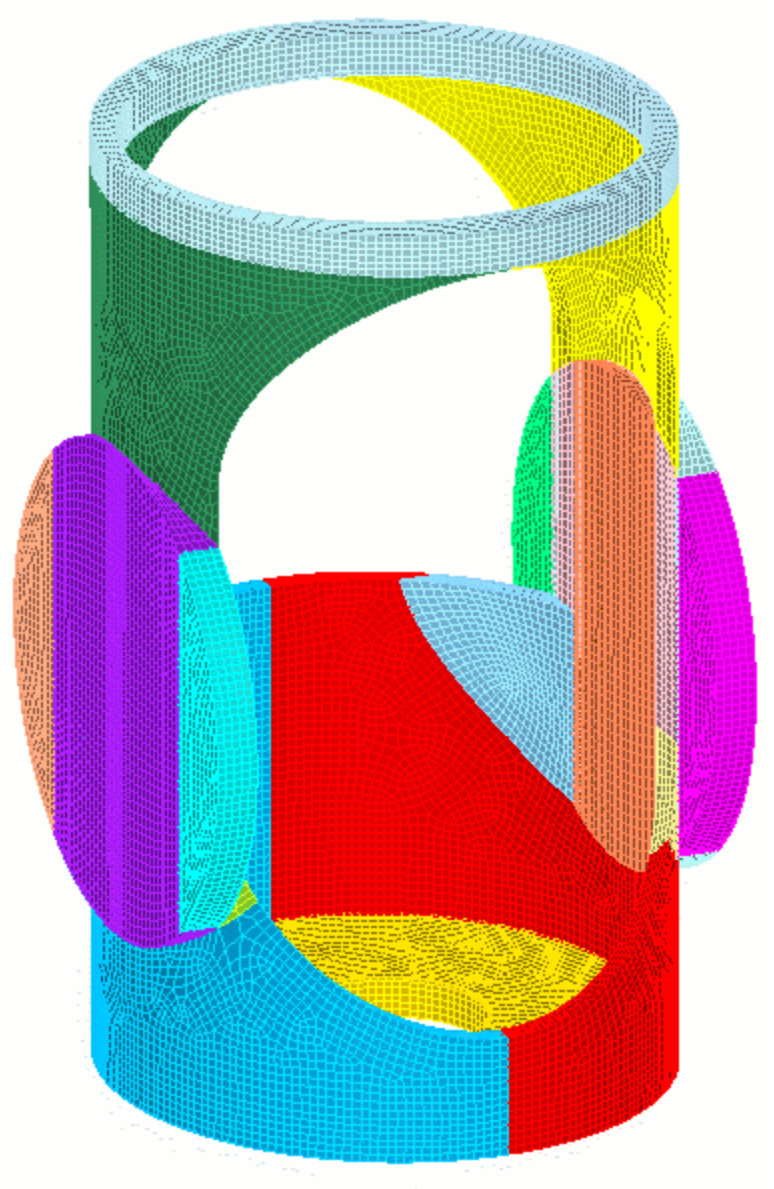



```
CUBIT> volume 27 scheme sweep source
surface  217  target  surface  219
rotate off
CUBIT> volume 27 sweep smooth auto
```

| | |
|---|---|
|  | **CUBIT> volume 3 scheme sweep source surface 197 187 target surface 200 rotate off**<br>**CUBIT> volume 3 sweep smooth auto** |
|  | **CUBIT> volume 23 scheme sweep source surface 212 193 target surface 210 rotate off**<br>**CUBIT> volume 23 sweep smooth auto**<br>**CUBIT> volume all scheme auto**<br>**CUBIT> volume all size 0.2**<br>**CUBIT> mesh volume all** |

*Final mesh*

The final mesh is shown below.



gui
GUI Basic Tutorial

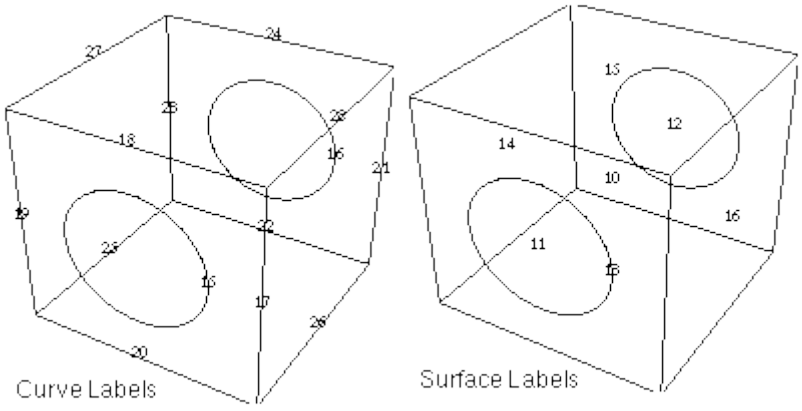
# *Overview*

This tutorial demonstrates the use of CUBIT to create and mesh a brick with a through-hole. The primary steps in performing this task are:

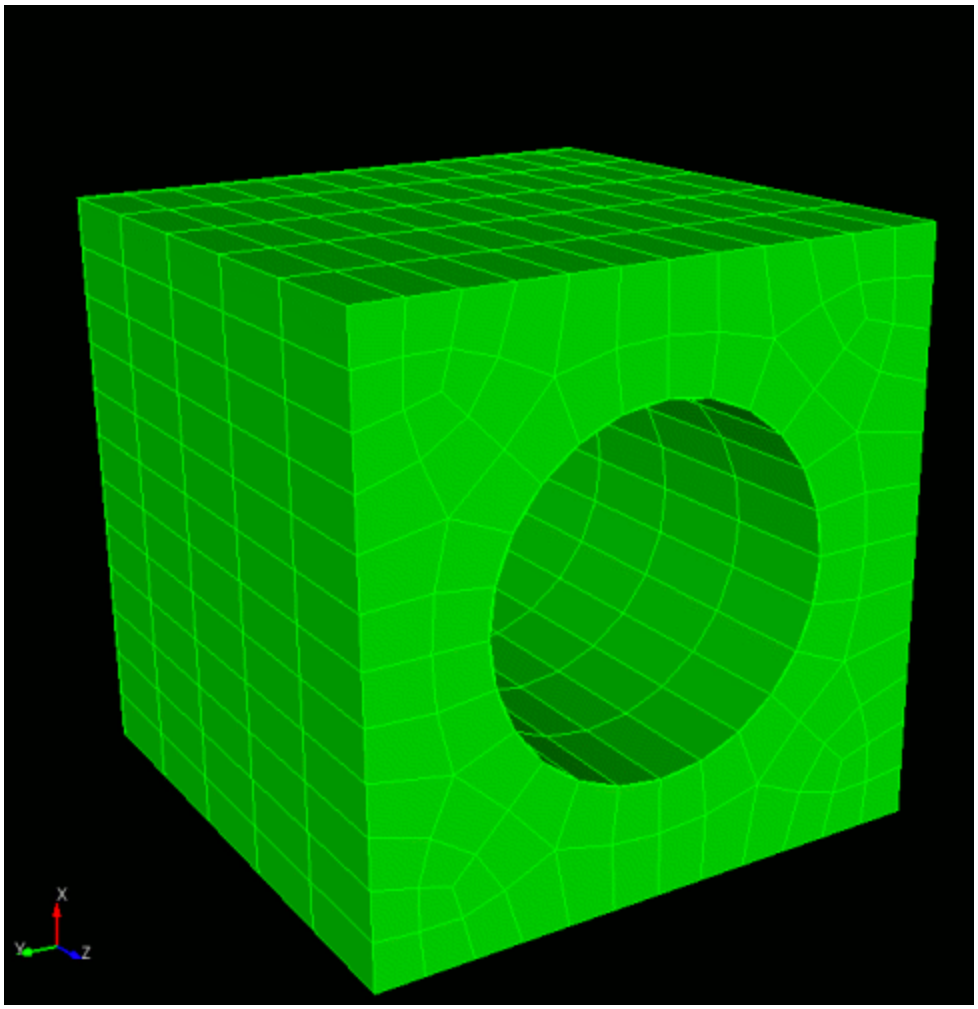
- Creating the geometry
- Setting the interval sizes and meshing schemes
- Meshing the geometry

- Specifying the boundary conditions
- Exporting the mesh

The geometry for this tutorial is a brick with a cylindrical hole in the center, shown in the figure below. This figure also shows the curve and surface identification (ID) numbers, which are referenced in the command lines options shown with each step. The final meshed body is shown in the next figure.




**Geometry for Brick with Cylindrical Hole**

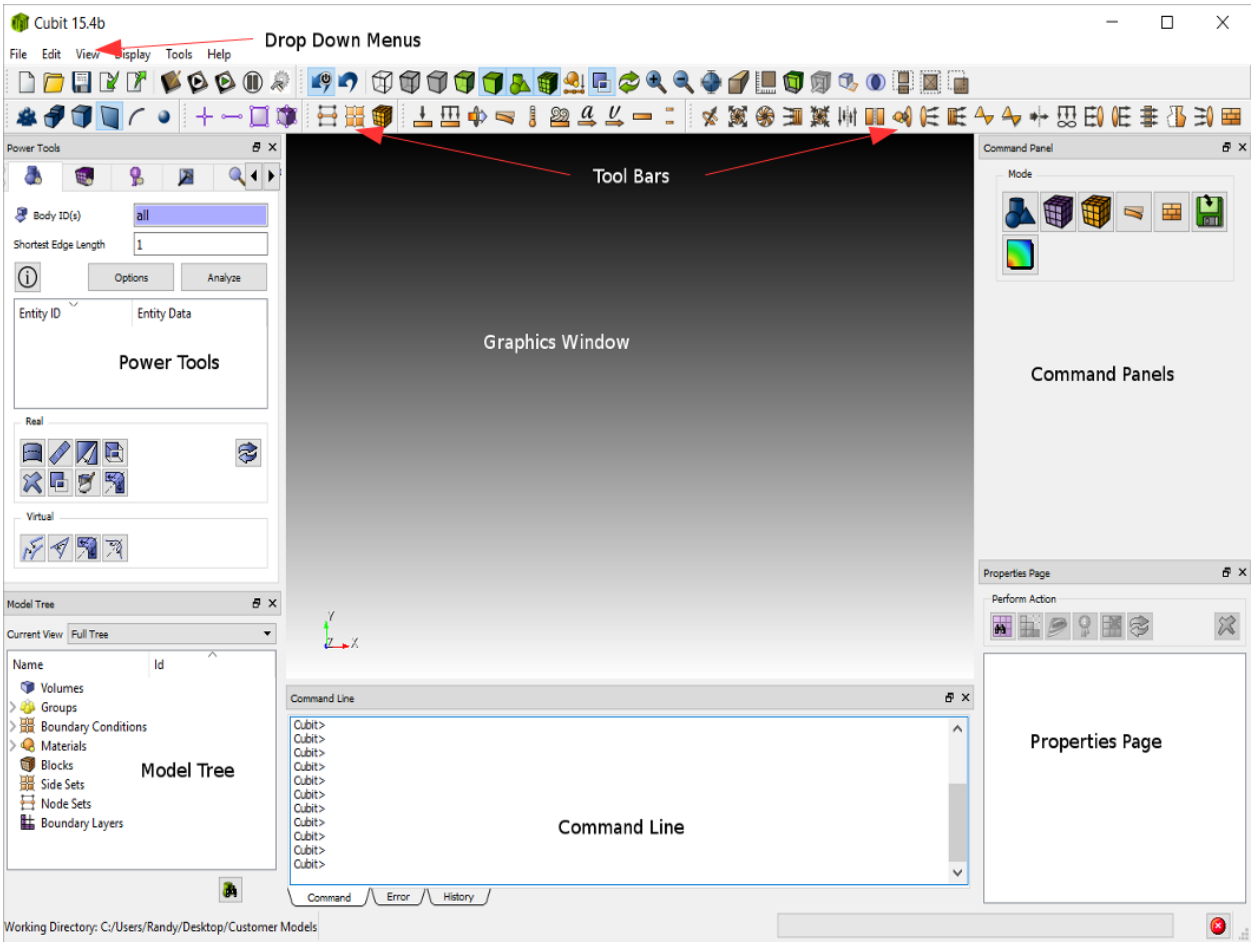



**Generated Mesh for Brick with Cylindrical Hole**

GUI Basic Tutorial Step 1

# *Step 1: Beginning Execution*

Type "**cubit**" from a UNIX prompt or select cubit from the start menu if you are running on a PC with Windows. The CUBIT Application Window will appear as illustrated below:




**CUBIT Application Window**

The use of each window in the CUBIT program is described briefly below

| | |
|---|---|
| **Graphics Window** | The current model will be displayed here. Zooming, panning, and rotating are also performed in this window. |
| **Drop Down Menus** | Functions such as file management, edit controls, display options, user preferences, journal file management, window manipulation, and help are available in the pull-down menus. |
| **Toolbars** | This is a large selection of selectable icons that duplicate the functions found in the pull-down menus. Additionally, picking types, and mouse selection controls are found here. |
| **Power Tools** | The Power Tools contains the ITEM workflow, geometry repair power tools, meshing power tools, mesh/geometry comparison tool, defeaturing tool, assembly tool, and mesh quality power tools. |

| | |
|---|---|
| **Command Line Workspace** | The command line workspace contains both the cubit command, error, history, and script windows. The command window is used to enter cubit commands and view the output. The error window is used to view cubit errors. The history window is used to view recent commands. The optional script window is used for Python programming. |
| **Command Panels** | Most Cubit commands are available in the command panels. The panel is organized topologically, by mode. |
| **Properties Page** | This is a list of properties of the selected geometry, mesh, boundary condition, or assembly. Some of the properties can also be edited from this window. |
| **Model Tree** | The model tree shows all geometric entities and their relationships, boundary conditions, boundary layers, and groups. |

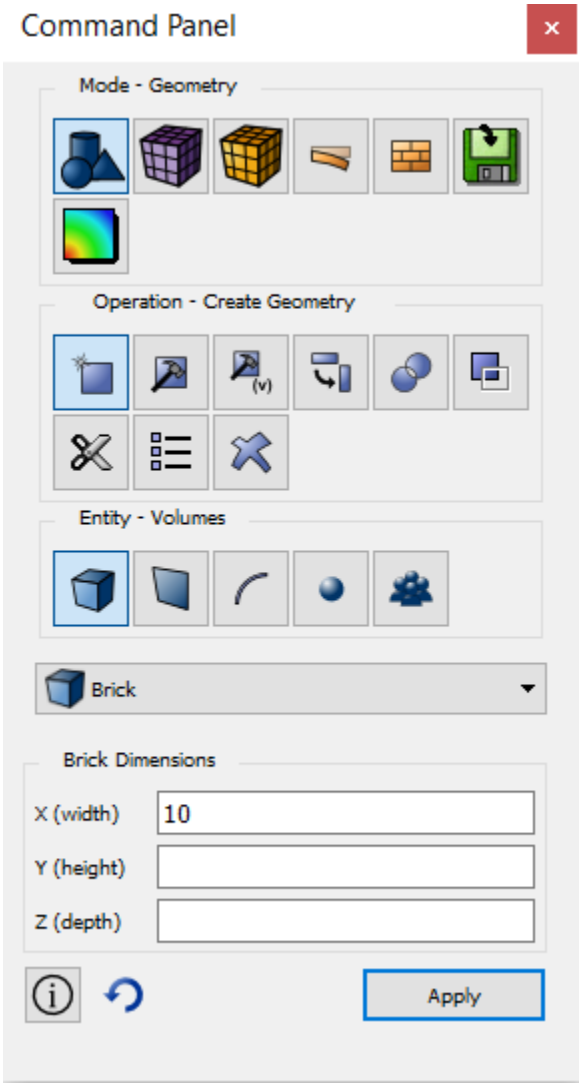


GUI Basic Tutorial Step 2

# *Step 2: Creating the Brick*

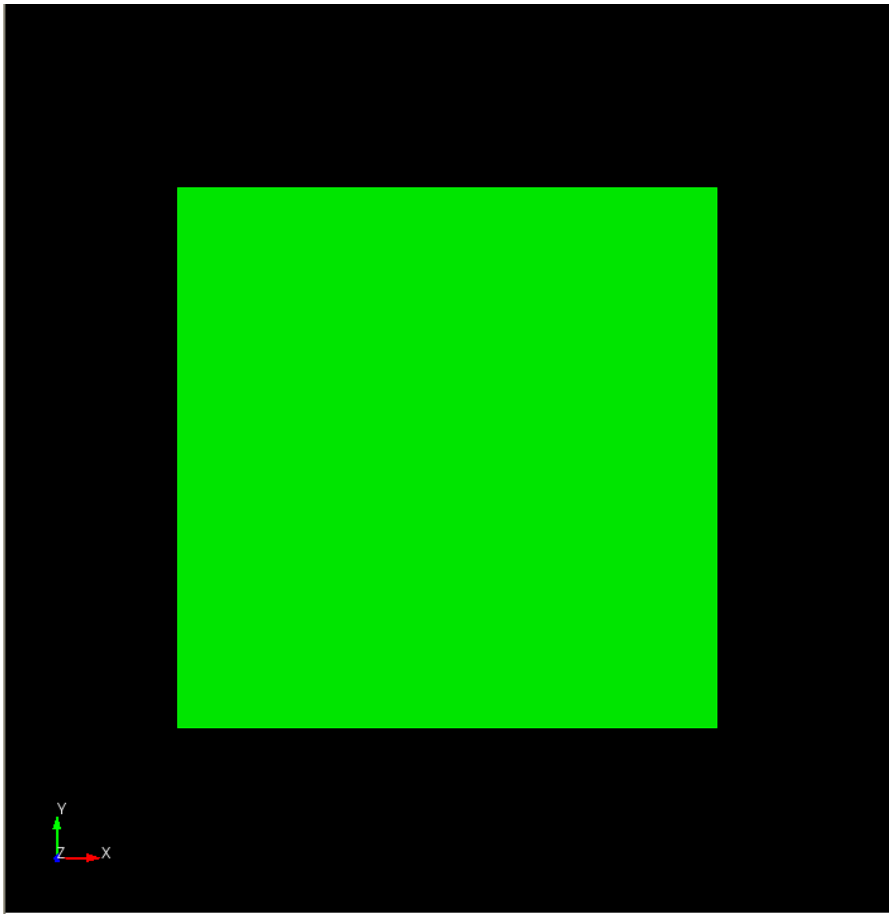
Now you may begin generating the geometry to be meshed. You will create a brick of width 10, depth 10 and height 10. The width and depth correspond to the x and y dimensions of the object being created. The "width" or x-dimension is screen-horizontal and the "depth" or y-dimension is screen-vertical. The height or z-dimension is out of the screen.

- On the Command Panel, select **Geometry**, then **Create Geometry**, then **Volume**. Brick is the default type.



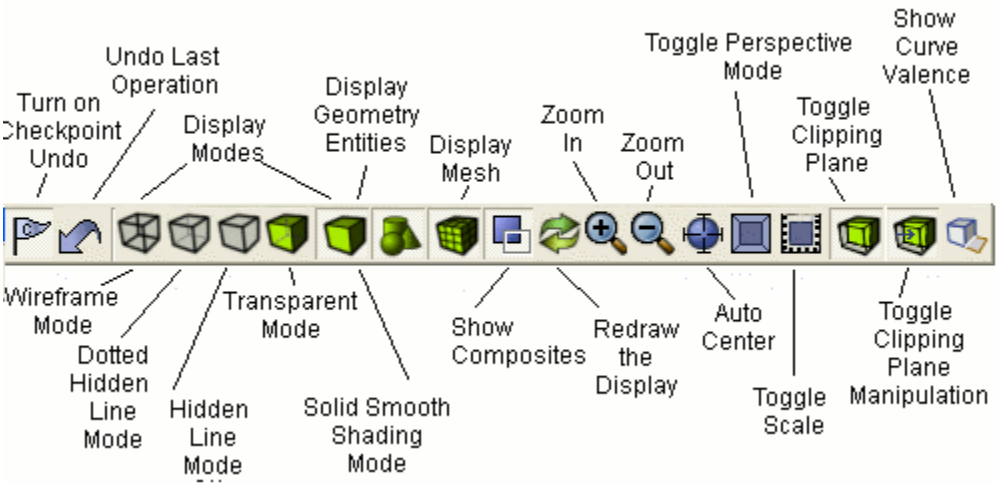

- Enter values for X, Y, and Z. Note, X (width) has a default value of 10. Select **Apply** to create the brick.

The brick should appear in your Graphics window as shown below.




**Display of Brick**

If you would like to change the rendering mode of your model, you may click on one of the view buttons in the Display Tools tool bar.

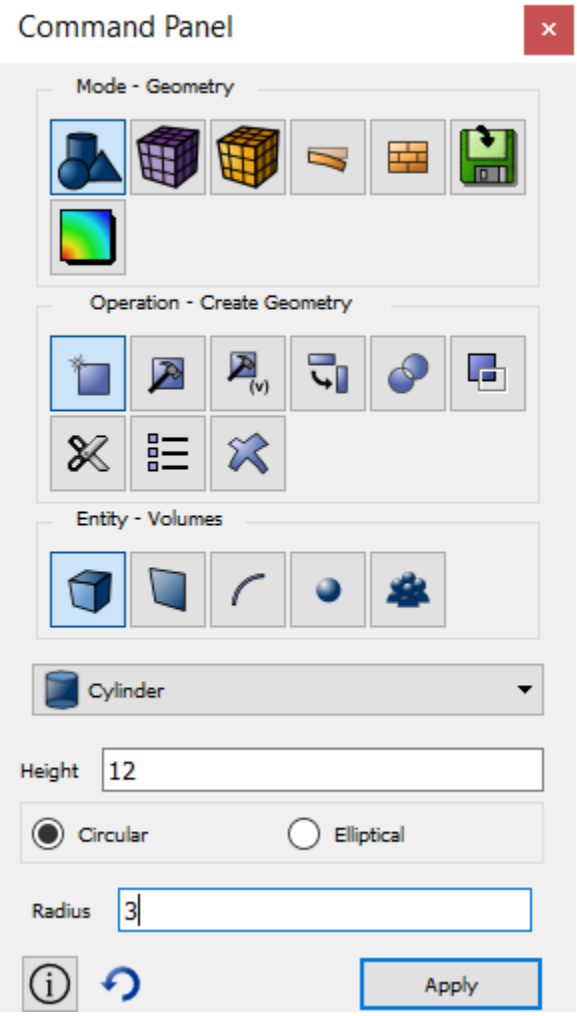


GUI Basic Tutorial Step 3

# *Step 3: Creating the Cylinder*

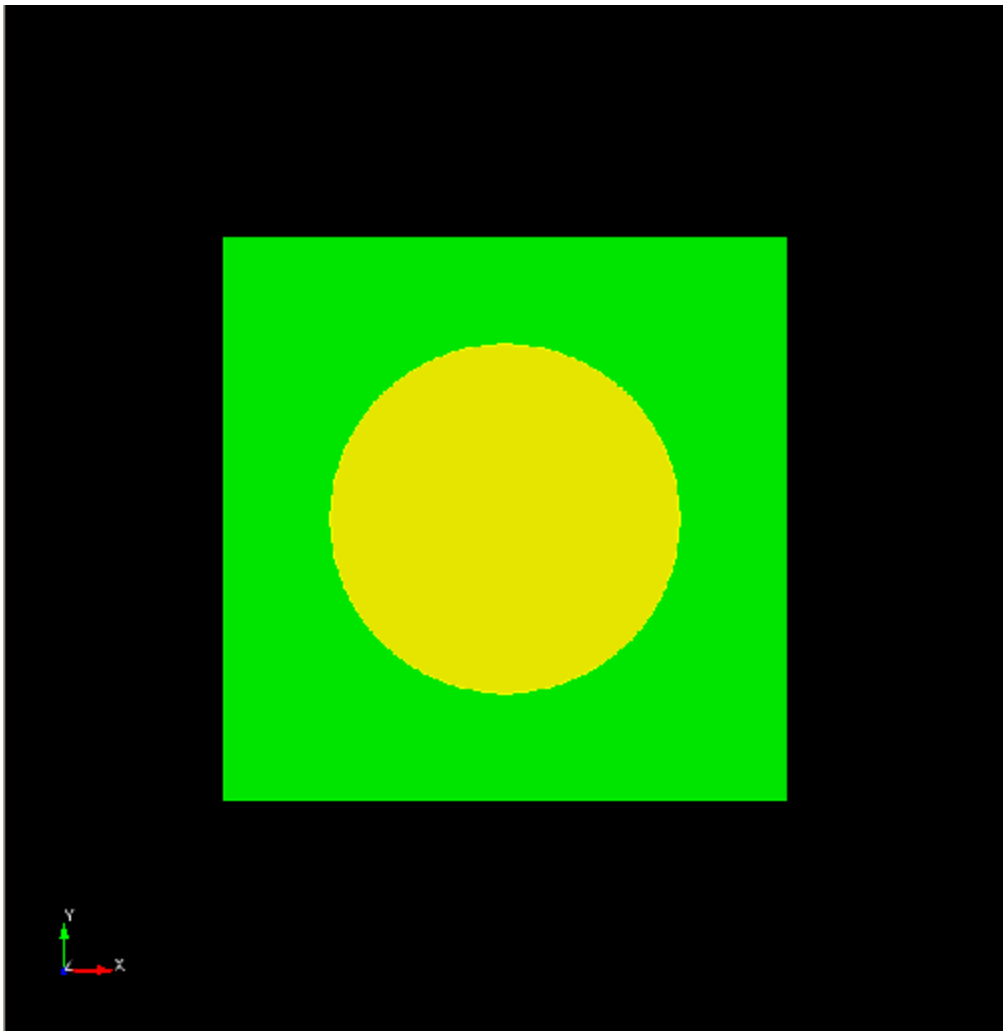
Now you must form the cylinder which will be used to cut a hole in the brick.

- Select **Cylinder** from the **Volume-type** combo box.




- Enter **12** for the height and **3** for the radius. Then select **Apply**.

The brick and the cylinder should appear in your display window as shown below:




**Brick and Cylinder**

GUI Basic Tutorial Step 4

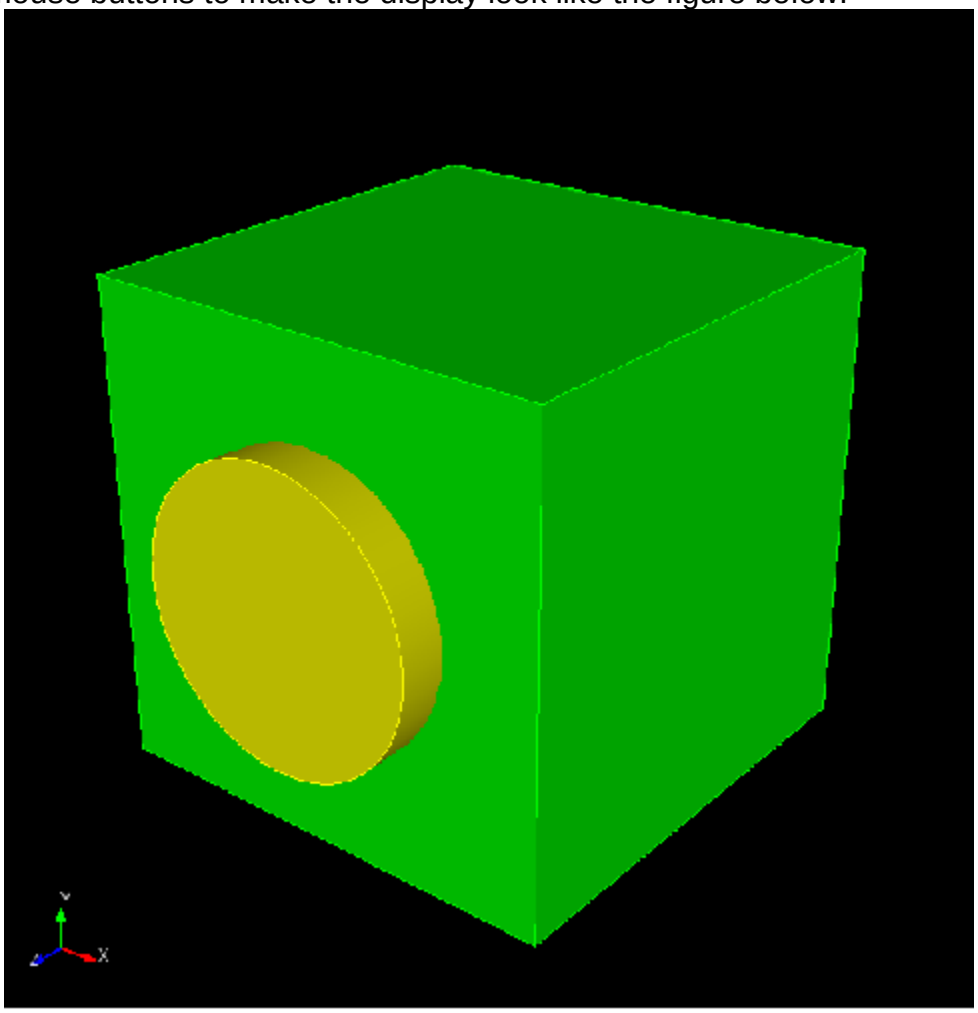
# *Step 4: Adjusting the Graphics Display*

The geometry is drawn in the graphics window in perspective mode, by default from a viewing direction of the +z axis. This view can now be adjusted to verify the proper orientation of the geometry just created.
The following button clicks apply for 3-button mice (these are the default GUI settings):

- **left** will pick when the mouse is over an entity. Left click will also pan when held down.

- **middle** will rotate
- **right** will show a context menu when an entity is selected. Right click will zoom when no entity is selected.

Mouse button behavior can be customized from the Tools-Options menu for use with non 3-button mice.
Use the mouse buttons to make the display look like the figure below.



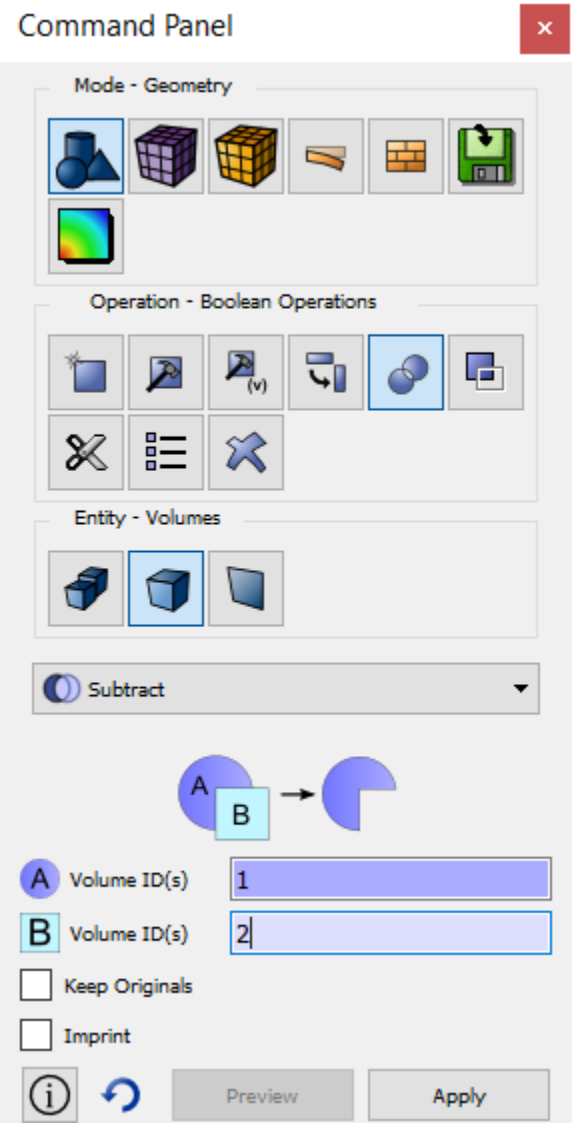
**View from Different Perspective**

GUI Basic Tutorial Step 5

# *Step 5: Forming the Hole*

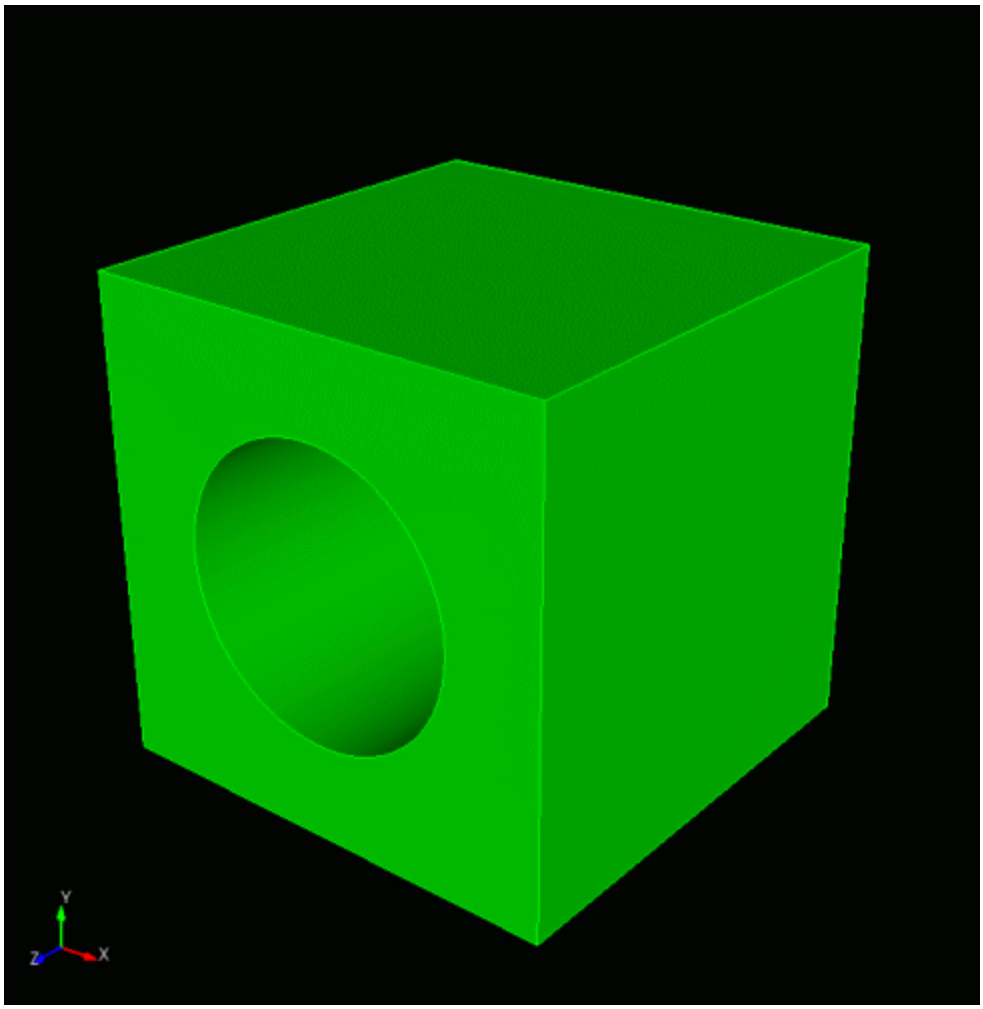
Now the cylinder can be subtracted from the brick to form the hole in the block.

- Select the **Boolean** operation button. Select **Volume** as the entity type. Then select **Subtract** from the Boolean combo box.




- Enter **2** for **Subtract Volume ID(s)** and **1** for **From Volume ID(s)**.
- Select the **Apply** button

You can also select the brick or cylinder interactively. Place the cursor in the **Subtract Volume ID(s)** field and click. This field is known as a **Pick Widget**. Clicking in a pick widget automatically sets the graphics pick mode for the entity type expected by the pick widget. Move the cursor to the graphics window and, using the left mouse button, select an entity. The id of the selected entity will be echoed into the pick widget field. Holding the control key while selecting entities in the graphics window will select multiple entities. Notice that both original volumes are deleted in the Boolean operation and replaced with a new volume, with an id of 1. The result of this operation is a single volume, a brick with a hole through it, as shown below.



**Brick after Subtracting Cylinder**

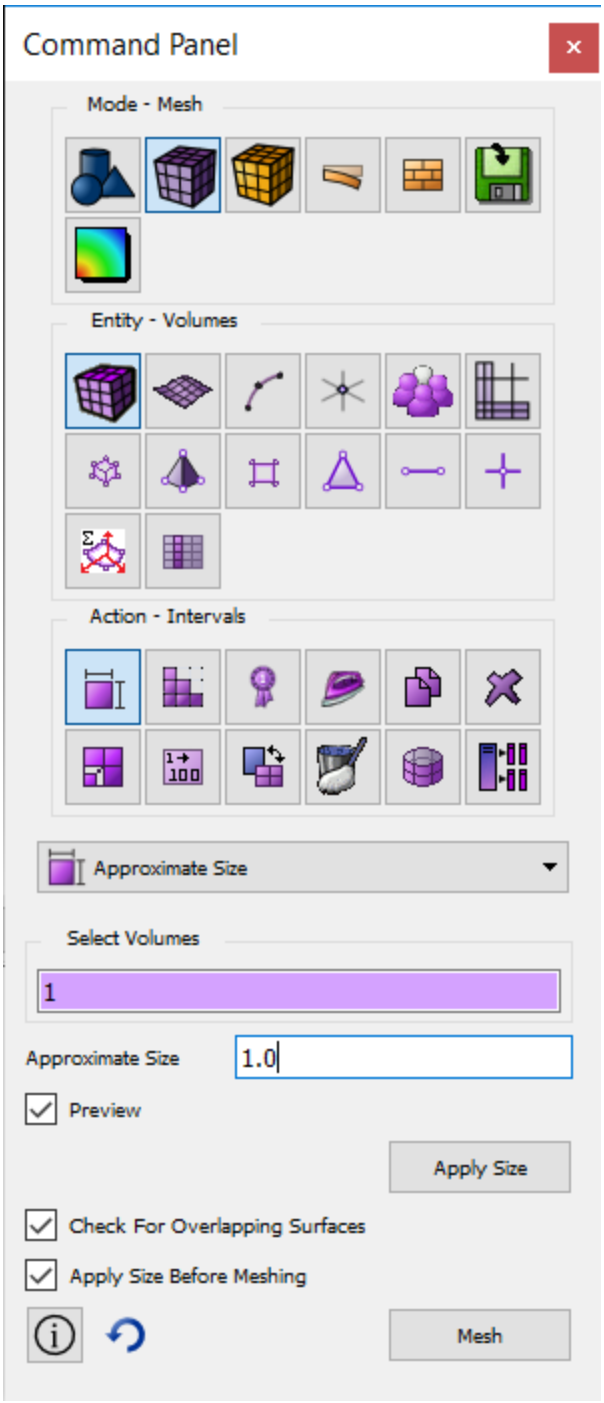
We have now completed creating the geometry, and are ready to generate a mesh.

GUI Basic Tutorial Step 6

# *Step 6: Setting Interval Sizes*

The volume shown in Step 5 will be meshed by sweeping a surface mesh from one side of the brick to the other. Before generating any mesh, the user must specify the size of the elements to be generated. In this example, one element size will be specified for the volume as a whole and a smaller size will be specified for around the hole. A direct interval setting will be specified for the sweep direction.
To set the interval size for the overall volume, do the following:

- Change the mode to **Meshing**, then select **Volume** followed by **Intervals**.

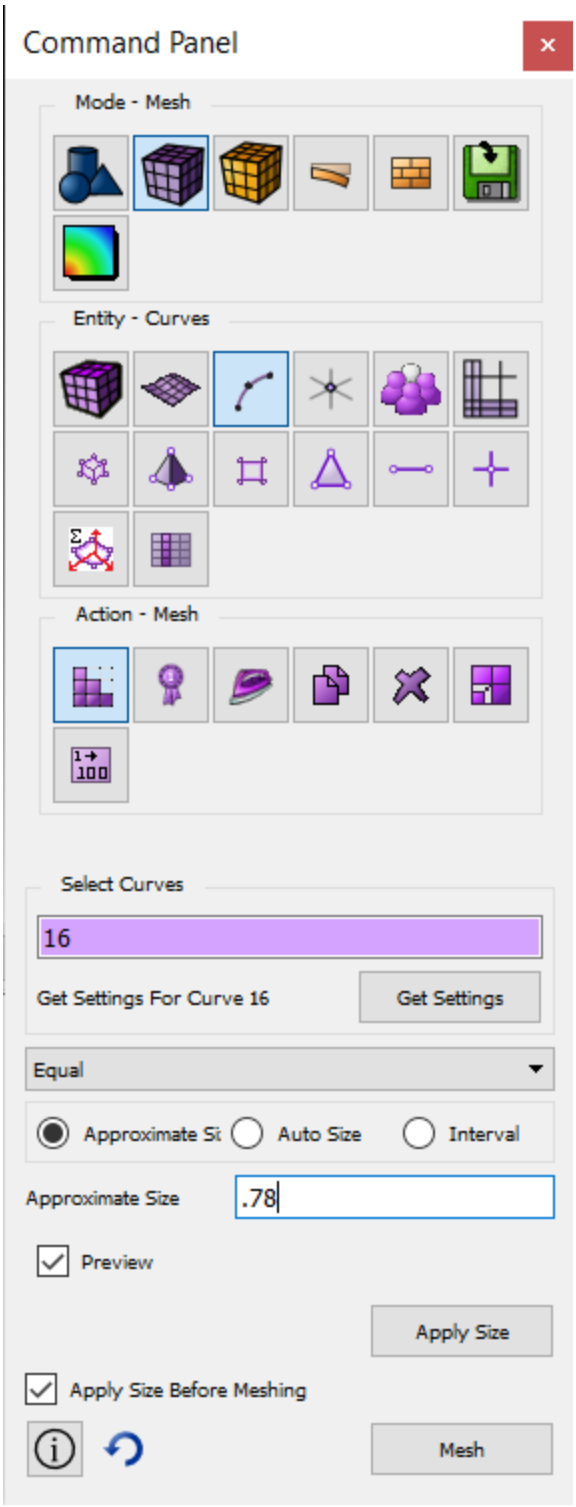



- Place the cursor into the **Select Volumes** field. Since this is a pick widget, click anywhere on the volume in the graphics window. Alternatively, type 1 in the field. Set the **Interval Size** to **1.0** and select **Apply Size**

Since the brick is 10 units in length on a side, this specifies that each straight curve is to receive approximately 10 mesh elements.
In order to better resolve the hole in the middle of the top surface, we set a smaller size for the curve bounding this hole.

- Change the object of the command panel to curve by selecting **Curve** from the **Entity** buttons and **Mesh** from the **Action** Buttons.

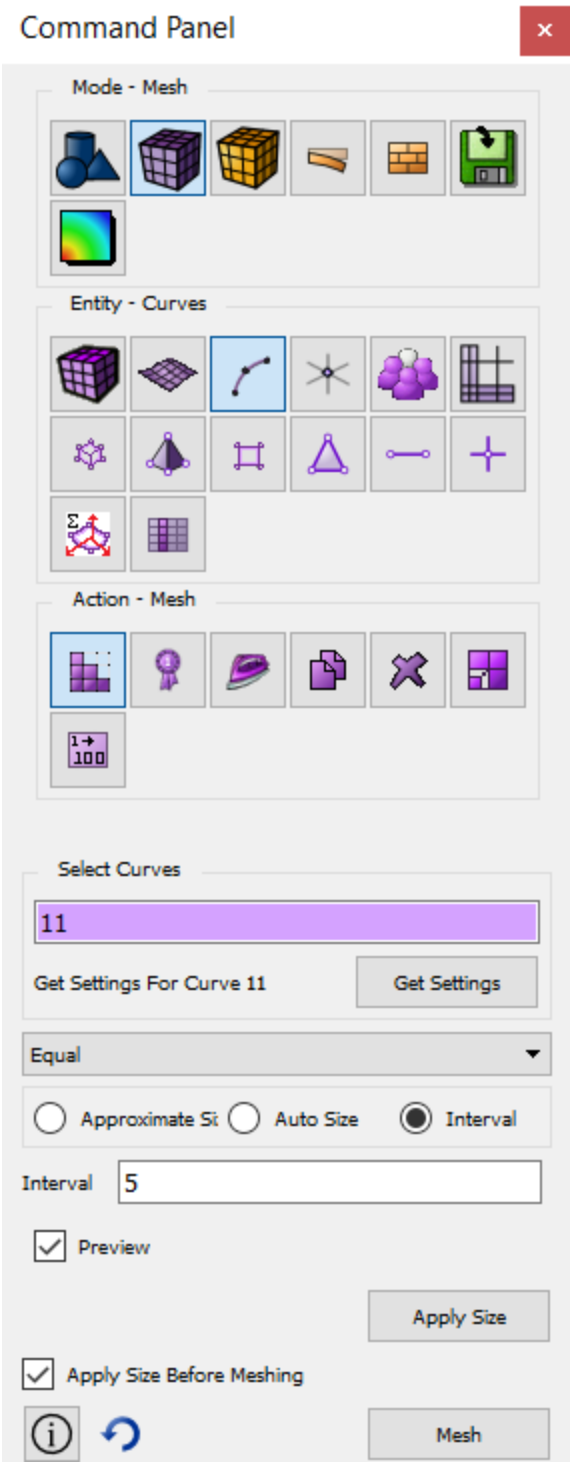
**Note:** There is not a separate interval action panel for curves. The interval and mesh actions for curves are grouped together in one panel.




- Place the cursor into the **Select Curves** pick widget field. Select the near end of the cylinder in the graphics window. Once you have selected the curve, the id of that curve, **16** should appear in the Selected Curves field. Select **Size**
- Enter **0.78** for the size and select **Apply Size**.

Finally, we would like to generate exactly 5 element layers in the sweep direction. This is accomplished by setting the intervals on one of the curves in the sweep direction.

- Place the cursor back into the **Selected Curves** field and enter **11**.
- Select the **Interval** radio button
- Enter an interval count of **5** and select **Apply**.

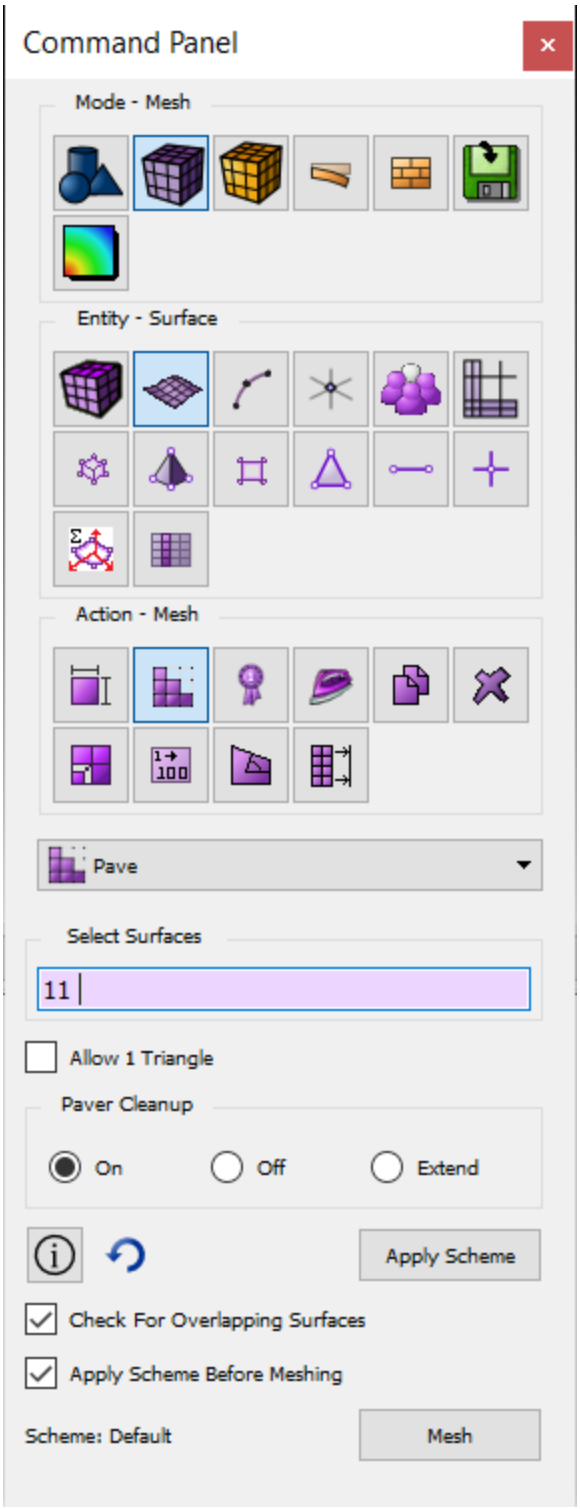



GUI Basic Tutorial Step 7

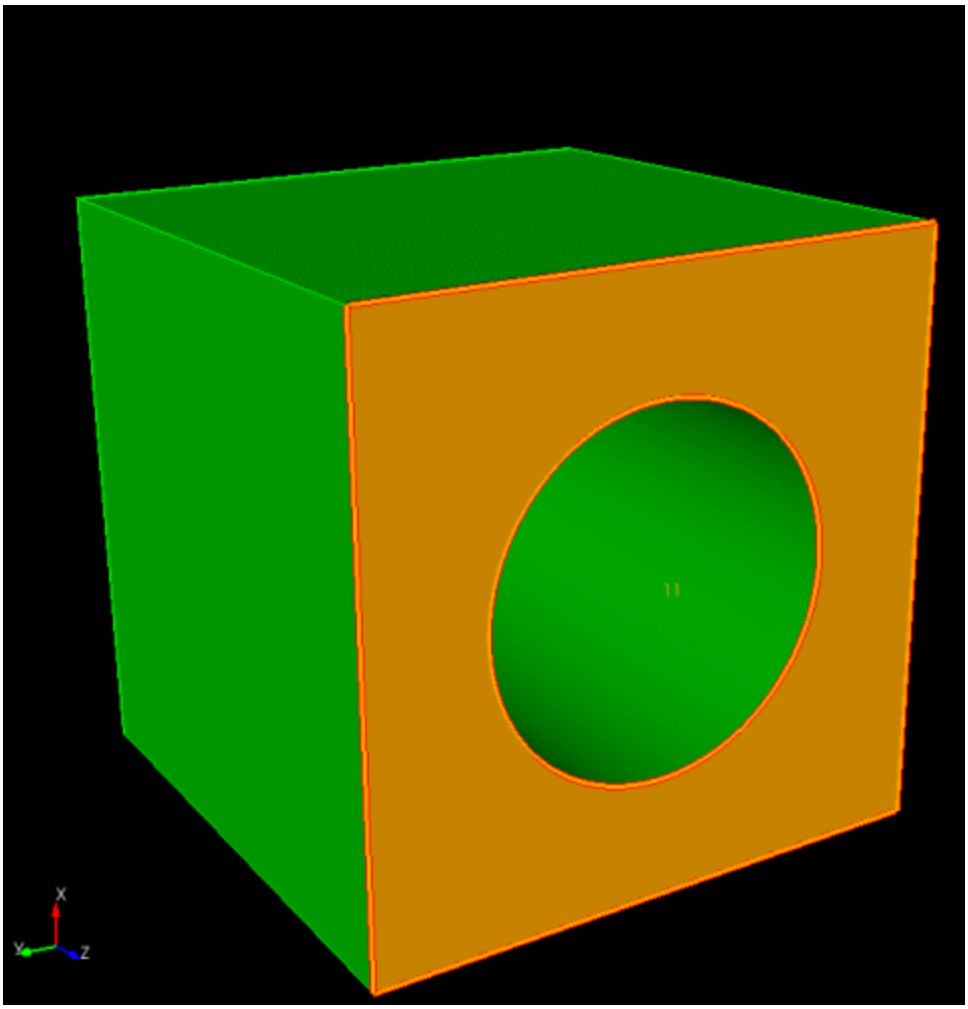
# *Step 7: Surface Meshing*

Now all necessary intervals have been set, the meshing can proceed. Begin by meshing the front surface (with the hole) using the paving algorithm. This is done in two steps. First, set the scheme for that surface to Pave, then issue the command to Mesh.

- Select **Surface** then **Mesh** buttons in the Control Panel.
- Select **Pave** in the **Available Mesh Schemes** combo box.

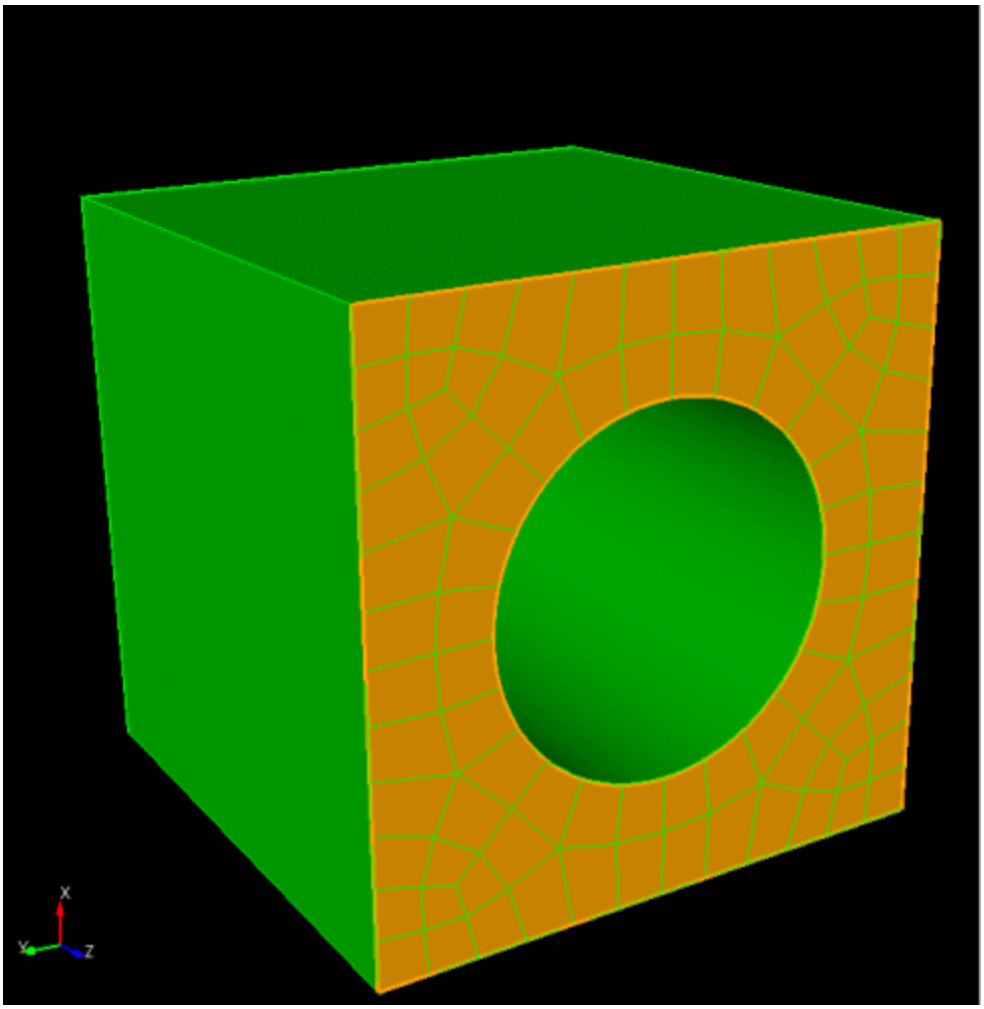



Place the cursor into the **Surface ID(s)** field. Select the front surface of the object by selecting anywhere within the region indicated. The id of **Surface 11** will be echoed in the field.




- Select the **Apply** button to set the scheme.
- Select the **Mesh** button to mesh the surface.

A mesh should be generated on surface 11 using the paving algorithm. The result is shown below.



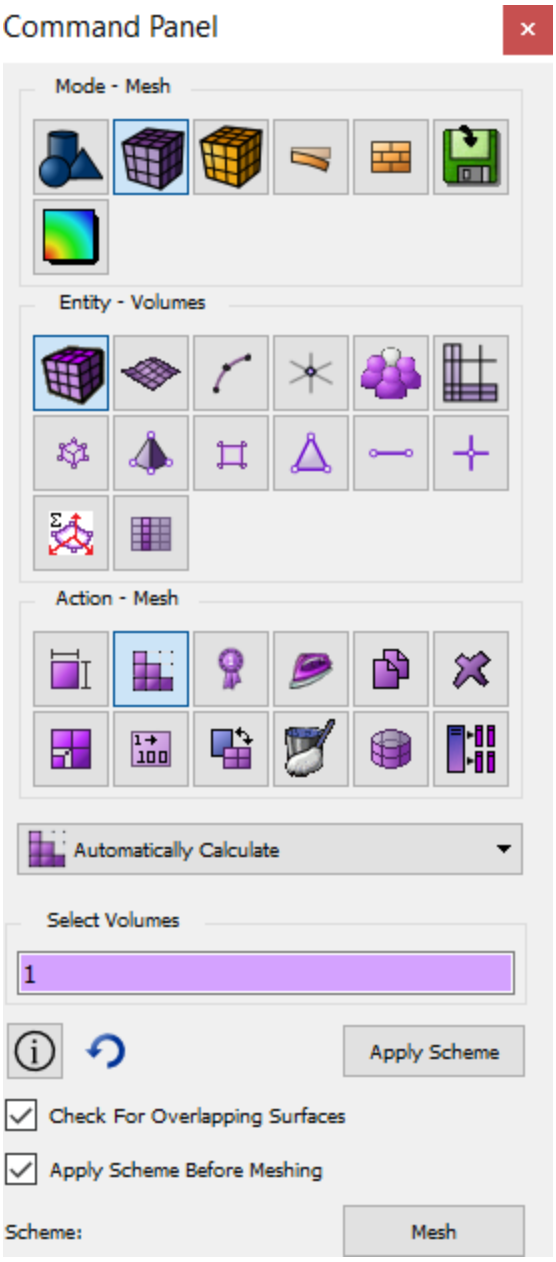
**Surface Meshed with Paving**

GUI Basic Tutorial Step 8

# *Step 8: Volume Meshing*

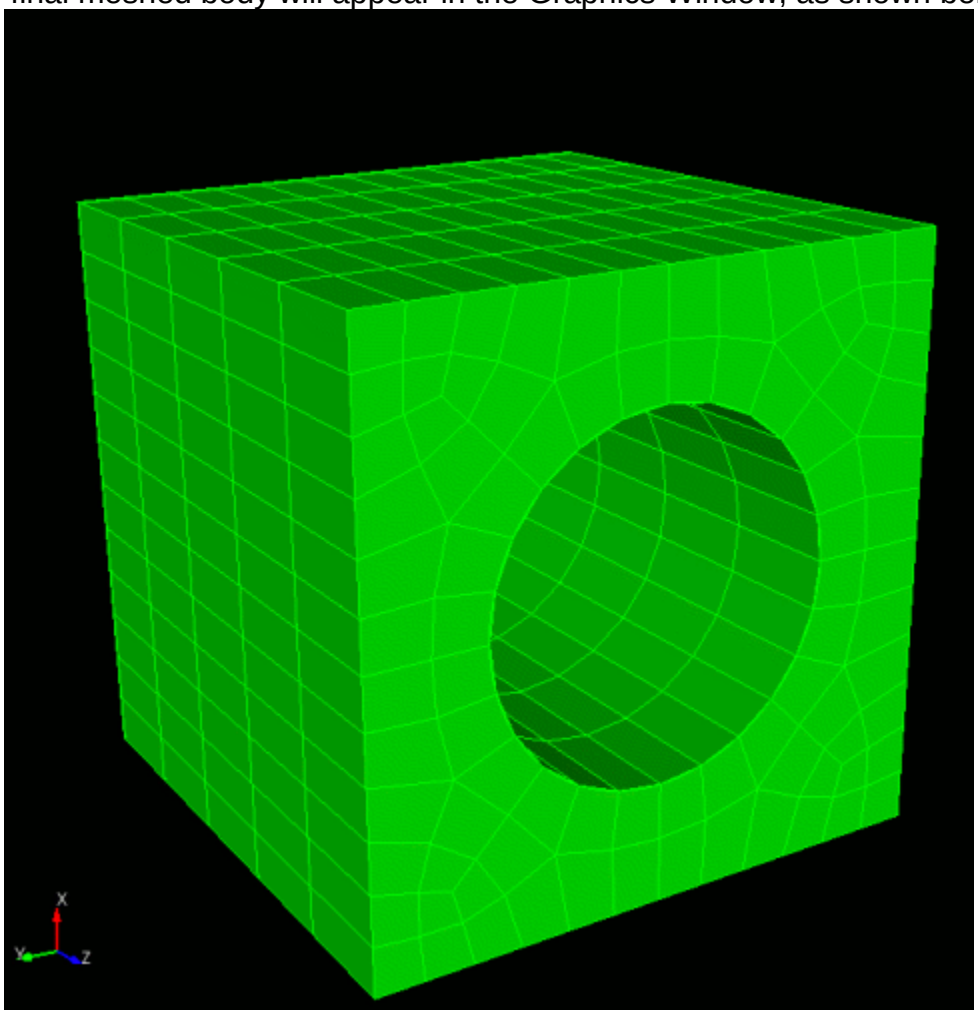
The volume mesh can now be generated. Again, the first step is to specify the type of meshing scheme that should be used and the second step is to issue the order to mesh. In certain cases, the scheme can be determined by CUBIT automatically. For sweepable volumes, the automatic scheme detection algorithm also identifies the source and target surfaces of the sweep automatically.
To instruct the code to automatically determine the meshing scheme, and in this case the source and target surfaces, do the following:

- Select **Volume** then **Mesh** on the control panel.




- Place the cursor into the **Volume ID(s)** field then select the volume in the Graphics Window. The id of **Volume 1** should appear in the field. Choose the **Automatically Calculate** scheme using the combo box provided.
- Select **Apply Scheme** to set the scheme. Then select **Mesh** to mesh the volume.

The final meshed body will appear in the Graphics Window, as shown below:

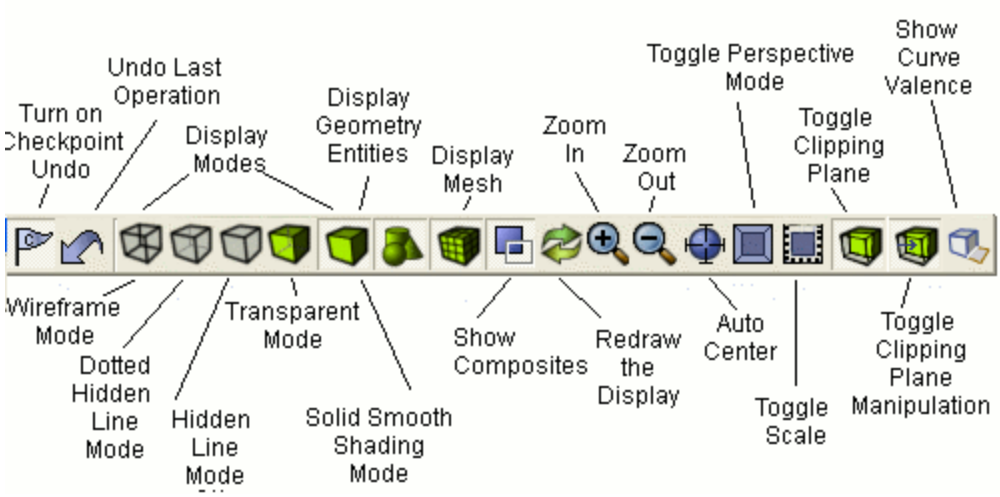



**Smooth Shade View of Volume Mesh**

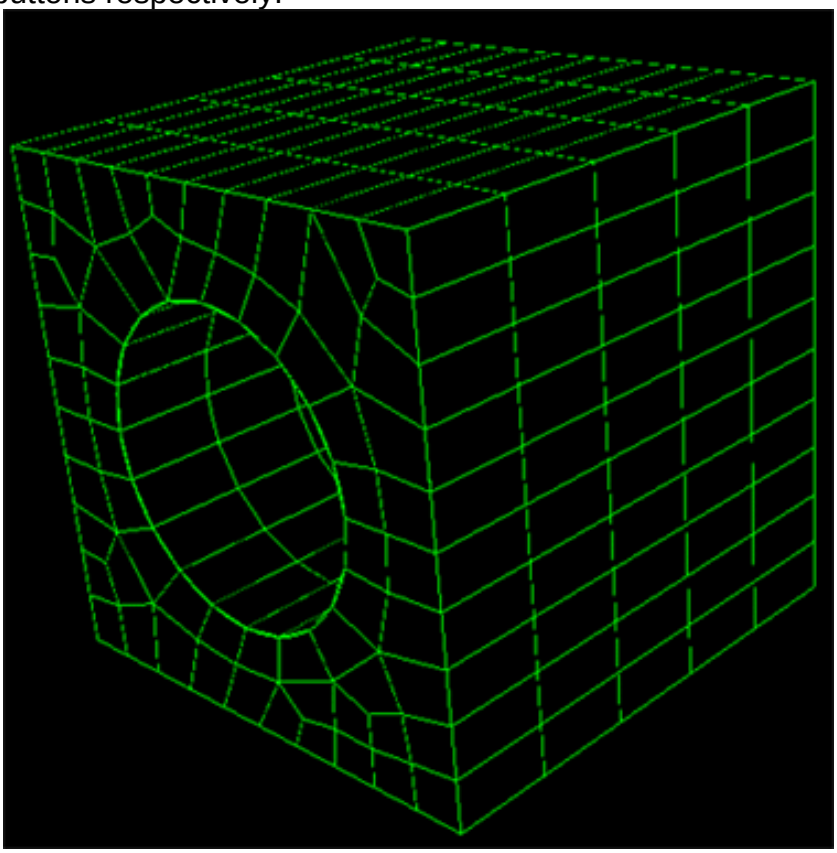
GUI Basic Tutorial Step 9

# *Step 9: Inspecting the Model*

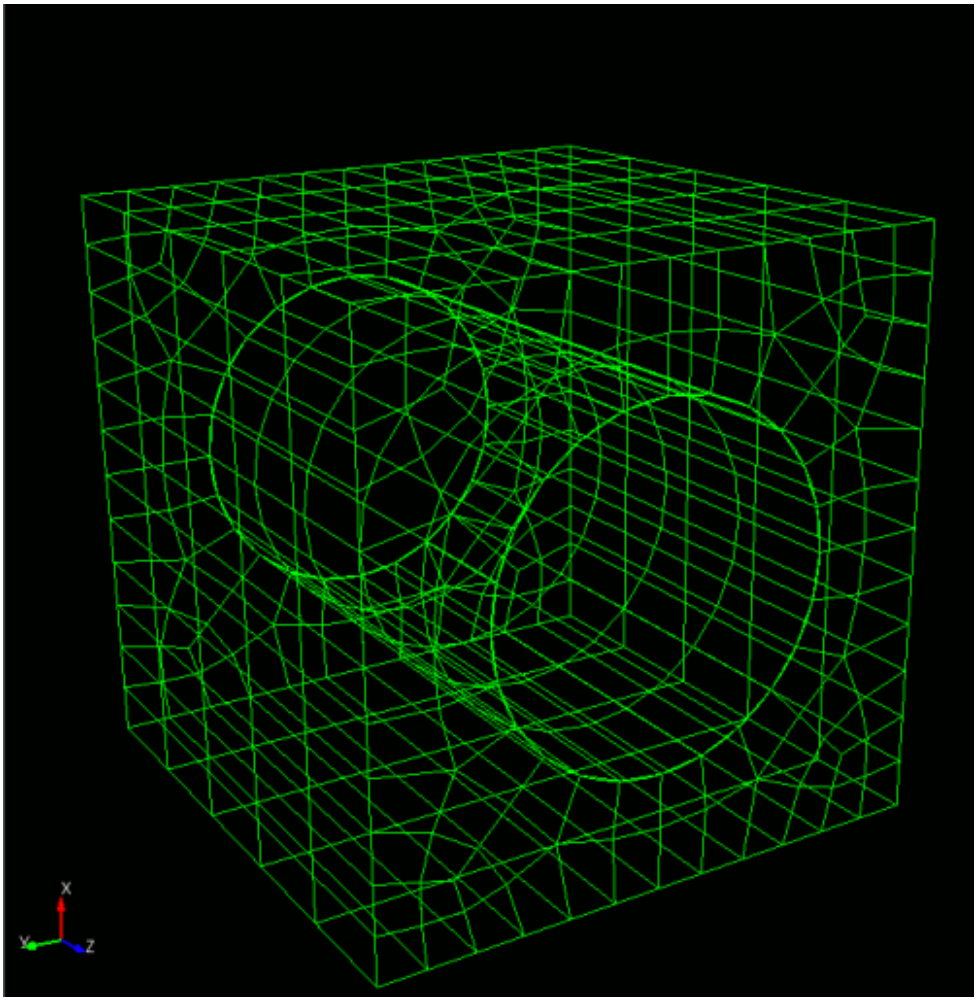
The type, quality, and speed of rendering the image can be controlled in CUBIT by selecting one of the buttons in the **Display** icon group. These icons appear by default in the icon bar above the graphics window. They can be used to change the display mode to wire frame, hidden line, true hidden line, transparent or smooth shade.




For example, the following two figures result from selecting the **Hidden Line** and **Wire Frame Mode** buttons respectively.



**Hidden Line View of Mesh**



**Wire Frame View of Mesh**

Although CUBIT automatically computes limited quality metrics after generating a mesh and warns the user about certain cases of bad quality, it is still a good idea to inspect a broader set of quality measures. To do this, use the **Command Window** to enter the command:

```
CUBIT> quality volume 1 Allmetrics
```

The results of the quality are displayed in the Command Window. For an explanation of each quality metric along with acceptable ranges, see Mesh Quality Assessment. For the purposes of this tutorial, you can assume the quality metrics shown are in an acceptable range.

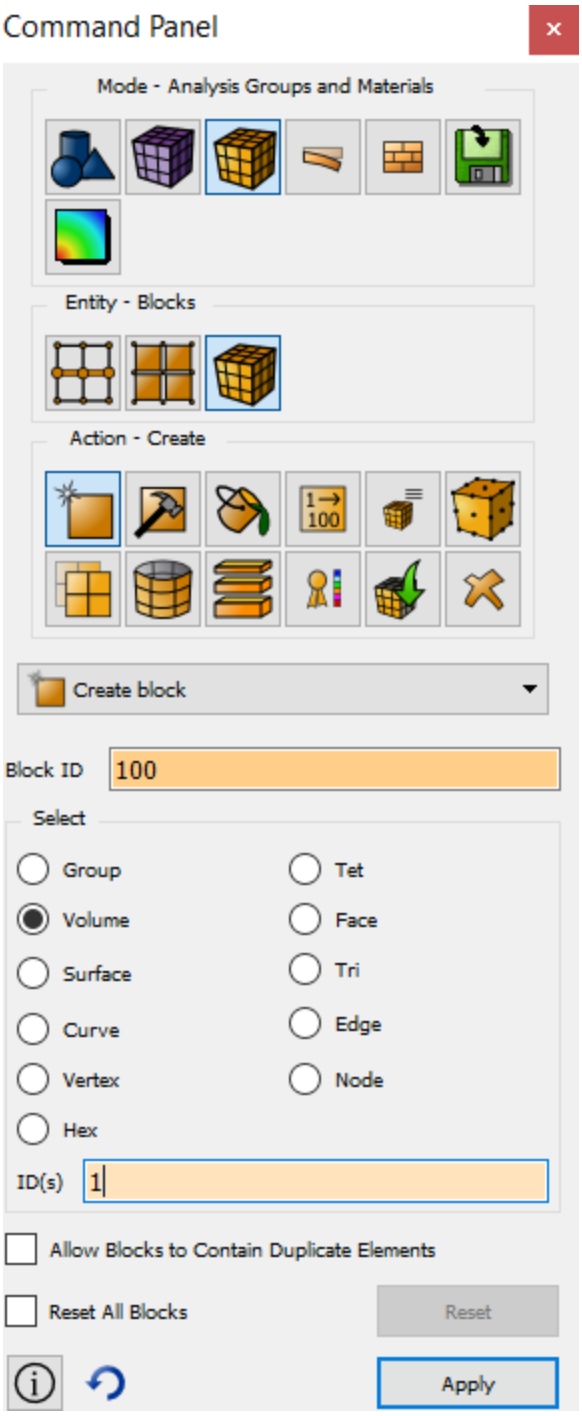


GUI Basic Tutorial Step 10

# *Step 10: Defining Boundary Conditions*

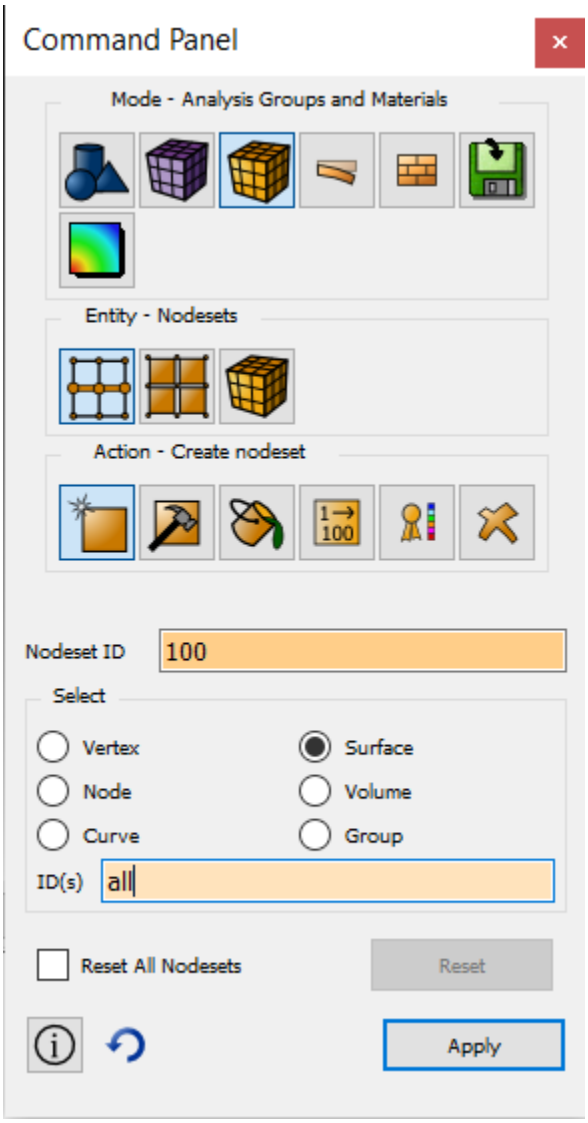
Let us assume that we need to define one material type for the entire mesh, and a single node-based boundary condition on all surfaces. This is accomplished by identifying an Element Block and a Nodeset, respectively; the id numbers assigned to these entities are assigned by the user, usually by some convention meaningful to the analysis to be done. The element block and nodeset are identified from the Materials and Properties button on the control panel.

- Select the **Analysis Groups and Materials** button and then **Blocks** in the Control Panel window
- Select the **Create** button
- Select **Create block** in the pull down menu
- Enter 100 into the Block ID field
- Select the **Volume** radio button
- Enter the id of **Volume 1** by selecting it in the graphics window, or just manually entering in **ID(s)** field
- Press **Apply**




Create a nodeset by following the steps below

- Open the **Nodeset** window on the Control Panel
- Select the **Create** button
- Enter a Nodeset id of **100**
- Select the **Surface** radio button and type **all** in the ID(s) field
- Press **Apply**

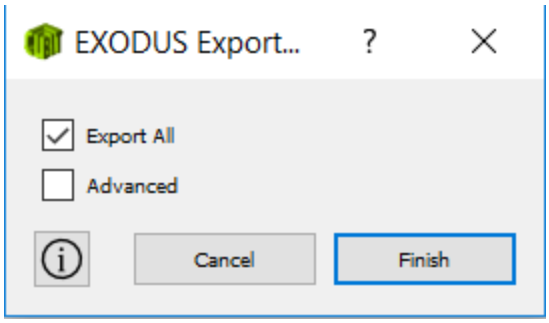



GUI Basic Tutorial Step 11

# *Step 11: Exporting the Mesh*

Finally, the mesh needs to be written to an Exodus II file. This is easily done:

- From the **File** menu, select **Export**.
- Set the file export type to **Genesis Files** from file type combo box.
- Enter a file name in the dialog, such as **brick_with_hole.g**, and select **Save**. Since this is a standard file management dialog, the user may browse or use any other file management functionality supported by the platform.
- Select the **Export All** check box




- Select **Finish** to export the mesh.



# command line

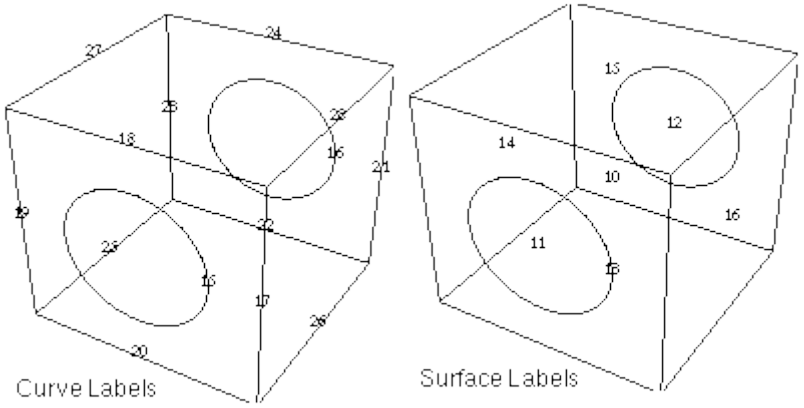
Command Line Basic Tutorial

# *Overview*

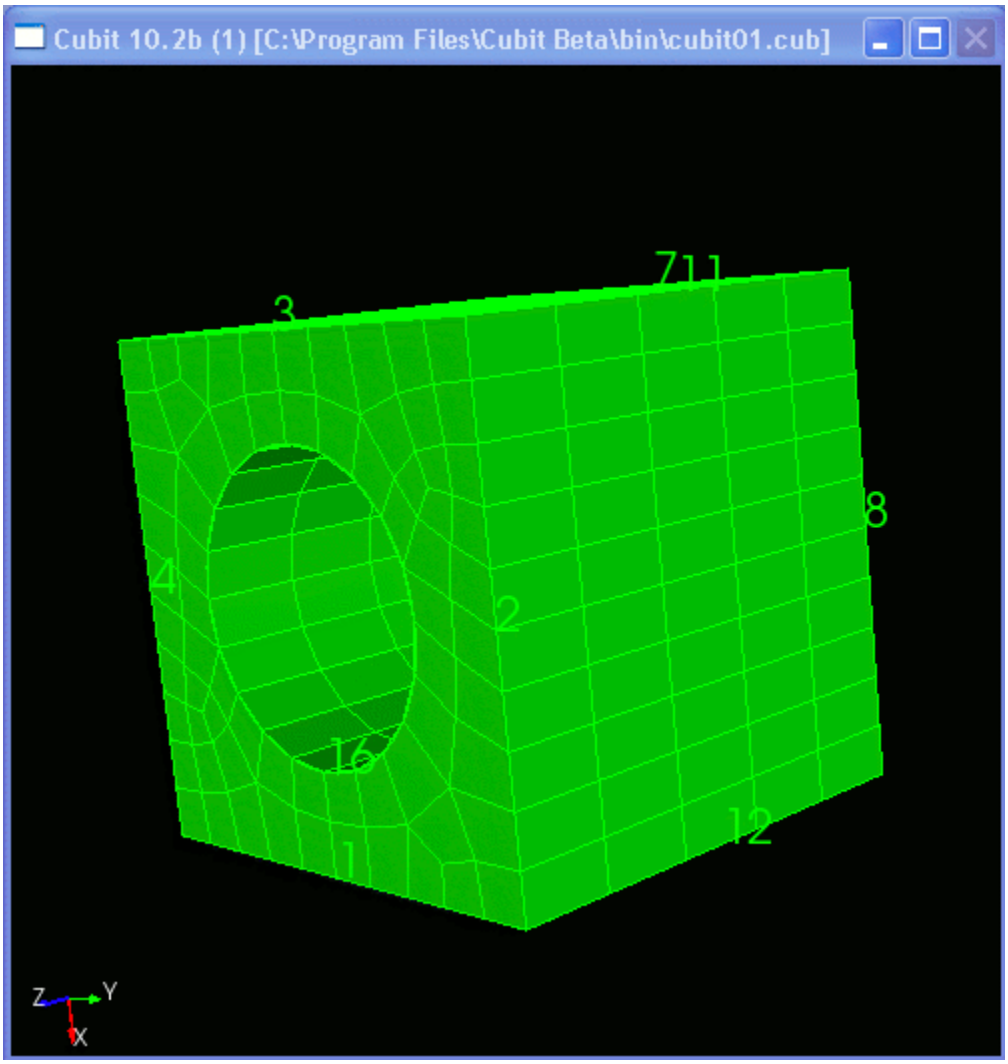
This tutorial demonstrates the use of CUBIT to create and mesh a brick with a through-hole. The primary steps in performing this task are:

- Creating the geometry
- Setting the interval sizes and meshing schemes
- Meshing the geometry
- Specifying the boundary conditions
- Exporting the mesh

Each of these steps is described in detail in the following sections. The geometry in this tutorial is a brick with a cylindrical hole in the center, shown in the figure below. This figure also shows the curve and surface identification (ID) numbers, which are referenced in the command lines shown with each step. The final meshed body is shown in the next figure.



**Geometry for Cube with Cylindrical Hole**



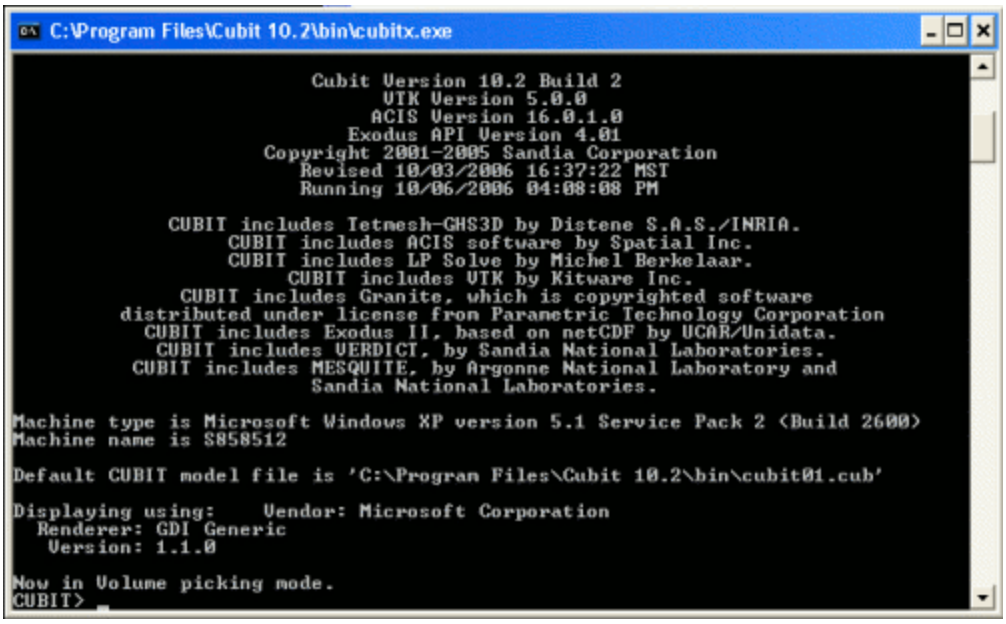

**Generated Mesh for Cube with Cylindrical Hole**



CL Basic Tutorial Step 1

# *Step 1: Beginning Execution*

Type "cubit" from a UNIX prompt to begin execution of CUBIT. A CUBIT console window will appear which tells the user which CUBIT version is being run and the most recent revision date. An example of the UNIX output window is shown below. This window echoes the commands and relays information about the success or failure of attempted actions.




Some things to notice are:

- At the top of the CUBIT window you will be told where the commands entered in this CUBIT session will be journaled. For example: "Commands will be journaled to `cubit01.jou' for this example.
- In addition to the CUBIT version, the code also reports the versions of ACIS and VTK that have been compiled into CUBIT.
- The command line prompt appears after the banner screen, and appears as "**CUBIT>**".
- Commands are entered at that prompt, followed by the "Enter" key.
- Upon startup, a graphics window should also appear, with an axis triad in the lower left hand corner (this window will not appear if CUBIT is started with the -nographics option.)



CL Basic Tutorial Step 2

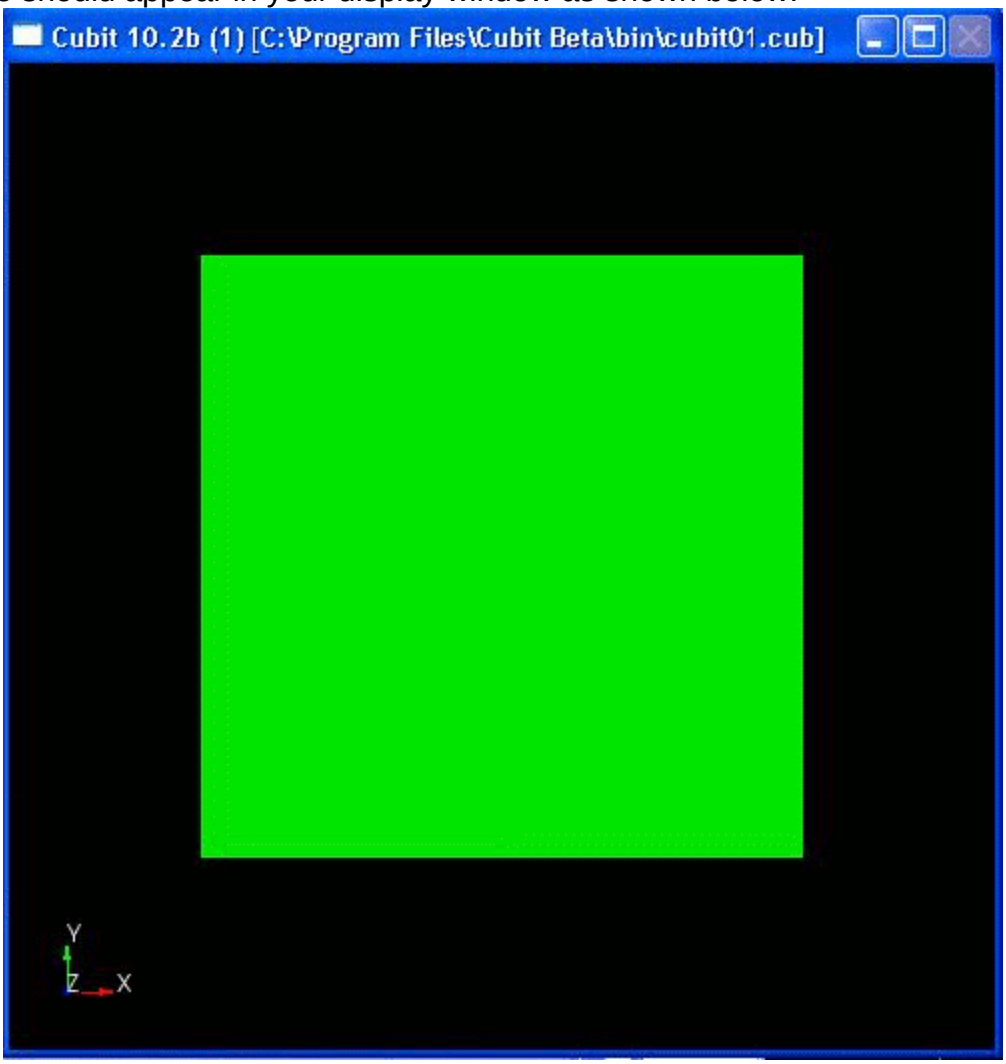
# *Step 2: Beginning Execution*

Now you may begin generating the geometry to be meshed. You will create a brick of width 10, depth 10 and height 10. The width and depth correspond to the x and y dimensions of the object being created. The "width" or x-dimension is screen-horizontal and the "depth" or y-dimension is screen-vertical. The height or z-dimension is out of the screen. The command to create this object is:

```
cubit> create brick width 10 depth 10 height 10 (OR)
```

```
cubit> create brick x 10
```

The cube should appear in your display window as shown below:




**Display of Brick**

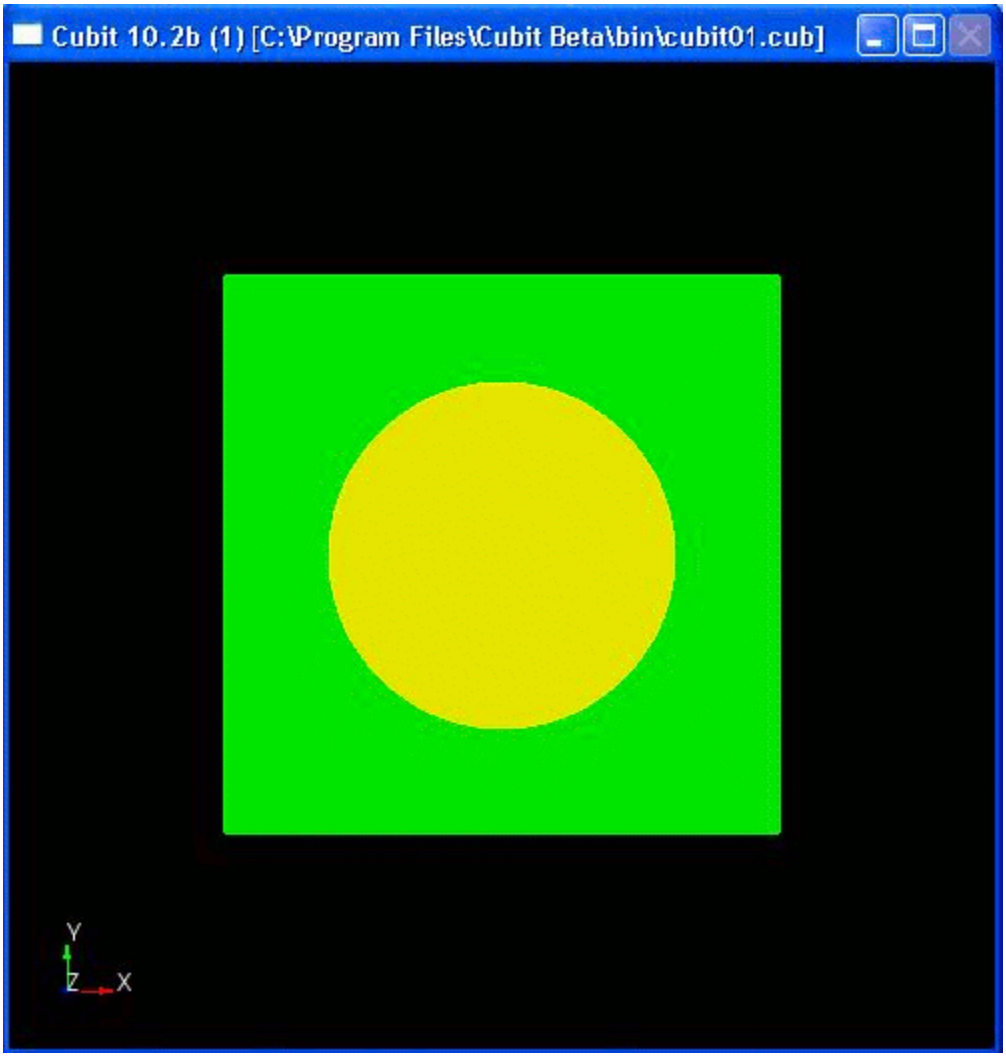
CL Basic Tutorial Step 3

# *Step 3: Creating the Cylinder*

Now you must form the cylinder which will be used to cut the hole from the brick. This is accomplished with the command:

```
cubit> create cylinder height 12 radius 3
```

At this point you will see both a brick and a cylinder appear in the CUBIT display window, as shown below:




**Brick and Cylinder**



CL Basic Tutorial Step 4

# *Step 4: Adjusting the Graphics Display*

The geometry is drawn in the graphics display in perspective mode by default from a viewing direction of the +z axis. This view can now be adjusted to verify the proper orientation of the geometry just created. The orientation of the geometry can be adjusted using the command line or interactively with the mouse.

#### Command Line

You can adjust the orientation of the object from the command line. For example, the from command can be used as follows

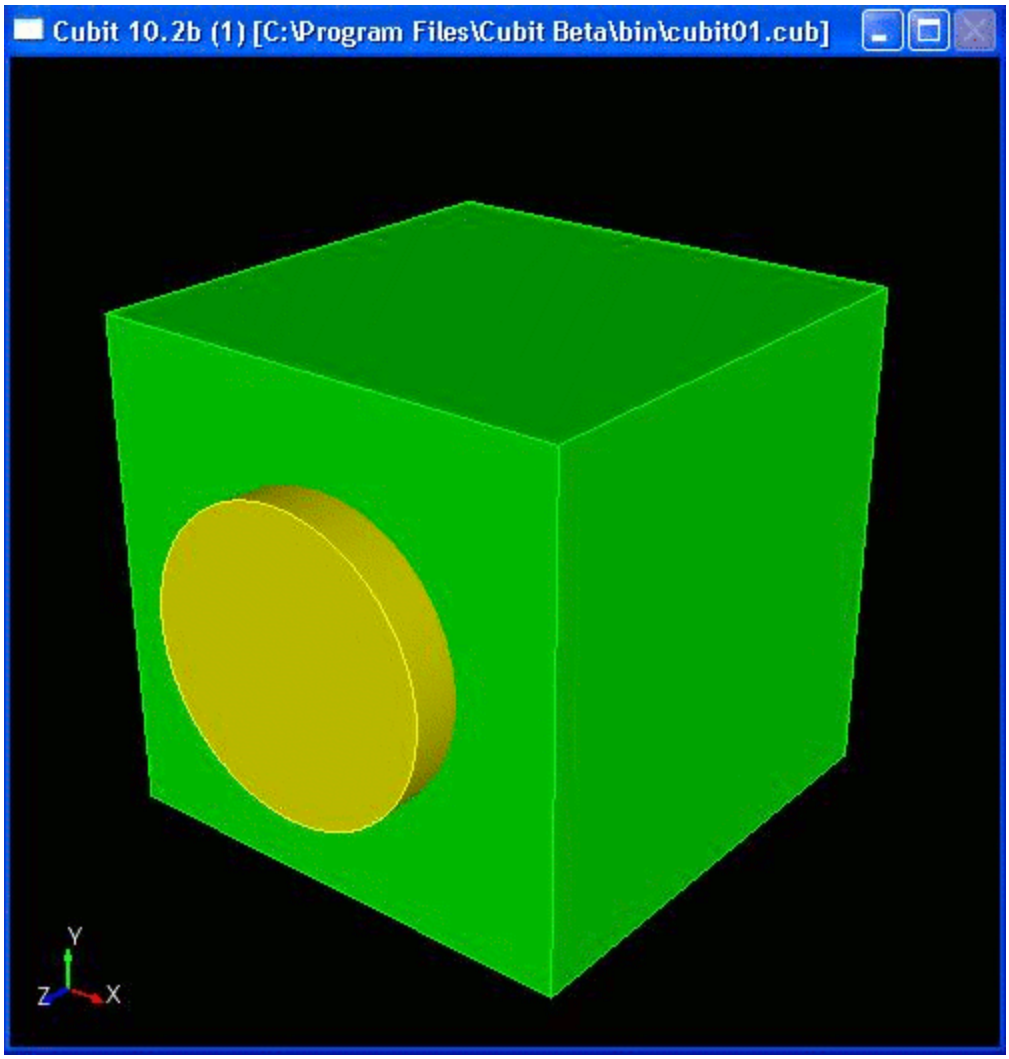
```
cubit>from 20 15 25

cubit>display
```

#### Mouse

To interactively change the orientation, activate your graphics window by placing your cursor in the window or by clicking at the top of it (this will vary depending upon your window settings in your operating system).

- Use the **left mouse button** to interactively rotate the view
- Use the **middle mouse button** to zoom in or out
- Use the **right mouse button** to pan the view.

Use the mouse buttons to make the display look the figure below:




**View from a Different Perspective**

CL Basic Tutorial Step 5

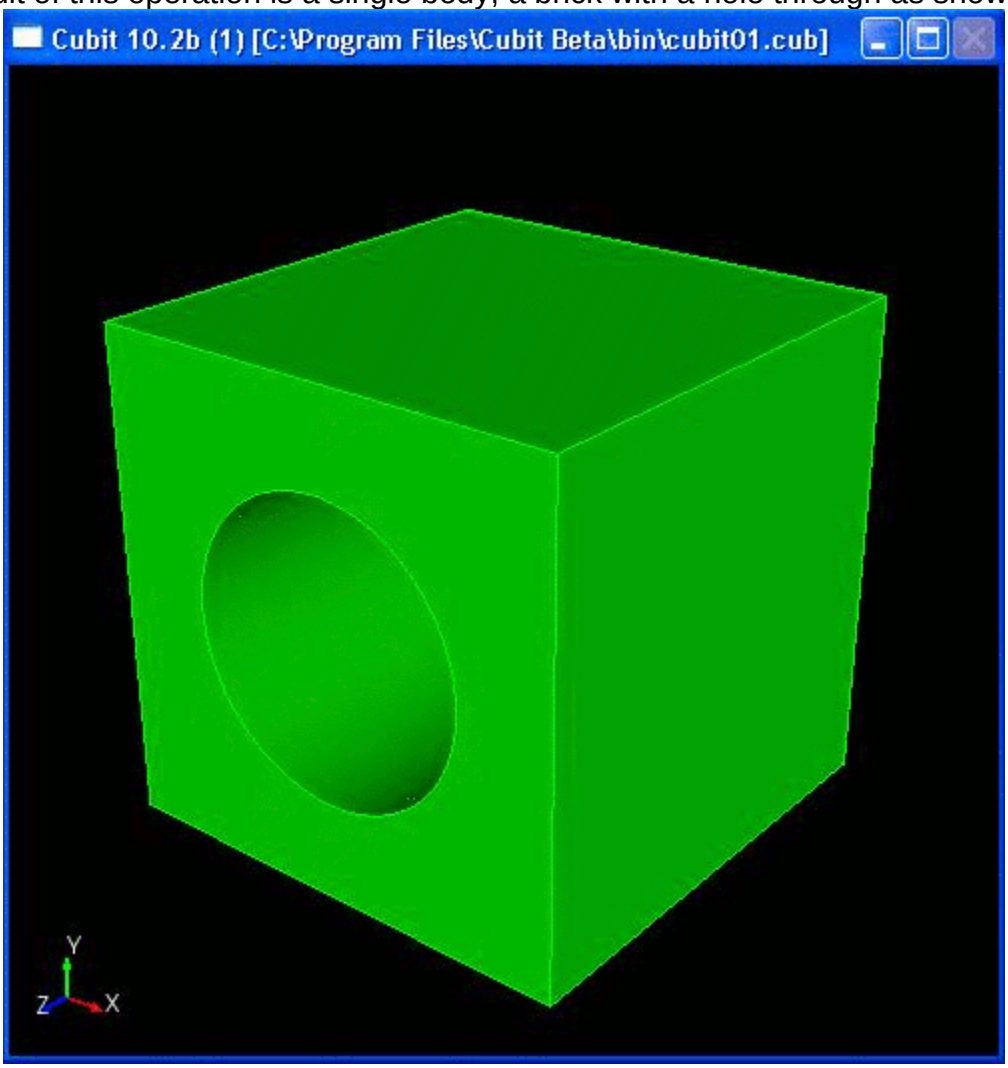
# *Step 5: Forming the Hole*

Now, the cylinder can be subtracted from the brick to form the hole in the block. Issue the following command:

```
cubit> subtract 2 from 1
```

Note that both original volumes are deleted in the Boolean operation and replaced with a new volume (with an id of 1) which is the result of the Boolean operation Subtract .

The result of this operation is a single body, a brick with a hole through as shown below:




**Brick after Subtracting the Cylinder**

We have now completed creating the geometry, and are ready to generate a mesh.



CL Basic Tutorial Step 6

# *Step 6: Setting Interval Sizes*

The volume shown in Step 5 will be meshed by sweeping a surface mesh from one side of the brick to the other. Before generating any mesh, the user must specify the size of the elements to be generated. In this example, one element size will be specified for the volume as a whole and a smaller size will be specified for around the hole. A direct interval setting will be specified for the sweep direction.

To set the interval size for the overall volume, enter the command

```
cubit> volume 1 size 1.0
```

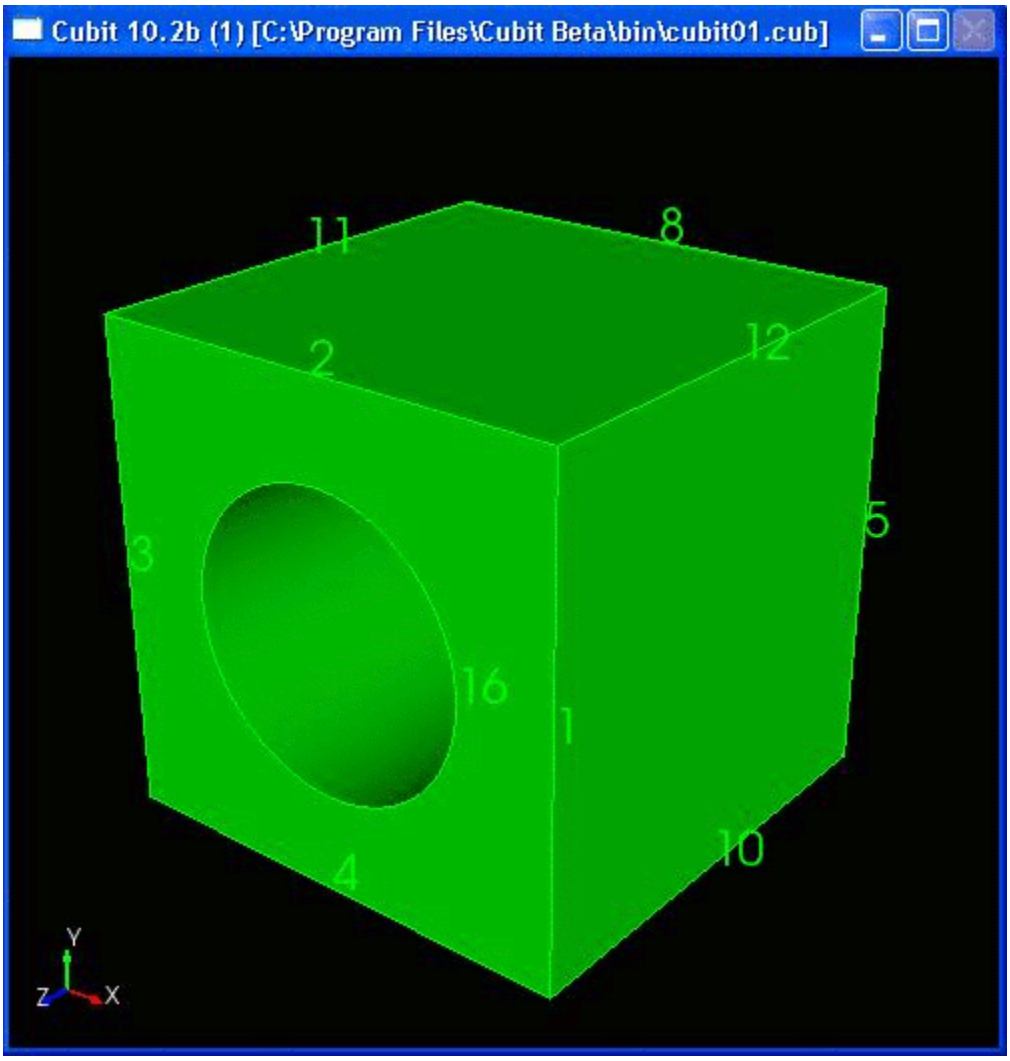
Since the brick is 10 units in length on a side, this specifies that each straight curve is to receive approximately 10 mesh elements.
In order to better resolve the hole in the middle of the top surface, we set a smaller size for the curve bounding this hole. To find the id number of the curve bounding the hole, the user can either pick the curve (See Selecting Entities with the Mouse) or turn curve labels on and regenerate the view. To do the latter, use the command

```
cubit> label curve on

cubit> display
```

The default size of the labels can sometimes be too small to read. To change the text size, use the graphics text size command:

```
cubit> graphics text size 2

cubit> display
```

The result is shown in the figure below. Then the interval size can be set for the appropriate curve:




**Geometry with Curve Labeling Turned on**

```
cubit> curve 16 interval size 0.78
```

Finally, we would like to generate exactly 5 element layers in the sweep direction. This is accomplished by setting the intervals on curve 11:

```
cubit> curve 11 interval 5
```



CL Basic Tutorial Step 7

# *Step 7: Surface Meshing*

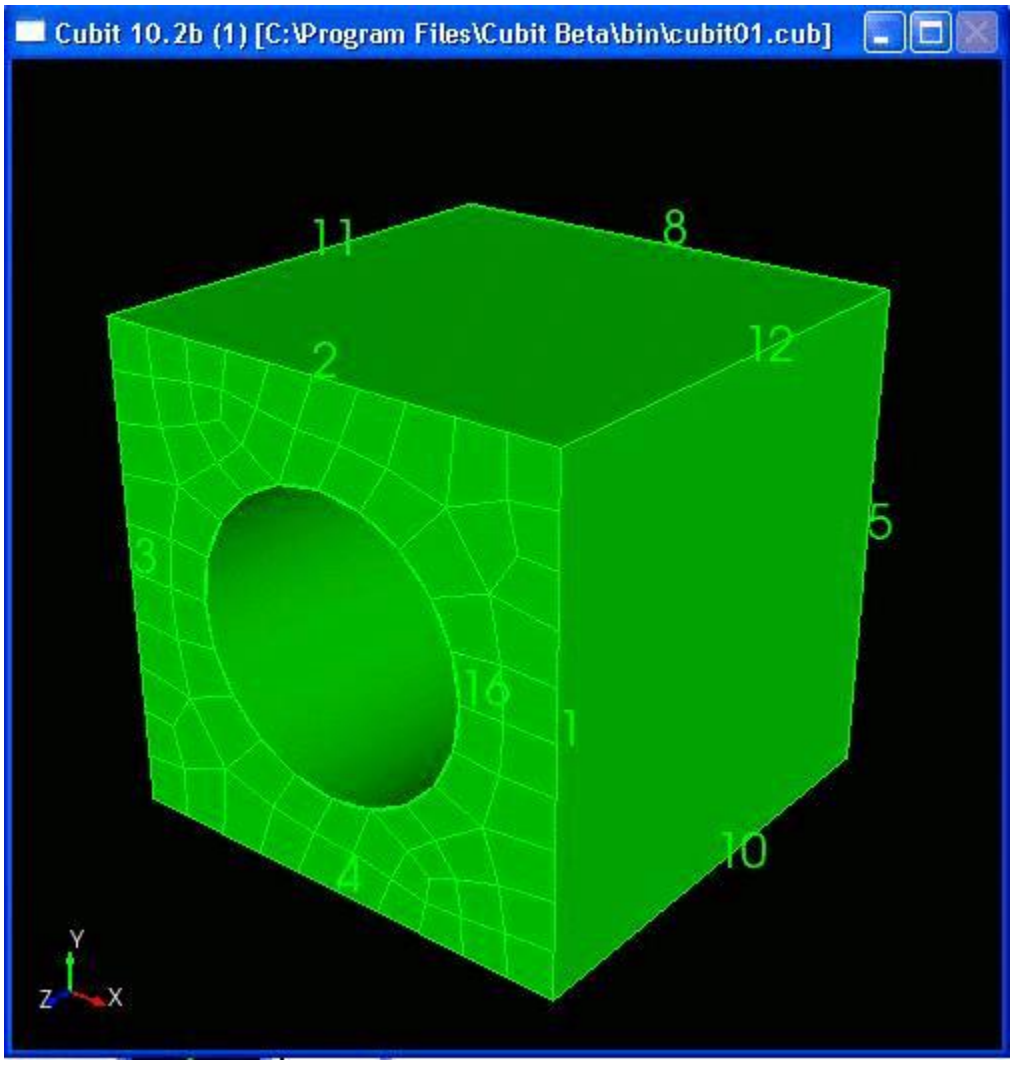
Now that all the necessary intervals have been set, the meshing can proceed. Begin by meshing the front surface (with the hole) using the paving algorithm. This is done in two steps. First, set the scheme for that surface to Pave; then, issue the command to Mesh. Since the surface to be paved is number 11, issue the command:

```
cubit> surface 11 scheme pave
```

With the meshing scheme specified, we proceed to mesh the surface:

```
cubit> mesh surface 11

cubit>display
```

The results are shown below:




**Surface Meshed with Paving**

CL Basic Tutorial Step 8

# *Step 8: Surface Meshing*

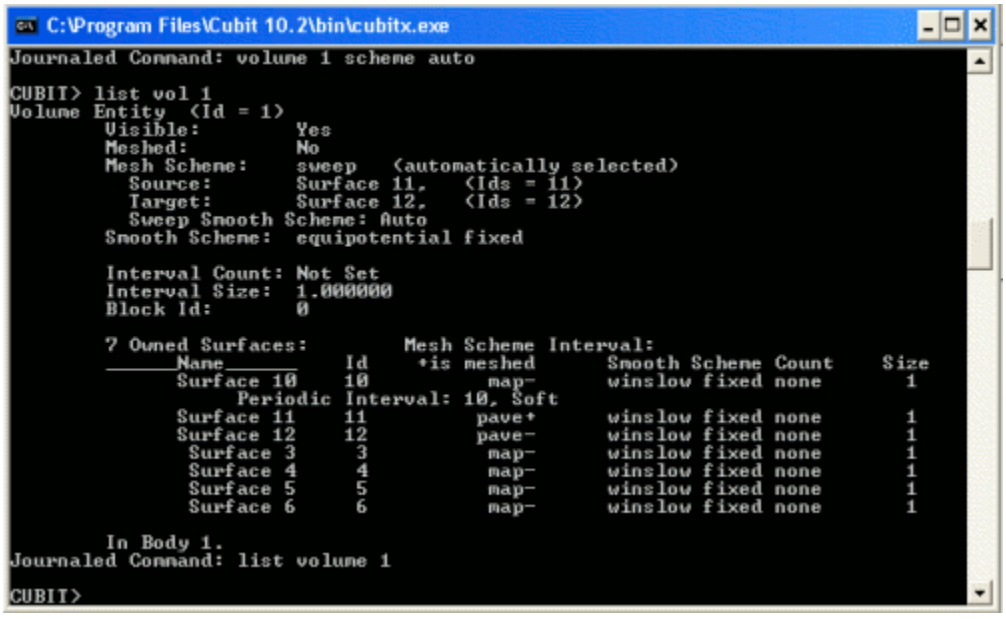
The volume mesh can now be generated. Again, the first step is to specify the type of meshing scheme to be used and the second step is to issue the order to mesh. In certain cases, the scheme can be determined by CUBIT automatically. For sweepable volumes, the automatic scheme detection algorithm also identifies the source and target surfaces of the sweep automatically.
To instruct the code to automatically determine the meshing scheme and in this case the source and target surfaces, enter the command:

```
cubit> volume 1 scheme auto
```

To view the results of auto scheme selection, certain data about the volume can be listed:

```
cubit> list volume 1
```

The results of this command are shown below; note that the scheme, and in this case the source and target surfaces, are reported toward the top of the list output.

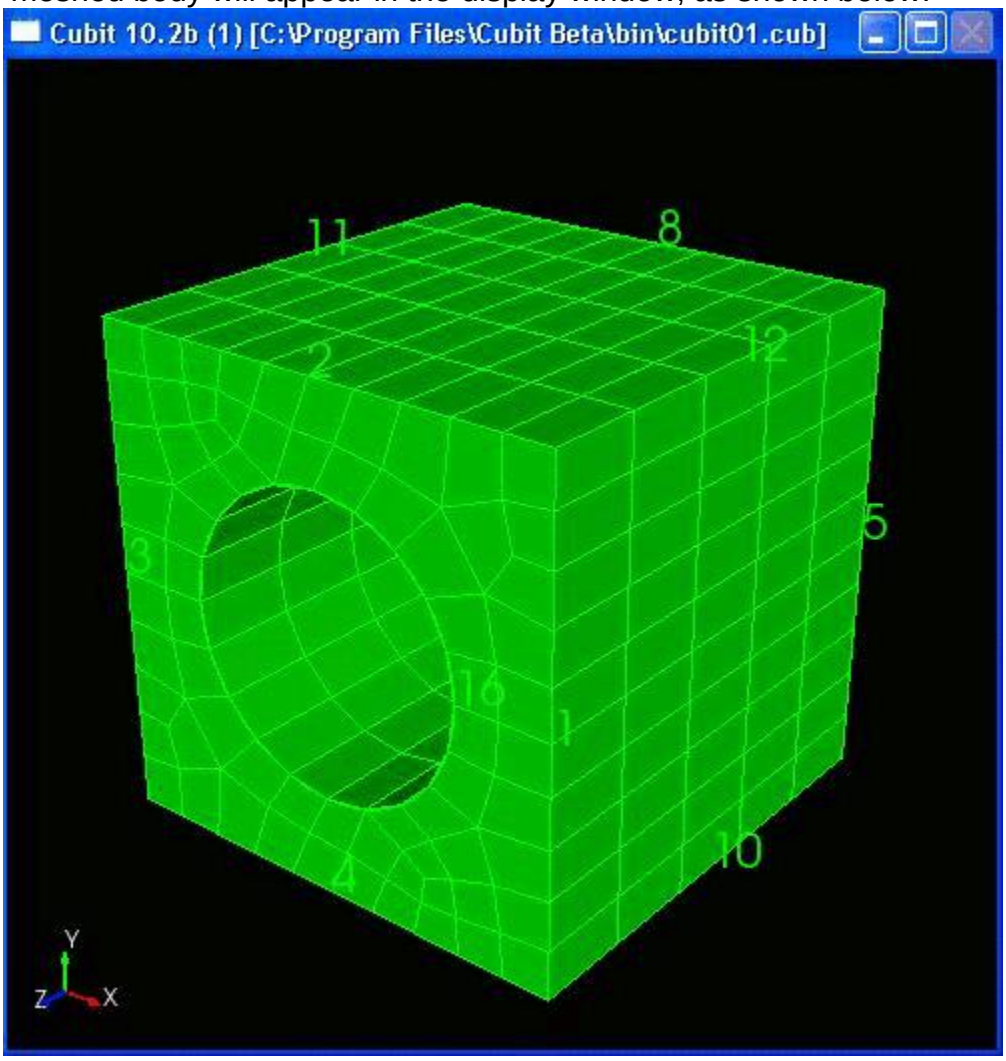



**Output from Listing Volume 1**

With the scheme set, the mesh command may be given:

```
cubit> mesh volume 1
```

The final meshed body will appear in the display window, as shown below:




**View of Volume Mesh**



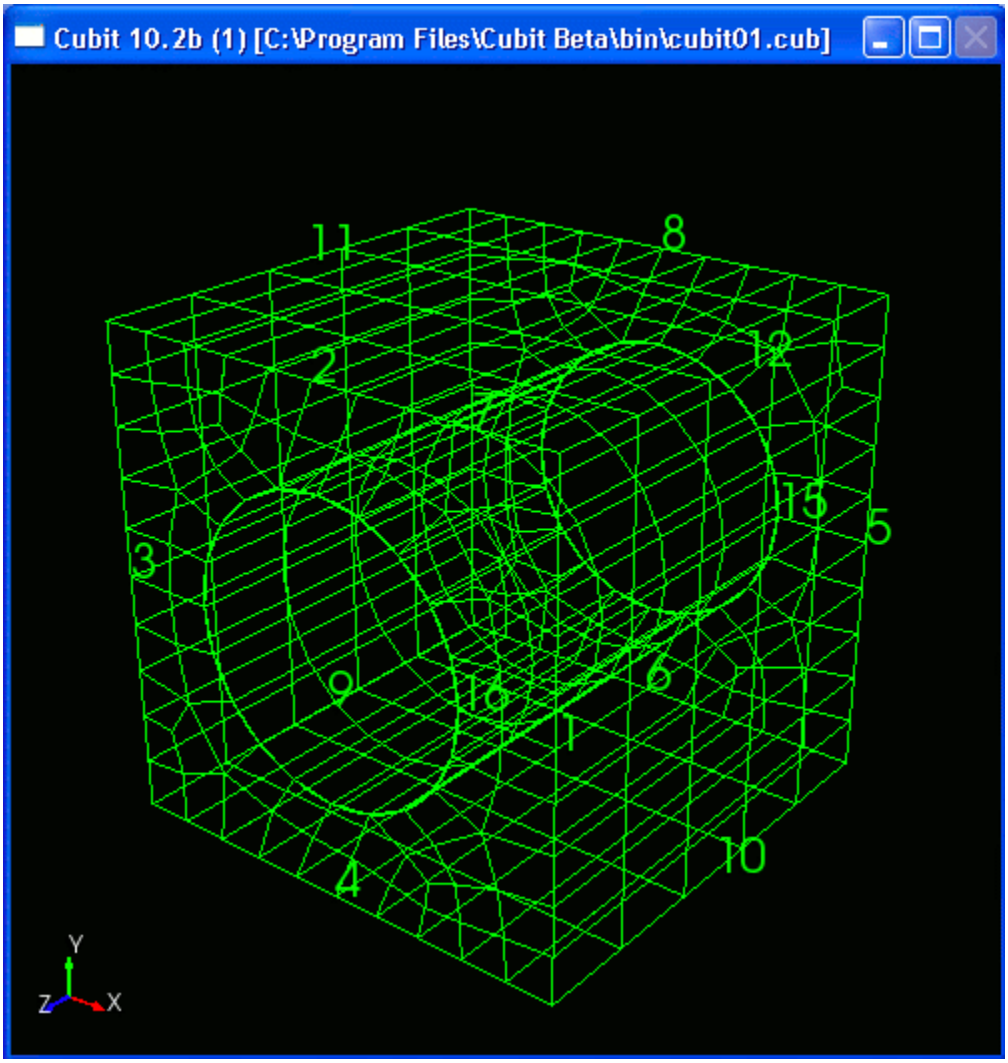
CL Basic Tutorial Step 9

# *Step 9: Inspecting the Model*

The type, quality, and speed of rendering the image can be controlled in CUBIT by using several graphics mode commands, such as Wire Frame, Hidden Line, Transparent and Smooth Shade. For example:

**cubit> graphics mode wireframe**

The wire frame display is illustrated below:

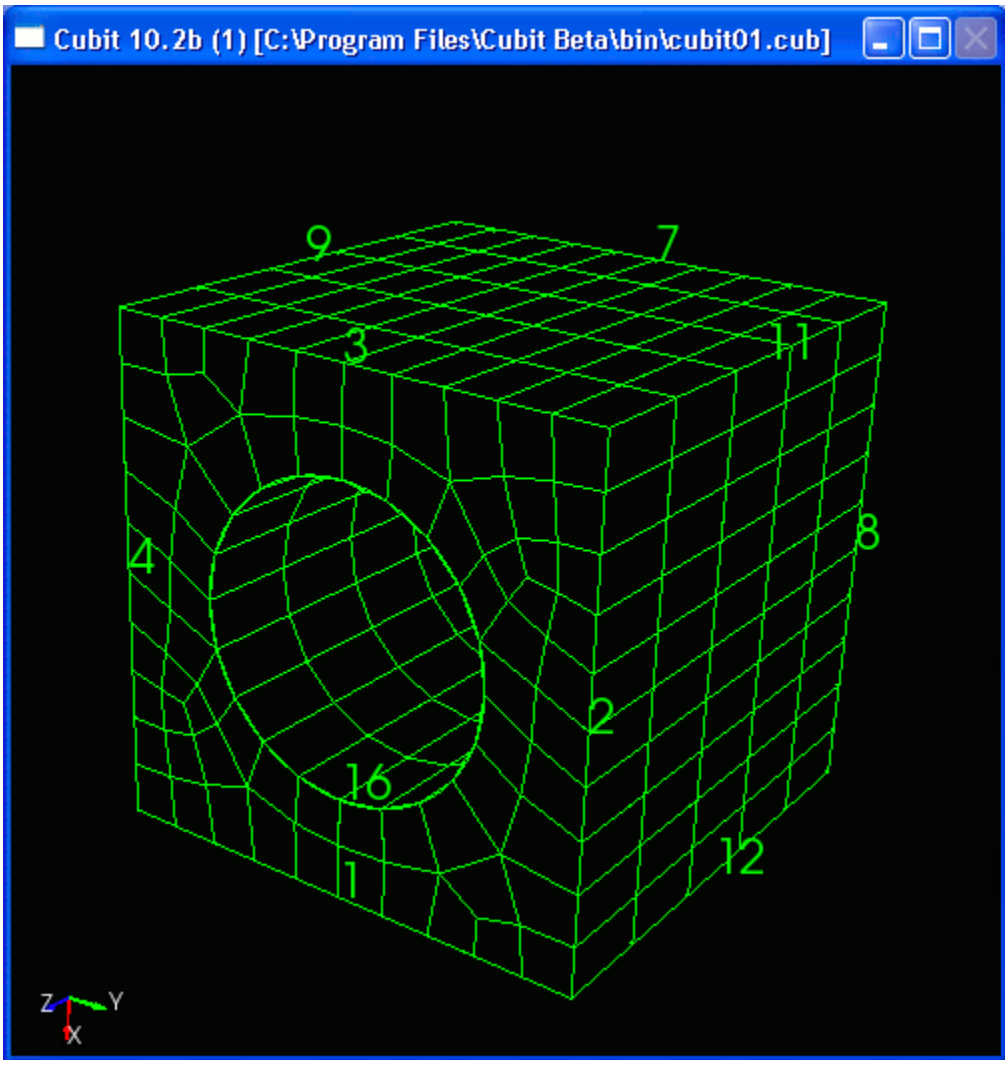



**Wire Frame View of Mesh**

Next, try:

**cubit> graphics mode hiddenline**

The hidden line display is illustrated below:

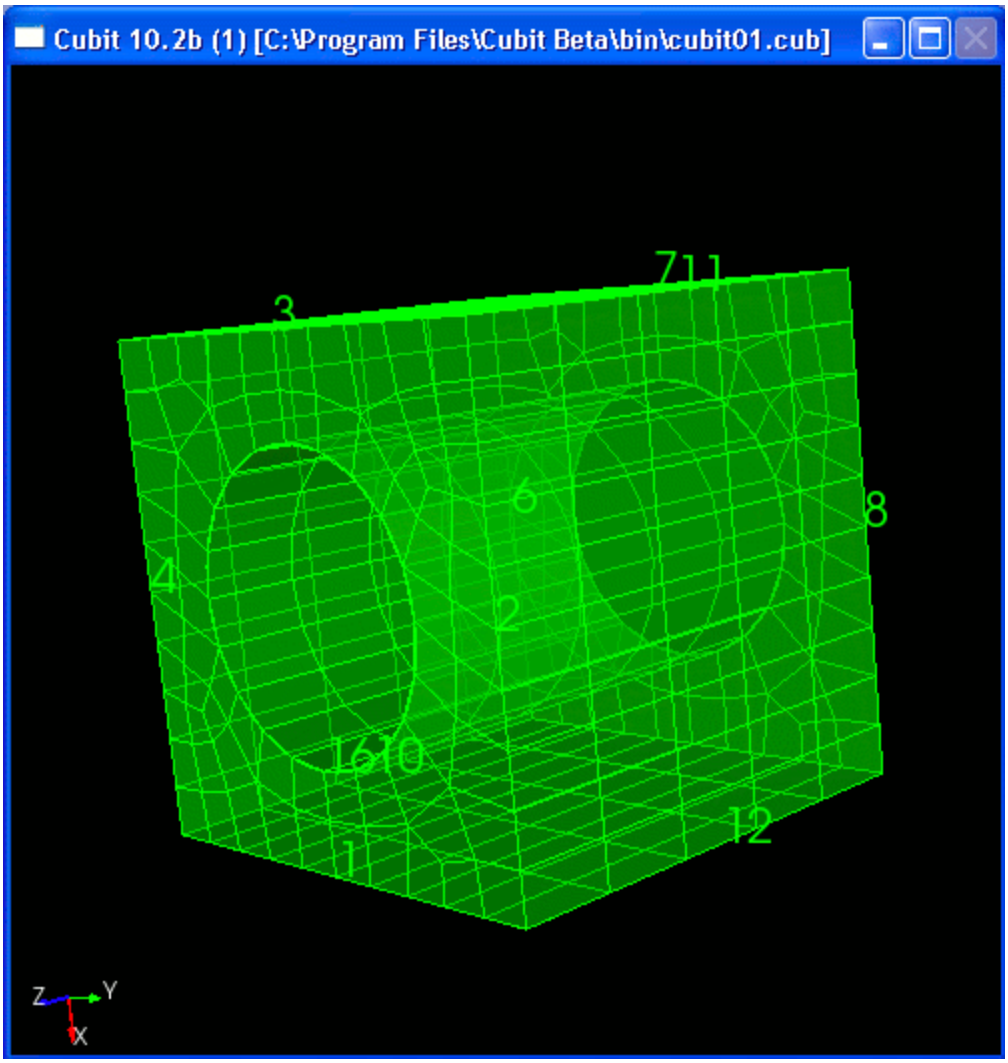



**Hidden Line View of Mesh**

Next, try:

```
cubit> graphics mode transparent
```

The transparent display is shown below.

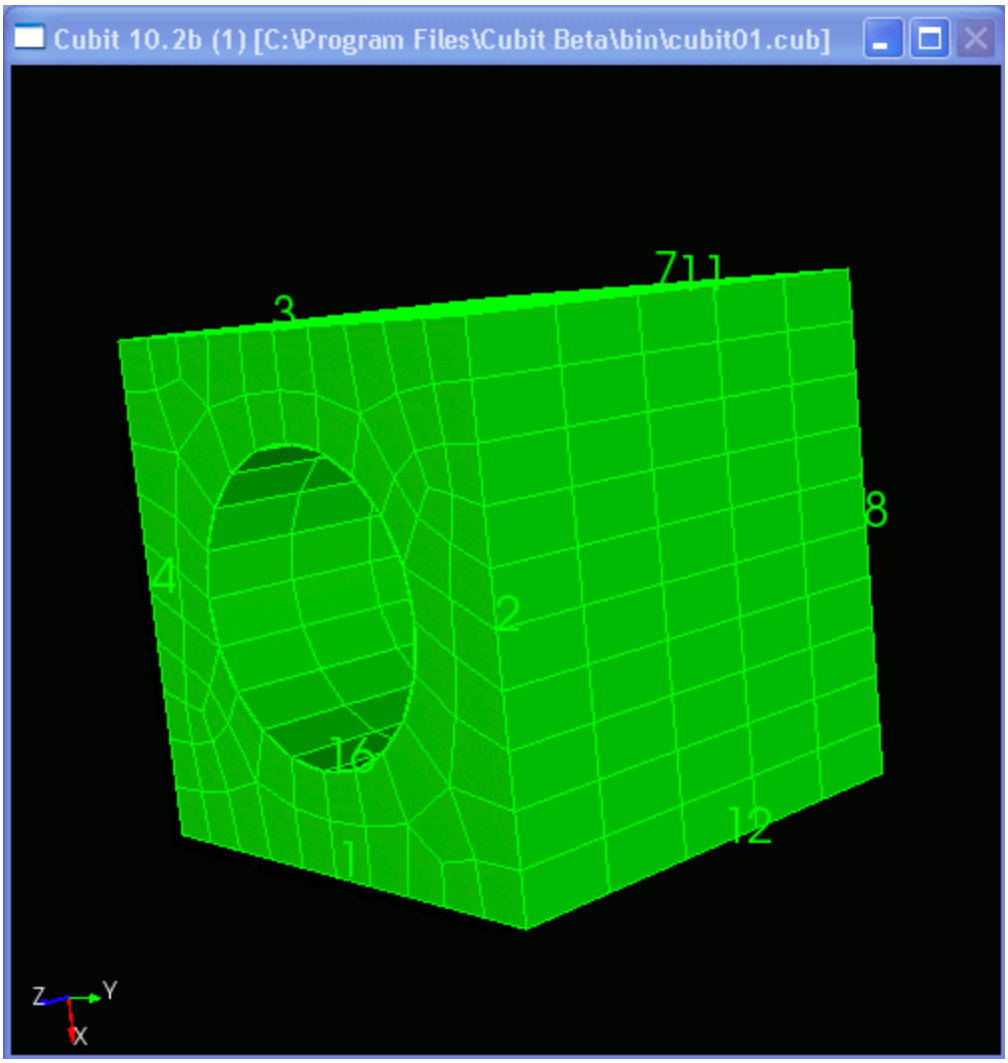



**Transparent View of Mesh**

Next, try:

```
cubit> graphics mode smoothshade
```

The smooth shade display is shown below. For detailed information on the viewing mode options, See Graphics Modes.

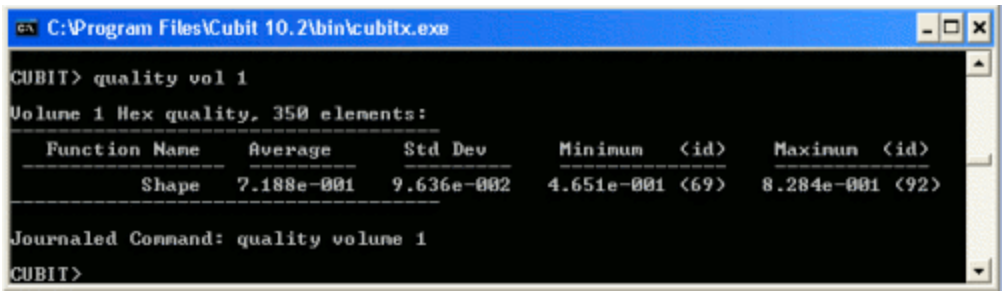



**Smooth Shade View of Mesh**

Although CUBIT automatically computes limited quality metrics after generating a mesh and warns the user about certain cases of bad quality, it is still a good idea to inspect a broader set of quality measures. To do this, enter the command:

```
cubit> quality volume 1
```

The results of the quality output are shown below. For an explanation of quality metrics along with acceptable ranges, see Mesh Quality Assessment. For the purposes of this tutorial, you can assume the quality metrics shown below are in an acceptable range.




**Quality Table from Volume 1's Hex Mesh**



CL Basic Tutorial Step 10

# *Step 10: Defining Boundary Conditions*

Let us assume that we need to define one material type for the entire mesh, and a single node-based boundary condition on all surfaces. This is accomplished by identifying an Element Block and a Nodeset, respectively; the id numbers assigned to these entities are assigned by the user, usually by some convention meaningful to the analysis to be done. The element block and nodeset are identified using the commands:

```
cubit> block 100 volume 1

cubit> nodeset 100 surface all in volume 1
```



CL Basic Tutorial Step 11

# *Step 11: Exporting the Mesh*

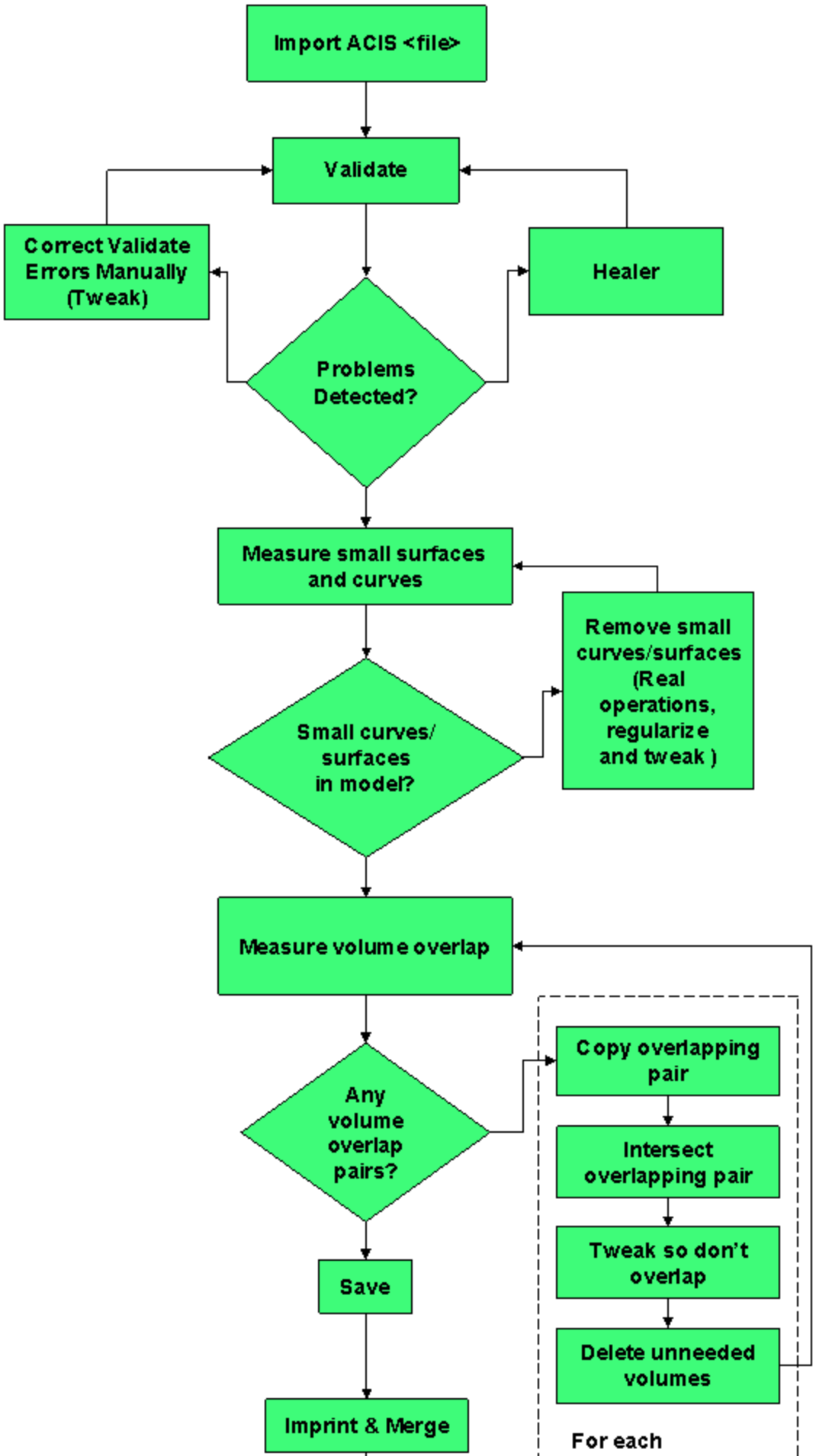
Finally, the mesh needs to be written to an ExodusII file. This is easily done:

```
cubit> export genesis `brick_with_hole.g'
```

The filename and extension are arbitrary and, like the block and nodeset numbers, are usually named according to a convention meaningful to the analysis.



# Geometry Cleanup Process Flow


Import ACIS <file>
Validate
Correct Validate Errors Manually (Tweak)
Healer
Problems Detected?
Measure small surfaces and curves
Remove small curves/surfaces (Real operations, regularize and tweak )
Small curves/ surfaces in model?
Measure volume overlap
Copy overlapping pair
Any volume overlap pairs?
Intersect overlapping pair
Tweak so don't overlap
Save
Delete unneeded volumes
Imprint & Merge
For each

# ITEM

## Immersive Topology Environment for Meshing (ITEM)

The Cubit Geometry and Meshing Toolkit team at Sandia has taken on the ambitious task of reducing the time for simulation by specifically addressing the bottlenecks in the mesh generation process. It is not unusual for the meshing process to take upwards of three-quarters of the entire simulation time. With its many tools developed for a wide range of application areas, it takes time to gain enough proficiency in Cubit to quickly generate a mesh from a complex geometry. As a result, the Immersive Topology Environment for Meshing (ITEM) was developed. ITEM is a user-interactive meshing tool that guides the user through a typical mesh generation process.

With the ultimate goal of reducing the time to generate a mesh for simulation, ITEM has been developed within the Cubit Geometry and Meshing Toolkit to take advantage of its extensive tool suite. Built on top of these tools it attempts to improve the user experience by accomplishing three main tasks:

1. Guiding the user through the workflow
2. Providing the user with smart options
3. Automating geometry and meshing tasks

### Guiding the user through the workflow.

In software of any complexity where usage may be occasional or infrequent, the overhead of learning the new tool to a point of proficiency may be daunting. Given a solid model that may have been designed for manufacturing purposes, the analysts may be faced with generating a mesh. They may not be working with Cubit on a daily basis, but would like to take advantage of the powerful tools provided by the software.

To address this, ITEM provides a wizard-like environment that steps the user through the geometry and meshing process. For someone unfamiliar with the software, it provides an interactive, step-by-step set of tools for accomplishing the major tasks in the process. For those more familiar with the tools, it serves as a reminder of the major tasks, but is flexible enough to accommodate a more iterative approach, allowing them to jump between major tasks easily. Currently restricting the workflow to models requiring three-dimensional, solid elements, ITEM uses the following steps:

1. **Define the Geometric Model:** Import a CAD model or create geometry within the Cubit environment.
2. **Set up the model:** Define basic information such as element shape, volumes to be meshed and element sizes or budgets.
3. **Clean up the geometry:** Detect common issues and simplify geometric features on the CAD model.
4. **Meshing:** Perform operations to make the model meshable, such as imprint/merge, scheme selection, decomposition and performing the meshing.
5. **Validate the Mesh:** Check element quality and perform mesh improvement operations

6. **Apply boundary conditions regions:** Define regions where boundary conditions may be applied using nodeset, sideset and block definitions.
7. **Export the mesh:** Define a target analysis code format and export the mesh.

## Providing the user with smart options.

Solid models used for analysis may have a huge variety of different characteristics that may prevent them from being easily meshed. Questions such as, What are the problems associated with my model? What are the current roadblocks to generating a mesh on this model? and What should I do to resolve the problems, are constantly being asked by the analysts. Without an extensive knowledge of the tools and algorithms, it may be difficult to answer these questions effectively.

ITEM addresses this issue by providing smart options to the user. Based on the current state of the model, it will automatically run diagnostics and determine potential solutions that the user may consider. For example, where unwanted small features may exist in the model, ITEM will direct the user to these features and provide a range of geometric solutions to the problem. Scrolling through the solutions provides a preview of the expected result. The user can then select the solution that seems most appropriate and execute the solution to change or simplify the geometry. This diagnostic-solution approach is the basis for the ITEM design and is the common mode of user interaction while in this environment. This contrasts with the more traditional hunt-and-guess approach of providing the user with an array of buttons and icons that they may choose from and guessing what may result. ITEM, on the other hand, serves in effect, as an expert providing guidance to the user as they proceed through the geometry and meshing process.

## Automating geometry and meshing tasks.

With all of the advanced research and development that has gone into the meshing and geometry problem, a push-button solution for any arbitrary solid model may seem like the ideal objective of any meshing tool. Although for many cases, this would be the best solution, for others it may not even be desirable. A push-button solution assumes a certain amount of trust in the geometric reasoning the software chooses to provide. This may be more trust than an occasional user who is tasked with a high consequence simulation may be willing to give. Even if the user is willing to accept full automation, in many cases, the geometric complexity of the model may be beyond the capability of current algorithms to adequately resolve.

On the other hand, once the user is familiar with the characteristics of the solutions that the software provides, they may not be concerned with examining and intervening on every detail of the model creation process. Instead, in the interest of increasing efficiency, they may want the fastest solution possible. Providing the option for the user to automate as much of the geometry and meshing process as possible is another important aspect of ITEM.

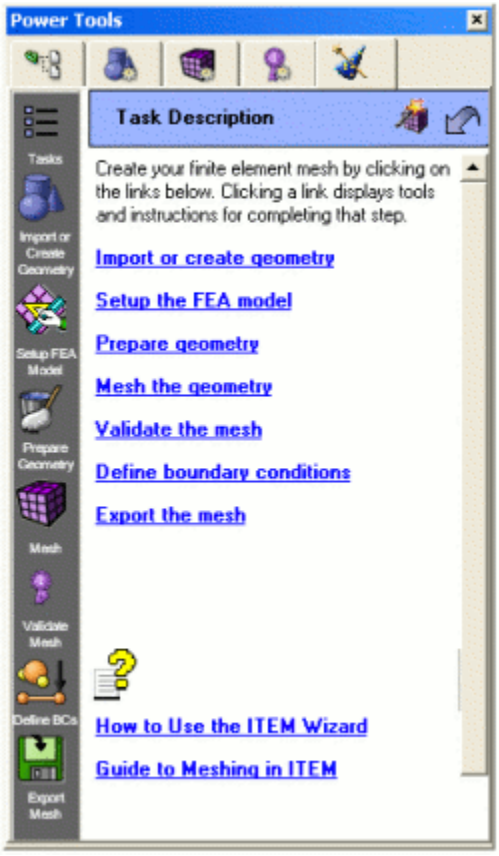
For various characteristic geometric problems that are encountered in a solid model, ITEM can determine from the potential geometric solutions, which of them may be most applicable and apply that solution without any user intervention. For many configurations of geometry, a completely automated solution may be available. For others, only a portion of the process may be able to be automated. Where an adequate solution cannot be determined automatically, the smart options described above are available to help guide

the user. As new advances in geometric reasoning and advanced meshing algorithms are developed, ITEM will incorporate these into the solutions for automation.
It should be clear that ITEM is not intended to be a fully automated system for meshing solid models. Instead it is intended to be a flexible environment that will guide the user through the model generation process by offering solution alternatives and providing automation should the user choose. The remainder of this document is organized according to the basic workflow used in ITEM. The objective is to describe the general problems that may be encountered in developing an analysis model and how ITEM and Cubit may be used to address the problems. In developing this environment, many new innovative tools were invented and developed to help support this new approach to mesh and model generation.

# How to Use the ITEM Wizard

## The ITEM Workflow

The Immersive Topology Environment for Meshing (ITEM) is a wizard-like environment that guides the user through the mesh generation process from geometry definition to export. ITEM was designed to provide a step-by-step set of tools to help new users generate a mesh with very little previous knowledge of the CUBIT program. But ITEM is also flexible enough to accomodate advanced users who want to use a more iterative approach, or who just want to use ITEM for a specific tool or panel.
The main ITEM task page is shown below. To access this page, click on the "wizard hat" icon from the Power Tools window.



## Main ITEM Task Panel

The main item tasks are shown both in the text window, and also along the sidebar. The icons in the sidebar are available from any of the ITEM panels. It is acceptable to jump to different tasks during the process, although beginning users may just want to follow the steps in order. To get to the main task page, click on the Task icon on the sidebar during any step in the process.

Many meshing tasks require an iterative approach to the mesh generation process. For your convenience, if you do click on one of the task buttons from a different panel, it will take you to the last visited panel in that section. For example, if you are on the mesh generation page, and you click on the prepare geometry section, it will take you to the last page you visited in the prepare geometry section.

There are two help links at the bottom of the main task page. The first link will open this document which describes the general ITEM process and how to use the panels. This page is only accessible from the main task page. The second link opens the main ITEM documentation which describes each process in the ITEM mesh generation process in detail. This document can be accessed from any of the ITEM panels.

To proceed through the ITEM panels you must either click on a task or click on the "Done" button at the bottom of each page. There is no "Back" button on the ITEM interface. But in most cases, clicking the "Done" button works like a "Back" button.

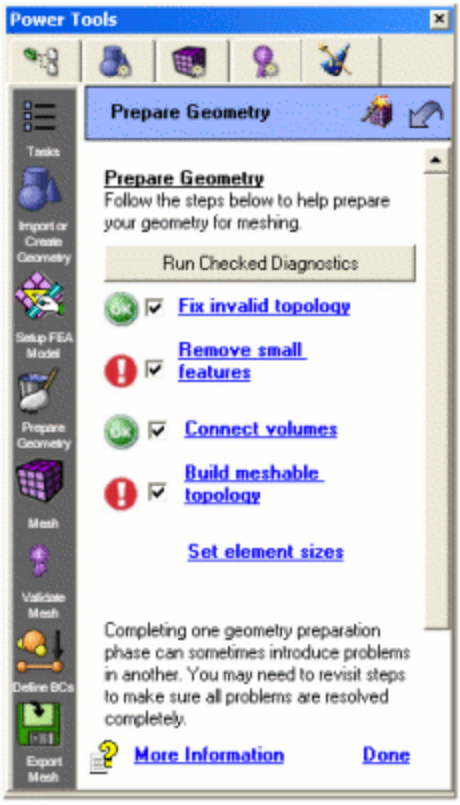
### Using an ITEM Panel

The item panels are designed to be self-explanatory, with plenty of documentation on each page, and access to more help if needed. However, it does help to be generally familiar with the main types of panels.

*Task panels that link to other ITEM panels*

Some ITEM panels provide a list of tasks that link to other ITEM panels. Sometimes the tasks are designed to be completed in sequential or iterative fashion. In that case, you will be returned to the task page after selecting done on each sub-panel where you can select the next task. The Prepare Geometry panel is an example of this case. Each of the tasks with a warning flag should be completed. As you return to this panel, you may need to run the diagnostics again, and possibly even revisit previous task pages.

In other cases, the list of tasks is a presents a list of choices, from which you will only select one option. The Import Geometry Page shown below is such an example. It gives a list of different geometry import/creation options and you just select one of the alternatives.




**Prepare Geometry ITEM Panel**




**Import Geometry ITEM Panel**

*Task Panels that Link to Control Panels*

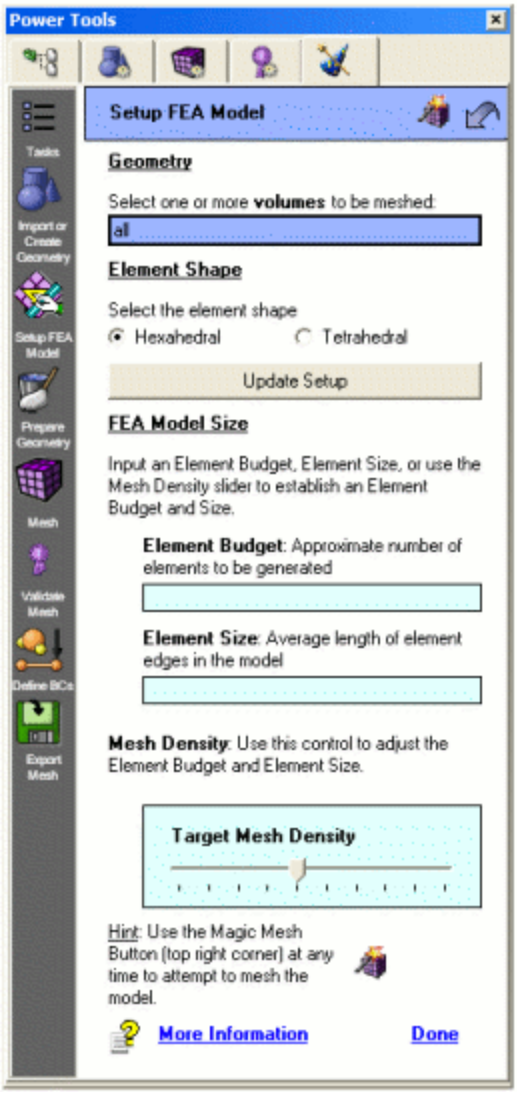
A few of the ITEM task panels will provide links to existing control panel topics. Clicking on a link from one of these panels will NOT open a new panel, but will open the corresponding control panel. The Define Boundary Conditions page is an example of this type of panel.




**Define Boundary Conditions Panel**

*Set-up Panels*

A set-up panel is used to provide input or set-up options for your model. The most prominent set-up panel is the Set-up FEA Model page which is used to define mesh budget, element type, and element size. Another set-up page is the Define Metrics page under the Validate Mesh task. This panel is used to define quality metrics for your model. These panels provide useful information for the diagnostics used in other panels.




**Setup FEA Model Panel**

*Diagnostic Panels*

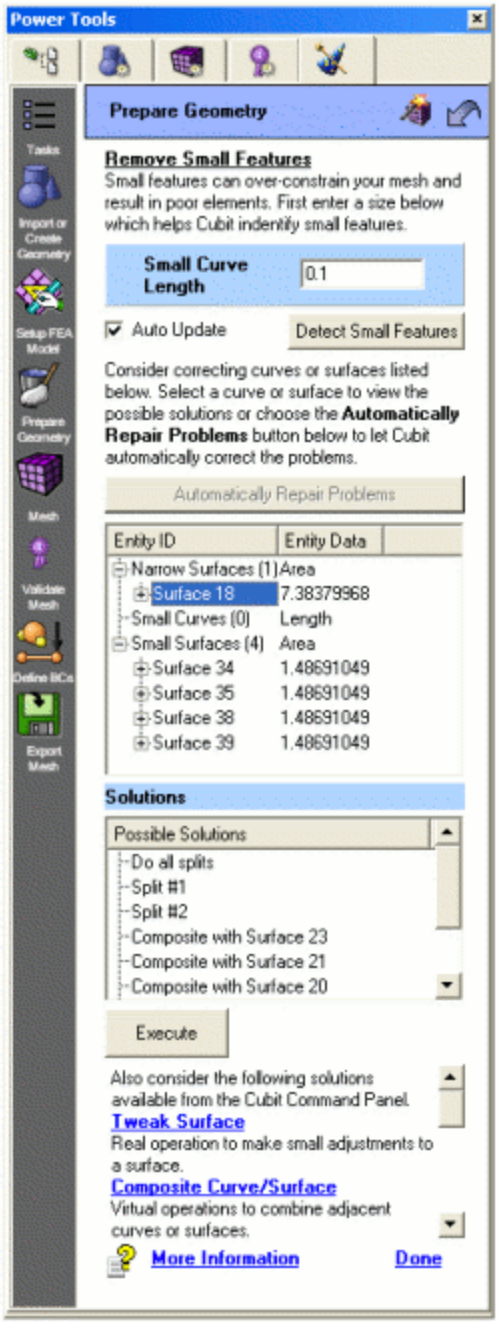
The most useful type of ITEM panel is the diagnostic panel. These panels each focus on a specific diagnostic such as invalid topology, small features, blend surfaces, overlapping surfaces, or meshability. Most of theses panels contain some or all of the following features.

- **Diagnostic Button** - Clicking on this button will run a series of tests on the model.
- **Output Window** - Displays the results of the diagnostics and lists entities with problems. Includes a right-click menu with visualization and other options.
- **Automatically Repair Button** - Tries to solve the problems automatically.
- **Solution Window** - Presents a list of specific solutions based on the entity you select in the output window. This window also contains several right-click context

menu items for each solution, including a "More Information" button which will open the documentation to information about that specific task. Another useful feature of the solution window is that in most cases clicking on one of the solutions will preview that option in the graphics window.

- **Execute Button** - Executes the solution selected.
- **Additional Options** - Sometimes you won't see your desired solution in the list. Additional solutions with brief descriptions are provided at the bottom of the panel. Clicking on these links will open the corresponding control panel.
- **More Information Link** - Opens a page describing the diagnostics and solutions used for this panel.

The Small Features Panel shows an example diagnostic panel in ITEM.




**Remove Small Features Diagnostic Panel**

## Undo Button

The Undo button allows you to reverse the most recent command. To enable the Undo button, click on the "Enable Undo" option from the Edit menu. The undo button works by saving information about your model after each step. For large or complex models, this can be time consuming, so you may need to disable the undo feature. Additionally, not all commands are enabled for undo. Many of the graphics and meshing commands, and various default settings are not included. Within ITEM, many commands are bundled into

a single button click. Clicking undo will attempt to reverse all of the executed commands. See the command line window for the results of the undo command.

## Magic Mesh Button

This button, shown at the top of each ITEM panel, provides the user with the opportunity to use Cubit's internal automation algorithms to generate a mesh. In addition to simply issuing a mesh command, it will attempt to execute the following steps.

- **Geometry Cleanup:** Check for small or ill-defined geometry and automatically resolve it
- **Auto-scheme:** Automatically set meshing schemes and select sources and targets for hex meshing
- **Decomposition:** If hex meshing, attempt to decompose the volume to admit a sweep or mapped mesh
- **Force Sweeps:** For almost-sweepable geometry, modify the linking surfaces to force a sweep
- **Imprint/Merge:** For assemblies, imprint adjacent volumes and merge common surfaces
- **Overlap check:** Check for any remaining overlapping volumes and attempt to resolve merge problems
- **Mesh sizing:** For tetrahedral meshing, automatically define a sizing function based on geometry characteristics
- **Interval Matching:** For hex meshing, coordinate the assignment of curve intervals.
- **Sweep grouping:** Determine an appropriate order to mesh volumes to reduce dependencies
- **Mesh:** Perform the mesh operation volume(s)
- **Mesh Quality:** Check mesh quality and locally optimize if necessary

If for any reason, Cubit is unable to complete these steps without further user intervention, the process will stop and the user will directed to continue with the ITEM workflow. For simple geometries, executing the magic mesh button at this phase of the workflow may be all that is necessary to completely define a good quality mesh. For other more complex geometry, considerable user intervention may be required.

The magic mesh button may be executed at any time during the ITEM workflow by selecting the button at the top right corner of the ITEM panel. Once the user has visited the various panels of the ITEM interface to provide user intervention, the automatic execution of the appropriate operations will not longer be attempted.

## Getting Help

There are several ways to get help from within the ITEM interface. Most of these have already been discussed, but they are listed here again for reference:

- **How to Use ITEM** - This document which is available only from the main task page
- **Guide to Meshing in ITEM** - A document which describes the ITEM workflow, and how to use the diagnostics on each page. This is accessible from each page using the More Information links.

- **Individual help topics for specific solutions** - Opens the documentation to help for each specific solution topic. This is accessible from the right-click menu when a command is selected in the solutions window.
- **Documentation included on panels** - Many of the panels contain brief descriptions and explanations to describe the features and tools on that panel.

# Setting up the Finite Element Model

Once the geometry to be meshed has been imported or created, the first step to defining the mesh is to set up the model. Basic parameters that are needed through the rest of the ITEM workflow are defined at this stage. Subsequent diagnostics and workflow may change based on how the model is initially set up.

**Element Shape**

Either a hexahedral or tetrahedral element shape may be selected. The meshing algorithm used to mesh the volumes will change based on this setting. Specific element characteristics such as the order of the element (i.e. TET10, HEX20) may be specified at a later time. The steps that will be displayed in the workflow will change based on the element type that is selected.

**FEA Model Size**

The number of elements or average size of the elements is an important aspect of defining your analysis model. Geometric features that are considerably smaller than the average element size, in most cases should be ignored since the mesh resolution will not be able to adequately capture them. Defining the element size at this point in the workflow permits subsequent diagnostic tests and operations to have a relative measure of what is “small”. More detailed sizing attributes such as biasing and geometry-adaptive sizing may be defined later in the ITEM workflow.

One of three different mechanisms may be used to define the size, element budget, element size and mesh density. Each of these values is dependent on the other. As a result, changing one value will automatically change the other.

- *Element Budget:* This value is an approximate number of elements that should be generated in the entire model. The element budget for hexahedra, Nhex, is related to the element size, esize, by the following relationship:

$$e_{size} = \sqrt[3]{\frac{V_{model}}{N_{hex}}}$$

- Where Vmodel is the geometric volume of the solid model. The element budget for tetrahedra vs. hexahedra is approximately 1:7. That is, for an equivalent edge length, a tetrahedral mesh will contain roughly seven times as many elements as a hexahedral mesh.
- *Element Size:* Element budget and mesh density are indirect methods for setting the element size, esize. This value can also be set explicitly. It represents the approximate average edge length of elements in the model. This size will determine the relative definition of small for subsequent diagnostic tests and will be used to set the mesh size the meshing algorithms will use.
- *Mesh Density:* The mesh density is represented by an integer between 1 and 10, where 1 is the finest resolution and 10 is the coarsest. It is a heuristic measure of

how fine of a mesh will be generated and permits the user to indirectly set an element size without explicitly defining a real value. In most cases, the mesh density, md is related to the element size, esize by the following heuristic relationship:

$$e_{size} = \sqrt[3]{V_{max}}\,(0.03 + 0.00045 m_d{}^{3.1})$$

Where Vmax is the of the geometric volume of the largest volume in the solid model. Changing the target mesh density will display a preview of the approximate nodal spacing on the curves of the model in the graphics window.

# Defining the Geometric Model

Various methods may be used to define a geometric model. In most cases, a solid model is created in a commercial CAD tool such as Pro/Engineer or Solidworks. It can also be generated natively within Cubit using geometry commands. One of the most time consuming tasks in developing an analysis model is in dealing with geometric anomalies. Carefully considering how the model is constructed and what format the model will be defined in can eliminate many potential problems downstream in the model creation workflow. The following describes the various solutions for defining geometry within Cubit along with their pros and cons:

- Geometry Formats
- Creating Your Own Geometry
- Scripting
- CUB Files

**Geometry Formats**

Cubit can use one of three different commercial geometry representations, ACIS (.sat, .sab), Pro/E (.g) or Catia (.cat). It may also use a facetted format (MBG) that is developed in-house at Sandia. When a model of any of these formats is imported, Cubit uses the appropriate third party geometry kernel to directly manage and evaluate the geometry. Since the geometry is considered “native” when any of these formats is used, no translation step is required.

Since commercial solid modelers do not necessarily agree on formats and representations, using a translation process to convert a non-native format to a native format, can introduce errors in the geometry. While this in itself may not be a show-stopper, it can frequently add hours to an otherwise simple process while the user is forced to clean up dirty geometry. Neutral formats such as STEP and IGES are common in the CAE industry. They can often be an ideal solution for representing the analysis solid model. In Cubit, when importing a neutral format, it is automatically translated to the ACIS format. The user should be careful however in selecting these formats as commercial solid modeling engines frequently interpret standard specifications for these formats in different ways sometimes resulting in unusual results. Wherever possible a native format should be used.

Native geometry kernels provide the most accurate way for transferring data between solid-model based applications. Since these geometry kernels must be licensed and incorporated into the Cubit distribution separately, one drawback is the additional licensing and cost for maintaining these kernels. Cubit is currently able to provide licenses for ACIS and Pro/E kernels for government and academic use. Additional licensing arrangements may be required for Catia or for any commercial use.

**Creating your own geometry**

Cubit offers a wide variety of tools for creating geometry natively. The advantage to this is the ability to control the geometry creation process without the need for another CAD tool. Although Cubit is not designed to be a CAD tool it does provide many tools for both bottom-up and primitive creation.

Bottom-up creation refers to the process of building geometry from its basic components starting with vertices, curves, surfaces and then volumes. This process can be somewhat tedious, but is often useful for generating auxiliary geometry once a CAD model has been imported.

Primitive creation refers to the various operations for generating geometric primitives such as bricks, spheres, cylinders and cones. Once defined, operations for repositioning the objects and performing Boolean operations between them may be used. Relatively complex models may be generated using this approach.

**Scripting**

One advantage to generating your own geometry within Cubit is the ability to parameterize the construction of the model. Cubit utilizes a rich command language that can be stored as a script or journal file. Parameters representing dimensions of objects may be defined in the script and conveniently adjusted to update the geometry representation. For more ambitious users, Cubit also has the ability to interpret python scripts, allowing a high degree of customization that can employ the full capability of the python scripting language.

It should be noted that when using Cubit, commands are automatically echoed to an external temporary journal file on disk and to the history window. Observing these commands is a good way to become familiar with Cubit's internal command language. Copying and pasting selected commands to a text editor is an ideal method for building a parameterized journal file. Journal files may be built up and played back to reproduce the entire process of building an analysis model.

**CUB Files**

A CUB file is Cubit's database file. You may want to think of it as a snap-shot of the current state of the model. While journal files record the process for creating the model, a CUB file stores only the end state. It can include both geometry in its native format and any mesh information as well as attributes and boundary condition information. Restoring a CUB file will write over any existing data you currently have defined.

# Generating a Mesh in ITEM

The mesh generation panel in ITEM is different from the other panels in Cubit. Meshing errors can arise from a number of different problems. Many of these problems are caused from improper geometry preparation/cleanup. Other problems can be caused from improper interval settings, or meshing schemes. Instead of suggesting specific operations as it does on other panels, the meshing panel in ITEM will suggest several possible

solutions based on the error message output. Each of these solutions may require significant user input, and may require you to revisit previous ITEM panels or Control panels. To open the appropriate Control panel, you can right click on the solution and select "Show Command Panel". For convenience, these general solutions are described here, including which ITEM panels and which Control panels they refer to. References to help topics are also included.

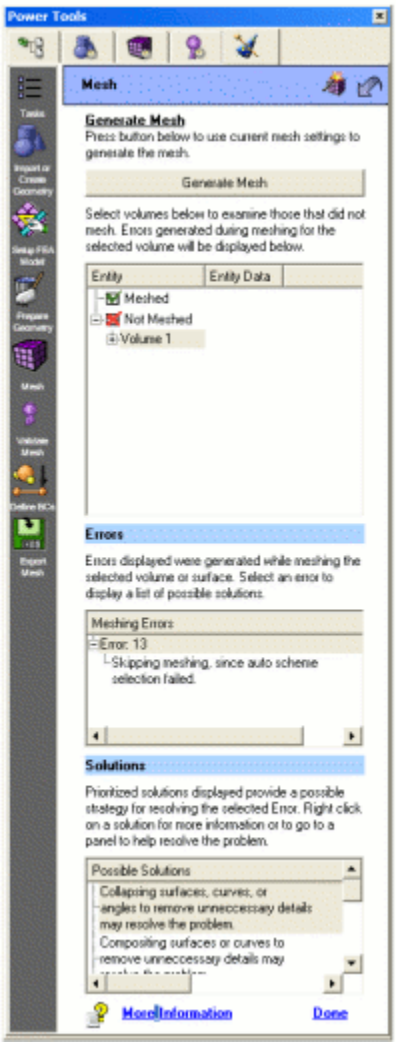



Figure 1. ITEM Mesh Panel

## ITEM Meshing Suggestions

1. ***The volume is not decomposed enough. It may need to be webcut.***

   **Diagnostic:** This solution message appears when auto scheme selection fails. There are many reasons that auto scheme selection may have failed. Check to make sure that your volume is broken up into meshable parts. For sweepable volumes, this means that each volume should only have one target surface.
   **Action:** Right-clicking on this solution and selecting the "Show command panel" option will open the webcutting commands on the control panel. Alternatively, you can also return to the ITEM decomposition panel for more webcutting suggestions.

**Help Topics:**
Geometry Decomposition explains diagnostics and solutions on the ITEM decomposition panel
Decomposition Tutorial has several webcutting tips and examples.
Web Cutting Documentation contains all of the syntax for webcutting commands in Cubit.

2. ***Meshing schemes may need to be manually set.***

   **Diagnostic:** This solution message appears when auto scheme selection fails, interval matching fails, or interval assignments fail. Setting the schemes manually may help resolve some of these issues. It may also help to set source and target surfaces explicitly for swept meshes.
   **Action:** The volume schemes can be set explicitly from the Volume-Mesh control panel. The "Set Source and Target" panel in ITEM can be used to aid in setting explicit source and target surfaces for swept meshes.

   **Help Topics:**
   Recognizing Nearly Sweepable Regions explains how ITEM might be used to recognize nearly sweepable regions.
   Meshing the Geometry has some suggestions for getting difficult geometry to mesh.
   Decomposition Tutorial has several examples where the meshing schemes have to be set manually.
   Meshing Schemes gives an overview of all of the meshing schemes in Cubit.

3. ***The mesh size or number of intervals on a volume may need to be changed.***

   **Diagnostic:** This solution message appears for many reasons: auto scheme selection fails, interval matching fails, interval assignments fail, inconsistent edge-face ratios, odd number of intervals on a paver loop, or connectivity problems. Setting explicit intervals may be necessary
   **Action:** The volume mesh intervals can be set explicitly from the Volume-Interval control panel. The "Set Element Sizes" panel in ITEM can be used to aid in setting explicit sizes and sizing functions for meshes.

   **Help Topics:**
   Interval Assignment has links to different interval assignment methods in Cubit
   Bias, Dualbias describes how to create a biased mesh and Controlling Mesh Quality describes how to propagate a curve bias.
   Decomposition Tutorial has several examples where the meshing intervals are set manually.
   Mesh Adaptivity and Sizing Functions describes how to use sizing functions in Cubit.

4. ***Compositing surfaces or curves to remove unnecessary details may resolve the problem.***

   **Diagnostic:** This solution message appears when auto-scheme selection fails. A model may contain small curves or surfaces that need to be composited with adjacent surfaces. Or it may just contain more detail than is needed for analysis. Compositing surfaces and curves does not affect the underlying geometry.
   **Action:** The Remove Small Features or Force Sweep Topology panels in ITEM may suggest several possible candidates for compositing. The Surface-Modify-Composite or the Curve-Modify-Composite panels can be used to composite surfaces or curves respectively. These panels are also used to delete virtual geometry from curves or surfaces.

   **Help Topics:**

   Removing Small and Narrow Features describes using ITEM to remove small and narrow features in your model.
   Forced Sweepability describes using ITEM to force sweepability using virtual geometry.
   Composite Curves explains how to composite curves in Cubit.
   Composite Surfaces explains how to composite surfaces in Cubit.
   Decomposition Tutorial Example 7 has an example of using composite curves to improve meshability.
   Power Tools Tutorial has another example of using composite geometry.

5. ***Collapsing surfaces, curves, or angles to remove unnecessary details may resolve the problem.***

   **Diagnostic:** This solution message appears when auto-scheme selection fails. Collapsing a surface involves splitting a surface, and compositing it with adjacent surfaces.
   **Action:** The Remove Small Features panel in ITEM may suggest several possible candidates for collapse. The Surface-Modify-Collapse, Curve-Modify-Collapse, or Vertex-Modify-Collapse Angle panels can also be used to collapse surfaces, curves, or angles respectively.

   **Help Topics:**
   Removing Small and Narrow Features describes using ITEM to remove small and narrow features in your model.
   Collapse Angle explains how to collapse angles in Cubit.
   Collapse Curves explains how to collapse curves in Cubit.
   Collapse Surfaces explains how to collapse surfaces in Cubit.

6. ***Removing unnecessary surfaces or curves to simplify geometry may improve the chances that a volume will mesh.***

**Diagnostic:** This solution message appears when auto-scheme selection fails. Removing unnecessary surfaces may improve meshability.
**Action:** The Remove Small Features panel in ITEM may suggest several possible candidates for removal. The Surface-Modify-Tweak panel, Surface-Modify-Remove panel, Curve-Modify-Tweak or the Volume-Modify-Remove Slivers panels are also used to remove unnecessary features in a model.

**Help Topics:**
Removing Small and Narrow Features describes using ITEM to remove small and narrow features in your model.
Removing Geometric Features describes the syntax for removing unneeded surfaces and vertices, including sliver surfaces.
Tweaking Geometry contains the syntax for tweaking surfaces, curves, and vertices.
Power Tools Tutorial has an example of using the tweak surface command to simplify a model.

7. ***Smoothing the mesh may improve the mesh quality.***

   **Diagnostic:** This solution message appears when mesh generation creates poor quality elements, particularly if it creates inverted or "negative Jacobian" elements. In some cases, smoothing a mesh may get rid of these bad elements.
   **Action:** Depending on the geometry type, the smoothing panel can be accessed from the Control panel under Volume-Smooth or Surface-Smooth panels. It is also helpful to use the Validate Mesh page in ITEM for assessing quality metrics.

   **Help Topics:**
   Mesh Smoothing describes the different smoothing schemes in Cubit and how to use them.
   Mesh Validation describes how to use quality metrics in ITEM and gives suggestions on smoothing schemes to try.
   Mesh Quality Assessment describes the different quality metrics in Cubit and how to use them.

8. ***Deleting the mesh on an entity in order to further decompose or modify it may be necessary.***

   **Diagnostic:** This solution message appears when mesh generation creates a poor quality mesh, due to negative Jacobians, inconsistent edge-face ratios, connectivity problems, or any other invalid mesh configuration. Mesh generation can be a very iterative process. It is sometimes necessary to delete a mesh and try different schemes, sizes, or even just change the meshing order. Sometimes you must further decompose or modify your geometry to get it to mesh.
   **Action:** To delete a mesh, you can select it in the graphics window and choose Delete Mesh from the right-click context menu. You can also delete a mesh from any of the Mesh-Entity-Delete panels on the Control Panel.

**Help Topics:**
Mesh Deletion describes command line syntax for deleting a mesh.

9. ***Changing vertex types may make the surface or volume meshable.***

   **Diagnostic:** This solution message appears when mesh generation fails to assign valid vertex types on mapped or submapped surfaces.
   **Action:**To change the vertex type on a surface, select the Surface-Mesh-Submap-Advanced or Surface-Mesh-Map-Advanced panels. From here you can assign and view vertex types.

   **Help Topics:**
   Surface Vertex Types describes how to change the vertex types on a geometry.

# Validating the Mesh in ITEM

Advancements in the mesh generation algorithms have significantly reduced the amount of quality problems seen in the initially generated mesh. Further, ITEM generally relies on the most robust meshing algorithms available in CUBIT, specifically sweeping for hexahedral mesh generation (Scott,05) and the MeshGems (George,91) meshing software (See http://www.distene.com). However, some problems can still exist, and therefore ITEM has integrated quality diagnostics and solution options.
***Diagnostics:*** After the mesh has been generated, the user may choose to perform element quality checks. ITEM utilizes the Verdict (Stimpson,07) library where a large number of mesh quality metrics have been defined and available as a modular library. If no user preference is specified, ITEM uses the Scaled Jacobian distortion metric to determine element quality. This check will warn users of any elements that are below a default or user-specified threshold, allowing various visualization options for displaying element quality.
***Solutions:*** If the current element quality is unacceptable, ITEM will present several possible mesh improvement solutions. The most promising solutions are provided through ITEM's interface to two smoothers: mean ratio optimization and Laplacian smoothing. These are provided as part of the Mesquite (Brewer,03) mesh quality improvement tool built within CUBIT. The user has the option of performing these improvements on the entire mesh, subsets of the mesh defined by the element quality groups, or on individual elements. The Laplacian smoothing scheme allows the users to smooth just the interior nodes or to simultaneously smooth both the interior and boundary nodes in an attempt to improve surface element quality.

# clean up

## Recognizing Nearly Sweepable Regions

The purpose of geometry operations such as decomposition is to transform an unmeshable region into one or more meshable regions. However, even the operations suggested by the decomposition tool can degenerate into guesswork if they are not

performed with a specific purpose in mind. Without a geometric goal to work toward, it can be difficult to recognize whether a particular operation will be useful.
Incorporated within the proposed ITEM environment are algorithms that are able to detect geometry that is nearly sweepable, but which are not fully sweepable due to some geometric feature or due to incompatible constraints between adjacent sections of geometry. By presenting potential sweeping configurations to the user, ITEM provides suggested goals to work towards, enabling the user to make informed decisions while preparing geometry for meshing.
Unlike the decomposition solutions presented in the previous section, the purpose of recognizing nearly sweepable regions is to show potential alternative source-target pairs for sweeping even when the autoscheme tool does not recognize the topology as strictly sweepable. When combined with the decomposition solutions and the forced sweepability capability described later, it provides the user with an additional powerful strategy for building a hexahedral mesh topology.
**Diagnostics:** In recognizing nearly sweepable regions, the diagnostic tool employed is once again the autoscheme tool described in [White, 00]. Volumes that do not meet the criteria defined for mapping or sweeping are presented to the user. The user may then select from these volume for which potential source-target pairs are computed.
**Solutions:** The current algorithm for determining possible sweep configurations is an extension of the autoscheme algorithm described in [White, 00]. Instead of rejecting a configuration which does not meet the required sweeping constraints, the sweep suggestion algorithm ignores certain sweeping roadblocks until it has identified a nearly feasible sweeping configuration. The suggestions are presented graphically, as seen in Figure 1. In most cases, the source-target pairs presented by the sweep suggestion algorithm are not yet feasible for sweeping given the current topology. The user may use this information for further decomposition or to apply solutions identified by the forced sweepability capability described next. The sweep suggest algorithm also provides the user with alternative feasible sweep direction solutions as shown in Figure 1. This is particularly useful when dealing with interconnected volumes where sweep directions are dependent on neighboring volumes.

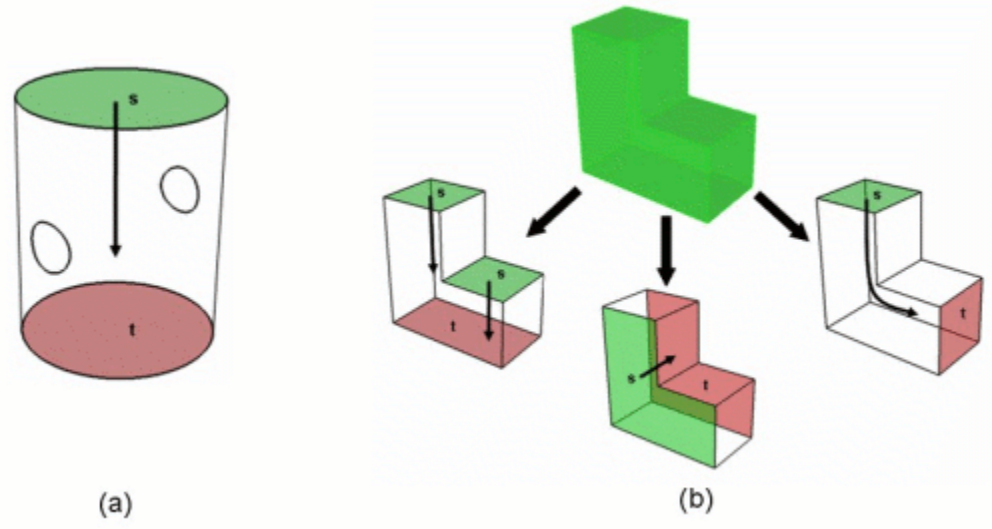


**Figure 1. (a) ITEM displays the source and target of a geometry that is nearly sweepable. The region is not currently sweepable due to circular imprints on the side of the cylinder. (b) Alternative feasible sweep directions are also computed.**

## Blend Surfaces

Blend surfaces are common in solid model meshing problems. A blend surface, also known as a fillet or chamfer, is problematic for sweeping algorithms which have trouble assigning vertex types on blend surfaces. While blend surfaces present a challenge for meshing applications, there are many tools within ITEM to help guide the user through possible solutions.

**Diagnostic:** Blend surfaces are detected by looping over the curves on a surface and examining the angles, surface normals, and curvature of curves and adjacent surfaces.

**Solutions:** The current solution to blend surfaces is to remove the surface and attempt to extend adjacent surfaces to fill in the gap. An example of blend surfaces that have been removed is shown below. This is useful for models which can be simplified without losing important topology.

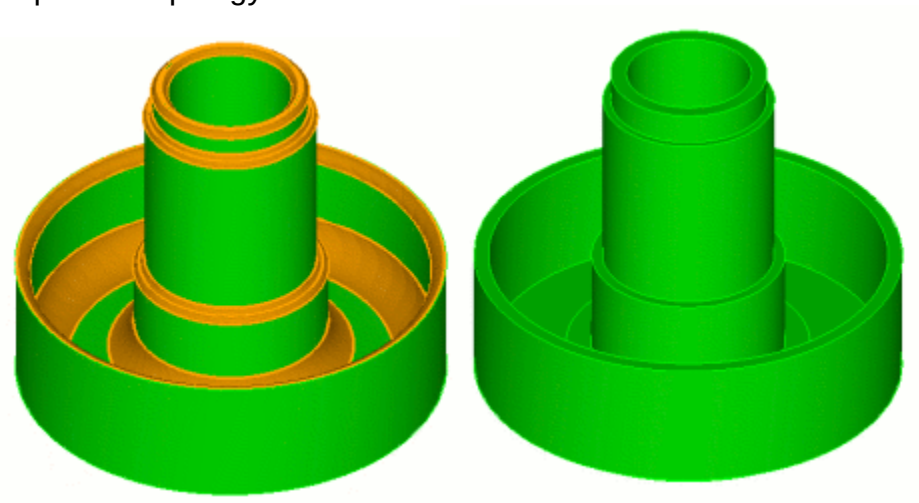


Figure 1. A volume which has been simplified by removing blend surfaces.

## Clean Up the Geometry

Meshing packages have the challenge of dealing with a host of geometry problems. Many of these problems can be generalized as file translation issues. Typically, the geometry used in a meshing package has not been created there but in one of many CAD packages. Exporting these files out of CAD and into a neutral file format (IGES, STEP, SAT) accepted by the meshing software can introduce misrepresentations in the geometry. If the CAD and meshing packages do not support the same file formats, a second translation may be necessary, possibly introducing even more problems.

Another complication caused by file translation is that of tolerances. Some CAD packages see two points as coincident if they are within 1e-3 units, while others use 1e-6. If the meshing software's tolerance is finer than the CAD package's, this disparity in tolerance can cause subsequent geometry modification operations in the meshing package to inadvertently create sliver features, which tend to be difficult and tedious to deal with. This tolerance problem also causes misalignment issues between adjacent volumes of

assemblies, hindering the sharing of coincident geometry in order to produce a conformal mesh.
Modeling errors caused by the user in the CAD package is another problem that the meshing package has to correct. In the CAD package, the user may not create the geometry correctly, causing some parts to overlap, or introduce small gaps between parts that should touch. Many times these problems are detected in the meshing package at a point when it is not feasible to simply go back into the CAD system and fix the problem, so the meshing package must be capable of correcting it.
Several approaches for addressing the geometry cleanup problem have been proposed in the literature, but they typically provide operations that are automatically applied to the geometry once one or more topology problems have been identified. While very effective in many cases, they generally lack the ability for the user to have control over the resolution of these CAD issues while still maintaining the option for automation. The ITEM environment provides tools to both diagnose these common issues and to provide a list of solutions from which the user may select that will correct the problems.
For the purposes of mesh generation, features in a solid model that should be carefully considered and addressed prior to meshing generally fit in one of four categories:

- Bad geometry representation As a result of translation errors between CAD representations, errors or differences in the way the geometry is interpreted may occur. Depending on the severity of the problem, sometimes a mesh can be generated even with a less-than perfect geometric representation, however, in most cases, these should be resolved before meshing.
- Small details in the model In some cases there exist small details in the geometry that, if meshed, would result in very small elements and a potentially huge element budget. Small curves and surfaces can sometimes result from details in the design solid model that may not be necessary for analysis or may even be a result of careless construction of the CAD model. In either case, it is important to remove or modify these features before meshing.
- Compatible topology for meshing scheme Several meshing algorithms, such as the structured, mapping and sweeping techniques require a specific configuration of vertices, curves and surfaces in order to operate. Operations to decompose the geometry into a meshable topology are often needed. Other unstructured techniques like paving, and tetrahedral meshing do not require decomposition.
- Conformal topology for assemblies Assemblies of parts are often required to have a conformal mesh across their interface. (i.e. Shared nodes at a common boundary). The operations imprint and merge are often required to connect parts together so that when meshed, the representation will be a single continuous mesh.

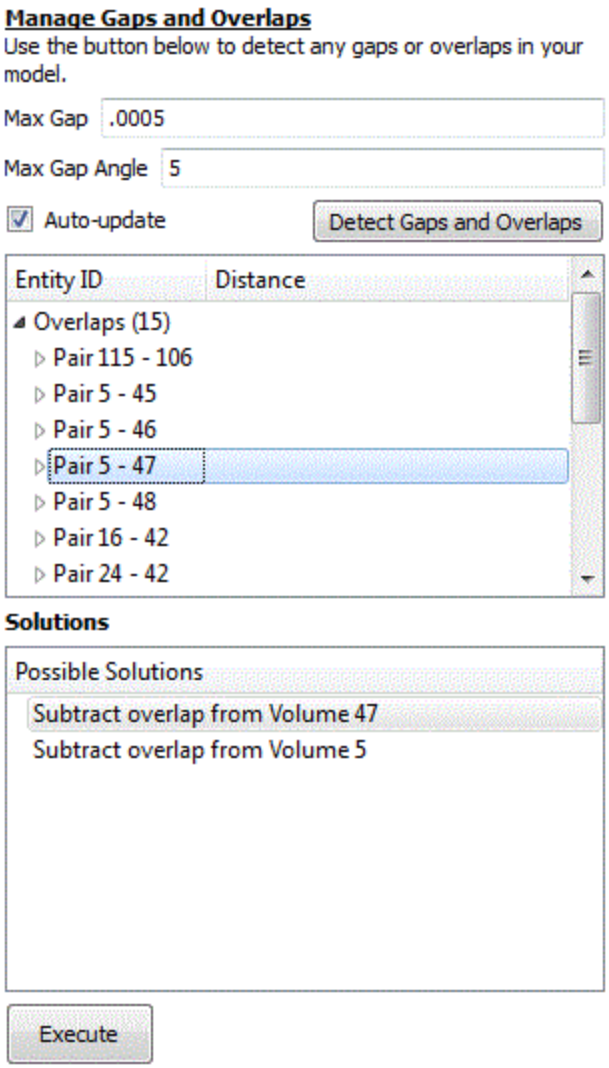
Being able to recognize when a problem exists and what operations to apply to resolve issues in each of the four categories described above, is indeed an art-form and requires significant experience to become proficient. ITEM will not take the place of an experienced user, but it is intended to offer the user help along the way by detecting potential problems and suggesting solutions they might consider.

## Resolving Problems with Conformal Assemblies

Where more than a single geometric volume is to be modeled, a variety of common problems may arise that must be resolved prior to mesh generation. These are typically a result of misaligned volumes defined in the CAD package or problems arising from the imprint and merge operations in the meshing package. ITEM addresses some of the same problems by allowing the option for user interaction as well as full automation using the CAD geometry representation. The proposed environment utilizes two main diagnostics to detect potential problems: the misalignment check, and the overlapping surfaces check. Associated with both of these are solutions that are specific to the entity and from which the user may preview and select to resolve the problem.

*Resolving Misaligned Volumes with Manage Gaps/Overlaps Tool*

The Manage Gaps/Overlaps Tool within the geometry cleanup area of ITEM allows the user to quickly search an assembly for gaps and overlaps between assembly components. The search criteria for gaps is a tolerance specified by the user and defines the maximum gap between components to look for. A gap angle can also be specified which specifies how "parallel" two entities must be to be considered in the gap check. The overlap check simply asks Cubit to see if any of the volumes are overlapping and doesn't require a tolerance from the user. The results are displayed in a list of pairs of volumes. The user can right-click on these pairs and tell Cubit to draw the pair. A useful graphical depiction of the gap or overlap will be displayed. When the user clicks on a pair in the list a set of solutions for fixing the gap or overlap will also be displayed below in a separate list. The user can select a solution and click the "Execute" button to execute it. The gap solutions are either a surface "tweak" operation and the overlap solution can be either a tweak operation or a Boolean operation to remove the overlap. This tool provides a powerful way to quickly work through the assembly and fix gaps and overlaps.




*Resolving Misaligned Volumes with Near Coincident Vertex Checks*

The near coincident vertex check or misalignment check is used to diagnose possible misalignments between adjacent volumes. This diagnostic is performed prior to the imprint operation in order to reduce the sliver surfaces and other anomalies which can occur as a result of imprinting misaligned volumes. With this diagnostic, the distance between pairs of vertices on different volumes are measured and flagged when they are just beyond the merge tolerance. The merge tolerance, **T**, is the maximum distance at which the geometry kernel will consider the vertices the same entity. A secondary tolerance, $\mathbf{T_s}$, is defined where $\mathbf{T_s} > \mathbf{T}$ which is used for determining which pairs of vertices may also be considered for merging. Pairs of vertices whose distance, **d** is $\mathbf{T} < \mathbf{d} > \mathbf{T_s}$ are presented to the user, indicating areas in the model that may need to be realigned. The misalignment check should also detect small distances between vertices and curves on adjacent volumes.

When pairs of vertices are found that are slightly out of tolerance, the current solution is to move one of the surfaces containing one vertex of the pair to another surface containing the other vertex in the pair. Moving or extending a surface is known as tweaking.

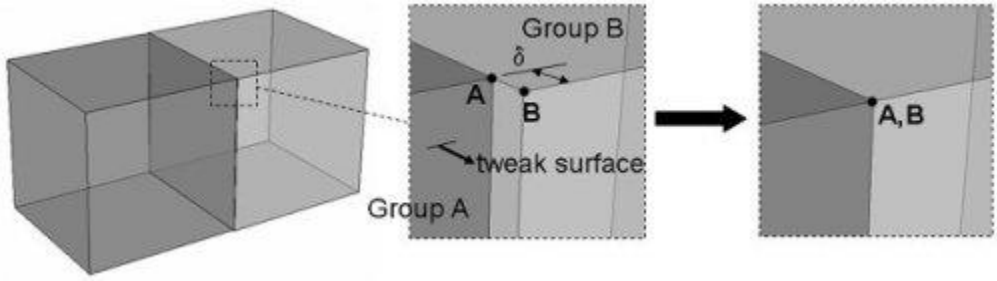



**Figure 1. Example of a solution generated to correct misaligned volumes using the tweak operator**

The result of this procedure will be a list of possible solutions that will be presented to the users. They can then graphically preview the solutions and select the one that is most appropriate to correct the problem.

*Correcting Merge Problems*

The merge operation is usually performed immediately following imprinting and is also subject to occasional tolerance problems. In spite of correcting misalignments in the volume, the geometry kernel may still miss merging surfaces that may occupy the same space on adjacent volumes. If volumes in an assembly are not correctly merged, the subsequent meshes generated on the volumes will not be conformal. As a result, it is vital that all merging issues be resolved prior to meshing. The ITEM environment provides a diagnostic and several solutions for addressing these issues.

An overlapping surface check is performed to diagnose the failed sharing of topology between adjacent volumes. In contrast to the misalignment check, the check for overlapping surfaces is performed after the imprinting and merging operations. The overlapping surface check will measure the distance between surfaces on neighboring volumes to ensure that they are greater than the merge tolerance apart. Pairs of surfaces that failed to merge and that are closer than the merge tolerance are flagged and displayed to the user as potential problems.

A test for nonmanifold curves and vertices is also performed after imprinting and merging to find geometry that was not merged correctly. The test for nonmanifold curves is looking for curves that are merged, but do not share merged surfaces. Similarly, the test for nonmanifold vertices is looking for merged vertices that do not share any merged curves. Another test for floating volumes is performed to identify volumes that are not attached to any other entities.

If imprinting and merging has been performed and a subsequent overlapping surface check finds overlapping surface pairs, the user may be offered three different options for correcting the problem: force merge, tolerant imprint of vertex locations and tolerant imprint of curves.

If the topology for both surfaces in the pair is identical, the force merge operation can generally be utilized. The merge operation will remove one of the surface definitions in

order to share a common surface between two adjacent volumes. Normally this is done only after topology and geometry have been determined to be identical, however the force merge will bypass the geometry criteria and perform the merge. Figure 2 shows a simple example where the bounding vertices are identical but the surface definitions are slightly different so that the merge operation fails. Force merge in this case would be an ideal choice.

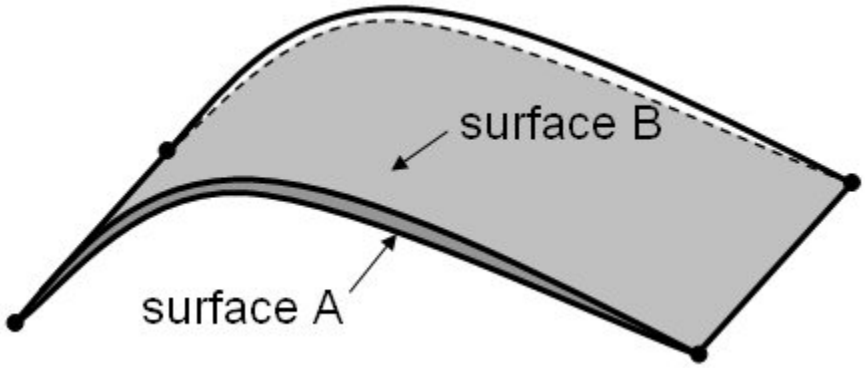



**Figure 2. Example where the merge operation will fail, but force merge will be successful**

The force merge operation is presented as a solution where a pair of overlapping surfaces are detected and if any of the following criteria are satisfied:

- All curves of both surfaces are merged
- All vertices between the two surfaces are merged and all the curves are coincident to within 1% of their length or 0.005, whichever is larger
- All the curves of both surfaces are either merged or overlapping and a vertex of any curve of one surface that will imprint onto any other curve of the other surface cannot be identified
- At least one curve of one surface may be imprinted onto the other and if both surfaces have an equal number of curves and vertices, and the overlapping area between the 2 surfaces is more than 99% of the area of each surface. This situation generally prevents generating sliver surfaces
- At least one vertex of surface B may be imprinted onto surface A, and if both surfaces have equal number of curves and vertices, and the vertex(s) of surface B to imprint onto surface A lies too close to any vertices of surface A
- All the curves of both surfaces are either merged or overlapping and no vertices of any curve of surface A will imprint onto any other curve of surface B

Individual vertices may need to be imprinted in order to accomplish a successful merge. The solution of imprinting a position x,y,z onto surface A or B is presented to the user if the following criteria is met

- Curves between the two surfaces overlap within tolerance, and a vertex of curve A lies within tolerance to curve B and outside tolerance to any vertex of curve B. Tolerance is 0.5% of the length of the smaller of the 2 curves or the merge tolerance (0.0005), whichever is greater.

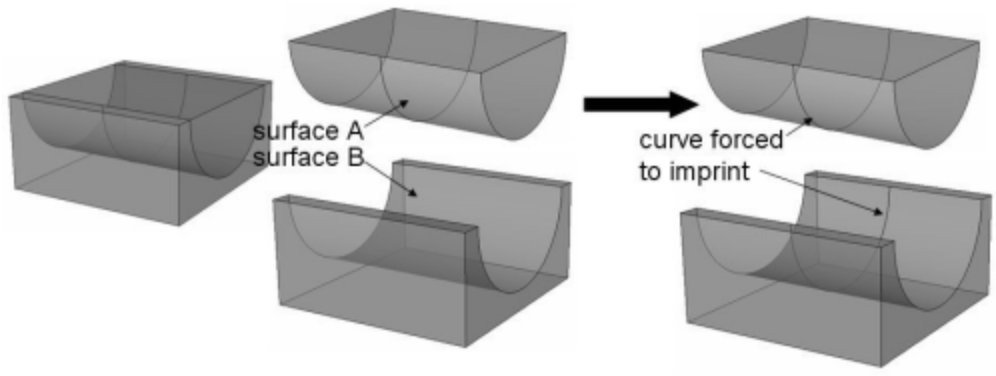



**Figure 3. Curve on surface A was not imprinted on surface B due to tolerance mismatch. Solution is defined to detect and imprint the curve**

In some cases one or more curves may not have been correctly imprinted onto an overlapping surface which may be preventing merging. This may again be the result of a tolerance mismatch in the CAD translation. If this situation is detected a tolerant imprint operation may be performed which will attempt to imprint the curve onto the adjacent volume. Figure 3 shows an example where a curve on surface A is forced to imprint onto surface B using tolerant imprint, because it did not imprint during normal imprinting. The solution of a curve of surface A to be imprinted onto surface B may be presented to the user if all 3 of the following conditions are satisfied:

- There are no positions to imprint onto the owning volume of either surface
- Curve of surface A is not overlapping another curve of surface B
- Curve of surface A passes tests to ensure that it is really ON surface B

## Contact Surfaces

A contact surface is two surfaces which overlap, but are not merged. In a physical sense, this could represent two surfaces which come in contact with each other, as opposed to two surfaces which merely form a partition for meshing purposes. It is easy using the ITEM interface to identify and select contact surfaces in your model. Simply select surfaces in the graphics window and press the "Add" button on the ITEM interface. The contact surfaces will be shown in the window.
To remove a contact surface from the list, right click on the surface and select "Not a Contact Surface" from the context menu to remove that specific surface, or "Remove all contact surfaces" to remove all contact surfaces. Several other visualization tools are also available from the context menu including Zoom, Fly-in, Draw, List, Locate, etc.

## Geometry Decomposition

Automatic decomposition has been researched and tools have been developed which have met with some limited success [Lu,99 , Staten,05]. Automatic decomposition requires complex feature detection and sub-division algorithms. The decomposition problem is at least on the same order of difficulty as the auto-hex meshing problem. Fully automatic methods for quality hexahedral meshing have been under research and

development for many years [Blacker,93 , Folwell,98 , Price,95]. However, a method that can reliably generate hexahedral meshes for arbitrary volumes, without user intervention and that will build meshes of an equivalent quality to mapping and sweeping techniques, has yet to be realized. Although fully automatic techniques continue to progress [Staten,06], the objective of the proposed environment is to reduce the amount of user intervention required while utilizing the tried and true mapping and sweeping techniques as its underlying meshing engine.
Instead of trying to solve the all-hex meshing problem automatically, the ITEM approach to this problem is to maintain user interaction. The ITEM algorithms determine possible decompositions and suggest these to the user. The user can then make the decision as to whether a particular cut is actually useful. This process helps guide new users by demonstrating the types of decompositions that may be useful. It also aids experienced users by reducing the amount of time required to set up decomposition commands.
**Diagnostics:** The current diagnostic for determining whether a volume is mappable or sweepable is based upon the autoscheme tool described in [White,00]. Given a volume, the autoscheme tool will determine if the topology will admit a mapping, sub-mapping or sweeping meshing scheme. For volumes where a scheme cannot be adequately determined, a set of decomposition solutions are generated and presented to the user.
**Solutions:** The current algorithm for determining possible cut locations is based on the algorithm outlined in [Lu,99] and is described here for clarity:

- Find all curves that form a dihedral angle less than an input value (currently 135)
- Build a graph of these curves to determine connectivity
- Find all curves that form closed loops
- For each closed loop:
    - Find the surfaces that bound the closed loop
    - Save the surface
    - Remove the curves in the closed loop from the processing list
- For each remaining curve:
    - Find the open loops that terminates at a boundary
    - For each open loop:
        - Find the surfaces that bound the open loop
        - Save the surfaces
- For each saved surface:
    - Create an extension of the surface
    - Present the extended surface to the user as a possible decomposition location.

This relatively simple algorithm detects many cases that are useful in decomposing a volume. Future work will include determining symmetry, sweep, and cylindrical core decompositions. These additional decomposition options should increase the likelihood of properly decomposing a volume for hexahedral meshing.
Figure 1 shows an example scenario for using this tool. The simple model at the top is analyzed using the above algorithm. This results in several different solutions being offered to the user, three of which are illustrated here. As each of the options is selected, the extended cutting surface is displayed providing rapid feedback to the user as to the

utility of the given option. Note that all solutions may not result in a volume that is closer to being successfully hex-meshed. Instead the system relies on some user understanding of the topology required for sweeping.
Each time a decomposition solution is selected and performed, additional volumes may be added, which will in turn be analyzed by the autoscheme diagnostic tool. This interactive process continues until the volume is successfully decomposed into a set of volumes which are recognized as either mappable or sweepable.

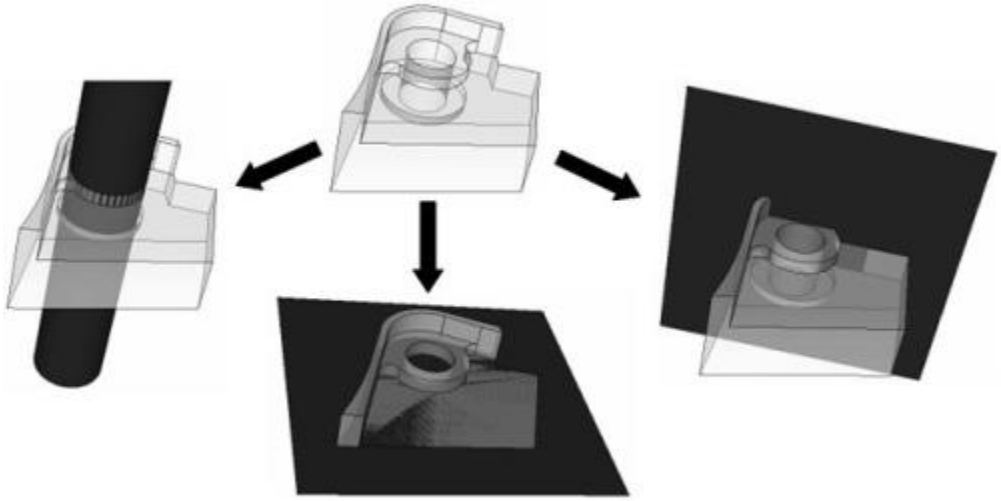


**Figure 1. ITEM decomposition tool shows 3 of the several solutions generated that can be selected to decompose the model for hex meshing**

## Forced Sweepability

In some cases, decomposition alone is not sufficient to provide the necessary topology for sweeping. The forced sweepability capability attempts to force a model to have sweepable topology given a set of source and target surfaces. The source-target pairs may have been identified manually by the user, or defined as one the solutions from the sweep suggestion algorithm described above. All of the surfaces between source and target surfaces are referred to as linking surfaces. Linking surfaces must be mappable or submappable in order for the sweeping algorithm to be successful. There are various topology configurations that will prevent linking surfaces from being mappable or submappable.
**Diagnostics:** The first check that is made is for small curves. Small curves will not necessarily introduce topology that is not mappable or submappable but will often enforce unneeded mesh resolution and will often degrade mesh quality as the mesh size has to transition from small to large. Next, the interior angles of each surface are checked to see if they deviate far from 90 multiples. As the deviation from 90 multiples increases the mapping and submapping algorithms have a harder time classifying corners in the surface. If either of these checks identify potential problems they are flagged and potential solutions are generated.
**Solutions:** If linking surface problems are identified ITEM will analyze the surface and generate potential solutions for resolving the problem. Compositing the problem linking surface with one of its neighbors is a current solution that is provided. ITEM will look at

the neighboring surfaces to decide which combination will be best. When remedying bad interior angles the new interior angles that would result after the composite are calculated in order to choose the composite that would produce the best interior angles. Another criterion that is considered is the dihedral angle between the composite candidates. Dihedral angles close to 180 are desirable. The suggested solutions are prioritized based on these criteria before being presented to the user. Figure 1 shows an example of a model before and after running the forced sweepability solutions. The top and bottom of the cylinder were chosen as the source and target surfaces respectively.

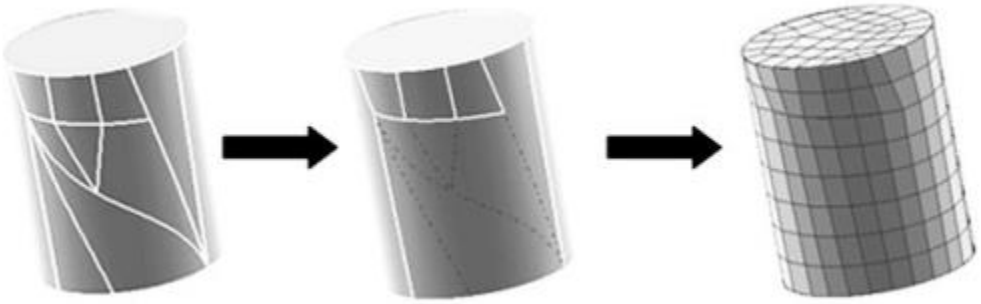


**Figure 1. Non-submappable linking surface topology is composited out to force a sweepable volume topology**

## Bad geometry representation

As a result of translation errors between CAD representations, errors or differences in the way the geometry is interpreted may occur. Depending on the severity of the problem, sometimes a mesh can be generated even with a less-than perfect geometric representation, however, in most cases, these should be resolved before meshing.

*Detecting Invalid Geometry*

In most cases, bad or invalid topology or geometry definition comes from problems which arise in the CAD translation process. CUBIT's main geometry kernel, ACIS is used to represent the model if it has been imported using an IGES or STEP format. Translation to and from these neutral formats is frequently the cause of bad geometry. ITEM will use the geometry validation procedures built into the ACIS kernel to detect if there is any bad geometry and will list the entities that may be causing a problem.

Since the validation procedures are specific to ACIS, models that may have been imported from another native format such as Pro/E will not provide this diagnostic. Although this may seem like a severe limitation, importing native formats rarely have bad geometry, since no translation process is necessary.

It is good practice to always check your model for bad geometry before proceeding to other geometry or meshing operations. In some cases, if a webcut or meshing operation fails, the cause is an invalid geometric definition that has not been adequately healed. Resolving bad geometry problems up front, in most cases is essential to obtaining a mesh. On the other hand, if the location of the bad geometry in the model is such that it will not effect subsequent Boolean or decomposition operations, there may be a chance that completely resolving bad geometry is not necessary. Simply ignoring bad geometry

that cannot be easily repaired with automatic procedures may be a reasonable solution, provided the user is aware of the potential limitations.

*Resolving Invalid Geometry*

To resolve invalid geometry, ITEM uses the heal procedure built into the ACIS geometry kernel. In almost all cases, this is a fully automatic procedure. Simply selecting the automatic repair button will make the appropriate adjustments to the geometry. This can be done one volume at a time by healing the owning volume, or by healing the full model all at once. If healing was successful, No problems detected should be displayed.
If auto repair does not successfully repair the geometry, you may want to try additional options available in Cubit for healing. See the Cubit documentation for a complete description of additional healing options.

## Determining an Appropriate Merge Tolerance

Determining the appropriate merge tolerance for a model can be essential for creating conformal meshes on some models. The merge tolerance is a value that identifies at which distance different entities can be considered the same entity. Many entities will fail to merge because of widespread geometry tolerance or alignment problems that are either too difficult, time-consuming or even impossible to resolve. Specifying a merge tolerance that is larger than these small discrepancies allows the user to account for geometry that is misaligned. But specifying a merge tolerance that is too large can combine features the user wishes to keep, and possibly corrupt the model. The ideal merge tolerance should be smaller than the smallest feature, but larger than the biggest gap or misalignment that cannot be resolved. Since it is not always a simple task to determine either of these features, the ITEM workflow provides a diagnostic tool designed to guide the user to find the small misalignments that may lead to merge problems. It then presents possible solutions to fix these problems, or the ability to change the merge tolerance to ignore them.

*Opening the Merge Tolerance Panel*

To open the merge tolerance tool from the ITEM Wizard, click on Prepare Geometry->Connect Volumes->Imprint and Merge. Then click on the button with three dots next to the **Merge Tolerance** input field.

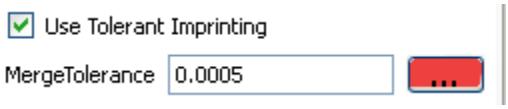



**Figure 1. How to open the merge tolerance panel**

The merge tolerance panel is shown in the following image.

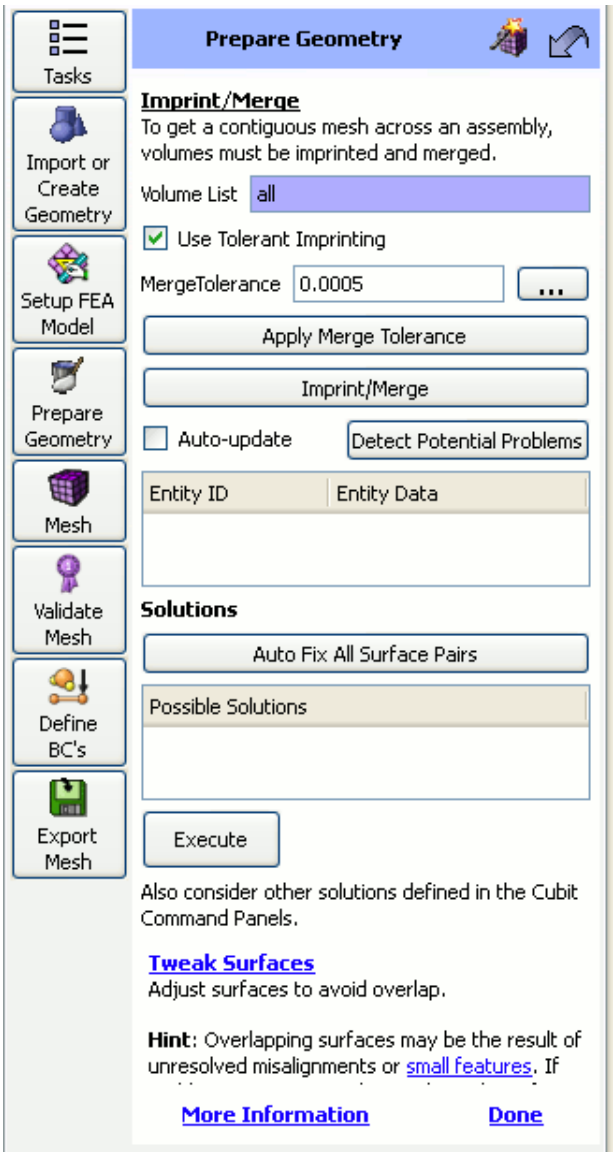



**Figure 2. The Merge Tolerance Diagnostic Panel**

*Estimating Merge Tolerance with Small Feature Size*

Since the merge tolerance must be smaller than the smallest feature in the mesh, the best place to start is by finding the smallest feature and using that value to create an estimate for the merge tolerance. To find the smallest feature, click on the small button with three dots next to the input box for Small Features.
**Note:** The small feature checks will not find misalignments between different volumes- it will only list vertex-vertex pairs and vertex-curve pairs on the same volume. The small feature size is used on the merge tolerance panel to find an initial estimate for the merge tolerance.
After determining the smallest feature size, click on the *Estimate Merge Tolerance* button to come up with a rough estimate for the merge tolerance. It is important to note that this

is only an estimate. After an initial estimate is made, it can be fine tuned using the Fine Tune Merge Tolerance tool.

*Fine Tuning the Merge Tolerance*

In the fine tune merge tolerance area, the user may search for vertex-vertex, vertex-curve, and vertex-surface pairs that are within user-specified ranges. This includes checks between entities on different volumes. This allows the user to determine if the merge tolerance he/she has determined will capture all of the merges he/she intends. The user can check/uncheck which pairs to search for and what range to look in. The results from the search will show up in the window below and the user can select the results, right click on it, and choose *Draw with Volumes* to zoom into that pair of features. For vertex-vertex pairs there may be tweak solutions presented to the user in the list box below for fixing the problems.

*Setting the Merge Tolerance*

The Apply button next to Estimated Merge Tolerance edit field is used to take the estimated merge tolerance and use it to set the merge tolerance in CUBIT by issuing the Merge Tolerance <val> command.

## Building a Sweepable Topology

The hex meshing problem presents a number of additional challenges to the user that tetrahedral meshing does not. Where a good quality tetrahedral mesh can generally be created once small features and imprint/merge problems have been addressed, the hexahedral meshing problem poses additional topology constraints which must be met.
Although progress has been made in automating the hex meshing process, the most robust meshing algorithms still rely on geometric primitives. Mapping [Cook, 82] and sub-mapping [Whiteley, 96] algorithms rely on parametric cubes and sweeping[Knupp, 98; Scott, 05] relies on extrusions. Most real world geometries do not automatically fit into one of these categories so the topology must be changed to match the criteria for one of these meshing schemes. ITEM addresses the hex meshing topology problem through four primary diagnostic and solution mechanisms.

1. Detecting blend surfaces
2. Detecting and suggesting decomposition operations
3. Recognizing nearly sweepable topologies and suggesting source-target pairs
4. Detecting and compositing surfaces to force a sweep topology

## Small details in the model

The small feature removal area of ITEM focuses on identifying and removing small features in the model that will either inhibit meshing or force excessive mesh resolution near the small feature. Small features may result from translating models from one format to another or may be intentional design features. Regardless of the origin small features must often be removed in order to generate a high quality mesh.

ITEM will recognize small features that fall in four classifications:

1. small curves
2. small surfaces
3. narrow surfaces
4. surfaces with narrow regions

These operations may involve either real, virtual or a combination of both types of operations to remove these features. A virtual operation is one in which does not modify the CAD model, but rather modifies an overlay topology on the original CAD model. Real operations, on the other hand directly modify the CAD model. Where real operations are provided by the solid modeling kernel upon which CUBIT is built, virtual operations are provided by CUBIT's CGM (Tautges, 00) module and are implemented independently of the solid modeling kernel. The following describes the diagnostics for finding each of the four classifications of small features and the methods for removing them.

*Small Curves*

***Diagnostic:*** Small curves are found by simply comparing each curve length in the model to a user-specified characteristic small curve size. A default epsilon ($\varepsilon$) is automatically calculated as 10 percent of the user specified mesh size, but can be overridden by the user.
***Solutions:*** ITEM provides three different solutions for eliminating small curves from the model. The first solution uses a virtual operation to composite surfaces. Two surfaces near the small curve can often be composited together to eliminate the small curve as shown in Figure 1(a).
The second solution for eliminating small curves is the collapse curve operation. This operation combines partitioning and compositing of surfaces near the small curve to generate a topology that is similar to pinching the two ends of the curve together into a single point. The partitioning can be done either as a real or virtual operation. Figure 1(b) illustrates the collapse curve operation.
The third solution for eliminating small curves is the remove topology operation. This operation can be thought of as cutting out an area around the small curve and then reconstructing the surfaces and curves in the cut-out region so that the small curves no longer exist. (Clark, 07) provides a detailed description of the remove topology operation. This operation has more impact on the actual geometry of the model because it redefines surfaces and curves in the vicinity of a small curve. The reconstruction of curves and surfaces is done using real operations followed by composites to remove extra topology introduced during the operation. Figure 1(c) shows the results using the remove topology operation.

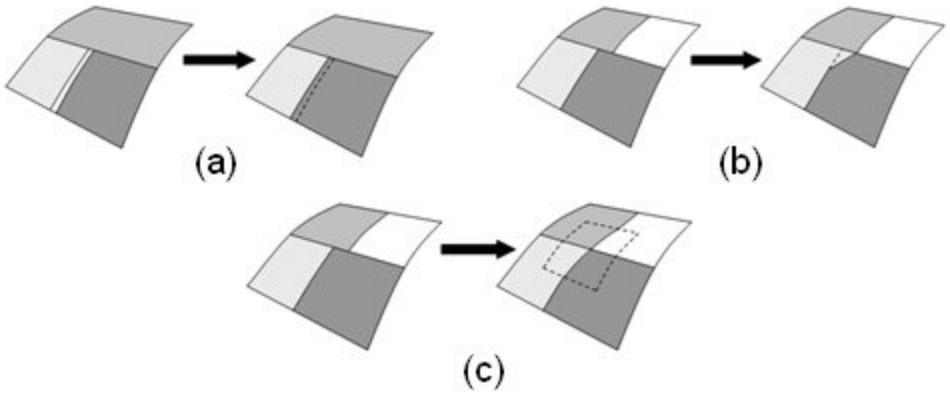



**Figure 1. Three operators used for removing small curves (a) composite; (b) collapse curve; (c) remove topology**

*Small and Narrow Surfaces*

ITEM also addresses the problem of small and narrow surfaces. Both are dealt with in a similar manner and are described here.

***Diagnostic:*** Small surfaces are found by comparing the surface area with a characteristic *small area*. The characteristic small area is defined simply as the characteristic small curve length squared or $\varepsilon^2$.

*Narrow surfaces* are distinguished from *surfaces with narrow regions* by the characteristic that the latter can be split such that the narrow region is separated from the rest of the surface. Narrow surfaces are themselves a narrow region and no further splits can be done to separate the narrow region. Figure 2 shows examples of each. ITEM provides the option to split off the narrow regions, subdividing the surface so the narrow surfaces can be dealt with independently.

Narrow regions/surfaces are also recognized using the characteristic value of $\varepsilon$. The distance, $d_i$ from the endpoints of each curve in the surface to the other curves in the surface are computed and compared to $\varepsilon$. When $d_i<\varepsilon$ other points on the curve are sampled to identify the beginning and end of the narrow region. If the narrow region encompasses the entire surface, the surface is classified as a narrow surface. If the region contains only a portion of the surface, it is classified as a surface with a narrow region.

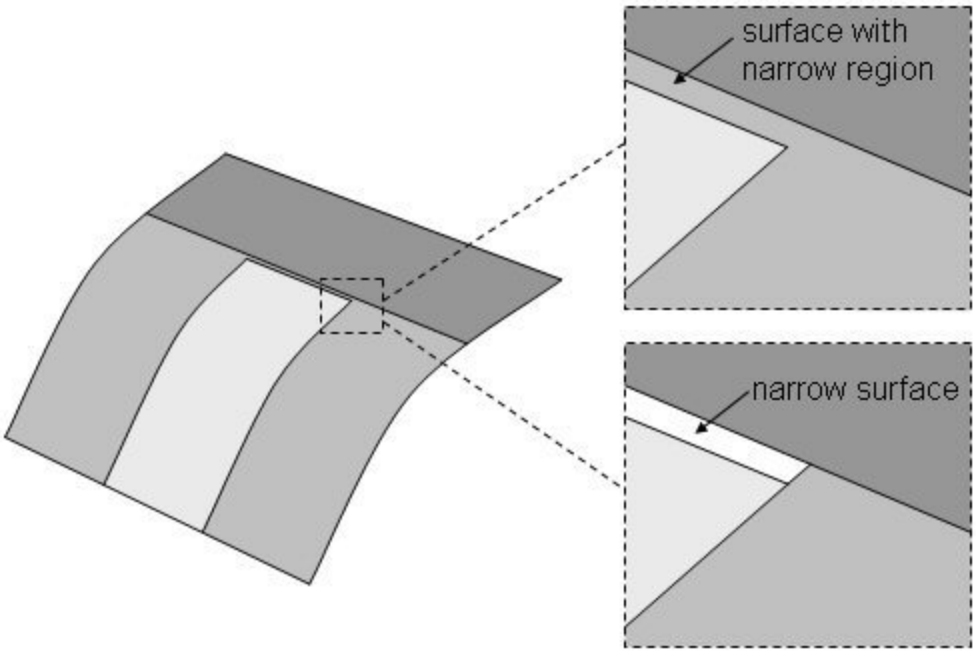



**Figure 2. Two cases illustrating the difference between surfaces with narrow regions and narrow surfaces**

***Solutions:*** ITEM provides four different solutions for eliminating small and narrow surfaces from the model. The first solution uses the regularize operation. Regularize is a real operation provided by the solid modeling kernel that removes unnecessary/redundant topology in the model. In many cases a small/narrow surface's definition may be the same as a surface next to it and therefore the curve between them is not necessary and can be regularized out. An example of regularizing a small/narrow surface out is shown in Figure 3.

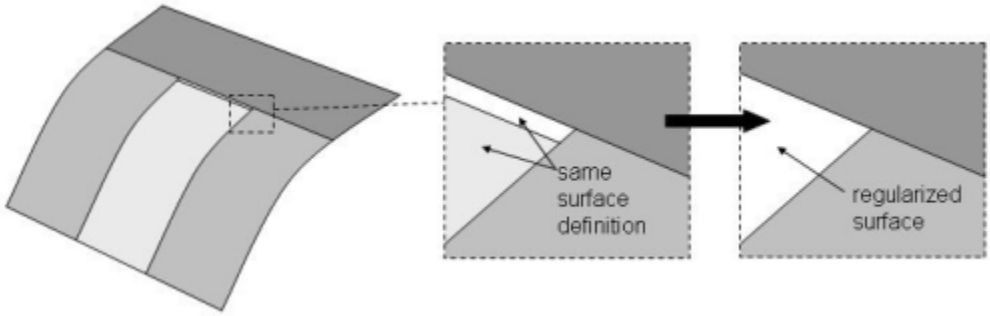



**Figure 3. When the small surface's underlying geometric definition is the same as a neighbor the curve between them can be regularized out.**

The second solution for removing small/narrow surfaces uses the remove operation. Remove is also a real operation provided by the solid modeling kernel. However, it differs from regularize in that it doesn't require the neighboring surface(s) to have the same geometric definition. Instead the remove operation removes the specified surface from

the model and then attempts to extend and intersect adjacent surfaces to close the volume. An example of using the remove solution is shown in Figure 4.

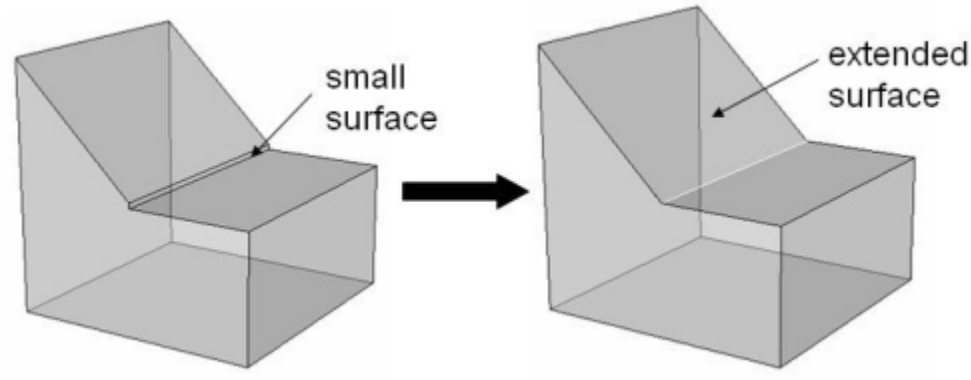



**Figure 4. The remove operation extends an adjacent surface to remove a small surface**

The third solution for removing small/narrow surfaces uses the virtual composite operation to composite the small surface with one of its neighbors. This is very similar to the use of composites for removing small curves. An example is shown in Figure 5.

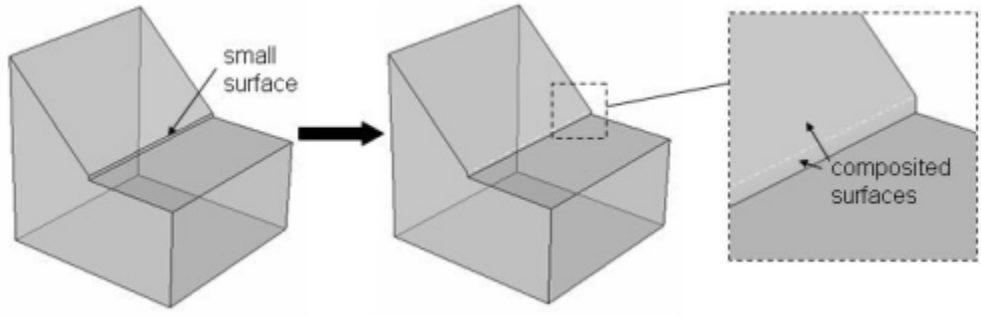



**Figure 5. Composite solution for removing a narrow surface**

The final solution for removing small/narrow surfaces uses the remove topology operation (Clark, 07). The remove topology operation behaves the same as when used for removing small curves in that it cuts out the area of the model around the small/narrow surface and replaces it with a simplified topology. In the case of a small surface where all of the curves on the surface are smaller than the characteristic small curve length, the small surface is replaced by a single vertex. In the case of a narrow surface where the surface is longer than the characteristic small curve length in one of its directions, the surface is replaced with a curve. The remove topology operation can be thought of as a local dimensional reduction to simplify the topology. The remove topology operation can also be used to remove networks of small/narrow surfaces in a similar fashion. Examples of using the remove topology solution to remove small/narrow surfaces are shown in Figures 6 and Figure 7.

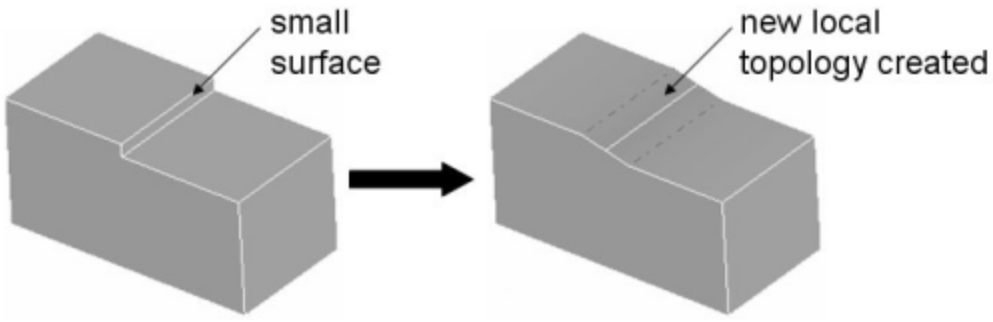



**Figure 6. Remove topology solution for removing a narrow surface**

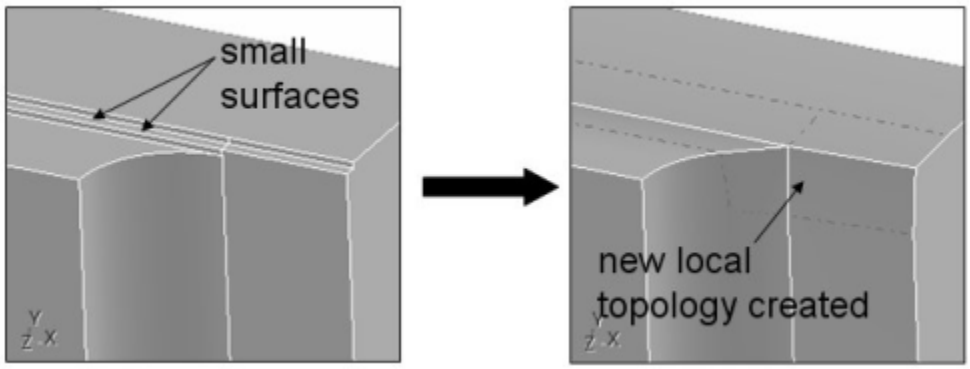



**Figure 7.Remove topology solution for removing a network of narrow surfaces**

## Determining the Small Feature Size

The smallest feature size is a value that represents the size of the smallest detail in the volume that the user wants to include in the final mesh. Any details that are smaller than this size should be removed from the model before completing the other steps of the meshing process. Small details can result from a variety of different reasons. Sometimes the model contains excessive detail that the user does not need. Other times, small features such as extra curves are created during import to account for a mismatching topology. Still other times, the small features are the result of webcutting or other decomposition methods. Ideally there should be a minimum threshold at which the user decides to keep all features above the given size, and remove the rest. The smallest feature size is used for other diagnostic tools, so selecting an appropriate feature size is important for other steps in the mesh generation process.

After the **Find Small Features** button is pressed, Cubit lists the 10 closests vertex-vertex and vertex-curve pairs. The pairs are listed in the display window from smallest to largest. To see more pairs, change the search parameter in the input box. To visualize each pair, the user can right click on a feature and select the *Draw Pair with Volumes* option. After determining the smallest feature size the user can enter it in the edit field at the bottom of the panel and it will be used in later calculations. The user can also right click on one of

the pairs in the list and choose *Use as smallest feature* to populate the edit field at the bottom of the panel.

*Why doesn't the list include small gaps between volumes?*

The smallest feature check is only searching over vertex-vertex and vertex-curve pairs in the same volume. Small gaps and misalignments are not included in this list. The purpose of the small feature diagnostic panel is to search for features that need to be removed prior to meshing. A feature is an entity such as a small curve or sliver surface that exists on a single volume which must be resolved by the mesh. A gap or misalignment is two entities that should be coincident, but are not, due to translation or other problems. Gaps and misalignments may not hinder mesh generation on a given volume, but they do prevent proper imprinting and merging.
The imprint/merge, merge tolerance, and overlapping volume panels contain diagnostics that check for misalignment problems. The purpose of those diagnostics is to enable imprinting and merging of a volume with small misalignments.
Note: The smallest feature size is used as a metric on the merge tolerance page, but it is only used to get an initial estimate for the merge tolerance. Small feature size and merge tolerance represent different metrics, and should not be confused.
In Figure 1, the small feature size diagnostic finds small features with lengths of 0.707, 0.15 and 0.25. The user may decide that the smallest feature he or she wishes to keep is the one at the 0.25 size. If he sets the small feature size to 0.25, the other features will be flagged as small curves and surfaces on the Small Features page. They can then be removed using tweaking and other geometry clean-up commands. If he sets the small feature size to 0.707, none of the features will be flagged as small features.
In addition to the features shown, this model contains two vertices that are slightly misaligned due to geometry translation problems. The nearly coincident vertices are not listed on the small features list because the vertices lie on different volumes. To find these near coincident vertices, the user would use the merge tolerance panel.

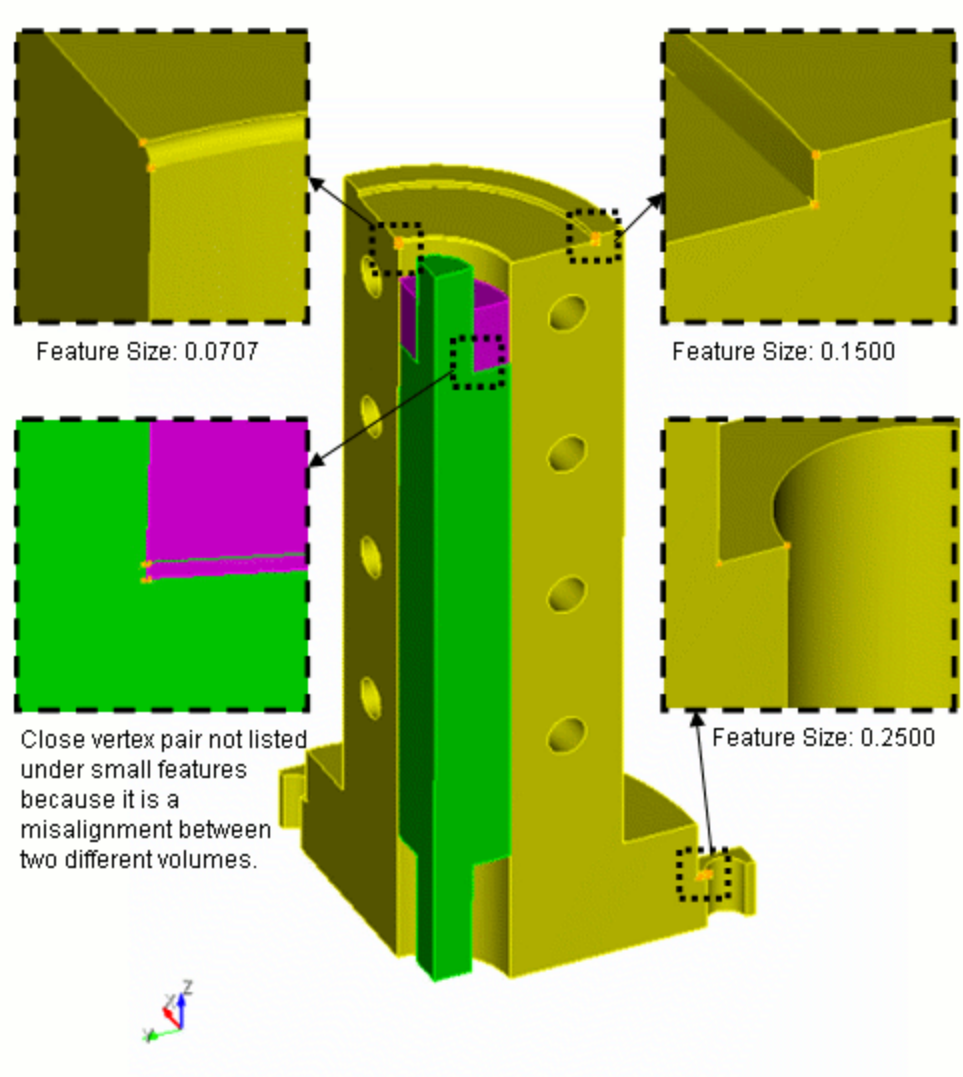



Figure 1. Small Features and Overlap on a Model

# Appendix

## Appendix


- Alpha Commands
- Available Colors
- Element Numbering
- FullHex vs. NodeHex Representation
- APREPRO
- Python Cubit Enhancement Scripts
- Cubit Python Interface
- Navigation XML Files
- FASTQ
- Periodic Space-filling Models (Tile)
- Generating Meshes for Adaptive Topological Optimization (ATO)
- References


## alpha

### Alpha Commands

CUBIT has several functions that are currently in development and are considered "Alpha" features. These features can be can be accessed or hidden within Cubit by typing the following command:

**Set Developer Commands {On|OFF}**

The commands that are currently developer commands are:

- Automatic Detail Suppression
- Cohesive Elements
- Deleting Mesh Elements
- Feature Size
- Optimize Jacobian
- Mesh Cutting
- Mesh Grafting
- Randomize Smoothing
- Refine Mesh Boundary
- Super Sizing Function
- Test Sizing Function
- Triangle Mesh Coarsening
- Transition
- Importing MBG Files
- Exporting MBG Files
- Remove Tiny Edge Length
- Exporting SGM Files

## Creating ACIS Geometry From Mesh

**Note:** These features are under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

Using the **Acis** options (in red) in the commands below will produce ACIS geometry instead of mesh-based geometry. ACIS geometry is generally more desirable than mesh-based geometry because it can be modified easily.

#### Importing a Mesh

**Import Mesh Geometry '<exodusII_filename>' [Block <id_range>|ALL] [Unique Genesis IDs] [Start_id <id>] [Use [NODESET|no_nodeset] [SIDESET|no_sideset] [Feature_Angle <angle>] [LINEAR|Gradient|Quadratic|Spline|Acis] [Deformed {Time <time>|Step <step>|Last} [Scale <value>] ] [MERGE|No_Merge] [Export_facets <1|2|3>] [Merge_nodes <tolerance>]**

This command tries to associate the mesh to the ACIS geometry that is created. If the association fails, the mesh ends up as free mesh. For more information on this command see: Importing Exodus II Files

#### Existing Mesh - Create Mesh Geometry

**Create Mesh Geometry {Hex|Tet|Face|Tri|Block} <range> [Feature_Angle <angle=135>] [Acis] [Keep]**

This command tries to associate the mesh to the ACIS geometry that is created. If the association fails, the mesh ends up as free mesh. For more information on this command see: Free Meshes

#### Existing Mesh - Create Geometry

**Create Geometry {Hex|Tet|Face|Tri} <id_range>**

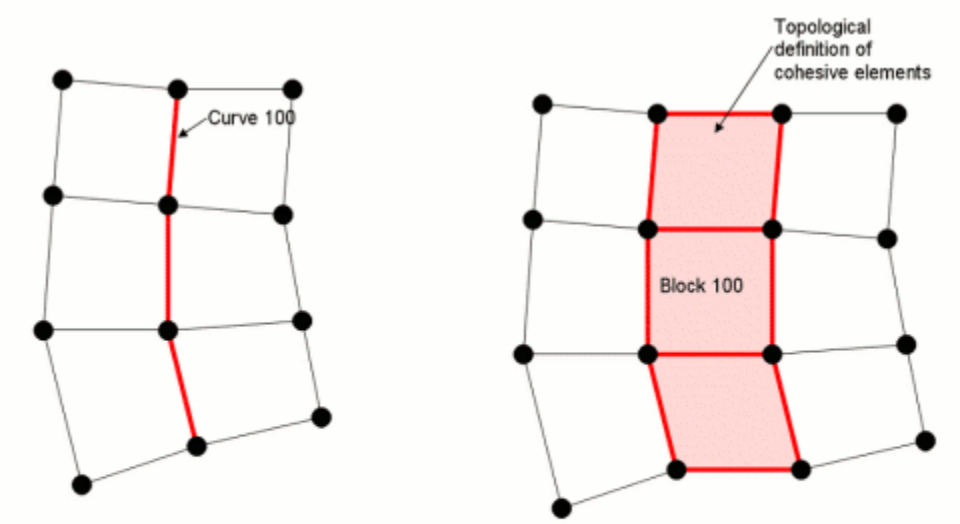
These two commands do not use blocks, sidesets, nodesets or a user-specified dihedral angle to create vertices, curves, and surfaces. These commands use a proprietary third-party routine to create geometry. They also do not associate the mesh to the ACIS geometry that is created.

## Cohesive Elements

**Note:** This feature is under development. The command to enable or disable features under development is:

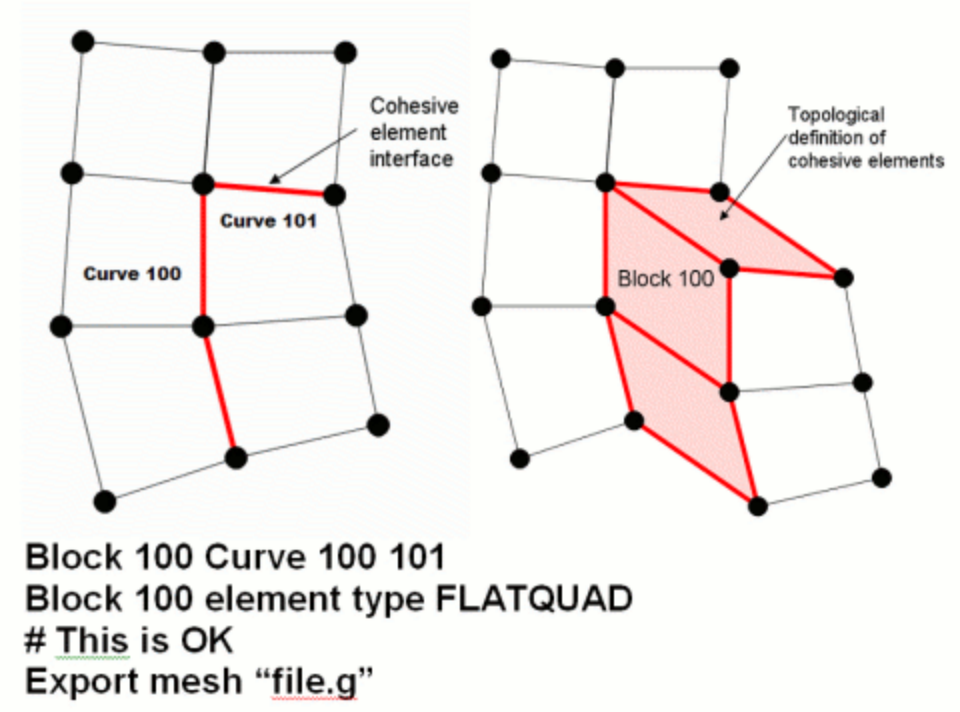
**Set Developer Commands {On|OFF}**

Cohesive elements are used to model things like adhesive that may lose its bond. Elements in a cohesive region originally have zero volume or area, and then expand as the simulation progresses.
Cubit supports 2D cohesive regions. Cohesive elements are implemented in Cubit as element blocks with an element type of FLATQUAD. The cohesive region is identified by assigning geometric curves to the FLATQUAD element block. When the element block is exported, each edge on the specified curves is represented in the exported file as a 4-noded quadrilateral element with zero area. The quadrilateral element is formed by duplicating each node in the original edge and then connecting the two original and two duplicate nodes to form a zero-area quadrilateral.
The image below shows how a FLATQUAD is represented in an exported mesh file. The figure on the left is how the mesh appears in Cubit. The figure on the right is how the mesh appears in the output file. Note that the figure on the right is a topological representation, not a true geometric representation. In reality, the nodes on the left side of block 100 are coincident with the nodes on the right side of block 100, causing the pink elements to have zero area.



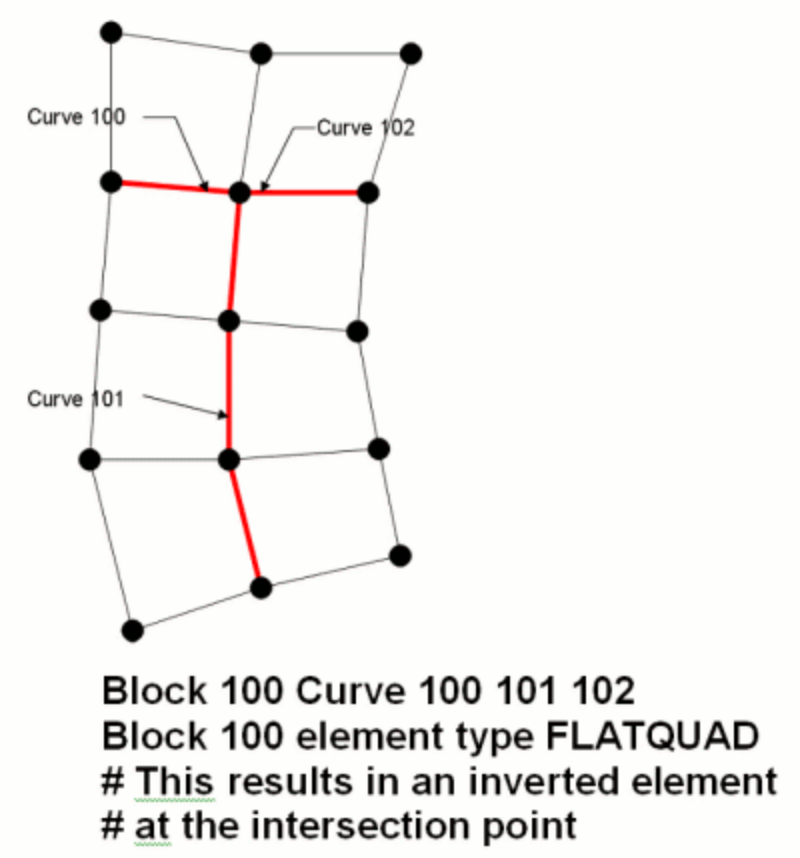

*Multiple Curves in FLATQUAD Blocks*

Multiple curves may be assigned to a single FLATQUAD element block, as long as the curves do not form a branching path. The figure below, for example, shows an acceptable configuration of multiple curves.




```
Block 100 Curve 100 101
Block 100 element type FLATQUAD
# This is OK
Export mesh "file.g"
```

Although multiple curves may be assigned to a single cohesive block, the curves assigned to a block of type FLATQUAD must not branch. A branch occurs whenever three or more curves share a common vertex, as shown in the figure below.




## Deleting Mesh Elements

Element deletion for owned geometry is no longer available unless the developer flag is turned on. Element deletion is still available without the developer flag for free meshes. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

The following forms of the delete commands operate on meshed entities only. They allow low-level editing of meshes to make minor corrections to a mostly correct mesh. They are not designed for major modifications to existing meshes. Because Cubit's display routines were not designed with these type of operations in mind, these commands may cause the current display of the affected entities to take an unexpected form. An appropriate drawing command can used to return the display to the desired view.

When deleting elements, the default behavior will be that the child mesh entities will be deleted when they become orphaned. For example, when a hex is deleted, if its faces, edges and vertices are no longer used by adjacent hex elements, then they will also be deleted. The **no_propagate** option will leave any child mesh entities regardless if they become orphaned.

The delete command removes one or more mesh entities from an existing mesh. Additional mesh entities may be deleted as well depending on the particular form of the command. Exactly which entities are removed is explained in the following descriptions.

**Delete {Hex|Tet} <range> [No_Propagate]**

Deletes the specified hexes or tets. All associated tris, faces, edges, and nodes are also deleted unless the *no_propagate* option is given.

**Delete Wedge <range>**

Deletes the specified wedges. No other mesh entities are affected.

**Delete {Face|Tri} <range> [No_Propagate]**

Deletes the specified faces or tris. For faces, all hexes that contain the face are also deleted. For tris, all tets that contain the tri are also deleted. All associated edges and nodes are also deleted unless the *no_propagate* option is given.

**Delete Edge <range> [No_Propagate]**

Deletes the specified edges. Any associated tris, faces, hexes, and tets are also deleted. Any associated nodes are also deleted unless the *no_propagate* option is given.

**Delete Node <range>**

Deletes the specified nodes. Any associated edges, tris, faces, hexes, and tets are also deleted.

## FeatureSize

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

**Applies to:** Curves
**Summary:** Meshes a curve based on its proximity to nearby geometry and size of nearby geometric features. This is an alpha feature and should be used with caution.
**Syntax:**

**Curve <range> Scheme Featuresize**

**Related Commands:**

**Curve <range> Density <density_factor>**

**Discussion:**
The user may also automatically bias the mesh from small elements near complicated geometry to large elements near expanses of simple geometry. Meshing a curve with scheme featuresize places nodes roughly proportional to the distance from the node to a

piece of geometry that is foreign to the curve. Foreign means that the geometric entity doesn't contain the curve, or any of its vertices (i.e. the entity's intersection with the curve is empty). It is known that featuresize is a continuous function that varies slowly. Featuresize meshing is very automatic and integrated with interval matching. Featuresize meshing works well with paving, and in some cases with structured surface-meshing schemes (map, submap) as well.
If desired, the user may specify the exact or goal number of intervals with a size or interval command, and then the featuresize function will be used to space the nodes.
The featuresize function may also be scaled by the user to produce a finer or coarser mesh using the **density** command as follows:

**Curve <range> Density <density_factor>**

The default scaling factor or **density** is 1. Higher densities also reduce the transition rate of the node spacing. A density of 2 usually gives a good quality mesh. A density below about 0.5 could produce rapid transitions and poor mesh quality. The following shows an example of different density values when using the featuresize scheme.




## Importing Abaqus Files

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set developer commands {on|OFF}**

The command to import a mesh from an Abaqus format file is:

**Import Abaqus [Mesh Geometry] '<input_filename>' [Feature Angle <angle>]**

For a description of importing mesh geometry see Importing Exodus II Files.

## Importing Meshed Based Geometry Files (MBG)

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

CUBIT provides the capability to import a model composed of mesh based geometry. The command to import meshed based geometry is:

**Import mbg "<filename>"**

**MBG** is created in Cubit when one meshes a volume or imputs the mesh from a previously meshed volume with the **import mesh geometry** command. Optionaly there one may create geometry with the "set dev on" option.

In order to create, import and export MBG one needs to set the geometry engine to facet with the following command "set geom eng facet".

The following commands create a brick and export and import it as a MBG file: set geometry engine facet set dev on brick x 10 export mbg "brick.mbg" overwrite reset import mbg "brick.mbg"

**set geometry engine facet**
**set dev on**
**brick x 10**
**export mbg "brick.mbg" overwrite**
**reset**
**import mbg "brick.mbg"**

## Exporting Meshed Based Geometry Files (MBG)

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

CUBIT provides the capability to import a model composed of mesh based geometry. The command to import meshed based geometry is:

**Export mbg "<filename>"**

MBG is created in Cubit when one meshes a volume or imputs the mesh from a previously meshed volume with the import mesh geometry command. Optionaly one may create geometry with the "set dev on" option.

In order to create, import and export MBG one needs to set the geometry engine to facet with the following command "set geom eng facet".

The following commands create a brick and export and import it as a MBG file:

**set geometry engine facet**
**set dev on**
**brick x 10**
**export mbg "brick.mbg" overwrite**
**reset**
**import mbg "brick.mbg"**

## Mesh Cutting

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

The term "mesh cutting" refers to modifying an existing mesh by moving nodes to a cutting entity and modifying the connectivity of the mesh so that the original mesh fits a new geometry. The behavior of mesh cutting is intended to be similar to web cutting in that the process results in a decomposition of the original geometry. The difference is that the decomposition is performed on meshed geometry and results in the creation of virtual geometry partitions. The underlying acis body remains unchanged. The user has the option to determine what is partitioned during mesh cutting: the volume, the surfaces only, or nothing.

The current scope of mesh cutting is limited to cutting hex meshed volumes with planes and extended surfaces. These cutting entities are also limited in that mesh cutting will not work if they pass through a vertex at the end of more than two curves. Mesh cutting does not work on tet meshes or surface meshes.

The steps of mesh cutting include:

- **Create a starting mesh**. This mesh is typically simpler than the desired final mesh and can be created with sweeping, mapping, or some other available meshing algorithm. Currently, the starting mesh must be a single volume: mesh cutting does not handle merged volumes or assemblies.
- **Create a cutting entity** that can be used to capture the new detail in the mesh. Currently, mesh cutting works with planes or sheets extended from surfaces. It is important to note that if an extended surface is used, mesh cutting will not capture any geometric features (curves or vertices) of the surface.
- **Issue the command** to cut the mesh. The meshcut commands are similar in syntax and behavior to the webcut commands.

The following entities with the associated commands are available for mesh cutting:

#### Coordinate Plane

A coordinate plane can be used to cut the model, and can optionally be offset a positive or negative distance from its position at the origin.

**Meshcut Volume <range> Plane {xplane|yplane|zplane} [offset <dist>]**

The planar surface to be used for mesh cutting can also be previewed using the Draw Plane command.

#### Planar Surface

An existing planar surface can also be used to cut the model.

**Meshcut Volume <range> Plane Surface <surface_id>**

The planar surface to be used for mesh cutting can also be previewed using the Draw Plane command.

#### Plane from 3 points

Any arbitrary planar surface can be used by specifying three nodes that define the plane.

**Meshcut Volume <range> Plane Node <3_node_ids>**

#### Extended Surface

An extended surface or "sheet" can also be used for mesh cutting. In this case, the sheet is not restricted to be planar and will be extended in all directions possible. When cutting with an extended surface mesh cutting will ignore all curves and vertices of the surface. Also, the resolution of the mesh will determine how well curved surfaces are captured with meshcutting. A surface with high curvature will not be captured accurately with a coarse mesh. Note that some spline surfaces are limited in extent and may not give an expected result from mesh cutting.

**Meshcut Volume <range> Sheet [Extended From] Surface <surface_id>**

**Note**: When cutting with surfaces extended from composite surfaces the default underlying surface approximation may result in a poor final mesh for mesh cutting. This problem can be fixed using the following command:

**Composite closest_pt surface <id> gme**

See the discussion on composite geometry for a more detailed description of this command.

*Meshcut Options*

The following options can be used with all the meshcut commands:
**[PARTITION VOLUME|partition surface|no_partition]**: By default, mesh cutting will create virtual partitions of the volume being cut to match the cutting entity. This option allows mesh cutting to also create only the surface partitions or create no partitions for the volume or surfaces.
**[no_refine]**: This option tells mesh cutting not to refine the mesh around the cutting entity.
**[no_smooth]**: This option tells mesh cutting not to perform the final smoothing step after the cut has been made.

*Meshcutting Scope*

The following is a list of the current scope and limitations of meshcutting.

- Meshcutting only works on hex meshes.
- Meshcutting only works for single volumes. It currently does not handle assembly meshes.
- Currently, only planes and extended surfaces can be used as the cutting entity.
- Curves and vertices on the cutting entity will not be captured in the mesh.
- Meshcutting will not work if the cutting entity passes through a meshed vertex that is at the end of more than two curves.
- The resolution of the mesh determines how well a non-planar cutting entity will be captured in the resulting mesh. Small features and high curvature will not be captured by a coarse mesh.
- Spline surfaces are limited in extent and may not give expected results if used as an extended cutting surface.

*Meshcutting Example*

The figures below show an example of mesh cutting. Figure 1 shows the body that will be meshed. This body is a brick with intersecting through-holes. The steps to create a mesh for this body are listed below.

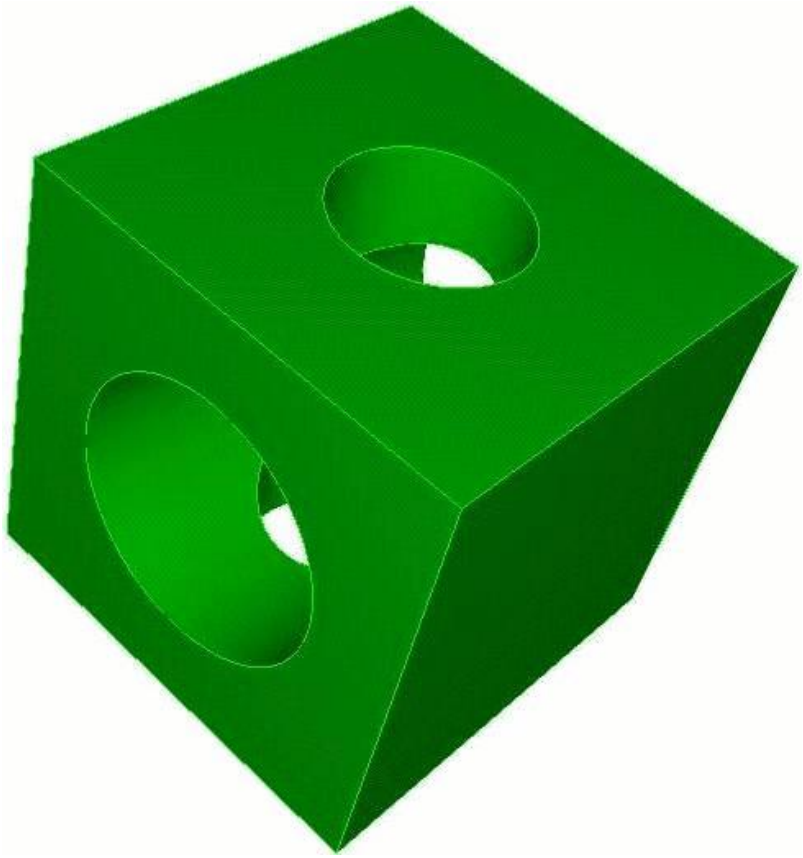


**Figure 1: The original, unmeshed body**

**Step 1**: Create a starting mesh. Figure 2 below shows the starting mesh for this problem. The commands for this mesh are:

**cubit> reset**
**cubit> set dev on**
**cubit> create brick x 10**
**cubit> create cylinder radius 3 z 15**
**cubit> subtract 2 from 1**
**cubit> volume 1 scheme sweep**
**cubit> volume 1 size .75**
**cubit> webcut volume 1 with plane xplane offset 0**
**cubit> merge all**
**cubit> mesh volume 1 3**

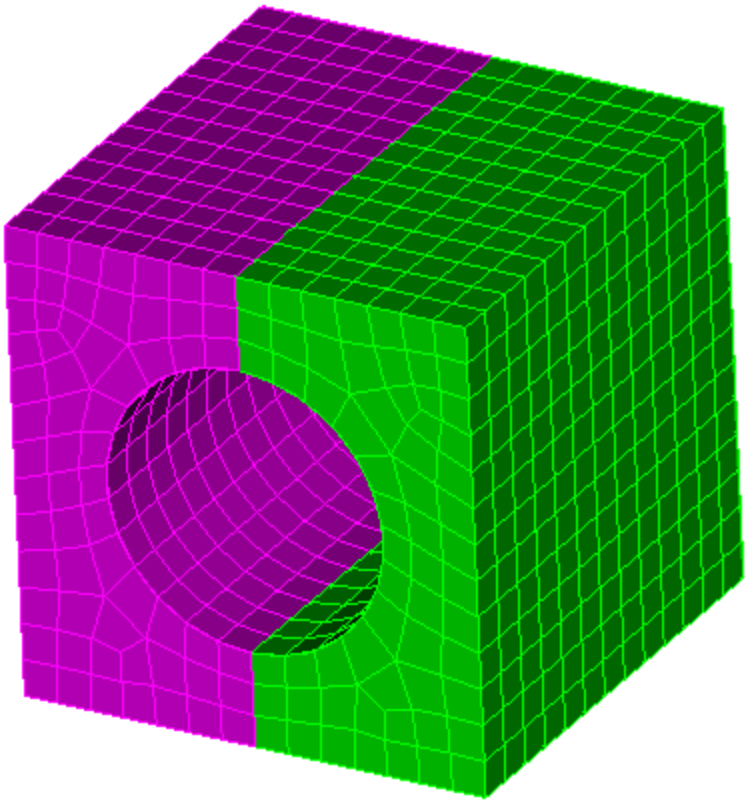


**Figure 2: The starting mesh**

**Step 2**: Create a cutting entity. Figure 3 shows the volume that will be used to cut the mesh. The commands are:

**cubit> create cylinder radius 2 z 15**
**cubit> rotate volume 4 about x angle 90**

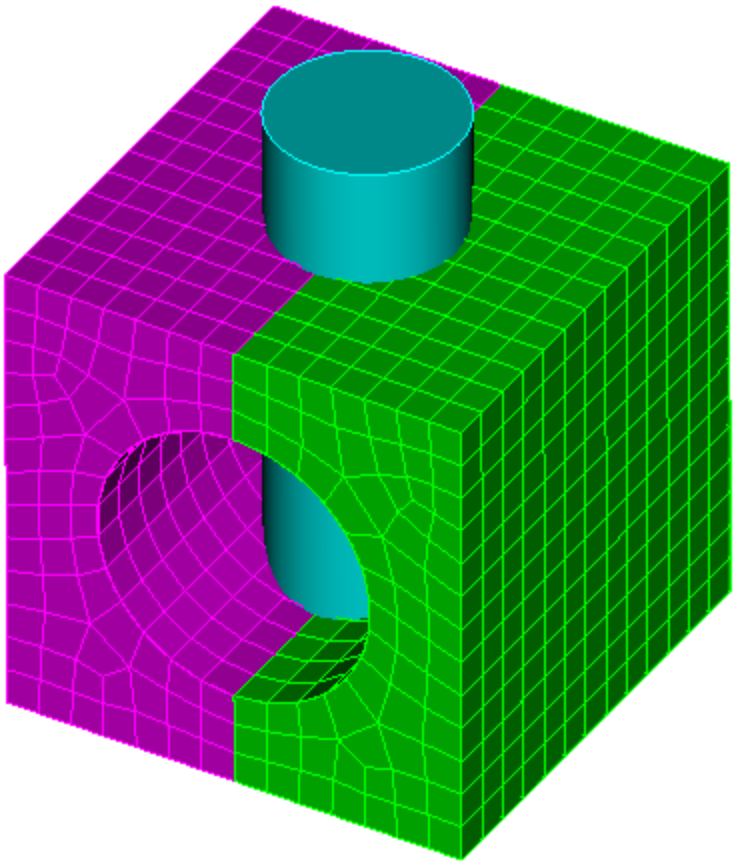


**Figure 3: The starting mesh and cutting entity**

**Step 3**: Cut the mesh. Figure 4 shows the new mesh after the original mesh has been cut. Also recommend smoothing the mesh:

**cubit> meshcut vol 1 3 sheet surface 27**
**cubit> surf in vol 1 3 smooth sch condition number beta 2 cpu 0.5**
**cubit> smooth surface in volume 1 3**
**cubit> vol 1 3 smooth scheme condition number beta 2 cpu 0.5**
**cubit> smooth volume 1 3**
**cubit> draw volume 1 3**

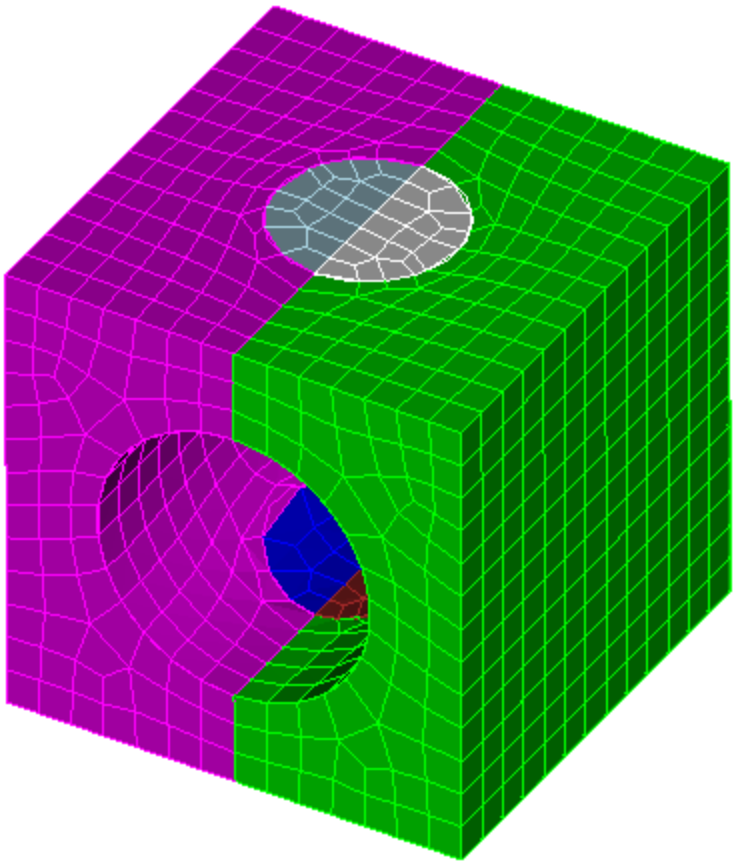


**Figure 4: The mesh after meshcutting**

**Step 4**: Final step. Figure 5 shows the quality of the final mesh. The command is:

**cubit> quality volume 1 3 scaled jacobian global draw mesh**

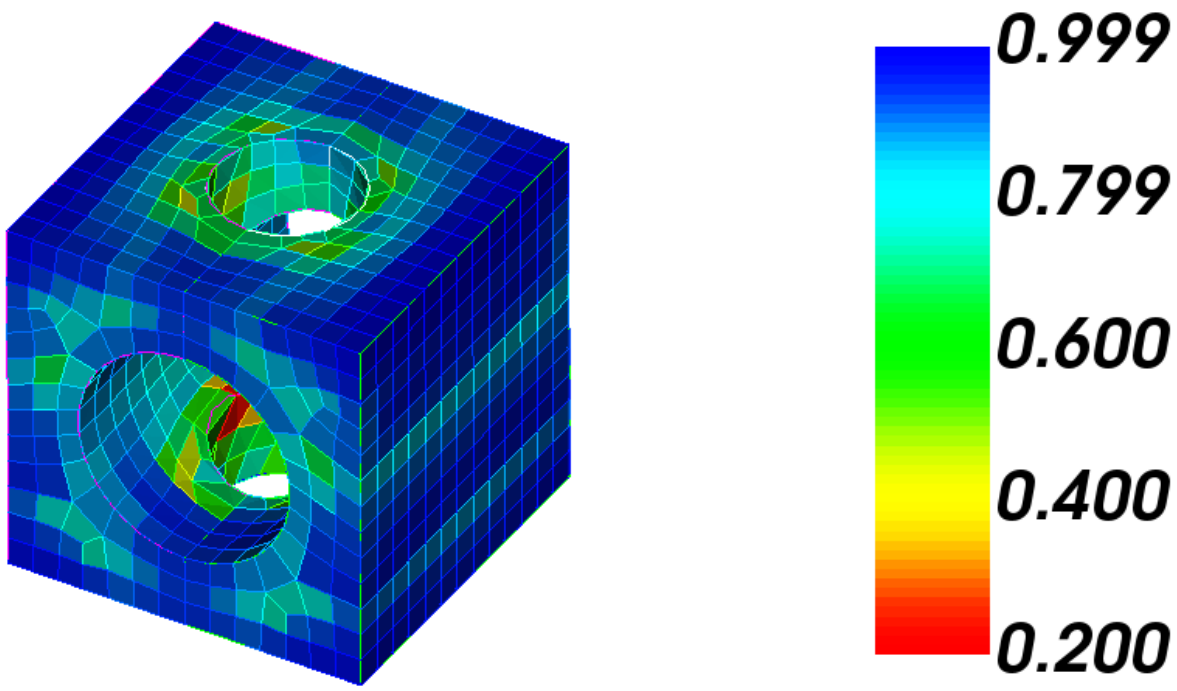



**Figure 5: Quality of final mesh**

## Mesh Grafting

**Note:** This feature is under development. The command to enable or disable features under development is:

```
Set Developer Commands {On|OFF}
```

Grafting is used to merge a meshed surface with a dissimilar unmeshed surface. In the process, the location of the nodes on the meshed surface will be adjusted to fit to the bounding curves of the unmeshed surface and the connectivity of the original mesh may be changed to improve the final quality of the mesh. This allows an unmeshed volume to be attached--or grafted--onto a meshed volume. Grafting is particularly useful for models that have intersecting sweep directions (see example below).
The command syntax for grafting is:

```
Graft {Surface <range> | Volume <id>} onto Volume <id> [no_refine]
[no_smooth]
```

The Graft command will check that the second volume is meshed. It then searches for surfaces on the second volume that overlap with the other volume or range of surfaces that is specified. If overlapping surfaces are found the mesh will then be adjusted on the second volume and after any needed imprinting is done the overlapping surfaces will be merged together.

*Grafting Options*

**[no_refine]:** This option tells grafting not to modify the connectivity of the original mesh. The mesh is still adjusted to fit the boundary of the branch surface but no new elements are added.
**[no_smooth]:** This option tells grafting not to perform the final smoothing of the modified surface or volume mesh.

*Grafting Scope*

The following is a list describing the current scope and limitations of grafting:

- Grafting only works on volumes meshed with hex elements.
- The unmeshed branch surface cannot have any point outside the boundary of the meshed trunk surface.
- Grafting may have difficulty with branch surfaces that are very thin with respect to the element size of the meshed surface or that have sharp angles.
- If grafting fails some of the nodes of the original mesh may have been moved. Check the mesh quality and re-smooth if needed.

**Grafting Example**
This example shows the four basic steps of grafting:

1. Partition the geometry (optional).
2. Mesh the trunk volume.
3. Graft the branch volume onto the trunk volume.
4. Mesh the branch volume.

**Step 1: Partition the geometry**
Figure 1 shows the model that will be meshed. The arrows in the figure show the two intersecting sweep directions. Figure 2 shows the model decomposed for grafting.

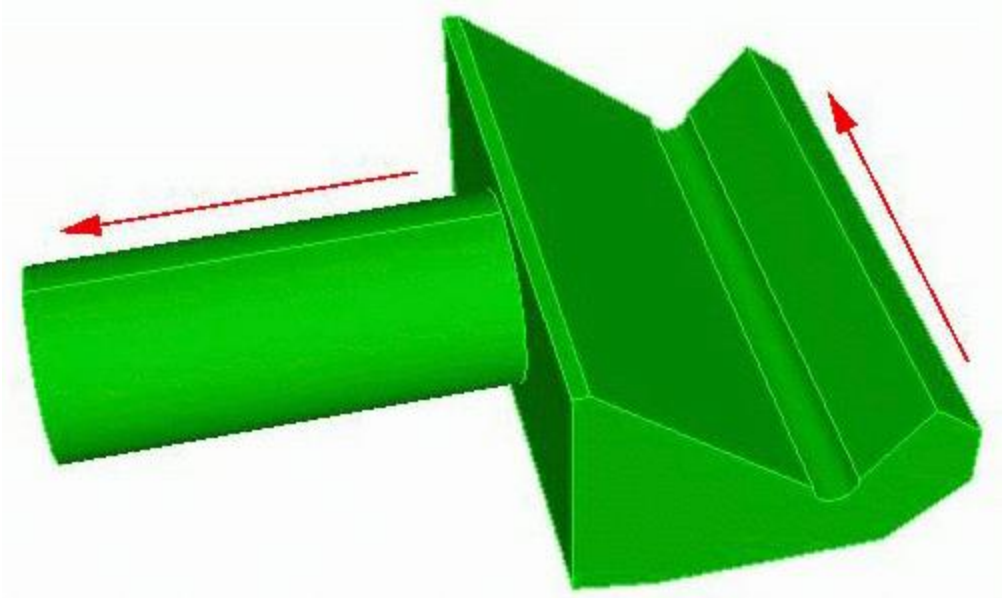


**Figure 1. A model with two intersecting sweep directions.**

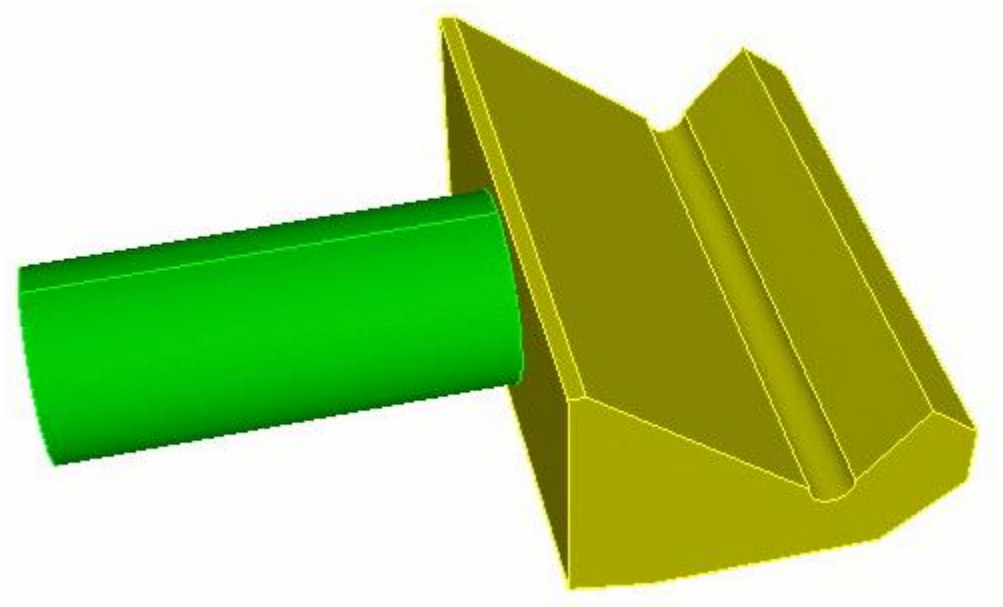


**Figure 2. The model decomposed for grafting**

**Step 2: Mesh the trunk volume.**
Figure 3 shows the mesh of the trunk volume. At this point the mesh on the trunk surface adjacent to the branch surface is a structured mesh that does not align with the boundary of the branch surface. The trunk and branch surfaces are two separate surfaces.

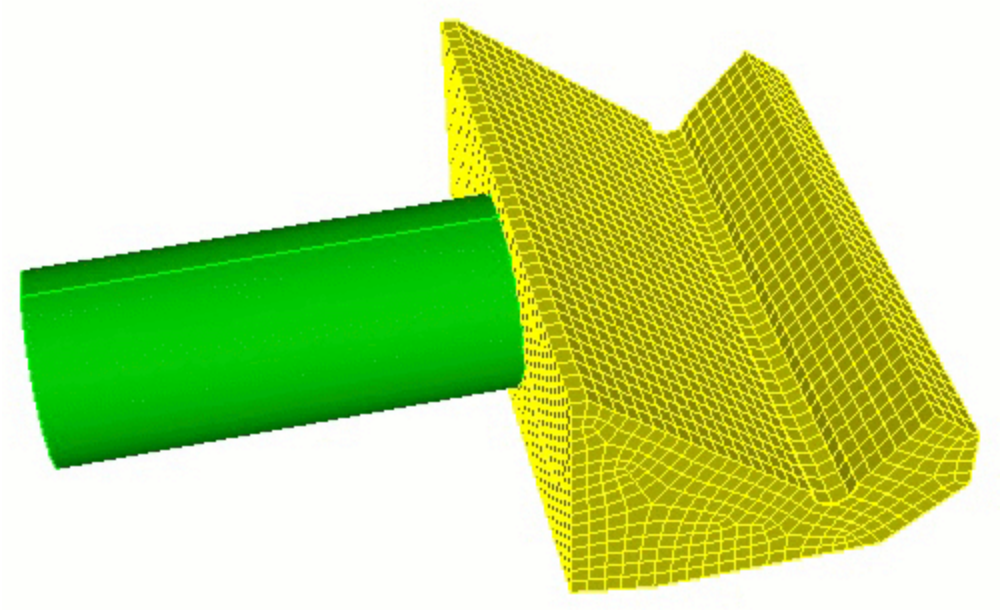


**Figure 3. Meshed trunk volume.**

**Step 3: Graft the branch onto the trunk**
Figure 4 shows the trunk surface after it has been modified to fit the branch surface. At this point the two surfaces have been merged together.

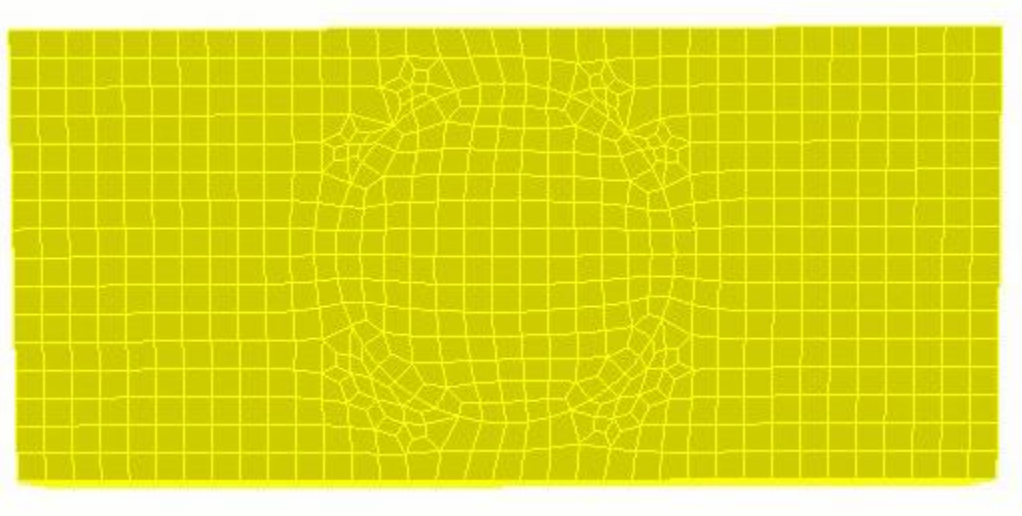


**Figure 4. Trunk surface after grafting.**

**Step 4: Mesh the branch volume.**
The final mesh is shown in Figure 5.

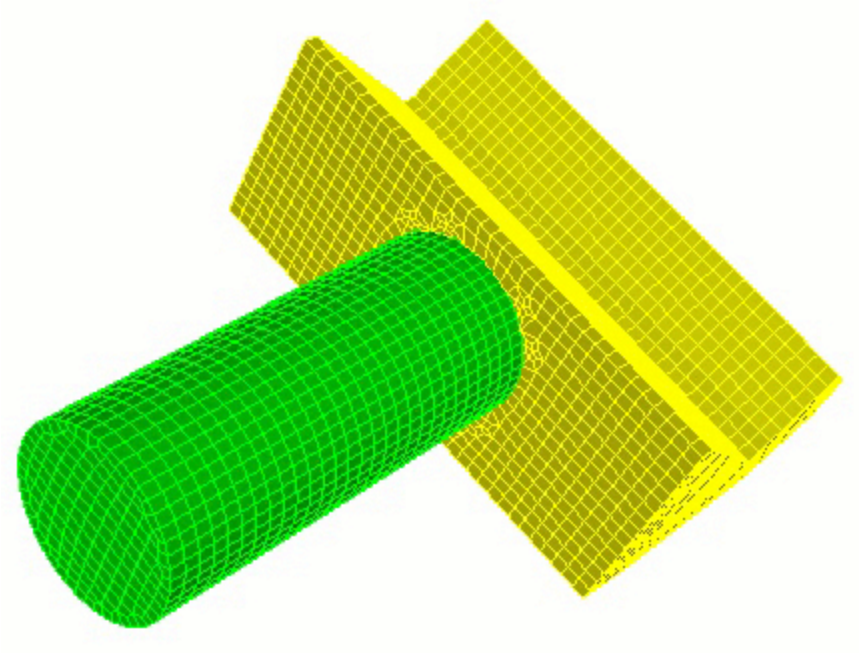


**Figure 5. Final mesh**

## Optimize Jacobian

**Note:** This feature is under development. The command to enable or disable features under development is:

```
Set Developer Commands {On|OFF}
```

**Applies to:** Volume meshes
**Summary:** Produces locally-uniform hex meshes by optimizing element Jacobians
**Syntax:**

```
Volume <range> Smooth Scheme Optimize Jacobian [param]
```

**Discussion:**
The Optimize Jacobian method minimizes the sum of the squares of the Jacobians (i.e., volumes) attached to the smooth node. Meshes smoothed by this means tend to have locally-uniform hex volumes.
The parameter <**param**> has a default value of 1, meaning that the method will attempt to make local volumes equal. The parameter, which should always be between 1 and 2 (with 1.05 recommended), can be used to sacrifice local volume equality in favor of moving towards meshes with all-positive Jacobians.

## Randomize

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

**Applies to**: Curve, Surface and Volume meshes
**Summary**: Randomizes the placement of nodes on a geometry entity
**Syntax**:

**{Surface|Volume} <range> Smooth Scheme Randomize [percent]**

**Discussion:**

This scheme will create non-smooth meshes. If a percent argument is given, this sets the amount by which nodes will be moved as a percentage of the local edge length. The default value for percent is 0.40. This smooth scheme is primarily a research scheme to help test other smooth schemes.

## Refine Mesh Boundary

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

Boundary effects to be modeled in the analysis code frequently require a refined mesh near a specific surface. CUBIT provides this capability with the Refine Mesh Boundary command. This command is similar to the Refine Mesh Volume Feature command except that it can insert multiple sheets of hexes near the specified surface.

**Refine Mesh Boundary Surface <range> Volume <id> {Bias <double>} {First_delta <double> | Thickness <double>} [Layer <num_layers=1>] [SMOOTH|No_smooth]**

With this command **num_layers** of hexes can be inserted at the first interval from the specified surface. A **bias factor** indicating the change in element size must be specified. You must also indicate a **first_delta** or **thickness** which represents the distance to the first inserted layer. The mesh in Figure 5 with bias 1.0 and first_delta of 5. The default smooth option provides the capability to smooth the mesh following the refinement procedure.

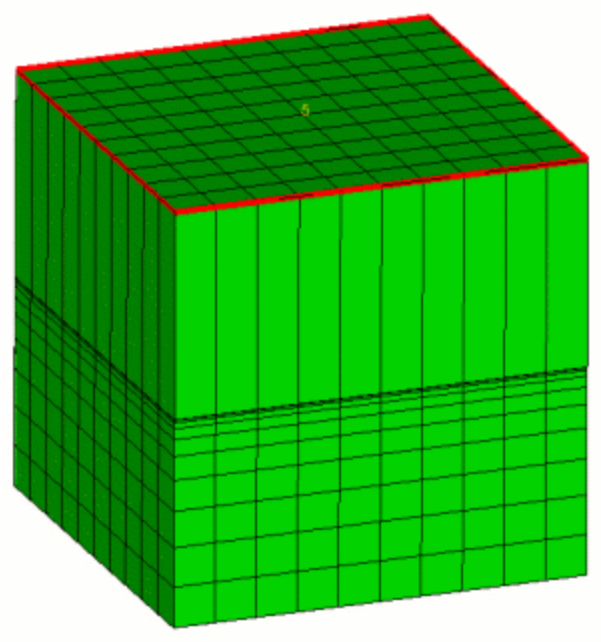


**Figure 5. Example of Boundary Surface Refinement**

## Remove Tiny Edge Length

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

**Applies to:** Trimesh Surface Scheme

**Summary:** Tolerance specified to prevent small edges in a triangle mesh

**Syntax:**

**[Set] Trimesher Remove Tiny Edge Length {<value>|[off]}**

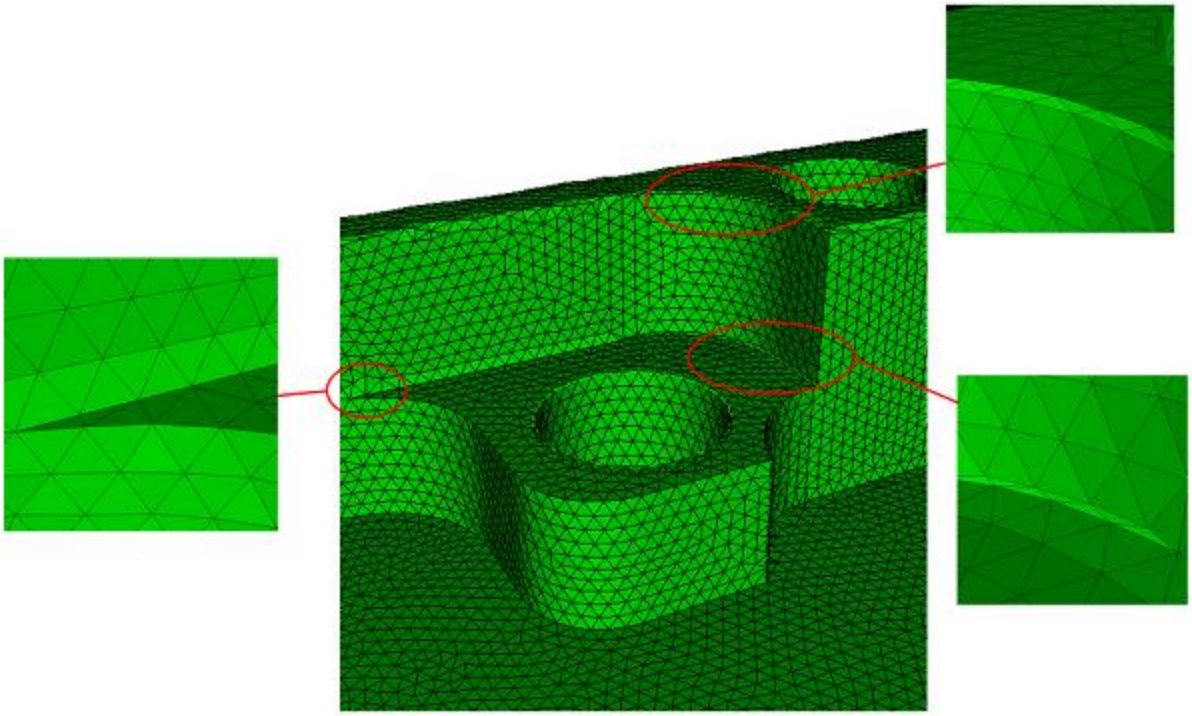
**Discussion**:

Setting the tiny edge length forces the MeshGems trimesher to generate triangles with edges greater than the specified value. It is actually a post processing step that collapses triangles with edges less than the specified value. This setting is necessary because the MeshGems triangle mesher sometimes inserts triangles with small edges along high curvature features, even though a larger size has been specified and geometry approximation has been turned off. Using this setting is the only way to guarantee that no edges smaller than the specified value will be created.

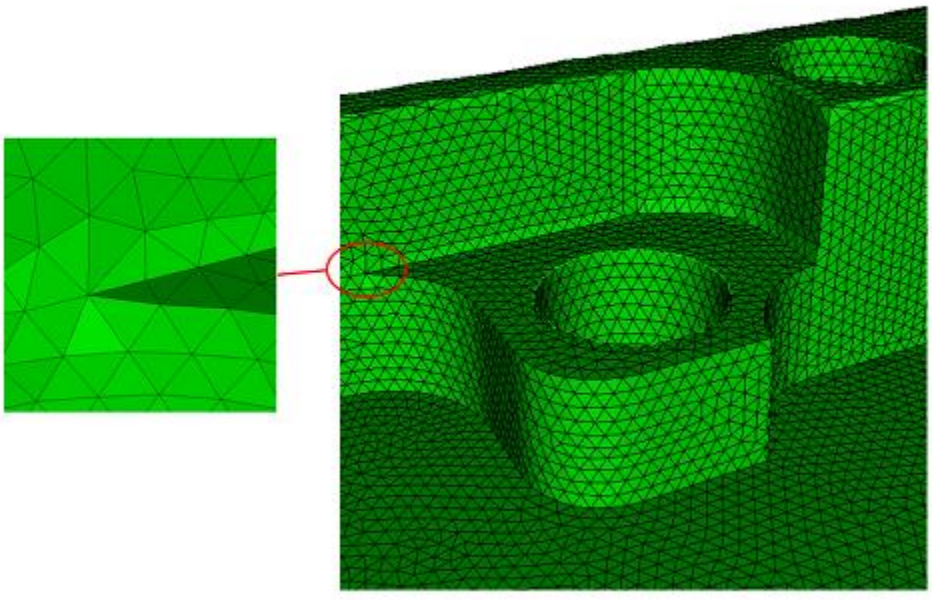
The **off** option resets the 'tiny edge length' value so it is not used.

The user should not use 'tiny edge length' values approaching the mesh size because an invalid mesh can result.

The images below show meshing a surface with and without setting a 'tiny edge length' value. In this example all surfaces have been composited into a single surface. Compositing small surfaces with larger neighbors in conjunction with using 'tiny edge length' has the effect of washing-over small features.



**Without using 'remove tiny edge length' triangles with short edges remain.**



**With appropriate 'remove tiny edge length' value triangles with short edges are collapsed.**

Super Sizing Function

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

The **Super** sizing function computes both the Curvature and the Linear function and takes the smaller value of the two. This is an alpha feature and should be used with caution. The following is an example of Super element sizing.

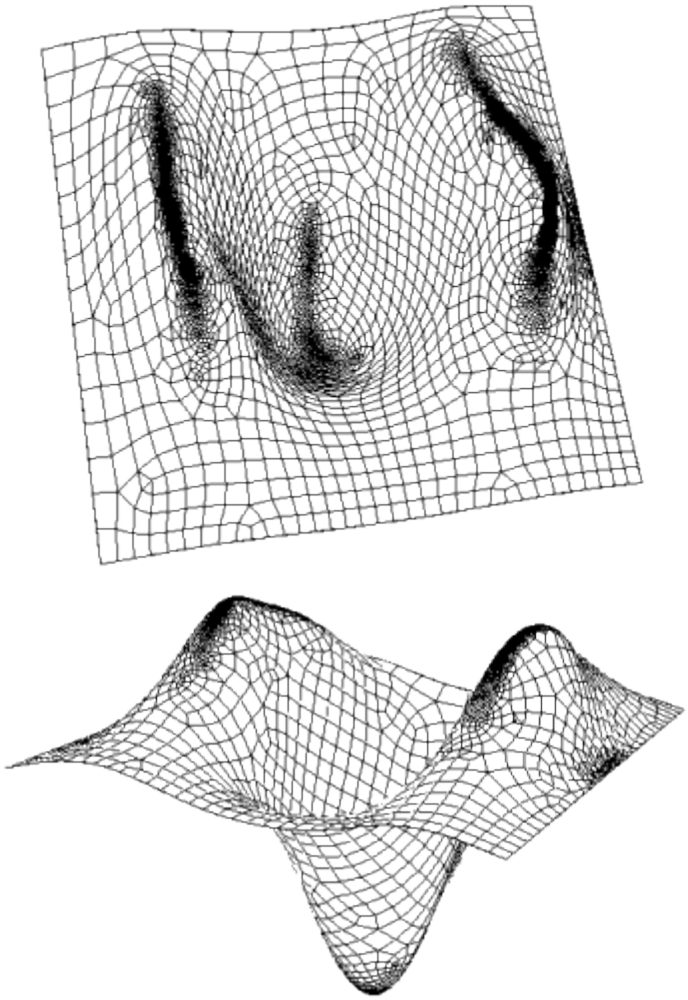


**Figure 1. NURB mesh with super sizing function, 34 by 16 density**

## Test Sizing Function

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

The **Test** sizing function is a hardwired numerical function used to demonstrate the transitional effect of sizing function-based and adaptive paving. The function is a periodic function which is repeated in 50x50 unit intervals on a 2D surface in the first quadrant (x

> 0, y > 0, z = 0). This is an alpha feature and should be used with caution. An example of a surface meshed with this sizing function is shown in Figure 1.

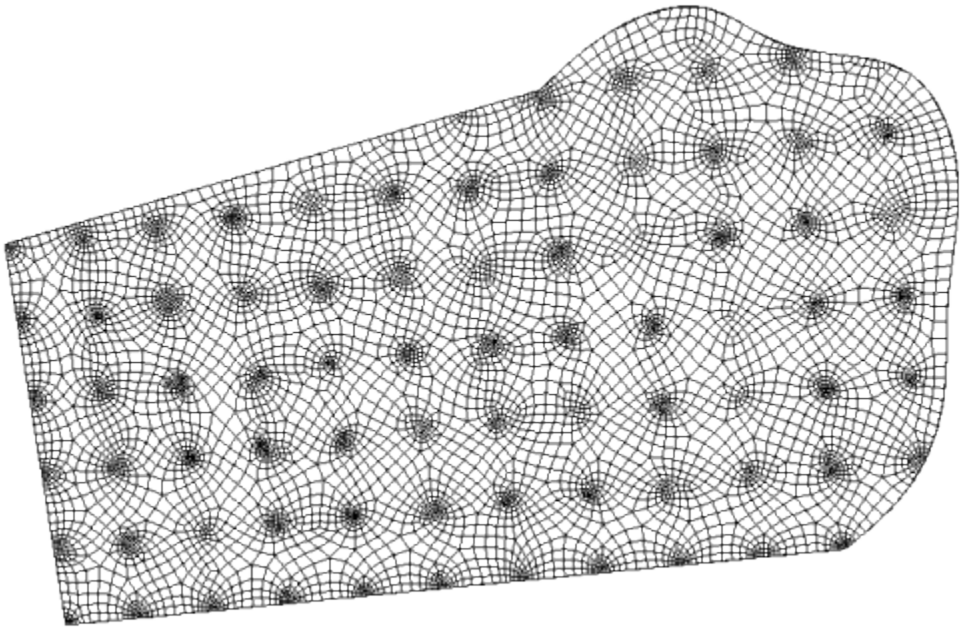


**Figure 1. Test sizing function for spline geometry**

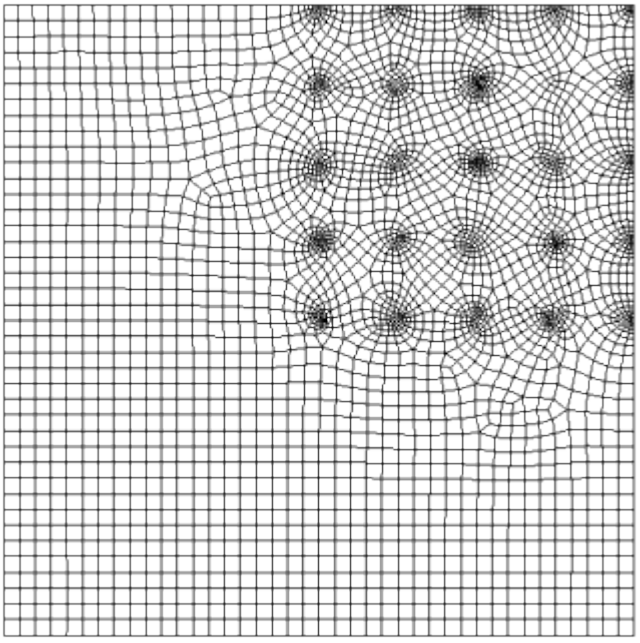


**Figure 2. Test sizing function for square geometry**

## Transition

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

**Applies to:** Surfaces

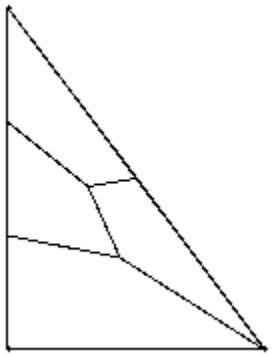
**Summary:** Produces a specified transition mesh for specific situations

**Syntax:**

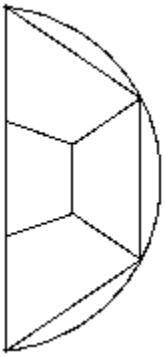
**Surface <range> Scheme Transition {Triangle|Half_circle|Three_to_one|Two_to_one|Convex_corner|Four_to_two} [Source Curve <id>] [Source Vertex <id>]**

**Discussion**:

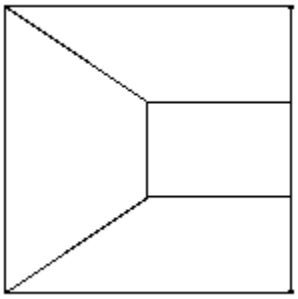
The *transition* scheme supplies a set of transition primitives which serve to transition a mesh from one density to another across a given surface. The six transition sub-types are demonstrated here.



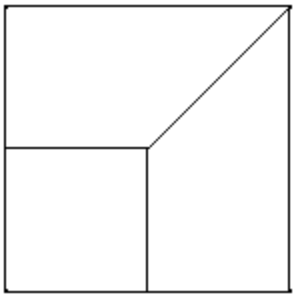
Scheme Transition **Triangle** creates four quads in a triangle that has sides of three, two, and one intervals.



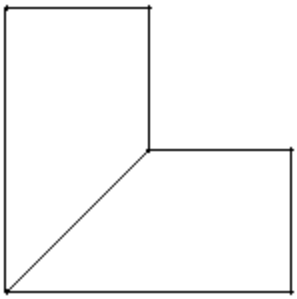
Scheme Transition **Half_circle** creates three intervals on the flat and three on the curved part of the half-circle, then creates four quads in the surface.



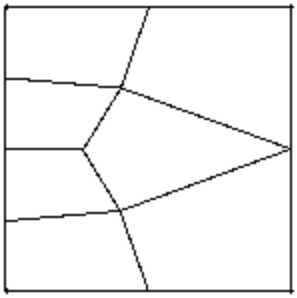
Scheme Transition **Three_to_one** creates four quads on a rectangular surface that has intervals of three, one, one, and one on its sides.



Scheme Transition **Two_to_one** creates three quads on a rectangular surface that has intervals of two, two, one and one on its sides :



Scheme Transition **Convex_corner** takes a six-sided block with a convex corner and connects that inner corner to the opposite one, creating two quads on the surface.



Scheme Transition **Four_to_two** creates seven quads on a rectangular surface that has intervals of four, two, two, and two on its sides.

The user also has the option of specifying a source curve and/or a source vertex. The rules for these specifications are as follows

- If both a curve and vertex are specified, the vertex must be on the curve.
- The Convex_corner sub-type does not allow a source curve.
- The Four_to_two sub-type does not allow a source vertex.
- The source curve will be the curve that will be given the fewest intervals.
- The source vertex will specify which corner will be used for the scheme, in cases where this makes sense (primarily in the Triangle, and Two_to_one cases).
- If none of the optional information is given, the program will assign the source curve to be the shortest one on the face, in keeping with the most probable

## Triangle Mesh Coarsening

**Note:** This feature is under development. The command to enable or disable features under development is:

**Set Developer Commands {On|OFF}**

CUBIT provides the capability for coarsening triangle surface meshes. Triangle coarsening uses a technique known as edge collapsing to coarsen a mesh. With this technique, triangle edges are selectively eliminated from the mesh until the specified criteria have been met. The following commands will coarsen an existing triangle surface mesh:

**Coarsen {Node|Edge|Tri} <range> {Factor|Size <double> [Bias <double>]} [Depth <int>|Radius <double>] [Sizing_Function] [no_smooth]**

**Coarsen {Vertex|Curve|Surface} <range> {Factor|Size<double> [Bias<double>]} [Depth<int>|Radius<double>] [Sizing_Function] [no_smooth]**

**Important**: These commands are currently implemented only for *triangle* shaped elements.
To use these commands, first select mesh or geometric entities at which you would like to perform coarsening. Coarsening operations will be applied to all mesh entities associated with or within proximity of the entities. The **all** keyword may be used to uniformly coarsen all triangles in the model.
Following is a description of each of the coarsen options:
**Factor**
Defines the approximate size relative to the existing edge lengths for which the coarsening will be applied. For example, a factor of 2 will attempt to make every edge length within the specified region approximately twice the size. A factor of 3 will make everything three times the size. Valid input values for factor must be greater than 1. Figure 1 shows an example where a coarsening factor of 2 was applied

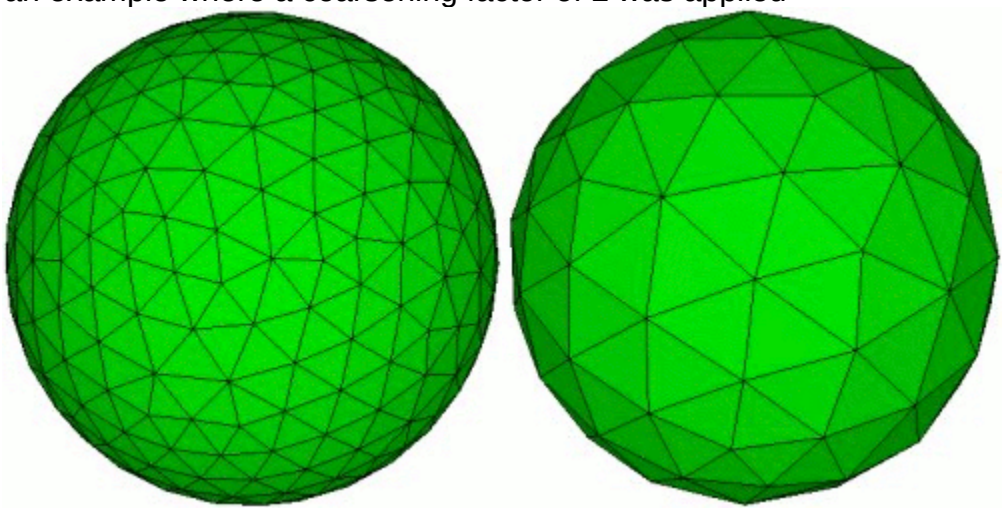


**Figure 1. Example of coarsening all triangles with a factor of 2.**

**Size, Bias**
The Size and Bias options are useful when a specific element size is desired at a known location. This might be used for locally coarsening around a vertex or curve. The Bias argument can be used with the Size option to define the rate at which the element sizes will change to meet the existing element sizes on the model. Valid input values for Bias

are greater than 1.0 and represent the maximum change in element size from one element to the next. Since coarsening is a discrete operation, the Size and Bias options can only approximate the desired input values. This may cause apparent discontinuities in the element sizes. Using the default smooth option can lessen this effect. It should also be noted that the Size option is exclusive of the Factor option. Either Factor or Size can be specified, but not both.

**Depth**

The Depth option permits the user to specify how many elements away from the specified entity will also be coarsened. Default Depth is 1.

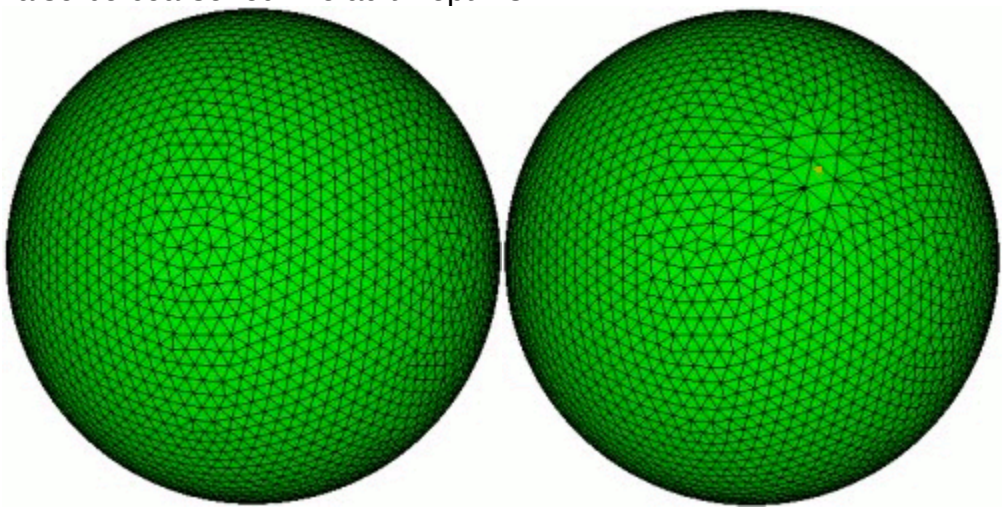


**Figure 2. Coarsening performed at a node with factor = 3 and depth = 3**

**Radius**

Instead of specifying the number of elements to describe how far to propagate the coarsening, a real Radius may be entered.

**Sizing Function**

Coarsening may also be controlled by a sizing function. CUBIT uses sizing functions to control the local density of a mesh. Various options for setting up a sizing function are provided, including importing scalar field data from an exodus file. In order to use this option, a sizing function must first be specified on the surface on which the coarsening will be applied. See Adaptive Meshing for a description of how to define a sizing function.

**No_Smooth**

The default mode for coarsening operations is to perform smoothing after coarsening the elements. This will generally provide better quality elements. In some cases it may be necessary to retain the original node locations after coarsening. The no_smooth option provides this capability.

## Higher Order Element Metrics

**Note:** This feature is under development.

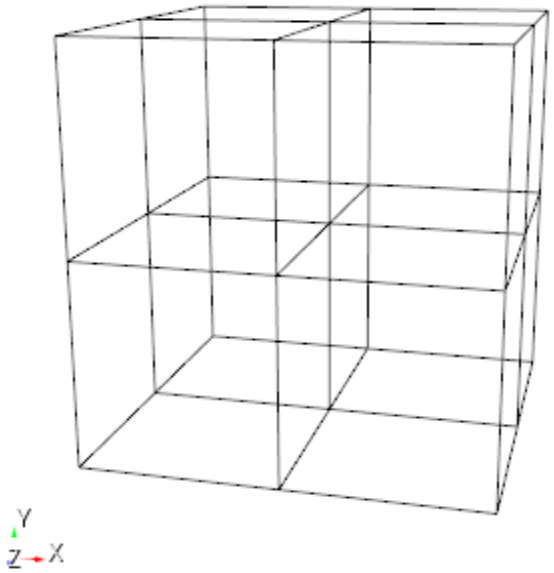
Cubit contains an **Node Distance** and **Altitude** quality metric for higher order elements. Currently, HEX27, WEDGE21 and TET15 are supported. The metrics are designed to help identify small distances which limit the analysis time step.

These metrics may be accessed by using the commands:

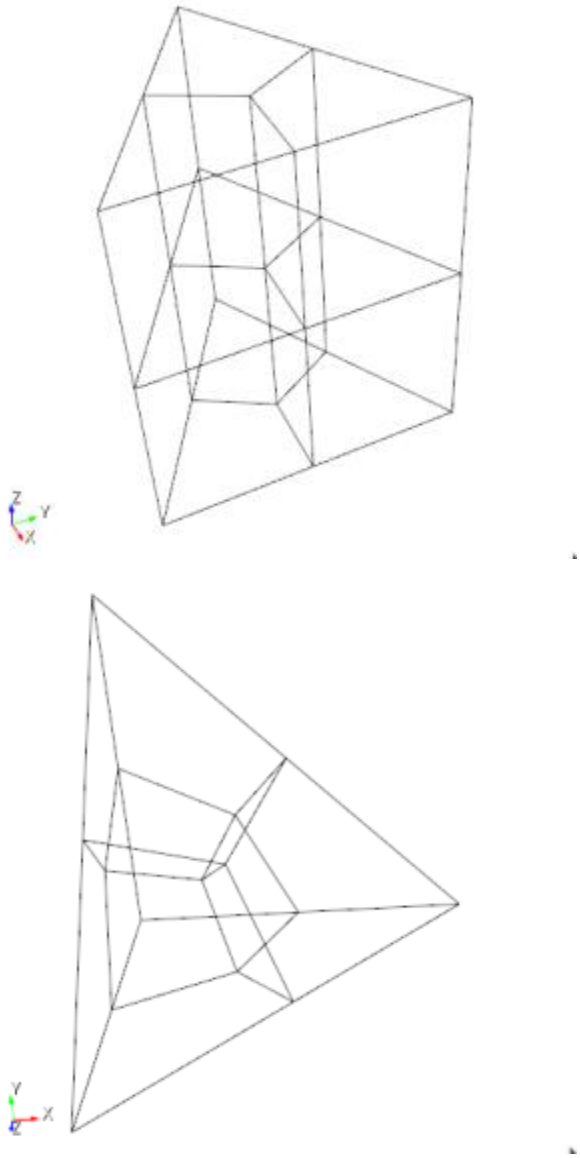
**Quality <element list> Node Distance**

**Quality <element list> Altitude**

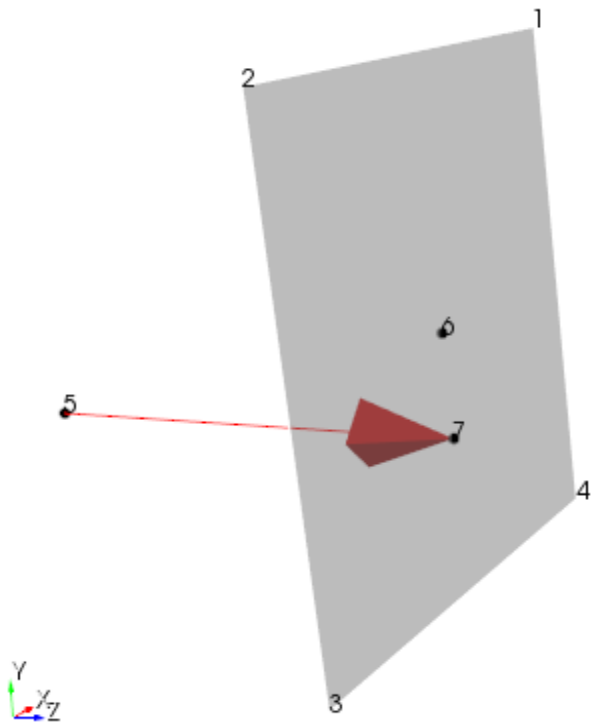
The **Node Distance** quality metric for a single element is the minimum distance between nodes within that element. Not every node is compared against every other node within an element, rather, a structured approach is used. For a single HEX27 element, the element can be subdivided using the 27 nodes to form 8 logical hexes with 8 nodes each. The edge lengths of the logical hexes are the distances considered for the HEX27 element. See the image below for a representation of the edges used to compute node distance in a single HEX27 element.




Additionally, the representation of edges used for a single WEDGE21 and TET15 element are given in the following images.




The **Altitude** quality metric for a single element is the minimum distance between the body center node and the projection of the body center node to the faces. Specifically, for a HEX27, the body center node is projected to each of the 6 faces of the element, and the minimum distance between the body center node and the 6 projections is the altitude metric for that element. The face projection is based on the linear approximation of the higher order face. The image below shows an example of the distance calculated when a body center node is projected to a face.




# aprepro

## APREPRO

**W**ithin CUBIT there is support for a programming language called *APREPRO* (An Algebraic Preprocessor for Parameterizing Finite Element Analyses). In addition to the standard APREPRO functionality, CUBIT extends the language with its own functions to aid in the meshing process. Included here is a summary of the CUBIT-specific APREPRO functionality. For a description of the APREPRO language and its usage, please refer to the APREPRO user's manual (PDF).

- Using APREPRO in CUBIT
- APREPRO Functions
- APREPRO Journaling

Note: APREPRO variables can be created and modified from the GUI. Enable/disable the editor from the View/Aprepro editor menu option. The editor is a docking window and can be placed anywhere in the GUI.

### Using APREPRO in CUBIT

To use APREPRO within CUBIT, simply enclose APREPRO statements within curly braces '**{}**' as part of the CUBIT command. Any APREPRO statements included in a command will be evaluated before the command is executed. For example, if the APREPRO variable 'my_x' is given the value of 3, the command

**brick x {my_x}**

will become

**brick x 3**

before the command is executed by CUBIT. Note that this means APREPRO will NOT give CUBIT parametric modeling abilities. In the above example, if the value of 'my_x' is later changed to 5, the size of the brick already created will not automatically change to five.
APREPRO expressions can also exist on separate lines. When doing this, it is recommended to add the CUBIT comment character '#' before the APREPRO statement. This will tell CUBIT to treat the evaluated expression as a comment, which will prevent errors from being issued in many cases.
Consider the following example:

**#{my_x = 3}**

**#{my_y = my_x + 2}**

**#{if(my_y < my_x)}**

**brick x {my_x} y {my_y}**

**#{else}**

**create cylinder radius {my_x} height {my_y}**

**#{endif}**

In the first two lines, only APREPRO statements are being executed (values are assigned to the variables 'my_x' and 'my_y'). After being evaluated by APREPRO, these two lines will be sent to CUBIT as

#3
#5

If the comment character was omitted instead CUBIT would issue several errors about incorrect command syntax. However, because these lines start with the comment character, they are ignored by CUBIT. Also note that the character '$' may be used in place of '#' for comments.

*Loops*

Repeated processing of a group of lines can be controlled with the **{loop(control)}**, **{endloop}** commands, as noted in section 6.2.5 of the APREPRO documentation.
A loop may also be terminated before running the specified number of times using a **#{break}** statement. As soon as a **#{break}** statement is encountered, the loop is exited and the rest of the statements in the loop will not execute. Additional iterations of the loop will not be executed either.
For example, the following commands will create 3 bricks:

```
#{_x = 1}
#{loop(10)}
  brick x 1
  #{if(_x == 2)}
    #{break}
  #{endif}
  #{_x++}
  pause
  brick x 1
#{endloop}
```

When a **#{break}** statement executes, anything in the loop following the **#{break}** statement will be skipped, including the **#{endif}**. For this reason, a **#{break}** statement not only exits the loops, but also terminates the most recent **#{if}** statement exactly as **#{endif}** would do. **#{break}** statements should not be used outside of **#{if}** statements.
It is also possible to terminate a loop using the **#{abortloop}** statement. **#{abortloop}** will terminate all loops (including nested loops) without executing the contents of the loop(s). This can be useful when a typo is made while manually entering a loop at the command line. Instead of ending the loop normally and waiting for the loop to execute with numerous errors, the loop will end immediately without any execution or errors. Please note, however, that the **#{abortloop}** statement is only valid within a loop block; otherwise, it will generate errors.
When creating a loop, APREPRO will record all lines that are given to the command line until the corresponding **#{endloop}** is reached. During this process, no commands will be passed to CUBIT. Once the terminating **#{endloop}** is reached, APREPRO will expand the loop, repeating the recorded lines the number of times specified by the loop counter, and send the expanded list of commands to CUBIT. If the terminating **#{endloop}** is accidentally omitted, CUBIT may appear to be unresponsive to commands because APREPRO is still recording lines for the loop. In situations like these, the **#{abortloop}** statement may be used to terminate any unfinished loops and restore the command line to a working state.
Also note that it is not recommended to use the '**pause**' command within a loop, as it can lead to situations in which the user must repeatedly enter the command '**resume**' to execute the entire loop. In situations like these **#{abortloop}** will NOT terminate the loop because it has already been expanded by APREPRO and CUBIT is simply executing a list of commands.

*Deleting APREPRO Variables*

There are two ways to delete an APREPRO variable in CUBIT. The first is to use the APREPRO '**delete**' function. The delete function takes the name of the variable to be deleted as its argument, as shown in the following example:

**#{my_var = 2}**

...

**#{delete('my_var')}**

The second way to delete an APREPRO variable is by using the '**reset aprepro**' command:

**#{my_var = 2}**

**#{some_var = 3}**

...

**reset aprepro**

This will delete all APREPRO variables and reset APREPRO to its initial state.

*Other Examples*

The following example shows the use of some of the string functions.

**#{t1 = "ATAN2"}{t2="(0,-1)"}**

**#{t3 = tolower(t1 // t2)}**

... The variable t3 is equal to the string atan2(0,-1)

**#{execute(t3)}**

...t3 = 3.141592654

The result is the same as executing {atan2(0,-1)} This is admittedly a very contrived example; however, it does illustrate the workings of several of the functions. In the first example, an expression is constructed by concatenating two strings together and converting the resulting string to lowercase. This string is then executed.

The following example uses the **rescan** function to illustrate a basic macro capability in APREPRO. The example creates vertices in CUBIT equally spaced about the circumference of a 180 degree arc of radius 10. Note that the macro is 5 lines long (2 of

the lines start with #, with the exception of the looping constructs - the actual journal file for this would not continue lines but would put each one on one long line).

```
#{num = 0} {rad = 10} {nintv = 10} {nloop = nintv + 1}

#{line = 'Create Vertex {polarX(rad,(++num-1)*180/nintv)} {polarY(rad,(num-1)*180/nintv)}'}

#{loop(nloop)}

{rescan(line)}

#{endloop}
```

Note the loop construct to automatically repeat the **rescan** line. To modify this example to calculate the coordinates of 101 points rather than eleven, the only change necessary would be to set {nintv=100}.

## APREPRO Functions

CUBIT adds a number of APREPRO functions to aid in the meshing process. A description of each function is available in the categories below.

- Geometry Query Functions
- Mesh Query Functions
- Group, Block, and Assembly Functions
- Id Functions
- Miscellaneous Functions
- Pre-defined Variables

***Table 1. Geometry Functions***

| Syntax | Description |
|---|---|
| **BBox_XMin("type", id)** | Returns the xmin value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **BBox_XMax("type", id)** | Returns the xmax value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the |

| | |
|---|---|
| | geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **BBox_YMin("type", id)** | Returns the ymin value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **BBox_YMax("type", id)** | Returns the ymax value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **BBox_ZMin("type", id)** | Returns the zmin value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **BBox_ZMax("type", id)** | Returns the zmax value of the bounding box of the specified geometric entity. “type” can be “volume”, “surface”, “curve”, “vertex”, or “group”. If “volume”, “surface”, “curve”, or “vertex” it will calculate the bounding box for the entity with the given id. If “group” it will calculate the combined bounding box for the group. A group can have any of the geometry types (vol, surf, curve, vert) in it and can be of mixed types. |
| **GeomCentroid_X("type", id)** | Returns the x coordinate of the centroid of the specified geometric entity. "type" can be "volume" or "group". If “volume” it calculates the centroid for the volume with the given id (single volume). If “group” it must be a group of volumes and it will calculate the combined centroid for the whole group with the given id. |
| **GeomCentroid_Y("type", id)** | Returns the y coordinate of the centroid of the specified geometric entity. "type" can be "volume" or "group". If “volume” it calculates the centroid for the volume with the given id (single volume). If “group” it must be a group of |

| | |
|---|---|
| | volumes and it will calculate the combined centroid for the whole group with the given id. |
| **GeomCentroid_Z("type", id)** | Returns the z coordinate of the centroid of the specified geometric entity. "type" can be "volume" or "group". If “volume” it calculates the centroid for the volume with the given id (single volume). If “group” it must be a group of volumes and it will calculate the combined centroid for the whole group with the given id. |
| **Length(id)** | Returns the length of the curve with the given id. |
| **Length(x, y, z, ord)** | Returns the length of the curve identified by the given center point coordinates and ordinal value. |
| **NumCurves()** | Returns the number of curves in the model. |
| **NumSurfaces()** | Returns the number of surfaces in the model. |
| **NumVertices()** | Returns the number of vertices in the model. |
| **NumVolumes()** | Returns the number of volumes in the model. |
| **Radius(id)** | Returns the radius of the curve at its midpoint. |
| **Radius(x, y, z, ord)** | Returns the radius of the curve identified by the given center point coordinates and ordinal value. |
| **SurfaceArea(id)** | Returns the surface area of the surface with the given id. |
| **SurfaceArea(x, y, z, ord)** | Returns the surface area of the surface identified by the given center point coordinates and ordinal value. |
| **Type("entity name")** | Returns the type of the specified entity name |
| **Volume(id)** | Gets the geometric volume of the volume with the given id. |
| **Volume(x, y, z, ord)** | Gets the geometric volume of the volume identified by the given center point coordinates and ordinal value. |
| **Vx(id), Vy(id), Vz(id)** | Gets the x, y or z coordinate of vertex with the given id. |
| **Vx(x, y, z, ord) Vy(x, y, z, ord) Vz(x, y, z, ord)** | Gets the x, y or z coordinate of vertex identified by the given center point coordinates and ordinal value. |
| **VertexAt(x, y, z, ordinal)** | Returns the id of the vertex with the idless reference, x,y,z,ordinal. |
| **CurveAt(x, y, z, ordinal)** | Returns the id of the curve with the idless reference, x,y,z,ordinal. |
| **SurfaceAt(x, y, z, ordinal)** | Returns the id of the surface with the idless reference, x,y,z,ordinal. |
| **VolumeAt(x, y, z, ordinal)** | Returns the id of the volume with the idless reference, x,y,z,ordinal. |

***Table 2. Mesh Functions***

| Syntax | Description |
|---|---|
| **EdgeLength(id)** | Returns the length of the edge with the given id. |
| **EdgeLength(x, y, z, ord)** | Returns the length of the edge identified by the given center point coordinates and ordinal value. |
| **FaceArea(id)** | Returns the area of the face with the given id. |
| **FaceArea(x, y, z, ord)** | Returns the area of the face identified by the given center point coordinates and ordinal value. |
| **HexVolume(id)** | Returns the volume of the hex with the given id. |
| **HexVolume(x, y, z, ord)** | Returns the volume of the hex identified by the given center point coordinates and ordinal value. |
| **IntNum(id)** | Returns the number of intervals on a curve with the given id. |
| **IntNum(x, y, z, ord)** | Returns the number of intervals on a curve identified by the given center point coordinates and ordinal value. |
| **IntSize(id)** | Returns the interval size on a curve with the given id. |
| **IntSize(x, y, z, ord)** | Returns the interval size on a curve identified by the given center point coordinates and ordinal value. |
| **MeshCentroid_X("type", id)** | Returns the x coordinate of the centroid of the specified mesh entity. “type” can be “volume”, “block”, or “group”. If “volume” it calculates the centroid of the 3D elements in the volume with the given id. If “block” it calculates the centroid of the elements in the block with the given id. If “group” it must be a group of volumes and it calculates the centroid of the group with the given id. |
| **MeshCentroid_Y("type", id)** | Returns the y coordinate of the centroid of the specified mesh entity. “type” can be “volume”, “block”, or “group”. If “volume” it calculates the centroid of the 3D elements in the volume with the given id. If “block” it calculates the centroid of the elements in the block with the given id. If “group” it must be a group of volumes and it calculates the centroid of the group with the given id. |
| **MeshCentroid_Z("type", id)** | Returns the z coordinate of the centroid of the specified mesh entity. “type” can be “volume”, “block”, or “group”. If “volume” it calculates the centroid of the 3D elements in the volume with the given id. If “block” it calculates the centroid of the elements in the block with the given id. If “group” it must be a group of volumes and it calculates the centroid of the group with the given id. |
| **MeshLength(id)** | Gets the length of the meshed curve with the given id. |
| **MeshLength(x, y, z, ord)** | Gets the length of the meshed curve identified by the given center point coordinates and ordinal value. |

| | |
|---|---|
| **MeshSurfaceArea(id)** | Returns the total area of all triangle or quadrilateral elements on the surface with the given id. This will vary from the geometric surface area since the mesh approximates the boundary with linear mesh edges. |
| **MeshSurfaceArea(x, y, z, ord)** | Returns the total area of all triangle or quadrilateral elements on the surface identified by the given center point coordinates and ordinal value. This will vary from the geometric surface area since the mesh approximates the boundary with linear mesh edges. |
| **MeshVolume(id)** | Returns the total volume of all mesh elements in the volume with the given id. This will vary from the actual geometric volume since the mesh approximates curved boundaries with linear mesh edges. |
| **MeshVolume(x, y, z, ord)** | Returns the total volume of all mesh elements in the volume identified by the given center point coordinates and ordinal value. This will vary from the actual geometric volume since the mesh approximates curved boundaries with linear mesh edges. |
| **MinSurfaceMeshQuality(id, "metric")** | Returns the worst value of the specified element quality metric of all elements on the given surface.<br><br>Acceptable metrics include:<br>shape<br>aspect ratio<br>condition no<br>distortion<br>element area<br>jacobian<br>maximum angle<br>minimum angle<br>relative size<br>scaled jacobian<br>shape and size<br>shear and size<br>shear<br>skew<br>stretch<br>taper<br>warpage |
| **MinSurfaceMeshQuality(x, y, z, ord, "metric")** | Returns the worst value of the specified element quality metric of all elements on the surface identified by the given center point coordinates and ordinal value. |

| | | | |
|---|---|---|---|
| | Acceptable | metrics | include: |
| | shape | | |
| | aspect | | ratio |
| | condition | | no |
| | distortion | | |
| | element | | area |
| | jacobian | | |
| | maximum | | angle |
| | minimum | | angle |
| | relative | | size |
| | scaled | | jacobian |
| | shape | and | size |
| | shear | and | size |
| | shear | | |
| | skew | | |
| | stretch | | |
| | taper | | |
| | warpage | | |
| **MinVolumeMeshQuality(id, "metric")** | Returns the worst value of the specified element quality metric of all elements in the volume with the given id. | | |
| | Acceptable | metrics | include: |
| | shape | | |
| | aspect | ration | bet |
| | aspect | ratio | gam |
| | aspect | | ratio |
| | condition | | no |
| | diagonal | | ratio |
| | dimension | | |
| | distortion | | |
| | element | | volume |
| | jacobian | | |
| | mass | increase | ratio |
| | mean | | ratio |
| | inradius | | |
| | node | | distance |
| | normalized | | inradius |
| | relative | | size |
| | scaled | | jacobian |
| | shape | and | size |
| | shear | and | size |
| | shear | | |
| | skew | | |
| | stretch | | |

| | |
|---|---|
| | taper<br>timestep |
| **MinVolumeMeshQuality(x, y, z, ord, "metric")** | Returns the worst value of the specified element quality metric of all elements in the volume identified by the given center point coordinates and ordinal value.<br><br>Acceptable metrics include:<br>shape<br>aspect ration bet<br>aspect ratio gam<br>aspect ratio<br>condition no<br>diagonal ratio<br>dimension<br>distortion<br>element volume<br>jacobian<br>mass increase ratio<br>mean ratio<br>inradius<br>node distance<br>normalized inradius<br>relative size<br>scaled jacobian<br>shape and size<br>shear and size<br>shear<br>skew<br>stretch<br>taper<br>timestep |
| **NumEdgesOnCurve(id)** | Returns the number of edges on the curve with the given id. |
| **NumEdgesOnCurve(x, y, z, ord)** | Returns the number of edges on the curve identified by the given center point coordinates and ordinal value. |
| **NumElemsOnSurface(id)** | Returns the number of elements on the surface with the given id. |
| **NumElemsOnSurface(x, y, z, ord)** | Returns the number of elements on the surface identified by the given center point coordinates and ordinal value. |
| **NumElemsInVolume(id)** | Returns the number of elements in the volume with the given id. |
| **NumElemsInVolume(x, y, z, ord)** | Returns the number of elements in the volume identified by the given center point coordinates and ordinal value. |

| | |
|---|---|
| **Nx(id), Ny(id), Nz(id)** | Gets the x, y or z coordinate of node with the given id. |
| **Nx(x, y, z, ord) Ny(x, y, z, ord) Nz(x, y, z, ord)** | Gets the x, y or z coordinate of node identified by the given center point coordinates and ordinal value. |
| **TetVolume(id)** | Returns the volume of the tet with the given id. |
| **TetVolume(x, y, z, ord)** | Returns the volume of the tet identified by the given center point coordinates and ordinal value. |
| **TriArea(id)** | Returns the area of the tri with the given id. |
| **TriArea(x, y, z, ord)** | Returns the area of the tri identified by the given center point coordinates and ordinal value. . |

***Table 3. Group, Block, and Assemblyl Metadata Functions***

| Syntax | Description |
|---|---|
| **BlockAttributeName(id, index)** | Returns the name for the specified attribute index in the block within the given id |
| **BlockAttributeValue(id, index)** | Returns the value for the specified attribute index in the block within the given id |
| **NumBlocks()** | Returns the number of blocks in the model. |
| **NumSidesets()** | Returns the number of sidesets in the model. |
| **NumNodesets()** | Returns the number of nodesets in the model. |
| **NumInGrp("groupname")** | Returns the number of entities in the given group. |
| **NumTypeInGroup("group_name", "entity_type")** | Returns the number of "entity_type" in group "group_name". |
| **NumVolsInPart("part_name")** | Returns the number of volumes assigned to the part with the specified name. |
| **PartInVol(id)** | Returns the name and instance number of the part that the volume has been assigned to. |

***Table 4. ID Functions***

| Syntax | Description |
|---|---|
| **CurveAt(x, y, z, ordinal)** | Returns the id of the curve with the idless reference, x,y,z,ordinal. |
| **EdgeAt(x, y, z, ordinal)** | Returns the id of the edge with the idless reference, x,y,z,ordinal. |
| **FaceAt(x, y, z, ordinal)** | Returns the id of the quad face with the idless reference, x,y,z,ordinal. |

| | |
|---|---|
| **GroupMemberId("group_name", "entity_type", index)** | Returns the ID of "entity_type" in group "group_name" at the specified index. If the group contains multiple entity types the index will only be relevant for the entity type specified and will behave as if the group only contained that entity type. |
| **HexAt(x, y, z, ordinal)** | Returns the id of the hexahedra element with the idless reference, x,y,z,ordinal. |
| **Id("type")** | Returns the ID of the entity most recently created with the specified type. Acceptable types include: "body", "volume", "surface", "curve", "vertex", "group", "node", "edge", "quad", "face", "tri", "hex", "tet", or "pyramid". |
| **NodeAt(x, y, z)** | Returns the id of the node closest to the xyz location. |
| **NodeAt(x, y, z, ordinal)** | Returns the id of the Node with the idless reference, x,y,z,ordinal. |
| **PyramidAt(x, y, z, ordinal)** | Returns the id of the pyramid element with the idless reference, x,y,z,ordinal. |
| **SurfaceAt(x, y, z, ordinal)** | Returns the id of the surface with the idless reference, x,y,z,ordinal. |
| **TetAt(x, y, z, ordinal)** | Returns the id of the tetraheral element with the idless reference, x,y,z,ordinal. |
| **TriAt(x, y, z, ordinal)** | Returns the id of the triangle with the idless reference, x,y,z,ordinal. |
| **VertexAt(x, y, z, ordinal)** | Returns the id of the vertex with the idless reference, x,y,z,ordinal. |
| **VolumeAt(x, y, z, ordinal)** | Returns the id of the volume with the idless reference, x,y,z,ordinal. |
| **WedgeAt(x, y, z, ordinal)** | Returns the id of the wedge element with the idless reference, x,y,z,ordinal. |

*Table 5. Miscellaneous Functions*

| **Syntax** | **Description** |
|---|---|
| **FileExists("file name")** | Checks if the given file exists. Returns non-zero if true. |
| **GeometryEngineVersion("engine name")** | Get the version for the specified geometry engine. |
| **get_error_count()** | Returns the current error count in CUBIT |
| **get_warning_count()** | Returns the current warning count in CUBIT |

| | |
|---|---|
| **HasFeature("feature name")** | Checks if the specified feature is available. Returns non-zero if the feature is enabled. |
| **Print(msg)** | Prints msg |
| **PrintError(svar)** | Outputs the string **svar** to stderr. |
| **Quote(svar)** | Returns the string **svar**, enclosed in single quotes. |
| **SessionId()** | Returns the unique id for the current CUBIT session. |
| **GetMachineType()** | Returns the string name of the platform Cubit is running on ("Linux", "Darwin", "Microsoft Windows", etc.). |
| **set_error_count(val)** | Sets the error count in CUBIT to val |
| **set_warning_count(val)** | Sets the warning count in CUBIT to val |
| **TimerStart()** | Starts the CPU timer |
| **TimerStop()** | Stops the CPU timer |

*Table 6. Pre-defined Variables*

The following APREPRO variables are predefined in CUBIT.

| Variable | Description |
|---|---|
| **CUBIT** | Variable to indicate that CUBIT is defined |
| **CUBIT_VERSION** | Current version of CUBIT (not to be confused with **VERSION**, which stores the current version of APREPRO) |

## APREPRO Journaling

When using APREPRO, statements can be echoed to a journal file. To do so, use the following command:

```
[set] Journal [Graphics|Names|Aprepro|Errors] [on|off]
```

Simply typing "journal aprepro" without an argument will display the current aprepro journaling setting.
For example,

```
bri x {2*5.0}
```

is journaled as

```
brick x {2*5.0}
```

if aprepro journaling is ON, or

```
brick x 10
```

if aprepro journaling is off. The default is **ON**.

*APREPRO Comments*

Comments are also journaled. This is useful for documenting aprepro definitions and descriptions.
Comments on the same line as a command get split into two separate lines in the journal file.

*Significant Figures*

When journal aprepro is **ON**, numbers are journaled exactly as they are entered. The maximum number of significant digits is determined by the command input.
When journal aprepro is **off**, numeric results of aprepro statements are journaled according to the maximum number of significant digits hard-coded into CUBIT, using the value of **DBL_DIG**.

*Loops and Journaling*

Loops are not journaled as loops, per se. For example, the APREPRO expression:

```
{loop(3)}
    bri x {x}
{endloop}
```

is journaled as:

```
bri x {x}
bri x {x}
bri x {x}
```

*Multi-line Strings*

Multi-line strings are currently not journaled (both definitions and when they are expanded). For example,

```
#{line = 'bri x 10
mesh vol 1'}
{line}
```

will be journaled as

```
bri x 10
mesh vol 1
```

Note that **bri x 10\n mesh vol 1** was not journaled as **{line}**

# python

## Python Interface

The following Python functions and objects provide capability to query and modify Cubit models.

*Functions*

CubitInterface - Cubit model query and modify functions.

*Classes*

Entity - The base class of all the geometry and mesh types.
GeomEntity - The base class for specifically the Geometry types (Body, Surface, etc.).
Body - Defines a body object that mostly parallels Cubit's Body class.
Volume - Defines a volume object that mostly parallels Cubit's RefVolume class
Surface - Defines a surface object that mostly parallels Cubit's RefFace class.
Curve - Defines a curve object that mostly parallels Cubit's RefEdge class.
Vertex - Defines a vertex object that mostly parallels Cubit's RefVertex class.
CFD BC Interface -Defines the interface to CFD Boundary Condition entities
Direction Interface -Defines the interface to a Direction object
Location Interface -Defines the interface to a Location object
CubitFailureException - An exception class to alert the caller when the underlying Cubit function fails.
InvalidEntityException - An exception class to alert the caller that an invalid entity was attempted to be used.
InvalidInputException - An exception class to alert the caller of a function that invalid inputs were entered.
MeshError - Mesh error interface
MeshImport - Mesh import interface
MeshModify - Mesh modify interface
AssemblyItem - AssemblyItem interface

## Importing Cubit into Python

Python users are able to import Cubit into Python and make calls into Cubit via CubitInterface and the other Python classes described in this section. Below is a simple Python script. The key parts are ensuring the Cubit libraries are on the path and ensuring the cubit.init() call is made first.

```
import sys

# add Cubit libraries to your path
sys.path.append('/opt/cubit/bin')

import cubit

#start cubit - this step is key
#cubit.init does not require any arguments.
```

```
#If you do want to provide arguments, you must
#provide 2 or more, where the first must
#be "cubit", and user args start as the 2nd argument.
#If only one argument is used, it will be ignored.
cubit.init(['cubit','-nojournal'])

height = 1.2
blockHexRadius = 0.1732628

#hexagon
baseBlock = cubit.prism(height, 6, blockHexRadius, blockHexRadius)

#etc . . .
```

## Python Cubit Enhancement Scripts

The Python-Cubit enhancement code base is intended to be used as an extension to already existing Cubit functionality. It provides the user with a number of functionalities that are either currently outside the realm of the python functions which cubit supplies internally (such as vector math), or that are comprised of commonly used combinations of already existing python functionalities.
Python Cubit Enhancement Scripts

Cubit Python API 16.18

Classes

# CubitInterface Namespace Reference

The **CubitInterface** provides a Python/C++ interface into Cubit. More...

## Classes

class **AssemblyItem**

Class to implement assembly tree interface. More...

class **Body**

Defines a body object that mostly parallels Cubit's **Body** class. More...

class **CFD_BC_Entity**

Class to implement cfd bc data retrieval. More...

class **CubitFailureException**

An exception class to alert the caller when the underlying Cubit function fails. More...

class **Curve**

Defines a curve object that mostly parallels Cubit's RefEdge class. More...

class **Dir**

Defines a direction object. More...

class **Entity**

The base class of all the geometry and mesh types. More...

class **GeomEntity**

The base class for specifically the Geometry types (**Body**, **Surface**, etc.) More...

class **InvalidEntityException**

An exception class to alert the caller that an invalid entity was attempted to be used. Likely the user is attempting to use an **Entity** who's underlying CubitEntity has been deleted. More...

class **InvalidInputException**

An exception class to alert the caller of a function that invalid inputs were entered. More...

class **Loc**

Defines a location object. More...

class **MeshErrorFeedback**

Class to implement mesh command feedback processing. More...

class **Surface**

Defines a surface object that mostly parallels Cubit's RefFace class. More...

class **Vertex**

Defines a vertex object that mostly parallels Cubit's RefVertex class. More...

class **Volume**

Defines a volume object that mostly parallels Cubit's RefVolume class. More...

## Functions

### System Control and Data

| | |
|---|---|
| void | **set_progress_handler** (CubitProgressHandler *progress)<br>Register a progress-bar callback handler with Cubit. Deletes the current progress handler if it exists. |
| CubitProgressHandler * | **replace_progress_handler** (CubitProgressHandler *progress)<br>Register a new progress-bar callback handler with Cubit and return the the previous progress-handler without deleting it. |
| void | **set_cubit_interrupt** (bool interrupt)<br>This sets the global flag in Cubit that stops all interruptable processes. |
| void | **set_playback_paused_on_error** (bool pause)<br>Sets whether or not playback is paused when an error occurs. |
| bool | **is_playback_paused_on_error** ()<br>Gets whether or not playback is paused when an error occurs. |
| void | **pause_playback** ()<br>Pause playback. |
| void | **stop_playback** ()<br>Pause playback. |

| | |
|---|---|
| void | **resume_playback** ()<br>resume playback. |
| bool | **is_playback_paused** () |
| bool | **developer_commands_are_enabled** ()<br>This checks to see whether developer commands are enabled. |
| CubitBaseInterface * | **get_interface** (std::string interface_name)<br>Get the interface of a given name. |
| bool | **release_interface** (CubitBaseInterface *instance)<br>Release the interface with the given name. |
| CubitPluginManager * | **plugin_manager** () |
| void | **add_filename_to_recent_file_list** (std::string &filename)<br>Adds the filename to the recent file list. |
| std::string | **get_version** ()<br>Get the Cubit version. |
| std::string | **get_revision_date** ()<br>Get the Cubit revision date. |
| std::string | **get_build_number** ()<br>Get the Cubit build number. |
| std::string | **get_acis_version** ()<br>Get the Acis version number. |
| int | **get_acis_version_as_int** () |

| | |
|---|---|
| | Get the Acis version number as an int. |
| std::string | **get_exodus_version** ()<br>Get the Exodus version number. |
| std::string | **get_meshgems_version** ()<br>Get the MeshGems version number. |
| double | **get_cubit_digits_setting** ()<br>Get the Cubit digits setting. |
| std::string | **get_graphics_version** ()<br>Get the VTK version number. |
| std::string | **get_python_version** ()<br>get the python version used in cubit |
| void | **print_cmd_options** ()<br>Used to print the command line options. |
| bool | **is_modified** ()<br>Get the modified status of the model. |
| void | **set_modified** ()<br>Set the status of the model (**is_modified()** is now false). If you modify the model after you set this flag, it will register true. |
| bool | **is_undo_save_needed** ()<br>Get the status of the model relative to undo checkpointing. |
| void | **set_undo_saved** ()<br>Set the status of the model relative to undo checkpointin. |

| | |
|---|---|
| bool | **is_performing_undo** ()<br>Check if an undo command is currently being performed. |
| bool | **is_command_echoed** ()<br>Check the echo flag in cubit. |
| std::string | **get_command_from_history** (int command_number)<br>Get a specific command from Cubit's command history buffer. |
| int | **get_command_history_count** () |
| std::string | **get_next_command_from_history** ()<br>Get 'next' command from history buffer. |
| std::string | **get_previous_command_from_history** ()<br>Get 'previous' command from history buffer. |
| bool | **is_volume_meshable** (int volume_id)<br>Check if volume is meshable with current scheme. |
| bool | **is_surface_meshable** (int surface_id)<br>Check if surface is meshable with current scheme. |
| void | **journal_commands** (bool state)<br>Set the journaling flag in cubit. |
| bool | **is_command_journaled** ()<br>Check the journaling flag in cubit. |
| void | **write_to_journal** (std::string words)<br>Write a string to the active journal. |

| | |
|---|---|
| void | **override_journal_stream** (JournalStreamBase *jnl_stream)<br>Override the Journal Stream in CUBIT. |
| std::string | **get_current_journal_file** ()<br>Gets the current journal file name. |
| bool | **is_working_dir_set** ()<br>Create BCVizInterface for CompSimUI. |
| bool | **cmd** (const char *input_string)<br>Pass a command string into Cubit. |
| bool | **cmd_single** (const char *input_string)<br>Pass a command string into Cubit. |
| bool | **cubit_or_python_cmd** (const std::string &input_string) |
| bool | **cubit_or_python_cmds** (const std::vector< std::string > &input_strings, std::function< void(const std::string &)> &&history) |
| bool | **cubit_or_python_cmds_as_file** (const std::vector< std::string > &input_strings, std::function< void(const std::string &)> &&history) |
| bool | **command_is_python_mode** ()<br>Gets whether command handling is in python mode. |
| bool | **silent_cmd** (const char *input_string)<br>Pass a command string into Cubit and have it executed without being verbose at the command prompt. |

| | |
|---|---|
| bool | **was_last_cmd_undoable** ()<br>Report whether the last executed command was undoable. |
| std::vector< int > | **parse_cubit_list** (const std::string &type, std::string entity_list_string)<br>Parse a Cubit style entity list into a list of integers. |
| std::vector< std::array< double, 3 > > | **parse_locations** (const std::string &location_str) |
| std::string | **string_from_id_list** (std::vector< int > ids)<br>Parse a list of integers into a Cubit style id list. Includes carriage return and line breaks at column 80. |
| std::string | **get_id_string** (const std::vector< int > &entity_ids)<br>Parse a list of integers into a Cubit style id list. Return string will not include carriage returns or line break. |
| void | **print_raw_help** (const char *input_line, int order_dependent, int consecutive_dependent)<br>Used to print out help when a ?, & or ! is pressed. |
| int | **get_error_count** ()<br>Get the number of errors in the current Cubit session. |
| std::vector< std::string > | **get_mesh_error_solutions** (int error_code)<br>Get the paired list of mesh error solutions and help context cues. |
| void | **complete_filename** (std::string &line, int &num_chars, bool &found_quote) |

| | |
|---|---|
| | Get the file completion inside a quote based on files in the current directory. This handles completion of directories as well as filtering on specific types (.jou, .g, .sat, etc.) |

**Graphics Manipulation and Data**

| | |
|---|---|
| double | **get_view_distance** ()<br>Get the distance from the camera to the model (from - at) |
| std::array< double, 3 > | **get_view_at** ()<br>Get the camera 'at' point. |
| std::array< double, 3 > | **get_view_from** ()<br>Get the camera 'from' point. |
| std::array< double, 3 > | **get_view_up** ()<br>Get the camera 'up' direction. |
| void | **reset_camera** ()<br>reset the camera in all open windows this includes resetting the view, closing the histogram and color windows and clearing the scalar bar, highlight, and picked entities. |
| void | **flush_graphics** ()<br>Flush the graphics. |
| void | **clear_drawing_set** (const std::string &set_name)<br>Clear a named drawing set (this is for mesh preview) |
| void | **unselect_entity** (const std::string &entity_type, int entity_id)<br>Unselect an entity that is currently selected. |

| | |
|---|---|
| int | **get_rubberband_shape** ()<br>Get the current rubberband select mode. |
| bool | **is_perspective_on** ()<br>Get the current perspective mode. |
| bool | **is_occlusion_on** ()<br>Get the current occlusion mode. |
| bool | **is_scale_visibility_on** ()<br>Get the current scale visibility setting. |
| bool | **is_mesh_visibility_on** ()<br>Get the current mesh visibility setting. |
| bool | **is_geometry_visibility_on** ()<br>Get the current geometry visibility setting. |
| bool | **is_select_partial_on** ()<br>Get the current select partial setting. |
| int | **get_rendering_mode** ()<br>Get the current rendering mode. |
| void | **set_rendering_mode** (int mode)<br>Set the current rendering mode. |
| void | **clear_highlight** ()<br>Clear all entity highlights. |
| void | **clear_preview** ()<br>Clear preview graphics without affecting other display settings. |

| | |
|---|---|
| void | **highlight** (const std::string &entity_type, int entity_id)<br>Highlight the given entity. |
| std::vector< int > | **get_selected_ids** ()<br>Get a list of the currently selected ids. |
| int | **get_selected_id** (int index)<br>Get the selected id based on an index. |
| std::string | **get_selected_type** (int index)<br>Get the selected type based on an index. |
| const char * | **get_pick_type** ()<br>Get the current pick type. |
| void | **set_pick_type** (const std::string &pick_type, bool silent=false)<br>Set the pick type. |
| void | **set_filter_types** (int num_types, const std::vector< std::string > filter_types)<br>Set the pick filter types. |
| void | **add_filter_type** (const std::string &filter_type)<br>Add a filter type. |
| void | **remove_filter_type** (const std::string &filter_type)<br>Remove a filter type. |
| bool | **is_type_filtered** (const std::string &filter_type)<br>Determine whether a type is filtered. |

| | |
|---|---|
| std::vector< std::string > | **get_pick_filters** ()<br>Get a list of the current pick filters. |
| void | **clear_picked_list** ()<br>Clear the picked list. |
| void | **step_next_possible_selection** ()<br>Step to the next possible selection (selected next dialog) |
| void | **step_previous_possible_selection** ()<br>Step to the previous possible selection (selected next dialog) |
| void | **print_current_selections** ()<br>Print the current selections. |
| void | **print_currently_selected_entity** ()<br>Print the current selection. |
| int | **current_selection_count** ()<br>Get the current count of selected items. |

**Mesh Query Support**

| | |
|---|---|
| double | **get_mesh_edge_length** (int edge_id)<br>Get the length of a mesh edge. |
| double | **get_meshed_volume_or_area** (const std::string &geometry_type, std::vector< int > entity_ids)<br>Get the total volume/area of a entity's mesh. |
| int | **get_mesh_intervals** (const std::string &geometry_type, int entity_id)<br>Get the interval count for a specified entity. |

| | |
|---|---|
| double | **get_mesh_size** (const std::string &geometry_type, int entity_id)<br>Get the mesh size for a specified entity. |
| double | **get_requested_mesh_size** (const std::string &geometry_type, int id)<br>Get the requested mesh size for a specified entity. This returns a size that has been set specifically on the entity and not averaged from parents. |
| int | **has_valid_size** (const std::string &geometry_type, int entity_id)<br>Get whether an entity has a size. All entities have a size unless the auto sizing is off. If the auto sizing is off, an entity has a size only if it has been set. |
| bool | **auto_size_needs_to_be_calculated** ()<br>Get whether the auto size needs to be calculated. Calculating the auto size may be expensive on complex models. The auto size may be outdated if the model has changed. |
| double | **get_default_auto_size** ()<br>Get auto size needs for the current set of geometry. |
| int | **get_requested_mesh_intervals** (const std::string &geometry_type, int entity_id)<br>Get the interval count for a specified entity as set specifically on that entity. |
| double | **get_auto_size** (const std::string &geometry_type, std::vector< int > entity_id_list, double size)<br>Get the auto size for a given set of enitities. Note, this does not actually set the interval size on the volumes. It simply returns the |

| | |
|---|---|
| | size that would be set if an 'size auto factor n' command were issued. |
| int | **get_element_budget** (const std::string &element_type, std::vector< int > entity_id_list, int auto_factor)<br>Get the element budget based on current size settings for a list of volumes. |
| std::string | **get_exodus_sizing_function_variable_name** ()<br>Get the exodus sizing function variable name. |
| std::string | **get_exodus_sizing_function_file_name** ()<br>Get the exodus sizing function file name. |
| std::string | **get_sizing_function_name** (const std::string &entity_type, int surface_id)<br>Get the sizing function name for a surface or volume. |
| bool | **exodus_sizing_function_file_exists** ()<br>return whether the exodus sizing funnction file exists |
| bool | **get_vol_sphere_params** (std::vector< int > sphere_id_list, int &rad_intervals, int &az_intervals, double &bias, double &fract, int &max_smooth_iterations)<br>get the current sphere parameters for a sphere volume |
| std::string | **get_curve_bias_type** (int curve_id) |
| double | **get_curve_bias_geometric_factor** (int curve_id) |

| | |
|---|---|
| double | **get_curve_bias_geometric_factor2** (int curve_id) |
| double | **get_curve_bias_first_interval_length** (int curve_id) |
| double | **get_curve_bias_first_interval_fraction** (int curve_id) |
| double | **get_curve_bias_fine_size** (int curve_id) |
| double | **get_curve_bias_coarse_size** (int curve_id) |
| double | **get_curve_bias_first_last_ratio1** (int curve_id) |
| double | **get_curve_bias_first_last_ratio2** (int curve_id) |
| double | **get_curve_bias_last_first_ratio1** (int curve_id) |
| double | **get_curve_bias_last_first_ratio2** (int curve_id) |
| bool | **get_curve_bias_from_start** (int curve_id, bool &value) |
| bool | **get_curve_bias_from_start_set** (int curve_id) |
| int | **get_curve_bias_start_vertex_id** (int curve_id) |
| double | **get_curve_mesh_scheme_curvature** (int curve_id)<br>Get the curvature mesh scheme value of a curve. |

| | |
|---|---|
| bool | **get_curve_mesh_scheme_stretch_values** (int curve_id, double &first_size, double &factor, double &last_size, bool &start, int &vertex_id) |
| std::vector< double > | **get_curve_mesh_scheme_pinpoint_locations** (int curve_id) |
| void | **get_quality_stats** (const std::string &entity_type, std::vector< int > id_list, const std::string &metric_name, double single_threshold, bool use_low_threshold, double low_threshold, double high_threshold, double &min_value, double &max_value, double &mean_value, double &std_value, int &min_element_id, int &max_element_id, std::vector< int > &mesh_list, std::string &element_type, int &bad_group_id, bool make_group=false)<br>Get the quality stats for a specified entity. |
| std::vector< double > | **get_elem_quality_stats** (const std::string &entity_type, const std::vector< int > id_list, const std::string &metric_name, const double single_threshold, const bool use_low_threshold, const double low_threshold, const double high_threshold, const bool make_group)<br>python callable version of the get_quality_stats without pass by reference arguments. All return values are stuffed into a double array |
| std::vector< double > | **get_quality_stats_at_geometry** (const std::string &geom_type, const std::string &mesh_type, const std::vector< int > geom_id_list, const int expand_levels, const std::string &metric_name, const double single_threshold, const bool use_low_threshold, const double low_threshold, const double high_threshold, const bool make_group) |

| | |
|---|---|
| | get element quality at a list of geometry entities. Finds all elements with nodes ON/IN the specified geometry and finds the quality of all elements of the specfied element type that are connected. Same arguments and return values as get_elem_quality_stats except a geometry and element type are used as arguments |
| double | **get_quality_value** (const std::string &mesh_type, int mesh_id, const std::string &metric_name)<br>Get the metric value for a specified mesh entity. |
| std::vector< double > | **get_quality_values** (const std::string &mesh_type, std::vector< int > mesh_ids, const std::string &metric_name)<br>Get the metric values for specified mesh entities. |
| std::string | **get_mesh_scheme** (const std::string &geometry_type, int entity_id)<br>Get the mesh scheme for the specified entity. |
| std::string | **get_mesh_scheme_firmness** (const std::string &geometry_type, int entity_id)<br>Get the mesh scheme firmness for the specified entity. |
| std::string | **get_mesh_interval_firmness** (const std::string &geometry_type, int entity_id)<br>Get the mesh interval firmness for the specified entity. This may include influence from connected mesh intervals on connected geometry. |
| std::string | **get_requested_mesh_interval_firmness** (const std::string &geometry_type, int entity_id) |

| | |
|---|---|
| | Get the mesh interval firmness for the specified entity as set specifically on the entity. |
| std::string | **get_mesh_size_type** (const std::string &geometry_type, int entity_id)<br>Get the mesh size setting type for the specified entity. This may include influence from attached geometry. |
| std::string | **get_requested_mesh_size_type** (const std::string &geometry_type, int entity_id)<br>Get the mesh size setting type for the specified entity as set specifically on the entity. |
| bool | **get_tetmesh_proximity_flag** (int volume_id)<br>Get the proximity flag for tet meshing. |
| int | **get_tetmesh_proximity_layers** (int volume_id)<br>Get the number of proximity layers for tet meshing. This is the number of layers between close surfaces. |
| double | **get_tetmesh_growth_factor** (int volume_id)<br>Get the tetmesh growth factor. |
| bool | **get_tetmesh_parallel** ()<br>Get the parallel flag for tet meshing. Defines whether to use parallel mesher. |
| int | **get_tetmesh_num_anisotropic_layers** ()<br>Get the number of anisotropic tet layers. Global setting. |
| int | **get_tetmesh_optimization_level** () |

| | |
|---|---|
| | Get the optimization level for tetmeshing. Global setting. |
| bool | **get_tetmesh_insert_mid_nodes** ()<br>Get the state of the flag to insert midnodes during meshing. Global setting. |
| bool | **get_tetmesh_optimize_mid_nodes** ()<br>Get the state of the flag to optimize midnodes during meshing. Global setting. |
| bool | **get_tetmesh_optimize_overconstrained_tets** ()<br>Get the state of the flag to optimize overconstrained tets. Global setting. |
| bool | **get_tetmesh_optimize_overconstrained_edges** ()<br>Get the state of the flag to optimize overconstrained edges. Global setting. |
| bool | **get_tetmesh_minimize_slivers** ()<br>Get the state of the flag to minimize sliver tets. Global setting. |
| bool | **get_tetmesh_minimize_interior_points** ()<br>Get the state of the flag to minimize interior points in tetmesher. Global setting. |
| bool | **get_tetmesh_relax_surface_constraints** ()<br>Get the state of the flag to relax surface mesh constraints in tetmesher. Global setting. |
| double | **get_mesh_geometry_approximation_angle** (std::string geometry_type, int entity_id)<br>Get the geometry approximation angle set for tri/tet meshing. |

| | |
|---|---|
| double | **get_trimesh_surface_gradation** ()<br>Get the global surface mesh gradation set for meshing with MeshGems. |
| double | **get_trimesh_volume_gradation** ()<br>Get the global volume mesh gradation set for meshing with MeshGems. |
| bool | **get_trimesh_use_surface_proximity** ()<br>Get the global trimesh surface proximity setting with MeshGems. |
| double | **get_trimesh_surface_proximity_ratio** ()<br>Get the global trimesh surface proximity max aspect ratio setting with MeshGems. |
| double | **get_trimesh_target_min_size** (std::string geom_type, int entity_id)<br>Get the trimesh target min size for the entity. local setting for surfaces. |
| bool | **get_trimesh_geometry_sizing** ()<br>Get the global geometry sizing flag for trimesher. |
| int | **get_trimesh_num_anisotropic_layers** ()<br>Get the global number of anisotropic layers for trimeshing. |
| bool | **get_trimesh_split_overconstrained_edges** ()<br>Get the global setting for trimesher split over-constrained edges. |
| int | **best_edge_to_collapse_interior_node** (int node_id) |

| | |
|---|---|
| | Finds the best edge to collapse this node along to remove the interior node. |
| double | **get_trimesh_tiny_edge_length** ()<br>Get the global setting for tiny edge length in trimesher. |
| double | **get_trimesh_ridge_angle** ()<br>Get the global setting for ridge angle in trimesher. |
| bool | **is_meshed** (const std::string &geometry_type, int entity_id)<br>Determines whether a specified entity is meshed. |
| bool | **is_merged** (const std::string &geometry_type, int entity_id)<br>Determines whether a specified entity is merged. |
| std::string | **get_smooth_scheme** (const std::string &geometry_type, int entity_id)<br>Get the smooth scheme for a specified entity. |
| int | **get_hex_count** ()<br>Get the count of hexes in the model. |
| int | **get_pyramid_count** ()<br>Get the count of pyramids in the model. |
| int | **get_tet_count** ()<br>Get the count of tets in the model. |
| int | **get_quad_count** ()<br>Get the count of quads in the model. |

| | |
|---|---|
| int | **get_tri_count** ()<br>Get the count of tris in the model. |
| int | **get_edge_count** ()<br>Get the count of edges in the model. |
| int | **get_sphere_count** ()<br>Get the count of sphere elements in the model. |
| int | **get_wedge_count** ()<br>Get the count of wedge elements in the model. |
| int | **get_node_count** ()<br>Get the count of nodes in the model. |
| int | **get_element_count** ()<br>Get the count of elements in the model. |
| int | **get_volume_element_count** (int volume_id)<br>Get the count of elements in a volume. |
| int | **get_surface_element_count** (int surface_id)<br>Get the count of elements in a surface. |
| bool | **volume_contains_tets** (int volume_id)<br>Determine whether a volume contains tets. |
| std::vector< int > | **get_hex_sheet** (int node_id_1, int node_id_2)<br>Get the list of hex elements forming a hex sheet through the given two node ids. The |

| | |
|---|---|
| | nodes must be adjacent in the connectivity of the hex i.e. they form an edge of the hex. |
| std::string | **get_default_element_type** ()<br>Get the current default setting for the element type that will be used when meshing. |

**Geometry Query Support**

| | |
|---|---|
| bool | **is_visible** (const std::string &geometry_type, int entity_id)<br>Query visibility for a specific entity. |
| bool | **is_virtual** (const std::string &geometry_type, int entity_id)<br>Query virtualality for a specific entity. |
| bool | **contains_virtual** (const std::string &geometry_type, int entity_id)<br>Query virtualality of an entity's children. |
| std::vector< int > | **get_source_surfaces** (int volume_id)<br>Get a list of a volume's sweep source surfaces. |
| std::vector< int > | **get_target_surfaces** (int volume_id)<br>Get a list of a volume's sweep target surfaces. |
| int | **get_common_curve_id** (int surface_1_id, int surface_2_id)<br>Given 2 surfaces, get the common curve id. |
| int | **get_common_vertex_id** (int curve_1_id, int curve_2_id)<br>Given 2 curves, get the common vertex id. |

| | |
|---|---|
| std::vector< std::vector< double > > | **project_unit_square** (std::vector< std::vector< double > > pts, int surface_id, int quad_id, int node00_id, int node10_id)<br>Given points in a unit square, map them to the given quad using the orientation info, then project them onto the given surface, and return their projected positions. |
| std::string | **get_merge_setting** (const std::string &geometry_type, int entity_id)<br>Get the merge setting for a specified entity. |
| std::string | **get_curve_type** (int curve_id)<br>Get the curve type for a specified curve. |
| std::string | **get_surface_type** (int surface_id)<br>Get the surface type for a specified surface. |
| std::array< double, 3 > | **get_surface_normal** (int surface_id)<br>Get the surface normal for a specified surface. |
| std::array< double, 3 > | **get_surface_normal_at_coord** (int surface_id, std::array< double, 3 >)<br>Get the surface normal for a specified surface at a location. |
| std::array< double, 3 > | **get_surface_centroid** (int surface_id)<br>Get the surface centroid for a specified surface. |
| std::string | **get_surface_sense** (int surface_id)<br>Get the surface sense for a specified surface. |
| std::vector< std::string > | **get_entity_modeler_engine** (const std::string &geometry_type, int entity_id) |

| | |
|---|---|
| | Get the modeler engine type for a specified entity. |
| std::string | **get_default_geometry_engine** ()<br>Get the name of the default modeler engine. |
| std::array< double, 10 > | **get_bounding_box** (const std::string &geometry_type, int entity_id)<br>Get the bounding box for a specified entity. |
| std::array< double, 10 > | **get_total_bounding_box** (const std::string &geometry_type, std::vector< int > entity_list)<br>Get the bounding box for a list of entities. |
| std::array< double, 15 > | **get_tight_bounding_box** (const std::string &geometry_type, std::vector< int > entity_list)<br>Get the tight bounding box for a list of entities. |
| double | **get_total_volume** (std::vector< int > volume_list)<br>Get the total volume for a list of volume ids. |
| std::string | **get_entity_name** (const std::string &entity_type, int entity_id, bool no_default=false)<br>Get the name of a specified entity. |
| std::vector< std::string > | **get_entity_names** (const std::string &entity_type, int entity_id, bool no_default=false, bool first_name_only=false)<br>same as get_entity_name but includes all name attributes set on the entity, not just the first one (unless first_name_only is set) |

| | |
|---|---|
| bool | **set_entity_name** (const std::string &entity_type, int entity_id, const std::string &new_name)<br>Set the name of a specified entity. |
| std::array< double, 4 > | **get_entity_color** (const std::string &entity_type, int entity_id)<br>Get the color of a specified entity. |
| int | **get_entity_color_index** (const std::string &entity_type, int entity_id) |
| bool | **is_multi_volume** (int body_id)<br>Query whether a specified body is a multi volume body. |
| bool | **is_sheet_body** (int volume_id)<br>Query whether a specified volume is a sheet body. |
| bool | **is_interval_count_odd** (int surface_id)<br>Query whether a specified surface has an odd loop. |
| bool | **is_periodic** (const std::string &geometry_type, int entity_id)<br>Query whether a specified surface or curve is periodic. |
| bool | **is_surface_planer** (int surface_id)<br>Query whether a specified surface is planer. |
| bool | **is_surface_planar** (int surface_id) |
| void | **get_periodic_data** (const std::string &geometry_type, int entity_id, double &returned_interval, std::string &returned_firmness, int |

| | |
|---|---|
| | &returned_lower_bound, std::string &returned_upper_bound)<br>Get the periodic data for a surface or curve. |
| bool | **get_undo_enabled** ()<br>Query whether undo is currently enabled. |
| int | **number_undo_commands** ()<br>Query whether there are any undo commands to execute. |
| std::vector< std::string > | **get_aprepro_vars** ()<br>Gets the current aprepro variable names. |
| std::string | **get_aprepro_value_as_string** (std::string variable_name)<br>Gets the string value of an aprepro variable. |
| bool | **get_aprepro_value** (std::string variable_name, int &returned_variable_type, double &returned_double_val, std::string &returned_string_val)<br>Get the value of an aprepro variable. |
| double | **get_aprepro_numeric_value** (std::string variable_name)<br>get the value of the given aprepro variable |
| bool | **get_node_constraint** ()<br>Query current setting for node constraint (move nodes to geometry) |
| int | **get_node_constraint_value** ()<br>Query current setting for node constraint (move nodes to geometry) |
| double | **get_node_constraint_smart_threshold** () |

| | |
|---|---|
| | Query current setting for node constraint smart threshold. |
| std::string | **get_node_constraint_smart_metric** ()<br>Query current setting for node constraint smart metric Currently only for tets. Return either "distortion" of "normalized inradius". |
| std::string | **get_vertex_type** (int surface_id, int vertex_id)<br>Get the **Vertex** Types for a specified vertex on a specified surface. **Vertex** types include "side", "end", "reverse", "unknown". |
| std::vector< int > | **get_relatives** (const std::string &source_geometry_type, int source_id, const std::string &target_geom_type)<br>Get the relatives (parents/children) of a specified entity. |
| std::vector< int > | **get_adjacent_surfaces** (const std::string &geometry_type, int entity_id)<br>Get a list of adjacent surfaces to a specified entity. |
| std::vector< int > | **get_adjacent_volumes** (const std::string &geometry_type, int entity_id)<br>Get a list of adjacent volumes to a specified entity. |
| std::vector< int > | **get_entities** (const std::string &entity_type)<br>Get all entities of a specified type (including geometry, mesh, etc...) |
| std::vector< int > | **get_list_of_free_ref_entities** (const std::string &geometry_type)<br>Get all free entities of a given geometry type. |

| | |
|---|---|
| int | **get_owning_body** (const std::string &geometry_type, int entity_id)<br>Get the owning body for a specified entity. |
| int | **get_owning_volume** (const std::string &geometry_type, int entity_id)<br>Get the owning volume for a specified entity. |
| int | **get_owning_volume_by_name** (const std::string &entity_name)<br>Get the owning volume for a specified entity. |
| double | **get_curve_length** (int curve_id)<br>Get the length of a specified curve. |
| double | **get_arc_length** (int curve_id)<br>Get the arc length of a specified curve. |
| double | **get_distance_from_curve_start** (double x_coordinate, double y_coordinate, double z_coordinate, int curve_id)<br>Get the distance from a point on a curve to the curve's start point. |
| double | **get_curve_radius** (int curve_id)<br>Get the radius of a specified arc. |
| std::array< double, 3 > | **get_curve_center** (int curve_id)<br>Get the center point of the arc. |
| double | **get_surface_area** (int surface_id)<br>Get the area of a surface. |
| std::vector< int > | **get_similar_curves** (std::vector< int > curve_ids, double tol=1e-3, bool use_percent_tol=true, bool on_similar_vols=true) |

| | |
|---|---|
| | Get similar curves with the same length. |
| std::vector< int > | **get_similar_surfaces** (std::vector< int > surface_ids, double tol=1e-3, bool use_percent_tol=true, bool on_similar_vols=true)<br>Get similar surfaces with the same area and number of curves. |
| std::vector< int > | **get_connected_surfaces** (std::vector< int > surf_ids)<br>Get a connected set of surfaces to the ones specified. |
| std::vector< int > | **get_similar_volumes** (std::vector< int > volume_ids, double tol=1e-3, bool use_percent_tol=true)<br>Get similar volumes with the same volume and number of faces. |
| std::vector< double > | **get_surface_principal_curvatures** (int surface_id)<br>Get the principal curvatures of a surface at surface mid_point. |
| double | **get_volume_area** (int volume_id)<br>Get the area of a volume. |
| double | **get_volume_volume** (int vol_id)<br>Get the volume of a volume. |
| int | **get_num_volume_shells** (int volume_id)<br>Get the number of shells in this volume. |
| double | **get_hydraulic_radius_surface_area** (int surface_id)<br>Get the area of a hydraulic surface. |

| | |
|---|---|
| double | **get_hydraulic_radius_volume_area** (int volume_id)<br>Get the area of a hydraulic volume. |
| std::array< double, 3 > | **get_center_point** (const std::string &entity_type, int entity_id)<br>Get the center point of a specified entity. |
| int | **get_valence** (int vertex_id)<br>Get the valence for a specific vertex. |
| double | **get_distance_between** (int vertex_id_1, int vertex_id_2)<br>Get the distance between two vertices. |
| double | **get_distance_between_entities** (std::string geom_type_1, int entity_id_1, std::string geom_type_2, int entity_id_2)<br>Get the distance between two geom entities. |
| int | **is_point_contained** (const std::string &geometry_type, int entity_id, const std::array< double, 3 > &xyz_point)<br>Determine if given point is inside, outside, on or unknown the given entity. note that this is typically used for volumes or sheet bodies. |
| void | **print_surface_summary_stats** ()<br>Print the surface summary stats to the console. |
| void | **print_volume_summary_stats** ()<br>Print the volume summary stats to the console. |
| int | **get_block_count** ()<br>Get the current number of blocks. |

| | |
|---|---|
| int | **get_sideset_count** ()<br>Get the current number of sidesets. |
| int | **get_nodeset_count** ()<br>Get the current number of sidesets. |
| int | **get_volume_count** ()<br>Get the current number of nodesets. |
| int | **get_body_count** ()<br>Get the current number of bodies. |
| int | **get_surface_count** ()<br>Get the current number of surfaces. |
| int | **get_vertex_count** ()<br>Get the current number of vertices. |
| int | **get_curve_count** ()<br>Get the current number of curves. |
| int | **get_curve_count_in_volumes** (std::vector< int > target_volume_ids)<br>Get the current number of curves in the passed-in volumes. |
| std::vector< int > | **get_current_ids** (const std::string &entity_type)<br>Get the current body ids. |
| bool | **is_catia_engine_available** ()<br>Determine whether catia engine is available. |
| bool | **is_acis_engine_available** () |
| bool | **is_opencascade_engine_available** () |

| | |
|---|---|
| std::vector< int > | **evaluate_exterior_angle** (const std::vector< int > &curve_list, const double test_angle)<br>find all curves in the given list with an exterior angle (the angle between surfaces) less than the test angle. This is equivalent to the df parser "exterior_angle" test. (draw curve with exterior_angle >90) |
| double | **evaluate_exterior_angle_at_curve** (int curve_id, int volume_id)<br>return exterior angle at a single curve with respect to a volume |
| double | **evaluate_surface_angle_at_vertex** (int surf_id, int vert_id)<br>return interior angle at a vertex on a specified surface |
| double | **get_overlap_max_gap** (void)<br>Get the max gap setting for calculating surface overlaps. |
| void | **set_overlap_max_gap** (const double maximum_gap)<br>Set the max gap setting for calculating surface overlaps. |
| double | **get_overlap_min_gap** (void)<br>Get the min gap setting for calculating surface overlaps. |
| void | **set_overlap_min_gap** (const double min_gap)<br>Set the min gap setting for calculating surface overlaps. |
| double | **get_overlap_max_angle** (void) |

| | |
|---|---|
| | Get the max angle setting for calculating surface overlaps. |
| void | **set_overlap_max_angle** (const double maximum_angle)<br>Set the max angle setting for calculating surface overlaps. |

**Geometry Repair Support**

| | |
|---|---|
| void | **get_small_surfaces_hydraulic_radius** (std::vector< int > target_volume_ids, double mesh_size, std::vector< int > &returned_small_surfaces, std::vector< double > &returned_small_radius)<br>Get the list of small hydraulic radius surfaces for a list of volumes. |
| std::vector< int > | **get_small_surfaces_HR** (std::vector< int > target_volume_ids, double mesh_size)<br>Python callable version Get the list of small hydraulic radius surfaces for a list of volumes. |
| void | **get_small_volumes_hydraulic_radius** (std::vector< int > target_volume_ids, double mesh_size, std::vector< int > &returned_small_volumes, std::vector< double > &returned_small_radius)<br>Get the list of small hydraulic radius volumes for a list of volumes. |
| std::vector< int > | **get_small_curves** (std::vector< int > target_volume_ids, double mesh_size)<br>Get the list of small curves for a list of volumes. |
| std::vector< int > | **get_smallest_curves** (std::vector< int > target_volume_ids, int number_to_return) |

| | |
|---|---|
| | Get a list of the smallest curves in the list of volumes. The number returned is specified by 'num_to_return'. |
| std::vector< int > | **get_small_surfaces** (std::vector< int > target_volume_ids, double mesh_size)<br>Get the list of small surfaces for a list of volumes. |
| bool | **is_narrow_surface** (int surface_id, double mesh_size)<br>return whether the surface is narrow (has a width smaller than mesh_size) |
| std::vector< int > | **get_narrow_surfaces** (std::vector< int > target_volume_ids, double mesh_size)<br>Get the list of narrow surfaces for a list of volumes. |
| std::vector< int > | **get_small_and_narrow_surfaces** (std::vector< int > target_ids, double small_area, double small_curve_size)<br>Get the list of small or narrow surfaces from a list of volumes. |
| std::vector< int > | **get_closed_narrow_surfaces** (std::vector< int > target_ids, double narrow_size)<br>Get the list of closed, narrow surfaces from a list of volumes. |
| std::vector< int > | **get_surfs_with_narrow_regions** (std::vector< int > target_ids, double narrow_size)<br>Get the list of surfaces with narrow regions. |
| std::vector< int > | **get_narrow_regions** (std::vector< int > target_ids, double narrow_size)<br>Get the list of surfaces with narrow regions. |

| | |
|---|---|
| std::vector< int > | **get_small_volumes** (std::vector< int > target_volume_ids, double mesh_size)<br>Get the list of small volumes from a list of volumes. |
| bool | **is_cylinder_surface** (int surface_id)<br>return whether the surface is a cylinder |
| bool | **is_chamfer_surface** (int surface_id, double thickness_threshold)<br>return whether the surface is a chamfer |
| std::vector< std::vector< double > > | **get_chamfer_surfaces** (std::vector< int > target_volume_ids, double thickness_threshold)<br>Get the list of chamfer surfaces for a list of volumes. |
| bool | **is_blend_surface** (int surface_id)<br>return whether the surface is a blend |
| std::vector< int > | **get_blend_surfaces** (std::vector< int > target_volume_ids)<br>Get the list of blend surfaces for a list of volumes. |
| std::vector< int > | **get_small_radius_blend_surfaces** (std::vector< int > target_volume_ids, double max_radius)<br>Get the list of blend surfaces for a list of volumes that have a radius of curvature smaller than max_radius. |
| bool | **is_close_loop_surface** (int surface_id, double mesh_size)<br>return whether the has one or more close loops |

| | |
|---|---|
| std::vector< int > | **get_close_loops** (std::vector< int > target_volume_ids, double mesh_size)<br>Get the list of close loops (surfaces) for a list of volumes. |
| std::vector< std::vector< double > > | **get_close_loops_with_thickness** (std::vector< int > target_volume_ids, double mesh_size, int genus)<br>Get the list of close loops (surfaces) for a list of volumes also return the corresponding minimum distances for each surface. |
| double | **get_close_loop_thickness** (int surface_id)<br>Get the thickness of a close loop surface. |
| std::vector< std::vector< std::string > > | **get_solutions_for_close_loop** (int surface_id, double mesh_size)<br>Get the solution list for a given close loop surface. |
| std::vector< int > | **get_tangential_intersections** (std::vector< int > target_volume_ids, double upper_bound, double lower_bound)<br>Get the list of bad tangential intersections for a list of volumes. |
| std::vector< int > | **get_coincident_vertices** (std::vector< int > target_volume_ids, double high_tolerance) |
| std::vector< int > | **get_close_vertex_curve_pairs** (std::vector< int > target_volume_ids, double high_tolerance)<br>Get the list of close vertex-curve pairs (python callable) |
| std::vector< std::vector< std::string > > | **get_solutions_for_near_coincident_vertices** (int vertex_id_1, int vertex_id_2)<br>Get lists of display strings and command strings for near coincident vertices. |

| | |
|---|---|
| std::vector< std::vector< std::string > > | **get_solutions_for_bad_geometry** (std::string geom_type, int geom_id)<br>Get lists of display strings and command strings for bad geometry. |
| std::vector< std::vector< std::string > > | **get_solutions_for_overlapping_volumes** (int volume_id_1, int volume_id_2, double maximum_gap_tolerance, double maximum_gap_angle)<br>Get lists of display strings and command strings for overlapping volumes. |
| std::vector< std::vector< std::string > > | **get_solutions_for_overlapping_surfaces** (int surface_id_1, int surface_id_2)<br>Get lists of display strings and command strings for overlapping surfaces. |
| std::vector< std::vector< std::string > > | **get_volume_gap_solutions** (int surface_id_1, int surface_id_2) |
| std::vector< std::vector< std::string > > | **get_solutions_for_near_coincident_vertex_and_curve** (int vertex_id, int curve_id)<br>Get lists of display strings and command strings for near coincident vertices and curves. |
| std::vector< std::vector< std::string > > | **get_solutions_for_near_coincident_vertex_and_surface** (int vertex_id, int surface_id)<br>Get lists of display strings and command strings for near coincident vertices and surfaces. |
| std::vector< std::vector< std::string > > | **get_solutions_for_imprint_merge** (int surface_id1, int surface_id2)<br>Get lists of display strings and command strings for imprint/merge solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_forced_sweepability** (int volume_id, std::vector< int > |

| | |
|---|---|
| | &source_surface_id_list, std::vector< int > &target_surface_id_list, double small_curve_size=-1.0)<br>**This function only works from C++** Get lists of display strings and command strings for forced sweepability solutions |
| std::vector< std::vector< std::string > > | **get_solutions_for_volumes** (int vol_id, double small_curve_size, double mesh_size)<br>Get lists of display, preview and command strings for small volume solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_classified_volume** (std::string classification, int vol_id)<br>Get lists of display, preview and command strings for a classified volume. |
| std::vector< std::vector< std::string > > | **get_solutions_for_bolt** (int bolt_id, int insert_id, int threaded_vol_id)<br>get the solutions for a volume classified as a bole. faster than get_solutions_for_classified_volume if the lower volume and insert volumes are alraedy known |
| std::vector< std::vector< std::string > > | **get_solutions_for_bolt_hole** (int bearing_hole, std::vector< int > threaded_holes)<br>get the solutions for a set of concentric holes used as a fastener pilot hole |
| std::vector< std::vector< std::string > > | **get_solutions_for_classified_surface** (std::string classification, int surf_id)<br>Get lists of display, preview and command strings for a classified surface. |
| std::vector< std::vector< std::string > > | **get_solutions_for_thin_volume** (int vol_id, std::vector< int > near_vols, bool include_weights=false, bool include_type=false) |

| | |
|---|---|
| | Get lists of display, preview and command strings for a volume to reduce to shell. |
| std::vector< std::vector< std::string > > | **get_solutions_for_sheet_volumes** (std::vector< int > vol_ids, std::vector< double > thickness)<br>Get lists of display, preview and command strings to connect sheet bodies. |
| std::vector< std::vector< std::string > > | **get_solutions_for_sheet_volume_connection** (std::vector< int > vol1_sheets, std::vector< int > vol2_sheets, double thickness1, double thickness2, std::string close_type="", int close_id=0)<br>Get lists of display, preview and command strings for two neighboring sheet volume sets. each set should be part of a common parent 3D volume. |
| std::vector< std::vector< std::string > > | **get_solutions_for_small_surfaces** (int surface_id, double small_curve_size, double mesh_size)<br>Get lists of display, preview and command strings for small surface solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_small_curves** (int curve_id, double small_curve_size, double mesh_size)<br>Get lists of display, preview and command strings for small curve solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_sharp_angle_vertex** (int vertex_id, double small_curve_size, double mesh_size)<br>Get lists of display, preview and command strings for sharp angle solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_surfaces_with_narrow_regions** (int surface_id, double small_curve_size, double mesh_size) |

| | |
|---|---|
| | Get lists of display, preview and command strings for surfaces with narrow regions solutions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_cone_surface** (int surface_id)<br>Get lists of display, preview and command strings for surfaces with defined as cones. |
| bool | **get_solutions_for_source_target** (int volume_id, std::vector< std::vector< int > > &feasible_source_surface_id_list, std::vector< std::vector< int > > &feasible_target_surface_id_list, std::vector< std::vector< int > > &infeasible_source_surface_id_list, std::vector< std::vector< int > > &infeasible_target_surface_id_list)<br>Get a list of suggested sources and target surface ids given a specified volume. |
| void | **get_sharp_surface_angles** (std::vector< int > target_volume_ids, std::vector< int > &returned_large_surface_angles, std::vector< int > &returned_small_surface_angles, std::vector< double > &returned_large_angles, std::vector< double > &returned_small_angles, double upper_bound, double lower_bound)<br>Get the list of sharp surface angles for a list of volumes. |
| void | **get_sharp_curve_angles** (std::vector< int > target_volume_ids, std::vector< int > &returned_large_curve_angles, std::vector< int > &returned_small_curve_angles, std::vector< double > &returned_large_angles, std::vector< double > &returned_small_angles, double upper_bound, double lower_bound) |

| | |
|---|---|
| | Get the list of sharp curve angles for a list of volumes. |
| std::vector< std::vector< double > > | **get_sharp_angle_vertices** (std::vector< int > target_volume_ids, double upper_bound, double lower_bound)<br>Get the list of vertices at sharp curve angles for a list of volumes returns two parallel arrays. First array are the vertex ids and second are the associated angles at the vertices. |
| bool | **is_cone_surface** (int surface_id)<br>return whether the surface is a cone |
| std::vector< int > | **get_cone_surfaces** (std::vector< int > target_volume_ids)<br>return a list of surfaces that are cones defined by a conic surface and a hard point |
| void | **get_bad_geometry** (std::vector< int > target_volume_ids, std::vector< int > &returned_body_list, std::vector< int > &returned_volume_list, std::vector< int > &returned_surface_list, std::vector< int > &returned_curve_list)<br>**This function only works from C++** Get the list of bad geometry for a list of volumes |
| std::vector< std::vector< int > > | **get_overlapping_surfaces_in_bodies** (std::vector< int > body_ids, bool filter_slivers=false)<br>returns a vector of vectors defining surface overlaps The first surface (id) in each vector overlaps with all subsequent surfaces in the vector. |
| void | **get_overlapping_surfaces_in_volumes** (std::vector< int > target_volume_ids, std::vector< int > &returned_surface_list_1, std::vector< int > &returned_surface_list_2, |

| | |
|---|---|
| | std::vector< double > &returned_distance_list, std::vector< double > &returned_overlap_area_list, bool filter_slivers=false, bool filter_volume_overlaps=false, int cache_overlaps=0)<br>**This function only works from C++** Get the list of overlapping surfaces for a list of volumes |
| void | **get_overlapping_surfaces** (std::vector< int > target_surface_ids, std::vector< int > &returned_surface_list_1, std::vector< int > &returned_surface_list_2, std::vector< double > &returned_distance_list, std::vector< double > &returned_overlap_area_list, bool filter_slivers=false, bool filter_volume_overlaps=false, int cache_overlaps=0)<br>**This function only works from C++** Get the list of overlapping surfaces for a list of surfaces |
| void | **get_overlapping_curves** (std::vector< int > target_surface_ids, double min_gap, double max_gap, std::vector< int > &returned_curve_list_1, std::vector< int > &returned_curve_list_2, std::vector< double > &returned_distance_list) |
| void | **get_volume_gaps** (std::vector< int > target_volume_ids, std::vector< int > &returned_surface_list_1, std::vector< int > &returned_surface_list_2, std::vector< double > &returned_distance_list, std::vector< double > &returned_overlap_area_list, double maximum_gap_tolerance, double maximum_gap_angle, int cache_overlaps=0)<br>**This function only works from C++** Get the list of gaps for a list of volumes |

| | |
|---|---|
| std::vector< VolumeGap > | **get_gaps_between_volumes** (std::vector< int > target_volume_ids, double maximum_gap_tolerance, double maximum_gap_angle, int cache_overlaps=0) |
| std::vector< int > | **get_overlapping_volumes** (std::vector< int > target_volume_ids)<br>Get the list of overlapping volumes for a list of volumes. |
| std::vector< int > | **get_overlapping_volumes_at_volume** (int volume_id, std::vector< int > compare_volumes)<br>Get the list of overlapping volumes from the model for a single volume. |
| std::vector< int > | **get_overlapping_surfaces_at_surface** (int surface_id, std::vector< int > compare_volumes, int cache_overlaps=0)<br>Get the list of overlapping surfaces from the model for a single surface. |
| std::vector< int > | **get_nearby_entities** (std::string gtype, std::vector< int > ent_ids, std::vector< int > compare_ents, double distance)<br>Get the list of nearby entities of type curve, surface or volumes from the model for a list of the same entity type. |
| std::vector< int > | **get_nearby_volumes_at_volume** (int volume_id, std::vector< int > compare_volumes, double distance)<br>Get the list of nearby volumes from the model for a single volume. |
| std::vector< int > | **get_unmerged_curves_on_shells** (std::vector< int > shell_vols, std::vector< double > thickness) |

| | |
|---|---|
| | return a list of curve IDs on the given shell volumes that are in proximity to one of the other shell volumes in the list |
| void | **get_mergeable_entities** (std::vector< int > target_volume_ids, std::vector< std::vector< int > > &returned_surface_list, std::vector< std::vector< int > > &returned_curve_list, std::vector< std::vector< int > > &returned_vertex_list, double merge_tol=-1)<br>**This function only works from C++** Get the list of mergeable entities from a list of volumes |
| std::vector< std::vector< int > > | **get_mergeable_vertices** (std::vector< int > target_volume_ids)<br>Get the list of mergeable vertices from a list of volumes/bodies. |
| std::vector< std::vector< int > > | **get_mergeable_curves** (std::vector< int > target_volume_ids)<br>Get the list of mergeable curves from a list of volumes/bodies. |
| std::vector< std::vector< int > > | **get_mergeable_surfaces** (std::vector< int > target_volume_ids)<br>Get the list of mergeable surfaces from a list of volumes/bodies. |
| void | **get_closest_vertex_curve_pairs** (std::vector< int > target_ids, int &returned_number_to_return, std::vector< int > &returned_vertex_ids, std::vector< int > &returned_curve_ids, std::vector< double > &returned_distances)<br>Find the n closest vertex pairs in the model. |
| void | **get_smallest_features** (std::vector< int > target_ids, int &returned_number_to_return, std::vector< int > &returned_type_1_list, std::vector< int > &returned_type_2_list, |

| | |
|---|---|
| | std::vector< int > &returned_id_1_list, std::vector< int > &returned_id_2_list, std::vector< double > &returned_distance_list)<br>Finds all of the smallest features. |
| double | **estimate_merge_tolerance** (std::vector< int > target_volume_ids, bool accurate_in=false, bool report_in=false, double low_value_in=-1.0, double high_value_in=-1.0, int number_calculations_in=10, bool return_calculations_in=false, std::vector< double > *merge_tolerance_list=NULL, std::vector< int > *number_of_proximities_list=NULL)<br>Estimate a good merge tolerance for the passed-in volumes. |
| void | **find_floating_volumes** (std::vector< int > target_volume_ids, std::vector< int > &returned_floating_id_list)<br>Get the list of volumes with no merged children. |
| void | **find_nonmanifold_curves** (std::vector< int > target_volume_ids, std::vector< int > &returned_curve_list)<br>Get the list of nonmanifold curves in the volume list. |
| void | **find_nonmanifold_vertices** (std::vector< int > target_volume_ids, std::vector< int > &returned_vertex_list)<br>Get the list of nonmanifold vertices in the volume list. |
| void | **get_coincident_entity_pairs** (std::vector< int > target_volume_ids, std::vector< int > &returned_v_v_vertex_list, std::vector< int > &returned_v_c_vertex_list, std::vector< int > |

| | |
|---|---|
| | &returned_v_c_curve_list, std::vector< int > &returned_v_s_vertex_list, std::vector< int > &returned_v_s_surf_list, std::vector< double > &returned_vertex_distance_list, std::vector< double > &returned_curve_distance_list, std::vector< double > &returned_surf_distance_list, double low_value, double high_value, bool do_vertex_vertex=true, bool do_vertex_curve=true, bool do_vertex_surf=true, bool filter_same_volume_cases=false)<br>Get the list of coincident vertex-vertex, vertex-curve, and vertex-surface pairs and distances from a list of volumes. |
| void | **get_coincident_vertex_vertex_pairs** (std::vector< int > target_volume_ids, std::vector< int > &returned_vertex_pair_list, std::vector< double > &returned_distance_list, double low_value, double threshold_value, bool filter_same_volume_cases=false)<br>Get the list of coincident vertex pairs and distances from a list of volumes. |
| void | **get_coincident_vertex_curve_pairs** (std::vector< int > target_volume_ids, std::vector< int > &returned_vertex_list, std::vector< int > &returned_curve_list, std::vector< double > &returned_distance_list, double low_value, double threshold_value, bool filter_same_volume_cases=false)<br>Get the list of coincident vertex/curve pairs and distances from a list of volumes. |
| void | **get_coincident_vertex_surface_pairs** (std::vector< int > target_volume_ids, std::vector< int > &returned_vertex_list, std::vector< int > &returned_surface_list, std::vector< double > &returned_distance_list, double low_value, |

| | |
|---|---|
| | double threshold_value, bool filter_same_volume_cases=false)<br>Get the list of coincident vertex/surface pairs and distances from a list of volumes. |
| std::vector< std::vector< std::string > > | **get_solutions_for_decomposition** (const std::vector< int > &volume_list, double exterior_angle, bool do_imprint_merge, bool tolerant_imprint)<br>Get the list of possible decompositions. |
| std::vector< std::vector< std::string > > | **get_solutions_for_blends** (int surface_id)<br>Get the solution list for a given blend surface. |
| std::vector< std::vector< int > > | **get_blend_chains** (int surface_id)<br>Returns the blend chains for a surface. |
| std::vector< std::vector< int > > | **get_chamfer_chains** (int surface_id)<br>Returns the chamfer chains for a surface. |
| bool | **are_adjacent_surfaces** (std::vector< int > surface_ids)<br>return whether two or more surfaces share at least one manifold curve (common curve is part of exactly two surfaces) |
| bool | **are_adjacent_curves** (std::vector< int > curve_ids)<br>return whether two or more curves share at least one manifold vertex (common vertex is part of exactly two curves) |
| bool | **is_continuous_surface** (int surface_id, double angle_tol)<br>return whether the surface has any adjacent surfaces that are continuous (exterior angle is 180 degrees +- angle_tol) |

| | |
|---|---|
| std::vector< int > | **get_continuous_surfaces** (int surface_id, double angle_tol)<br>Returns the adjacent surfaces that are continuous (exterior angle is 180 degrees +- angle_tol) |
| std::vector< int > | **get_continuous_curves** (int curve_id, double angle_tol)<br>Returns the adjacent curves that are continuous (angle is 180 degrees +- angle_tol) |
| bool | **is_cavity_surface** (int surface_id)<br>return whether the surface is part of a cavity |
| bool | **is_hole_surface** (int surface_id, double radius_threshold)<br>return whether the surface is part of a hole |
| std::vector< int > | **get_cavity_surfaces** (int surface_id)<br>Returns the adjacent surfaces in a cavity for a surface. |
| std::vector< int > | **get_hole_surfaces** (int surface_id)<br>Returns the adjacent surfaces in a hole for a surface. |
| std::vector< std::vector< int > > | **get_surface_cavity_collections** (const std::vector< int > &volume_list, double hr_threshold, double area_threshold, std::vector< double > &return_cavity_hrs, std::vector< double > &return_cavity_areas)<br>Returns the collections of surfaces that comprise holes or cavities in the specified volumes. Filter by hydarulic radius and area of the cavity. |

| | |
|---|---|
| std::vector< std::pair< std::vector< int >, double > > | **get_surface_hole_collections** (const std::vector< int > &volume_list, double radius_threshold)<br>Returns the collections of surfaces that comprise holes in the specified volumes. Filter by radius of the hole. |
| std::vector< std::pair< std::vector< int >, double > > | **get_blend_chain_collections** (const std::vector< int > &volume_list, double radius_threshold)<br>Returns the collections of surfaces that comprise blend chains in the specified volumes. Filter by radius threshold. |
| std::vector< std::pair< std::vector< int >, double > > | **get_chamfer_chain_collections** (const std::vector< int > &volume_list, double thickness_threshold)<br>Returns the collections of surfaces that comprise chamfers in the specified volumes. Filter by thickness of chamfer. |
| std::vector< std::vector< std::string > > | **get_solutions_for_cavity_surface** (int surface_id)<br>Get the solution list for a given cavity surface. |
| double | **get_merge_tolerance** ()<br>Get the current merge tolerance value. |

**Blocks, Sidesets, and Nodesets**

| | |
|---|---|
| std::string | **get_exodus_entity_name** (const std::string entity_type, int entity_id)<br>Get the name associated with an exodus entity. |
| std::string | **get_exodus_entity_type** (std::string entity_type, int entity_id)<br>Get the type of an exodus entity. |

| | |
|---|---|
| std::string | **get_exodus_entity_description** (std::string entity_type, int entity_id)<br>Get the description associated with an exodus entity. |
| std::vector< double > | **get_all_exodus_times** (const std::string &filename)<br>Open an exodus file and get a vector of all stored time stamps. |
| std::vector< std::string > | **get_all_exodus_variable_names** (const std::string &filename, const std::string &variable_type)<br>Open an exodus file and get a list of all stored variable names. |
| int | **get_block_id** (std::string entity_type, int entity_id)<br>Get the associated block id for a specific curve, surface, or volume. |
| std::vector< int > | **get_block_ids** (const std::string &mesh_geometry_file_name)<br>Get list of block ids from a mesh geometry file. |
| std::vector< int > | **get_block_id_list** ()<br>Get a list of all blocks. |
| std::vector< int > | **get_nodeset_id_list** ()<br>Get a list of all nodesets. |
| std::vector< int > | **get_sideset_id_list** ()<br>Get a list of all sidesets. |
| std::vector< int > | **get_bc_id_list** (CI_BCTypes bc_type_enum)<br>Get a list of all bcs of a specified type. |

| | |
|---|---|
| std::string | **get_bc_name** (CI_BCTypes bc_type_enum, int bc_id)<br>Get the name for the specified bc. |
| std::vector< int > | **get_nodeset_id_list_for_bc** (CI_BCTypes bc_type_enum, int bc_id)<br>Get a list of all nodesets the specified bc is applied to. |
| std::vector< int > | **get_sideset_id_list_for_bc** (CI_BCTypes bc_type_enum, int bc_id)<br>Get a list of all sidesets the specified bc is applied to. |
| int | **get_next_sideset_id** ()<br>Get a next available sideset id. |
| int | **get_next_nodeset_id** ()<br>Get a next available nodeset id. |
| int | **get_next_block_id** ()<br>Get a next available block id. |
| std::string | **get_copy_nodeset_on_geometry_copy_setting** ()<br>Get the copy nodeset on geometry copy setting. |
| std::string | **get_copy_sideset_on_geometry_copy_setting** ()<br>Get the copy nodeset on geometry copy setting. |
| std::string | **get_copy_block_on_geometry_copy_setting** ()<br>Get the copy nodeset on geometry copy setting. |

| | |
|---|---|
| bool | **set_copy_nodeset_on_geometry_copy_setting** (std::string val)<br>Set the copy nodeset on geometry copy setting "ON", "USE_ORIGINAL", or "OFF". |
| bool | **set_copy_sideset_on_geometry_copy_setting** (std::string val)<br>Set the copy sideset on geometry copy setting "ON", "USE_ORIGINAL", or "OFF". |
| bool | **set_copy_block_on_geometry_copy_setting** (std::string val)<br>Set the copy block on geometry copy setting "ON", "USE_ORIGINAL", or "OFF". |
| void | **get_block_children** (int block_id, std::vector< int > &returned_group_list, std::vector< int > &returned_node_list, std::vector< int > &returned_sphere_list, std::vector< int > &returned_edge_list, std::vector< int > &returned_tri_list, std::vector< int > &returned_face_list, std::vector< int > &returned_pyramid_list, std::vector< int > &returned_tet_list, std::vector< int > &returned_hex_list, std::vector< int > &returned_wedge_list, std::vector< int > &returned_volume_list, std::vector< int > &returned_surface_list, std::vector< int > &returned_curve_list, std::vector< int > &returned_vertex_list)<br>Get lists of any and all possible children of a block. |
| void | **get_nodeset_children** (int nodeset_id, std::vector< int > &returned_node_list, std::vector< int > &returned_volume_list, std::vector< int > &returned_surface_list, std::vector< int > &returned_curve_list, std::vector< int > &returned_vertex_list)<br>get lists of any and all possible children of a nodeset |

| | |
|---|---|
| void | **get_sideset_children** (int sideset_id, std::vector< int > &returned_face_list, std::vector< int > &returned_surface_list, std::vector< int > &returned_curve_list)<br>get lists of any and all possible children of a sideset |
| std::vector< int > | **get_block_volumes** (int block_id)<br>Get a list of volume ids associated with a specific block. |
| std::vector< int > | **get_block_surfaces** (int block_id)<br>Get a list of surface associated with a specific block. |
| std::vector< int > | **get_block_curves** (int block_id)<br>Get a list of curve associated with a specific block. |
| std::vector< int > | **get_block_vertices** (int block_id)<br>Get a list of vertices associated with a specific block. |
| bool | **get_block_elements_and_nodes** (int block_id, std::vector< int > &returned_node_list, std::vector< int > &returned_sphere_list, std::vector< int > &returned_edge_list, std::vector< int > &returned_tri_list, std::vector< int > &returned_face_list, std::vector< int > &returned_pyramid_list, std::vector< int > &returned_wedge_list, std::vector< int > &returned_tet_list, std::vector< int > &returned_hex_list)<br>Get lists of the nodes and different element types associated with this block. This function is recursive, meaning that if the block was created pointing to a piece of geometry, it will traverse down and get the mesh entities associated to that geometry. |

| | |
|---|---|
| std::vector< int > | **get_edges_to_swap** (int curve_id)<br>Given a curve defining a knife edge between two triangle-meshed surfaces, return a list of edges on triangles at the curve that are good candidates for swapping. A good candidate for swapping means that if swapped, the two triangles at the knife's edge will have a larger interior dihedral angle between them, allowing a larger volume to accommodate tetmeshing. |
| std::vector< int > | **get_block_nodes** (int block_id)<br>Get a list of nodes associated with a specific block. |
| std::vector< int > | **get_block_edges** (int block_id)<br>Get a list of edges associated with a specific block. |
| std::vector< int > | **get_block_tris** (int block_id)<br>Get a list of tris associated with a specific block. |
| std::vector< int > | **get_block_faces** (int block_id)<br>Get a list of faces associated with a specific block. |
| std::vector< int > | **get_block_pyramids** (int block_id)<br>Get a list of pyramids associated with a specific block. |
| std::vector< int > | **get_block_wedges** (int block_id)<br>Get a list of wedges associated with a specific block. |
| std::vector< int > | **get_block_tets** (int block_id)<br>Get a list of tets associated with a specific block. |

| | |
|---|---|
| std::vector< int > | **get_block_hexes** (int block_id)<br>Get a list of hexes associated with a specific block. |
| std::vector< int > | **get_volume_hexes** (int volume_id)<br>get the list of any hex elements in a given volume |
| std::vector< int > | **get_volume_tets** (int volume_id)<br>get the list of any tet elements in a given volume |
| std::vector< int > | **get_volume_wedges** (int volume_id)<br>get the list of any wedge elements in a given volume |
| std::vector< int > | **get_volume_pyramids** (int volume_id)<br>get the list of any pyramid elements in a given volume |
| std::vector< int > | **get_nodeset_volumes** (int nodeset_id)<br>Get a list of volume ids associated with a specific nodeset. |
| std::vector< int > | **get_nodeset_surfaces** (int nodeset_id)<br>Get a list of surface ids associated with a specific nodeset. |
| std::vector< int > | **get_nodeset_curves** (int nodeset_id)<br>Get a list of curve ids associated with a specific nodeset. |
| std::vector< int > | **get_nodeset_vertices** (int nodeset_id)<br>Get a list of vertex ids associated with a specific nodeset. |

| | |
|---|---|
| std::vector< int > | **get_nodeset_nodes** (int nodeset_id)<br>Get a list of node ids associated with a specific nodeset. This only returns the nodes that were specifically assigned to this nodeset. If the nodeset was created as a piece of geometry, get_nodeset_nodes will not return the nodes on that geometry See also get_nodeset_nodes_inclusive. |
| std::vector< int > | **get_nodeset_nodes_inclusive** (int nodeset_id)<br>Get a list of node ids associated with a specific nodeset. This includes all nodes specifically assigned to the nodeset, as well as nodes associated to a piece of geometry which was used to define the nodeset. |
| std::vector< int > | **get_sideset_curves** (int sideset_id)<br>Get a list of curve ids associated with a specific sideset. |
| std::vector< int > | **get_sideset_edges** (int sideset_id)<br>Get a list of any quads in a sideset. |
| std::vector< int > | **get_curve_edges** (int curve_id)<br>get the list of any edge elements on a given curve |
| std::vector< int > | **get_sideset_surfaces** (int sideset_id)<br>Get a list of any surfaces in a sideset. |
| std::vector< int > | **get_sideset_quads** (int sideset_id)<br>Get a list of any quads in a sideset. |
| std::vector< int > | **get_sideset_tris** (int sideset_id)<br>Get a list of any tris in a sideset. |
| double | **get_sideset_area** (int sideset_id) |

| | |
|---|---|
| | Get area of the sideset. |
| std::vector< int > | **get_surface_quads** (int surface_id)<br>get the list of any quad elements on a given surface |
| std::vector< int > | **get_surface_tris** (int surface_id)<br>get the list of any tri elements on a given surface |
| int | **get_surface_num_loops** (int surface_id)<br>get the number of loops on the surface |
| std::vector< std::vector< int > > | **get_surface_loop_nodes** (int surface_id)<br>get the ordered list of nodes on the loops of this surface |
| std::string | **get_entity_sense** (std::string source_type, int source_id, int sideset_id)<br>Get the sense of a sideset item. |
| std::string | **get_wrt_entity** (std::string source_type, int source_id, int sideset_id)<br>Get the with-respect-to entity. |
| std::vector< std::string > | **get_geometric_owner** (std::string mesh_entity_type, std::string mesh_entity_list)<br>Get a list of geometric owners given a list of mesh entities. |
| std::vector< std::string > | **get_all_geometric_owners** (std::string mesh_entity_type, std::string mesh_entity_list)<br>Get a list of geometric owners given a list of mesh entities. returns geometric owners of entity as well as all of its child mesh entities. |

### Geometry-Mesh Entity Support

| | |
|---|---|
| std::vector< int > | **get_volume_nodes** (int volume_id)<br>Get list of node ids owned by a volume. Excludes nodes owned by bounding surfs, curves and verts. |
| std::vector< int > | **get_surface_nodes** (int surface_id)<br>Get list of node ids owned by a surface. Excludes nodes owned by bounding curves and verts. |
| std::vector< int > | **get_curve_nodes** (int curve_id)<br>Get list of node ids owned by a curve. Excludes nodes owned by bounding vertices. |
| int | **get_vertex_node** (int vertex_id)<br>Get the node owned by a vertex. |

### Group Support

| | |
|---|---|
| int | **get_id_from_name** (const std::string &name)<br>Get id for a named entity. |
| std::vector< int > | **get_all_ids_from_name** (const std::string &geo_type, const std::string &name)<br>Get all ids of a geometry type with the prefix given by string. |
| void | **get_group_children** (int group_id, std::vector< int > &returned_group_list, std::vector< int > &returned_body_list, std::vector< int > &returned_volume_list, std::vector< int > &returned_surface_list, std::vector< int > &returned_curve_list, std::vector< int > &returned_vertex_list, int &returned_node_count, int &returned_edge_count, int &returned_hex_count, int |

| | |
|---|---|
| | &returned_quad_count, int &returned_tet_count, int &returned_tri_count, int &returned_wedge_count, int &returned_pyramid_count, int &returned_sphere_count)<br>Get group children. |
| std::vector< int > | **get_group_groups** (int group_id)<br>Get group groups (groups that are children of another group) |
| std::vector< int > | **get_group_volumes** (int group_id)<br>Get group volumes (volumes that are children of a group) |
| std::vector< int > | **get_group_bodies** (int group_id)<br>Get group bodies (bodies that are children of a group) |
| std::vector< int > | **get_group_surfaces** (int group_id)<br>Get group surfaces (surfaces that are children of a group) |
| std::vector< int > | **get_group_curves** (int group_id)<br>Get group curves (curves that are children of a group) |
| std::vector< int > | **get_group_vertices** (int group_id)<br>Get group vertices (vertices that are children of a group) |
| std::vector< int > | **get_group_nodes** (int group_id)<br>Get group nodes (nodes that are children of a group) |
| std::vector< int > | **get_group_edges** (int group_id) |

| | |
|---|---|
| | Get group edges (edges that are children of a group) |
| std::vector< int > | **get_group_quads** (int group_id)<br>Get group quads (quads that are children of a group) |
| std::vector< int > | **get_group_tris** (int group_id)<br>Get group tris (tris that are children of a group) |
| std::vector< int > | **get_group_tets** (int group_id)<br>Get group tets (tets that are children of a group) |
| std::vector< int > | **get_group_wedges** (int group_id)<br>Get group wedges (wedges that are children of a group) |
| std::vector< int > | **get_group_pyramids** (int group_id)<br>Get group pyramids (pyramids that are children of a group) |
| std::vector< int > | **get_group_spheres** (int group_id) |
| std::vector< int > | **get_group_hexes** (int group_id) |
| int | **get_next_group_id** ()<br>Get the next available group id from Cubit. |
| void | **delete_all_groups** ()<br>Delete all groups. |
| void | **delete_group** (int group_id)<br>Delete a specific group. |
| void | **set_max_group_id** (int maximum_group_id) |

| | |
|---|---|
| | Reset Cubit's max group id This is really dangerous to use and exists only to overcome a limitation with Cubit. Cubit keeps track of the next group id to assign. But those ids just keep incrementing in Cubit. Some of the power tools in the Cubit GUI make groups 'under the covers' for various operations. The groups are immediately deleted. But, creating those groups will cause Cubit's group id to increase and downstream journal files may be messed up because those journal files are expecting a certain ID to be available. |
| int | **create_new_group** ()<br>Create a new group. |
| void | **remove_entity_from_group** (int group_id, int entity_id, const std::string &entity_type)<br>Remove a specific entity from a specific group. |
| void | **add_entity_to_group** (int group_id, int entity_id, const std::string &entity_type)<br>Add a specific entity to a specific group. |
| void | **add_entities_to_group** (int group_id, const std::vector< int > &entity_id, const std::string &entity_type)<br>Add a list of entities to a specific group. |
| void | **group_list** (std::vector< std::string > &name_list, std::vector< int > &returned_id_list)<br>Get the names and ids of all the groups (excluding the pick group) that are defined by the current cubit session. |
| std::vector< std::pair< std::string, int > > | **group_names_ids** () |

| | |
|---|---|
| | Get the names and ids of all the groups returned in a name/id structure that are defined by the current cubit session. |
| std::vector< int > | **get_mesh_group_parent_ids** (const std::string &element_type, int element_id)<br>Get the group ids which are parents to the indicated mesh element. |
| bool | **is_mesh_element_in_group** (const std::string &element_type, int element_id)<br>Indicates whether a mesh element is in a group. |

**General Purpose Utility**

| | |
|---|---|
| bool | **is_part_of_list** (int target_id, std::vector< int > id_list)<br>Routine to check for the presence of an id in a list of ids. |
| int | **get_last_id** (const std::string &entity_type)<br>Get the id of the last created entity of the given type. |
| bool | **entity_exists** (const std::string &entity_type, int id)<br>return whether an entity of specified ID exists |
| std::string | **get_idless_signature** (std::string entity_type, int entity_id)<br>get the idless signature of a geometric or mesh entity |
| std::string | **get_idless_signatures** (std::string entity_type, const std::vector< int > &entity_id_list)<br>get the idless signatures of a range of geometric or mesh entities |

**Metadata Support**

| | |
|---|---|
| std::string | **get_assembly_classification_level** ()<br>Get Classification Level for metadata. |
| std::string | **get_assembly_classification_category** ()<br>Get Classification Category for metadata. |
| std::string | **get_assembly_weapons_category** ()<br>Get Weapons Category for metadata. |
| std::string | **get_assembly_metadata** (int volume_id, int data_type)<br>Get metadata for a specified volume id. |
| bool | **is_assembly_metadata_attached** (int volume_id)<br>Determine whether metadata is attached to a specified volume. |
| std::string | **get_assembly_name** (int assembly_id)<br>Get the stored name of an assembly node. |
| std::string | **get_assembly_path** (int assembly_id)<br>Get the stored path of an assembly node. |
| std::string | **get_assembly_type** (int assembly_id)<br>Get the stored type of an assembly node. |
| std::string | **get_parent_assembly_path** (int assembly_id)<br>Get the stored path of an assembly node' parent. |
| int | **get_assembly_level** (int assembly_id)<br>Get the stored level of an assembly node. |

| | |
|---|---|
| std::string | **get_assembly_description** (int assembly_id)<br>Get the stored description of an assembly node. |
| int | **get_assembly_instance** (int assembly_id)<br>Get the stored instance number of an assembly node. |
| int | **get_parent_assembly_instance** (int assembly_id)<br>Get the stored instance number of an assembly node's instance. |
| std::string | **get_assembly_file_format** (int assembly_id)<br>Get the stored file format of an assembly node. |
| std::string | **get_assembly_units** (int assembly_id)<br>Get the stored units measure of an assembly node. |
| std::string | **get_assembly_material_description** (int assembly_id)<br>Get the stored material description of an assembly part. |
| std::string | **get_assembly_material_specification** (int assembly_id)<br>Get the stored material specification of an assembly part. |

**Mesh Element Queries**

| | |
|---|---|
| int | **get_exodus_id** (const std::string &entity_type, int entity_id)<br>Get the exodus/genesis id for this element. |

| | |
|---|---|
| std::string | **get_geometry_owner** (const std::string &entity_type, int entity_id)<br>Get the geometric owner of this mesh element. |
| std::vector< int > | **get_connectivity** (const std::string &entity_type, int entity_id)<br>Get the list of node ids contained within a mesh entity. |
| std::vector< int > | **get_expanded_connectivity** (const std::string &entity_type, int entity_id)<br>Get the list of node ids contained within a mesh entity, including interior nodes. |
| std::vector< int > | **get_sub_elements** (const std::string &entity_type, int entity_id, int dimension)<br>Get the lower dimesion entities associated with a higher dimension entities. For example get the faces associated with a hex or the edges associated with a tri. |
| bool | **get_node_exists** (int node_id)<br>Check the existance of a node. |
| bool | **get_element_exists** (int element_id)<br>Check the existance of an element. |
| std::string | **get_element_type** (int element_id)<br>return the type of a given element |
| int | **get_element_type_id** (int element_id)<br>return the type id of a given element |
| int | **get_element_block** (int element_id)<br>return the block that a given element is in. |

| | |
|---|---|
| int | **get_global_element_id** (const std::string &element_type, int id)<br>Given a hex, tet, etc. id, return the global element id. |
| int | **get_hex_global_element_id** (int hex_id)<br>Given a hex id, return the global element id. |
| int | **get_tet_global_element_id** (int tet_id)<br>Given a tet id, return the global element id. |
| int | **get_wedge_global_element_id** (int wedge_id)<br>Given a wedge id, return the global element id. |
| int | **get_pyramid_global_element_id** (int pyramid_id)<br>Given a pyramid id, return the global element id. |
| int | **get_tri_global_element_id** (int tri_id)<br>Given a tri id, return the global element id. |
| int | **get_quad_global_element_id** (int quad_id)<br>Given a quad id, return the global element id. |
| int | **get_edge_global_element_id** (int edge_id)<br>Given a edge id, return the global element id. |
| int | **get_sphere_global_element_id** (int edge_id)<br>Given a sphere id, return the global element id. |
| int | **get_node_global_id** (int node_id) |

| | |
|---|---|
| | Given a node id, return the global element id that is assigned when the mesh is exported. |
| int | **get_closest_node** (double x_coordinate, double y_coordinate, double z_coordinate)<br>Get the node closest to the given coordinates. |
| std::array< double, 3 > | **get_nodal_coordinates** (int node_id)<br>Get the nodal coordinates for a given node id. |
| std::vector< int > | **get_node_faces** (int node_id) |
| std::vector< int > | **get_node_tris** (int node_id) |
| std::vector< int > | **get_node_edges** (int node_id)<br>Get the edge ids that share a node. |
| bool | **get_node_position_fixed** (int node_id)<br>Query "fixedness" state of node. A fixed node is not affecting by smoothing. |
| std::vector< std::pair< int, int > > | **get_submap_corner_types** (int surface_id)<br>Get a list of vertex ids and the corresponding corner vertex types if the surface were defined as submap surface. There are no side affects. This does not actually assign corner types or change the underlying mesh scheme of the surface. |
| std::string | **get_sideset_element_type** (int sideset_id)<br>Get the element type of a sideset. |
| std::string | **get_block_element_type** (int block_id)<br>Get the element type of a block. |

| | |
|---|---|
| int | **get_exodus_element_count** (int entity_id, std::string entity_type)<br>Get the number of elements in a exodus entity. |
| int | **get_block_attribute_count** (int block_id)<br>Get the number of attributes in a block. |
| int | **get_block_element_attribute_count** (int block_id)<br>Get the number of attributes in a block element. |
| double | **get_block_attribute_value** (int block_id, int attribute_index)<br>Get a specific block attribute value. |
| std::string | **get_block_attribute_name** (int block_id, int attribute_index)<br>Get a specific block attribute name. |
| std::vector< std::string > | **get_block_element_attribute_names** (int block_id)<br>Get a specific block element attribute name. |
| std::vector< std::string > | **get_valid_block_element_types** (int block_id)<br>Get a list of potential element types for a block. |
| int | **get_block_material** (int block_id)<br>Get the id of the material assigned to the specified block. |
| std::vector< std::vector< int > > | **get_blocks_with_materials** ()<br>Get the block ids and ids of the respective materials assigned to each block. |

| | |
|---|---|
| int | **get_exodus_variable_count** (std::string container_type, int container_id)<br>Get the number of exodus variables in a nodeset, sideset, or block. |
| std::vector< std::string > | **get_exodus_variable_names** (std::string container_type, int container_id)<br>Get the names of exodus variables in a nodeset, sideset, or block. |
| int | **get_nodeset_node_count** (int nodeset_id)<br>Get the number of nodes in a nodeset. |
| int | **get_geometry_node_count** (const std::string &entity_type, int entity_id)<br>Get the node count for a specific geometric entity. |
| void | **get_owning_volume_ids** (const std::string &entity_type, std::vector< int > &entity_list, std::vector< int > &volume_ids)<br>Gets the id's of the volumes that are owners of one of the specified entities. |
| std::string | **get_mesh_element_type** (const std::string &entity_type, int entity_id)<br>Get the mesh element type contained in the specified geometry. |

**Boundary Condition Support**

| | |
|---|---|
| bool | **is_on_thin_shell** (CI_BCTypes bc_type_enum, int entity_id)<br>Determine whether a BC is on a thin shell. Valid for temperature, convection and heatflux. |
| bool | **temperature_is_on_solid** (CI_BCTypes bc_type_enum, int entity_id) |

| | |
|---|---|
| | Determine whether a BC temperature is on a solid. Valid for convection and temperature. |
| bool | **convection_is_on_solid** (int entity_id)<br>Determine whether a BC convection is on a solid. Valid for convection. |
| bool | **convection_is_on_shell_area** (int entity_id, CI_BCEntityTypes shell_area_enum)<br>Determine whether a BC convection is on a shell top or bottom. Valid for convection. |
| double | **get_convection_coefficient** (int entity_id, CI_BCEntityTypes bc_type_enum)<br>Get the convection coefficient. |
| double | **get_bc_temperature** (CI_BCTypes bc_type_enum, int entity_id, CI_BCEntityTypes temp_type_enum)<br>Get the temperature. Valid for convection, temperature. |
| bool | **temperature_is_on_shell_area** (CI_BCTypes bc_type_enum, CI_BCEntityTypes bc_area_enum, int entity_id)<br>Determine whether a BC temperature is on a shell area. Valid for convection and temperature and on top, bottom, gradient, and middle. |
| bool | **heatflux_is_on_shell_area** (CI_BCEntityTypes bc_area_enum, int entity_id)<br>Determine whether a BC heatflux is on a shell area. |
| double | **get_heatflux_on_area** (CI_BCEntityTypes bc_area_enum, int entity_id) |

| | |
|---|---|
| | Get the heatflux on a specified area. |
| int | **get_cfd_type** (int entity_id)<br>Get the cfd subtype for a specified cfd BC. |
| double | **get_contact_pair_friction_value** (int entity_id)<br>Get the contact pair's friction value. |
| double | **get_contact_pair_tolerance_value** (int entity_id)<br>Get the contact pair's upper bound tolerance value. |
| double | **get_contact_pair_tol_lower_value** (int entity_id)<br>Get the contact pair's lower bound tolerance value. |
| bool | **get_contact_pair_tied_state** (int entity_id)<br>Get the contact pair's tied state. |
| bool | **get_contact_pair_general_state** (int entity_id)<br>Get the contact pair's general state. |
| bool | **get_contact_pair_exterior_state** (int entity_id)<br>Get the contact pair's exterior state. |
| const double * | **get_displacement_dof_values** (int entity_id)<br>**This function only available from C++** Get the displacement's dof values |
| const int * | **get_displacement_dof_signs** (int entity_id) |

| | |
|---|---|
| | **This function only available from C++** Get the displacement's dof signs |
| const double * | **get_velocity_dof_values** (int entity_id)<br>**This function only available from C++** Get the velocity's dof values |
| const int * | **get_velocity_dof_signs** (int entity_id)<br>**This function only available from C++** Get the velocity's dof signs |
| std::string | **get_velocity_combine_type** (int entity_id)<br>Get the velocity's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine". |
| const double * | **get_acceleration_dof_values** (int entity_id)<br>**This function only available from C++** Get the acceleration's dof values |
| const int * | **get_acceleration_dof_signs** (int entity_id)<br>**This function only available from C++** Get the acceleration's dof signs |
| std::string | **get_acceleration_combine_type** (int entity_id)<br>Get the acceleration's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine". |
| std::string | **get_displacement_combine_type** (int entity_id)<br>Get the displacement's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine". |
| double | **get_pressure_value** (int entity_id)<br>Get the pressure value. |

| | |
|---|---|
| std::string | **get_pressure_function** (int entity_id)<br>Get the pressure function. |
| double | **get_force_magnitude** (int entity_id)<br>Get the force magnitude from a force. |
| double | **get_moment_magnitude** (int entity_id)<br>Get the moment magnitude from a force. |
| std::array< double, 3 > | **get_force_direction_vector** (int entity_id)<br>Get the direction vector from a force. |
| std::array< double, 3 > | **get_force_moment_vector** (int entity_id)<br>Get the moment vector from a force. |
| std::string | **get_constraint_type** (int constraint_id)<br>Get the type of a specified constraint. |
| std::string | **get_constraint_reference_point** (int constraint_id)<br>Get the reference point of a specified constraint. |
| std::string | **get_constraint_dependent_entity_point** (int constraint_id)<br>Get the dependent entity of a specified constraint. |
| double | **get_material_property** (CI_MaterialProperty material_property_enum, int entity_id)<br>Get the specified material property value. |
| int | **get_media_property** (int entity_id)<br>Get the media property value. |

| | |
|---|---|
| std::vector< std::string > | **get_material_name_list** ()<br>Get a list of all defined material names. |
| std::vector< std::string > | **get_media_name_list** ()<br>Get a list of all defined material names. |
| std::string | **get_material_name** (int material_id)<br>Get the name of the material (or cfd media) with the given id. |
| double | **calculate_timestep_estimate** (std::string entity_type, std::vector< int > entity_ids)<br>Calculates time step estimate on elements of/in entity type: "Tet" or "Hex" or "Volume" or "Block" or "Group" The hexes or tets must belong to a single block and that block must have a material assigned to it. That material must have elastic_modulus, poisson_ratio, and density defined. |
| double | **calculate_timestep_estimate_with_props** (std::string entity_type, std::vector< int > entity_id_list, double density, double youngs_modulus, double poissons_ratio)<br>Calculates time step estimate on elements of/in entity type: "Tet" or "Hex" or "Volume" or "Block" or "Group". |
| double | **get_target_timestep** ()<br>Returns the target timestep threshold used in the timestep density multiplier metric. |
| std::vector< std::array< double, 3 > > | **snap_locations_to_geometry** (const std::vector< std::array< double, 3 > > &locations, std::string entity_type, int entity_id, double tol)<br>Snaps xyz locations to closest point on entity. Then snaps to child curves or vertices |

| | |
|---|---|
| | within given tolerance. Vertices snapped to before curves. |
| std::vector< double > | **measure_between_entities** (std::string entity_type1, int entity_id1, std::string entity_type2, int entity_id2)<br>returns distance between two geometry entities and their closest points |
| std::vector< int > | **gather_surfaces_by_orientation** (std::vector< int > seed_surf_ids, std::vector< int > all_surf_ids)<br>Gathers connected surfaces to those in 'seed_surf_ids' that use common curves in an opposite sense. For example, if a surface A in 'seed_surf_ids' uses a curve in the FORWARD sense and it can find surface, B, that uses that same curve in a REVERSED sense, it adds B to the list. The search continues with all of surface B's curves. All the surfaces in 'seed_surf_ids' will be returned. If the user wants to limit the scope of possible surfaces that are searched, 'all_surf_ids' can be populated. If 'all_surf_ids' is empty, all surfaces are candidates. This function can be helpful in finding enclosures when you have a set of non-manifold surfaces. |
| void | **set_label_type** (const char *entity_type, int label_flag)<br>make calls to SVDrawTool::set_label_type |
| int | **get_label_type** (const char *entity_type)<br>make calls to SVDrawTool::get_label_type |
| std::vector< int > | **get_coordinate_systems_id_list** ()<br>get a list of coordinate system ids |

| | |
|---|---|
| std::vector< double > | **get_n_largest_distances_between_meshes** (int n, std::string entity_type1, std::vector< int > ids1, std::string entity_type2, std::vector< int > ids2)<br>Finds the 'n' largest distances between two meshes. These distances are from the nodes on the entities of 'ids1' to the elements in 'ids2'. Only triangle and face (quads) element types are supported. It is assumed that the meshes approximately line up.<br>Each distance is returned with three values: |
| void | **compare_geometry_and_mesh** (std::vector< int > volume_ids, std::vector< int > block_ids, std::vector< int > hex_ids, std::vector< int > tet_ids, double tolerance, int &returned_unmatched_volumes_count, int &returned_unmatched_elements_count, std::vector< int > &returned_full_matches_group_ids_list, std::vector< int > &returned_partial_matches_group_ids_list, int &returned_volume_curves_group_id)<br>Compare the geometry and mesh. |
| double | **get_dbl_sculpt_default** (const char *variable)<br>return sculpt default value |
| int | **get_int_sculpt_default** (const char *variable) |
| bool | **get_bool_sculpt_default** (const char *variable) |
| std::string | **get_string_sculpt_default** (const char *variable) |
| double | **get_blunt_tangency_default_depth** (int vert_id, double angle, bool add_material) |

| | |
|---|---|
| | get default depth value for blunt tangency operation |
| std::vector< double > | **get_reduce_bolt_core_default_dimensions** (int vol_id)<br>get default dimensions for reduce vol bolt core operation |
| double | **get_bolt_diameter** (int vol_id)<br>get diameter of bolt shank |
| std::vector< double > | **get_bolt_axis** (int vol_id)<br>get axis vector of bolt |
| double | **get_bolt_shigley_radius** (int vol_id, double angle)<br>get the equivalent shigley's frustum radius at the interface surfaces between upper and lower volumes for a bolt |
| std::vector< std::vector< double > > | **get_bolt_coordinate_system** (std::string geom_type, int id)<br>get the local coordinate system for a bolt (or bolt hole) |
| std::vector< int > | **get_bolt_clamped_members** (int vol_id, std::vector< int > nearby_vols)<br>return the clamped members associated with a bolt in order: bearing surface member, sandwiched members (if any), threaded member |
| std::vector< int > | **get_bolts_in_clamped_members** (std::string geo_type, std::vector< int > clamped_vols, std::vector< int > candidate_bolts)<br>return the bolts that are common to the clamped members Note: uses ML classification to determine "bolt" category |

| | |
|---|---|
| std::vector< BoltHoleInfo > | **get_bolt_holes_info** (std::string geo_type, std::vector< int > clamped_members, double radius_threshold, double gap_threshold)<br>return the pilot hole definitions from a list of volumes. Returns a vector of BoltHole structs |
| std::vector< std::vector< int > > | **get_bolt_holes** (std::string geo_type, std::vector< int > clamped_members, double radius_threshold, double gap_threshold)<br>returns the surfaces from upper and lower holes from the specified clamped volumes or blocks. Results can be used directly in the reduce bolt commands. |
| std::vector< int > | **get_2D_sheet_volumes** (int vol_id)<br>get the 2D sheet volumes associated with this 3D thin volume |
| int | **get_3D_thin_volume** (int vol_id)<br>get the 3D sheet volume associated with this 2D sheet volume |

**Boundary Layer Support**

| | |
|---|---|
| int | **get_next_boundary_layer_id** () |
| bool | **is_boundary_layer_id_available** (int boundary_layer_id) |
| std::string | **get_boundary_layer_algorithm** (int boundary_layer_id) |
| std::vector< int > | **get_boundary_layers_by_base** (const std::string &base_type, int base_id) |
| std::vector< int > | **get_boundary_layers_by_pair** (const std::string &base_type, int base_id, int parent_id) |

| | |
|---|---|
| bool | **get_boundary_layer_uniform_parameters** (int boundary_layer_id, double &returned_first_row_height, double &returned_growth_factor, int &returned_number_rows) |
| bool | **get_boundary_layer_aspect_first_parameters** (int boundary_layer_id, double &returned_first_row_aspect, double &returned_growth_factor, int &returned_number_rows) |
| bool | **get_boundary_layer_aspect_last_parameters** (int boundary_layer_id, double &returned_first_row_height, int &returned_number_rows, double &returned_last_row_aspect) |
| bool | **get_boundary_layer_curve_surface_pairs** (int boundary_layer_id, std::vector< int > &returned_curve_list, std::vector< int > &returned_surface_list) |
| bool | **get_boundary_layer_surface_volume_pairs** (int boundary_layer_id, std::vector< int > &returned_surface_list, std::vector< int > &returned_volume_list) |
| bool | **get_boundary_layer_vertex_intersection_types** (std::vector< int > &returned_vertex_list, std::vector< int > &returned_surface_list, std::vector< std::string > &returned_types) |
| bool | **get_boundary_layer_curve_intersection_types** (std::vector< int > &returned_curve_list, std::vector< int > &returned_volume_list, std::vector< std::string > &returned_types) |
| bool | **get_boundary_layer_continuity** (int boundary_layer_id) |

| | |
|---|---|
| std::vector< int > | **get_boundary_layer_id_list** () |
| std::vector< **CFD_BC_Entity** > | **get_all_cfd_bcs** () |
| std::vector< **AssemblyItem** > | **get_assembly_items** () |
| std::vector< **AssemblyItem** > | **get_top_level_assembly_items** () |
| std::vector< **AssemblyItem** > | **get_assembly_children** (int assembly_id) |
| std::vector< int > | **get_volumes_for_node** (std::string node_name, int node_instance) |
| std::vector< **MeshErrorFeedback** * > | **get_mesh_errors** () |
| int | **get_mesh_error_count** () |

**Geometry from ids**

| | |
|---|---|
| **CubitInterface::Body** | **body** (int id_in)<br>Gets the body object from an ID. |
| **CubitInterface::Volume** | **volume** (int id_in)<br>Gets the volume object from an ID. |
| **CubitInterface::Surface** | **surface** (int id_in)<br>Gets the surface object from an ID. |
| **CubitInterface::Curve** | **curve** (int id_in)<br>Gets the curve object from an ID. |
| **CubitInterface::Vertex** | **vertex** (int id_in)<br>Gets the vertex object from an ID. |
| void | **reset** () |

| | |
|---|---|
| | Executes a reset within cubit. |

**Geometry Creation Functions**

| | |
|---|---|
| **Body** | **brick** (double width, double depth=-1, double height=-1)<br>Creates a brick of specified width, depth, and height. |
| **Body** | **sphere** (double radius, int x_cut=0, int y_cut=0, int z_cut=0, double inner_radius=0)<br>Creates all or part of a sphere. |
| **Body** | **prism** (double height, int sides, double major, double minor)<br>Creates a prism of the specified dimensions. |
| **Body** | **pyramid** (double height, int sides, double major, double minor, double top=0.0)<br>Creates a pyramid of the specified dimensions. |
| **Body** | **cylinder** (double height, double x_radius, double y_radius, double top_radius)<br>creates a cylinder of the specified dimensions |
| **Body** | **torus** (double center_radius, double swept_radius)<br>creates a torus of the specified dimensions |
| **Vertex** | **create_vertex** (double x=0, double y=0, double z=0)<br>Creates a vertex at a x,y,z. |
| **Curve** | **create_curve** (**Vertex** v0, **Vertex** v1)<br>Creates a curve between two vertices. |

| | |
|---|---|
| **Curve** | **create_arc_curve** (**Vertex** v0, **Vertex** v1, std::array< double, 3 > intermediate_point)<br>Creates a arc curve using end vertices and an intermediate point. |
| **Curve** | **create_spline** (std::vector< std::array< double, 3 > > points, int surface_id)<br>create spline through the given 3d points |
| **Body** | **create_surface** (std::vector< **Curve** > curves)<br>Creates a surface from boundary curves. |
| std::vector< **Body** > | **sweep_curve** (std::vector< **Curve** > curves, std::vector< **Curve** > along_curves, double draft_angle=0, int draft_type=0, bool rigid=false)<br>Create a **Body** or a set of Bodies from a swept curve. |
| **Body** | **copy_body** (**Body** init_body)<br>Creates a copy of the input **Body**. |

**Geometry Manipulation Functions**

| | |
|---|---|
| std::vector< **Body** > | **tweak_surface_offset** (std::vector< **Surface** > surfaces, std::vector< double > distances)<br>Performs a tweak surface offset command. |
| std::vector< **CubitInterface::Body** > | **tweak_surface_remove** (std::vector< **Surface** > surfaces, bool extend_ajoining=true, bool keep_old=false, bool preview=false)<br>Removes a surface from a body and extends the surrounding surfaces if extend_ajoining is true. |

| | |
|---|---|
| std::vector< **CubitInterface::Body** > | **tweak_curve_remove** (std::vector< **Curve** > curves, bool keep_old=false, bool preview=false)<br>Removes a curve from a body and extends the surrounding surface to fill the gap. |
| std::vector< **Body** > | **tweak_curve_offset** (std::vector< **Curve** > curves, std::vector< double > distances, bool keep_old=false, bool preview=false)<br>Performs a tweak curve offset command. |
| std::vector< **Body** > | **tweak_vertex_fillet** (std::vector< **Vertex** > verts, double radius, bool keep_old=false, bool preview=false)<br>Performs a tweak vertex fillet command. |
| std::vector< **Body** > | **subtract** (std::vector< **CubitInterface::Body** > tool_in, std::vector< **CubitInterface::Body** > from_in, bool imprint_in=false, bool keep_old_in=false)<br>Performs a boolean subtract operation. |
| std::vector< **Body** > | **unite** (std::vector< **CubitInterface::Body** > body_in, bool keep_old_in=false)<br>Performs a boolean unite operation. |
| void | **move** (**Entity** entity, std::array< double, 3 > vector, bool preview=false)<br>Moves the **Entity** the specified vector. |
| void | **scale** (**Entity** entity, double factor, bool preview=false)<br>Scales the **Entity** according to the specified factor. |
| void | **reflect** (**Entity** entity, std::array< double, 3 > axis, bool preview=false)<br>Reflect the **Entity** about the specified axis. |

| | |
|---|---|
| void | **set_nodal_variable** (std::vector< int > node_ids, std::string variable_name, std::vector< double > variables)<br>Sets nodal variables. |
| void | **set_element_variable** (std::vector< int > element_ids, std::string variable_name, std::vector< double > variables)<br>Sets element variables. |

**Machine Learning Support**

| | |
|---|---|
| bool | **load_ML** (std::string model_type="all")<br>load the machine learning training data |
| void | **unload_ML** (std::string model_type="all")<br>unload the machine learning training data |
| std::vector< std::vector< double > > | **get_ML_operation_features** (std::vector< std::string > ml_op_names, std::vector< size_t > entity1_ids, std::vector< size_t > entity2_ids, std::vector< std::vector< double > > params, double mesh_size, bool reduced_features=false)<br>get machine learning features for a list of cubit operations |
| std::vector< std::vector< double > > | **get_ML_predictions** (std::vector< std::string > ml_op_names, std::vector< size_t > entity1_ids, std::vector< size_t > entity2_ids, std::vector< std::vector< double > > params, double mesh_size, bool reduced_features=false)<br>get machine learning predictions for the list of operations and corresponding entities. This function will load the ML training data if not already loaded. It will first compute features and then run predictions from training data. Uses scikit-learn EDT (Ensembles of Decision Trees) for predictions |

| | |
|---|---|
| std::string | **get_ML_classification** (std::string geom_type, size_t ent_id)<br>return the name of the classification category for this surface or volume. uses same methods as get_ML_predictions for volume_no_op or classify_surface. Same as calling get_ML_operation_features + get_ML_predictions with volume_no_op and classify_surface, but returns category with highest probablity. |
| std::vector< std::string > | **get_ML_classifications** (std::string geom_type, std::vector< size_t > ent_ids)<br>same as get_ML_classification, but classifies multiple volumes or surfaces with a single call (more efficient) |
| std::vector< std::string > | **get_ML_classification_categories** (std::string geom_type)<br>return a list of strings representing all possible calssification categories for volumes or surfaces |
| bool | **ML_train** (std::string geom_type)<br>force a new training. Currently implemented for only classification methods, volume_no_op and surface_no_op. |
| std::vector< std::string > | **get_ML_operation_feature_types** (const std::string ml_op_name, bool reduced_features=false)<br>for the given operation type described by get_ML_operation_features, return a vector of strings indicating the type of data for each feature in the vector. Will return one of the following for each index: |
| std::vector< std::string > | **get_ML_operation_feature_names** (const std::string ml_op_name, bool reduced_features=false) |

| | |
|---|---|
| | for the given operation type described by get_ML_operation_features, return a vector of strings indicating the name of data for each feature in the vector. |
| int | **get_ML_operation_feature_size** (const std::string ml_op_name, const bool reduced_features=false)<br>for the given operation type described by get_ML_operation_features, return the expected size of the feature vector |
| int | **get_ML_operation_label_size** (const std::string ml_op_name)<br>for the given operation type return length of label vector the expected size of the feature vector |
| std::vector< std::string > | **get_ML_classification_models** ()<br>get the available classification ML model names |
| std::vector< std::string > | **get_ML_regression_models** ()<br>get the available regression ML model names |
| int | **get_ML_model_ID** (std::string)<br>get a unique ID for the given operation/model name |
| std::string | **get_ML_model_name** (int model_ID)<br>get the name for the given operation/model ID |
| std::vector< std::string > | **get_ML_operation** (const std::string op_name, const size_t entity_id1, const size_t entity_id2, const std::vector< double > params, const double small_curve_size, const double mesh_size) |

| | |
|---|---|
| | get the command, display and preview strings for a given ML operation type |
| std::vector< double > | **get_ML_feature_importances** (const std::string op_name)<br>return the vector of feature importances for a given operation type |
| double | **get_ML_feature_distance** (const std::string op_name, std::vector< double > &f1, std::vector< double > &f2)<br>feature distance is defined as a weighted distance between two feature vectors of the same size. Features are weighted on EDT (ensembles of decision trees) importance values. |
| void | **set_ML_base_user_dir** (const std::string path, const bool **print_info**=false)<br>set the path to any user training data. (classification only) |

CubitInterface Control

| | |
|---|---|
| const int | **CI_ERROR** = -1 |
| void | **init** (const std::vector< std::string > &argv)<br>Use init to initialize Cubit. Using a blank list as the input parameter is acceptable. |
| int | **destroy** ()<br>Closes the current journal file. |
| void | **ensure_init** () |
| void | **report_usage** () |
| void | **process_input_files** ()<br>**C++ only** |

| | |
|---|---|
| void | **enable_signal_handling** (bool on)<br>initialize/uninitialize signal handling **C++ only** |
| void | **print_info** (const std::string &message)<br>Print a message using the cubit message handler. |
| void | **set_cubit_message_handler** (CubitMessageHandler *hdlr)<br>redirect the output from cubit. |
| CubitMessageHandler * | **get_cubit_message_handler** ()<br>get the default message handler |
| void | **set_exit_handler** (ExternalExitHandler *hdlr)<br>Set the exit handler. |

## Detailed Description

The **CubitInterface** provides a Python/C++ interface into Cubit.
It provides an object oriented structure that allows a developer to manipulate objects familiar to Cubit such as bodies, volumes, surfaces, etc. It also allows developers to create and manipulate as well as query geometry.

## Function Documentation

### ◆ add_entities_to_group()

```
void add_entities_to_group(int                        group_id,
                           const std::vector< int > & entity_id,
                           const std::string &        entity_type
                          )
```

Add a list of entities to a specific group.

```
CubitInterface::add_entities_to_group (3, list, "surface" );
```

CubitInterface::add_entities_to_group
void add_entities_to_group(int group_id, const std::vector< int > &entity_id, const std::string &entity_type)
Add a list of entities to a specific group.

```
cubit.add_entities_to_group(3, list, "surface" )
```

Parameters

| | |
|---|---|
| **group_id** | ID of group to which the entity will be added |
| **list** | a vector of IDs of the entities to be added to the group |

**entity_type** Type of the entity to be added to the group. Note that this function is valid only for geometric entities

### ◆ add_entity_to_group()

void add_entity_to_group ( int *group_id*,
int *entity_id*,
const std::string & *entity_type*
)

Add a specific entity to a specific group.

```
CubitInterface::add_entity_to_group (3, 22, "surface" );
```

CubitInterface::add_entity_to_group
void add_entity_to_group(int group_id, int entity_id, const std::string &entity_type)
Add a specific entity to a specific group.

```
cubit.add_entity_to_group(3, 22, "surface" )
```

Parameters

**group_id** ID of group to which the entity will be added
**entity_id** ID of the entity to be added to the group
**entity_type** Type of the entity to be added to the group. Note that this function is valid only for geometric entities

### ◆ add_filename_to_recent_file_list()

void add_filename_to_recent_file_list ( std::string & *filename* )

Adds the filename to the recent file list.
Parameters

**filename** to be added to the recent file list.

### ◆ add_filter_type()

void add_filter_type ( const std::string & *filter_type* )

Add a filter type.

### ◆ are_adjacent_curves()

bool are_adjacent_curves ( std::vector< int > *curve_ids* )

return whether two or more curves share at least one manifold vertex (common vertex is part of exactly two curves)
Parameters

**curve_ids** IDs of curves to query

Returns

whether the curves are adjacent

### ◆ are_adjacent_surfaces()

bool are_adjacent_surfaces ( std::vector< int > *surface_ids* )

return whether two or more surfaces share at least one manifold curve (common curve is part of exactly two surfaces)
Parameters

**surface_ids** IDs of surfaces to query

Returns

whether the surface are adjacent

### ◆ auto_size_needs_to_be_calculated()

bool auto_size_needs_to_be_calculated ( )

Get whether the auto size needs to be calculated. Calculating the auto size may be expensive on complex models. The auto size may be outdated if the model has changed.

### ◆ best_edge_to_collapse_interior_node()

int best_edge_to_collapse_interior_node (int *node_id*)

Finds the best edge to collapse this node along to remove the interior node.
Returns

int - return the id of the edge, 0 if no edge found

### ◆ body()

**CubitInterface::Body** body (int *id_in*)

Gets the body object from an ID.
Parameters

**id_in** The ID of the body

Returns

The body object

### ◆ brick()

**Body** brick (double *width*,
double *depth* = `-1`,
double *height* = `-1`
)

Creates a brick of specified width, depth, and height.
Parameters

`[in]` **width** The width of the brick being created
`[in]` **depth** The depth of the brick being created
`[in]` **height** The height of the brick being created

Returns

A **Body** object of the newly created brick

### ◆ calculate_timestep_estimate()

double calculate_timestep_estimate ( std::string *entity_type*,
std::vector< int > *entity_ids*
)

Calculates time step estimate on elements of/in entity type: "Tet" or "Hex" or "Volume" or "Block" or "Group" The hexes or tets must belong to a single block and that block must have a material assigned to it. That material must have elastic_modulus, poisson_ratio, and density defined.

```
double cubit.calculate_timestep_estimate("volume" , vol_list)
```

Parameters

**entity_type** Specifies the entity type (hex, tet, volume, block, group)
**ids** Specifies the ids of the entity type

Returns

time step estimate (smallest time step)

### ◆ calculate_timestep_estimate_with_props()

double calculate_timestep_estimate_with_props ( std::string *entity_type*,
std::vector< int > *entity_id_list*,
double *density*,
double *youngs_modulus*,
double *poissons_ratio*
)

Calculates time step estimate on elements of/in entity type: "Tet" or "Hex" or "Volume" or "Block" or "Group".

```
double cubit.calculate_timestep_estimate_with_props("volume" , vol_list, 2.7, 70, 0.35 )
```

Parameters

**entity_type** Specifies the entity type (hex, tet, volume, block, group)
**ids** Specifies the ids of the entity type
**density** Specifies the density
**youngs_modulus** Specifies the Young's modulus
**poissons_ratio** Specifies the Poisson's ratio

Returns

time step estimate (smallest time step)

### ◆ clear_drawing_set()

void clear_drawing_set ( const std::string & *set_name* )

Clear a named drawing set (this is for mesh preview)

### ◆ clear_highlight()

void clear_highlight ( )

Clear all entity highlights.

### ◆ clear_picked_list()

void clear_picked_list ( )

Clear the picked list.

### ◆ clear_preview()

void clear_preview ( )

Clear preview graphics without affecting other display settings.

### ◆ cmd()

bool cmd ( const char * *input_string* )

Pass a command string into Cubit.
Passing a command into Cubit using this method will result in an immediate execution of the command. The command is passed directly to Cubit without any validation or other checking.

```
CubitInterface::cmd ("create brick x 10" );
```

CubitInterface::cmd
bool cmd(const char *input_string)
Pass a command string into Cubit.

```
cubit.cmd("brick x 10" )
```

Parameters

**input_string** Pointer to a string containing a complete Cubit command

### ◆ cmd_single()

bool cmd_single ( const char * *input_string* )

Pass a command string into Cubit.
Passing a command into Cubit using this method will result in an immediate execution of the command. The command is passed directly to Cubit without any validation or other checking.

```
CubitInterface::cubit_or_python_cmd ("create brick x 10" );
```

CubitInterface::cubit_or_python_cmd
bool cubit_or_python_cmd(const std::string &input_string)

```
cubit.cubit_or_python_cmd("brick x 10" )
```

Parameters

**input_string** Pointer to a string containing a complete Cubit command

### ◆ command_is_python_mode()

bool command_is_python_mode ( )

Gets whether command handling is in python mode.

### ◆ compare_geometry_and_mesh()

```
void
compare_geometry_and_m
esh                          (std::vector< int >  volume_ids,
                              std::vector< int >  block_ids,
                              std::vector< int >  hex_ids,
                              std::vector< int >  tet_ids,
                              double              tolerance,
                              int &               returned_unmatched_volumes_count,
                                                  returned_unmatched_elements_count
                              int &               ,
                              std::vector< int >  returned_full_matches_group_ids_list
                              &                   ,
                              std::vector< int >  returned_partial_matches_group_ids_
                              &                   list,
                              int &               returned_volume_curves_group_id
                             )
```

Compare the geometry and mesh.

### ◆ complete_filename()

```
void complete_filename(std::string & line,
                       int &         num_chars,
                       bool &        found_quote
                      )
```

Get the file completion inside a quote based on files in the current directory. This handles completion of directories as well as filtering on specific types (.jou, .g, .sat, etc.)

Parameters

**line** [in/out] the line to be completed and the completed line num_chars [out] the number of characters added to the input line. If 0 there are multiple completions found_quote [out] if the end of quote was found

### ◆ contains_virtual()

```
bool contains_virtual(const std::string & geometry_type,
                      int                 entity_id
                     )
```

Query virtualality of an entity's children.

```
      if (CubitInterface::contains_virtual ("surface" , 134)) . . .
```

CubitInterface::contains_virtual

bool contains_virtual(const std::string &geometry_type, int entity_id)

Query virtualality of an entity's children.

```
      if cubit.contains_virtual("surface" , 134)):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

### ◆ convection_is_on_shell_area()

```
bool convection_is_on_shell_area ( int                 entity_id,
                                   CI_BCEntityTypes    shell_area_enum
                                 )
```

Determine whether a BC convection is on a shell top or bottom. Valid for convection.
Parameters

**entity_id** Id of the BC convection
**shell_area** enum of BCEntityTypes. Use 7 to check if on top, 8 to check if on bottom

Returns
true if convection is on the shell area, otherwise false

### ◆ convection_is_on_solid()

```
bool convection_is_on_solid ( int entity_id )
```

Determine whether a BC convection is on a solid. Valid for convection.
Parameters

**entity_id** Id of the BC convection

Returns
true if convection is on a solid, otherwise false

### ◆ copy_body()

```
Body copy_body ( Body init_body )
```

Creates a copy of the input **Body** .
Parameters

[in] **init_body** The **Body** to be copied

Returns
A **Body** identical to the input **Body**

### ◆ create_arc_curve()

```
Curve create_arc_curve ( Vertex                  v0,
                         Vertex                  v1,
                         std::array< double, 3 > intermediate_point
                       )
```

Creates a arc curve using end vertices and an intermediate point.
Parameters

[in] **v0** The start vertex
[in] **v1** The end vertex
[in] **intermediate_point** intermideate coord

Returns
A **Curve** object

◆ create_curve()

**Curve** create_curve(**Vertex** *v0*,
**Vertex** *v1*
)

Creates a curve between two vertices.
Parameters

[in] **v0** The start vertex
[in] **v1** The end vertex

Returns
A **Curve** object

◆ create_new_group()

int create_new_group( )

Create a new group.
Returns
group_id ID of new group

◆ create_spline()

**Curve** create_spline(std::vector< std::array< double, 3 > > *points*,
int *surface_id*
)

create spline through the given 3d points
Parameters

[in] **coordinates** of points on the spline, in order
[in] **id** of the surface on which to create the spline

Returns
the created curve object

◆ create_surface()

**Body** create_surface(std::vector< **Curve** > *curves*)

Creates a surface from boundary curves.
Parameters

[in] **curves** A list of curve objects from which to make the surface

Returns
A **Body** object of the newly created **Surface**

◆ create_vertex()

**Vertex** create_vertex(double *x* = 0,

double *y* = 0,
double *z* = 0
)

Creates a vertex at a x,y,z.
Parameters

[in] **x** The x location of the vertex (default to 0)
[in] **y** The y location of the vertex (default to 0)
[in] **z** The z location of the vertex (default to 0)

Returns
A **Vertex** object

### ◆ cubit_or_python_cmd()

bool cubit_or_python_cmd(const std::string & *input_string*)

### ◆ cubit_or_python_cmds()

bool cubit_or_python_cmds(const std::vector< std::string > & *input_strings*,
std::function< void(const std::string &)> && *history*
)

### ◆ cubit_or_python_cmds_as_file()

bool cubit_or_python_cmds_as_file(const std::vector< std::string > & *input_strings*,
std::function< void(const std::string &)> && *history*
)

### ◆ current_selection_count()

int current_selection_count( )

Get the current count of selected items.

### ◆ curve()

**CubitInterface::Curve** curve(int *id_in*)

Gets the curve object from an ID.
Parameters

**id_in** The ID of the curve

Returns
The curve object

### ◆ cylinder()

**Body** cylinder(double *height*,
double *x_radius*,

double *y_radius*,
double *top_radius*
)

creates a cylinder of the specified dimensions
Parameters

[in] **hi** the height of the cylinder
[in] **r1** radius in the x direction
[in] **r2** radius in the y direction
[in] **r3** used to adjust the top. If set to 0, will produce a cone. If set to the larger of r1 or r2 it will create a straight cylinder.

Returns
A body object of the newly created cylinder

### ◆ delete_all_groups()

void delete_all_groups ( )

Delete all groups.

### ◆ delete_group()

void delete_group ( int *group_id* )

Delete a specific group.
Parameters

**group_id** ID of group to delete

### ◆ destroy()

int destroy ( )

Closes the current journal file.

### ◆ developer_commands_are_enabled()

bool developer_commands_are_enabled ( )

This checks to see whether developer commands are enabled.
Returns
True if developer commands are enabled, otherwise False

### ◆ enable_signal_handling()

void enable_signal_handling ( bool *on* )

initialize/uninitialize signal handling **C++ only**
Parameters

**on** Set to true to initialize signal handling, false to uninitialize.

### ◆ ensure_init()

void ensure_init ( )

◆ entity_exists()

bool entity_exists ( const std::string & *entity_type*,
int *id*
)

return whether an entity of specified ID exists

```
bool exists = CubitInterface::entity_exists ("surface" , 12);
```

CubitInterface::entity_exists
bool entity_exists(const std::string &entity_type, int id)
return whether an entity of specified ID exists

Parameters

**entity_type** Type of the entity being queried
**id** ID of entity

Returns

true of false whether entity exists

◆ estimate_merge_tolerance()

double
estimate_merge_tolerance ( std::vector< int > *target_volume_ids*,
bool *accurate_in* = `false`,
bool *report_in* = `false`,
double *low_value_in* = `-1.0`,
double *high_value_in* = `-1.0`,
int *number_calculations_in* = `10`,
bool *return_calculations_in* = `false`,
std::vector< double > * *merge_tolerance_list* = `NULL`,
*number_of_proximities_list* = `NULL`
std::vector< int > *
)

Estimate a good merge tolerance for the passed-in volumes.
Given a list of volumes try to estimate a good merge tolerance.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |
| **accurate_in** | Flag specifying whether to do a lengthier, more accurate calculation. |
| **report_in** | Flag specifying whether to report results to the command line. |
| **lo_val_in** | Low value of range to search for merge tolerance. |
| **hi_val_in** | High value of range to search for merge tolerance. |

**num_calculations_in** Number of intervals to split search range up into.
**return_calculations_in** Flag specifying whether to return the number of proximities at each step.
**merge_tols** List containing merge tolerance at each step of calculation.
**num_proximities** List containing number of proximities at each step of calculation.

◆ evaluate_exterior_angle()

```
std::vector< int > evaluate_exterior_angle ( const std::vector< int > & curve_list,
                                             const double               test_angle
                                           )
```

find all curves in the given list with an exterior angle (the angle between surfaces) less than the test angle. This is equivalent to the df parser "exterior_angle" test. (draw curve with exterior_angle >90)
Parameters

**curve_list** a list of curve ids (integers)
**test_angle** the value (in degrees) that will be used in testing the exterior angle
Returns
A list (python tuple) of curve ids that meet the angle test.

◆ evaluate_exterior_angle_at_curve()

```
double evaluate_exterior_angle_at_curve ( int curve_id,
                                          int volume_id
                                        )
```

return exterior angle at a single curve with respect to a volume
Parameters

**curve** id (integer) volume id (integer)
Returns
angle in degrees

◆ evaluate_surface_angle_at_vertex()

```
double evaluate_surface_angle_at_vertex ( int surf_id,
                                          int vert_id
                                        )
```

return interior angle at a vertex on a specified surface
Parameters

**surf** id (integer) vert id (integer)
Returns
angle in degrees

◆ exodus_sizing_function_file_exists()

bool exodus_sizing_function_file_exists ( )

return whether the exodus sizing funnction file exists
Returns
whether the exodus sizing function file exists

### ◆ find_floating_volumes()

void find_floating_volumes ( std::vector< int > *target_volume_ids*,
std::vector< int > & *returned_floating_id_list*
)

Get the list of volumes with no merged children.
Given a list of volumes find all of the volumes that are not attached to any other entity through a merge.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |
| **volume_list** | User specified list where the ids of floating volumes are returned |

### ◆ find_nonmanifold_curves()

void find_nonmanifold_curves ( std::vector< int > *target_volume_ids*,
std::vector< int > & *returned_curve_list*
)

Get the list of nonmanifold curves in the volume list.
Given a list of volumes find all of the nonmanifold curves. This is found by seeing if there is at least one merged face attached to any merged curve. If there exist merged curves that don't belong to merged faces it represents a nonmanifold case.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |
| **curve_list** | User specified list where the ids of nonmanifold curves are returned |

### ◆ find_nonmanifold_vertices()

void find_nonmanifold_vertices ( std::vector< int > *target_volume_ids*,
std::vector< int > & *returned_vertex_list*
)

Get the list of nonmanifold vertices in the volume list.
Given a list of volumes find all of the nonmanifold vertices. This is found by seeing if there is at least one merged curve attached to any merged vertex. If there exist merged vertices that don't belong to merged curves it represents a nonmanifold case.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |

**vertex_list** User specified list where the ids of nonmanifold vertices are returned

### ◆ flush_graphics()

void flush_graphics ( )

Flush the graphics.

### ◆ gather_surfaces_by_orientation()

std::vector< int > gather_surfaces_by_orientation ( std::vector< int > *seed_surf_ids*,
std::vector< int > *all_surf_ids*
)

Gathers connected surfaces to those in 'seed_surf_ids' that use common curves in an opposite sense. For example, if a surface A in 'seed_surf_ids' uses a curve in the FORWARD sense and it can find surface, B, that uses that same curve in a REVERSED sense, it adds B to the list. The search continues with all of surface B's curves.
All the surfaces in 'seed_surf_ids' will be returned. If the user wants to limit the scope of possible surfaces that are searched, 'all_surf_ids' can be populated. If 'all_surf_ids' is empty, all surfaces are candidates. This function can be helpful in finding enclosures when you have a set of non-manifold surfaces.

```
       seed_surf_ids = [ 15, 19, 24, 88 ]
       all_surf_ids = [ 15, 19, 24, 88, 26, 104, 44, 23, 95, 342, 533, 23,
... ]
       orientation_surfs = cubit.gather_surfaces_by_orientation(
seed_surf_ids, all_surf_ids )
```

### ◆ get_2D_sheet_volumes()

std::vector< int > get_2D_sheet_volumes ( int *vol_id* )

get the 2D sheet volumes associated with this 3D thin volume
Parameters

**vol_id** volume ID. Should represent a 3D thin volume

Returns

ids of the 2D sheet volumes associated with the 3D thin volume

### ◆ get_3D_thin_volume()

int get_3D_thin_volume ( int *vol_id* )

get the 3D sheet volume associated with this 2D sheet volume
Parameters

**vol_id** volume ID. Should represent a 2D sheet volume

Returns

id of the 3D thin volume associated with the 2D sheet volume

### ◆ get_acceleration_combine_type()

std::string get_acceleration_combine_type(int *entity_id*)

Get the acceleration's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine".
Parameters

**entity_id** Id of the acceleration

Returns

The combine type for the given acceleration

◆ get_acceleration_dof_signs()

const int * get_acceleration_dof_signs(int *entity_id*)

**This function only available from C++** Get the acceleration's dof signs
Parameters

**entity_id** Id of the acceleration

Returns

An array of ints which are the dof signs

◆ get_acceleration_dof_values()

const double * get_acceleration_dof_values(int *entity_id*)

**This function only available from C++** Get the acceleration's dof values
Parameters

**entity_id** Id of the acceleration

Returns

An array of doubles which are the dof values

◆ get_acis_version()

std::string get_acis_version( )

Get the Acis version number.
Returns

A string containing the Acis version number

◆ get_acis_version_as_int()

int get_acis_version_as_int( )

Get the Acis version number as an int.
Returns

An integer containing the Acis version number

◆ get_adjacent_surfaces()

std::vector< int > get_adjacent_surfaces(const std::string & *geometry_type*,
int *entity_id*
)

Get a list of adjacent surfaces to a specified entity.

For a specified entity, find all surfaces that own the entity and surfaces that touch the surface that owns this entity.

```
      std::vector<int> surface_id_list;
      surface_id_list = CubitInterface::get_adjacent_surfaces ("curve" ,
22);
```

CubitInterface::get_adjacent_surfaces

std::vector< int > get_adjacent_surfaces(const std::string &geometry_type, int entity_id)

Get a list of adjacent surfaces to a specified entity.

```
      surface_id_list = cubit.get_adjacent_surfaces("curve" , 22)
```

Parameters

**geom_type** Specifies the geometry type of the entity

**entity_id** Specifies the id of the entity

Returns

A list (python tuple) of surfaces ids

### ◆ get_adjacent_volumes()

std::vector< int > get_adjacent_volumes ( const std::string & *geometry_type*,
int *entity_id*
)

Get a list of adjacent volumes to a specified entity.

For a specified entity, find all volumes that own the entity and volumes that touch the volume that owns this entity.

```
      std::vector<int> volume_id_list;
      volume_id_list = CubitInterface::get_adjacent_volumes ("curve" ,
22);
```

CubitInterface::get_adjacent_volumes

std::vector< int > get_adjacent_volumes(const std::string &geometry_type, int entity_id)

Get a list of adjacent volumes to a specified entity.

```
      volume_id_list = cubit.get_adjacent_volumes("curve" , 22)
```

Parameters

**geom_type** Specifies the geometry type of the entity

**entity_id** Specifies the id of the entity

Returns

A list (python tuple) of volume ids

### ◆ get_all_cfd_bcs()

std::vector< **CFD_BC_Entity** > get_all_cfd_bcs ( )

### ◆ get_all_exodus_times()

std::vector< double > get_all_exodus_times ( const std::string & *filename* )

Open an exodus file and get a vector of all stored time stamps.

Parameters

**filename** Fully qualified exodus file name

Returns

List (python tuple) of time stamps in the exodus file

◆ get_all_exodus_variable_names()

```
std::vector< std::string >
get_all_exodus_variable_names (const std::string & filename,
                               const std::string & variable_type
                              )
```

Open an exodus file and get a list of all stored variable names.

Parameters

**filename** Fully qualified exodus file name

**type** Variable type - 'g', 'n', or 'e'

Returns

List (python tuple) of variable names in the exodus file

◆ get_all_geometric_owners()

```
std::vector< std::string > get_all_geometric_owners (std::string mesh_entity_type,
                                                     std::string mesh_entity_list
                                                    )
```

Get a list of geometric owners given a list of mesh entities. returns geometric owners of entity as well as all of its child mesh entities.

```
      std::vector<std::string> owner_list;
      owner_list = CubitInterface::get_all_geometric_owners ("quad" ,
id_list);
```

CubitInterface::get_all_geometric_owners

std::vector< std::string > get_all_geometric_owners(std::string mesh_entity_type, std::string mesh_entity_list)

Get a list of geometric owners given a list of mesh entities. returns geometric owners of entity as w...

```
      owner_list = cubit.get_all_geometric_owners("quad" , id_list)
```

Parameters

**mesh_entity_type** The type of mesh entity. Works for 'quad, 'face', 'tri', 'hex', 'tet', 'edge', 'node'

**mesh_entity_list** A string containing space delimited ids, Cubit command form (i.e. 'all', '1 to 8', '1 2 3', etc)

Returns

A list (python tuple) of geometry owners in the form of 'surface x', 'curve y', etc.

◆ get_all_ids_from_name()

```
std::vector< int > get_all_ids_from_name (const std::string & geo_type,
                                          const std::string & name
```

)

Get all ids of a geometry type with the prefix given by string.

```
        std::vector<int> entity_ids = CubitInterface::get_all_ids_from_name
("volume" , "member_2" );
```

CubitInterface::get_all_ids_from_name

std::vector< int > get_all_ids_from_name(const std::string &geo_type, const std::string &name)

Get all ids of a geometry type with the prefix given by string.

```
        entity_ids = cubit.get_all_ids_from_name("volume" , "member_2" )
```

Parameters

**geo_type** type of geometry entity (vertex, curve, surface, volume) name Prefix of entity name return vector of integers representing the entities that have the given name as a prefix

## ◆ get_aprepro_numeric_value()

double get_aprepro_numeric_value(std::string *variable_name*)

get the value of the given aprepro variable

Returns

value as double on failure returns CUBIT_DBL_MAX

## ◆ get_aprepro_value()

bool get_aprepro_value(std::string *variable_name*,
int & *returned_variable_type*,
double & *returned_double_val*,
std::string & *returned_string_val*
)

Get the value of an aprepro variable.

Parameters

**var_name** aprepro variable name
**var_type** return 0, 1 or 3 where 0=undefined 1=double/int 2=string
**dval** return integer or double value if var_type=1
**sval** return string if var_type=2

Returns

1 = success, 0 = failure (no such variable name)

## ◆ get_aprepro_value_as_string()

std::string get_aprepro_value_as_string(std::string *variable_name*)

Gets the string value of an aprepro variable.

Parameters

**var_name** aprepro variable name

Returns

The string value of the aprepro variable name

### ◆ get_aprepro_vars()

std::vector< std::string > get_aprepro_vars ( )

Gets the current aprepro variable names.
Returns
A list (python tuple) of the current aprepro variable names

### ◆ get_arc_length()

double get_arc_length ( int *curve_id* )

Get the arc length of a specified curve.
Parameters

**curve_id** ID of the curve
Returns
Arc length of the curve

### ◆ get_assembly_children()

std::vector< **AssemblyItem** > get_assembly_children ( int *assembly_id* )

### ◆ get_assembly_classification_category()

std::string get_assembly_classification_category ( )

Get Classification Category for metadata.
Returns
Requested data

### ◆ get_assembly_classification_level()

std::string get_assembly_classification_level ( )

Get Classification Level for metadata.
Returns
Requested data

### ◆ get_assembly_description()

std::string get_assembly_description ( int *assembly_id* )

Get the stored description of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node
Returns
Description of the assembly node

### ◆ get_assembly_file_format()

std::string get_assembly_file_format ( int *assembly_id* )

Get the stored file format of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node

Returns

File Format of the assembly node

◆ get_assembly_instance()

int get_assembly_instance(int *assembly_id*)

Get the stored instance number of an assembly node.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Instance of the assembly node

◆ get_assembly_items()

std::vector< **AssemblyItem** > get_assembly_items()

◆ get_assembly_level()

int get_assembly_level(int *assembly_id*)

Get the stored level of an assembly node.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Level of the assembly node - Level == 0 == Root

◆ get_assembly_material_description()

std::string get_assembly_material_description(int *assembly_id*)

Get the stored material description of an assembly part.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Material Description of the assembly part

◆ get_assembly_material_specification()

std::string get_assembly_material_specification(int *assembly_id*)

Get the stored material specification of an assembly part.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Material Specification of the assembly part

◆ get_assembly_metadata()

std::string get_assembly_metadata(int *volume_id*,

int *data_type*
)

Get metadata for a specified volume id.
Parameters

**volume_id** ID of the volume
**data_type** Magic number representing the type of assembly information to return. 1 = Part Number, 2 = Description, 3 = Material Description 4 = Material Specification, 5 = Assembly Path, 6 = Original File

Returns
Requested data

### ◆ get_assembly_name()

std::string get_assembly_name(int *assembly_id*)

Get the stored name of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node

Returns
Name of the assembly node

### ◆ get_assembly_path()

std::string get_assembly_path(int *assembly_id*)

Get the stored path of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node

Returns
Path of the assembly node

### ◆ get_assembly_type()

std::string get_assembly_type(int *assembly_id*)

Get the stored type of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node

Returns
Type of the assembly node – 'part' or 'assembly'

### ◆ get_assembly_units()

std::string get_assembly_units(int *assembly_id*)

Get the stored units measure of an assembly node.
Parameters

**assembly_id** Id that identifies the assembly node

Returns
Units of the assembly node

◆ get_assembly_weapons_category()

std::string get_assembly_weapons_category ( )

Get Weapons Category for metadata.
Returns
Requested data

◆ get_auto_size()

double get_auto_size (const std::string & *geometry_type*,
std::vector< int > *entity_id_list*,
double *size*
)

Get the auto size for a given set of enitities. Note, this does not actually set the interval size on the volumes. It simply returns the size that would be set if an 'size auto factor n' command were issued.
Parameters

**entity_type** Specifies the geometry type of the entity
**enitty_id_list** List (vector) of entity ids
**size** The auto factor for the AutoSizeTool

Returns
The interval size from the AutoSizeTool

◆ get_bad_geometry()

void get_bad_geometry (std::vector< int > *target_volume_ids*,
std::vector< int > & *returned_body_list*,
std::vector< int > & *returned_volume_list*,
std::vector< int > & *returned_surface_list*,
std::vector< int > & *returned_curve_list*
)

**This function only works from C++** Get the list of bad geometry for a list of volumes Bad geometry can be any number of problems associated with poorly defined ACIS geometry.
Parameters

**target_volume_ids** List of volume ids to examine.
**body_list** User specified list where ids of bad bodies will be returned
**volume_list** User specified list where ids of bad volumes will be returned
**surface_list** User specified list where ids of bad surfaces will be returned
**curve_list** User specified list where ids of bad curves will be returned

◆ get_bc_id_list()

std::vector< int > get_bc_id_list ( CI_BCTypes *bc_type_enum* )

Get a list of all bcs of a specified type.
Parameters

**bc_type_in** as an enum defined by CI_BCTypes. 1-9 is FEA, 10-30 is CFD

Returns

List (python tuple) of all active bc ids

### ◆ get_bc_name()

std::string get_bc_name ( CI_BCTypes *bc_type_enum*,
int *bc_id*
)

Get the name for the specified bc.
Parameters

**bc_type_in** type of bc, as defined by enum CI_BCTypes. 1-9 is FEA, 10-30 is CFD

**bc_id** ID of the desired bc.

Returns

The bc name

### ◆ get_bc_temperature()

double get_bc_temperature ( CI_BCTypes *bc_type_enum*,
int *entity_id*,
CI_BCEntityTypes *temp_type_enum*
)

Get the temperature. Valid for convection, temperature.
Parameters

**bc_type** enum of CI_BCTypes. temperature = 4, convection = 7

**entity_id** Id of the BC convection

**temp_type** enum of CI_BCEntityTypes (normal, shell top, shell bottom). For convection, 2 if on solid, 7 if on top, 8 if on bottom. For temperature, 3 if on solid, 7 for top, 8 for bottom, 9 for gradient, 10 for middle

Returns

The value of the specified BC temperature

### ◆ get_blend_chain_collections()

std::vector< std::pair< std::vector< int >,
double > > get_blend_chain_collections ( const std::vector< int > & *volume_list*,
double *radius_threshold*
)

Returns the collections of surfaces that comprise blend chains in the specified volumes. Filter by radius threshold.

Parameters

| | |
|---|---|
| **volume_list** | List of volumes to query |
| **radius_threshold** | resturn only blend chains less than radius_threshold |
| **return_radii** | return a vector of blend chain radii corresponding to the return blend chains lists |

Returns

A list of lists of surface id's grouped by their individual blend chain

### ◆ get_blend_chains()

std::vector< std::vector< int > > get_blend_chains(int *surface_id*)

Returns the blend chains for a surface.

Parameters

| | |
|---|---|
| **surface_id** | surface to retrieve the blend chains from |

Returns

A list of lists of id's in each blend chain. Note: If using python, lists will be python tuples.

### ◆ get_blend_surfaces()

std::vector< int > get_blend_surfaces(std::vector< int > *target_volume_ids*)

Get the list of blend surfaces for a list of volumes.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |

Returns

List (python tuple) of blend surface ids

### ◆ get_block_attribute_count()

int get_block_attribute_count(int *block_id*)

Get the number of attributes in a block.

Parameters

| | |
|---|---|
| **block_id** | The block id |

Returns

Number of attributes in the block

### ◆ get_block_attribute_name()

std::string get_block_attribute_name(int *block_id*,
int *attribute_index*
)

Get a specific block attribute name.

Parameters

| | |
|---|---|
| **block_id** | The block id |
| **index** | The index of the attribute |

Returns
Attribute name as a std::string

### ◆ get_block_attribute_value()

```
double get_block_attribute_value ( int block_id,
                                   int attribute_index
                                 )
```

Get a specific block attribute value.
Parameters

**block_id** The block id
**index** The index of the attribute

Returns
List of attributes

### ◆ get_block_children()

```
void get_block_children ( int                        block_id,
                          std::vector< int > &       returned_group_list,
                          std::vector< int > &       returned_node_list,
                          std::vector< int > &       returned_sphere_list,
                          std::vector< int > &       returned_edge_list,
                          std::vector< int > &       returned_tri_list,
                          std::vector< int > &       returned_face_list,
                          std::vector< int > &       returned_pyramid_list,
                          std::vector< int > &       returned_tet_list,
                          std::vector< int > &       returned_hex_list,
                          std::vector< int > &       returned_wedge_list,
                          std::vector< int > &       returned_volume_list,
                          std::vector< int > &       returned_surface_list,
                          std::vector< int > &       returned_curve_list,
                          std::vector< int > &       returned_vertex_list
                        )
```

Get lists of any and all possible children of a block.
A block can contain a variety of entity types. This routine will return all contents of a specified block.
Parameters

| | |
|---|---|
| **block_id** | ID of block to examine |
| **group_list** | User specified list where groups associated with this block are returned |
| **node_list** | User specified list where nodes associated with this block are returned |
| **edge_list** | User specified list where edges associated with this block are returned |

| | |
|---|---|
| **tri_list** | User specified list where tris associated with this block are returned |
| **face_list** | User specified list where faces associated with this block are returned |
| **pyramid_list** | User specified list where pyramids associated with this block are returned |
| **tet_list** | User specified list where tets associated with this block are returned |
| **hex_list** | User specified list where hexes associated with this block are returned |
| **volume_list** | User specified list where volumes associated with this block are returned |
| **surface_list** | User specified list where surfaces associated with this block are returned |
| **curve_list** | User specified list where curves associated with this block are returned |
| **vertex_list** | User specified list where vertices associated with this block are returned |

### ◆ get_block_count()

int get_block_count ( )

Get the current number of blocks.

Returns

The number of blocks in the current model, if any

### ◆ get_block_curves()

std::vector< int > get_block_curves ( int *block_id* )

Get a list of curve associated with a specific block.

Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of curve ids contained in the block

### ◆ get_block_edges()

std::vector< int > get_block_edges ( int *block_id* )

Get a list of edges associated with a specific block.

Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of edge ids contained in the block

### ◆ get_block_element_attribute_count()

int get_block_element_attribute_count ( int *block_id* )

Get the number of attributes in a block element.
Parameters

**block_id** The block id

Returns

Number of attributes in the block element

◆ get_block_element_attribute_names()

std::vector< std::string > get_block_element_attribute_names(int *block_id*)

Get a specific block element attribute name.
Parameters

**block_id** The block id

**index** The index of the attribute

Returns

Attribute name as a std::string

◆ get_block_element_type()

std::string get_block_element_type(int *block_id*)

Get the element type of a block.
Parameters

**block_id** The block id

Returns

Element type

◆ get_block_elements_and_nodes()

```
bool get_block_elements_and_nodes(int                    block_id,
                                  std::vector< int > &   returned_node_list,
                                  std::vector< int > &   returned_sphere_list,
                                  std::vector< int > &   returned_edge_list,
                                  std::vector< int > &   returned_tri_list,
                                  std::vector< int > &   returned_face_list,
                                  std::vector< int > &   returned_pyramid_list,
                                  std::vector< int > &   returned_wedge_list,
                                  std::vector< int > &   returned_tet_list,
                                  std::vector< int > &   returned_hex_list
                                 )
```

Get lists of the nodes and different element types associated with this block. This function is recursive, meaning that if the block was created pointing to a piece of geometry, it will traverse down and get the mesh entities associated to that geometry.
Parameters

**block_id** User specified id of the desired block A list (python tuple) of node ids contained in the block A list (python tuple) of edge ids contained in the

block A list (python tuple) of tri ids contained in the block A list (python tuple) of quad ids contained in the block A list (python tuple) of pyramid ids contained in the block A list (python tuple) of wedge ids contained in the block A list (python tuple) of tet ids contained in the block A list (python tuple) of hex ids contained in the block

Returns

true for success, otherwise false

### ◆ get_block_faces()

std::vector< int > get_block_faces ( int *block_id* )

Get a list of faces associated with a specific block.

Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of face ids contained in the block

### ◆ get_block_hexes()

std::vector< int > get_block_hexes ( int *block_id* )

Get a list of hexes associated with a specific block.

Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of hex ids contained in the block

### ◆ get_block_id()

int get_block_id ( std::string *entity_type*,
int *entity_id*
)

Get the associated block id for a specific curve, surface, or volume.

```
int block_id = CubitInterface::get_block_id ("surface" , 33);
```

CubitInterface::get_block_id

int get_block_id(std::string entity_type, int entity_id)

Get the associated block id for a specific curve, surface, or volume.

```
block_id = cubit.get_block_id("surface" , 33)
```

Parameters

**entity_type** Type of entity

**entity_id** Id of entity in question

Returns

Block id associated with this entity or zero (0) if none

### ◆ get_block_id_list()

std::vector< int > get_block_id_list ( )

Get a list of all blocks.
Returns
List (python tuple) of all active block ids

◆ get_block_ids()

std::vector< int > get_block_ids(const std::string & *mesh_geometry_file_name*)

Get list of block ids from a mesh geometry file.
Parameters

**mesh_geom_file_name** Fully qualified name of a mesh geometry file
Returns
List of block ids in the mesh geometry file

◆ get_block_material()

int get_block_material(int *block_id*)

Get the id of the material assigned to the specified block.
Returns
The material id. If no material has been assigned to the block, returns 0.

◆ get_block_nodes()

std::vector< int > get_block_nodes(int *block_id*)

Get a list of nodes associated with a specific block.
Parameters

**block_id** User specified id of the desired block
Returns
A list (python tuple) of node ids contained in the block

◆ get_block_pyramids()

std::vector< int > get_block_pyramids(int *block_id*)

Get a list of pyramids associated with a specific block.
Parameters

**block_id** User specified id of the desired block
Returns
A list (python tuple) of pyramid ids contained in the block

◆ get_block_surfaces()

std::vector< int > get_block_surfaces(int *block_id*)

Get a list of surface associated with a specific block.
Parameters

**block_id** User specified id of the desired block
Returns
A list (python tuple) of surface ids contained in the block

### ◆ get_block_tets()

std::vector< int > get_block_tets(int *block_id*)

Get a list of tets associated with a specific block.
Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of tet ids contained in the block

### ◆ get_block_tris()

std::vector< int > get_block_tris(int *block_id*)

Get a list of tris associated with a specific block.
Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of tri ids contained in the block

### ◆ get_block_vertices()

std::vector< int > get_block_vertices(int *block_id*)

Get a list of vertices associated with a specific block.
Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of vertex ids contained in the block

### ◆ get_block_volumes()

std::vector< int > get_block_volumes(int *block_id*)

Get a list of volume ids associated with a specific block.
Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of volume ids contained in the block

### ◆ get_block_wedges()

std::vector< int > get_block_wedges(int *block_id*)

Get a list of wedges associated with a specific block.
Parameters

**block_id** User specified id of the desired block

Returns

A list (python tuple) of wedges ids contained in the block

### ◆ get_blocks_with_materials()

std::vector< std::vector< int > > get_blocks_with_materials ( )

Get the block ids and ids of the respective materials assigned to each block.
Returns
List of tuples ([block_1_id, material_1_id], [block_2_id, material_2_id], ...) for each block, whether or not it has a material. If no material has been assigned to the block, returns 0.

### ◆ get_blunt_tangency_default_depth()

double get_blunt_tangency_default_depth ( int *vert_id*,
double *angle*,
bool *add_material*
)

get default depth value for blunt tangency operation
Returns
depth

### ◆ get_body_count()

int get_body_count ( )

Get the current number of bodies.
Returns
The number of bodies in the current model, if any

### ◆ get_bolt_axis()

std::vector< double > get_bolt_axis ( int *vol_id* )

get axis vector of bolt
Parameters

**vol_id** volume ID. Should represent bolt geometry
Returns
normalized axis vector of bolt ([0,0,0] if invalid volume or couldn't determine)

### ◆ get_bolt_clamped_members()

std::vector< int > get_bolt_clamped_members ( int *vol_id*,
std::vector< int > *nearby_vols*
)

return the clamped members associated with a bolt in order: bearing surface member, sandwiched members (if any), threaded member
Parameters

**vol_id** volume ID. Should represent bolt geometry
**nearby_vols** (optional) list of volumes to select from. If empty, it will compute these, however is less efficient

Returns
list of ordered clamped members as volume IDs

### ◆ get_bolt_coordinate_system()

```
std::vector< std::vector< double > >
get_bolt_coordinate_system (std::string geom_type,
                            int         id
                           )
```

get the local coordinate system for a bolt (or bolt hole)
Parameters

| | |
|---|---|
| **geom_type** | either "bolt" or "hole". geometry we are returning the coordinate from |
| **id** | id of a bolt volume or id of one of the surfaces in the hole geometry |

Returns
coordinate system as three point defined as double vectors

```
coord_system = cubit.get_bolt_coord_system("bolt" , bolt_id)
origin = coord_system[0]
z_point = coord_system[1]
xz_point = coord_system[2]
```

### ◆ get_bolt_diameter()

```
double get_bolt_diameter(int vol_id)
```

get diameter of bolt shank
Parameters

| | |
|---|---|
| **vol_id** | volume ID. Should represent bolt geometry |

Returns
diameter of bolt (0 if invalid volume or couldn't determine)

### ◆ get_bolt_holes()

```
std::vector< std::vector< int > > get_bolt_holes(std::string          geo_type,
                                                 std::vector< int > clamped_members,
                                                 double             radius_threshold,
                                                 double             gap_threshold
                                                )
```

returns the surfaces from upper and lower holes from the specified clamped volumes or blocks. Results can be used directly in the reduce bolt commands.
Parameters

| | |
|---|---|
| **geo_type** | "volume" or "block" |
| **clamped_members** | list of volume IDs that may have one or more concentic holes used as pilot holes for fasteners |
| **radius_threshold** | looks for holes with radius less than or equal to this value |
| **gap_threshold** | looks for concentric holes at neighboring volumes that are closer than this value |

Returns

pair of lists of surfaces. First list contains the upper holes and the second contains the lower holes. Only one surface per hole is included in the list. There will be at least one coaxial hole surface in in the lower that matches the upper. For case of sandwiched volumes, there may be additional lower.

### ◆ get_bolt_holes_info()

```
std::vector< BoltHoleInfo > get_bolt_holes_info(std::string          geo_type,
                                                std::vector< int >   clamped_members,
                                                double               radius_threshold,
                                                double               gap_threshold
                                               )
```

return the pilot hole definitions from a list of volumes. Returns a vector of BoltHole structs

Parameters

| | |
|---|---|
| **geo_type** | "volume" or "block" |
| **clamped_members** | list of volume IDs that may have one or more concentic holes used as pilot holes for fasteners |
| **radius_threshold** | looks for holes with radius less than or equal to this value |
| **gap_threshold** | looks for concentric holes at neighboring volumes that are closer than this value |

Returns

list of BoltHoleInfo structs. Describes the ordered list of hole surfaces, diameters, axis and associated clamped volumes

### ◆ get_bolt_shigley_radius()

```
double get_bolt_shigley_radius(int      vol_id,
                               double   angle
                              )
```

get the equivalent shigley's frustum radius at the interface surfaces between upper and lower volumes for a bolt

Parameters

| | |
|---|---|
| **vol_id** | volume ID. Should represent bolt geometry |
| **angle** | Angle used to define the Shigley's frustum. 30 degrees is standard |

Returns

radius of a circle at the bolt. (-1 if error)

### ◆ get_bolts_in_clamped_members()

```
std::vector< int > get_bolts_in_clamped_members(std::string          geo_type,
                                                std::vector< int >   clamped_vols,
                                                std::vector< int >   candidate_bolts
                                               )
```

return the bolts that are common to the clamped members Note: uses ML classification to determine "bolt" category
Parameters

| | |
|---|---|
| **geo_type** | "volume" or "block". |
| **clamped_vols** | list of volume or block IDs that are clamped by one or more bolts |
| **candidate_bolts** | optional list of bolts to choose from. Note that the ML classification will be run on nearby volumes to determine if they are bolts. This vector is useful if the classification of parts has already been done and there are known list of bolts in the volume. Avoids having to recompute ML classifications. |

Returns
list of volume IDs that are classified as bolts that clamp the given volumes or blocks.

◆ get_bool_sculpt_default()

bool get_bool_sculpt_default(const char * *variable*)

◆ get_boundary_layer_algorithm()

std::string get_boundary_layer_algorithm(int *boundary_layer_id*)

◆ get_boundary_layer_aspect_first_parameters()

```
bool
get_boundary_layer_aspect_first_parameters    (int          boundary_layer_id,
                                               double &     returned_first_row_aspect,
                                               double &     returned_growth_factor,
                                               int &        returned_number_rows
                                              )
```

◆ get_boundary_layer_aspect_last_parameters()

```
bool
get_boundary_layer_aspect_last_parameters    (int          boundary_layer_id,
                                              double &     returned_first_row_height,
                                              int &        returned_number_rows,
                                              double &     returned_last_row_aspect
                                             )
```

◆ get_boundary_layer_continuity()

bool get_boundary_layer_continuity(int *boundary_layer_id*)

◆ get_boundary_layer_curve_intersection_types()

```
bool
get_boundary_layer_curve_intersection_types (std::vector< int > & returned_curve_list,
                                             std::vector< int > & returned_volume_list,
                                             std::vector< std::string > & returned_types
                                            )
```

### ◆ get_boundary_layer_curve_surface_pairs()

```
bool
get_boundary_layer_curve_surface_pairs (int boundary_layer_id,
                                        std::vector< int > & returned_curve_list,
                                        std::vector< int > & returned_surface_list
                                       )
```

### ◆ get_boundary_layer_id_list()

```
std::vector< int > get_boundary_layer_id_list ( )
```

### ◆ get_boundary_layer_surface_volume_pairs()

```
bool
get_boundary_layer_surface_volume_pairs (int boundary_layer_id,
                                         std::vector< int > & returned_surface_list,
                                         std::vector< int > & returned_volume_list
                                        )
```

### ◆ get_boundary_layer_uniform_parameters()

```
bool get_boundary_layer_uniform_parameters (int boundary_layer_id,
                                            double & returned_first_row_height,
                                            double & returned_growth_factor,
                                            int & returned_number_rows
                                           )
```

### ◆ get_boundary_layer_vertex_intersection_types()

```
bool
get_boundary_layer_vertex_intersection_types (std::vector< int > & returned_vertex_list,
                                              std::vector< int > & returned_surface_list,
                                              std::vector< std::string > & returned_types
                                             )
```

### ◆ get_boundary_layers_by_base()

std::vector< int > get_boundary_layers_by_base ( const std::string & *base_type*,
int *base_id*
)

◆ get_boundary_layers_by_pair()

std::vector< int > get_boundary_layers_by_pair ( const std::string & *base_type*,
int *base_id*,
int *parent_id*
)

◆ get_bounding_box()

std::array< double, 10 > get_bounding_box ( const std::string & *geometry_type*,
int *entity_id*
)

Get the bounding box for a specified entity.

```
std::array<double, 10> vector_list;
vector_list = CubitInterface::get_bounding_box ("surface" , 22);
```

CubitInterface::get_bounding_box
std::array< double, 10 > get_bounding_box(const std::string &geometry_type, int entity_id)
Get the bounding box for a specified entity.

```
vector_list = cubit.get_bounding_box("surface" , 22)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

A vector (python tuple) of coordinates describing the entity's bounding box. Ten (10) values will be: [0] = minx [1] = maxx [2] = boxx range [3] = miny [4] = maxy [5] = boxy range [6] = minz [7] = maxz [8] = boxz range [9] = box diagonal length

◆ get_build_number()

std::string get_build_number ( )

Get the Cubit build number.
Returns

A string containing the current Cubit build number

◆ get_cavity_surfaces()

std::vector< int > get_cavity_surfaces ( int *surface_id* )

Returns the adjacent surfaces in a cavity for a surface.
Parameters

**surface_id** that is part of the cavity

Returns
A list of surface id's in the cavity (including surface_id).

◆ get_center_point()

std::array< double, 3 > get_center_point ( const std::string & *entity_type*,
int *entity_id*
)

Get the center point of a specified entity.

```
std::array<double,3> center_point;
center_point = CubitInterface::get_center_point ("surface" , 22);
```

CubitInterface::get_center_point
std::array< double, 3 > get_center_point(const std::string &entity_type, int entity_id)
Get the center point of a specified entity.

```
center_point = cubit.get_center_point("surface" , 22)
```

Parameters

**entity_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns
Vector (python tuple) of doubles representing x y z

◆ get_cfd_type()

int get_cfd_type ( int *entity_id* )

Get the cfd subtype for a specified cfd BC.
Parameters

**entity_id** ID of the cfd BC

Returns
Integer corresponding to the type of cfd, as defined by CI_BCTypes

◆ get_chamfer_chain_collections()

std::vector< std::pair< std::vector< int >, double > > get_chamfer_chain_collections ( const std::vector< int > & *volume_list*,
double *thickness_threshold*
)

Returns the collections of surfaces that comprise chamfers in the specified volumes. Filter by thickness of chamfer.
Parameters

**volume_list** List of volumes to query
**radius_threshold** resturn only chamfer chains less than thickness_threshold
**return_radii** return a vector of chamfer chain radii corresponding to the return chamfer chains lists

Returns

A list of lists of surface id's grouped by their individual chamfer_chain

### ◆ get_chamfer_chains()

std::vector< std::vector< int > > get_chamfer_chains(int *surface_id*)

Returns the chamfer chains for a surface.
Parameters

**surface_id** surface to retrieve the chamfer chains from

Returns

A list of lists of id's in each chamfer chain. Note: If using python, lists will be python tuples.

### ◆ get_chamfer_surfaces()

std::vector< std::vector< double > >
get_chamfer_surfaces (std::vector< int > *target_volume_ids*,
double *thickness_threshold*
)

Get the list of chamfer surfaces for a list of volumes.
Parameters

**target_volume_ids** List of volume ids to examine.
**thickness_threshold** max thickness criteria for chamfer

Returns

List (python tuple) of chamfer surface ids (as doubles) and their thicknesses

### ◆ get_close_loop_thickness()

double get_close_loop_thickness(int *surface_id*)

Get the thickness of a close loop surface.
Parameters

**surafce** id

Returns

List (python tuple) of close loop (surface) ids

### ◆ get_close_loops()

std::vector< int > get_close_loops(std::vector< int > *target_volume_ids*,
double *mesh_size*
)

Get the list of close loops (surfaces) for a list of volumes.
'Small' or 'Close' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.
Parameters

**target_volume_ids** List of volume ids to examine.

**mesh_size** Indicate the mesh size used as the threshold

Returns

List (python tuple) of close loop (surface) ids

◆ get_close_loops_with_thickness()

```
std::vector< std::vector< double > >
get_close_loops_with_thickness (std::vector< int > target_volume_ids,
                                double             mesh_size,
                                int                genus
                               )
```

Get the list of close loops (surfaces) for a list of volumes also return the corresponding minimum distances for each surface.
'Small' or 'Close' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.

Parameters

**target_volume_ids** List of volume ids to examine.

**mesh_size** Indicate the mesh size used as the threshold

**genus** Indicate the genus of the surfaces requested. Genus is defined as the number of loops on the surface minus 1. To return any genus surface in the volume(s), use genus < 0

Returns

List (python tuple) of close loop (surface) ids

◆ get_close_vertex_curve_pairs()

```
std::vector< int > get_close_vertex_curve_pairs (std::vector< int > target_volume_ids,
                                                 double             high_tolerance
                                                )
```

Get the list of close vertex-curve pairs (python callable)

Parameters

**target_volume_list** List of volumes ids to examine.

Returns

Paired list (python tuple) of vertex and curve ids considered coincident

◆ get_closed_narrow_surfaces()

```
std::vector< int > get_closed_narrow_surfaces (std::vector< int > target_ids,
                                               double             narrow_size
                                              )
```

Get the list of closed, narrow surfaces from a list of volumes.

Parameters

**target_volume_ids** List of volume ids to examine.

**narrow_size** Indicate the narrow size threshold

Returns
List (python tuple) of close, narrow surface ids

### ◆ get_closest_node()

```
int get_closest_node ( double x_coordinate,
                       double y_coordinate,
                       double z_coordinate
                     )
```

Get the node closest to the given coordinates.
Parameters

**x** coordinate
**y** coordinate
**z** coordinate

Returns
id of closest node, 0 if none found

### ◆ get_closest_vertex_curve_pairs()

```
void
get_closest_vertex_curve_pairs ( std::vector< int >          target_ids,
                                 int &                       returned_number_to_return,
                                 std::vector< int > &        returned_vertex_ids,
                                 std::vector< int > &        returned_curve_ids,
                                 std::vector< double > &     returned_distances
                               )
```

Find the n closest vertex pairs in the model.
Given a list of volumes find the n closest vertex curve pairs. The checks will be done on a surface by surface basis so that only curve-vertex pairs within a given surface will be returned. This function is for finding the smallest features within the surfaces of the model.
Parameters

| | |
|---|---|
| **target_ids** | List of volumes ids to examine. |
| **num_to_return** | Number of vertex curve pairs to return. |
| **vert_ids** | Ids of returned vertices. |
| **curve_ids** | Ids of returned curves. |
| **distances** | Vertex-curve pair distances. |

### ◆ get_coincident_entity_pairs()

```
void
get_coincident_entity_pairs ( std::vector< int >          target_volume_ids,
                              std::vector< int > &        returned_v_v_vertex_list,
                              std::vector< int > &        returned_v_c_vertex_list,
```

```
                                    std::vector< int > &          returned_v_c_curve_list,
                                    std::vector< int > &          returned_v_s_vertex_list,
                                    std::vector< int > &          returned_v_s_surf_list,
                                    std::vector< double >
                                    &                             returned_vertex_distance_list,
                                    std::vector< double >
                                    &                             returned_curve_distance_list,
                                    std::vector< double >
                                    &                             returned_surf_distance_list,
                                    double                        low_value,
                                    double                        high_value,
                                    bool                          do_vertex_vertex = true,
                                    bool                          do_vertex_curve = true,
                                    bool                          do_vertex_surf = true,
                                                                  filter_same_volume_cases =
                                    bool                          false
                                  )
```

Get the list of coincident vertex-vertex, vertex-curve, and vertex-surface pairs and distances from a list of volumes.

Given a list of volumes get lists of coincident vertex-vertex, vertex-curve, and vertex-surface pairs and their distances based on the passed-in thresholds. The returned lists will be exactly double the size of the distance lists. For each distance, 2 entities will be associated at the same relative place in the list.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |
| **do_vertex_vertex** | Parameter specifying whether to do vertex-vertex check. |
| **do_vertex_curve** | Parameter specifying whether to do vertex-curve check. |
| **do_vertex_surf** | Parameter specifying whether to do vertex-surface check. |
| **v_v_vertex_list** | User specified list where the ids of coincident vertex pairs are returned |
| **v_c_vertex_list** | User specified list where the ids of the vertices of coincident vertex-curve pairs are returned |
| **v_c_curve_list** | User specified list where the ids of the curves of coincident vertex-curve pairs are returned |
| **v_s_vertex_list** | User specified list where the ids of the vertices of coincident vertex-surface pairs are returned |
| **v_s_surf_list** | User specified list where the ids of the surfaces of coincident vertex-surface pairs are returned |
| **vertex_distance_list** | User specified list where the vertex-vertex distance values will be returned |

| | |
|---|---|
| **curve_distance_list** | User specified list where the vertex-curve distance values will be returned |
| **surf_distance_list** | User specified list where the vertex-surface distance values will be returned |
| **low_value** | User specified low threshold value |
| **hi_value** | User specified high threshold value |
| **filter_same_volume_cases** | Parameter specifying whether to weed out entity pairs that are in the same volume. |

◆ get_coincident_vertex_curve_pairs()

```
void
get_coincident_vertex_curve_
pairs                          (std::vector< int >          target_volume_ids,
                                std::vector< int > &        returned_vertex_list,
                                std::vector< int > &        returned_curve_list,
                                std::vector< double >
                                &                           returned_distance_list,
                                double                      low_value,
                                double                      threshold_value,
                                                            filter_same_volume_cases =
                                bool                        false
                               )
```

Get the list of coincident vertex/curve pairs and distances from a list of volumes.
Given a list of volumes get a list of coincident vertex/curve pairs and their distances based on the current merge tolerance value and a threshold value. The returned lists will be of equal length and matched by order.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of vertices ids to examine. |
| **vertex_list** | User specified list for the ids of coincident vertices |
| **curve_list** | User specified list for the ids of coincident curves |
| **distance_list** | User specified list where the distance values will be returned |
| **threshold_value** | User specified threshold value |

◆ get_coincident_vertex_surface_pairs()

```
void
get_coincident_vertex_surface_
pairs                          (std::vector< int >          target_volume_ids,
                                std::vector< int > &        returned_vertex_list,
                                std::vector< int > &        returned_surface_list,
                                std::vector< double
                                > &                         returned_distance_list,
                                double                      low_value,
                                double                      threshold_value,
```

| | |
|---|---|
| | *filter_same_volume_cases* = |
| bool | `false` |
| ) | |

Get the list of coincident vertex/surface pairs and distances from a list of volumes.
Given a list of volumes get a list of coincident vertex/pairs pairs and their distances based on the current merge tolerance value and a threshold value. The returned lists will be of equal length and matched by order.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of vertices ids to examine. |
| **vertex_list** | User specified list for the ids of coincident vertices |
| **surface_list** | User specified list for the ids of coincident surfaces |
| **distance_list** | User specified list where the distance values will be returned |
| **threshold_value** | User specified threshold value |

◆ get_coincident_vertex_vertex_pairs()

| | | |
|---|---|---|
| void get_coincident_vertex_vertex_pairs | (std::vector< int > | *target_volume_ids*, |
| | std::vector< int > & | *returned_vertex_pair_list*, |
| | std::vector< double > & | *returned_distance_list*, |
| | double | *low_value*, |
| | double | *threshold_value*, |
| | | *filter_same_volume_cases* = |
| | bool | `false` |
| | ) | |

Get the list of coincident vertex pairs and distances from a list of volumes.
Given a list of volumes get a list of coincident vertex pairs and their distances based on the current merge tolerance value and a threshold. The returned vertex list will be exactly double the size of the distance list. For each distance, 2 vertices will be associated at the same relative place in the list.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volumes ids to examine. |
| **vertex_pair_list** | User specified list where the ids of coincident vertex pairs be returned |
| **distance_list** | User specified list where the distance values will be returned |
| **threshold_value** | User specified threshold value |

◆ get_coincident_vertices()

| | | |
|---|---|---|
| std::vector< int > get_coincident_vertices | (std::vector< int > | *target_volume_ids*, |
| | double | *high_tolerance* |
| | ) | |

Get the list of coincident vertex pairs
Parameters

**target_volume_list** List of volumes ids to examine.

Returns

Paired list (python tuple) of vertex ids considered coincident

### ◆ get_command_from_history()

std::string get_command_from_history(int *command_number*)

Get a specific command from Cubit's command history buffer.
Returns

A string which is the command at the given index

### ◆ get_command_history_count()

int get_command_history_count( )

### ◆ get_common_curve_id()

int get_common_curve_id(int *surface_1_id*,
int *surface_2_id*
)

Given 2 surfaces, get the common curve id.
Parameters

**surface_1_id** The id of one of the surfaces
**surface_2_id** The id of the other surface

Returns

The id of the curve common to the two surfaces

### ◆ get_common_vertex_id()

int get_common_vertex_id(int *curve_1_id*,
int *curve_2_id*
)

Given 2 curves, get the common vertex id.
Parameters

**curve_1_id** The id of one of the curves
**curve_2_id** The id of the other curves

Returns

The id of the vertex common to the two curves, 0 if there is none

### ◆ get_cone_surfaces()

std::vector< int > get_cone_surfaces(std::vector< int > *target_volume_ids*)

return a list of surfaces that are cones defined by a conic surface and a hard point
Parameters

**target_volume_ids** List of volume ids to examine.

◆ get_connected_surfaces()

std::vector< int > get_connected_surfaces ( std::vector< int > *surf_ids* )

Get a connected set of surfaces to the ones specified.
Given the surfaces ids specified, finds all connected surfaces. The surfaces are first grouped into 'patches' of connected surfaces (surfaces sharing common curves). In all cases below, merged surfaces are excluded. Depending on the patches, the ensuing behavior is:

1. If a single patch is passed in, all connected surfaces are found, excluding any merged surfaces.
2. If two patches are passed in, get all surfaces between the two patches.
3. If more than one patch is found, return an empty list.

Parameters

**surface_ids** IDs of surfaces to start with

Returns

list of IDs of connected surfaces

◆ get_connectivity()

std::vector< int > get_connectivity ( const std::string & *entity_type*,
int *entity_id*
)

Get the list of node ids contained within a mesh entity.

```
std::vector<int> node_id_list;
node_id_list = CubitInterface::get_connectivity ("hex" , 221);
```

CubitInterface::get_connectivity
std::vector< int > get_connectivity(const std::string &entity_type, int entity_id)
Get the list of node ids contained within a mesh entity.

```
node_id_list = cubit.get_connectivity("hex" , 221)
```

Parameters

**entity_type** The mesh element type
**entity_id** The mesh element id

Returns

List (python tuple) of node ids

◆ get_constraint_dependent_entity_point()

std::string get_constraint_dependent_entity_point ( int *constraint_id* )

Get the dependent entity of a specified constraint.
Parameters

**constraint_id** ID of the constraint

Returns
A std::string indicating the dependent entity

◆ get_constraint_reference_point()

std::string get_constraint_reference_point(int *constraint_id*)

Get the reference point of a specified constraint.
Parameters

**constraint_id** ID of the constraint
Returns
A std::string indicating the reference point

◆ get_constraint_type()

std::string get_constraint_type(int *constraint_id*)

Get the type of a specified constraint.
Parameters

**constraint_id** ID of the constraint
Returns
A std::string indicating the type – Kinematic, Distributing, Rigidbody

◆ get_contact_pair_exterior_state()

bool get_contact_pair_exterior_state(int *entity_id*)

Get the contact pair's exterior state.
Parameters

**entity_id** Id of the contact pair
Returns
The exterior state of the contact pair

◆ get_contact_pair_friction_value()

double get_contact_pair_friction_value(int *entity_id*)

Get the contact pair's friction value.
Parameters

**entity_id** Id of the contact pair
Returns
The friction value of the contact pair

◆ get_contact_pair_general_state()

bool get_contact_pair_general_state(int *entity_id*)

Get the contact pair's general state.
Parameters

**entity_id** Id of the contact pair
Returns

The general state of the contact pair

### ◆ get_contact_pair_tied_state()

bool get_contact_pair_tied_state(int *entity_id*)

Get the contact pair's tied state.
Parameters

**entity_id** Id of the contact pair
Returns
The tied state of the contact pair

### ◆ get_contact_pair_tol_lower_value()

double get_contact_pair_tol_lower_value(int *entity_id*)

Get the contact pair's lower bound tolerance value.
Parameters

**entity_id** Id of the contact pair
Returns
The tolerance value of the contact pair

### ◆ get_contact_pair_tolerance_value()

double get_contact_pair_tolerance_value(int *entity_id*)

Get the contact pair's upper bound tolerance value.
Parameters

**entity_id** Id of the contact pair
Returns
The tolerance value of the contact pair

### ◆ get_continuous_curves()

std::vector< int > get_continuous_curves(int *curve_id*,
double *angle_tol*
)

Returns the adjacent curves that are continuous (angle is 180 degrees +- angle_tol)
Parameters

**curve_id** that is part of the cavity
**angle_tol** angle tolerance for continuity
Returns
A list of curve id's in the continuous set (including curve_id).

### ◆ get_continuous_surfaces()

std::vector< int > get_continuous_surfaces(int *surface_id*,
double *angle_tol*
)

Returns the adjacent surfaces that are continuous (exterior angle is 180 degrees +- angle_tol)
Parameters

**surface_id** that is part of the cavity
**angle_tol** angle tolerance for continuity

Returns
A list of surface id's in the continuous set (including surface_id).

### ◆ get_convection_coefficient()

```
double get_convection_coefficient ( int                    entity_id,
                                    CI_BCEntityTypes       bc_type_enum
                                  )
```

Get the convection coefficient.
Parameters

**entity_id** Id of the BC convection
**cc_type** enum of CI_BCEntityTypes (1-normal, 5-shell top, 6-shell bottom)

Returns
The value of the convection coefficient

### ◆ get_coordinate_systems_id_list()

```
std::vector< int > get_coordinate_systems_id_list ( )
```

get a list of coordinate system ids
Returns
List (python tuple) of ids

### ◆ get_copy_block_on_geometry_copy_setting()

```
std::string get_copy_block_on_geometry_copy_setting ( )
```

Get the copy nodeset on geometry copy setting.
Returns
copy nodeset setting

### ◆ get_copy_nodeset_on_geometry_copy_setting()

```
std::string get_copy_nodeset_on_geometry_copy_setting ( )
```

Get the copy nodeset on geometry copy setting.
Returns
copy nodeset setting

### ◆ get_copy_sideset_on_geometry_copy_setting()

```
std::string get_copy_sideset_on_geometry_copy_setting ( )
```

Get the copy nodeset on geometry copy setting.
Returns
copy nodeset setting

### ◆ get_cubit_digits_setting()

double get_cubit_digits_setting ( )

Get the Cubit digits setting.
Returns
A double containing the digits. -1 if no digits are set

### ◆ get_cubit_message_handler()

CubitMessageHandler * get_cubit_message_handler ( )

get the default message handler

### ◆ get_current_ids()

std::vector< int > get_current_ids ( const std::string & *entity_type* )

Get the current body ids.
Returns
The body IDs in the current model, if any

### ◆ get_current_journal_file()

std::string get_current_journal_file ( )

Gets the current journal file name.
Returns
The current journal file name.

### ◆ get_curve_bias_coarse_size()

double get_curve_bias_coarse_size ( int *curve_id* )

Get the bias coarse size of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns
The bias coarse size of the curve.

### ◆ get_curve_bias_fine_size()

double get_curve_bias_fine_size ( int *curve_id* )

Get the bias fine size of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns
The bias fine size of the curve.

### ◆ get_curve_bias_first_interval_fraction()

double get_curve_bias_first_interval_fraction ( int *curve_id* )

Get the bias first interval fraction of a curve

Parameters

**curve_id** Specifies the id of the curve

Returns

The bias first interval fraction of the curve.

◆ get_curve_bias_first_interval_length()

double get_curve_bias_first_interval_length(int *curve_id*)

Get the bias first interval length of a curve

Parameters

**curve_id** Specifies the id of the curve

Returns

The bias first interval length of the curve.

◆ get_curve_bias_first_last_ratio1()

double get_curve_bias_first_last_ratio1(int *curve_id*)

Get the bias first/last ratio at start of a curve

Parameters

**curve_id** Specifies the id of the curve

Returns

The bias coarse size of the curve.

◆ get_curve_bias_first_last_ratio2()

double get_curve_bias_first_last_ratio2(int *curve_id*)

Get the bias first/last ratio at end of a curve

Parameters

**curve_id** Specifies the id of the curve

Returns

The bias coarse size of the curve.

◆ get_curve_bias_from_start()

bool get_curve_bias_from_start(int *curve_id*,
bool & *value*
)

Get whether the bias is from the start of a curve

Parameters

**curve_id** Specifies the id of the curve

**value** Returns whether the bias is from the start of the curve.

Returns

True/False A curve with the curve_id exists.

◆ get_curve_bias_from_start_set()

bool get_curve_bias_from_start_set(int *curve_id*)

Get whether the bias from the start of a curve settings has been set
Parameters

**curve_id** Specifies the id of the curve
**value** Returns whether the bias from the start of the curve settings has been set.

Returns

True/False A curve with the curve_id exists.

### ◆ get_curve_bias_geometric_factor()

double get_curve_bias_geometric_factor(int *curve_id*)

Get the first bias geometric factor of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias geometric factor of the curve.

### ◆ get_curve_bias_geometric_factor2()

double get_curve_bias_geometric_factor2(int *curve_id*)

Get the second bias geometric factor of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias geometric factor of the curve.

### ◆ get_curve_bias_last_first_ratio1()

double get_curve_bias_last_first_ratio1(int *curve_id*)

Get the bias last/first ratio at start of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias coarse size of the curve.

### ◆ get_curve_bias_last_first_ratio2()

double get_curve_bias_last_first_ratio2(int *curve_id*)

Get the bias last/first ratio at end of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias coarse size of the curve.

### ◆ get_curve_bias_start_vertex_id()

int get_curve_bias_start_vertex_id(int *curve_id*)

Get the bias start vertex id of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias start vertex id of a curve.

### ◆ get_curve_bias_type()

std::string get_curve_bias_type(int *curve_id*)

Get the bias type of a curve
Parameters

**curve_id** Specifies the id of the curve

Returns

The bias type of the curve.

### ◆ get_curve_center()

std::array< double, 3 > get_curve_center(int *curve_id*)

Get the center point of the arc.
Parameters

**curve_id** ID of the curve

Returns

x, y, z center point of the curve in a vector (python tuple)

### ◆ get_curve_count()

int get_curve_count( )

Get the current number of curves.
Returns

The number of curves in the current model, if any

### ◆ get_curve_count_in_volumes()

int get_curve_count_in_volumes(std::vector< int > *target_volume_ids*)

Get the current number of curves in the passed-in volumes.
Returns

The number of curves in the volumes

### ◆ get_curve_edges()

std::vector< int > get_curve_edges(int *curve_id*)

get the list of any edge elements on a given curve
Parameters

**curve_id** User specified id of the desired curve

Returns

A list (python tuple) of the edge element ids on the curve

### ◆ get_curve_length()

double get_curve_length(int *curve_id*)

Get the length of a specified curve.

Parameters

**curve_id** ID of the curve

Returns

Length of the curve

### ◆ get_curve_mesh_scheme_curvature()

double get_curve_mesh_scheme_curvature(int *curve_id*)

Get the curvature mesh scheme value of a curve.

Parameters

**curve_id** Specifies the id of the curve

Returns

The curvature mesh scheme value of a curve.

### ◆ get_curve_mesh_scheme_pinpoint_locations()

std::vector< double > get_curve_mesh_scheme_pinpoint_locations(int *curve_id*)

Get the pinpoint mesh scheme locations of a curve

Parameters

**curve_id** Specifies the id of the curve

Returns

The pinpoint mesh scheme locations for a curve.

### ◆ get_curve_mesh_scheme_stretch_values()

```
bool get_curve_mesh_scheme_stretch_values(int    curve_id,
                                          double & first_size,
                                          double & factor,
                                          double & last_size,
                                          bool &   start,
                                          int &    vertex_id
                                         )
```

Get the stretch mesh scheme values of a curve

Parameters

**curve_id** Specifies the id of the curve

**first_size** Returns the first_size

**factor** Returns the factor

**last_size** Returns the last_size
**start** Returns whether the scheme is from the start of the curve.
**vertex_id** Returns the vertex id used for the start of the scheme.

Returns

True/False A curve with the curve_id exists.

### ◆ get_curve_nodes()

std::vector< int > get_curve_nodes(int *curve_id*)

Get list of node ids owned by a curve. Excludes nodes owned by bounding vertices.

```
int curv_id = 12;
vector<int> curve_nodes = CubitInterface::get_curve_nodes (curv_id);
```

CubitInterface::get_curve_nodes
std::vector< int > get_curve_nodes(int curve_id)
Get list of node ids owned by a curve. Excludes nodes owned by bounding vertices.

Parameters

**curv_id** id of curve

Returns

List (python tuple) of IDs of nodes owned by the curve

### ◆ get_curve_radius()

double get_curve_radius(int *curve_id*)

Get the radius of a specified arc.

Parameters

**curve_id** ID of the curve

Returns

Radius of the curve. If the curve is not an arc, the radius returned is the radius of curvature at the curve's midpoint. If the curve is linear, zero is returned.

### ◆ get_curve_type()

std::string get_curve_type(int *curve_id*)

Get the curve type for a specified curve.

Parameters

**curve_id** ID of the curve

Returns

Type of curve

### ◆ get_dbl_sculpt_default()

double get_dbl_sculpt_default(const char * *variable*)

return sculpt default value

### ◆ get_default_auto_size()

double get_default_auto_size ( )

Get auto size needs for the current set of geometry.

◆ get_default_element_type()

std::string get_default_element_type ( )

Get the current default setting for the element type that will be used when meshing.
Returns

A string indicating the default mesh type:

- "tri" indicates a tri/tet mesh default
- "hex" indicates a quad/hex mesh default
- "none" indicates no default has been assigned

◆ get_default_geometry_engine()

std::string get_default_geometry_engine ( )

Get the name of the default modeler engine.

```
std::string engine;
engine = CubitInterface::get_default_geometry_engine ();
```

CubitInterface::get_default_geometry_engine
std::string get_default_geometry_engine()
Get the name of the default modeler engine.

```
engine = cubit.get_default_geometry_engine()
```

Returns

The name of the default modeler engine in the form ACIS, CATIA, OCC, facet

◆ get_displacement_combine_type()

std::string get_displacement_combine_type ( int *entity_id* )

Get the displacement's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine".
Parameters

**entity_id** Id of the displacement

Returns

The combine type for the given displacement

◆ get_displacement_dof_signs()

const int * get_displacement_dof_signs ( int *entity_id* )

**This function only available from C++** Get the displacement's dof signs
Parameters

**entity_id** Id of the displacement

Returns

◆ get_displacement_dof_values()

const double * get_displacement_dof_values(int *entity_id*)

**This function only available from C++** Get the displacement's dof values
Parameters

**entity_id** Id of the displacement
Returns

◆ get_distance_between()

double get_distance_between(int *vertex_id_1*,
int *vertex_id_2*
)

Get the distance between two vertices.
Parameters

**vertex_id_1** ID of vertex 1 vertex_id_2 ID of vertex 2
Returns
distance

◆ get_distance_between_entities()

double get_distance_between_entities(std::string *geom_type_1*,
int *entity_id_1*,
std::string *geom_type_2*,
int *entity_id_2*
)

Get the distance between two geom entities.
Parameters

**geom_type_1** geometry type of entity 1: "vertex", "curve", "surface", "volume" entity_id_1 ID of entity 1 geom_type_2 geometry type of entity 2: "vertex", "curve", "surface", "volume" entity_id_2 ID of entity 2
Returns
distance

◆ get_distance_from_curve_start()

double get_distance_from_curve_start(double *x_coordinate*,
double *y_coordinate*,
double *z_coordinate*,
int *curve_id*
)

Get the distance from a point on a curve to the curve's start point.
Parameters

**x** value of the point to measure
**y** value of the point to measure
**z** value of the point to measure

**curve_id** ID of the curve

Returns

Distance from the xyz to the curve start

◆ get_edge_count()

int get_edge_count ( )

Get the count of edges in the model.

Returns

The number of edges in the model

◆ get_edge_global_element_id()

int get_edge_global_element_id ( int *edge_id* )

Given a edge id, return the global element id.

```
int gid = CubitInterface::get_edge_global_element_id (22);
```

CubitInterface::get_edge_global_element_id
int get_edge_global_element_id(int edge_id)
Given a edge id, return the global element id.

Parameters

**edge_id** Specifies the id of the edge

Returns

The corresponding element id

◆ get_edges_to_swap()

std::vector< int > get_edges_to_swap ( int *curve_id* )

Given a curve defining a knife edge between two triangle-meshed surfaces, return a list of edges on triangles at the curve that are good candidates for swapping. A good candidate for swapping means that if swapped, the two triangles at the knife's edge will have a larger interior dihedral angle between them, allowing a larger volume to accommodate tetmeshing.

Parameters

**curve_id** User-specified id of the curve

Returns

A list (python tuple) of edge ids contained

◆ get_elem_quality_stats()

```
std::vector< double >
get_elem_quality_stats (const std::string &       entity_type,
                        const std::vector< int > id_list,
                        const std::string &       metric_name,
                        const double              single_threshold,
                        const bool                use_low_threshold,
                        const double              low_threshold,
```

const double *high_threshold*,
const bool *make_group*
)

python callable version of the get_quality_stats without pass by reference arguments. All return values are stuffed into a double array

```
        std::vector<int> id_list = {223, 226, 256};
        double single_threshold = 0.2;
        bool use_low_threshold = false ;
        double low_threshold = 0.0;
        double high_threshold = 0.0;
        bool make_group = true ;
        std::vector<double>
        quality_data  =  CubitInterface::get_elem_quality_stats  ("hex"  ,
id_list, "scaled jacobian" ,
        single_threshold, use_low_threshold,
        low_threshold, high_threshold,
        make_group);
        double min_value = quality_data[0];
        double max_value = quality_data[1];
        double mean_value = quality_data[2];
        double std_value = quality_data[3];
        int min_element_id = (int)quality_data[4];
        int max_element_id = (int)quality_data[5];
        int element_type = (int)quality_data[6];
        int bad_group_id = (int)quality_data[7];
        int num_elems = (int)quality_data[8];
        std::vector<int> elem_ids(num_elems);
        for (int i=9, j=0; i<quality_data.size(); i++, j++)
        elem_ids[j] = (int )quality_data[i];
```

CubitInterface::get_elem_quality_stats

std::vector< double > get_elem_quality_stats(const std::string &entity_type, const std::vector< int > id_list, const std::string &metric_name, const double single_threshold, const bool use_low_threshold, const double low_threshold, const double high_threshold, const bool make_group)

python callable version of the get_quality_stats without pass by reference arguments....

Parameters

| | |
|---|---|
| **entity_type** | Specifies the geometry type of the entity |
| **id_list** | Specifies a list of ids to work on |
| **metric_name** | Specify the metric used to determine the quality |
| **single_threshold** | Quality threshold value |
| **use_low_threshold** | use threshold as lower or upper bound |
| **low_threshold** | Quality threshold when using a lower and upper range |
| **high_threshold** | Quality threshold when using a lower and upper range |

Returns

[0] min_value [1] max_value [2] mean_value [3] std_value [4] min_element_id [5] max_element_id [6] element_type 0 = edge, 1 = tri, 2 = quad, 3 = tet, 4 = hex [7] bad_group_id [8] size of mesh_list [9]...[n-1] mesh_list

### ◆ get_element_block()

int get_element_block(int *element_id*)

return the block that a given element is in.
Parameters

**element_id** The element id (i.e. the global element export id)

Returns

block_id, the id of the containing block

### ◆ get_element_budget()

int get_element_budget(const std::string & *element_type*,
std::vector< int > *entity_id_list*,
int *auto_factor*
)

Get the element budget based on current size settings for a list of volumes.
Parameters

**element_type** "hex" or "tet"
**entity_id_list** List (vector) of volume ids
**auto_factor** The current auto size factor value

Returns

The approximate number of elements that will be generated

### ◆ get_element_count()

int get_element_count( )

Get the count of elements in the model.
Returns

The number of quad, hex, tet, tri, wedge, edge, spheres, etc. which have been assigned to a block, given a global element id, and will be exported.

### ◆ get_element_exists()

bool get_element_exists(int *element_id*)

Check the existance of an element.
Parameters

**element_id** The element id (i.e. the global element export id)

Returns

true or false

### ◆ get_element_type()

std::string get_element_type(int *element_id*)

return the type of a given element
Parameters

**element_id** The element id (i.e. the global element export id)

Returns

The type

◆ get_element_type_id()

int get_element_type_id ( int *element_id* )

return the type id of a given element

Parameters

**element_id** The element id (i.e. the global element export id)

Returns

type_id The hex, tet, wedge, etc. id is returned.

◆ get_entities()

std::vector< int > get_entities ( const std::string & *entity_type* )

Get all entities of a specified type (including geometry, mesh, etc...)

```
std::vector<int> entity_id_list;
entity_id_list = CubitInterface::get_entities ("volume" );
```

CubitInterface::get_entities

std::vector< int > get_entities(const std::string &entity_type)

Get all entities of a specified type (including geometry, mesh, etc...)

```
entity_id_list = cubit.get_entities("volume" )
```

Parameters

**entity_type** Specifies the type of the entity

Returns

A list (python tuple) of ids of the specified geometry type

◆ get_entity_color()

std::array< double, 4 > get_entity_color ( const std::string & *entity_type*,
int *entity_id*
)

Get the color of a specified entity.

```
std::array<int,4> color_rgb = CubitInterface::get_entity_color
("curve" , 33);
```

CubitInterface::get_entity_color

std::array< double, 4 > get_entity_color(const std::string &entity_type, int entity_id)

Get the color of a specified entity.

```
color = cubit.get_entity_color("curve" , 33)
```

Parameters

**entity_type** Specifies the type of the entity

**entity_id** Specifies the id of the entity

Returns

The color of the entity

◆ get_entity_color_index()

```
int get_entity_color_index(const std::string & entity_type,
                           int                 entity_id
                          )
```

◆ get_entity_modeler_engine()

```
std::vector< std::string >
get_entity_modeler_engine (const std::string & geometry_type,
                           int                 entity_id
                          )
```

Get the modeler engine type for a specified entity.

```
        std::vector<std::string> engine_list;
        engine_list = CubitInterface::get_entity_modeler_engine ("surface" ,
47);
```

CubitInterface::get_entity_modeler_engine
std::vector< std::string > get_entity_modeler_engine(const std::string &geometry_type, int entity_id)
Get the modeler engine type for a specified entity.

```
        engine_list = cubit.get_entity_modeler_engine("surface" , 47)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

A vector (python tuple) of modeler engines associated with this entity

◆ get_entity_name()

```
std::string get_entity_name(const std::string & entity_type,
                            int                 entity_id,
                            bool                no_default = false
                           )
```

Get the name of a specified entity.
Names returned are of two types: 1) user defined names which are actually stored in Cubit when the name is defined, and 2) 'default' names supplied by Cubit at run-time which are not stored in Cubit. The second variety of name cannot be used to query Cubit.

```
        std::string name = CubitInterface::get_entity_name ("vertex" , 22);
```

CubitInterface::get_entity_name
std::string get_entity_name(const std::string &entity_type, int entity_id, bool no_default=false)
Get the name of a specified entity.

```
        name = cubit.get_entity_name("vertex" , 22)
```

Parameters

**entity_type** Specifies the type of the entity
**entity_id** Specifies the id of the entity
**no_default** True to return an empty string if no name is set

Returns

The name of the entity

◆ get_entity_names()

```
std::vector< std::string >
get_entity_names (const std::string & entity_type,
                  int entity_id,
                  bool no_default = false,
                  bool first_name_only = false
                 )
```

same as get_entity_name but includes all name attributes set on the entity, not just the first one (unless first_name_only is set)

◆ get_entity_sense()

```
std::string get_entity_sense(std::string source_type,
                             int source_id,
                             int sideset_id
                            )
```

Get the sense of a sideset item.

```
std::string sense;
sense = CubitInterface::get_entity_sense ("face" , 332, 2);
```

CubitInterface::get_entity_sense

std::string get_entity_sense(std::string source_type, int source_id, int sideset_id)

Get the sense of a sideset item.

```
sense = cubit.get_entity_sense("face" , 332, 2)
```

Parameters

**source_type** Item type - could be 'face', 'quad' or 'tri'
**source_id** ID of entity
**sideset_id** ID of the sideset

Returns

Sense of the source_type/source_id in specified sideset

◆ get_error_count()

```
int get_error_count( )
```

Get the number of errors in the current Cubit session.

Returns

The number of errors in the Cubit session.

◆ get_exodus_element_count()

```
int get_exodus_element_count(int entity_id,
```

std::string *entity_type*
)

Get the number of elements in a exodus entity.

```
int element_count = CubitInterface::get_exodus_element_count (2,
"sideset" );
```

CubitInterface::get_exodus_element_count
int get_exodus_element_count(int entity_id, std::string entity_type)
Get the number of elements in a exodus entity.

```
element_count = cubit.get_exodus_element_count(2, "sideset" )
```

Parameters

**entity_id** The id of the entity
**entity_type** The type of the entity

Returns

Number of Elements

◆ get_exodus_entity_description()

std::string get_exodus_entity_description (std::string *entity_type*,
int *entity_id*
)

Get the description associated with an exodus entity.

```
std::string entity_description;
entity_description = CubitInterface::get_exodus_entity_description
("sideset" , 33);
```

CubitInterface::get_exodus_entity_description
std::string get_exodus_entity_description(std::string entity_type, int entity_id)
Get the description associated with an exodus entity.

```
entity_description = cubit.get_exodus_entity_description("sideset"
, 33)
```

Parameters

**entity_type** "block", "sideset", nodeset" @param entity_id Id of the entity in question \return Description of the entity or "" if none

◆ get_exodus_entity_name()

std::string get_exodus_entity_name (const std::string *entity_type*,
int *entity_id*
)

Get the name associated with an exodus entity.

```
std::string entity_name;
entity_name = CubitInterface::get_exodus_entity_name ("sideset" ,
33);
```

CubitInterface::get_exodus_entity_name
std::string get_exodus_entity_name(const std::string entity_type, int entity_id)
Get the name associated with an exodus entity.

```
        entity_name = cubit.get_exodus_entity_name("sideset" , 33)
```

Parameters

**entity_type** "block", "sideset", nodeset" @param entity_id Id of the entity in question \return Name of the entity or "" if none

◆ get_exodus_entity_type()

std::string get_exodus_entity_type ( std::string *entity_type*,
int *entity_id*
)

Get the type of an exodus entity.

```
        std::string entity_description;
        entity_description = CubitInterface::get_exodus_entity_description
("sideset" , 33);
```

```
        entity_description = cubit.get_exodus_entity_type("sideset" , 33)
```

Parameters

**entity_type** "block", "sideset", nodeset" @param entity_id Id of the entity in question \return Type of the entity or "" if none. Returns "lite" or ""

◆ get_exodus_id()

int get_exodus_id ( const std::string & *entity_type*,
int *entity_id*
)

Get the exodus/genesis id for this element.

```
        int exodus_id = CubitInterface::get_exodus_id ("hex" , 221);
```

CubitInterface::get_exodus_id
int get_exodus_id(const std::string &entity_type, int entity_id)
Get the exodus/genesis id for this element.

```
        exodus_id = cubit.get_exodus_id("hex" , 221)
```

Parameters

**entity_type** The mesh element type
**entity_id** The mesh element id

Returns

Exodus id of the element if element has been written out, otherwise 0

◆ get_exodus_sizing_function_file_name()

std::string get_exodus_sizing_function_file_name ( )

Get the exodus sizing function file name.

Returns

The sizing function file name

◆ get_exodus_sizing_function_variable_name()

std::string get_exodus_sizing_function_variable_name ( )

Get the exodus sizing function variable name.
Returns
The sizing function variable name

◆ get_exodus_variable_count()

int get_exodus_variable_count(std::string *container_type*,
int *container_id*
)

Get the number of exodus variables in a nodeset, sideset, or block.
Parameters

**entity_type**: nodeset, sideset, or block block_id The block id
Returns
Number of exodus variables

◆ get_exodus_variable_names()

std::vector< std::string > get_exodus_variable_names(std::string *container_type*,
int *container_id*
)

Get the names of exodus variables in a nodeset, sideset, or block.
Parameters

**entity_type**: nodeset, sideset, or block block_id The block id
Returns
Names of exodus variables

◆ get_exodus_version()

std::string get_exodus_version( )

Get the Exodus version number.
Returns
A string containing the Exodus version number

◆ get_expanded_connectivity()

std::vector< int > get_expanded_connectivity(const std::string & *entity_type*,
int *entity_id*
)

Get the list of node ids contained within a mesh entity, including interior nodes.

```
       std::vector<int> node_id_list;
       node_id_list  =  CubitInterface::get__expanded_connectivity("hex"  ,
221);
       node_id_list = cubit.get__expanded_connectivity("hex" , 221)
```

Parameters

**entity_type**The mesh element type

**entity_id** The mesh element id

Returns

List (python tuple) of all node ids associated with the element, including interior nodes

### ◆ get_force_direction_vector()

std::array< double, 3 > get_force_direction_vector(int *entity_id*)

Get the direction vector from a force.

Parameters

**entity_id** Id of the force

Returns

A vector (python tuple) [x,y,z] of the direction the given force is acting

### ◆ get_force_magnitude()

double get_force_magnitude(int *entity_id*)

Get the force magnitude from a force.

Parameters

**entity_id** Id of the force

Returns

Magnitude of the given force

### ◆ get_force_moment_vector()

std::array< double, 3 > get_force_moment_vector(int *entity_id*)

Get the moment vector from a force.

Parameters

**entity_id** Id of the force

Returns

A vector (python tuple) [x,y,z] of the direction of the moment for the given force

### ◆ get_gaps_between_volumes()

```
std::vector< VolumeGap >
get_gaps_between_volumes (std::vector< int > target_volume_ids,
                          double             maximum_gap_tolerance,
                          double             maximum_gap_angle,
                          int                cache_overlaps = 0
                         )
```

### ◆ get_geometric_owner()

```
std::vector< std::string > get_geometric_owner(std::string mesh_entity_type,
                                               std::string mesh_entity_list
                                              )
```

Get a list of geometric owners given a list of mesh entities.

```
std::vector<std::string> owner_list;
owner_list = CubitInterface::get_geometric_owner ("quad" , id_list);
```

CubitInterface::get_geometric_owner

std::vector< std::string > get_geometric_owner(std::string mesh_entity_type, std::string mesh_entity_list)

Get a list of geometric owners given a list of mesh entities.

```
owner_list = cubit.get_geometric_owner("quad" , id_list)
```

Parameters

| | |
|---|---|
| **mesh_entity_type** | The type of mesh entity. Works for 'quad, 'face', 'tri', 'hex', 'tet', 'edge', 'node' |
| **mesh_entity_list** | A string containing space delimited ids, Cubit command form (i.e. 'all', '1 to 8', '1 2 3', etc) |

Returns

A list (python tuple) of geometry owners in the form of 'surface x', 'curve y', etc.

### ◆ get_geometry_node_count()

int get_geometry_node_count ( const std::string & *entity_type*,
int *entity_id*
)

Get the node count for a specific geometric entity.

Parameters

| | |
|---|---|
| **entity_type** | The geometry type ("surface", "curve", etc) |
| **entity_id** | The entity id |

Returns

Number of nodes in the geometry

### ◆ get_geometry_owner()

std::string get_geometry_owner ( const std::string & *entity_type*,
int *entity_id*
)

Get the geometric owner of this mesh element.

```
std::string geom_owner = CubitInterface::get_geometry_owner ("hex" , 221);
```

CubitInterface::get_geometry_owner

std::string get_geometry_owner(const std::string &entity_type, int entity_id)

Get the geometric owner of this mesh element.

```
geom_owner = cubit.get_geometry_owner("hex" , 221)
```

Parameters

| | |
|---|---|
| **entity_type** | The mesh element type |
| **entity_id** | The mesh element id |

Returns

Name of owner

### ◆ get_global_element_id()

int get_global_element_id ( const std::string & *element_type*,
int *id*
)

Given a hex, tet, etc. id, return the global element id.

```
int gid = CubitInterface::get_global_element_id ("hex" , 22);
```

CubitInterface::get_global_element_id
int get_global_element_id(const std::string &element_type, int id)
Given a hex, tet, etc. id, return the global element id.

Parameters

**id** Specifies the id of the hex, tet, etc. elem_type the type of the entity ("hex", "tet", "wedge", "pyramid", "tri", "face", "quad", "edge", or "sphere")

Returns

The corresponding element id

### ◆ get_graphics_version()

std::string get_graphics_version ( )

Get the VTK version number.

Returns

A string containing the VTK version number

### ◆ get_group_bodies()

std::vector< int > get_group_bodies ( int *group_id* )

Get group bodies (bodies that are children of a group)
This routine returns a list of bodies that are contained in a specified group.

Parameters

**group_id** ID of the group to examine return List (python tuple) of bodies ids contained in the specified group

### ◆ get_group_children()

void get_group_children ( int *group_id*,
std::vector< int > & *returned_group_list*,
std::vector< int > & *returned_body_list*,
std::vector< int > & *returned_volume_list*,
std::vector< int > & *returned_surface_list*,
std::vector< int > & *returned_curve_list*,
std::vector< int > & *returned_vertex_list*,
int & *returned_node_count*,
int & *returned_edge_count*,
int & *returned_hex_count*,

int & *returned_quad_count*,
int & *returned_tet_count*,
int & *returned_tri_count*,
int & *returned_wedge_count*,
int & *returned_pyramid_count*,
int & *returned_sphere_count*
)

Get group children.
This routine returns a list for each geometry entity type in the group. Since groups may contain both geometry and mesh entities, this routine also returns the count of any mesh entity contained in the group. For groups contained in the group, the group_list will only contain one generation. In other words, if this routine is examining Group ABC, and Group ABC contains Group XYZ and Group XYZ happens to contain other groups (which in turn may contain other groups) this routine will only return the id of Group XYZ.
Parameters

| | |
|---|---|
| **group_id** | ID of the group to examine |
| **group_list** | User specified list where group ids will be returned |
| **body_list** | User specified list where body ids will be returned |
| **volume_list** | User specified list where volume ids will be returned |
| **surface_list** | User specified list where surface ids will be returned |
| **curve_list** | User specified list where curve ids will be returned |
| **vertex_list** | User specified list where vertex ids will be returned |
| **node_count** | User specified variable where the number of nodes will be returned |
| **edge_count** | User specified variable where the number of edges will be returned |
| **hex_count** | User specified variable where the number of hexes will be returned |
| **quad_count** | User specified variable where the number of quads will be returned |
| **tet_count** | User specified variable where the number of tets will be returned |
| **tri_count** | User specified variable where the number of tris will be returned |

## ◆ get_group_curves()

std::vector< int > get_group_curves ( int *group_id* )

Get group curves (curves that are children of a group)
This routine returns a list of curves that are contained in a specified group.
Parameters

| | |
|---|---|
| **group_id** | ID of the group to examine return List (python tuple) of curve ids contained in the specified group |

## ◆ get_group_edges()

std::vector< int > get_group_edges ( int *group_id* )

Get group edges (edges that are children of a group)
This routine returns a list of edges that are contained in a specified group.

Parameters

**group_id** ID of the group to examine return List (python tuple) of edge ids contained in the specified group

◆ get_group_groups()

std::vector< int > get_group_groups(int *group_id*)

Get group groups (groups that are children of another group)
This routine returns a list a groups that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of group ids contained in the specified group

◆ get_group_hexes()

std::vector< int > get_group_hexes(int *group_id*)

Get group hexes (hexes that are children of a group)
This routine returns a list of hexes that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of hex ids contained in the specified group

◆ get_group_nodes()

std::vector< int > get_group_nodes(int *group_id*)

Get group nodes (nodes that are children of a group)
This routine returns a list of nodes that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of node ids contained in the specified group

◆ get_group_pyramids()

std::vector< int > get_group_pyramids(int *group_id*)

Get group pyramids (pyramids that are children of a group)
This routine returns a list of pyramids that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of pyramid ids contained in the specified group

◆ get_group_quads()

std::vector< int > get_group_quads(int *group_id*)

Get group quads (quads that are children of a group)
This routine returns a list of quads that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of quad ids contained in the specified group

- get_group_spheres()

std::vector< int > get_group_spheres ( int *group_id* )

Get group spheres (sphere elements that are children of a group)
This routine returns a list of spheres that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of sphere ids contained in the specified group

- get_group_surfaces()

std::vector< int > get_group_surfaces ( int *group_id* )

Get group surfaces (surfaces that are children of a group)
This routine returns a list of surfaces that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of surface ids contained in the specified group

- get_group_tets()

std::vector< int > get_group_tets ( int *group_id* )

Get group tets (tets that are children of a group)
This routine returns a list of tets that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of tet ids contained in the specified group

- get_group_tris()

std::vector< int > get_group_tris ( int *group_id* )

Get group tris (tris that are children of a group)
This routine returns a list of tris that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of tri ids contained in the specified group

- get_group_vertices()

std::vector< int > get_group_vertices ( int *group_id* )

Get group vertices (vertices that are children of a group)
This routine returns a list of vertices that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of vertex ids contained in the specified group

### ◆ get_group_volumes()

std::vector< int > get_group_volumes ( int *group_id* )

Get group volumes (volumes that are children of a group)
This routine returns a list of volumes that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of volume ids contained in the specified group

### ◆ get_group_wedges()

std::vector< int > get_group_wedges ( int *group_id* )

Get group wedges (wedges that are children of a group)
This routine returns a list of wedges that are contained in a specified group.
Parameters

**group_id** ID of the group to examine return List (python tuple) of wedge ids contained in the specified group

### ◆ get_heatflux_on_area()

double get_heatflux_on_area ( CI_BCEntityTypes *bc_area_enum*,
int *entity_id*
)

Get the heatflux on a specified area.
Parameters

**bc_area** enum of CI_BCEntityTypes. If on solid, use 4. If on thin shell, use 7 for top, 8 for bottom
**entity_id** ID of the heatflux
Returns
The value or magnitude of the specified heatflux

### ◆ get_hex_count()

int get_hex_count ( )

Get the count of hexes in the model.
Returns
The number of hexes in the model

### ◆ get_hex_global_element_id()

int get_hex_global_element_id ( int *hex_id* )

Given a hex id, return the global element id.

```
int gid = CubitInterface::get_hex_global_element_id (22);
```

CubitInterface::get_hex_global_element_id
int get_hex_global_element_id(int hex_id)
Given a hex id, return the global element id.

Parameters

**hex_id** Specifies the id of the hex

Returns

The corresponding element id

◆ get_hex_sheet()

std::vector< int > get_hex_sheet(int *node_id_1*,
int *node_id_2*
)

Get the list of hex elements forming a hex sheet through the given two node ids. The nodes must be adjacent in the connectivity of the hex i.e. they form an edge of the hex.

Returns

A list (python tuple) of hex ids in the hex sheet

◆ get_hole_surfaces()

std::vector< int > get_hole_surfaces(int *surface_id*)

Returns the adjacent surfaces in a hole for a surface.

Parameters

**surface_id** that is part of the hole

Returns

A list of surface id's in the hole (including surface_id).

◆ get_hydraulic_radius_surface_area()

double get_hydraulic_radius_surface_area(int *surface_id*)

Get the area of a hydraulic surface.

Parameters

**surface_id** ID of the surface

Returns

Hydraulic area of the surface

◆ get_hydraulic_radius_volume_area()

double get_hydraulic_radius_volume_area(int *volume_id*)

Get the area of a hydraulic volume.

Parameters

**volume_id** ID of the volume

Returns

Hydraulic area of the volume

### ◆ get_id_from_name()

int get_id_from_name(const std::string & *name*)

Get id for a named entity.
This routine returns an integer id for the entity whose name is passed in.

```
int entity_id = CubitInterface::get_id_from_name ("member_2" );
```

CubitInterface::get_id_from_name
int get_id_from_name(const std::string &name)
Get id for a named entity.

```
entity_id = cubit.get_id_from_name("member_2" )
```

Parameters

**name** Name of the entity to examine return Integer representing the entity

### ◆ get_id_string()

std::string get_id_string(const std::vector< int > & *entity_ids*)

Parse a list of integers into a Cubit style id list. Return string will not include carriage returns or line break.

```
std::vector<int> entity_ids = {1, 2, 3, 4};
std::string id_string = CubitInterface::get_id_string (entity_ids);
// id_string is "1 to 4";
```

CubitInterface::get_id_string
std::string get_id_string(const std::vector< int > &entity_ids)
Parse a list of integers into a Cubit style id list. Return string will not include carriage returns ...

```
entity_ids = [1,2,3,4]
id_string = cubit.get_all_ids_from_name(entity_ids)
# id_string is '1 to 4'
```

Parameters

**entity_ids** vector of integers return A string representing the id list without line breaks

### ◆ get_idless_signature()

std::string get_idless_signature(std::string *entity_type*,
int *entity_id*
)

get the idless signature of a geometric or mesh entity
Parameters

**type** the type of the requested entity
**id** the id of the requested entity

Returns
the idless signature i.e. curve at (1 1 0 ordinal 2)

### ◆ get_idless_signatures()

std::string get_idless_signatures(std::string *entity_type*,
const std::vector< int > & *entity_id_list*
)

get the idless signatures of a range of geometric or mesh entities
Parameters

**type** the type of the requested entity
**idlist** a list of ids

Returns

the idless signature i.e. curve at (1 1 0 ordinal 2) curve at (0 0 1 ordinal 1) ...

### ◆ get_int_sculpt_default()

int get_int_sculpt_default(const char * *variable*)

### ◆ get_interface()

CubitBaseInterface * get_interface(std::string *interface_name*)

Get the interface of a given name.
Parameters

**interface_name** the name of interface

### ◆ get_label_type()

int get_label_type(const char * *entity_type*)

make calls to SVDrawTool::get_label_type
Returns

label type currently associated with entity_type

### ◆ get_last_id()

int get_last_id(const std::string & *entity_type*)

Get the id of the last created entity of the given type.

```
int last_id = CubitInterface::get_last_id ("surface" );
```

CubitInterface::get_last_id
int get_last_id(const std::string &entity_type)
Get the id of the last created entity of the given type.

```
last_id = cubit.get_last_id("surface" )
```

Parameters

**entity_type** Type of the entity being queried

Returns

Integer id of last created entity

### ◆ get_list_of_free_ref_entities()

std::vector< int > get_list_of_free_ref_entities(const std::string & *geometry_type*)

Get all free entities of a given geometry type.

```
      std::vector<int> free_curve_id_list;
      free_curve_id_list  =  CubitInterface::get_list_of_free_ref_entities
("curve" );
```

CubitInterface::get_list_of_free_ref_entities

std::vector< int > get_list_of_free_ref_entities(const std::string &geometry_type)

Get all free entities of a given geometry type.

```
      free_curve_id_list = cubit.get_list_of_free_ref_entities("curve" )
```

Parameters

**geom_type** Specifies the geometry type of the free entity

Returns

A list (python tuple) of ids of the specified geometry type

◆ get_material_name()

std::string get_material_name ( int *material_id* )

Get the name of the material (or cfd media) with the given id.

Returns

A std::string with the material's name.

◆ get_material_name_list()

std::vector< std::string > get_material_name_list ( )

Get a list of all defined material names.

Returns

List (python tuple) of all the material names.

◆ get_material_property()

double get_material_property ( CI_MaterialProperty *material_property_enum*,
int *entity_id*
)

Get the specified material property value.

Parameters

**mp** enum of CI_MaterialProperty. 0-Elastic Modulus, 1-Shear Modulus, 2-Poisson Ratio, 3-Density, 4-Specific Heat, 5-Conductivity

**entity_id** Id of the material

Returns

Value of the specified property for that material

◆ get_media_name_list()

std::vector< std::string > get_media_name_list ( )

Get a list of all defined material names.

Returns

List (python tuple) of all the material names.

◆ get_media_property()

int get_media_property(int *entity_id*)

Get the media property value.
Parameters

**entity_id** Id of the media

Returns

Value of the media property, 0 == FLUID, 1 == POROUS, 2 == SOLID

◆ get_merge_setting()

std::string get_merge_setting(const std::string & *geometry_type*,
int *entity_id*
)

Get the merge setting for a specified entity.

```
std::string merge_setting = CubitInterface::get_merge_setting
("surface" , 33);
```

CubitInterface::get_merge_setting
std::string get_merge_setting(const std::string &geometry_type, int entity_id)
Get the merge setting for a specified entity.

```
merge_setting = cubit.get_merge_setting("surface" , 33)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

A text string that indicates the merge setting for the entity

◆ get_merge_tolerance()

double get_merge_tolerance( )

Get the current merge tolerance value.
Returns

The value of the current merge tolerance

◆ get_mergeable_curves()

std::vector< std::vector< int > >
get_mergeable_curves (std::vector< int > *target_volume_ids*)

Get the list of mergeable curves from a list of volumes/bodies.
Given a list of volume ids, this will return a list of potentially mergeable curves. The returned lists include lists of the merge partners.
Parameters

**target_volume_ids** List of volume ids to examine.

Returns

list of lists of mergeable curves (potentially more than a pair) Note: If using python, lists will be python tuples.

### ◆ get_mergeable_entities()

```
void get_mergeable_entities(std::vector< int >                    target_volume_ids,
                            std::vector< std::vector< int > > &   returned_surface_list,
                            std::vector< std::vector< int > > &   returned_curve_list,
                            std::vector< std::vector< int > > &   returned_vertex_list,
                            double                                merge_tol = -1
                           )
```

**This function only works from C++** Get the list of mergeable entities from a list of volumes
Given a list of volume ids, this will return 3 lists of potential merge candidates. The returned lists include lists of the merge partners.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **surface_list** | User specified list where mergeable surfaces will be stored |
| **curve_list** | User specified list where mergeable curves will be stored |
| **vertex_list** | User specified list where mergeable vertices will be stored |
| **merge_tol** | merge tolerance to determine mergable (optional). Uses the default merge_tolerance if not specified |

### ◆ get_mergeable_surfaces()

```
std::vector< std::vector< int > >
get_mergeable_surfaces(std::vector< int > target_volume_ids)
```

Get the list of mergeable surfaces from a list of volumes/bodies.
Given a list of volume ids, this will return a list of potentially mergeable surfaces. The returned lists include lists of the merge partners.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |

Returns

list of lists of mergeable surfaces (potentially more than a pair) Note: If using python, lists will be python tuples.

### ◆ get_mergeable_vertices()

```
std::vector< std::vector< int > >
get_mergeable_vertices(std::vector< int > target_volume_ids)
```

Get the list of mergeable vertices from a list of volumes/bodies.
Given a list of volume ids, this will return a list of potentially mergeable vertices. The returned lists include lists of the merge partners.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |

Returns

list of lists of mergeable vertices (potentially more than a pair) Note: If using python, lists will be python tuples.

### ◆ get_mesh_edge_length()

double get_mesh_edge_length(int *edge_id*)

Get the length of a mesh edge.

Parameters

**edge_id** Specifies the id of the edge

Returns

The length of the mesh edge

### ◆ get_mesh_element_type()

std::string get_mesh_element_type(const std::string & *entity_type*,
int *entity_id*
)

Get the mesh element type contained in the specified geometry.

```
std::string element_type = CubitInterface::get_mesh_element_type
("surface" , 2);
```

CubitInterface::get_mesh_element_type

std::string get_mesh_element_type(const std::string &entity_type, int entity_id)

Get the mesh element type contained in the specified geometry.

```
element_type = cubit.get_mesh_element_type("surface" , 2)
```

Parameters

**entity_type** The type of entity

**entity_id** The id of the entity

Returns

Mesh element type for that entity

### ◆ get_mesh_error_count()

int get_mesh_error_count( )

### ◆ get_mesh_error_solutions()

std::vector< std::string > get_mesh_error_solutions(int *error_code*)

Get the paired list of mesh error solutions and help context cues.

Parameters

**error_code** The error code associated with the error solution

Returns

List (python tuple) of 'married' strings. First string is solution text. Second string is help context cue. Third string is command_panel cue.

### ◆ get_mesh_errors()

std::vector< **MeshErrorFeedback** * > get_mesh_errors( )

### ◆ get_mesh_geometry_approximation_angle()

double get_mesh_geometry_approximation_angle ( std::string *geometry_type*,
int *entity_id*
)

Get the geometry approximation angle set for tri/tet meshing.
Parameters

**geom_type** either "surface" or "volume"
**entity_id** the entity id

Returns

boolean value as to whether or not the proximity flag is set

### ◆ get_mesh_group_parent_ids()

std::vector< int > get_mesh_group_parent_ids ( const std::string & *element_type*,
int *element_id*
)

Get the group ids which are parents to the indicated mesh element.

```
std::vector<int> parent_id_list;
parent_id_list = CubitInterface::get_mesh_group_parent_ids ("tri" , 332);
```

CubitInterface::get_mesh_group_parent_ids
std::vector< int > get_mesh_group_parent_ids(const std::string &element_type, int element_id)
Get the group ids which are parents to the indicated mesh element.

```
parent_id_list = cubit.get_mesh_group_parent_ids("tri" , 332)
```

Parameters

**element_type** Mesh type of the element
**element_id** ID of the mesh element return List (python tuple) of group ids that contain this mesh element

### ◆ get_mesh_interval_firmness()

std::string get_mesh_interval_firmness ( const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh interval firmness for the specified entity. This may include influence from connected mesh intervals on connected geometry.

```
std::string firmness;
CubitInterface::get_mesh_interval_firmness ("surface" , 12);
```

CubitInterface::get_mesh_interval_firmness
std::string get_mesh_interval_firmness(const std::string &geometry_type, int entity_id)
Get the mesh interval firmness for the specified entity. This may include influence from connected me...

```
        firmness = cubit.get_mesh_interval_firmness("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's meshing firmness (HARD, SOFT, LIMP) HARD = set directly SOFT = computed LIMP = not set

### ◆ get_mesh_intervals()

```
int get_mesh_intervals ( const std::string &  geometry_type,
                         int                  entity_id
                       )
```

Get the interval count for a specified entity.

```
        int intervals = CubitInterface::get_mesh_intervals ("surface" , 12);
```

CubitInterface::get_mesh_intervals
int get_mesh_intervals(const std::string &geometry_type, int entity_id)
Get the interval count for a specified entity.

```
        intervals = cubit.get_mesh_intervals("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's interval count

### ◆ get_mesh_scheme()

```
std::string get_mesh_scheme ( const std::string &  geometry_type,
                              int                  entity_id
                            )
```

Get the mesh scheme for the specified entity.

```
        std::string scheme;
        CubitInterface::get_mesh_scheme ("surface" , 12, scheme);
```

CubitInterface::get_mesh_scheme
std::string get_mesh_scheme(const std::string &geometry_type, int entity_id)
Get the mesh scheme for the specified entity.

```
        scheme = cubit.get_mesh_scheme("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's meshing scheme

### ◆ get_mesh_scheme_firmness()

std::string get_mesh_scheme_firmness(const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh scheme firmness for the specified entity.

```
std::string firmness;
CubitInterface::get_mesh_firmness("surface" , 12);
```

```
firmness = cubit.get_mesh_firmness("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's meshing firmness (HARD, LIMP, SOFT)

### ◆ get_mesh_size()

double get_mesh_size(const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh size for a specified entity.

```
double mesh_size = CubitInterface::get_mesh_size ("volume" , 2);
```

CubitInterface::get_mesh_size
double get_mesh_size(const std::string &geometry_type, int entity_id)
Get the mesh size for a specified entity.

```
mesh_size = cubit.get_mesh_size("volume" , 2)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's mesh size

### ◆ get_mesh_size_type()

std::string get_mesh_size_type(const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh size setting type for the specified entity. This may include influence from attached geometry.

```
std::string firmness;
CubitInterface::get_mesh_size_setting_type("surface" , 12);
```

```
firmness = cubit.get_mesh_size_setting_type("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns
The entity's mesh size type (USER_SET, CALCULATED, NOT_SET)

### ◆ get_meshed_volume_or_area()

double get_meshed_volume_or_area(const std::string & *geometry_type*,
std::vector< int > *entity_ids*
)

Get the total volume/area of a entity's mesh.

```
double area = CubitInterface::get_meshed_volume_or_area ("volume" , 1);
```

CubitInterface::get_meshed_volume_or_area
double get_meshed_volume_or_area(const std::string &geometry_type, std::vector< int > entity_ids)
Get the total volume/area of a entity's mesh.

```
area = cubit.get_meshed_volume_or_area("volume" , 1)
```

Parameters

**geom_type** Specifies the type of entity - volume, surface, hex, tet, tri, quad
**entity_ids** A list of ids for the entity type

Returns
The entity's meshed volume or area

### ◆ get_meshgems_version()

std::string get_meshgems_version( )

Get the MeshGems version number.
Returns
A string containing the MeshGems version number

### ◆ get_ML_classification()

std::string get_ML_classification(std::string *geom_type*,
size_t *ent_id*
)

return the name of the classification category for this surface or volume. uses same methods as get_ML_predictions for volume_no_op or classify_surface. Same as calling get_ML_operation_features + get_ML_predictions with volume_no_op and classify_surface, but returns category with highest probablity.
Parameters

**geom_type** "volume" or "surface"
**ent_id** id of volume or surface to classify

Returns
string representing classification.

### ◆ get_ML_classification_categories()

std::vector< std::string > get_ML_classification_categories(std::string *geom_type*)

return a list of strings representing all possible calssification categories for volumes or surfaces
Parameters

**geom_type** "volume" or "surface"

◆ get_ML_classification_models()

std::vector< std::string > get_ML_classification_models ( )

get the available classification ML model names

◆ get_ML_classifications()

std::vector< std::string > get_ML_classifications ( std::string *geom_type*,
std::vector< size_t > *ent_ids*
)

same as get_ML_classification, but classifies multiple volumes or surfaces with a single call (more efficient)
Parameters

**geom_type** "volume" or "surface"
**ent_ids** vector of ids of volumes or surfaces to classify

Returns

vector of strings representing classification. Same order as vol_ids.

◆ get_ML_feature_distance()

double get_ML_feature_distance ( const std::string *op_name*,
std::vector< double > & *f1*,
std::vector< double > & *f2*
)

feature distance is defined as a weighted distance between two feature vectors of the same size. Features are weighted on EDT (ensembles of decision trees) importance values.
Parameters

**op_name** operation name (see get_ML_operation_features)
**f1** first feature vector
**f2** second feature vector

◆ get_ML_feature_importances()

std::vector< double > get_ML_feature_importances ( const std::string *op_name* )

return the vector of feature importances for a given operation type

◆ get_ML_model_ID()

int get_ML_model_ID ( std::string )

get a unique ID for the given operation/model name

Returns
0 if failure or postive integer otherwise

### ◆ get_ML_model_name()

```
std::string get_ML_model_name(int model_ID)
```

get the name for the given operation/model ID
Returns
empty string if failure

### ◆ get_ML_operation()

```
std::vector< std::string >
get_ML_operation (const std::string op_name,
                  const size_t entity_id1,
                  const size_t entity_id2,
                  const std::vector< double > params,
                  const double small_curve_size,
                  const double mesh_size
                 )
```

get the command, display and preview strings for a given ML operation type
Parameters

**op_name** operation name (see get_ML_operation_features)
**entity1_id** first entity associated with operation (see table)
**entity2_id** second entity associated with operation (see table)
**params** optional parameters for operation

### ◆ get_ML_operation_feature_names()

```
std::vector< std::string >
get_ML_operation_feature_names (const std::string ml_op_name,
                                bool reduced_features = false
                               )
```

for the given operation type described by get_ML_operation_features, return a vector of strings indicating the name of data for each feature in the vector.
Parameters

**ml_op_name** name of ML model
**reduced_features** optional currently supported only for volume_no_op. Uses 9 instead of 46 features for more efficient predictions

### ◆ get_ML_operation_feature_size()

```
int get_ML_operation_feature_size(const std::string ml_op_name,
                                  const bool reduced_features = false
                                 )
```

for the given operation type described by get_ML_operation_features, return the expected size of the feature vector
Parameters

| | |
|---|---|
| **ml_op_name** | name of ML model |
| **reduced_features** | optional currently supported only for volume_no_op. Uses 9 instead of 46 features for more efficient predictions |

### ◆ get_ML_operation_feature_types()

```
std::vector< std::string >
get_ML_operation_feature_types (const std::string ml_op_name,
                                bool reduced_features = false
                               )
```

for the given operation type described by get_ML_operation_features, return a vector of strings indicating the type of data for each feature in the vector. Will return one of the following for each index:

1. boolean 1 or 0
2. categorical usually positive integer representing a unique category assignment (ie. planar vs conic vs spline surface type)
3. continuous could be double or integer describing a continuous range (i.e. number of adjacent curves, area of a surface, etc..)

Parameters

| | |
|---|---|
| **ml_op_name** | name of ML model |
| **reduced_features** | optional currently supported only for volume_no_op. Uses 9 instead of 46 features for more efficient predictions |

### ◆ get_ML_operation_features()

```
std::vector< std::vector< double > >
get_ML_operation_features (std::vector< std::string > ml_op_names,
                           std::vector< size_t > entity1_ids,
                           std::vector< size_t > entity2_ids,
                           std::vector< std::vector< double > > params,
                           double mesh_size,
                           bool reduced_features = false
                          )
```

get machine learning features for a list of cubit operations

```
      std::vector<std_string>    ml_op_names    =    {"surface_no_op"    ,
"surface_no_op" };
```

```
        std::vector<int> entity1_ids = {20, 25}; // surface IDs
        std::vector<int> entity2_ids = {0, 0}; // none for surface_no_op
        std::vector<std::vector<double>> params = {{-1, -1, -1}, {-1, -1, -
1}}; // dummy for surface no_op
        double mesh_size = 1.5924; // target mesh size
        std::vector<std::vector<double>>
        features = CubitInterface::get_ML_operation_features (ml_op_names,
        entity1_ids, entity2_ids,
        params, mesh_size);
```

CubitInterface::get_ML_operation_features

std::vector< std::vector< double > > get_ML_operation_features(std::vector< std::string > ml_op_names, std::vector< size_t > entity1_ids, std::vector< size_t > entity2_ids, std::vector< std::vector< double > > params, double mesh_size, bool reduced_features=false)

get machine learning features for a list of cubit operations

```
        ml_op_names = ['surface_no_op' , 'surface_no_op' ]
        entity1_ids = [20, 25] # surface IDs
        entity2_ids = [0, 0] # none for surface_no_op
        params = [[-1, -1, -1], [-1, -1, -1]] # dummy forsurface no_op
        mesh_size = 1.5924 # target mesh size
        features = cubit.get_ML_operation_features(ml_op_names,
        entity1_ids, entity2_ids,
        params, mesh_size)
```

CubitInterface::surface

CubitInterface::Surface surface(int id_in)

Gets the surface object from an ID.

Parameters

**ml_op_name** ML operation/model name. One of the following IDs: see also get_ML_regression_models and get_ML_classification_models

1. ID of operation
2. type of model (R) regression (C) classification
3. number of labels

### 1. 2. 3. ml_op_name Entity1 Entity2 Params

1 R 3 vertex_no_op vertex none 2 R 3 curve_no_op curve none 3 R 3 surface_no_op surface none 4 C 1 volume_no_op volume none 5 R 3 remove_surface surface none 6 R 3 tweak_replace_surface surface surface 7 R 3 composite_surfaces surface surface 8 R 3 collapse_curve curve vertex 9 R 3 remove_topology_curve curve curve 10 R 3 virtual_collapse_curve curve vertex 11 R 3 remove_topology_surface surface surface 12 R 3 blunt_tangency vertex none remove_mat, angle, depth 13 R 3 remove_cone surface none 14 R 3 collapse_angle vertex none real_split, angle, comp_vertex 15 R 3 remove_blend surface none 16 R 3 remove_cavity surface none 17 R 3 copy_surface volume surface nsurfs, id0, loft0, thick0, ... 18 R 3 midsurface volume surf0 surf1, loft, thick 19 C 1 classify_surface surface none

Parameters

**entity1_ids** list of first entity ids associated with operation (see above table)

**entity2_ids** list of second entity associated with operation (see above table)

| | |
|---|---|
| **params** | array of parameters the operation needs to execute (see above table) |
| **mesh_size** | target mesh size for operation |
| **reduced_features** | optional currently supported only for "Volume No Operation=4". Uses 9 instead of 46 features for more efficient predictions |

### ◆ get_ML_operation_label_size()

int get_ML_operation_label_size ( const std::string *ml_op_name* )

for the given operation type return length of label vector the expected size of the feature vector

Parameters

| | |
|---|---|
| **ml_op_name** | name of ML model |

### ◆ get_ML_predictions()

```
std::vector<
std::vector< double >
> get_ML_predictions (std::vector< std::string >              ml_op_names,
                      std::vector< size_t >                  entity1_ids,
                      std::vector< size_t >                  entity2_ids,
                      std::vector< std::vector< double > >   params,
                      double                                 mesh_size,
                      bool                                   reduced_features = false
                     )
```

get machine learning predictions for the list of operations and corresponding entities. This function will load the ML training data if not already loaded. It will first compute features and then run predictions from training data. Uses scikit-learn EDT (Ensembles of Decision Trees) for predictions

Parameters

| | |
|---|---|
| **ml_op_names** | list of ML operation/model names |
| **entity1_ids** | list of first entity ids associated with operation (see table) |
| **entity2_ids** | list of second entity associated with operation (see table) |
| **params** | array of parameters the operation needs to execute (see table) |
| **mesh_size** | target mesh size for operation |
| **reduced_features** | experimental optional - currently supported only for volume_no_op uses 9 instead of 46 features for more efficient predictions |

Returns

resulting predictions. Vector of vectors - ordered one per operation. Length of return vector should be the number of labels used for the regression or classification method. Classification methods: volume_no_op, classify_surface length is number of categories. Values are confidence (0 to 1) in order of categories Regression methods: Copy Operation=17, Midsurface operation=18 length = 1, reward based fitness of operation (0 to 1) All other

regression methods: predicted quality metric outcome of indicated operation: [0] Success or Failure. (1 or 0). success or failure of operation [1] Scaled Jacobian (-1 to 1). scaled jacobian metric of operation at entity [2] Scaled In-radius (0 to 1). scaled in-radius of operation at entity. Scales tet in-radius by normalizing in-radius of an equilateral tet where the size is edge-length=mesh_size [3] Scaled Deviation (0 to inf). predicted deviation of the tets near the entity from the geometry. Is the distance from a tet face (tri center) at the boundary to the geometry scaled by mesh_size

### ◆ get_ML_regression_models()

```
std::vector< std::string > get_ML_regression_models ( )
```

get the available regression ML model names

### ◆ get_moment_magnitude()

```
double get_moment_magnitude (int entity_id)
```

Get the moment magnitude from a force.
Parameters

**entity_id** Id of the force

Returns

magnitude of the moment on the given force

### ◆ get_n_largest_distances_between_meshes()

```
std::vector< double >
get_n_largest_distances_between_meshes (int n,
                                        std::string entity_type1,
                                        std::vector< int > ids1,
                                        std::string entity_type2,
                                        std::vector< int > ids2
                                        )
```

Finds the 'n' largest distances between two meshes. These distances are from the nodes on the entities of 'ids1' to the elements in 'ids2'. Only triangle and face (quads) element types are supported. It is assumed that the meshes approximately line up. Each distance is returned with three values:

1. The distance between a node and element.
2. The node id.
3. The element id. The output is given in a vector of three doubles for each distance. So if the user asked for the three largest distances, the vector would contain the 9 doubles, with the distances in descending order.

Returns

vector of distance, node id, element id, distance nod id, element id, ...

### ◆ get_narrow_regions()

```
std::vector< int > get_narrow_regions(std::vector< int > target_ids,
                                      double             narrow_size
                                     )
```

Get the list of surfaces with narrow regions.
Parameters

**target_volume_ids** List of volume ids to examine.
**narrow_size** Indicate the size that defines 'narrowness'

Returns

List (python tuple) of surface ids

◆ get_narrow_surfaces()

```
std::vector< int > get_narrow_surfaces(std::vector< int > target_volume_ids,
                                       double             mesh_size
                                      )
```

Get the list of narrow surfaces for a list of volumes.
'Narrow' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'narrow' is.
Parameters

**target_volume_ids** List of volume ids to examine.
**mesh_size** Indicate the mesh size used as the threshold

Returns

List (python tuple) of small surface ids

◆ get_nearby_entities()

```
std::vector< int > get_nearby_entities(std::string        gtype,
                                       std::vector< int > ent_ids,
                                       std::vector< int > compare_ents,
                                       double             distance
                                      )
```

Get the list of nearby entities of type curve, surface or volumes from the model for a list of the same entity type.
Parameters

**gtype** "curve", "surface" or "volume"
**ent_ids** return entities close to the entities in this list
**entites** of same type to check against. If empty, will check against all of them
**distance** maximum distance betwen entities. Optional. Use -1 to compute default tolerance

Returns

list of entities of type gtype that are nearby to ent_ids from compare_volumes list

◆ get_nearby_volumes_at_volume()

```
std::vector< int > get_nearby_volumes_at_volume (int volume_id,
                                                 std::vector< int > compare_volumes,
                                                 double distance
                                                 )
```

Get the list of nearby volumes from the model for a single volume.
Parameters

**volume_id** volume to check.
**volumes** to check against. If empty, will check against all volumes in model
**maximum** distance betwen volumes. Optional. Use -1 to compute default tolerance

Returns
list of volumes that are nearby to volume_id from compare_volumes list

### ◆ get_next_block_id()

```
int get_next_block_id ( )
```

Get a next available block id.
Returns
Next available block id

### ◆ get_next_boundary_layer_id()

```
int get_next_boundary_layer_id ( )
```

### ◆ get_next_command_from_history()

```
std::string get_next_command_from_history ( )
```

Get 'next' command from history buffer.
Returns
A string which is the command

### ◆ get_next_group_id()

```
int get_next_group_id ( )
```

Get the next available group id from Cubit.

### ◆ get_next_nodeset_id()

```
int get_next_nodeset_id ( )
```

Get a next available nodeset id.
Returns
Next available nodeset id

### ◆ get_next_sideset_id()

```
int get_next_sideset_id ( )
```

Get a next available sideset id.

Returns
Next available sideset id

### ◆ get_nodal_coordinates()

std::array< double, 3 > get_nodal_coordinates ( int *node_id* )

Get the nodal coordinates for a given node id.
Parameters

**node_id** The node id

Returns
a triple (python tuple) containing the x, y, and z coordinates

### ◆ get_node_constraint()

bool get_node_constraint ( )

Query current setting for node constraint (move nodes to geometry)
Returns
True if constrained, otherwise false

### ◆ get_node_constraint_smart_metric()

std::string get_node_constraint_smart_metric ( )

Query current setting for node constraint smart metric Currently only for tets. Return either "distortion" of "normalized inradius".
Returns
Returns quality metric name for projecting mid-nodes

### ◆ get_node_constraint_smart_threshold()

double get_node_constraint_smart_threshold ( )

Query current setting for node constraint smart threshold.
Returns
Returns quality threshold for projecting mid-nodes

### ◆ get_node_constraint_value()

int get_node_constraint_value ( )

Query current setting for node constraint (move nodes to geometry)
Returns
Returns 0 (off), 1(on), 2(smart)

### ◆ get_node_count()

int get_node_count ( )

Get the count of nodes in the model.
Returns
The number of nodes in the model

### ◆ get_node_edges()

std::vector< int > get_node_edges(int *node_id*)

Get the edge ids that share a node.
Parameters

**node_id** The node id

Returns

List (python tuple) of edge ids adjacent to the node

◆ get_node_exists()

bool get_node_exists(int *node_id*)

Check the existance of a node.
Parameters

**node_id** The node id

Returns

true or false

◆ get_node_faces()

std::vector< int > get_node_faces(int *node_id*)

|brief Get the face/quad ids that share a node
Parameters

**node_id** The node id

Returns

List (python tuple) of face/quad ids adjacent the node

◆ get_node_global_id()

int get_node_global_id(int *node_id*)

Given a node id, return the global element id that is assigned when the mesh is exported.

```
int gid = CubitInterface::get_node_global_id (22);
```

CubitInterface::get_node_global_id
int get_node_global_id(int node_id)
Given a node id, return the global element id that is assigned when the mesh is exported.

Parameters

**node_id** Specifies the id of the sphere

Returns

The corresponding global node id

◆ get_node_position_fixed()

bool get_node_position_fixed(int *node_id*)

Query "fixedness" state of node. A fixed node is not affecting by smoothing.
Parameters

**node_id** The node id

Returns

True if constrained, otherwise false

◆ get_node_tris()

std::vector< int > get_node_tris ( int *node_id* )

|brief Get the tri ids that share a node

Parameters

**node_id** The node id

Returns

List (python tuple) of tri ids adjacent the node

◆ get_nodeset_children()

void get_nodeset_children ( int *nodeset_id*,
std::vector< int > & *returned_node_list*,
std::vector< int > & *returned_volume_list*,
std::vector< int > & *returned_surface_list*,
std::vector< int > & *returned_curve_list*,
std::vector< int > & *returned_vertex_list*
)

get lists of any and all possible children of a nodeset

A nodeset can contain a variety of entity types. This routine will return all contents of a specified nodeset.

Parameters

**nodeset_id** User specified id of the desired nodeset

**node_list** User specified list where nodes associated with this nodeset are returned

**volume_list** User specified list where volumes associated with this nodeset are returned

**surface_list** User specified list where surfaces associated with this nodeset are returned

**curve_list** User specified list where curves associated with this nodeset are returned

**vertex_list** User specified list where vertices associated with this nodeset are returned

◆ get_nodeset_count()

int get_nodeset_count ( )

Get the current number of sidesets.

Returns

The number of sidesets in the current model, if any

◆ get_nodeset_curves()

std::vector< int > get_nodeset_curves ( int *nodeset_id* )

Get a list of curve ids associated with a specific nodeset.
Parameters

**nodeset_id** User specified id of the desired nodeset

Returns

A list (python tuple) of curve ids contained in the nodeset

◆ get_nodeset_id_list()

std::vector< int > get_nodeset_id_list ( )

Get a list of all nodesets.
Returns

List (python tuple) of all active nodeset ids

◆ get_nodeset_id_list_for_bc()

std::vector< int > get_nodeset_id_list_for_bc ( CI_BCTypes *bc_type_enum*,
int *bc_id*
)

Get a list of all nodesets the specified bc is applied to.
Parameters

**bc_type_in** Type of bc to query, as defined by enum CI_BCTypes. 1-9 is FEA, 10-30 is CFD

**bc_id** ID of the bc to query

Returns

A list (python tuple) of nodeset ID's associated with that bc

◆ get_nodeset_node_count()

int get_nodeset_node_count ( int *nodeset_id* )

Get the number of nodes in a nodeset.
Parameters

**nodeset_id** The nodeset id

Returns

Number of nodes in the nodeset

◆ get_nodeset_nodes()

std::vector< int > get_nodeset_nodes ( int *nodeset_id* )

Get a list of node ids associated with a specific nodeset. This only returns the nodes that were specifically assigned to this nodeset. If the nodeset was created as a piece of geometry, get_nodeset_nodes will not return the nodes on that geometry See also get_nodeset_nodes_inclusive.
Parameters

**nodeset_id** User specified id of the desired nodeset

Returns
A list (python tuple) of node ids contained in the nodeset

◆ get_nodeset_nodes_inclusive()

std::vector< int > get_nodeset_nodes_inclusive(int *nodeset_id*)

Get a list of node ids associated with a specific nodeset. This includes all nodes specifically assigned to the nodeset, as well as nodes associated to a piece of geometry which was used to define the nodeset.
Parameters

**nodeset_id** User specified id of the desired nodeset
Returns
A list (python tuple) of node ids contained in the nodeset

◆ get_nodeset_surfaces()

std::vector< int > get_nodeset_surfaces(int *nodeset_id*)

Get a list of surface ids associated with a specific nodeset.
Parameters

**nodeset_id** User specified id of the desired nodeset
Returns
A list (python tuple) of surface ids contained in the nodeset

◆ get_nodeset_vertices()

std::vector< int > get_nodeset_vertices(int *nodeset_id*)

Get a list of vertex ids associated with a specific nodeset.
Parameters

**nodeset_id** User specified id of the desired nodeset
Returns
A list (python tuple) of vertex ids contained in the nodeset

◆ get_nodeset_volumes()

std::vector< int > get_nodeset_volumes(int *nodeset_id*)

Get a list of volume ids associated with a specific nodeset.
Parameters

**nodeset_id** User specified id of the desired nodeset
Returns
A list (python tuple) of volume ids contained in the nodeset

◆ get_num_volume_shells()

int get_num_volume_shells(int *volume_id*)

Get the number of shells in this volume.
Parameters

**volume_id** ID of the volume

Returns

Number of shells in the volume

### ◆ get_overlap_max_angle()

double get_overlap_max_angle(void )

Get the max angle setting for calculating surface overlaps.

Returns

The max angle setting

### ◆ get_overlap_max_gap()

double get_overlap_max_gap(void )

Get the max gap setting for calculating surface overlaps.

Returns

The max gap setting

### ◆ get_overlap_min_gap()

double get_overlap_min_gap(void )

Get the min gap setting for calculating surface overlaps.

Returns

The min gap setting

### ◆ get_overlapping_curves()

```
void get_overlapping_curves(std::vector< int >          target_surface_ids,
                            double                      min_gap,
                            double                      max_gap,
                            std::vector< int > &        returned_curve_list_1,
                            std::vector< int > &        returned_curve_list_2,
                            std::vector< double > &     returned_distance_list
                           )
```

For every occurance of two overlapping curves, two curve ids are returned. Those ids are returned in the indicated lists and are aligned. In other words the first id in curv_list_1 overlaps with the first id in curv_list_2. The second id in curv_list_1 overlaps with the second id in curv_list-2, and so on.

Parameters

| | |
|---|---|
| **target_surface_ids** | List of surface ids to examine. |
| **min_gap** | minimum overlap distance between curves to return |
| **max_gap** | maximum overlap distance between curves to return |
| **returned_curve_list_1** | User specified list where the ids of overlapping curves will be returned |
| **returned_curve_list_2** | User specified list where the ids of overlapping curves will be returned |

| | |
|---|---|
| **returned_distance_list** | Corresponding user specified list where distances between curves will be returned |

◆ get_overlapping_surfaces()

```
void
get_overlapping_surfaces (std::vector< int >          target_surface_ids,
                          std::vector< int > &        returned_surface_list_1,
                          std::vector< int > &        returned_surface_list_2,
                          std::vector< double > &     returned_distance_list,
                          std::vector< double > &     returned_overlap_area_list,
                          bool                        filter_slivers = false,
                          bool                        filter_volume_overlaps = false,
                          int                         cache_overlaps = 0
                          )
```

**This function only works from C++** Get the list of overlapping surfaces for a list of surfaces
For every occurance of two overlapping surfaces, two surfaces ids are returned. Those ids are returned in the indicated lists and are aligned. In other words the first id in surf_list_1 overlaps with the first id in surf_list_2. The second id in surf_list_1 overlaps with the second id in surf_list-2, and so on.
Parameters

| | |
|---|---|
| **target_surface_ids** | List of surface ids to examine. |
| **returned_surface_list_1** | User specified list where the ids of overlapping surfaces will be returned |
| **returned_surface_list_2** | User specified list where the ids of overlapping surfaces will be returned |
| **returned_distance_list** | Corresponding user specified list where distances between surfaces will be returned |
| **returned_overlap_area_list** | Corresponding user specified list where overlap areas between surfaces will be returned |
| **filter_slivers** | whether to return filter slivers |
| **filter_volume_overlaps** | whether to return surfaces on the same volume |
| **cache_overlaps** | speed up overlaps by caching and using previously computed results. Default 0 = no caching. 1 = clear out old values first. 2 = use and add to existing cache |

◆ get_overlapping_surfaces_at_surface()

```
std::vector< int >
get_overlapping_surfaces_at_surface (int                surface_id,
                                     std::vector< int > compare_volumes,
                                     int                cache_overlaps = 0
                                     )
```

Get the list of overlapping surfaces from the model for a single surface.

Parameters

**surface_id** surface to check.

**compare_volumes** volumes to check against. If empty, will check against all volumes in model

Returns

list of surfaces that overlap surface_id from compare_volumes list

◆ get_overlapping_surfaces_in_bodies()

```
std::vector< std::vector< int > >
get_overlapping_surfaces_in_bodies (std::vector< int > body_ids,
                                    bool filter_slivers = false
                                    )
```

returns a vector of vectors defining surface overlaps The first surface (id) in each vector overlaps with all subsequent surfaces in the vector.

Parameters

**body_ids** List of bodies to search for surface overlaps

**filter_sliver** Optional parameter that removes false positives from the output omitting overlapping pairs sharing a merged curve sharing merged curves.

```
      bodies = [ 15, 19, 24, 88 ]

      my_overlaps = cubit.get_overlapping_surfaces_in_bodies( bodies )
```

◆ get_overlapping_surfaces_in_volumes()

```
void
get_overlapping_surfaces_in_volumes (std::vector< int > target_volume_ids,
                                     std::vector< int > & returned_surface_list_1,
                                     std::vector< int > & returned_surface_list_2,
                                     std::vector< double > & returned_distance_list,
                                     std::vector< double > & returned_overlap_area_list,
                                     bool filter_slivers = false,
                                     bool filter_volume_overlaps = false,
                                     int cache_overlaps = 0
                                     )
```

**This function only works from C++** Get the list of overlapping surfaces for a list of volumes

For every occurance of two overlapping surfaces, two surfaces ids are returned. Those ids are returned in the indicated lists and are aligned. In other words the first id in surf_list_1 overlaps with the first id in surf_list_2. The second id in surf_list_1 overlaps with the second id in surf_list-2, and so on.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **surf_list_1** | User specified list where the ids of overlapping surfaces will be returned |
| **surf_list_2** | User specified list where the ids of overlapping surfaces will be returned |
| **returned_distance_list** | Corresponding user specified list where distances between surfaces will be returned |
| **returned_overlap_area_list** | Corresponding user specified list where overlap areas between surfaces will be returned |
| **filter_slivers** | whether to return filter slivers |
| **filter_volume_overlaps** | whether to return surfaces on the same volume |
| **cache_overlaps** | speed up overlaps by caching and using previously computed results. Default 0 = no caching. 1 = clear out old values first. 2 = use and add to existing cache |

### ◆ get_overlapping_volumes()

std::vector< int > get_overlapping_volumes ( std::vector< int > *target_volume_ids* )

Get the list of overlapping volumes for a list of volumes.
For every occurance of two overlapping volumes, two volume ids are returned in volume_list. Modulus 2 of the volume_list should always be 0 (the list should contain an even number of volume ids). The first volume id in the returned list overlaps with the second volume id. The third volume id overlaps with the fourth volume id, and so on.
Parameters

**target_volume_ids** List of volume ids to examine.

Returns

List (python tuple) of overlapping volumes ids

### ◆ get_overlapping_volumes_at_volume()

std::vector< int > get_overlapping_volumes_at_volume ( int *volume_id*, std::vector< int > *compare_volumes* )

Get the list of overlapping volumes from the model for a single volume.
Parameters

**volume_id** volume to check.
**volumes** to check against. If empty, will check against all volumes in model

Returns
list of volumes that overlap volume_id from compare_volumes list

◆ get_owning_body()

int get_owning_body(const std::string & *geometry_type*,
int *entity_id*
)

Get the owning body for a specified entity.

```
int body_id = CubitInterface::get_owning_body ("curve" , 12);
```

CubitInterface::get_owning_body
int get_owning_body(const std::string &geometry_type, int entity_id)
Get the owning body for a specified entity.

```
body_id = cubit.get_owning_body("curve" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns
ID of the specified entity's owning body

◆ get_owning_volume()

int get_owning_volume(const std::string & *geometry_type*,
int *entity_id*
)

Get the owning volume for a specified entity.

```
int volume_id = CubitInterface::get_owning_volume ("curve" , 12);
```

CubitInterface::get_owning_volume
int get_owning_volume(const std::string &geometry_type, int entity_id)
Get the owning volume for a specified entity.

```
volume_id = cubit.get_owning_volume("curve" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns
ID of the specified entity's owning volume

◆ get_owning_volume_by_name()

int get_owning_volume_by_name(const std::string & *entity_name*)

Get the owning volume for a specified entity.

```
int volume_id = CubitInterface::get_owning_volume_by_name
("TipSurface" );
```

CubitInterface::get_owning_volume_by_name
int get_owning_volume_by_name(const std::string &entity_name)

Get the owning volume for a specified entity.

```
volume_id = cubit.get_owning_volume_by_name("TipSurface" )
```

Parameters

**entity_name** Specifies the name (supplied by Cubit) of the entity

Returns

ID of the specified entity's owning volume or 0 if name is unknown

### ◆ get_owning_volume_ids()

```
void get_owning_volume_ids(const std::string &  entity_type,
                           std::vector< int > & entity_list,
                           std::vector< int > & volume_ids
                          )
```

Gets the id's of the volumes that are owners of one of the specified entities.

Parameters

**entity_type**
**entity_list**
**vol_ids**

### ◆ get_parent_assembly_instance()

```
int get_parent_assembly_instance(int assembly_id)
```

Get the stored instance number of an assembly node's instance.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Instance of the assembly node' instance

### ◆ get_parent_assembly_path()

```
std::string get_parent_assembly_path(int assembly_id)
```

Get the stored path of an assembly node' parent.

Parameters

**assembly_id** Id that identifies the assembly node

Returns

Path of the assembly node' parent

### ◆ get_periodic_data()

```
void get_periodic_data(const std::string & geometry_type,
                       int                 entity_id,
                       double &            returned_interval,
                       std::string &       returned_firmness,
                       int &               returned_lower_bound,
                       std::string &       returned_upper_bound
```

)

Get the periodic data for a surface or curve.
Parameters

| | |
|---|---|
| **geom_type** | Specifies the geometry type of the entity |
| **entity_id** | Specifies the id of the entity |
| **interval** | User specified variable where interval count for the specified entity is returned |
| **firmness** | User specified variable where a firmness of 'hard', 'soft', or 'default' is returned |
| **lower_bound** | User specified variable where the lower bound value is returned |
| **upper_bound** | User specified variable where the upper bound value is returned |

### ◆ get_pick_filters()

std::vector< std::string > get_pick_filters ( )

Get a list of the current pick filters.

### ◆ get_pick_type()

const char * get_pick_type ( )

Get the current pick type.
Returns
The current pick type of the graphics system

### ◆ get_pressure_function()

std::string get_pressure_function ( int *entity_id* )

Get the pressure function.
Parameters

**entity_id** Id of the pressure

Returns
The pressure function

### ◆ get_pressure_value()

double get_pressure_value ( int *entity_id* )

Get the pressure value.
Parameters

**entity_id** Id of the pressure

Returns
The value or magnitude of the given pressure

### ◆ get_previous_command_from_history()

std::string get_previous_command_from_history ( )

Get 'previous' command from history buffer.

Returns
A string which is the command

### ◆ get_pyramid_count()

int get_pyramid_count ( )

Get the count of pyramids in the model.
Returns
The number of pyramids in the model

### ◆ get_pyramid_global_element_id()

int get_pyramid_global_element_id ( int *pyramid_id* )

Given a pyramid id, return the global element id.

```
int gid = CubitInterface::get_pyramid_global_element_id (22);
```

CubitInterface::get_pyramid_global_element_id
int get_pyramid_global_element_id(int pyramid_id)
Given a pyramid id, return the global element id.

Parameters

**pyramid_id** Specifies the id of the pyramid

Returns
The corresponding element id

### ◆ get_python_version()

std::string get_python_version ( )

get the python version used in cubit
Returns
A string containing the python version number

### ◆ get_quad_count()

int get_quad_count ( )

Get the count of quads in the model.
Returns
The number of quads in the model

### ◆ get_quad_global_element_id()

int get_quad_global_element_id ( int *quad_id* )

Given a quad id, return the global element id.

```
int gid = CubitInterface::get_quad_global_element_id (22);
```

CubitInterface::get_quad_global_element_id
int get_quad_global_element_id(int quad_id)
Given a quad id, return the global element id.

Parameters

**quad_id** Specifies the id of the quad

Returns

The corresponding element id

◆ get_quality_stats()

```
void get_quality_stats(const std::string &   entity_type,
                       std::vector< int >    id_list,
                       const std::string &   metric_name,
                       double                single_threshold,
                       bool                  use_low_threshold,
                       double                low_threshold,
                       double                high_threshold,
                       double &              min_value,
                       double &              max_value,
                       double &              mean_value,
                       double &              std_value,
                       int &                 min_element_id,
                       int &                 max_element_id,
                       std::vector< int > &  mesh_list,
                       std::string &         element_type,
                       int &                 bad_group_id,
                       bool                  make_group = false
                      )
```

Get the quality stats for a specified entity.

Parameters

| | |
|---|---|
| **entity_type** | Specifies the geometry type of the entity |
| **id_list** | Specifies a list of ids to work on |
| **metric_name** | Specify the metric used to determine the quality |
| **single_threshold** | Quality threshold value |
| **use_low_threshold** | use threshold as lower or upper bound |
| **low_threshold** | Quality threshold when using a lower and upper range |
| **high_threshold** | Quality threshold when using a lower and upper range |
| **min_value** | Quality value of the worst element |
| **max_value** | Quality value of the best element |
| **mean_value** | Average quality value of all elements |
| **std_value** | Std deviationvalue of all elements |
| **min_element_id** | ID of the worst element |
| **max_element_id** | ID of the best element |
| **mesh_list** | list of failed elements |
| **element_type** | type of failed elements (does not support mixed element types) |
| **make_group** | whether to create a group or not |
| **bad_group_id** | ID of the created group |

| | |
|---|---|
| **min_value** | User specified variable where the minimum quality value will be returned |
| **max_value** | User specified variable where the maximum quality value will be returned |
| **mean_value** | User specified variable where the mean quality value will be returned |
| **std_value** | User specified variable where the standard deviation quality value will be returned |

◆ get_quality_stats_at_geometry()

```
std::vector< double >
get_quality_stats_at_geometry (const std::string & geom_type,
                               const std::string & mesh_type,
                               const std::vector< int > geom_id_list,
                               const int expand_levels,
                               const std::string & metric_name,
                               const double single_threshold,
                               const bool use_low_threshold,
                               const double low_threshold,
                               const double high_threshold,
                               const bool make_group
                              )
```

get element quality at a list of geometry entities. Finds all elements with nodes ON/IN the specified geometry and finds the quality of all elements of the specfied element type that are connected. Same arguments and return values as get_elem_quality_stats except a geometry and element type are used as arguments

```
      std::vector<int> geom_ids = {4, 5};
      expand_levels = 2
      double single_threshold = 0.2;
      bool use_low_threshold = false ;
      double low_threshold = 0.0;
      double high_threshold = 0.0;
      bool make_group = true ;
      std::vector<double>
      quality_data     =     CubitInterface::get_quality_stats_at_geometry
("surface" , "tet" ,
      geom_ids, expand_levels, "scaled jacobian" ,
      single_threshold, use_low_threshold,
      low_threshold, high_threshold,
      make_group);
      double min_value = quality_data[0];
      double max_value = quality_data[1];
      double mean_value = quality_data[2];
      double std_value = quality_data[3];
      int min_element_id = (int)quality_data[4];
      int max_element_id = (int)quality_data[5];
      int element_type = (int)quality_data[6];
      int bad_group_id = (int)quality_data[7];
```

```
int num_elems = (int)quality_data[8];
std::vector<int> elem_ids(num_elems);
for (int i=9, j=0; i<quality_data.size(); i++, j++)
elem_ids[j] = (int )quality_data[i];
```

CubitInterface::get_quality_stats_at_geometry

std::vector< double > get_quality_stats_at_geometry(const std::string &geom_type, const std::string &mesh_type, const std::vector< int > geom_id_list, const int expand_levels, const std::string &metric_name, const double single_threshold, const bool use_low_threshold, const double low_threshold, const double high_threshold, const bool make_group)

get element quality at a list of geometry entities. Finds all elements with nodes ON/IN the specified...

Parameters

| | |
|---|---|
| **geom_type** | Specifies the geometry type of the entities |
| **mesh_type** | Specifies the element type to find quality at geom entities |
| **id_list** | Specifies a list of geometry entity ids to work on |
| **expand_levels** | Number of element levels from target geometry to expand |
| **metric_name** | Specify the metric used to determine the quality |
| **single_threshold** | Quality threshold value |
| **use_low_threshold** | use threshold as lower or upper bound |
| **low_threshold** | Quality threshold when using a lower and upper range |
| **high_threshold** | Quality threshold when using a lower and upper range |

Returns

[0] min_value [1] max_value [2] mean_value [3] std_value [4] min_element_id [5] max_element_id [6] element_type 0 = edge, 1 = tri, 2 = quad, 3 = tet, 4 = hex [7] bad_group_id [8] size of mesh_list [9]...[n-1] mesh_list

## ◆ get_quality_value()

```
double get_quality_value ( const std::string & mesh_type,
                           int                 mesh_id,
                           const std::string & metric_name
                         )
```

Get the metric value for a specified mesh entity.

```
CubitInterface::get_quality_value ("hex" , 223, "skew" );
```

CubitInterface::get_quality_value

double get_quality_value(const std::string &mesh_type, int mesh_id, const std::string &metric_name)

Get the metric value for a specified mesh entity.

Parameters

| | |
|---|---|
| **mesh_type** | Specifies the mesh entity type (hex, tet, tri, quad) |
| **mesh_id** | Specifies the id of the mesh entity |
| **metric_name** | Specifies the name of the metric (skew, taper, jacobian, etc) |

Returns

The value of the quality metric

### ◆ get_quality_values()

std::vector< double > get_quality_values ( const std::string & *mesh_type*,
std::vector< int > *mesh_ids*,
const std::string & *metric_name*
)

Get the metric values for specified mesh entities.

```
CubitInterface::get_quality_value ("hex" , [223, 224, 225] "skew" );
```

Parameters

**mesh_type** Specifies the mesh entity type (hex, tet, tri, quad)
**mesh_ids** Specifies the ids of the mesh entities
**metric_name** Specifies the name of the metric (skew, taper, jacobian, etc)

Returns

The values of the quality metric

### ◆ get_reduce_bolt_core_default_dimensions()

std::vector< double > get_reduce_bolt_core_default_dimensions ( int *vol_id* )

get default dimensions for reduce vol bolt core operation

Parameters

**vol_id** volume ID. Should represent bolt geometry

Returns

c1, c2, c3 dimensions

### ◆ get_relatives()

std::vector< int > get_relatives ( const std::string & *source_geometry_type*,
int *source_id*,
const std::string & *target_geom_type*
)

Get the relatives (parents/children) of a specified entity.
This can be used to get either ancestors or predecessors for a specific entity. Only one specified entity type is returned with one use of the routine. For example, to get all surface parents associated with **Curve** 1, 'curve' is the source_geometry_type, '1' is the source_id, and 'surface' is the target_geom_type.

```
std::vector<int> relative_list;
curve_list = CubitInterface::get_relatives ("surface" , 12, "curve"
);
```

CubitInterface::get_relatives

std::vector< int > get_relatives(const std::string &source_geometry_type, int source_id, const std::string &target_geom_type)

Get the relatives (parents/children) of a specified entity.

```
curve_list = cubit.get_relatives("surface" , 12, "curve" )
```

Parameters

**source_geom_type** The entity type of the source entity
**source_id** The id of the source entity
**target_geom_type** The target geometry type

Returns

A list (python tuple) of ids of the target geometry type

### ◆ get_rendering_mode()

int get_rendering_mode ( )

Get the current rendering mode.

Returns

The current rendering mode of the graphics subsystem

### ◆ get_requested_mesh_interval_firmness()

std::string get_requested_mesh_interval_firmness ( const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh interval firmness for the specified entity as set specifically on the entity.

```
std::string firmness;
CubitInterface::get_requested_mesh_interval_firmness ("surface" , 12);
```

CubitInterface::get_requested_mesh_interval_firmness
std::string get_requested_mesh_interval_firmness(const std::string &geometry_type, int entity_id)
Get the mesh interval firmness for the specified entity as set specifically on the entity.

```
firmness = cubit.get_requested_mesh_interval_firmness("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns

The entity's meshing firmness (HARD, SOFT, LIMP) HARD = set directly SOFT = computed LIMP = not set

### ◆ get_requested_mesh_intervals()

int get_requested_mesh_intervals ( const std::string & *geometry_type*,
int *entity_id*
)

Get the interval count for a specified entity as set specifically on that entity.

```
int intervals = CubitInterface::get_meshed_intervals("surface" , 12);
```

```
intervals = cubit.get_meshed_intervals("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity

**entity_id** Specifies the id of the entity

Returns

The entity's interval count

### ◆ get_requested_mesh_size()

double get_requested_mesh_size ( const std::string & *geometry_type*,
int *id*
)

Get the requested mesh size for a specified entity. This returns a size that has been set specifically on the entity and not averaged from parents.

```
double mesh_size = CubitInterface::get_requested_meshed_size("volume" , 2);
mesh_size = cubit.get_mesh_size("volume" , 2)
```

Parameters

**geom_type** Specifies the geometry type of the entity

**entity_id** Specifies the id of the entity

Returns

The entity's requested mesh size

### ◆ get_requested_mesh_size_type()

std::string get_requested_mesh_size_type ( const std::string & *geometry_type*,
int *entity_id*
)

Get the mesh size setting type for the specified entity as set specifically on the entity.

```
std::string firmness;
CubitInterface::get_requested_mesh_size_setting_type("surface" , 12);
firmness = cubit.get_requested_mesh_size_setting_type("surface" , 12)
```

Parameters

**geom_type** Specifies the geometry type of the entity

**entity_id** Specifies the id of the entity

Returns

The entity's mesh size type (USER_SET, CALCULATED, NOT_SET)

### ◆ get_revision_date()

std::string get_revision_date ( )

Get the Cubit revision date.

Returns

A string containing Cubit's last date of revision

### ◆ get_rubberband_shape()

int get_rubberband_shape ( )

Get the current rubberband select mode.
Returns
0 for box, 1, for polygon, 2 for circle

### ◆ get_selected_id()

int get_selected_id(int *index*)

Get the selected id based on an index.
Returns
An id based on the passed in index

### ◆ get_selected_ids()

std::vector< int > get_selected_ids( )

Get a list of the currently selected ids.
Returns
A list of the currently selected ids

### ◆ get_selected_type()

std::string get_selected_type(int *index*)

Get the selected type based on an index.
Returns
A type based on the passed in index

### ◆ get_sharp_angle_vertices()

```
std::vector< std::vector< double > >
get_sharp_angle_vertices (std::vector< int > target_volume_ids,
                          double             upper_bound,
                          double             lower_bound
                          )
```

Get the list of vertices at sharp curve angles for a list of volumes returns two parallel arrays. First array are the vertex ids and second are the associated angles at the vertices.
'Sharp' is a function of the upper_bound and lower_bound threshold parameters. The id of vertices is returned. Similar to get_sharp_curve_angles except only vertices are returned with angles above upper_bound and below lower_bound
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **upper_bound** | Upper threshold angle |
| **lower_bound** | Lower threshold angle |

### ◆ get_sharp_curve_angles()

```
void get_sharp_curve_angles(std::vector< int >   target_volume_ids,
                            std::vector< int > & returned_large_curve_angles,
```

```
                                    std::vector< int > &        returned_small_curve_angles,
                                    std::vector< double > &     returned_large_angles,
                                    std::vector< double > &     returned_small_angles,
                                    double                      upper_bound,
                                    double                      lower_bound
                                   )
```

Get the list of sharp curve angles for a list of volumes.
'Sharp' is a function of the upper_bound and lower_bound threshold parameters. The id of curves are returned when any angle associated with a curve is less than the lower_bound or greater than the upper_bound.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **large_curve_angles** | User specified list where the ids of curves with curve angles will be returned |
| **small_curve_angles** | User specified list where the ids of curves with small angles will be returned |
| **large_angles** | User specified list where the angles associated with large_curve_angles will be returned. Angles returned are in the same order as the ids returned in large_curve_angles. |
| **small_angles** | User specified list where the angles associated with small_curve_angles will be returned. Angles returned are in the same order as the ids returned in small_curve_angles. |
| **upper_bound** | Upper threshold angle |
| **lower_bound** | Lower threshold angle |

◆ get_sharp_surface_angles()

```
void
get_sharp_surface_angles   (std::vector< int >          target_volume_ids,
                            std::vector< int > &        returned_large_surface_angles,
                            std::vector< int > &        returned_small_surface_angles,
                            std::vector< double > &     returned_large_angles,
                            std::vector< double > &     returned_small_angles,
                            double                      upper_bound,
                            double                      lower_bound
                           )
```

Get the list of sharp surface angles for a list of volumes.
'Sharp' is a function of the upper_bound and lower_bound threshold parameters. The id of surfaces are returned when any angle associated with a surface is less than the lower_bound or greater than the upper_bound.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |

| | |
|---|---|
| **large_surface_angles** | User specified list where the ids of surfaces with large angles will be returned |
| **small_surface_angles** | User specified list where the ids of surfaces with small angles will be returned |
| **large_angles** | User specified list where the angles associated with large_surface_angles will be returned. Angles returned are in the same order as the ids returned in large_surface_angles. |
| **small_angles** | User specified list where the angles associated with small_surface_angles will be returned. Angles returned are in the same order as the ids returned in small_surface_angles. |
| **upper_bound** | Upper threshold angle |
| **lower_bound** | Lower threshold angle |

### ◆ get_sideset_area()

double get_sideset_area ( int *sideset_id* )

Get area of the sideset.
A sideset can contain tris or face elements. This function will return area of tri or face elements if they exist. Otherwise it will return zero.
Parameters

| | |
|---|---|
| **sideset_id** | User specified id of the desired sideset |

Returns
summation of area of all tris or faces

### ◆ get_sideset_children()

```
void get_sideset_children ( int                    sideset_id,
                            std::vector< int > &   returned_face_list,
                            std::vector< int > &   returned_surface_list,
                            std::vector< int > &   returned_curve_list
                          )
```

get lists of any and all possible children of a sideset
A nodeset can contain a variety of entity types. This routine will return all contents of a specified sideset.
Parameters

| | |
|---|---|
| **sideset_id** | User specified id of the desired sideset |
| **face_list** | User specified list where faces associated with this sideset are returned |
| **surface_list** | User specified list where surfaces associated with this sideset are returned |

**curve_list** User specified list where curves associated with this sideset are returned

◆ get_sideset_count()

int get_sideset_count ( )

Get the current number of sidesets.
Returns
The number of sidesets in the current model, if any

◆ get_sideset_curves()

std::vector< int > get_sideset_curves ( int *sideset_id* )

Get a list of curve ids associated with a specific sideset.
Parameters

**sideset_id** User specified id of the desired sideset
Returns
A list (python tuple) of curve ids contained in the sideset

◆ get_sideset_edges()

std::vector< int > get_sideset_edges ( int *sideset_id* )

Get a list of any quads in a sideset.
A sideset can contain edge elements. This function will return those edge elements if they exist. An empty list will be returned if there are no edges in the sideset.
Parameters

**sideset_id** User specified id of the desired sideset
Returns
A list (python tuple) of the edges in the sideset

◆ get_sideset_element_type()

std::string get_sideset_element_type ( int *sideset_id* )

Get the element type of a sideset.
Parameters

**sideset_id** The id of the sideset to be queried
Returns
Element type

◆ get_sideset_id_list()

std::vector< int > get_sideset_id_list ( )

Get a list of all sidesets.
Returns
List (python tuple) of all active sideset ids

◆ get_sideset_id_list_for_bc()

std::vector< int > get_sideset_id_list_for_bc ( CI_BCTypes *bc_type_enum*,
int *bc_id*
)

Get a list of all sidesets the specified bc is applied to.
Parameters

**bc_type_in** Type of bc to query, as defined by enum CI_BCTypes. 1-9 is FEA, 10-30 is CFD
**bc_id** ID of the bc to query

Returns
A list (python tuple) of sideset ID's associated with that bc

◆ get_sideset_quads()

std::vector< int > get_sideset_quads ( int *sideset_id* )

Get a list of any quads in a sideset.
A sideset can contain quadrilateral elements. This function will return those quad elements if they exist. An empty list will be returned if there are no quads in the sideset.
Parameters

**sideset_id** User specified id of the desired sideset

Returns
A list (python tuple) of the quads in the sideset

◆ get_sideset_surfaces()

std::vector< int > get_sideset_surfaces ( int *sideset_id* )

Get a list of any surfaces in a sideset.
A sideset can contain surfaces. This function will return those surfaces if they exist. An empty list will be returned if there are no surfaces in the sideset.
Parameters

**sideset_id** User specified id of the desired sideset

Returns
A list (python tuple) of the surfaces defining the sideset

◆ get_sideset_tris()

std::vector< int > get_sideset_tris ( int *sideset_id* )

Get a list of any tris in a sideset.
A sideset can contain tris elements. This function will return those tri elements if they exist. An empty list will be returned if there are no tris in the sideset.
Parameters

**sideset_id** User specified id of the desired sideset

Returns

A list (python tuple) of the tris in the sideset

### ◆ get_similar_curves()

```
std::vector< int > get_similar_curves(std::vector< int > curve_ids,
                                      double             tol = 1e-3,
                                      bool               use_percent_tol = true,
                                      bool               on_similar_vols = true
                                     )
```

Get similar curves with the same length.
Parameters

| | |
|---|---|
| **curve_ids** | IDs of curve to compare against |
| **tol** | tolerance for comparison |
| **use_percent_tol** | tolerance is a percentage (0-1) of length, otherwise absolute length |
| **on_similar_vols** | check only curves on volumes that are similar to the curve_ids' owning volume(s) |

Returns
list of IDs of similar curves

### ◆ get_similar_surfaces()

```
std::vector< int > get_similar_surfaces(std::vector< int > surface_ids,
                                        double             tol = 1e-3,
                                        bool               use_percent_tol = true,
                                        bool               on_similar_vols = true
                                       )
```

Get similar surfaces with the same area and number of curves.
Parameters

| | |
|---|---|
| **surface_ids** | IDs of surface to compare against |
| **tol** | tolerance for comparison |
| **use_percent_tol** | tolerance is a percentage (0-1) of area, otherwise absolute area |
| **on_similar_vols** | check only surfaces on volumes that are similar to the surface_ids' owning volume(s) |

Returns
list of IDs of similar surfaces

### ◆ get_similar_volumes()

```
std::vector< int > get_similar_volumes(std::vector< int > volume_ids,
                                       double             tol = 1e-3,
                                       bool               use_percent_tol = true
                                      )
```

Get similar volumes with the same volume and number of faces.

Parameters

| | |
|---|---|
| **volume_ids** | IDs of volume(s) to compare against //! |
| **tol** | tolerance for comparison |
| **use_percent_tol** | tolerance is a percentage (0-1) of volume, otherwise absolute volume |

Returns

list of IDs of similar volumes

### ◆ get_sizing_function_name()

```
std::string get_sizing_function_name ( const std::string & entity_type,
                                       int                 surface_id
                                     )
```

Get the sizing function name for a surface or volume.

Parameters

| | |
|---|---|
| **entity_type** | Type (volume or surface) |
| **entity_id** | Id of the entity |

Returns

The sizing function name (constant, curvature, interval, inverse, linear, super, test, exodus, none)

### ◆ get_small_and_narrow_surfaces()

```
std::vector< int >
get_small_and_narrow_surfaces ( std::vector< int > target_ids,
                                double             small_area,
                                double             small_curve_size
                              )
```

Get the list of small or narrow surfaces from a list of volumes.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **small_area** | Indicate the area threshold |
| **small_curve_size** | Indicate size for 'narrowness' |

Returns

List (python tuple) of small or narrow surface ids

### ◆ get_small_curves()

```
std::vector< int > get_small_curves ( std::vector< int > target_volume_ids,
                                      double             mesh_size
                                    )
```

Get the list of small curves for a list of volumes.
'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.

Parameters

**target_volume_ids** List of volume ids to examine. in Cubit is valid as input here.
**mesh_size** Indicate the mesh size used as the threshold

Returns

List (python tuple) of small curve ids

◆ get_small_radius_blend_surfaces()

```
std::vector< int > get_small_radius_blend_surfaces (std::vector< int > target_volume_ids,
                                                    double max_radius
                                                    )
```

Get the list of blend surfaces for a list of volumes that have a radius of curvature smaller than max_radius.

Parameters

**target_volume_ids** List of volume ids to examine. max_radius maximum radius of curvature for which blend surfaces will be returned if max_radius = 0, then all blend surfaces will be returned.

Returns

List (python tuple) of blend surface ids

◆ get_small_surfaces()

```
std::vector< int > get_small_surfaces (std::vector< int > target_volume_ids,
                                       double mesh_size
                                       )
```

Get the list of small surfaces for a list of volumes.
'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.

Parameters

**target_volume_ids** List of volume ids to examine.
**mesh_size** Indicate the mesh size used as the threshold

Returns

List (python tuple) of small surface ids

◆ get_small_surfaces_HR()

```
std::vector< int > get_small_surfaces_HR (std::vector< int > target_volume_ids,
                                          double mesh_size
                                          )
```

Python callable version Get the list of small hydraulic radius surfaces for a list of volumes.

'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **mesh_size** | Indicate the mesh size used as the threshold |

Returns
return the list of small hydraulic radius surfaces (same as returned_small_surfaces)

◆ get_small_surfaces_hydraulic_radius()

```
void
get_small_surfaces_hydraulic_radius (std::vector< int > target_volume_ids,
                                     double mesh_size,
                                     std::vector< int > & returned_small_surfaces,
                                     std::vector< double > & returned_small_radius
                                     )
```

Get the list of small hydraulic radius surfaces for a list of volumes.
'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **mesh_size** | Indicate the mesh size used as the threshold |
| **returned_small_surfaces** | ids of small hydraulic radius surfaces will be returned |
| **returned_small_radius** | User The hydrualic radius of each small surface will be returned. The order of the radius values is the same as the order of the returned ids. |

Returns
return the list of small hydraulic radius surfaces (same as returned_small_surfaces)

◆ get_small_volumes()

```
std::vector< int > get_small_volumes (std::vector< int > target_volume_ids,
                                      double mesh_size
                                      )
```

Get the list of small volumes from a list of volumes.
'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. volumes with volume < 10*mesh_size^3 will be returned.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **mesh_size** | Indicate the mesh size used as the threshold |

Returns

List (python tuple) of small volume ids

### ◆ get_small_volumes_hydraulic_radius()

```
void
get_small_volumes_hydraulic_radiu
s                                    (std::vector< int >          target_volume_ids,
                                      double                      mesh_size,
                                                                  returned_small_volumes
                                      std::vector< int > &        ,
                                      std::vector< double > &     returned_small_radius
                                     )
```

Get the list of small hydraulic radius volumes for a list of volumes.
'Small' is a function of the mesh_size passed into the routine. The mesh_size parameter will act as the threshold for determining what 'small' is. A small entity is one that has an edge length smaller than mesh_size.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **mesh_size** | Indicate the mesh size used as the threshold |
| **small_volumes** | User specified list where the ids of small volumes will be returned |
| **small_radius** | User specified list where the radius of each small volume will be returned. The order of the radius values is the same as the order of the returned ids. |

### ◆ get_smallest_curves()

```
std::vector< int > get_smallest_curves(std::vector< int >  target_volume_ids,
                                       int                 number_to_return
                                      )
```

Get a list of the smallest curves in the list of volumes. The number returned is specified by 'num_to_return'.

Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. in Cubit is valid as input here. |
| **num_to_return** | Indicate the number of curves to return |

Returns

List (python tuple) of smallest curve ids

### ◆ get_smallest_features()

```
void get_smallest_features(std::vector< int >  target_ids,
```

```
                                    int &                    returned_number_to_return,
                                    std::vector< int > &     returned_type_1_list,
                                    std::vector< int > &     returned_type_2_list,
                                    std::vector< int > &     returned_id_1_list,
                                    std::vector< int > &     returned_id_2_list,
                                    std::vector< double > &  returned_distance_list
                                  )
```

Finds all of the smallest features.
Parameters

| | |
|---|---|
| **target_ids** | The entities to query |
| **num_to_return** | number of small features to return |
| **type1_list** | |
| **type2_list** | |
| **id1_list** | |
| **id2_list** | |
| **distance_list** | |

### ◆ get_smooth_scheme()

```
std::string get_smooth_scheme( const std::string &  geometry_type,
                               int                   entity_id
                             )
```

Get the smooth scheme for a specified entity.

```
std::string smooth_scheme;
CubitInterface::get_smooth_scheme ("curve" , 122, smooth_scheme);
```

CubitInterface::get_smooth_scheme
std::string get_smooth_scheme(const std::string &geometry_type, int entity_id)
Get the smooth scheme for a specified entity.

```
smooth_scheme = cubit.get_smooth_scheme("curve" , 122)
```

Parameters

| | |
|---|---|
| **geom_type** | Specifies the geometry type of the entity |
| **entity_id** | Specifies the id of the entity |

Returns
The smooth scheme associated with the entity

### ◆ get_solutions_for_bad_geometry()

```
std::vector< std::vector< std::string > >
get_solutions_for_bad_geometry (std::string  geom_type,
                                int          geom_id
                               )
```

Get lists of display strings and command strings for bad geometry.
Parameters

**geom_type** "curve", "surface", "volume" or "body"
**geom_id** ID of geometry entity

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_blends()

std::vector< std::vector< std::string > > get_solutions_for_blends(int *surface_id*)

Get the solution list for a given blend surface.

Parameters

**surface_id** the surface being queried
**max_radius** the maximum radius of curvature for which solutions will be returned
max_radius=-1 will return solutions for any blend

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Note: If using python, vectors will be python tuples.

### ◆ get_solutions_for_bolt()

std::vector< std::vector< std::string > > get_solutions_for_bolt(int *bolt_id*,
int *insert_id*,
int *threaded_vol_id*
)

get the solutions for a volume classified as a bole. faster than get_solutions_for_classified_volume if the lower volume and insert volumes are alraedy known

Parameters

**bolt_id** ID of a volume classified as a bolt
**insert_id** (optional) ID of a volume classified as insert. Used if insert is present at the bolt
**threaded_vol_id** (optional) if known, include the volume ID of the threaded (lower) volume for the bolt

Returns

cubit solutions

### ◆ get_solutions_for_bolt_hole()

std::vector< std::vector< std::string > > get_solutions_for_bolt_hole (int *bearing_hole*,
std::vector< int > *threaded_holes*

)

get the solutions for a set of concentric holes used as a fastener pilot hole
Parameters

**bearing_hole** ID of a surface at a hole. Only one surface per hole required
**threaded_holes** list of surfaces at threaded holes concentric to bearing hole. Only one surface is required

Returns
cubit solutions

◆ get_solutions_for_cavity_surface()

std::vector< std::vector< std::string > > get_solutions_for_cavity_surface (int *surface_id*)

Get the solution list for a given cavity surface.
Parameters

**surface_id** the surface being queries

Returns
Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Note: If using python, vectors will be python tuples.

◆ get_solutions_for_classified_surface()

std::vector< std::vector< std::string > > get_solutions_for_classified_surface (std::string *classification*, int *surf_id*)

Get lists of display, preview and command strings for a classified surface.

◆ get_solutions_for_classified_volume()

std::vector< std::vector< std::string > > get_solutions_for_classified_volume (std::string *classification*, int *vol_id*)

Get lists of display, preview and command strings for a classified volume.
Parameters

**classification** string defining the classification type: "bolt", "nut", "washer", "spring", "ball", "race", "pin", "gear", "insert", "other"
**vol_id**

Returns
Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Vector 4 will contain operation strings for machine

learning Note: If using this function in python, returned vectors will be python tuples.

◆ get_solutions_for_close_loop()

```
std::vector< std::vector< std::string > >
get_solutions_for_close_loop (int surface_id,
                              double mesh_size
                              )
```

Get the solution list for a given close loop surface.
Parameters

**surface_id** the surface being queried
**mesh_size** Indicate size for 'narrowness'

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Note: If using python, vectors will be python tuples.

◆ get_solutions_for_cone_surface()

```
std::vector< std::vector< std::string > >
get_solutions_for_cone_surface (int surface_id)
```

Get lists of display, preview and command strings for surfaces with defined as cones.
Parameters

**surface_id** cone surface

◆ get_solutions_for_decomposition()

```
std::vector< std::vector< std::string > >
get_solutions_for_decomposition (const std::vector< int > & volume_list,
                                 double exterior_angle,
                                 bool do_imprint_merge,
                                 bool tolerant_imprint
                                 )
```

Get the list of possible decompositions.
Parameters

**volume_list** List of volumes to query
**exterior_angle** Threshold value for the exterior angle
**do_imprint_merge** Set to true (1) if you want the imprint and merge to be done
**tol_imprint** Set to true (1) if you want to do a tolerant imprint

◆ get_solutions_for_forced_sweepability()

```
std::vector< std::vector< std::string > >
get_solutions_for_forced_sweepability (int volume_id,
                                       std::vector< int > & source_surface_id_list,
```

std::vector< int > & *target_surface_id_list*,
double *small_curve_size* = -1.0
)

**This function only works from C++** Get lists of display strings and command strings for forced sweepability solutions
Parameters

**volume_id** id of volume source_surface_id_list list of source surface ids target_surface_id_list list of target surface ids small_curve_size optional paramtere to specify small curve size

Returns

Vector of two string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

◆ get_solutions_for_imprint_merge()

std::vector< std::vector< std::string > >
get_solutions_for_imprint_merge (int *surface_id1*,
int *surface_id2*
)

Get lists of display strings and command strings for imprint/merge solutions.
Parameters

**surface_id1** overlapping surface 1 surface_id2 overlapping surface 2

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

◆ get_solutions_for_near_coincident_vertex_and_curve()

std::vector< std::vector< std::string > >
get_solutions_for_near_coincident_vertex_and_curve (int *vertex_id*,
int *curve_id*
)

Get lists of display strings and command strings for near coincident vertices and curves.
Parameters

**vertex_id** ID of the vertex
**curve_id** ID of the curve

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_near_coincident_vertex_and_surface()

```
std::vector< std::vector< std::string > >
get_solutions_for_near_coincident_vertex_and_surface (int vertex_id,
                                                      int surface_id
                                                     )
```

Get lists of display strings and command strings for near coincident vertices and surfaces.

Parameters

**vertex_id** ID of the vertex

**surface_id** ID of the surface

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_near_coincident_vertices()

```
std::vector< std::vector< std::string > >
get_solutions_for_near_coincident_vertices (int vertex_id_1,
                                            int vertex_id_2
                                           )
```

Get lists of display strings and command strings for near coincident vertices.

Parameters

**target_vertex_ids** **Vertex** list

**high_tolerance** The upper threshold tolerance value

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_overlapping_surfaces()

```
std::vector< std::vector< std::string > >
get_solutions_for_overlapping_surfaces (int surface_id_1,
                                        int surface_id_2
                                       )
```

Get lists of display strings and command strings for overlapping surfaces.
Parameters

**id** of surface 1
**id** of surface 2

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_overlapping_volumes()

```
std::vector< std::vector< std::string > >
get_solutions_for_overlapping_volumes (int    volume_id_1,
                                       int    volume_id_2,
                                       double maximum_gap_tolerance,
                                       double maximum_gap_angle
                                      )
```

Get lists of display strings and command strings for overlapping volumes.
Parameters

**id** of volume 1
**id** of volume 2

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_sharp_angle_vertex()

```
std::vector< std::vector< std::string > >
get_solutions_for_sharp_angle_vertex (int    vertex_id,
                                      double small_curve_size,
                                      double mesh_size
                                     )
```

Get lists of display, preview and command strings for sharp angle solutions.

Parameters

| | |
|---|---|
| **vertex_id** | vertex with sharp angle |
| **small_curve_size** | Threshold value used to determine what 'small' is |
| **mesh_size** | Element size of the model |

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Vector 4 will contain operation strings for machine learning Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_sheet_volume_connection()

```
std::vector< std::vector< std::string > >
get_solutions_for_sheet_volume_connection (std::vector< int > vol1_sheets,
                                           std::vector< int > vol2_sheets,
                                           double thickness1,
                                           double thickness2,
                                           std::string close_type = "",
                                           int close_id = 0
                                          )
```

Get lists of display, preview and command strings for two neighboring sheet volume sets. each set should be part of a common parent 3D volume.

Parameters

| | |
|---|---|
| **vol1_sheets** | volume IDs of sheet volumes with common solid parent |
| **vol2_sheets** | volume IDs of sheet volumes with common solid parent |
| **thickness1** | thickness of sheet volumes in vol_sheets1 |
| **thickness2** | thickness of sheet volumes in vol_sheets2 |
| **close_type** | optional limit solutions close to entity with type and ID |
| **close_id** | optional limit solutions close to entity with type and ID tolerance of near_location. Default will return all solutions |

### ◆ get_solutions_for_sheet_volumes()

```
std::vector< std::vector< std::string > >
get_solutions_for_sheet_volumes (std::vector< int > vol_ids,
                                 std::vector< double > thickness
                                )
```

Get lists of display, preview and command strings to connect sheet bodies.

### ◆ get_solutions_for_small_curves()

```
std::vector< std::vector< std::string > >
get_solutions_for_small_curves (int curve_id,
                                double small_curve_size,
```

```
                                                                    double mesh_size
                                                                  )
```

Get lists of display, preview and command strings for small curve solutions.
Parameters

| | |
|---|---|
| **curve_id** | Small curve |
| **small_curve_size** | Threshold value used to determine what 'small' is |
| **mesh_size** | Element size of the model |

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Vector 4 will contain operation strings for machine learning Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_small_surfaces()

```
std::vector< std::vector< std::string > >
get_solutions_for_small_surfaces                    (int    surface_id,
                                                     double small_curve_size,
                                                     double mesh_size
                                                    )
```

Get lists of display, preview and command strings for small surface solutions.
Parameters

| | |
|---|---|
| **surface_id** | Small surface |
| **small_curve_size** | Threshold value used to determine what 'small' is |
| **mesh_size** | Element size of the model |

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Vector 4 will contain operation strings for machine learning Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_source_target()

```
bool
get_solutions_for_source_t
arget                    (int                                  volume_id,
                          std::vector< std::vector< int >      feasible_source_surface_id
                          > &                                  _list,
                          std::vector< std::vector< int >      feasible_target_surface_id_l
                          > &                                  ist,
                          std::vector< std::vector< int >      infeasible_source_surface_i
                          > &                                  d_list,
```

```
std::vector< std::vector< int > infeasible_target_surface_id
> &                              _list
)
```

Get a list of suggested sources and target surface ids given a specified volume.

### ◆ get_solutions_for_surfaces_with_narrow_regions()

```
std::vector< std::vector< std::string > >
get_solutions_for_surfaces_with_narrow_regions (int surface_id,
                                                double small_curve_size,
                                                double mesh_size
                                               )
```

Get lists of display, preview and command strings for surfaces with narrow regions solutions.

Parameters

| | |
|---|---|
| **surface_id** | Small surface |
| **small_curve_size** | Threshold value used to determine what 'small' is |
| **mesh_size** | Element size of the model |

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_solutions_for_thin_volume()

```
std::vector< std::vector< std::string > >
get_solutions_for_thin_volume (int vol_id,
                               std::vector< int > near_vols,
                               bool include_weights = false,
                               bool include_type = false
                              )
```

Get lists of display, preview and command strings for a volume to reduce to shell.

Parameters

| | |
|---|---|
| **vol_id** | solutions will be returned for this volume |
| **near_vols** | limit influence to these volumes. If empty, will use all |
| **include_weights** | include the heuristic weights with the operation string (1=best, 0=worst) |
| **include_type** | include the connection type in the operation string long_long = 4, continuous = 3, midsurface = 2, copy = 1 |

### ◆ get_solutions_for_volumes()

```
std::vector< std::vector< std::string > >
get_solutions_for_volumes (int vol_id,
```

double *small_curve_size*,
double *mesh_size*
)

Get lists of display, preview and command strings for small volume solutions.
Parameters

**vol_id**
**small_curve_size** Threshold value used to determine what 'small' is
**mesh_size** Element size of the model

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. Vector 3 will contain Cubit preview strings. Vector 4 will contain operation strings for machine learning Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_source_surfaces()

std::vector< int > get_source_surfaces ( int *volume_id* )

Get a list of a volume's sweep source surfaces.
Parameters

**volume_id** Specifies the volume id

Returns

List (python tuple) of surface ids

### ◆ get_sphere_count()

int get_sphere_count ( )

Get the count of sphere elements in the model.
Returns

The number of spheres in the model

### ◆ get_sphere_global_element_id()

int get_sphere_global_element_id ( int *edge_id* )

Given a sphere id, return the global element id.

```
int gid = CubitInterface::get_sphere_global_element_id (22);
```

CubitInterface::get_sphere_global_element_id
int get_sphere_global_element_id(int edge_id)
Given a sphere id, return the global element id.

Parameters

**sphere_id** Specifies the id of the sphere

Returns

The corresponding element id

### ◆ get_string_sculpt_default()

std::string get_string_sculpt_default(const char * *variable*)

◆ get_sub_elements()

std::vector< int > get_sub_elements(const std::string & *entity_type*,
int *entity_id*,
int *dimension*
)

Get the lower dimesion entities associated with a higher dimension entities. For example get the faces associated with a hex or the edges associated with a tri.

```
std::vector<int> face_id_list;
face_id_list = CubitInterface::get_sub_elements ("hex" , 221, 2);
```

CubitInterface::get_sub_elements

std::vector< int > get_sub_elements(const std::string &entity_type, int entity_id, int dimension)

Get the lower dimesion entities associated with a higher dimension entities. For example get the face...

```
face_id_list = cubit.get_sub_elements("hex" , 221, 2)
```

Parameters

**entity_type** The mesh element type of the higher dimension entity
**entity_id** The mesh element id
**dimension** The dimension of the desired sub entities

Returns

List (python tuple) of ids of the desired dimension

◆ get_submap_corner_types()

std::vector< std::pair< int, int > > get_submap_corner_types(int *surface_id*)

Get a list of vertex ids and the corresponding corner vertex types if the surface were defined as submap surface. There are no side affects. This does not actually assign corner types or change the underlying mesh scheme of the surface.

Parameters

**the**id of the surface

Returns

a vector of pairs of <id, corner_type> The corner_types are defined as follows UNSET_TYPE = -1, END_TYPE = 1, SIDE_TYPE, CORNER_TYPE, REVERSAL_TYPE, TRIANGLE_TYPE, NON_TRIANGLE_TYPE };

◆ get_surface_area()

double get_surface_area(int *surface_id*)

Get the area of a surface.

Parameters

**surface_id** ID of the surface

Returns

Area of the surface

### ◆ get_surface_cavity_collections()

```
std::vector< std::vector< int > >
get_surface_cavity_collections (const std::vector< int > & volume_list,
                                double hr_threshold,
                                double area_threshold,
                                std::vector< double > & return_cavity_hrs,
                                std::vector< double > & return_cavity_areas
                                )
```

Returns the collections of surfaces that comprise holes or cavities in the specified volumes. Filter by hydarulic radius and area of the cavity.
Parameters

| | |
|---|---|
| **volume_list** | List of volumes to query |
| **hr_threshold** | return cavities with computed hydraulic radius less than hr_threshold. Where hr = 4*Area/Perimeter. Use hr_threshold < 0.0 to return all cavities |
| **area_threshold** | return cavities with computed surface area less than area_threshold. Use area_threshold < 0.0 to return all cavities |
| **return** | a vector of cavity areas corresponding to the return cavity id lists |

Returns

A list of lists of surface id's grouped by their individual cavity or hole

### ◆ get_surface_centroid()

```
std::array< double, 3 > get_surface_centroid (int surface_id)
```

Get the surface centroid for a specified surface.
Parameters

| | |
|---|---|
| **surface_id** | ID of the surface |

Returns

surface centroid

### ◆ get_surface_count()

```
int get_surface_count ( )
```

Get the current number of surfaces.
Returns

The number of surfaces in the current model, if any

### ◆ get_surface_element_count()

```
int get_surface_element_count (int surface_id)
```

Get the count of elements in a surface.
Returns

The number of quads, and triangles in a surface. NOTE: This count does not distinguish between elements which have been put into a block or not.

### ◆ get_surface_hole_collections()

std::vector< std::pair< std::vector< int >, double > > get_surface_hole_collections (const std::vector< int > & *volume_list*, double *radius_threshold* )

Returns the collections of surfaces that comprise holes in the specified volumes. Filter by radius of the hole.

Parameters

**volume_list** List of volumes to query

**radius_threshold** return holes with computed radius less than or equal to radius_threshold.

**return** a vector of hole radii corresponding to the return hole id lists

Returns

A list of lists of surface id's grouped by their individual hole

### ◆ get_surface_loop_nodes()

std::vector< std::vector< int > > get_surface_loop_nodes (int *surface_id*)

get the ordered list of nodes on the loops of this surface

Parameters

**surface_id** User specified id of the desired surface

Returns

A list of lists (python tuple of tuples) one list per loop first loop is the external

### ◆ get_surface_nodes()

std::vector< int > get_surface_nodes (int *surface_id*)

Get list of node ids owned by a surface. Excludes nodes owned by bounding curves and verts.

```
int surf_id = 5;
vector<int>   surface_nodes   =   CubitInterface::get_surface_nodes
(surface_id);
```

CubitInterface::get_surface_nodes

std::vector< int > get_surface_nodes(int surface_id)

Get list of node ids owned by a surface. Excludes nodes owned by bounding curves and verts.

Parameters

**surf_id** id of surface

Returns

List (python tuple) of IDs of nodes owned by the surface

### ◆ get_surface_normal()

std::array< double, 3 > get_surface_normal(int *surface_id*)

Get the surface normal for a specified surface.
Parameters

**surface_id** ID of the surface

Returns

surface normal at the center

◆ get_surface_normal_at_coord()

std::array< double, 3 > get_surface_normal_at_coord(int *surface_id*, std::array< double, 3 > )

Get the surface normal for a specified surface at a location.
Parameters

**surface_id** ID of the surface

**coord** array of x,y,z location on surface

Returns

surface normal at coord

◆ get_surface_num_loops()

int get_surface_num_loops(int *surface_id*)

get the number of loops on the surface
Parameters

**surface_id** User specified id of the desired surface

Returns

number of loops on the surface

◆ get_surface_principal_curvatures()

std::vector< double > get_surface_principal_curvatures(int *surface_id*)

Get the principal curvatures of a surface at surface mid_point.
Parameters

**surface_id** ID of the surface

Returns

two scalars that are the principal curvatures at midpoint

◆ get_surface_quads()

std::vector< int > get_surface_quads(int *surface_id*)

get the list of any quad elements on a given surface
Parameters

**surface_id** User specified id of the desired surface

Returns
A list (python tuple) of the quad ids on the surface

◆ get_surface_sense()

std::string get_surface_sense(int *surface_id*)

Get the surface sense for a specified surface.
Parameters

**surface_id** ID of the surface

Returns
surface sense as "Reversed" or "Forward" or "Both"

◆ get_surface_tris()

std::vector< int > get_surface_tris(int *surface_id*)

get the list of any tri elements on a given surface
Parameters

**surface_id** User specified id of the desired surface

Returns
A list (python tuple) of the tri ids on the surface

◆ get_surface_type()

std::string get_surface_type(int *surface_id*)

Get the surface type for a specified surface.
Parameters

**surface_id** ID of the surface

Returns
Type of surface

◆ get_surfs_with_narrow_regions()

std::vector< int > get_surfs_with_narrow_regions(std::vector< int > *target_ids*,
double *narrow_size*
)

Get the list of surfaces with narrow regions.
Parameters

**target_volume_ids** List of volume ids to examine.
**narrow_size** Indicate the size that defines 'narrowness'

Returns
List (python tuple) of surface ids

◆ get_tangential_intersections()

std::vector< int > get_tangential_intersections(std::vector< int > *target_volume_ids*,
double *upper_bound*,

double *lower_bound*
)

Get the list of bad tangential intersections for a list of volumes.
'Bad' is a function of the upper_bound and lower_bound threshold parameters. The id of surfaces are returned when any tangential angle associated with a surface is less than the lower_bound or greater than the upper_bound.
Parameters

**target_volume_ids** List of volume ids to examine.
**upper_bound** Upper threshold angle
**lower_bound** Lower threshold angle

Returns
List (python tuple) of surface ids associated with bad tangential angles

◆ get_target_surfaces()

std::vector< int > get_target_surfaces ( int *volume_id* )

Get a list of a volume's sweep target surfaces.
Parameters

**volume_id** Specifies the volume id

Returns
List (python tuple) of surface ids

◆ get_target_timestep()

double get_target_timestep ( )

Returns the target timestep threshold used in the timestep density multiplier metric.

◆ get_tet_count()

int get_tet_count ( )

Get the count of tets in the model.
Returns
The number of tets in the model

◆ get_tet_global_element_id()

int get_tet_global_element_id ( int *tet_id* )

Given a tet id, return the global element id.

```
int gid = CubitInterface::get_tet_global_element_id (22);
```

CubitInterface::get_tet_global_element_id
int get_tet_global_element_id(int tet_id)
Given a tet id, return the global element id.

Parameters

**tet_id** Specifies the id of the tet

Returns

The corresponding element id

### ◆ get_tetmesh_growth_factor()

double get_tetmesh_growth_factor ( int *volume_id* )

Get the tetmesh growth factor.
Returns
the volume growth factor

### ◆ get_tetmesh_insert_mid_nodes()

bool get_tetmesh_insert_mid_nodes ( )

Get the state of the flag to insert midnodes during meshing. Global setting.
Returns
boolean - true if insert midnodes during meshing

### ◆ get_tetmesh_minimize_interior_points()

bool get_tetmesh_minimize_interior_points ( )

Get the state of the flag to minimize interior points in tetmesher. Global setting.
Returns
boolean - true if minimizing interior points during meshing

### ◆ get_tetmesh_minimize_slivers()

bool get_tetmesh_minimize_slivers ( )

Get the state of the flag to minimize sliver tets. Global setting.
Returns
boolean - true if minimizing sliver tets during meshing

### ◆ get_tetmesh_num_anisotropic_layers()

int get_tetmesh_num_anisotropic_layers ( )

Get the number of anisotropic tet layers. Global setting.
Returns
number of anisotropic layers (0 if not using anisotropy)

### ◆ get_tetmesh_optimization_level()

int get_tetmesh_optimization_level ( )

Get the optimization level for tetmeshing. Global setting.
Returns
integer from 1 to 6

### ◆ get_tetmesh_optimize_mid_nodes()

bool get_tetmesh_optimize_mid_nodes ( )

Get the state of the flag to optimize midnodes during meshing. Global setting.
Returns
boolean - true if optimize midnodes during meshing

- get_tetmesh_optimize_overconstrained_edges()

  bool get_tetmesh_optimize_overconstrained_edges ( )

Get the state of the flag to optimize overconstrained edges. Global setting.
Returns
boolean - true if optimizing overconstrained edges during meshing

- get_tetmesh_optimize_overconstrained_tets()

  bool get_tetmesh_optimize_overconstrained_tets ( )

Get the state of the flag to optimize overconstrained tets. Global setting.
Returns
boolean - true if optimizing overconstrained tets during meshing

- get_tetmesh_parallel()

  bool get_tetmesh_parallel ( )

Get the parallel flag for tet meshing. Defines whether to use parallel mesher.
Returns
boolean value as to whether or not the parallel tet mesher is used

- get_tetmesh_proximity_flag()

  bool get_tetmesh_proximity_flag ( int *volume_id* )

Get the proximity flag for tet meshing.
Parameters

**volume_id** the volume id

Returns
boolean value as to whether or not the proximity flag is set

- get_tetmesh_proximity_layers()

  int get_tetmesh_proximity_layers ( int *volume_id* )

Get the number of proximity layers for tet meshing. This is the number of layers between close surfaces.
Parameters

**volume_id** the volume id

Returns
boolean value as to whether or not the proximity flag is set

- get_tetmesh_relax_surface_constraints()

  bool get_tetmesh_relax_surface_constraints ( )

Get the state of the flag to relax surface mesh constraints in tetmesher. Global setting.
Returns
boolean - true if relaxing surface mesh constraints during meshing

- get_tight_bounding_box()

std::array< double, 15 > get_tight_bounding_box ( const std::string & *geometry_type*,
std::vector< int > *entity_list*
)

Get the tight bounding box for a list of entities.

```
      std::array<double> vector_list;
      vector_list = CubitInterface::get_tight_bounding_box ("surface" ,
entity_list);
```

CubitInterface::get_tight_bounding_box
std::array< double, 15 > get_tight_bounding_box(const std::string &geometry_type, std::vector< int > entity_list)
Get the tight bounding box for a list of entities.

```
      vector_list = cubit.get_tight_bounding_box("surface" , entity_list)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_list** List of ids associated with geom_type

Returns

A vector (python tuple) of coordinates and axis (0-2) center (3-5, 6-8, 9-11) u, v, x normalized coordinate axis of the box (12-14) length in u, v, w

## ◆ get_top_level_assembly_items()

std::vector< **AssemblyItem** > get_top_level_assembly_items ( )

## ◆ get_total_bounding_box()

std::array< double, 10 > get_total_bounding_box ( const std::string & *geometry_type*,
std::vector< int > *entity_list*
)

Get the bounding box for a list of entities.

```
      std::array<double,10> vector_list;
      vector_list = CubitInterface::get_total_bounding_box ("surface" ,
entity_list);
```

CubitInterface::get_total_bounding_box
std::array< double, 10 > get_total_bounding_box(const std::string &geometry_type, std::vector< int > entity_list)
Get the bounding box for a list of entities.

```
      vector_list = cubit.get_total_bounding_box("surface" , entity_list)
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_list** List of ids associated with geom_type

Returns

An array of coordinates for the entity's bounding box. Ten (10) values will be returned. [ x-min, x-max, x-range, y-min, y-max, y-range, z-min, z-max, z-range, diagonal].

## ◆ get_total_volume()

double get_total_volume(std::vector< int > *volume_list*)

Get the total volume for a list of volume ids.
Parameters

**volume_list** List of volume ids

Returns

The total volume of all volumes indicated in the id list

### ◆ get_tri_count()

int get_tri_count( )

Get the count of tris in the model.
Returns

The number of tris in the model

### ◆ get_tri_global_element_id()

int get_tri_global_element_id(int *tri_id*)

Given a tri id, return the global element id.

```
int gid = CubitInterface::get_tri_global_element_id (22);
```

CubitInterface::get_tri_global_element_id
int get_tri_global_element_id(int tri_id)
Given a tri id, return the global element id.

Parameters

**tri_id** Specifies the id of the tri

Returns

The corresponding element id

### ◆ get_trimesh_geometry_sizing()

bool get_trimesh_geometry_sizing( )

Get the global geometry sizing flag for trimesher.
Returns

boolean - true if geometry sizing is on

### ◆ get_trimesh_num_anisotropic_layers()

int get_trimesh_num_anisotropic_layers( )

Get the global number of anisotropic layers for trimeshing.
Returns

number of anistropic tri layers (0 if not using anisotropy)

### ◆ get_trimesh_ridge_angle()

double get_trimesh_ridge_angle( )

Get the global setting for ridge angle in trimesher.
Returns

ridge angle

### ◆ get_trimesh_split_overconstrained_edges()

bool get_trimesh_split_overconstrained_edges ( )

Get the global setting for trimesher split over-constrained edges.
Returns
boolen - true if trimesher split over-constrained edges setting is on

### ◆ get_trimesh_surface_gradation()

double get_trimesh_surface_gradation ( )

Get the global surface mesh gradation set for meshing with MeshGems.
Returns
the surface gradation

### ◆ get_trimesh_surface_proximity_ratio()

double get_trimesh_surface_proximity_ratio ( )

Get the global trimesh surface proximity max aspect ratio setting with MeshGems.
Returns
the surface proximity max aspect ratio

### ◆ get_trimesh_target_min_size()

double get_trimesh_target_min_size ( std::string *geom_type*,
int *entity_id*
)

Get the trimesh target min size for the entity. local setting for surfaces.
Returns
the target min size for entity ID

### ◆ get_trimesh_tiny_edge_length()

double get_trimesh_tiny_edge_length ( )

Get the global setting for tiny edge length in trimesher.
Returns
tiny edge length

### ◆ get_trimesh_use_surface_proximity()

bool get_trimesh_use_surface_proximity ( )

Get the global trimesh surface proximity setting with MeshGems.
Returns
the surface proximity state

### ◆ get_trimesh_volume_gradation()

double get_trimesh_volume_gradation ( )

Get the global volume mesh gradation set for meshing with MeshGems.
Returns

the volume gradation

### ◆ get_undo_enabled()

bool get_undo_enabled ( )

Query whether undo is currently enabled.
Returns
True if undo is enabled, otherwise false

### ◆ get_unmerged_curves_on_shells()

std::vector< int > get_unmerged_curves_on_shells ( std::vector< int > *shell_vols*,
std::vector< double > *thickness*
)

return a list of curve IDs on the given shell volumes that are in proximity to one of the other shell volumes in the list
Parameters

**shell_vols** list of volum IDs to check. Ignores any volumes that are not sheet volumes

**thickness** list of thicknesses corresponding to volume IDs. Normally this is the designated thickness of each of the shell volumes. Length of thickness should be the same as shell_vols

Returns
list of curve IDs that meet the criteria

### ◆ get_valence()

int get_valence ( int *vertex_id* )

Get the valence for a specific vertex.
Parameters

**vertex_id** ID of vertex

### ◆ get_valid_block_element_types()

std::vector< std::string > get_valid_block_element_types ( int *block_id* )

Get a list of potential element types for a block.
Parameters

**block_id** The block id

Returns
List (python tuple) of potential element types

### ◆ get_velocity_combine_type()

std::string get_velocity_combine_type ( int *entity_id* )

Get the velocity's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine".
Parameters

**entity_id** Id of the velocity

Returns

The combine type for the given velocity

### ◆ get_velocity_dof_signs()

const int * get_velocity_dof_signs(int *entity_id*)

**This function only available from C++** Get the velocity's dof signs

Parameters

**entity_id** Id of the velocity

Returns

### ◆ get_velocity_dof_values()

const double * get_velocity_dof_values(int *entity_id*)

**This function only available from C++** Get the velocity's dof values

Parameters

**entity_id** Id of the velocity

Returns

### ◆ get_version()

std::string get_version( )

Get the Cubit version.

Returns

A string containing the current version of Cubit

### ◆ get_vertex_count()

int get_vertex_count( )

Get the current number of vertices.

Returns

The number of vertices in the current model, if any

### ◆ get_vertex_node()

int get_vertex_node(int *vertex_id*)

Get the node owned by a vertex.

```
int vert_id = 22;
int node_id = CubitInterface::get_vertex_node (vert_id);
```

CubitInterface::get_vertex_node

int get_vertex_node(int vertex_id)

Get the node owned by a vertex.

Parameters

**vert_id** id of vertex

Returns

ID of node owned by the vertex. returns -1 of doesn't exist

### ◆ get_vertex_type()

```
std::string get_vertex_type(int surface_id,
                            int vertex_id
                           )
```

Get the **Vertex** Types for a specified vertex on a specified surface. **Vertex** types include "side", "end", "reverse", "unknown".

Parameters

**surface_id** Id of the surface associated with the vertex

**vertex_id** Id of the vertex

Returns

The type – "side", "end", "reverse", or "unknown"

### ◆ get_view_at()

```
std::array< double, 3 > get_view_at( )
```

Get the camera 'at' point.

Returns

The xyz coordinates of the camera's current position

### ◆ get_view_distance()

```
double get_view_distance( )
```

Get the distance from the camera to the model (from - at)

Returns

Distance from the camera to the model

### ◆ get_view_from()

```
std::array< double, 3 > get_view_from( )
```

Get the camera 'from' point.

Returns

The xyz coordinates of the camera's from position

### ◆ get_view_up()

```
std::array< double, 3 > get_view_up( )
```

Get the camera 'up' direction.

Returns

The xyz coordinates of the camera's up direction

### ◆ get_vol_sphere_params()

```
bool get_vol_sphere_params(std::vector< int > sphere_id_list,
                           int &              rad_intervals,
                           int &              az_intervals,
                           double &           bias,
                           double &           fract,
```

int & *max_smooth_iterations*
)

get the current sphere parameters for a sphere volume
Parameters

| | |
|---|---|
| **sphere_id_list** | list of volume ids (should be spheres) |
| **rad_intervals** | number of radial intervals (around circle) |
| **az_intervals** | number of intervals from inner mapped box to surface |
| **bias** | bias from inner mapped box to surface (<1 increases size to boundary) |
| **fract** | fraction of radius to use as size of interior mapped box |
| **max_smooth_iterations** | max number of smooth iterations to perform after meshing |

## ◆ get_volume_area()

double get_volume_area(int *volume_id*)

Get the area of a volume.
Parameters

**volume_id** ID of the volume

Returns

Area of the volume

## ◆ get_volume_count()

int get_volume_count( )

Get the current number of nodesets.
Returns

The number of nodesets in the current model, if any

## ◆ get_volume_element_count()

int get_volume_element_count(int *volume_id*)

Get the count of elements in a volume.
Returns

The number of hexes, tets, pyramids, and wedges in a volume. NOTE: This count does not distinguish between elements which have been put into a block or not.

## ◆ get_volume_gap_solutions()

std::vector< std::vector< std::string > > get_volume_gap_solutions(int *surface_id_1*,
int *surface_id_2*
)

Get lists of display strings and command strings for gaps
Parameters

**id** of surface 1

**id** of surface 2

Returns

Vector of three string vectors. Vector 1 will contain display strings to be shown to users. Vector 2 will contain Cubit command strings. This second set of strings may contain concatenated strings delimited by ';'. In other words, one instance of command string may in fact contain multiple commands separated by the ';' sequence. Vector 3 will contain Cubit preview strings. Note: If using this function in python, returned vectors will be python tuples.

### ◆ get_volume_gaps()

```
void get_volume_gaps(std::vector< int >          target_volume_ids,
                     std::vector< int > &        returned_surface_list_1,
                     std::vector< int > &        returned_surface_list_2,
                     std::vector< double > &     returned_distance_list,
                     std::vector< double > &     returned_overlap_area_list,
                     double                      maximum_gap_tolerance,
                     double                      maximum_gap_angle,
                     int                         cache_overlaps = 0
                    )
```

**This function only works from C++** Get the list of gaps for a list of volumes
For every occurance of a gap, two surfaces ids are returned. Those ids are returned in the indicated lists and are aligned. In other words the first id in surf_list_1 overlaps with the first id in surf_list_2. The second id in surf_list_1 overlaps with the second id in surf_list-2, and so on.
Parameters

| | |
|---|---|
| **target_volume_ids** | List of volume ids to examine. |
| **surf_list_1** | User specified list where the ids of the gap surfaces will be returned |
| **surf_list_2** | User specified list where the ids of the gap surfaces will be returned |
| **distance_list** | User specified list where the distance between the gap surface will be returned |
| **max_gap_tolerance** | User specified tolerance used to find the gaps. |
| **cache_overlaps** | speed up overlaps by caching and using previously computed results. Default 0 = no caching. 1 = clear out old values first. 2 = use and add to existing cache |

### ◆ get_volume_hexes()

```
std::vector< int > get_volume_hexes(int volume_id)
```

get the list of any hex elements in a given volume
Parameters

**volume_id** User specified id of the desired volume

Returns

A list (python tuple) of the hex ids in the volume

◆ get_volume_nodes()

std::vector< int > get_volume_nodes ( int *volume_id* )

Get list of node ids owned by a volume. Excludes nodes owned by bounding surfs, curves and verts.

```
int vol_id = 1;
vector<int>    volume_nodes    =    CubitInterface::get_volume_nodes
(vol_id);
```

CubitInterface::get_volume_nodes

std::vector< int > get_volume_nodes(int volume_id)

Get list of node ids owned by a volume. Excludes nodes owned by bounding surfs, curves and verts.

Parameters

**vol_id** id of volume

Returns

List (python tuple) of IDs of nodes owned by the volume

◆ get_volume_pyramids()

std::vector< int > get_volume_pyramids ( int *volume_id* )

get the list of any pyramid elements in a given volume

Parameters

**volume_id** User specified id of the desired volume

Returns

A list (python tuple) of the pyramid ids in the volume

◆ get_volume_tets()

std::vector< int > get_volume_tets ( int *volume_id* )

get the list of any tet elements in a given volume

Parameters

**volume_id** User specified id of the desired volume

Returns

A list (python tuple) of the tet ids in the volume

◆ get_volume_volume()

double get_volume_volume ( int *vol_id* )

Get the volume of a volume.

Parameters

**volume_id** ID of the volume

Returns

volume

### ◆ get_volume_wedges()

std::vector< int > get_volume_wedges ( int *volume_id* )

get the list of any wedge elements in a given volume
Parameters

**volume_id** User specified id of the desired volume

Returns

A list (python tuple) of the wedge ids in the volume

### ◆ get_volumes_for_node()

std::vector< int > get_volumes_for_node ( std::string *node_name*,
int *node_instance*
)

### ◆ get_wedge_count()

int get_wedge_count ( )

Get the count of wedge elements in the model.
Returns

The number of wedges in the model

### ◆ get_wedge_global_element_id()

int get_wedge_global_element_id ( int *wedge_id* )

Given a wedge id, return the global element id.

```
int gid = CubitInterface::get_wedge_global_element_id (22);
```

CubitInterface::get_wedge_global_element_id
int get_wedge_global_element_id(int wedge_id)
Given a wedge id, return the global element id.

Parameters

**wedge_id** Specifies the id of the wedge

Returns

The corresponding element id

### ◆ get_wrt_entity()

std::string get_wrt_entity ( std::string *source_type*,
int *source_id*,
int *sideset_id*
)

Get the with-respect-to entity.

```
std::string wrt_entity;
wrt_entity = CubitInterface::get_wrt_entity ("face" , 332, 2);
```

CubitInterface::get_wrt_entity

std::string get_wrt_entity(std::string source_type, int source_id, int sideset_id)
Get the with-respect-to entity.

```
wrt_entity = cubit.get_wrt_entity("face" , 332, 2)
```

Parameters

**source_type** Item type - could be 'face', 'quad' or 'tri'
**source_id** ID of entity
**sideset_id** ID of the sideset

Returns

'with-respect-to' entity of the source_type/source_id in specified sideset

◆ group_list()

```
void group_list ( std::vector< std::string > & name_list,
                  std::vector< int > &         returned_id_list
                )
```

Get the names and ids of all the groups (excluding the pick group) that are defined by the current cubit session.
Parameters

**name_list** User specified list where the active group names will be returned
**id_list** User specified list where the ids of all active groups will be returned

◆ group_names_ids()

```
std::vector< std::pair< std::string, int > > group_names_ids ( )
```

Get the names and ids of all the groups returned in a name/id structure that are defined by the current cubit session.
return A list of std::pair<std::string, int> structure instances

◆ has_valid_size()

```
int has_valid_size ( const std::string & geometry_type,
                     int                 entity_id
                   )
```

Get whether an entity has a size. All entities have a size unless the auto sizing is off. If the auto sizing is off, an entity has a size only if it has been set.

◆ heatflux_is_on_shell_area()

```
bool heatflux_is_on_shell_area ( CI_BCEntityTypes bc_area_enum,
                                 int              entity_id
                               )
```

Determine whether a BC heatflux is on a shell area.
Parameters

**bc_area** enum of CI_BCEntityTypes. Use 7 to check if on top, 8 to check if on bottom

**entity_id** Id of the BC

Returns

true if BC heatflux is on specified shell area, otherwise false

◆ highlight()

void highlight ( const std::string & *entity_type*,
int *entity_id*
)

Highlight the given entity.

◆ init()

void init ( const std::vector< std::string > & *argv* )

Use init to initialize Cubit. Using a blank list as the input parameter is acceptable.

Parameters

**argv** List of start-up directives. A blank list such as [''] will suffice. See Cubit Help for details

◆ is_acis_engine_available()

bool is_acis_engine_available ( )

◆ is_assembly_metadata_attached()

bool is_assembly_metadata_attached ( int *volume_id* )

Determine whether metadata is attached to a specified volume.

Parameters

**volume_id** ID of the volume

Returns

True if metadata exists, otherwise false

◆ is_blend_surface()

bool is_blend_surface ( int *surface_id* )

return whether the surface is a blend

Parameters

**surface_id** ID ofsurface

Returns

whether the surface is a blend

◆ is_boundary_layer_id_available()

bool is_boundary_layer_id_available ( int *boundary_layer_id* )

◆ is_catia_engine_available()

bool is_catia_engine_available ( )

Determine whether catia engine is available.
Returns
True if catia engine is available, otherwise false

### ◆ is_cavity_surface()

bool is_cavity_surface(int *surface_id*)

return whether the surface is part of a cavity
Parameters

**surface_id** ID ofsurface
Returns
whether the surface is part of a cavity

### ◆ is_chamfer_surface()

bool is_chamfer_surface(int *surface_id*,
double *thickness_threshold*
)

return whether the surface is a chamfer
Parameters

**surface_id** ID ofsurface
**thickness_threshold** max thickness criteria for chamfer
Returns
whether the surface is a chamfer (if < 0 then 3*mesh_size)

### ◆ is_close_loop_surface()

bool is_close_loop_surface(int *surface_id*,
double *mesh_size*
)

return whether the has one or more close loops
Parameters

**surface_id** ID ofsurface
**mesh_size** Indicate the mesh size used as the threshold
Returns
whether the surface has one or more close loops

### ◆ is_command_echoed()

bool is_command_echoed( )

Check the echo flag in cubit.
Returns
A boolean indicating whether commands should be echoed in Cubit

### ◆ is_command_journaled()

bool is_command_journaled ( )

Check the journaling flag in cubit.
Returns
A boolean indicating whether commands are journaled by Cubit

◆ is_cone_surface()

bool is_cone_surface ( int *surface_id* )

return whether the surface is a cone
Parameters

**surface_id** ID ofsurface
Returns
whether the surface is a cone

◆ is_continuous_surface()

bool is_continuous_surface ( int *surface_id*,
double *angle_tol*
)

return whether the surface has any adjacent surfaces that are continuous (exterior angle is 180 degrees +- angle_tol)
Parameters

**surface_id** ID ofsurface
**angle_tol** angle tolerance for continuity
Returns
whether the surface has adjacent continuous surfaces

◆ is_cylinder_surface()

bool is_cylinder_surface ( int *surface_id* )

return whether the surface is a cylinder
Parameters

**surface_id** ID ofsurface
Returns
whether the surface is a cylinder

◆ is_geometry_visibility_on()

bool is_geometry_visibility_on ( )

Get the current geometry visibility setting.
Returns
True if scale is visible, otherwise false

◆ is_hole_surface()

bool is_hole_surface ( int *surface_id*,

double *radius_threshold*
)

return whether the surface is part of a hole
Parameters

**surface_id** ID of surface
**radius_threshold** max radius criteria for hole (if < 0, then default is 3*mesh_size)

Returns

whether the surface is part of a hole

◆ is_interval_count_odd()

bool is_interval_count_odd(int *surface_id*)

Query whether a specified surface has an odd loop.
Parameters

**surface_id** Id of the surface

Returns

True if surface is/contains an odd looop, otherwise false.

◆ is_merged()

bool is_merged(const std::string & *geometry_type*,
int *entity_id*
)

Determines whether a specified entity is merged.

```
if (CubitInterface::is_merged ("surface" , 137)) . . .
```

CubitInterface::is_merged
bool is_merged(const std::string &geometry_type, int entity_id)
Determines whether a specified entity is merged.

```
if cubit.is_merged("surface" , 137):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

◆ is_mesh_element_in_group()

bool is_mesh_element_in_group(const std::string & *element_type*,
int *element_id*
)

Indicates whether a mesh element is in a group.

```
if (CubitInterface::is_mesh_element_in_group ("tet" , 445)) ...
```

CubitInterface::is_mesh_element_in_group
bool is_mesh_element_in_group(const std::string &element_type, int element_id)
Indicates whether a mesh element is in a group.

```
if cubit.is_mesh_element_in_group("tet" , 445):
```

Parameters

**element_type** Mesh type of the element
**element_id** ID of the mesh element return True if in a group, otherwise false

### ◆ is_mesh_visibility_on()

bool is_mesh_visibility_on ( )

Get the current mesh visibility setting.
Returns
True if scale is visible, otherwise false

### ◆ is_meshed()

bool is_meshed ( const std::string & *geometry_type*,
int *entity_id*
)

Determines whether a specified entity is meshed.

```
if (CubitInterface::is_meshed ("surface" , 137)) . . .
```

CubitInterface::is_meshed
bool is_meshed(const std::string &geometry_type, int entity_id)
Determines whether a specified entity is meshed.

```
if cubit.is_meshed("surface" , 137):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

### ◆ is_modified()

bool is_modified ( )

Get the modified status of the model.
Returns
A boolean indicating whether the model has been modified

### ◆ is_multi_volume()

bool is_multi_volume ( int *body_id* )

Query whether a specified body is a multi volume body.
Parameters

**body_id** Id of the body
Returns
True if body contains multiple volumes, otherwise false.

### ◆ is_narrow_surface()

bool is_narrow_surface ( int *surface_id*,
double *mesh_size*

)

return whether the surface is narrow (has a width smaller than mesh_size)
Parameters

**surface_id** ID ofsurface
**mesh_size** threshold used to determine if is narrow

Returns
whether the surface is narrow

◆ is_occlusion_on()

bool is_occlusion_on ( )

Get the current occlusion mode.
Returns
True if occlusion is on, otherwise false

◆ is_on_thin_shell()

bool is_on_thin_shell ( CI_BCTypes *bc_type_enum*,
int *entity_id*
)

Determine whether a BC is on a thin shell. Valid for temperature, convection and heatflux.
Parameters

**bc_type_in** enum of CI_BCTypes. temperature = 4, convection = 7, heatflux = 8
**entity_id** Id of the BC

Returns
true if BC is on thin shell element, otherwise false

◆ is_opencascade_engine_available()

bool is_opencascade_engine_available ( )

◆ is_part_of_list()

bool is_part_of_list ( int *target_id*,
std::vector< int > *id_list*
)

Routine to check for the presence of an id in a list of ids.
Parameters

**target_id** Target id
**id_list** List of ids

Returns
True if target_id is member of id_list, otherwise false

◆ is_performing_undo()

bool is_performing_undo ( )

Check if an undo command is currently being performed.
Returns
True or false.

◆ is_periodic()

bool is_periodic ( const std::string & *geometry_type*,
int *entity_id*
)

Query whether a specified surface or curve is periodic.

```
if (CubitInterface::is_periodic ("surface" , 22)) . . .
```

CubitInterface::is_periodic
bool is_periodic(const std::string &geometry_type, int entity_id)
Query whether a specified surface or curve is periodic.

```
if cubit.is_periodic("surface" , 22):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

Returns
True is entity is periodic, otherwise false

◆ is_perspective_on()

bool is_perspective_on ( )

Get the current perspective mode.
Returns
True if perspective is on, otherwise false

◆ is_playback_paused()

bool is_playback_paused ( )

◆ is_playback_paused_on_error()

bool is_playback_paused_on_error ( )

Gets whether or not playback is paused when an error occurs.
Returns
True if playback should be paused when an error occurs.

◆ is_point_contained()

int is_point_contained ( const std::string & *geometry_type*,
int *entity_id*,
const std::array< double, 3 > & *xyz_point*
)

Determine if given point is inside, outside, on or unknown the given entity. note that this is typically used for volumes or sheet bodies.
Parameters

**geom_type** string defining geometry type (volume or body) id ID of the geometric entity point xyz triplet defining the point (note that it must be std::array<double,3>

Returns

-1 failure, 0 outside, 1, inside, 2 on

### ◆ is_scale_visibility_on()

bool is_scale_visibility_on ( )

Get the current scale visibility setting.
Returns

True if scale is visible, otherwise false

### ◆ is_select_partial_on()

bool is_select_partial_on ( )

Get the current select partial setting.
Returns

True if partial select is on, otherwise false

### ◆ is_sheet_body()

bool is_sheet_body ( int *volume_id* )

Query whether a specified volume is a sheet body.
Parameters

**volume_id** Id of the volume

Returns

True if volume is a sheet body, otherwise false

### ◆ is_surface_meshable()

bool is_surface_meshable ( int *surface_id* )

Check if surface is meshable with current scheme.
Returns

A boolean indicating whether surface is meshable with current scheme

### ◆ is_surface_planar()

bool is_surface_planar ( int *surface_id* )

### ◆ is_surface_planer()

bool is_surface_planer ( int *surface_id* )

Query whether a specified surface is planer.

```
if (CubitInterface::is_surface_planar (22)) . . .
```

CubitInterface::is_surface_planar
bool is_surface_planar(int surface_id)

```
if cubit.is_surface_planar(22):
```

Parameters

**surface_id** Specifies the id of the surface

Returns

True is surface is planer, otherwise false

### ◆ is_type_filtered()

bool is_type_filtered ( const std::string & *filter_type* )

Determine whether a type is filtered.

### ◆ is_undo_save_needed()

bool is_undo_save_needed ( )

Get the status of the model relative to undo checkpointing.

Returns

A boolean indicating whether the model has been modified

### ◆ is_virtual()

bool is_virtual ( const std::string & *geometry_type*,
int *entity_id*
)

Query virtualality for a specific entity.

```
if (CubitInterface::is_virtual ("surface" , 134)) . . .
```

CubitInterface::is_virtual
bool is_virtual(const std::string &geometry_type, int entity_id)
Query virtualality for a specific entity.

```
if cubit.is_virtual("surface" , 134)):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

### ◆ is_visible()

bool is_visible ( const std::string & *geometry_type*,
int *entity_id*
)

Query visibility for a specific entity.

```
if (CubitInterface::is_visible ("volume" , 4)) . . .
```

CubitInterface::is_visible
bool is_visible(const std::string &geometry_type, int entity_id)
Query visibility for a specific entity.

```
if cubit.is_visible("volume" , 4)):
```

Parameters

**geom_type** Specifies the geometry type of the entity
**entity_id** Specifies the id of the entity

### ◆ is_volume_meshable()

bool is_volume_meshable(int *volume_id*)

Check if volume is meshable with current scheme.
Returns
A boolean indicating whether volume is meshable with current scheme

### ◆ is_working_dir_set()

bool is_working_dir_set( )

Create BCVizInterface for CompSimUI.
Returns
was the -workingdir passed in from the command line
Returns
boolean value indicating whether -working dir was set

### ◆ journal_commands()

void journal_commands(bool *state*)

Set the journaling flag in cubit.
Parameters

**state** A boolean that turns journaling on (1) and off (0)

### ◆ load_ML()

bool load_ML(std::string *model_type* = `"all"`)

load the machine learning training data
Parameters

**model_type** should be one of "all", "classification" or "regression"

### ◆ measure_between_entities()

std::vector< double > measure_between_entities(std::string *entity_type1*,
int *entity_id1*,
std::string *entity_type2*,
int *entity_id2*
)

returns distance between two geometry entities and their closest points

```
      std::vector<double> dist_info =
      CubitInterface::measure_between_entities ("curve" , 10, "surface" ,
12)
      double dist = dist_info[0]
```

```
      std::vector<double  >  curv_point  =  {dist_info[1],  dist_info[2],
dist_info[3]};
      std::vector<double>  surf_point  =  {dist_info[4],  dist_info[5],
dist_info[6]};
```

CubitInterface::measure_between_entities

std::vector< double > measure_between_entities(std::string entity_type1, int entity_id1, std::string entity_type2, int entity_id2)

returns distance between two geometry entities and their closest points

```
      dist_info = cubit.measure_between_entities("curve" , 10, "surface"
, 12)
      dist = dist_info[0]
      curv_point = [dist_info[1], dist_info[2], dist_info[3]]
      surf_point = [dist_info[4], dist_info[5], dist_info[6]]
```

Parameters

| | |
|---|---|
| **entity_type1** | type of first entity |
| **entity_id1** | id of first entity |
| **entity_type2** | type of second entity |
| **entity_id2** | id of second entity |

◆ ML_train()

bool ML_train ( std::string *geom_type* )

force a new training. Currently implemented for only classification methods, volume_no_op and surface_no_op.

Parameters

| | |
|---|---|
| **geom_type** | "volume" or "surface" |

◆ move()

void move ( **Entity** *entity*,
std::array< double, 3 > *vector*,
bool *preview* = `false`
)

Moves the **Entity** the specified vector.

Parameters

| | | |
|---|---|---|
| [in] | **entity** | The **Entity** to be moved |
| [in] | **vector** | The vector the **Entity** will be moved |
| [in] | **preview** | Flag to show the preview or not, default is false |

◆ number_undo_commands()

int number_undo_commands ( )

Query whether there are any undo commands to execute.

Returns

The number of commands in the undo stack

◆ override_journal_stream()

void override_journal_stream(JournalStreamBase * *jnl_stream*)

Override the Journal Stream in CUBIT.
Returns

## ◆ parse_cubit_list()

std::vector< int > parse_cubit_list(const std::string & *type*,
std::string *entity_list_string*
)

Parse a Cubit style entity list into a list of integers.
Users are allowed to input many variations of entities and IDs for any given command. This routine parses the input and returns a regular list of valid IDs for the specified entity type. For example: parse_cubit_list('surface', '1 to 12') parse_cubit_list('surface', 'with name "myname*"') parse_cubit_list('surface', 'in volume 5 to 23')
Parameters

**type** The specific entity type represented by the list of entities
**int_list** The string that contains the entity list

Returns
A vector (python tuple) of validated integers

## ◆ parse_locations()

std::vector< std::array< double, 3 > > parse_locations(const std::string & *location_str*)

## ◆ pause_playback()

void pause_playback( )

Pause playback.

## ◆ plugin_manager()

CubitPluginManager * plugin_manager( )

## ◆ print_cmd_options()

void print_cmd_options( )

Used to print the command line options.

## ◆ print_current_selections()

void print_current_selections( )

Print the current selections.

## ◆ print_currently_selected_entity()

void print_currently_selected_entity( )

Print the current selection.

### ◆ print_info()

void print_info(const std::string & *message*)

Print a message using the cubit message handler.
Parameters

**message** The message to print.

### ◆ print_raw_help()

void print_raw_help(const char * *input_line*,
int *order_dependent*,
int *consecutive_dependent*
)

Used to print out help when a ?, & or ! is pressed.
Parameters

**input_line** The current command line being typed by the user
**order_dependent** Is set to '1' if the key pressed is not &, otherwise '0'
**consecutive_dependent** Is set to '1' if the pressed is '?', otherwise '0'

### ◆ print_surface_summary_stats()

void print_surface_summary_stats( )

Print the surface summary stats to the console.

### ◆ print_volume_summary_stats()

void print_volume_summary_stats( )

Print the volume summary stats to the console.

### ◆ prism()

**Body** prism(double *height*,
int *sides*,
double *major*,
double *minor*
)

Creates a prism of the specified dimensions.
Parameters

[in] **height** The height of the prism
[in] **sides** The number of sides of the prism
[in] **major** The major radius
[in] **minor** The minor radius

Returns

A **Body** object of the newly created prism

### ◆ process_input_files()

```
void process_input_files ( )
```

**C++ only**

### ◆ project_unit_square()

```
std::vector< std::vector< double > >
project_unit_square                 (std::vector< std::vector< double > > pts,
                                     int                                  surface_id,
                                     int                                  quad_id,
                                     int                                  node00_id,
                                     int                                  node10_id
                                    )
```

Given points in a unit square, map them to the given quad using the orientation info, then project them onto the given surface, and return their projected positions.
Parameters

| | |
|---|---|
| **pts** | The x,y (abstract u,v) coordinates of the input points. Should be in [0,1]. |
| **surf_id** | The surface. |
| **quad_id** | The quad. |
| **node00_id** | The id of the node of the quad corresponding to an input point with coordinates (0,0) |
| **node10_id** | The id of the node of the quad corresponding to an input point with coordinates (1,0) |

Returns

Return the position on the surface of each input node, in the same order as the input was given

### ◆ pyramid()

```
Body pyramid (double height,
              int    sides,
              double major,
              double minor,
              double top = 0.0
             )
```

Creates a pyramid of the specified dimensions.
Parameters

| | |
|---|---|
| [in] **height** | The height of the pyramid |
| [in] **sides** | The number of sides of the pyramid |
| [in] **major** | The major radius |
| [in] **minor** | The minor radius |
| [in] **top** | determines size for the top of the pyramid. Defaults to 0, meaning it will go to a point |

Returns
A **Body** object of the newly created pyramid

◆ reflect()

void reflect(**Entity** *entity*,
std::array< double, 3 > *axis*,
bool *preview* = `false`
)

Reflect the **Entity** about the specified axis.
Parameters

`[in]` **entity** The **Entity** to be reflected
`[in]` **axis** The axis to be reflected about
`[in]` **preview** Flag to show the preview or not, default is false

◆ release_interface()

bool release_interface(CubitBaseInterface * *instance*)

Release the interface with the given name.
Parameters

**interface_name** the name of interface

◆ remove_entity_from_group()

void remove_entity_from_group(int *group_id*,
int *entity_id*,
const std::string & *entity_type*
)

Remove a specific entity from a specific group.

```
CubitInterface::remove_entity_from_group (3, 22, "surface" );
```

CubitInterface::remove_entity_from_group
void remove_entity_from_group(int group_id, int entity_id, const std::string &entity_type)
Remove a specific entity from a specific group.

```
cubit.remove_entity_from_group(3, 22, "surface" )
```

Parameters

**group_id** ID of group from which the entity will be removed
**entity_id** ID of the entity to be removed from the group
**entity_type** Type of the entity to be removed from the group. Note that only geometric entities can be removed

◆ remove_filter_type()

void remove_filter_type(const std::string & *filter_type*)

Remove a filter type.

### ◆ replace_progress_handler()

```
CubitProgressHandler                    *
replace_progress_handler                 (CubitProgressHandler * progress)
```

Register a new progress-bar callback handler with Cubit and return the the previous progress-handler without deleting it.
Parameters

**progress** A pointer to a CubitProgressHandler instance

Returns

pointer to previous progress handler

### ◆ report_usage()

```
void report_usage ( )
```

### ◆ reset()

```
void reset ( )
```

Executes a reset within cubit.

### ◆ reset_camera()

```
void reset_camera ( )
```

reset the camera in all open windows this includes resetting the view, closing the histogram and color windows and clearing the scalar bar, highlight, and picked entities.

### ◆ resume_playback()

```
void resume_playback ( )
```

resume playback.

### ◆ scale()

```
void scale ( Entity  entity,
             double  factor,
             bool    preview = false
           )
```

Scales the **Entity** according to the specified factor.
Parameters

`[in]` **entity** The **Entity** to be scaled
`[in]` **factor** The scale factor
`[in]` **preview** Flag to show the preview or not, default is false

### ◆ set_copy_block_on_geometry_copy_setting()

```
bool set_copy_block_on_geometry_copy_setting ( std::string val )
```

Set the copy block on geometry copy setting "ON", "USE_ORIGINAL", or "OFF".

Returns
success/fail setting the setting

◆ set_copy_nodeset_on_geometry_copy_setting()

bool set_copy_nodeset_on_geometry_copy_setting(std::string *val*)

Set the copy nodeset on geometry copy setting "ON", "USE_ORIGINAL", or "OFF".
Returns
success/fail setting the setting

◆ set_copy_sideset_on_geometry_copy_setting()

bool set_copy_sideset_on_geometry_copy_setting(std::string *val*)

Set the copy sideset on geometry copy setting "ON", "USE_ORIGINAL", or "OFF".
Returns
success/fail setting the setting

◆ set_cubit_interrupt()

void set_cubit_interrupt(bool *interrupt*)

This sets the global flag in Cubit that stops all interruptable processes.
Parameters

**interrupt** Boolean set to TRUE if process is to be stopped

◆ set_cubit_message_handler()

void set_cubit_message_handler(CubitMessageHandler * *hdlr*)

redirect the output from cubit.
Parameters

**hdlr**

◆ set_element_variable()

void set_element_variable(std::vector< int > *element_ids*,
std::string *variable_name*,
std::vector< double > *variables*
)

Sets element variables.
Parameters

[in] **element_ids** Elements to be set with variables.
[in] **variable_name** The name of the element variable.
[in] **variable** The variables for each elmenet. Must be either the same size as the 'element_ids' list or one value. Example: cubit.set_element_variable( [1,4,5,12], 'compression', [1.2, 1.53, 2.3, 9.5] )

◆ set_entity_name()

bool set_entity_name(const std::string & *entity_type*,
int *entity_id*,
const std::string & *new_name*
)

Set the name of a specified entity.

```
CubitInterface::set_entity_name ("vertex" , 22, "new_name" );
```

CubitInterface::set_entity_name
bool set_entity_name(const std::string &entity_type, int entity_id, const std::string &new_name)
Set the name of a specified entity.

Parameters

**entity_type** Specifies the type of the entity
**entity_id** Specifies the id of the entity
**new_name** Specifies what the name of the entity should be changed to

Returns

true if entity was found and rename, otherwise false.

◆ set_exit_handler()

void set_exit_handler(ExternalExitHandler * *hdlr*)

Set the exit handler.
Parameters

**An** instance of a class that inherits from ExternalExitHandler

◆ set_filter_types()

void set_filter_types(int *num_types*,
const std::vector< std::string > *filter_types*
)

Set the pick filter types.

◆ set_label_type()

void set_label_type(const char * *entity_type*,
int *label_flag*
)

make calls to SVDrawTool::set_label_type
Returns

none.

◆ set_max_group_id()

void set_max_group_id(int *maximum_group_id*)

Reset Cubit's max group id This is really dangerous to use and exists only to overcome a limitation with Cubit. Cubit keeps track of the next group id to assign. But

those ids just keep incrementing in Cubit. Some of the power tools in the Cubit GUI make groups 'under the covers' for various operations. The groups are immediately deleted. But, creating those groups will cause Cubit's group id to increase and downstream journal files may be messed up because those journal files are expecting a certain ID to be available.
When using this call the user must ensure the group max_group_id is under their control. Typically, a user will create a group, use it, then immediately delete it. This call will only work if the max_group_id is the same as Cubit's max group id. If it is Cubit's max id will be reset. If not, nothing will happen.
Parameters

**max_id** ID of group to make 'max'

◆ set_ML_base_user_dir()

```
void set_ML_base_user_dir(const std::string path,
                          const bool        print_info = false
                         )
```

set the path to any user training data. (classification only)
Parameters

**path** top level training directory (should contain ml/volume_no_op dir)
**print_info** print info to output
**only** use only user training data – don't use cubit training data

◆ set_modified()

```
void set_modified( )
```

Set the status of the model (**is_modified()** is now false). If you modify the model after you set this flag, it will register true.

◆ set_nodal_variable()

```
void set_nodal_variable(std::vector< int >    node_ids,
                        std::string           variable_name,
                        std::vector< double > variables
                       )
```

Sets nodal variables.
Parameters

[in] **node_ids** Nodes to be set with variables.
[in] **variable_name** The name of the node variable.
[in] **variable** The variables for each node. Must be either the same size as the 'node_ids' list or one value. Example: cubit.set_nodal_variable( [1,4,5,12], 'compression', [1.2, 1.53, 2.3, 9.5] )

◆ set_overlap_max_angle()

void set_overlap_max_angle(const double *maximum_angle*)

Set the max angle setting for calculating surface overlaps.
Parameters

**max**angle
Returns

◆ set_overlap_max_gap()

void set_overlap_max_gap(const double *maximum_gap*)

Set the max gap setting for calculating surface overlaps.
Parameters

**max**gap
Returns

◆ set_overlap_min_gap()

void set_overlap_min_gap(const double *min_gap*)

Set the min gap setting for calculating surface overlaps.
Parameters

**min_gap**
Returns

◆ set_pick_type()

void set_pick_type(const std::string & *pick_type*,
bool *silent* = `false`
)

Set the pick type.

◆ set_playback_paused_on_error()

void set_playback_paused_on_error(bool *pause*)

Sets whether or not playback is paused when an error occurs.
Parameters

**pause**True if playback should be paused when an error occurs.

◆ set_progress_handler()

void set_progress_handler(CubitProgressHandler * *progress*)

Register a progress-bar callback handler with Cubit. Deletes the current progress handler if it exists.
Parameters

**progress**A pointer to a CubitProgressHandler instance

◆ set_rendering_mode()

void set_rendering_mode(int *mode*)

Set the current rendering mode.
Parameters

**mode** Integer associated with the rendering mode. Options are 1,7,2,8, or 5

◆ set_undo_saved()

void set_undo_saved()

Set the status of the model relative to undo checkpointin.

◆ silent_cmd()

bool silent_cmd(const char * *input_string*)

Pass a command string into Cubit and have it executed without being verbose at the command prompt.
Passing a command into Cubit using this method will result in an immediate execution of the command. The command is passed directly to Cubit without any validation or other checking.

```
CubitInterface::silent_cmd ("display" );
```

CubitInterface::silent_cmd
bool silent_cmd(const char *input_string)
Pass a command string into Cubit and have it executed without being verbose at the command prompt.

```
cubit.silent_cmd("display" )
```

Parameters

**input_string** Pointer to a string containing a complete Cubit command

◆ snap_locations_to_geometry()

std::vector< std::array< double, 3 > > snap_locations_to_geometry (const std::vector< std::array< double, 3 > > & *locations*, std::string *entity_type*, int *entity_id*, double *tol*)

Snaps xyz locations to closest point on entity. Then snaps to child curves or vertices within given tolerance. Vertices snapped to before curves.

```
#give list of lists of 3 points in form: [ [x, y, z], [x, y, z], ...]
locations = [ [0.0, 1.0, 3.0], [0.1, 1.1, 4.2] ]
snapped_xyz_vec = cubit.snap_locations_to_geometry( locations, "volume" , 1, 0.001 )
```

### ◆ sphere()

```
Body sphere(double radius,
            int    x_cut = 0,
            int    y_cut = 0,
            int    z_cut = 0,
            double inner_radius = 0
           )
```

Creates all or part of a sphere.
Parameters

| | | |
|---|---|---|
| [in] | **radius** | The radius of the sphere |
| [in] | **x_cut** | If 1, cuts sphere by yz plane (default to 0) |
| [in] | **y_cut** | If 1, cuts sphere by xz plane (default to 0) |
| [in] | **z_cut** | If 1, cuts sphere by xy plane (default to 0) |
| [in] | **inner_radius** | The inside radius if the sphere is hollow (default to 0) |

Returns

A **Body** object of the newly created sphere

### ◆ step_next_possible_selection()

```
void step_next_possible_selection()
```

Step to the next possible selection (selected next dialog)

### ◆ step_previous_possible_selection()

```
void step_previous_possible_selection()
```

Step to the previous possible selection (selected next dialog)

### ◆ stop_playback()

```
void stop_playback()
```

Pause playback.

### ◆ string_from_id_list()

```
std::string string_from_id_list(std::vector< int > ids)
```

Parse a list of integers into a Cubit style id list. Includes carriage return and line breaks at column 80.
For example: string_from_id_list(<1, 2, 3, 4, 5, 6, 7, 8>) returns '1 to 8' example: string_from_id_list(<1, 2, 3, 100, 5, 6, 7, 8>) returns '1 to 3, 5 to 8, 100'

```
std::vector<int> entity_ids = {1, 2, 3, 4};
std::string id_string = CubitInterface::get_id_string (entity_ids);
// id_string is "1 to 4\n";
```

```
entity_ids = [1,2,3,4]
id_string = cubit.get_all_ids_from_name(entity_ids)
# id_string is '1 to 4\n'
```

Parameters

**ids** The vector of integer ids

Returns

A string representing the id list with line breaks

◆ subtract()

```
std::vector< Body >
subtract (std::vector< CubitInterface::Body > tool_in,
          std::vector< CubitInterface::Body > from_in,
          bool imprint_in = false,
          bool keep_old_in = false
         )
```

Performs a boolean subtract operation.

Parameters

| | | |
|---|---|---|
| [in] | **tool_in** | List of **Body** objects to subtract |
| [in] | **from_in** | List of **Body** objects to be subtracted from |
| [in] | **imprint_in** | Flag to set the imprint (defaults to false) |
| [in] | **keep_old_in** | Flag to keep the old volume (defaults to false) |

Returns

A list of changed body objects

◆ surface()

**CubitInterface::Surface** surface(int *id_in*)

Gets the surface object from an ID.

Parameters

**id_in** The ID of the surface

Returns

The surface object

◆ sweep_curve()

```
std::vector< Body > sweep_curve(std::vector< Curve > curves,
                                std::vector< Curve > along_curves,
                                double draft_angle = 0,
                                int draft_type = 0,
                                bool rigid = false
                               )
```

Create a **Body** or a set of Bodies from a swept curve.

Parameters

| | | |
|---|---|---|
| [in] | **curves** | A list of curves to sweep |
| [in] | **along_curves** | A list of curves to sweep along |
| [in] | **draft_angle** | The sweep draft angle (default to 0) |

[in] **draft_type** The draft type (default to 0) 0 => extended (draws two straight tangent lines from the ends of each segment until they intersect) 1 => rounded (create rounded corner between segments) 2 => natural (extends the shapes along their natural curve) ***

[in] **rigid** The inside radius if the sphere is hollow (default to False)

Returns

A List of newly created Bodies

### ◆ temperature_is_on_shell_area()

```
bool temperature_is_on_shell_area ( CI_BCTypes       bc_type_enum,
                                    CI_BCEntityTypes bc_area_enum,
                                    int              entity_id
                                  )
```

Determine whether a BC temperature is on a shell area. Valid for convection and temperature and on top, bottom, gradient, and middle.

Parameters

**bc_type** enum of CI_BCTypes. temperature = 4, convection = 7

**bc_area** enum of CI_BCEntityTypes. Use 7 for top, 8 for bottom, 9 for gradient, 10 for middle

**entity_id** Id of the BC

Returns

true if BC temperature is on the shell area, otherwise false

### ◆ temperature_is_on_solid()

```
bool temperature_is_on_solid ( CI_BCTypes bc_type_enum,
                               int        entity_id
                             )
```

Determine whether a BC temperature is on a solid. Valid for convection and temperature.

Parameters

**bc_type_in** enum of CI_BCTypes. temperature = 4, convection = 7

**entity_id** Id of the BC

Returns

true if BC temperature is on a solid, otherwise false

### ◆ torus()

```
Body torus ( double center_radius,
             double swept_radius
           )
```

creates a torus of the specified dimensions

Parameters

[in] **r1** radius from center to center of circle to be swept (r1>r2)
[in] **r2** radius of circle swept to create torus (r1>r2)

Returns
A **Body** object of the newly created torus

◆ tweak_curve_offset()

```
std::vector< Body > tweak_curve_offset( std::vector< Curve > curves,
                                        std::vector< double > distances,
                                        bool keep_old = false,
                                        bool preview = false
                                      )
```

Performs a tweak curve offset command.
Parameters

[in] **curves** A list of curve objects to offset
[in] **distances** A list of distances associated with the offset for each curve
[in] **keep_old** Keep the old body (defaults to false)
[in] **preview** Flag to show the preview (defaults to false)

Returns
A list of changed body objects

◆ tweak_curve_remove()

```
std::vector< CubitInterface::Body >
tweak_curve_remove                  ( std::vector< Curve > curves,
                                      bool keep_old = false,
                                      bool preview = false
                                    )
```

Removes a curve from a body and extends the surrounding surface to fill the gap.
Removes a curve from a body
Parameters

[in] **surfaces** A list of the curves to be removed
[in] **keep_old** Keep the old body (defaults to false)
[in] **preview** Flag to show the preview (defaults to false)

Returns
A list of changed body objects

◆ tweak_surface_offset()

```
std::vector< Body > tweak_surface_offset( std::vector< Surface > surfaces,
                                          std::vector< double > distances
                                        )
```

Performs a tweak surface offset command.
Parameters

**surfaces** A list of surface objects to offset
**distances** A list of distances associated with the offset for each surface

Returns
A list of the body objects of the modified bodies

◆ tweak_surface_remove()

```
std::vector< CubitInterface::Body >
tweak_surface_remove (std::vector< Surface > surfaces,
                      bool extend_ajoining = true,
                      bool keep_old = false,
                      bool preview = false
                     )
```

Removes a surface from a body and extends the surrounding surfaces if extend_ajoining is true.
Removes a surface from a body
Parameters

[in] **surfaces** The surfaces to be removed
[in] **extend_ajoining** Extend the ajoining surfaces (default to true)
[in] **keep_old** Keep the old body (default to false)
[in] **preview** Flag to show the preview or not (default to false)

Returns
A list of changed body objects

◆ tweak_vertex_fillet()

```
std::vector< Body > tweak_vertex_fillet (std::vector< Vertex > verts,
                                         double radius,
                                         bool keep_old = false,
                                         bool preview = false
                                        )
```

Performs a tweak vertex fillet command.
Parameters

[in] **verts** A list of vertex objects to fillet
[in] **r0** radius of the fillet
[in] **keep_old** Keep the old body (defaults to false)
[in] **preview** Flag to show the preview (defaults to false)

Returns
A list of changed body objects

◆ unite()

```
std::vector< Body > unite (std::vector< CubitInterface::Body > body_in,
                           bool keep_old_in = false
                          )
```

Performs a boolean unite operation.
Parameters

[in] **body_in** A list of body objects to unite
[in] **keep_old_in** Flag to keep old bodies (defaults to false)

Returns
A list of changed bodies

### ◆ unload_ML()

void unload_ML(std::string *model_type* = `"all"`)

unload the machine learning training data
Parameters

**model_type** should be one of "all", "classification" or "regression"

### ◆ unselect_entity()

void unselect_entity(const std::string & *entity_type*,
int *entity_id*
)

Unselect an entity that is currently selected.
Unselecting an entity will unhighlight it in the graphics window and remove it from the global pick list.

```
CubitInterface::unselect_entity ("curve" , 221);
```

CubitInterface::unselect_entity
void unselect_entity(const std::string &entity_type, int entity_id)
Unselect an entity that is currently selected.

```
cubit.unselect_entity("curve" , 221)
```

Parameters

**entity_type** The type of the entity to be unselected
**entity_id** The ID of the entity to be unselected

### ◆ vertex()

**CubitInterface::Vertex** vertex(int *id_in*)

Gets the vertex object from an ID.
Parameters

**id_in** The ID of the vertex

Returns
The vertex object

### ◆ volume()

**CubitInterface::Volume** volume(int *id_in*)

Gets the volume object from an ID.
Parameters

**id_in** The ID of the volume

Returns

The volume object

### ◆ volume_contains_tets()

bool volume_contains_tets(int *volume_id*)

Determine whether a volume contains tets.

Returns

bool

### ◆ was_last_cmd_undoable()

bool was_last_cmd_undoable( )

Report whether the last executed command was undoable.

Returns

true if the last executed command was undoable

### ◆ write_to_journal()

void write_to_journal(std::string *words*)

Write a string to the active journal.

words string to write to journal file

Returns

A boolean indicating whether commands are journaled by Cubit

## Variable Documentation

### ◆ CI_ERROR

const int CI_ERROR = -1



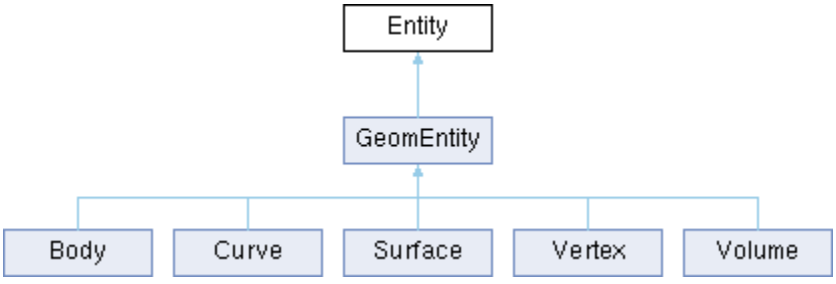


# Entity

The base class of all the geometry and mesh types. More...

`#include <CubitInterfaceEx.hpp>`

Inheritance diagram for Entity:



## Public Member Functions

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Protected Attributes

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

The base class of all the geometry and mesh types.

```
import cubit
br = cubit.brick(1,1,1)
cubit.scale(br,2)
cubit.cmd('delete body 1' )
```

## Constructor & Destructor Documentation

◆ ~Entity()

~**Entity**( )

◆ Entity() [1/3]

**Entity**( )

◆ Entity() [2/3]

**Entity**(CubitEntity * *entity_ptr*)

◆ Entity() [3/3]

**Entity**(**Entity** const & *copy_from*)

## Member Function Documentation

◆ bounding_box()

std::array< double, 6 > bounding_box( )

Get the bounding box of the **Entity** .

```
std::array<double,6> b_box = entity->bounding_box();
b_box = entity.bounding_box()
```

Returns

The bounding box as a vector (or list) where the indices correspond to the values as follows:
0 - minimum x value
1 - minimum y value
2 - minimum z value
3 - maximum x value
4 - maximum y value
5 - maximum z value

◆ center_point()

std::array< double, 3 > center_point( )

Get the center point of the **Entity** .

```
std::array<double,3> center = entity->center_point();
center = entity.center_point()
```

Returns

The center point as a vector (or list) where the indices correspond to the values as follows:
0 - x value
1 - y value
2 - z value

◆ destroy_cubit_entity()

void destroy_cubit_entity( )

◆ entity_ptr()

CubitEntity * entity_ptr ( )

◆ id()

int id ( )

Get the id of the **Entity** .

```
intid = entity->id();
id = entity.id()
```

Returns
The id of the **Entity**

◆ operator=()

**Entity** & operator= ( const **Entity** & *rhs* )

Member Data Documentation

◆ cubitWatcher

CubObserver* cubitWatcher protected

◆ mEntityPtr

CubitEntity* mEntityPtr protected



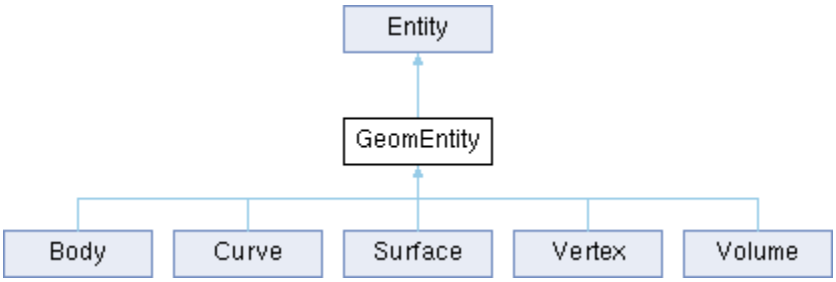
Public Member Functions | Protected Member Functions

# GeomEntity

The base class for specifically the Geometry types (**Body** , **Surface** , etc.) More...

```
#include <CubitInterfaceEx.hpp>
```

Inheritance diagram for GeomEntity:




Public Member Functions

| | |
|---|---|
| std::vector< **Body** > | **bodies** ()<br>Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** () |

| | |
|---|---|
| | Get the curves in the **GeomEntity**. |
| int | **dimension** ()<br>Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** ()<br>Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** ()<br>Return the current mesh state of the **GeomEntity**. |
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |

| | |
|---|---|
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |
| void | **remove_entity_names** ()<br>Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| void | **set_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

**Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |

| | |
|---|---|
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Protected Member Functions

| |
|---|
| **GeomEntity** (const **GeomEntity** &other) |
| **GeomEntity** (CubitEntity ***entity_ptr**) |

## Additional Inherited Members

**Protected Attributes inherited from Entity**

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

The base class for specifically the Geometry types (**Body** , **Surface** , etc.)

## Constructor & Destructor Documentation

◆ GeomEntity() [1/2]

**GeomEntity** ( CubitEntity * *entity_ptr* ) inlineprotected

◆ GeomEntity() [2/2]

**GeomEntity** ( const **GeomEntity** & *other* ) inlineprotected

## Member Function Documentation

### ◆ bodies()

std::vector< **Body** > bodies ( )

Get the bodies in the **GeomEntity** .

```
std::vector<Body> bodies = geomEntity->bodies();
```

CubitInterface::GeomEntity::bodies
std::vector< Body > bodies()
Get the bodies in the GeomEntity.

```
bodies = geomEntity.bodies()
```

Returns
A vector (or list) of bodies contained within the **GeomEntity**

### ◆ curves()

std::vector< **Curve** > curves ( )

Get the curves in the **GeomEntity** .

```
std::vector<Curve> curves = geomEntity->curves();
```

CubitInterface::GeomEntity::curves
std::vector< Curve > curves()
Get the curves in the GeomEntity.

```
curves = geomEntity.curves()
```

Returns
A vector (or list) of curves contained within the **GeomEntity**

### ◆ dimension()

int dimension ( )

Get the dimensions of the **GeomEntity** .

```
int dim = geomEntity->dimension();
dim = geomEntity.dimension()
```

Returns
The dimension of the **GeomEntity**

### ◆ entity_name() [1/2]

std::string entity_name ( )

Return the first name of the **GeomEntity** .

```
std::string name = geomEntity->entity_name();
name = geomEntity.entity_name()
```

Returns
The first name of the **GeomEntity**

### ◆ entity_name() [2/2]

void entity_name ( std::string *name* )

Assign a name to the **GeomEntity** .

```
geomEntity->entity_name("Brick1" );
geomEntity.entity_name("Brick1" )
```

Parameters

[in] **name** The name to be assigned to the **GeomEntity**

### ◆ entity_names()

std::vector< std::string > entity_names ( )

Return the all the names of the **GeomEntity** .

```
std::vector<std::string> names = geomEntity->entity_names();
names = geomEntity.entity_names()
```

Returns

A vector (or list) of all the names of the **GeomEntity**

### ◆ is_meshed()

bool is_meshed ( )

Return the current mesh state of the **GeomEntity** .

```
boolmesh = geomEntity->is_meshed();
```

CubitInterface::GeomEntity::mesh
void mesh()
Mesh the GeomEntity.

```
mesh = geomEntity.is_meshed()
```

Returns

Whether the **GeomEntity** is meshed or not

### ◆ is_transparent() [1/2]

int is_transparent ( )

Get the tranparency state of the **Entity** .

```
int trans = entity->is_transparent();
trans = entity.is_transparent()
```

Returns

The current transparency state of the **Entity** (1 if transparent, 0 if not)

### ◆ is_transparent() [2/2]

void is_transparent ( int *transparency_flag* )

Set the tranparency state of the **Entity** .

```
entity->is_transparent(1);
entity.is_transparent(1)
```

Parameters

[in] **transparency_flag** The flag that sets whether the **Entity** is transparent (1) or not (0)

### ◆ is_visible() [1/2]

int is_visible ( )

Get the visibility state of the **Entity** .

```
int vis = entity->is_visible();
vis = entity.is_visible()
```

Returns

The current visiblity state of the **Entity** (1 if visible, 0 if not)

### ◆ is_visible() [2/2]

void is_visible ( bool *visibility_flag* )

Set the visibility state of the **Entity** .

```
entity->is_visible(1);
entity.is_visible(1)
```

Parameters

[in] **visibility_flag** The flag that sets whether the **Entity** is visible (1) or not (0)

### ◆ mesh()

void mesh ( )

Mesh the **GeomEntity** .

```
geomEntity->mesh();
geomEntity.mesh()
```

### ◆ num_names()

int num_names ( )

Return the number of names for the **GeomEntity** .

```
int num = geomEntity->num_names();
num = geomEntity.num_names()
```

Returns

The number of names for the **GeomEntity**

### ◆ remove_entity_name()

void remove_entity_name ( std::string *name* )

Remove a specific name from the list of names assigned to the **GeomEntity** .

```
geomEntity->remove_entity_name("Brick1" );
geomEntity.remove_entity_name("Brick1" )
```

Parameters

[in] **name** The name to be removed from the list of names assigned to the **GeomEntity**

### ◆ remove_entity_names()

void remove_entity_names ( )

Remove all the names assigned to the **GeomEntity** .

```
geomEntity->remove_entity_names();
geomEntity.remove_entity_names()
```

### ◆ remove_mesh()

void remove_mesh ( )

Removes the mesh on the **GeomEntity** .

```
geomEntity->remove_mesh();
geomEntity.remove_mesh()
```

### ◆ set_entity_name()

void set_entity_name ( std::string *name* )

Assign a name to the **GeomEntity** .

```
geomEntity->set_entity_name("Brick1" );
geomEntity.set_entity_name("Brick1" )
```

Parameters

[in] **name** The name to be assigned to the **GeomEntity**

### ◆ set_transparent()

void set_transparent ( int *transparency_flag* )

Set the tranparency state of the **Entity** .

```
entity->set_transparent(1);
entity.set_transparent(1)
```

Parameters

[in] **transparency_flag** The flag that sets whether the **Entity** is transparent (1) or not (0)

### ◆ set_visible()

void set_visible ( bool *visibility_flag* )

Set the visibility state of the **Entity** .

```
entity->set_visible(1);
entity.set_visible(1)
```

Parameters

[in] **visibility_flag** The flag that sets whether the **Entity** is visible (1) or not (0)

### ◆ smooth()

void smooth ( )

Smooths the mesh on the **GeomEntity** .

```
geomEntity->smooth();
geomEntity.smooth()
```

◆ surfaces()

std::vector< **Surface** > surfaces ( )

Get the surfaces in the **GeomEntity** .

```
        std::vector<Surface> surfaces = geomEntity->surfaces();
```

CubitInterface::GeomEntity::surfaces
std::vector< Surface > surfaces()
Get the surfaces in the GeomEntity.

```
        surfaces = geomEntity.surfaces()
```

Returns

A vector (or list) of surfaces contained within the **GeomEntity**

◆ vertices()

std::vector< **Vertex** > vertices ( )

Get the vertices in the **GeomEntity** .

```
        std::vector<Vertex> vertices = geomEntity->vertices();
```

CubitInterface::GeomEntity::vertices
std::vector< Vertex > vertices()
Get the vertices in the GeomEntity.

```
        vertices = geomEntity.vertices()
```

Returns

A vector (or list) of vertices contained within the **GeomEntity**

◆ volumes()

std::vector< **Volume** > volumes ( )

Get the volumes in the **GeomEntity** .

```
        std::vector<Volume> volumes = geomEntity->volumes();
```

CubitInterface::GeomEntity::volumes
std::vector< Volume > volumes()
Get the volumes in the GeomEntity.

```
        volumes = geomEntity.volumes()
```

Returns

A vector (or list) of volumes contained within the **GeomEntity**



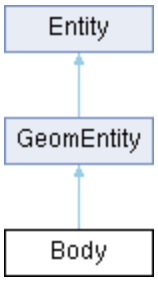
Public Member Functions

# Body

Defines a body object that mostly parallels Cubit's **Body** class. More...

```
#include <CubitInterfaceEx.hpp>
```

Inheritance diagram for Body:




## Public Member Functions

| | |
|---|---|
| | **Body** () |
| | **Body** (const **Body** &other) |
| | **Body** (CubitEntity ***entity_ptr**) |
| | **~Body** () |
| std::array< double, 3 > | **get_mass_props** ()<br>Get the mass properties of the **Body**, specifically the center of gravity. |
| bool | **is_sheet_body** ()<br>Get whether the **Body** is a sheet body or not. |
| int | **point_containment** (std::array< double, 3 > loc_in)<br>Get whether a point is in, on, or outside the **Body**. |
| double | **volume** ()<br>Get the volume of the **Body**. |

### Public Member Functions inherited from GeomEntity

| | |
|---|---|
| std::vector< **Body** > | **bodies** ()<br>Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** ()<br>Get the curves in the **GeomEntity**. |
| int | **dimension** () |

| | |
|---|---|
| | Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** ()<br>Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** ()<br>Return the current mesh state of the **GeomEntity**. |
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |

| | |
|---|---|
| void | **remove_entity_names** ()<br>Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| void | **set_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

**Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |

| | |
|---|---|
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Additional Inherited Members

**Protected Member Functions inherited from GeomEntity**

| | |
|---|---|
| | **GeomEntity** (const **GeomEntity** &other) |
| | **GeomEntity** (CubitEntity ***entity_ptr**) |

**Protected Attributes inherited from Entity**

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

Defines a body object that mostly parallels Cubit's **Body** class.

## Constructor & Destructor Documentation

### ◆ Body() [1/3]

**Body** ( ) inline

### ◆ Body() [2/3]

**Body** ( const **Body** & *other* ) inline

### ◆ ~Body()

~**Body** ( ) inline

◆ Body() [3/3]

**Body** ( CubitEntity * *entity_ptr* ) inline

Member Function Documentation

◆ get_mass_props()

std::array< double, 3 > get_mass_props ( )

Get the mass properties of the **Body** , specifically the center of gravity.

```
std::array<double,3> props = body ->get_mass_props ();
```

CubitInterface::Body::get_mass_props
std::array< double, 3 > get_mass_props()
Get the mass properties of the Body, specifically the center of gravity.
CubitInterface::body
CubitInterface::Body body(int id_in)
Gets the body object from an ID.

```
props = body .get_mass_props ()
```

Returns

A vector (or list) of numerical data corresponding to the center of gravity of the body with indices as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

◆ is_sheet_body()

bool is_sheet_body ( )

Get whether the **Body** is a sheet body or not.

```
bool is_sheet = body ->is_sheet_body ();
```

CubitInterface::Body::is_sheet_body
bool is_sheet_body()
Get whether the Body is a sheet body or not.

```
is_sheet = body .is_sheet_body ()
```

Returns

Whether the **Body** is a sheet body or not

◆ point_containment()

int point_containment ( std::array< double, 3 > *loc_in* )

Get whether a point is in, on, or outside the **Body** .

```
std::array<double,3> point(3, 0);
int on_out_in = body ->point_containment (point);
```

CubitInterface::Body::point_containment
int point_containment(std::array< double, 3 > loc_in)
Get whether a point is in, on, or outside the Body.

```
on_out_in = body .point_containment ([0,0,0])
```

Returns
Whether a point is unknown (-1), outside (0), in (1), or on (2) the **Body**

◆ volume()

double volume ( )

Get the volume of the **Body** .

```
double vol = body ->volume ();
```

CubitInterface::Body::volume
double volume()
Get the volume of the Body.

```
vol = body .volume ()
```

Returns
The volume of the **Body**



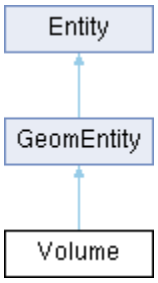
Public Member Functions

# Volume

Defines a volume object that mostly parallels Cubit's RefVolume class. More...

```
#include <CubitInterfaceEx.hpp>
```

Inheritance diagram for Volume:




## Public Member Functions

| | |
|---|---|
| | **Volume** () |
| | **Volume** (const **Volume** &other) |
| | **Volume** (CubitEntity ***entity_ptr**) |
| | **~Volume** () |
| std::array< double, 3 > | **centroid** ()<br>Get the centroid of the **Volume**. |

| | |
|---|---|
| std::array< double, 4 > | **color** ()<br>Get the color of the **Volume**. |
| std::array< double, 9 > | **principal_axes** ()<br>Get the principal axes of the **Volume**. |
| std::array< double, 3 > | **principal_moments** ()<br>Get the principal moments of the **Volume**. |
| void | **set_color** (std::array< double, 4 > value)<br>Set the color of the **Volume**. |
| double | **volume** ()<br>Get the volume of the **Volume**. |

**Public Member Functions inherited from GeomEntity**

| | |
|---|---|
| std::vector< **Body** > | **bodies** ()<br>Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** ()<br>Get the curves in the **GeomEntity**. |
| int | **dimension** ()<br>Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** ()<br>Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** () |

| | |
|---|---|
| | Return the current mesh state of the **GeomEntity**. |
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |
| void | **remove_entity_names** ()<br>Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |

| | |
|---|---|
| void | **set_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

**Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** () |

Get the id of the **Entity**.

**Entity** & **operator=** (const **Entity** &rhs)

## Additional Inherited Members

### Protected Member Functions inherited from GeomEntity

**GeomEntity** (const **GeomEntity** &other)

**GeomEntity** (CubitEntity ***entity_ptr**)

### Protected Attributes inherited from Entity

CubObserver * **cubitWatcher**

CubitEntity * **mEntityPtr**

## Detailed Description

Defines a volume object that mostly parallels Cubit's RefVolume class.

## Constructor & Destructor Documentation

### ◆ Volume() [1/3]

**Volume** ( ) inline

### ◆ ~Volume()

~**Volume** ( ) inline

### ◆ Volume() [2/3]

**Volume** ( const **Volume** & *other* ) inline

### ◆ Volume() [3/3]

**Volume** ( CubitEntity * *entity_ptr* ) inline

## Member Function Documentation

### ◆ centroid()

std::array< double, 3 > centroid ( )

Get the centroid of the **Volume** .

```
std::array<double,3> centroid = volume ->centroid();
```

CubitInterface::Volume::volume

double volume()

Get the volume of the Volume.

CubitInterface::Volume::centroid

std::array< double, 3 > centroid()

Get the centroid of the Volume.

```
centroid = volume .centroid()
```

Returns

A vector (or list) of the coordinates of the centroid of the volume with the indices of the vector corresponding to the values as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ color()

std::array< double, 4 > color ( )

Get the color of the **Volume** .

```
int col = volume ->color();
col = volume .color()
```

Returns

The color value associated with the volume's current color

### ◆ principal_axes()

std::array< double, 9 > principal_axes ( )

Get the principal axes of the **Volume** .

```
std::array<double,9> axes = volume ->principal_axes();
axes = volume .principal_axes()
```

Returns

A vector (or list) of the principal axes of the volume with the indices of the vector corresponding to the values as follows:
0 - axis 1 x value
1 - axis 1 y value
2 - axis 1 z value
3 - axis 2 x value
4 - axis 2 y value
5 - axis 2 z value
6 - axis 3 x value
7 - axis 3 y value
8 - axis 3 z value

### ◆ principal_moments()

std::array< double, 3 > principal_moments ( )

Get the principal moments of the **Volume** .

```
std::array<double,3> moments = volume ->principal_moments();
moments = volume .principal_moments()
```

Returns

A vector (or list) of the principal moments of the volume with the indices of the vector corresponding to the values as follows:
0 - x moment

1 - y moment
2 - z moment

◆ set_color()

void set_color(std::array< double, 4 > *value*)

Set the color of the **Volume** .

```
volume ->set_color(0);
volume .set_color(0)
```

Parameters

[in] **value** The color value that the volume will have

◆ volume()

double volume( )

Get the volume of the **Volume** .

```
double vol = volume ->volume();
vol = volume .volume()
```

Returns

The volume of the **Volume**



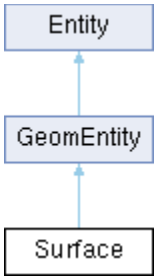
Public Member Functions

# Surface

Defines a surface object that mostly parallels Cubit's RefFace class. More...

`#include <CubitInterfaceEx.hpp>`

Inheritance diagram for Surface:




Public Member Functions

**Surface** ()

**Surface** (const **Surface** &other)

**Surface** (CubitEntity ***entity_ptr**)

| | |
|---|---|
| | **~Surface** () |
| double | **area** ()<br>Get area of the **Surface**. |
| std::array< double, 3 > | **closest_point_along_vector** (std::array< double, 3 > location, std::array< double, 3 > along_vector)<br>Get the nearest point on the **Surface** to point specified along the specified vector. |
| std::array< double, 3 > | **closest_point_trimmed** (**CubitInterface::Loc** location)<br>Get the nearest point on the **Surface** to point specified. |
| std::array< double, 3 > | **closest_point_trimmed** (std::array< double, 3 > location)<br>Get the nearest point on the **Surface** to point specified. |
| std::array< double, 4 > | **color** ()<br>Get the color of the surface. |
| std::array< double, 2 > | **get_param_range_U** ()<br>Get range of u for the **Surface**. |
| std::array< double, 2 > | **get_param_range_V** ()<br>Get range of v for the **Surface**. |
| bool | **is_cylindrical** ()<br>Get whether the **Surface** is cylindrical or not. |
| bool | **is_planar** ()<br>Get whether the **Surface** is planar or not. |
| std::array< double, 3 > | **normal_at** (std::array< double, 3 > location) |

| | |
|---|---|
| | Get the normal at a particular point on the **Surface**. |
| std::vector< std::vector< **Curve** > > | **ordered_loops** ()<br>Get the ordered loops of the **Surface**. |
| int | **point_containment** (std::array< double, 3 > point_in)<br>Get whether a point is on or off of the **Surface**. |
| std::array< double, 3 > | **position_from_u_v** (double u, double v)<br>Get the Cartesian coordinates from the uv coordinates on the **Surface**. |
| std::array< double, 2 > | **principal_curvatures** (std::array< double, 3 > point)<br>Get the principal curvatures of the **Surface**. |
| void | **set_color** (std::array< double, 4 > value)<br>Set the color of the surface. |
| std::array< double, 2 > | **u_v_from_position** (std::array< double, 3 > location)<br>Get the uv coordinates from the supplied Cartesian coordinates on the **Surface**. |

**Public Member Functions inherited from GeomEntity**

| | |
|---|---|
| std::vector< **Body** > | **bodies** ()<br>Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** ()<br>Get the curves in the **GeomEntity**. |
| int | **dimension** ()<br>Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** () |

| | |
|---|---|
| | Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** ()<br>Return the current mesh state of the **GeomEntity**. |
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |
| void | **remove_entity_names** () |

| | |
|---|---|
| | Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| void | **set_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

**Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |

| | |
|---|---|
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Additional Inherited Members

### Protected Member Functions inherited from GeomEntity

| | |
|---|---|
| | **GeomEntity** (const **GeomEntity** &other) |
| | **GeomEntity** (CubitEntity ***entity_ptr**) |

### Protected Attributes inherited from Entity

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

Defines a surface object that mostly parallels Cubit's RefFace class.

## Constructor & Destructor Documentation

### ◆ Surface() [1/3]

**Surface** ( ) inline

### ◆ ~Surface()

~**Surface** ( ) inline

### ◆ Surface() [2/3]

**Surface** ( const **Surface** & *other* ) inline

◆ Surface() [3/3]

**Surface** ( CubitEntity * *entity_ptr* ) inline

Member Function Documentation

◆ area()

double area ( )

Get area of the **Surface** .

```
doublearea = surface ->area ();
```

CubitInterface::Surface::area
double area()
Get area of the Surface.
CubitInterface::surface
CubitInterface::Surface surface(int id_in)
Gets the surface object from an ID.

```
area = surface .area ()
```

Returns
The area of the **Surface**

◆ closest_point_along_vector()

std::array< double, 3 >
closest_point_along_vector (std::array< double, 3 > *location*,
std::array< double, 3 > *along_vector*
)

Get the nearest point on the **Surface** to point specified along the specified vector.

```
std::array<double,3> point(3, 0);
std::array<double,3> along_vector{1,1,1};
std::array<double,3> nearest = surface ->closest_point_trimmed
(point);
```

CubitInterface::Surface::closest_point_trimmed
std::array< double, 3 > closest_point_trimmed(std::array< double, 3 > location)
Get the nearest point on the Surface to point specified.

```
nearest = surface .closest_point_along_vector ([0,0,0], [1,1,1])
```

CubitInterface::Surface::closest_point_along_vector
std::array< double, 3 > closest_point_along_vector(std::array< double, 3 > location,
std::array< double, 3 > along_vector)
Get the nearest point on the Surface to point specified along the specified vector.

Parameters

[in] **location** A vector containing three values that are the coordinates of a point

Returns
A vector (or list) of doubles representing values of nearest point as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ closest_point_trimmed() [1/2]

std::array< double, 3 > closest_point_trimmed ( **CubitInterface::Loc** *location* )

Get the nearest point on the **Surface** to point specified.

```
        std::array<double,3> point(3, 0);
        std::array<double,3> nearest = surface ->closest_point_trimmed
(point);
        nearest = surface .closest_point_trimmed ([0,0,0])
```

Parameters

[in] **location** A vector containing three values that are the coordinates of a point

Returns

A vector (or list) of doubles representing values of nearest point as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ closest_point_trimmed() [2/2]

std::array< double, 3 > closest_point_trimmed ( std::array< double, 3 > *location* )

Get the nearest point on the **Surface** to point specified.

```
        std::array<double,3> point(3, 0);
        std::array<double,3> nearest = surface ->closest_point_trimmed
(point);
        nearest = surface .closest_point_trimmed ([0,0,0])
```

Parameters

[in] **location** A vector containing three values that are the coordinates of a point

Returns

A vector (or list) of doubles representing values of nearest point as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ color()

std::array< double, 4 > color ( )

Get the color of the surface.

```
        int col = surface ->color ();
```

CubitInterface::Surface::color
std::array< double, 4 > color()
Get the color of the surface.

```
        col = surface .color ()
```

Returns

The color value associated with the surface's current color

### ◆ get_param_range_U()

std::array< double, 2 > get_param_range_U ( )

Get range of u for the **Surface** .

```
std::vector<double> bounds = surface ->get_param_range_U ();
```

CubitInterface::Surface::get_param_range_U
std::array< double, 2 > get_param_range_U()
Get range of u for the Surface.

```
bounds = surface .get_param_range_U ()
```

Returns

The curvature values:
0 - The lowest value in the u direction
1 - The highest value in the u direction

### ◆ get_param_range_V()

std::array< double, 2 > get_param_range_V ( )

Get range of v for the **Surface** .

```
std::vector<double> bounds = surface ->get_param_range_V ();
```

CubitInterface::Surface::get_param_range_V
std::array< double, 2 > get_param_range_V()
Get range of v for the Surface.

```
lower_bound, upper_bound = surface .get_param_range_V ()
```

Returns

The curvature values:
0 - The lowest value in the v direction
1 - The highest value in the v direction

### ◆ is_cylindrical()

bool is_cylindrical ( )

Get whether the **Surface** is cylindrical or not.

```
bool cyl = surface ->is_cylindrical ();
```

CubitInterface::Surface::is_cylindrical
bool is_cylindrical()
Get whether the Surface is cylindrical or not.

```
cyl = surface .is_cylindrical ()
```

Returns

Whether the **Surface** is cylindrical or not

### ◆ is_planar()

bool is_planar ( )

Get whether the **Surface** is planar or not.

```
bool planar = surface ->is_planar ();
```

CubitInterface::Surface::is_planar
bool is_planar()
Get whether the Surface is planar or not.

```
planar = surface .is_planar ()
```

Returns

Whether the **Surface** is planar or not

### ◆ normal_at()

std::array< double, 3 > normal_at(std::array< double, 3 > *location*)

Get the normal at a particular point on the **Surface** .

```
std::array<double,3> point(3, 0);
std::array<double,3> norm = surface ->normal_at (point);
```

CubitInterface::Surface::normal_at
std::array< double, 3 > normal_at(std::array< double, 3 > location)
Get the normal at a particular point on the Surface.

```
norm = surface .normal_at ([0,0,0])
```

Parameters

[in] **location** A vector containing three values that are the coordinates of a point

Returns

A vector (or list) of doubles representing values of normal vector as follows:
0 - x value
1 - y value
2 - z value

### ◆ ordered_loops()

std::vector< std::vector< **Curve** > > ordered_loops( )

Get the ordered loops of the **Surface** .

```
std::vector<std::vector<Curve> > loops = surface ->ordered_loops
();
```

CubitInterface::Surface::ordered_loops
std::vector< std::vector< Curve > > ordered_loops()
Get the ordered loops of the Surface.

```
loops = surface .ordered_loops ()
```

Returns

A vector of vectors (or list of lists) of Curves in loops:
0, 0 - loop 1 curve 1
0, 1 - loop 1 curve 2
1, 0 - loop 2 curve 1
etc...

### ◆ point_containment()

int point_containment(std::array< double, 3 > *point_in*)

Get whether a point is on or off of the **Surface** .

```
std::array<double,3> point(3, 0);
int on_off = surface ->point_containment (point);
```

CubitInterface::Surface::point_containment
int point_containment(std::array< double, 3 > point_in)
Get whether a point is on or off of the Surface.

```
        on_off = surface .point_containment ([0,0,0])
```

Parameters

[in] **point_in** A vector containing three values that are the coordinates of a point

Returns

A python boolean representing whether the point is off (0) or on (1) the **Surface**

◆ position_from_u_v()

std::array< double, 3 > position_from_u_v ( double *u*, double *v* )

Get the Cartesian coordinates from the uv coordinates on the **Surface** .

```
        std::vector<double> pos = surface ->position_from_u_v (0, 0);
```

CubitInterface::Surface::position_from_u_v

std::array< double, 3 > position_from_u_v(double u, double v)

Get the Cartesian coordinates from the uv coordinates on the Surface.

```
        pos = surface .position_from_u_v (0, 0)
```

Parameters

[in] **u** The u parameter

[in] **v** The v parameter

Returns

The Cartesian coordinates of the supplied uv coordinates as a vector:
0 - x coordinate
1 - y coordinate
2 - z coordinate

◆ principal_curvatures()

std::array< double, 2 > principal_curvatures ( std::array< double, 3 > *point* )

Get the principal curvatures of the **Surface** .

```
        std::vector<double> point(3, 0);
        std::vector<double> curvatures = surface ->principal_curvatures
(point);
```

CubitInterface::Surface::principal_curvatures

std::array< double, 2 > principal_curvatures(std::array< double, 3 > point)

Get the principal curvatures of the Surface.

```
        curvatures = surface .principal_curvatures ([0,0,0])
```

Parameters

[in] **point** A vector containing three values that are the coordinates of a point

Returns

A list of two floats representing the curvatures
0 - curvature 1
1 - curvature 2

◆ set_color()

void set_color(std::array< double, 4 > *value*)

Set the color of the surface.

```
surface ->set_color (0);
```

CubitInterface::Surface::set_color
void set_color(std::array< double, 4 > value)
Set the color of the surface.

```
surface .set_color (0)
```

Parameters

[in] **value** The color value that the surface will have

◆ u_v_from_position()

std::array< double, 2 > u_v_from_position(std::array< double, 3 > *location*)

Get the uv coordinates from the supplied Cartesian coordinates on the **Surface** .

```
std::vector<double> location(3, 0);
std::vector<double> uv = surface ->u_v_from_position (location);
```

CubitInterface::Surface::u_v_from_position
std::array< double, 2 > u_v_from_position(std::array< double, 3 > location)
Get the uv coordinates from the supplied Cartesian coordinates on the Surface.

```
uv = surface .position_from_u_v ([0,0,0])
```

Parameters

[in] **location** A vector containing the Cartesian coordinates

Returns

The curvature values:
0 - The u parameter
1 - The v parameter

Cubit Python API 16.18

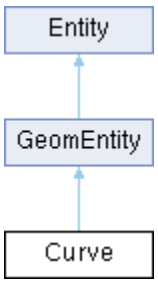
Public Member Functions

# Curve

Defines a curve object that mostly parallels Cubit's RefEdge class. More...

```
#include <CubitInterfaceEx.hpp>
```

Inheritance diagram for Curve:



## Public Member Functions

| | |
|---|---|
| | **Curve** () |
| | **Curve** (const **Curve** &other) |
| | **Curve** (CubitEntity ***entity_ptr**) |
| | **~Curve** () |
| std::array< double, 3 > | **closest_point** (std::array< double, 3 > point)<br>Get the curvature of the **Curve** at a particular point. |
| std::array< double, 3 > | **closest_point_trimmed** (std::array< double, 3 > point)<br>Get the curvature of the **Curve** at a particular point. |
| std::array< double, 4 > | **color** ()<br>Get the color of the **Curve**. |
| std::array< double, 3 > | **curvature** (std::array< double, 3 > point)<br>Get the curvature of the **Curve** at a particular point. |
| std::array< double, 3 > | **curve_center** ()<br>Get the center point of the **Curve**. |
| double | **end_param** ()<br>Get the highest value of the **Curve** in uv space. |
| double | **fraction_from_arc_length** (**Vertex** root_vertex, double **length**)<br>Get the fraction along the **Curve** a specified arc length is away from a given **Vertex**. |
| bool | **is_periodic** ()<br>Get whether the **Curve** is periodic or not. |
| double | **length** () |

| | |
|---|---|
| | Get the length of the **Curve**. |
| double | **length_from_u** (double parameter1, double parameter2)<br>Get the length between two specified parameters on a **Curve**. |
| std::array< double, 3 > | **point_from_arc_length** (double root_param, double arc_length)<br>Get the position on a **Curve** that is a specified arc length away from the specified root parameter. |
| std::array< double, 3 > | **position_from_fraction** (double fraction_along_curve)<br>Get the position of the point a specified fraction along the **Curve**. |
| std::array< double, 3 > | **position_from_u** (double u_value)<br>Get the position of a particular u value for the **Curve**. |
| void | **set_color** (std::array< double, 4 > value)<br>Set the color of the **Curve**. |
| double | **start_param** ()<br>Get the lowest value of the **Curve** in uv space. |
| std::array< double, 3 > | **tangent** (std::array< double, 3 > point)<br>Get the tangent to the **Curve** at a particular point. |
| double | **u_from_arc_length** (double root_param, double arc_length)<br>Get the u value for a point a specified arc length away from a specified root parameter on the **Curve**. |
| double | **u_from_position** (std::array< double, 3 > position)<br>Get the u value of a particular position on the **Curve**. |

**Public Member Functions inherited from GeomEntity**

| | |
|---|---|
| std::vector< **Body** > | **bodies** () |

| | |
|---|---|
| | Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** ()<br>Get the curves in the **GeomEntity**. |
| int | **dimension** ()<br>Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** ()<br>Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** ()<br>Return the current mesh state of the **GeomEntity**. |
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |

| | |
|---|---|
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |
| void | **remove_entity_names** ()<br>Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| void | **set_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

▸ **Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Additional Inherited Members

**Protected Member Functions inherited from GeomEntity**

| | |
|---|---|
| | **GeomEntity** (const **GeomEntity** &other) |
| | **GeomEntity** (CubitEntity ***entity_ptr**) |

**Protected Attributes inherited from Entity**

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

Defines a curve object that mostly parallels Cubit's RefEdge class.

## Constructor & Destructor Documentation

◆ Curve() [1/3]

**Curve** ( ) inline

◆ ~Curve()

~**Curve** ( ) inline

◆ Curve() [2/3]

**Curve** ( const **Curve** & *other* ) inline

◆ Curve() [3/3]

**Curve** ( CubitEntity * *entity_ptr* ) inline

## Member Function Documentation

◆ closest_point()

std::array< double, 3 > closest_point ( std::array< double, 3 > *point* )

Get the curvature of the **Curve** at a particular point.

```
std::vector<double> point(3, 0);
std::vector<double> close = curve ->closest_point (point);
```

CubitInterface::Curve::closest_point
std::array< double, 3 > closest_point(std::array< double, 3 > point)
Get the curvature of the Curve at a particular point.
CubitInterface::curve
CubitInterface::Curve curve(int id_in)
Gets the curve object from an ID.

```
close = curve .closest_point ([0,0,0])
```

Parameters

[in] **point** A vector containing 3 doubles representing coordinates of a location on the **Curve**

Returns

The closest point to the **Curve** from the location specified

◆ closest_point_trimmed()

std::array< double, 3 > closest_point_trimmed ( std::array< double, 3 > *point* )

Get the curvature of the **Curve** at a particular point.

```
std::vector<double> point(3, 0);
std::vector<double> close = curve ->closest_point (point);
close = curve .closest_point ([0,0,0])
```

Parameters

[in] **point** A vector containing 3 doubles representing coordinates of a location on the **Curve**

Returns

The closest point to the **Curve** from the location specified

### ◆ color()

std::array< double, 4 > color ( )

Get the color of the **Curve** .

```
int col = curve ->color ();
```

CubitInterface::Curve::color
std::array< double, 4 > color()
Get the color of the Curve.

```
col = curve .color ()
```

Returns

The color value associated with the curve's current color

### ◆ curvature()

std::array< double, 3 > curvature ( std::array< double, 3 > *point* )

Get the curvature of the **Curve** at a particular point.

```
std::vector<double> point(3, 0);
std::vector<double> curvature = curve ->curvature (point);
```

CubitInterface::Curve::curvature
std::array< double, 3 > curvature(std::array< double, 3 > point)
Get the curvature of the Curve at a particular point.

```
curvature = curve .curvature ([0,0,0])
```

Parameters

[in] **point** A vector containing 3 doubles representing coordinates of a location on the **Curve**

Returns

The curvature of the **Curve** at the location specified

### ◆ curve_center()

std::array< double, 3 > curve_center ( )

Get the center point of the **Curve** .

```
std::array<double,3> center = curve ->curve_center ();
```

CubitInterface::Curve::curve_center
std::array< double, 3 > curve_center()
Get the center point of the Curve.

```
center = curve .curve_center ()
```

Returns

A vector containing the coordinates of the **Curve**'s center according to the following:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ end_param()

double end_param ( )

Get the highest value of the **Curve** in uv space.

```
double end = curve ->end_param ();
```

CubitInterface::Curve::end_param
double end_param()
Get the highest value of the Curve in uv space.

```
end = curve .end_param ()
```

Returns
The ending value of the parameter

### ◆ fraction_from_arc_length()

double fraction_from_arc_length ( **Vertex** *root_vertex*,
double *length*
)

Get the fraction along the **Curve** a specified arc length is away from a given **Vertex** .

```
double fraction = curve ->fraction_from_arc_length (vertex , 0.5);
```

CubitInterface::Curve::fraction_from_arc_length
double fraction_from_arc_length(Vertex root_vertex, double length)
Get the fraction along the Curve a specified arc length is away from a given Vertex.
CubitInterface::vertex
CubitInterface::Vertex vertex(int id_in)
Gets the vertex object from an ID.

```
fraction = curve .fraction_from_arc_length (vertex , 0.5)
```

Parameters

[in] **root_vertex** The **Vertex** to start from (vertex object)
[in] **length** The length along the **Curve** away from the root **Vertex**

Returns
The fraction of the **Curve** that is the specified length away from the specified **Vertex**

### ◆ is_periodic()

bool is_periodic ( )

Get whether the **Curve** is periodic or not.

```
bool periodic = curve ->is_periodic ();
```

CubitInterface::Curve::is_periodic
bool is_periodic()
Get whether the Curve is periodic or not.

```
periodic = curve .is_periodic ()
```

Returns
Whether the **Curve** is periodic or not

### ◆ length()

double length ( )

Get the length of the **Curve** .

```
        double len = curve ->length ();
```

CubitInterface::Curve::length

double length()

Get the length of the Curve.

```
        len = curve .length ()
```

Returns

The length of the **Curve**

◆ length_from_u()

double length_from_u(double *parameter1*,
double *parameter2*
)

Get the length between two specified parameters on a **Curve** .

```
        doublelength = curve ->length_from_u (0, 0.5);
```

CubitInterface::Curve::length_from_u

double length_from_u(double parameter1, double parameter2)

Get the length between two specified parameters on a Curve.

```
        length = curve .length_from_u (0, 0.5)
```

Parameters

[in] **parameter1** The beginning parameter

[in] **parameter2** The ending parameter

Returns

The length between the two specified paramters along the **Curve**

◆ point_from_arc_length()

std::array< double, 3 > point_from_arc_length(double *root_param*,
double *arc_length*
)

Get the position on a **Curve** that is a specified arc length away from the specified root parameter.

```
        std::vector<double> position = curve ->point_from_arc_length (0,
0.5);
```

CubitInterface::Curve::point_from_arc_length

std::array< double, 3 > point_from_arc_length(double root_param, double arc_length)

Get the position on a Curve that is a specified arc length away from the specified root parameter.

```
        position = curve .point_from_arc_length (0, 0.5)
```

Parameters

[in] **root_param** The root parameter from which the arc length is added to

[in] **arc_length** The arc length along the **Curve** away from the root parameter

Returns

A vector that contains the coordinates of a position a specified arc length away from the root parameter

### ◆ position_from_fraction()

std::array< double, 3 > position_from_fraction(double *fraction_along_curve*)

Get the position of the point a specified fraction along the **Curve** .

```
std::array<double,3> pos = curve ->position_from_fraction (0.5);
```

CubitInterface::Curve::position_from_fraction
std::array< double, 3 > position_from_fraction(double fraction_along_curve)
Get the position of the point a specified fraction along the Curve.

```
pos = curve .position_from_fraction (0.5)
```

Parameters

[in] **fraction_along_curve** A decimal value between 0 and 1 to determine a particular position along the **Curve**

Returns

A vector containing the coordinates of the position a specified fraction along the **Curve**:
0 - x coordinate
1 - y coordinate
2 - z coordinate

### ◆ position_from_u()

std::array< double, 3 > position_from_u(double *u_value*)

Get the position of a particular u value for the **Curve** .

```
std::vector<double> position = curve ->position_from_u (0.5);
```

CubitInterface::Curve::position_from_u
std::array< double, 3 > position_from_u(double u_value)
Get the position of a particular u value for the Curve.

```
position = curve .position_from_u (0.5)
```

Parameters

[in] **u_value** The u value of the position along the **Curve**

Returns

A vector containing the coordinates of the output position

### ◆ set_color()

void set_color(std::array< double, 4 > *value*)

Set the color of the **Curve** .

```
curve ->set_color (0);
```

CubitInterface::Curve::set_color
void set_color(std::array< double, 4 > value)
Set the color of the Curve.

```
curve .set_color (0)
```

Parameters

[in] **value** The color value that the curve will have

◆ start_param()

double start_param ( )

Get the lowest value of the **Curve** in uv space.

```
double start = curve ->start_param ();
```

CubitInterface::Curve::start_param
double start_param()
Get the lowest value of the Curve in uv space.

```
start = curve .start_param ()
```

Returns

The beginning value of the parameter

◆ tangent()

std::array< double, 3 > tangent ( std::array< double, 3 > *point* )

Get the tangent to the **Curve** at a particular point.

```
std::vector<double> point(3, 0);
std::vector<double> tan = curve ->tangent (point);
```

CubitInterface::Curve::tangent
std::array< double, 3 > tangent(std::array< double, 3 > point)
Get the tangent to the Curve at a particular point.

```
tan = curve .tangent ([0,0,0])
```

Parameters

[in] **point** A vector containing 3 doubles representing coordinates of a location on the **Curve**

Returns

The tangent to the **Curve** at the location specified

◆ u_from_arc_length()

double u_from_arc_length ( double *root_param*,
double *arc_length*
)

Get the u value for a point a specified arc length away from a specified root parameter on the **Curve** .

```
double u = curve ->u_from_arc_length (0, 0.5);
```

CubitInterface::Curve::u_from_arc_length
double u_from_arc_length(double root_param, double arc_length)
Get the u value for a point a specified arc length away from a specified root parameter on the Curve.

```
u = curve .u_from_arc_length (0, 0.5)
```

Parameters

[in] **root_param** The beginning parameter from which the arc length is added to

[in] **arc_length** The length away from the root parameter of the output parameter

Returns

The u value of the **Curve** the arc length away from the root parameter

◆ u_from_position()

double u_from_position (std::array< double, 3 > *position*)

Get the u value of a particular position on the **Curve** .

```
std::vector<double> point(3, 0);
double u = curve ->u_from_position (point);
```

CubitInterface::Curve::u_from_position

double u_from_position(std::array< double, 3 > position)

Get the u value of a particular position on the Curve.

```
u = curve .u_from_position ([0,0,0])
```

Parameters

[in] **position** A vector containing the coordinates of the input position

Returns

The u value of the position along the **Curve**



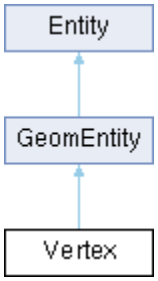
Public Member Functions

# Vertex

Defines a vertex object that mostly parallels Cubit's RefVertex class. More...

`#include <CubitInterfaceEx.hpp>`

Inheritance diagram for Vertex:




Public Member Functions

**Vertex** ()

**Vertex** (const **Vertex** &other)

**Vertex** (CubitEntity ***entity_ptr**)

| | |
|---|---|
| | **~Vertex** () |
| std::array< double, 4 > | **color** ()<br>Get the color of the **Vertex**. |
| std::array< double, 3 > | **coordinates** ()<br>Get the Cartesian coordinates of the **Vertex**. |
| **Vertex** & | **operator=** (const **Vertex** &other) |
| void | **set_color** (std::array< double, 4 > value)<br>Set the color of the **Vertex**. |

**Public Member Functions inherited from GeomEntity**

| | |
|---|---|
| std::vector< **Body** > | **bodies** ()<br>Get the bodies in the **GeomEntity**. |
| std::vector< **Curve** > | **curves** ()<br>Get the curves in the **GeomEntity**. |
| int | **dimension** ()<br>Get the dimensions of the **GeomEntity**. |
| std::string | **entity_name** ()<br>Return the first name of the **GeomEntity**. |
| void | **entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| std::vector< std::string > | **entity_names** ()<br>Return the all the names of the **GeomEntity**. |
| bool | **is_meshed** ()<br>Return the current mesh state of the **GeomEntity**. |

| | |
|---|---|
| int | **is_transparent** ()<br>Get the tranparency state of the **Entity**. |
| void | **is_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| int | **is_visible** ()<br>Get the visibility state of the **Entity**. |
| void | **is_visible** (bool visibility_flag)<br>Set the visibility state of the **Entity**. |
| void | **mesh** ()<br>Mesh the **GeomEntity**. |
| int | **num_names** ()<br>Return the number of names for the **GeomEntity**. |
| void | **remove_entity_name** (std::string name)<br>Remove a specific name from the list of names assigned to the **GeomEntity**. |
| void | **remove_entity_names** ()<br>Remove all the names assigned to the **GeomEntity**. |
| void | **remove_mesh** ()<br>Removes the mesh on the **GeomEntity**. |
| void | **set_entity_name** (std::string name)<br>Assign a name to the **GeomEntity**. |
| void | **set_transparent** (int transparency_flag)<br>Set the tranparency state of the **Entity**. |
| void | **set_visible** (bool visibility_flag) |

| | |
|---|---|
| | Set the visibility state of the **Entity**. |
| void | **smooth** ()<br>Smooths the mesh on the **GeomEntity**. |
| std::vector< **Surface** > | **surfaces** ()<br>Get the surfaces in the **GeomEntity**. |
| std::vector< **Vertex** > | **vertices** ()<br>Get the vertices in the **GeomEntity**. |
| std::vector< **Volume** > | **volumes** ()<br>Get the volumes in the **GeomEntity**. |

**Public Member Functions inherited from Entity**

| | |
|---|---|
| | **Entity** () |
| | **Entity** (CubitEntity ***entity_ptr**) |
| | **Entity** (**Entity** const &copy_from) |
| | **~Entity** () |
| std::array< double, 6 > | **bounding_box** ()<br>Get the bounding box of the **Entity**. |
| std::array< double, 3 > | **center_point** ()<br>Get the center point of the **Entity**. |
| void | **destroy_cubit_entity** () |
| CubitEntity * | **entity_ptr** () |
| int | **id** ()<br>Get the id of the **Entity**. |

| | |
|---|---|
| **Entity** & | **operator=** (const **Entity** &rhs) |

## Additional Inherited Members

**Protected Member Functions inherited from GeomEntity**

| | |
|---|---|
| | **GeomEntity** (const **GeomEntity** &other) |
| | **GeomEntity** (CubitEntity *__entity_ptr__) |

**Protected Attributes inherited from Entity**

| | |
|---|---|
| CubObserver * | **cubitWatcher** |
| CubitEntity * | **mEntityPtr** |

## Detailed Description

Defines a vertex object that mostly parallels Cubit's RefVertex class.

## Constructor & Destructor Documentation

### ◆ Vertex() [1/3]

**Vertex** ( ) inline

### ◆ ~Vertex()

~**Vertex** ( ) inline

### ◆ Vertex() [2/3]

**Vertex** ( const **Vertex** & *other* ) inline

### ◆ Vertex() [3/3]

**Vertex** ( CubitEntity * *entity_ptr* ) inline

## Member Function Documentation

### ◆ color()

std::array< double, 4 > color ( )

Get the color of the **Vertex** .

```
int col = vertex ->color ();
```

CubitInterface::Vertex::color
std::array< double, 4 > color()
Get the color of the Vertex.
CubitInterface::vertex
CubitInterface::Vertex vertex(int id_in)
Gets the vertex object from an ID.

```
col = vertex .color ()
```

Returns
The color value associated with the vertex's current color

◆ coordinates()

std::array< double, 3 > coordinates ( )

Get the Cartesian coordinates of the **Vertex** .

```
std::vector<double> position = vertex ->coordinates ();
```

CubitInterface::Vertex::coordinates
std::array< double, 3 > coordinates()
Get the Cartesian coordinates of the Vertex.

```
position = vertex .coordinates ()
```

Returns
A vector containing the coordinates of the **Vertex** with indices corresponding to the coordinates as follows:
0 - x coordinate
1 - y coordinate
2 - z coordinate

◆ operator=()

**Vertex** & operator= ( const **Vertex** & *other* ) inline

◆ set_color()

void set_color ( std::array< double, 4 > *value* )

Set the color of the **Vertex** .

```
vertex ->set_color (0);
```

CubitInterface::Vertex::set_color
void set_color(std::array< double, 4 > value)
Set the color of the Vertex.

```
vertex .set_color (0)
```

Parameters

[in] **value** The color value that the vertex will have



Public Member Functions

# CubitFailureException

An exception class to alert the caller when the underlying Cubit function fails. More...

```
#include <CubitInterfaceEx.hpp>
```

Inheritance diagram for CubitFailureException:




## Public Member Functions

const char * **what** () const throw ()

## Detailed Description

An exception class to alert the caller when the underlying Cubit function fails.

## Member Function Documentation

◆ what()

const char * what ( ) const
throw ( )

inline



Public Member Functions

# InvalidEntityException

An exception class to alert the caller that an invalid entity was attempted to be used. Likely the user is attempting to use an **Entity** who's underlying CubitEntity has been deleted. More...

`#include <CubitInterfaceEx.hpp>`

Inheritance diagram for InvalidEntityException:




## Public Member Functions

const char * **what** () const throw ()

## Detailed Description

An exception class to alert the caller that an invalid entity was attempted to be used. Likely the user is attempting to use an **Entity** who's underlying CubitEntity has been deleted.

## Member Function Documentation

◆ what()

const char * what ( ) const
throw ( )

inline

)

Cubit Python API 16.18

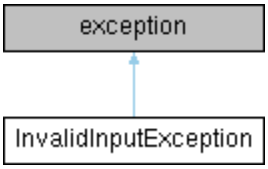
Public Member Functions

# InvalidInputException

An exception class to alert the caller of a function that invalid inputs were entered. More...

`#include <CubitInterfaceEx.hpp>`

Inheritance diagram for InvalidInputException:




## Public Member Functions

const char * **what** () const throw ()

## Detailed Description

An exception class to alert the caller of a function that invalid inputs were entered.

## Member Function Documentation

◆ what()

const char * what ( ) const
throw ( )

inline

Cubit Python API 16.18

Public Member Functions

# MeshErrorFeedback

Class to implement mesh command feedback processing. More...

`#include <CubitInterface.hpp>`

## Public Member Functions

**MeshErrorFeedback** ()

**~MeshErrorFeedback** ()

| | |
|---|---|
| int | **get_entity_id** () |
| std::string | **get_entity_type** () |
| int | **get_error_code** () |
| std::string | **get_error_text** () |
| void | **set_entity_id** (int anid) |
| void | **set_entity_type** (std::string atype) |
| void | **set_error_code** (int acode) |
| void | **set_error_text** (std::string atext) |

## Detailed Description

Class to implement mesh command feedback processing.

## Constructor & Destructor Documentation

### ◆ MeshErrorFeedback()

**MeshErrorFeedback** ( ) inline

### ◆ ~MeshErrorFeedback()

~**MeshErrorFeedback** ( ) inline

## Member Function Documentation

### ◆ get_entity_id()

int get_entity_id ( ) inline

### ◆ get_entity_type()

std::string get_entity_type ( ) inline

### ◆ get_error_code()

int get_error_code ( ) inline

### ◆ get_error_text()

std::string get_error_text ( ) inline

### ◆ set_entity_id()

void set_entity_id ( int *anid* ) inline

◆ set_entity_type()

void set_entity_type ( std::string *atype* ) inline

◆ set_error_code()

void set_error_code ( int *acode* ) inline

◆ set_error_text()

void set_error_text ( std::string *atext* ) inline



Functions

# CubitMeshInterface Namespace Reference

This document explains how to access Cubit mesh data from C++ or python. More...

## Functions

| | |
|---|---|
| void | **get_elem_connectivity** (DLIList< MRefEntity * > &mref_list, int *elem_connectivity)<br>Get the element connectivity. |
| void | **get_mesh_data_size** (DLIList< MRefEntity * > &mref_list, int &num_node, int &num_elem, int &num_node_per_elem, int &connectivity_size)<br>Get the size of mesh data to be returned from an MRefEntity. |
| void | **get_node_coordinates** (DLIList< MRefEntity * > &mref_list, double *node_data)<br>Get the node coordinate data. |
| void | **get_side_and_centroid_data** (MRefEntity *mref_entity, MRefEntity *mref_parent, std::vector< **AdjacencyInfo** > &adj_info_list) |
| void | **init_mesh_access** ()<br>Use init_mesh_access to initialize mesh data access. |

## Detailed Description

This document explains how to access Cubit mesh data from C++ or python.
The purpose of **CubitMeshInterface** is to provide developers with a query interface for Cubit mesh data.
Most function prototypes are easy to understand. A few are more ambiguous and for those some examples are provided. Note that input parameters that require entity types, such as "volume", "quad", "curve", and so forth, expect lower case spellings of those entity types.
Examples are provided for C++ and python developers.

## Function Documentation

### ◆ get_elem_connectivity()

```
void get_elem_connectivity ( DLIList< MRefEntity * > &  mref_list,
                             int *                      elem_connectivity
                           )
```

Get the element connectivity.
Parameters

| | |
|---|---|
| **mref** | Specifies the geometry entity for which the mesh data is to be queried |
| **elem_connectivity** | Pointer to memory where element connectivity will be written. Data is written to the elem_connectivity array as: number of nodes in element 1 1st node index in element 1 2nd node index in element 1 3rd node index in element 1 ... nth node index in element 1 number of nodes in element 2 1st node index in element 2 2nd node index in element 2 3rd node index in element 2 ... nth node index in element 2 number of nodes in element 3 ... |

### ◆ get_mesh_data_size()

```
void get_mesh_data_size ( DLIList< MRefEntity * > &  mref_list,
                          int &                      num_node,
                          int &                      num_elem,
                          int &                      num_node_per_elem,
                          int &                      connectivity_size
                        )
```

Get the size of mesh data to be returned from an MRefEntity.
Parameters

| | |
|---|---|
| **mref** | Specifies the geometry entity for which the mesh data is to be queried |
| **num_node** | The number of nodes in the mesh |
| **num_elem** | The number of elements in the mesh |
| **num_node_per_elem** | The number of nodes per element |
| **connectivity_size** | returns the number of data entries that will be written to the elem_connectivity array in **get_elem_connectivity()**. |

This gives the size of memory to be allocated for elem_connectivity before calling **get_elem_connectivity()**.

◆ get_node_coordinates()

void get_node_coordinates ( DLIList< MRefEntity * > & *mref_list*,
double * *node_data*
)

Get the node coordinate data.
Parameters

**mref** Specifies the geometry entity for which the mesh data is to be queried
**node_data** Pointer to memory where node coordinates will be written. Allocated size must be at least 3 * number of nodes in the mesh

◆ get_side_and_centroid_data()

void get_side_and_centroid_data ( MRefEntity * *mref_entity*,
MRefEntity * *mref_parent*,
std::vector< **AdjacencyInfo** > & *adj_info_list*
)

◆ init_mesh_access()

void init_mesh_access ( )

Use init_mesh_access to initialize mesh data access.



Functions

# CubitModifyInterface Namespace Reference

This document explains how to access Cubit mesh data from C++ or python. More...

## Functions

| | |
|---|---|
| void | **clear_window** ()<br>clear the graphics window |
| std::vector< int > | **create_free_nodes** (std::vector< double > &coords)<br>create free nodes in cubit |
| std::vector< int > | **create_free_tris** (std::vector< int > &tri_nodes) |

| | |
|---|---|
| | create free tris in cubit |
| void | **draw_line** (double x0, double y0, double z0, double x1, double y1, double z1, int color) |
| void | **draw_point** (double x, double y, double z, int color) |
| void | **flush** () |
| void | **graphics_transforms** ()<br>if a cubit graphics window is currently active, allows for interactive transformations of the model with the mouse |
| void | **init** ()<br>Use init to initialize cubit modify access. |
| void | **notify_mesh_modified** () |
| int | **set_node_coordinates** (int num_nodes, int *nodes, double *coords)<br>set the coordinate values for nodes |
| bool | **transfer_mesh** (const std::string &geom_type, int from_entity_id, int to_entity_id, bool snap_to)<br>transfer the ownership of all owned mesh entities from one geometry entity to another. transfers onlh mesh entities owned exclusively by the geometry entity. For example only nodes on the interior of a volume will be transferred to a new volume. Use a bottom up approach to successively transfer mesh from vertices, curves, surfaces and then volumes. If not performed completely, can leave the mesh in an invalid state |

## Detailed Description

This document explains how to access Cubit mesh data from C++ or python.
The purpose of **CubitMeshInterface** is to provide developers with a query interface for Cubit mesh data.
Most function prototypes are easy to understand. A few are more ambiguous and for those some examples are provided. Note that input parameters that require entity types, such as "volume", "quad", "curve", and so forth, expect lower case spellings of those entity types.

Examples are provided for C++ and python developers.

## Function Documentation

### ◆ clear_window()

```
void clear_window ( )
```

clear the graphics window

### ◆ create_free_nodes()

```
std::vector< int > create_free_nodes ( std::vector< double > & coords )
```

create free nodes in cubit

Parameters

**coords** vector of node locations, ordered x,y,z for each node

Returns

ids of new nodes created (in same order as coords array)

### ◆ create_free_tris()

```
std::vector< int > create_free_tris ( std::vector< int > & tri_nodes )
```

create free tris in cubit

Parameters

**tri_nodes** vector of node ids, ordered ccw for each tri

Returns

ids of new tris created (in same order as tri_nodes array)

### ◆ draw_line()

```
void draw_line ( double x0,
                 double y0,
                 double z0,
                 double x1,
                 double y1,
                 double z1,
                 int    color
               )
```

brief draw a line in space in the graphics window

### ◆ draw_point()

```
void draw_point ( double x,
                  double y,
                  double z,
                  int    color
                )
```

brief draw a point in space in the graphics window

### ◆ flush()

void flush ( )

brief flush the graphics pipeline

### ◆ graphics_transforms()

void graphics_transforms ( )

if a cubit graphics window is currently active, allows for interactive transformations of the model with the mouse

### ◆ init()

void init ( )

Use init to initialize cubit modify access.

### ◆ notify_mesh_modified()

void notify_mesh_modified ( )

brief updates all entities in cubit DB (including graphics) after the mesh has been modified

### ◆ set_node_coordinates()

```
int set_node_coordinates ( int        num_nodes,
                           int *      nodes,
                           double *   coords
                         )
```

set the coordinate values for nodes
Parameters

| | |
|---|---|
| **num_nodes** | number of nodes we are passing in |
| **nodes** | array of node ids (array size = num_nodes) |
| **coords** | array of x-y-z coordinates (array size = 3*num_nodes) |

Returns

The number of successful node-coordinate asignments (should be the same as num_nodes)

### ◆ transfer_mesh()

```
bool transfer_mesh ( const std::string & geom_type,
                     int                 from_entity_id,
                     int                 to_entity_id,
                     bool                snap_to
                   )
```

transfer the ownership of all owned mesh entities from one geometry entity to another. transfers onlh mesh entities owned exclusively by the geometry entity. For example only nodes on the interior of a volume will be transferred to a new volume. Use a

bottom up approach to successively transfer mesh from vertices, curves, surfaces and then volumes. If not performed completely, can leave the mesh in an invalid state

```
CubitModifyInterface::transfer_mesh ("surface" , 12, 24, true );
```

CubitModifyInterface::transfer_mesh
bool transfer_mesh(const std::string &geom_type, int from_entity_id, int to_entity_id, bool snap_to)
transfer the ownership of all owned mesh entities from one geometry entity to another....

Parameters

| | |
|---|---|
| **geom_type** | Specifies the geometry type of the entity |
| **from_entity_id** | id of an existing cubit entity of type geom_type that currently contains mesh entities. entity will no longer me "meshed" following this operation |
| **to_entity_id** | id of an existing cubit entity of type geom_type that does not currently meshed. entity will be meshed following this operation. |
| **snap_to** | project any nodes to the gometry entity with id to_entity_id |

Returns
whether the transfer was successful



Public Member Functions | Public Attributes

# AssemblyItem

Class to implement assembly tree interface. More...
`#include <CubitInterface.hpp>`

## Public Member Functions

| | |
|---|---|
| | **AssemblyItem** () |
| | **~AssemblyItem** () |
| int | **get_id** () |
| int | **get_instance** () |
| int | **get_level** () |
| std::string | **get_name** () |
| std::string | **get_path** () |

| | |
|---|---|
| std::string | **get_type** () |
| void | **set_id** (int aid) |
| void | **set_instance** (int ainstance) |
| void | **set_level** (int alevel) |
| void | **set_name** (std::string aname) |
| void | **set_path** (std::string apath) |
| void | **set_type** (std::string atype) |

## Public Attributes

| | |
|---|---|
| std::vector< int > | **volume_id_list** |

## Detailed Description

Class to implement assembly tree interface.

## Constructor & Destructor Documentation

### ◆ AssemblyItem()

**AssemblyItem** ( ) inline

### ◆ ~AssemblyItem()

~**AssemblyItem** ( ) inline

## Member Function Documentation

### ◆ get_id()

int get_id ( ) inline

### ◆ get_instance()

int get_instance ( ) inline

### ◆ get_level()

int get_level ( ) inline

### ◆ get_name()

std::string get_name ( ) inline

- get_path()

  std::string get_path ( ) inline

- get_type()

  std::string get_type ( ) inline

- set_id()

  void set_id ( int *aid* ) inline

- set_instance()

  void set_instance ( int *ainstance* ) inline

- set_level()

  void set_level ( int *alevel* ) inline

- set_name()

  void set_name ( std::string *aname* ) inline

- set_path()

  void set_path ( std::string *apath* ) inline

- set_type()

  void set_type ( std::string *atype* ) inline

## Member Data Documentation

- volume_id_list

  std::vector<int> volume_id_list



Public Member Functions

# CFD_BC_Entity

Class to implement cfd bc data retrieval. More...

```
#include <CubitInterface.hpp>
```

## Public Member Functions

**CFD_BC_Entity** ()

**~CFD_BC_Entity** ()

| | |
|---|---|
| int | **get_id** () |
| std::string | **get_name** () |
| std::string | **get_type** () |
| void | **set_id** (int id) |
| void | **set_name** (std::string name) |
| void | **set_type** (std::string type) |

## Detailed Description

Class to implement cfd bc data retrieval.

## Constructor & Destructor Documentation

### ◆ CFD_BC_Entity()

**CFD_BC_Entity** ( ) inline

### ◆ ~CFD_BC_Entity()

~**CFD_BC_Entity** ( ) inline

## Member Function Documentation

### ◆ get_id()

int get_id ( ) inline

### ◆ get_name()

std::string get_name ( ) inline

### ◆ get_type()

std::string get_type ( ) inline

### ◆ set_id()

void set_id ( int *id* ) inline

### ◆ set_name()

void set_name ( std::string *name* ) inline

### ◆ set_type()

void set_type ( std::string *type* ) inline

Cubit Python API 16.18

Public Member Functions | Public Attributes

# Dir

Defines a direction object. More...

`#include <CubitInterfaceEx.hpp>`

## Public Member Functions

| | |
|---|---|
| | **Dir** (double **x**, double **y**, double **z**) |
| | **~Dir** () |
| double | **angle** (**Dir** vec_in)<br>Returns the interior of two vectors. |
| **Dir** | **cross** (**Dir** vec2)<br>Returns the cross product of thisXvec2. |
| void | **dir_print** ()<br>Print the output. |
| double | **distance** (**Dir** vec_in)<br>get the distance between two points |
| double | **dot** (**Dir** vec2)<br>Returns the dot product. |
| std::array< double, 3 > | **get_xyz** ()<br>Get an array of the vector. |
| double | **length** ()<br>Returns the length. |
| void | **normalize** ()<br>Normalize 'this' vector. |

| | |
|---|---|
| std::vector< **CubitInterface::Dir** > | **orthogonal_vectors** ()<br>Finds 2 (arbitrary) vectors that are orthogonal to this one. |
| void | **set** (double **x**, double **y**, double **z**)<br>Set the xyz values of the vector. |
| void | **set_x** (double x_in)<br>Set the x location of the point. |
| void | **set_y** (double y_in)<br>Set the y location of the point. |
| void | **set_z** (double z_in)<br>Set the z location of the point. |
| double | **x** ()<br>Get the x location of the point. |
| double | **y** ()<br>Get the y location of the point. |
| double | **z** ()<br>Get the z location of the point. |

## Public Attributes

| | |
|---|---|
| double | **xVal** |
| double | **yVal** |
| double | **zVal** |

## Detailed Description

Defines a direction object.

## Constructor & Destructor Documentation

- Dir()

  **Dir** ( double *x*,
  double *y*,
  double *z*
  )

  inline

- ~Dir()

  ~**Dir** ( )

  inline

## Member Function Documentation

- angle()

  double angle ( **Dir** *vec_in* )

Returns the interior of two vectors.
return the angle is in radians

- cross()

  **Dir** cross ( **Dir** *vec2* )

Returns the cross product of thisXvec2.
Returns
cross product

- dir_print()

  void dir_print ( )

Print the output.

- distance()

  double distance ( **Dir** *vec_in* )

get the distance between two points

- dot()

  double dot ( **Dir** *vec2* )

Returns the dot product.

- get_xyz()

  std::array< double, 3 > get_xyz ( )

Get an array of the vector.

- length()

  double length ( )

Returns the length.

### ◆ normalize()

void normalize ( )

Normalize 'this' vector.
Returns
void

### ◆ orthogonal_vectors()

std::vector< **CubitInterface::Dir** > orthogonal_vectors ( )

Finds 2 (arbitrary) vectors that are orthogonal to this one.

### ◆ set()

void set ( double *x*,
double *y*,
double *z*
)

Set the xyz values of the vector.
Returns
void

### ◆ set_x()

void set_x ( double *x_in* )

Set the x location of the point.
Parameters

**x_in** the new x value

### ◆ set_y()

void set_y ( double *y_in* )

Set the y location of the point.
Parameters

**y_in** the new y value

### ◆ set_z()

void set_z ( double *z_in* )

Set the z location of the point.
Parameters

**z_in** the new z value

### ◆ x()

double x ( )

Get the x location of the point.

Returns
x value
See also
**y()**, **z()**

◆ y()

double y ( )

Get the y location of the point.
Returns
y value
See also
**x()**, **z()**

◆ z()

double z ( )

Get the z location of the point.
Returns
z value
See also
**y()**, **x()**

## Member Data Documentation

◆ xVal

double xVal

◆ yVal

double yVal

◆ zVal

double zVal





# Loc

Defines a location object. More...
`#include <CubitInterfaceEx.hpp>`

## Public Member Functions

**Loc** (double x, double y, double z)

**~Loc** ()

## Public Attributes

double **xVal**

double **yVal**

double **zVal**

## Detailed Description

Defines a location object.

## Constructor & Destructor Documentation

### ◆ Loc()

```
Loc ( double x,
      double y,
      double z
    )                                                    inline
```

### ◆ ~Loc()

```
~Loc ( )                                                 inline
```

## Member Data Documentation

### ◆ xVal

double xVal

### ◆ yVal

double yVal

### ◆ zVal

double zVal



Public Attributes

# AdjacencyInfo

`#include <CubitMeshInterface.hpp>`

## Public Attributes

CubitVector **Centroid**

| | |
|---|---|
| int | **ParentId** |
| int | **ParentSide** |

### Member Data Documentation

- Centroid

  CubitVector Centroid

- ParentId

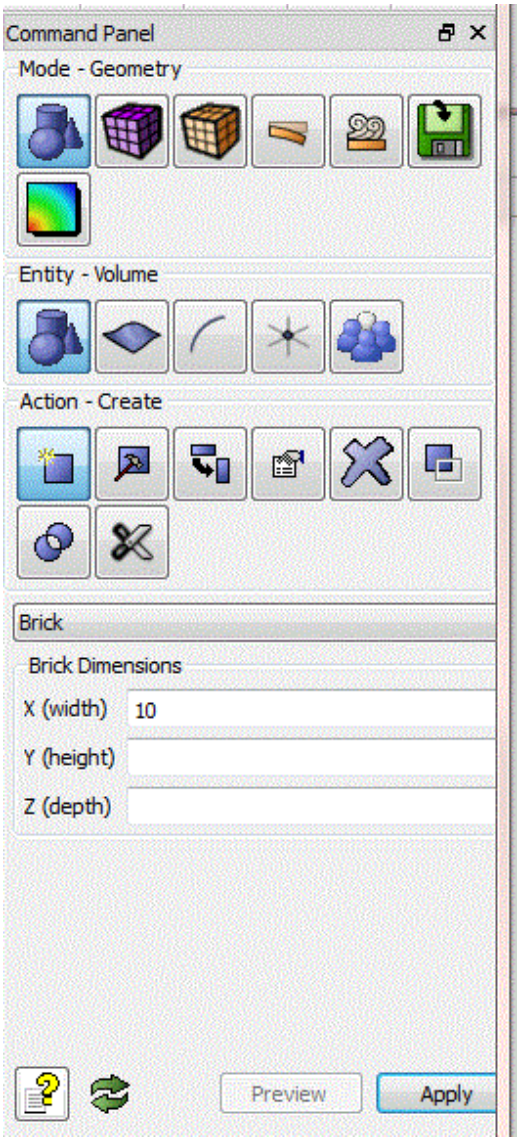
  int ParentId

- ParentSide

  int ParentSide

# Navigation XML Files

The Cubit GUI includes a section referred to as the Command Panel. It is comprised of a hierarchy of buttons used to navigate to panels that accept user input and generate Cubit command strings. The following example shows the command panel used to create a brick. The user navigates to the command panel by pressing the "**Mode - Geometry**" button, then the "**Entity - Volume**" button, followed by the "**Action - Create**" button, then finally selecting the "**Brick**" option from the pull-down menu.




Before Cubit 14.0, this hierarchy was not modifiable by any third party. With the release of Cubit 14.0, any user can modify the contents of the button hierarchy by adding, deleting, or modifying buttons and command panels. The button hierarchy is expressed in a series of XML files located in the directory 'bin/xml.'

The controlling XML file is named, "CubitNavigationRoot.xml." A snippet from the file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<Navigation version="1.1">
  <NavigationNode title="Entity" name="Geometry">
    <Label>Geometry</Label>
    <ToolTip>Manage Geometry</ToolTip>
    <Icon url=":/clarogui/images/i32geometry_cl.png"/>
    <NavigationNode name="Volume">
      <Label>Volume</Label>
      <ToolTip>Manage Volumes</ToolTip>
      <Icon url=":/cubitgui/i32gvolume.png"/>
    </NavigationNode>
    <NavigationNode name="Surface">
      <Label>Surface</Label>
      <ToolTip>Manage Surfaces</ToolTip>
      <Icon url=":/cubitgui/i32gsurface.png"/>
    </NavigationNode>
    <NavigationNode name="Curve">
```

The first two levels of the hierarchy are managed in this file. Subsequent levels of the hierarchy are managed in more specific XML files. For example, the remaining hierarchy associated with geometry volumes is managed in the file named, "GeometryVolumeNavigation.xml." A snippet from that file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<Navigation version="1.1">
  <NavigationReference path="Geometry/Volume">
    <NavigationNode marker="CreateGeometryVolume" name="Create">
      <Label>Create</Label>
      <ToolTip>Create volumes</ToolTip>
      <Icon url=":/cubitgui/i32gcreate.png"/>
      <NavigationNode marker="CreateVolumeBrick" name="Brick">
        <Label>Brick</Label>
        <ToolTip>Create a brick</ToolTip>
      </NavigationNode>
      <NavigationNode marker="CreateVolumeCone" name="Cone">
        <Label>Cone</Label>
        <ToolTip>Create a cone</ToolTip>
      </NavigationNode>
```

Users may modify the **Label**, **ToolTip**, or **Icon url**. Users may remove entire categories if necessary. Users should not modify **NavigationNode** or **NavigationReference** tags. Users may create their own command panels using Qt and add them to the hierarchy.

# Periodic Space Filling Models (Tile)

This appendix describes commands for producing good-quality meshes of models that tile space, such as polycrystalline materials models. Such models are often referred to as "periodic", but since that term already has a different meaning in Cubit, the keyword "tile" is used instead. Meshes may be smoothed across periodic boundaries. Periodic boundary conditions can be automatically set up, according to ALEGRA conventions (SAND99-2698).

Tile commands are alpha features and should be used with caution.

## Initial setup

First import the model and merge the surfaces. Then mesh it with any method that will create meshes that match across the tile (periodic) boundary, say with scheme polyhedron or sweep. Once the mesh is created, specify the "tile vectors", which lets Cubit know that the nodes across the periodic boundaries are actually the same node:

```
Tile {x <period> | y <period> | z <period>}
[x <period>] [y <period>] [z <period>]
```

The 'period' you specify is actually the vector offset from one boundary to its match. Specify one tile command for each coordinate axis that the model is periodic in. E.g.

```
Tile x 1
Tile y 1
Tile z 1
```

You can see which nodes are matched to a given node by some combination of tile vectors with the following command: Tile Debug Node <id>
If you later need to delete these tile vectors, use the following command:

```
Tile Off
```

## Creating Nodesets

Once the tile vectors are specified, you can set up periodic boundary conditions that meet ALEGRA specifications. The command is:

```
Tile Nodeset <start_id>
```

This will create a nodeset for all combinations of tile vectors that actually connect nodes. The nodesets created will be reported to you. The nodesets will be consecutive starting with the given 'start_id', except that if there are no nodes for a particular combination there will be no nodeset and the id space will have a hole. To delete these nodesets, use the

```
Tile Off
```

command rather than the usual commands to delete nodesets.

## Smoothing

Once a mesh has been created and the tile vectors have been specified, you can smooth the mesh and keep the periodic boundaries exactly offset by the tile vectors. Only hex meshes are currently supported. A variety of 3d smoothing schemes are supported, including laplacian, equipotential, untangle, and condition number.

```
Smooth Volume <volume_id_range> [Global [Float <dim>] ]
```

Use "Global" if you are smoothing a collection of volumes. Use "float 3" if you want nodes on surfaces, curves, and vertices to be able to move off of their geometric owner. Use "float 2" if you want just nodes on curves and vertices to be able to move off of their owner (but stay on an owning surface). It is often useful to specify that some of the nodes are fixed using the "node position fixed" command.

Example

```
# make the geometry
#{brick_size=500}
brick wid {brick_size}
brick wid {brick_size}
body 2 move {brick_size} 0 0
brick wid {brick_size}
body 3 move {brick_size} {brick_size} 0
brick wid {brick_size}
body 4 move 0 {brick_size} 0
brick wid {brick_size}
body 5 move 0 0 {brick_size}
brick wid {brick_size}
body 6 move {brick_size} 0 {brick_size}
brick wid {brick_size}
body 7 move {brick_size} {brick_size} {brick_size}
brick wid {brick_size}
body 8 move 0 {brick_size} {brick_size}
merge all

# mesh it
vol all int 3
mesh vol all

# set the tiling vectors
tile x {brick_size*2}
tile y {brick_size*2}
tile z {brick_size*2}
tile debug node 256
tile debug node 245

# set the tiling nodesets
tile nodeset

# mess up the mesh quality
# volume all smooth scheme randomize
# smooth volume all
surface all smooth scheme randomize
smooth surface all
draw hex all

# fix the mesh quality
node in volume all position fixed
node in surface all position free
volume all smooth scheme laplac
```

```
# volume all smooth scheme untangle beta 0.08
smooth volume all global float 3
draw hex all
```

# References

Attaway, Stephen W.; Mello, Frank J.; Heinstein, Martin W.; Swegle, Jeffrey W.; Ratner, Julie A.; Zadoks, Rick Ian, "PRONTO3D users' instructions: a transient dynamic code for nonlinear structural analysis," Sandia Report SAND 98-1361 Sandia National Laboratories, Albuquerque, NM (1998)

Attaway S. W., unpublished, (1993)

Blacker, T. D., FASTQ Users Manual Version 1.2, SAND88-1326, Sandia National Laboratories, (1988)

Blacker, Ted D. "An Adaptive Finite Element Technique Using Element Equilibrium and Paving", American Society of Mechanical Engineers, Annual Meeting Dallas Texas, November 25-30, 1990, ASME, Nov 1990

Blacker, Ted D., "Paving: A New Approach To Automated Quadrilateral Mesh Generation", International Journal For Numerical Methods in Engineering, John Wiley, Num 32, pp.811-847, 1991

Blacker T.D. and Meyers R.J,."Seams and Wedges in Plastering: A 3D Hexahedral Mesh Generation Algorithm", Engineering with Computers, Springer Verlag, Vol 2, Num 9, pp.83-93, 1993

Brewer, M., L. Diachin, P. Knupp, T. Leurent, and D. Melander, "The Mesquite Mesh Quality Improvement Toolkit", Proceedings, 12th International Meshing Roundtable, 2003

Brewer, M., "Geometry-Tolerant Meshing Using Advancing-Front Techniques", SAND Report, (6-2008)

Butlin, Geoffrey and Clive Stops, "CAD Data Repair", 5th International Meshing Roundtable, pp.7-12, 1996

Clark Brett W., "Removing Small Features with Real Solid Modeling Operations", Submitted to 16th International Meshing Roundtable, 2007

Cook, W. A. and W. R. Oakes (1982) Mapping methods for generating threedimensional meshes, Computers In Mechanical Engineering, CIME Research Supplement:67-72, August 1982

Folwell, Nathan T. and Scott A. Mitchell, "Reliable Whisker Weaving via Curve Contraction", Proceedings, 7th International Meshing Roundtable, Sandia National Lab, pp.365-378, October 1998

Freitag, Lori A. and Patrick M. Knupp , "Tetrahedral Element Shape Optimization via the Jacobian Determinant and Condition Number", Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, U.S.A., pp.247-258, October 1999

George, P.L., F. Hecht and E. Saltel, "Automatic Mesh Generator with Specified Boundary", Computer Methods in Applied Mechanics and Engineering, Vol. 92, pp. 269-288, 1991

Hardwick, Mike, "DART System Analysis Presented to Simulation Sciences Seminar", June 28, 2005

Jones, R.E., QMESH: A Self-Organizing Mesh Generation Program, SLA - 73 - 1088, Sandia National Laboratories, (1974).

Knupp, Patrick M., "Winslow Smoothing On Two-Dimensional Unstructured Meshes", Proceedings, 7th International Meshing Roundtable, Sandia National Lab, pp.449-457, October 1998
Knupp, Patrick M., "Matrix Norms & The Condition Number: A General Framework to Improve Mesh Quality Via Node-Movement", Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, U.S.A., pp.13-22, October 1999
Knupp, P., "Achieving Finite Element Mesh Quality via Optimization of the Jacobian Matrix Norm and Associated Quantities, Part I", Int. J. Num. Meth. Engr.. 2000
Lovejoy, S. C. and R. G. Whirley, DYNA3D Example Problem Manual, UCRL-MA--105259, University Of California and Lawrence Livermore National Laboratory, (1990).
Melander, Darryl J., Timothy J. Tautges, Steven E. Benzley "Generation of Multi-Million Element Meshes for Solid Model-Based Geometries: The Dicer Algorithm" AMD-Vol. 220 Trends in Unstructured Mesh Generation, ASME, pp.131-135, July 1997
Mezentsev, Andrey A., "Methods and Algorithms of Automated CAD Repair For Incremental Surface Meshing", Proceedings, 8th International Meshing Roundtable, pp.299-309, 1999
Murdoch, Peter and Steven E. Benzley, "The Spatial Twist Continuum", Proceedings, 4th International Meshing Roundtable, Sandia National Laboratories, pp.243-251, October 1995
Oddy, A., J. Goldak, M. McDill, and M. Bibby "A Distortion Metric for Isoparametric Finite Elements" Transactions of the Canadian Soc. Mech. Engr., pp213-217, Vol 12, No 4, 1988.
Owen, Steven J. and David R. White, "Mesh-Based Geometry: A Systematic Approach to Constructing Geometry from the Nodes and Elements of a Finite Element Mesh", 10th International Meshing Roundtable, Sandia National Laboratories, pp. 83-96, October 2001
Owen, Steven J., Clark, B.W., Melander, D.J., Brewer, M.B., Shepherd, J.F., Merkley, K., Ernst, C., Morris, R., "An Immersive Topology Environment for Meshing", Accepted to 16th International Meshing Roundtable, 2007
Parthasarathy V. N. et al, "A comparison of tetrahedron quality measures", Finite Elem. Anal. Des., Vol 15, 1993, 255-261.
Price, M.A. and C.G. Armstrong, "Hexahedral Mesh Generation by Medial Surface Subdivision: Part I, Solids With Convex Edges, International Journal for Numerical Methods in Engineering, Vol. 38, No. 19, pp. 3335-3359, 1995
W. Quadros, V. Vyas, M. Brewer, S. Owen, and K. Shimada, “A Computational Framework for Generating Sizing Function in Assembly Meshing”, *Proceedings*, *14 th International Meshing Roundtable*, 2005
W. R. Quadros, K. Shimada, and S. J. Owen, “Skeleton-based computational method for the generation of a 3D finite element mesh sizing function”, *Engineering with Computers*, Springer Verlag, Vol 20, Num 3, pp.249-264, 2004
W. R. Quadros, S. J. Owen, M. Brewer, and K. Shimada, “Finite Element Mesh Sizing for Surfaces using Skeleton”, *Proceedings*, *13 th International Meshing Roundtable*, 2004
Robinson, J., "CRE method of element testing and Jacobian shape parameters, Eng. Comput., Vol. 4 (1987).

Ruppert, Jim , "A New and Simple Algorithm for Quality 2-Dimensional Mesh Generation". Technical Report UCB/CSD 92/694, University of California at Berkely, Berkely California (1992)
Scott, Michael A., Matthew N. Earp, Steven E. Benzley, and Michael B. Stephenson, "Adaptive Sweeping Techniques," Proceedings of the 14th International Meshing Roundtable, Springer, pp. 417-432, 2005.
Schoof, L. A.and Victor R. Yarberry, "EXODUS II A Finite Element Data Model", SAND92-2137, Sandia National Laboratories, (1995).
Sheffer, A., "Model simplification for meshing using face clustering", Computer-Aided Design, Vol. 33, No. 13, pp. 925-934(10), 2001
Staten, Matthew L., Steven J. Owen, Ted D. Blacker, "Unconstrained Paving and Plastering: A New Idea for All Hexahedral Mesh Generation", Proceedings, 14th International Meshing Roundtable, pp.399-416, 2005
Staten, Matthew L., Robert A. Kerr, Steven J. Owen, Ted D. Blacker, "Unconstrained Paving and Plastering: Progress Update", Proceedings, 15th International Meshing Roundtable, pp.469-486, 2006
Staten, Matthew L., Brian Carnes, Corey McBride, Clint Stimpson, Jim Cox, "Mesh Scaling for Affordable Solution Verification", Proceedings, 25th International Meshing Roundtable, pp. 46-58, 2016
Stimpson, CJ, Ernst, CD, Knupp, P, Pebay; P, and Thompson, D. "The Verdict Geometric Quality Library", Sandia Report SAND2007-175, 2007
Tautges, Timothy J. and Scott A. Mitchell, "Whisker Weaving: Invalid Connectivity Resolution and Primal Construction Algorithm", Proceedings, 4th International Meshing Roundtable, Sandia National Laboratories, pp.115-127, October 1995
Tautges, Timothy J., Ted Blacker, Scott A. Mitchell, "The Whisker Weaving Algorithm: A Connectivity-Based Method for Constructing All-Hexahedral Finite Element Meshes", International Journal for Numerical Methods in Engineering, Wiley, Vol 39, pp.3327-3349, 1996
Tautges, Timothy J., "The Common Geometry Module (CGM): A Generic, Extensible Geometry Interface", Proceedings, 9th International Meshing Roundtable, pp. 337-348, 2000
Tautges, Timothy J., "Automatic Detail Reduction for Mesh Generation Applications", Proceedings, 10th International Meshing Roundtable, pp.407-418, 2001
Taylor, L. M. and D. P. Flanagan, "Pronto 3D--A Three-Dimensional Transient Solid Dynamics Program", SAND87-1912, Sandia National Laboratories, (1989).
Tipton ,R. E., "Grid Optimization by Equipotential Relaxation", unpublished, Lawrence Livermore National Laboratory, (1990)
Walton, D. J. and D. S. Meek, "A Triangular G1 Patch from Boundary Curves," Computer-Aided Design, Vol. 28 No. 2 pp. 113-123 (1996)
Watson, David F. , "Computing the Delaunay Tessellation with Application to Voronoi Polytopes", The Computer Journal, Vol 24(2) pp.167-172 (1981)
Wellman, Gerald W., "MAPVAR : a computer program to transfer solution data between finite element meshes", Sandia Report SAND 99-0466 Sandia National Laboratories, Albuquerque, NM (1999)

White, David R. and Paul Kinney, "Redesign of the Paving Algorithm: Robustness Enhancements through Element by Element Meshing", Proceedings, 6th International Meshing Roundtable, Sandia National Laboratories, pp.323-335, October 1997
White, David R. and Sunil Saigal (2002) Improved Imprint and Merge for Conformal Meshing, Proceedings, 11th International Meshing Roundtable, pp.285-296
White, David R. and Timothy J. Tautges, "Automatic Scheme Selection for Toolkit Hex Meshing", International Journal for Numerical Methods in Engineering, Vol. 49, No. 1, pp. 127-144, 2000
Whiteley, M., D. White, S. Benzley and T. Blacker, "Two and Three-Quarter Dimensional Meshing Facilitators", Engineering with Computers, Springer-Verlag, Vol 12, pp.155-167, December 1996
Yong Lu, Rajit Gadh, and Timothy J. Tautges, "Volume decomposition and feature recognition for hexahedral mesh generation", Proceedings, 8th International Meshing Roundtable, pp. 269-280, 1999

# Available Colors

In addition to color specification by RGB values, all color commands in CUBIT allow the specification of a color name. The following table lists the colors available by ID or name in CUBIT. The table lists the color number (#), color name, and the red, green, and blue components corresponding to each color, for reference.

| Number | Color Name | Red | Green | Blue |
|---|---|---|---|---|
| 0 | black | 0.000 | 0.000 | 0.000 |
| 1 | grey | 0.500 | 0.500 | 0.500 |
| 2 | orange | 0.000 | 1.000 | 0.000 |
| 3 | red | 1.000 | 1.000 | 0.000 |
| 4 | green | 1.000 | 0.000 | 0.000 |
| 5 | yellow | 1.000 | 0.000 | 1.000 |
| 6 | magenta | 0.000 | 1.000 | 1.000 |
| 7 | cyan | 0.000 | 0.000 | 1.000 |
| 8 | blue | 1.000 | 1.000 | 1.000 |
| 9 | white | 1.000 | 0.647 | 0.000 |
| 10 | brown | 0.647 | 0.165 | 0.165 |
| 11 | gold | 1.000 | 0.843 | 0.000 |
| 12 | lightblue | 0.678 | 0.847 | 0.902 |

| | | | | |
|---|---|---|---|---|
| 13 | lightgreen | 0.000 | 0.800 | 0.000 |
| 14 | salmon | 0.980 | 0.502 | 0.447 |
| 15 | coral | 1.000 | 0.498 | 0.314 |
| 16 | pink | 1.000 | 0.753 | 0.796 |
| 17 | purple | 0.627 | 0.125 | 0.941 |
| 18 | paleturquoise | 0.686 | 0.933 | 0.933 |
| 19 | lightsalmon | 1.000 | 0.627 | 0.478 |
| 20 | springgreen | 0.000 | 1.000 | 0.498 |
| 21 | slateblue | 0.416 | 0.353 | 0.804 |
| 22 | sienna | 0.627 | 0.322 | 0.176 |
| 23 | seagreen | 0.180 | 0.545 | 0.341 |
| 24 | deepskyblue | 0.000 | 0.749 | 1.000 |
| 25 | khaki | 0.941 | 0.902 | 0.549 |
| 26 | lightskyblue | 0.529 | 0.808 | 0.980 |
| 27 | turquoise | 0.251 | 0.878 | 0.816 |
| 28 | greenyellow | 0.678 | 1.000 | 0.184 |
| 29 | powderblue | 0.690 | 0.878 | 0.902 |
| 30 | mediumturquoise | 0.282 | 0.820 | 0.800 |
| 31 | skyblue | 0.529 | 0.808 | 0.922 |
| 32 | tomato | 1.000 | 0.388 | 0.278 |
| 33 | lightcyan | 0.878 | 1.000 | 1.000 |
| 34 | dodgerblue | 0.118 | 0.565 | 1.000 |
| 35 | aquamarine | 0.498 | 1.000 | 0.831 |
| 36 | lightgoldenrodyellow | 0.980 | 0.980 | 0.824 |
| 37 | darkgreen | 0.000 | 0.392 | 0.000 |

| | | | | |
|---|---|---|---|---|
| 38 | lightcoral | 0.941 | 0.502 | 0.502 |
| 39 | mediumslateblue | 0.482 | 0.408 | 0.933 |
| 40 | lightseagreen | 0.125 | 0.698 | 0.667 |
| 41 | goldenrod | 0.855 | 0.647 | 0.125 |
| 42 | indianred | 0.804 | 0.361 | 0.361 |
| 43 | mediumspringgreen | 0.000 | 0.980 | 0.604 |
| 44 | darkturquoise | 0.000 | 0.808 | 0.820 |
| 45 | yellowgreen | 0.604 | 0.804 | 0.196 |
| 46 | chocolate | 0.824 | 0.412 | 0.118 |
| 47 | steelblue | 0.275 | 0.510 | 0.706 |
| 48 | burlywood | 0.871 | 0.722 | 0.529 |
| 49 | hotpink | 1.000 | 0.412 | 0.706 |
| 50 | saddlebrown | 0.545 | 0.271 | 0.075 |
| 51 | violet | 0.933 | 0.510 | 0.933 |
| 52 | tan | 0.824 | 0.706 | 0.549 |
| 53 | mediumseagreen | 0.235 | 0.702 | 0.443 |
| 54 | thistle | 0.847 | 0.749 | 0.847 |
| 55 | palegoldenrod | 0.933 | 0.910 | 0.667 |
| 56 | firebrick | 0.698 | 0.133 | 0.133 |
| 57 | palegreen | 0.596 | 0.984 | 0.596 |
| 58 | lightyellow | 1.000 | 1.000 | 0.878 |
| 59 | darksalmon | 0.914 | 0.588 | 0.478 |
| 60 | orangered | 1.000 | 0.271 | 0.000 |
| 61 | palevioletred | 0.859 | 0.439 | 0.576 |
| 62 | limegreen | 0.196 | 0.804 | 0.196 |

| | | | | |
|---|---|---|---|---|
| 63 | mediumblue | 0.000 | 0.000 | 0.804 |
| 64 | blueviolet | 0.541 | 0.169 | 0.886 |
| 65 | deeppink | 1.000 | 0.078 | 0.576 |
| 66 | beige | 0.961 | 0.961 | 0.863 |
| 67 | royalblue | 0.255 | 0.412 | 0.882 |
| 68 | darkkhaki | 0.741 | 0.718 | 0.420 |
| 69 | lawngreen | 0.486 | 0.988 | 0.000 |
| 70 | lightgoldenrod | 0.933 | 0.867 | 0.510 |
| 71 | plum | 0.867 | 0.627 | 0.867 |
| 72 | sandybrown | 0.957 | 0.643 | 0.376 |
| 73 | lightslateblue | 0.518 | 0.439 | 1.000 |
| 74 | orchid | 0.855 | 0.439 | 0.839 |
| 75 | cadetblue | 0.373 | 0.620 | 0.627 |
| 76 | peru | 0.804 | 0.522 | 0.247 |
| 77 | olivedrab | 0.420 | 0.557 | 0.137 |
| 78 | mediumpurple | 0.576 | 0.439 | 0.859 |
| 79 | maroon | 0.690 | 0.188 | 0.376 |
| 80 | lightpink | 1.000 | 0.714 | 0.757 |
| 81 | darkslateblue | 0.282 | 0.239 | 0.545 |
| 82 | rosybrown | 0.737 | 0.561 | 0.561 |
| 83 | mediumvioletred | 0.780 | 0.082 | 0.522 |
| 84 | lightsteelblue | 0.690 | 0.769 | 0.871 |
| 85 | mediumaquamarine | 0.400 | 0.804 | 0.667 |
| 86 | proe_background | 0.000 | 0.000 | 0.300 |
| 87 | win2kgray | 0.831 | 0.815 | 0.784 |

| | | | | |
|---|---|---|---|---|
| 88 | proe_2001_top | 0.000 | 0.365 | 0.608 |
| 89 | proe_2001_bottom | 0.000 | 0.067 | 0.196 |
| 90 | wildfire_top | 0.620 | 0.608 | 0.569 |
| 91 | wildfire_bottom | 0.890 | 0.882 | 0.835 |

# Element Numbering

This appendix describes the element node and side numbering conventions used in Exodus II files written by CUBIT. This information is located here for convenience, but is identical to the information presented in the Exodus II manual; citation Schoof, 95

## Node Numbering

The node numbering used for the basic elements is shown Figure 1. Specific element types of lower order just contain the number of nodes needed for those elements; for example, QUAD4 or QUAD elements use just the first four nodes shown for quadrilaterals in Figure 1.

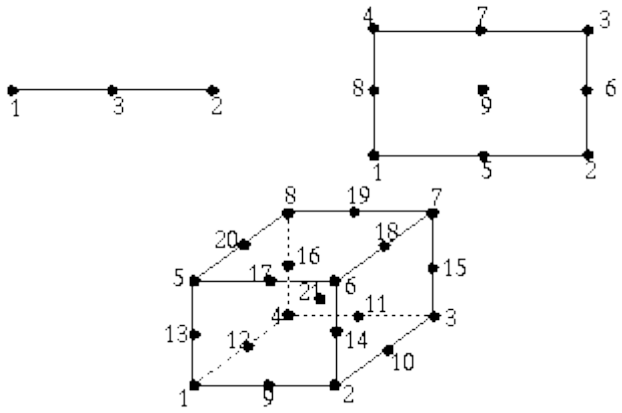



**Figure 1. Local Node Numbering for CUBIT element types**

## Side Numbering

Element sides are used to specify boundary conditions that act over a length or area, for example pressure- or flux-type boundary conditions. Each element side is represented in the Exodus II format by an element number and the local side number for that element. The local side numbering for the basic elements is shown in Figure 2.

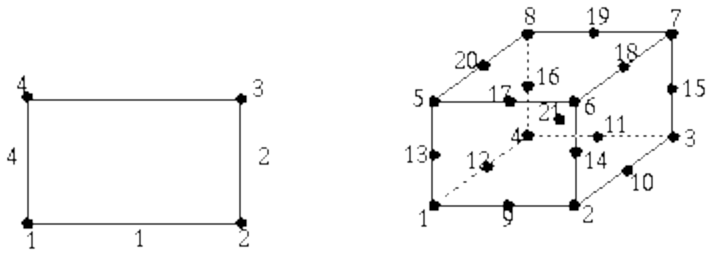



**Figure 2. Local side numbering for CUBIT element types**

## Triangular Shell Element Numbering

A three-dimensional shell element with triangular topology will have the element type 'TRISHELL'. This type can be modified for different element orders by appending the number of nodes onto the end of the type. For example, a 6-node shell could have the element type 'TRISHELL6'. However, any element whose type begins with the 8 letters 'TRISHELL' in upper, lower, or mixed case will refer to an element with a triangular topology. The element can exist in either three-space or two-space.

Attributes:

1. If the element exists in two-space, there are no required attributes.
2. If the element exists in three-space, there is one required attribute which is the thickness of the shell.
3. If the number of attributes is equal to the number of nodes in the connectivity of the element, then the attributes are assumed to specify the thickness of the element at each of the elements nodes. The ordering of the attributes matches the ordering of the elements nodes.

*Node Ordering*

The node ordering of the 3D triangle matches the node ordering of the 2D triangle as shown in Figure 3.

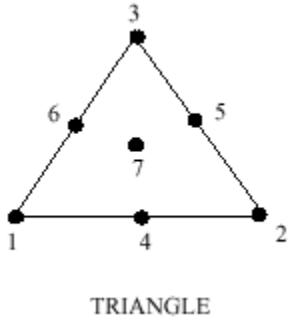



**Figure 3. Local Node Numbering for CUBIT triangular element types**

*Side Set Side Ordering*

The sideset side ordering is different for the element in the 2D and 3D instances.

In 2D, the sideset side ordering matches what is shown in Figure 4.

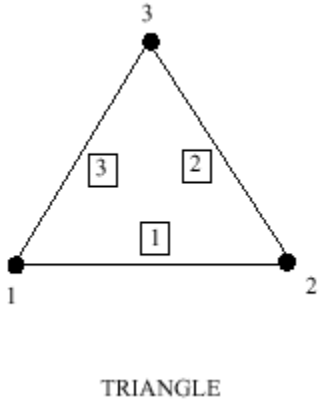



**Figure 4. Local sideset numbering for CUBIT triangular element types**

In 3D, the sideset side and node ordering is the same as for a quad shell except that there are only 3 or 6 nodes.
Then:

| | | | |
|---|---|---|---|
| side | 1 | == | {1,2,3} |
| side | 2 | == | {3,2,1} |
| side | 3 | == | {1,2} |
| side | 4 | == | {2,3} |

side 5 == {3,1}
If it is a higher order triangular shell (6 [or 7 nodes]), then the higher-order nodes are added on to the end of the above:

| | | | |
|---|---|---|---|
| side | 1 | == | {1,2,3,4,5,6[,7]} |
| side | 2 | == | {3,2,1,6,5,4[,7]} |
| side | 3 | == | {1,2,4} |
| side | 4 | == | {2,3,5} |

side 5 == {3,1,6}

# FASTQ

FASTQ is a program developed to create geometry and two-dimensional mesh. The user may choose to upload FASTQ files and work with the files in an environment that accepts a limited number of FASTQ commands.

**Table 1. FASTQ Commands Executable in Cubit**

| Syntax | Description |
|---|---|
| set fastq on | Cubit is in FASTQ mode. |
| set fastq off | Cubit exits FASTQ mode. |
| nine | Mesh will be generated using nine-node quadrilateral elements. |
| eight | Mesh will be generated using eight-node quadrilateral elements. |

| | |
|---|---|
| five | Mesh will be generated using five-node quadrilateral elements. |
| import fastq " *.fsq " | Imports FASTQ files into Cubit. |

**Table 2. Brief List of Importable FASTQ Commands Supported in Cubit**

| Syntax | Description |
|---|---|
| point <point_id> <x-coord> <y-coord> [<z-coord>] | This creates a point at the specified coordinates with the id given by the user. The z-coordinate is optional because FASTQ is a two-dimensional meshing tool. |
| line <line_id> str <begin_pt> <end_pt> 0 [interval] [factor] | This creates a straight line with the given beginning and end points and an id is assigned to the line. The interval option determines the number of intervals or subdivisions of the line for mesh generation. The factor option is the ratio of the interval lengths as the intervals progress towards the end point of the line. For example, if a factor of 2 is specified, each interval will be 2 times longer than the interval before it. If a factor is not specified, the default factor is 1. |
| line <line_id> circ <begin_pt> <end_pt> <center_pt> [interval] [factor] | The command creates a circular arc (or logarithmic spiral) about a center point. The beginning and ending points specify where to position the circular arc. The third point in the command specifies the center of the circular arc. Interval and factor are defined in the explanation for the Line (STR) Command. |
| line <line_id> cirm <begin_pt> <end_pt> <center_pt> [interval] [factor] | The CIRM line is similar to the CIRC line. The difference between the CIRM line and the CIRC line is the function of the third point. The third point on a CIRM line is between the beginning and end points and |

| | |
|---|---|
| | becomes a part of the circular arc. The arc will be drawn through all three points. |
| line <line_id> cirr <begin_pt> <end_pt> <center_pt> [interval] [factor] | The command creates a circular arc. The beginning and end points function the same as the other commands to create a circular arc, but the third point is used differently. The x value of the third point will be used as the radius of the arc to be created. If the x value is positive, the center point is placed on the left of a straight line drawn through the beginning and end points. If the x value is negative, the center is placed on the right side of the line. |
| line <line_id> para <begin_pt> <end_pt> <center_pt> [interval] [factor] | This command creates the tip of a parabolic arc. The third point is the peak of the parabola. The beginning and end points must be equidistant from the third point. |
| line <line_id> corn <begin_pt> <end_pt> <center_pt> [interval] [factor] | The command creates a corner formed by two line segments. The first segment is created by connecting the first and third points. The second segment is created by connecting the third and second points. The line segments can have their interval size set as if the two lines were one. |
| side <side_id> <list_of_lines> | This creates a group made up of the given lines and assigns the id given by the user. |
| region <region_id> <block_id> <list_of_lines_or_sides> | A region is a list of lines/sides that enclose an area to be meshed. The region is formed from the list of lines and/or sides; the region is given the id specified by the user. |
| barset <barset_id> <block_id> <inside> <list_of_lines> | The basis for two and three node element generation is the barset. The barset id is the identifying number for the barset. The block |

| | |
|---|---|
| | id is the id assigned to all elements in the barset. The inside point is a point on the inside of all lines in the barset. All lines specified at the end of the command will be included in the barset. |
| interval <interval> <list_of_lines> | This sets the number of intervals on a given line or lines. |
| factor <factor> <list_of_lines> | This command sets the ratio of the interval lengths as the intervals progress towards the end point of the line. For example, if a factor of 2 is specified, each interval will be 2 times longer than the interval before it. If a factor is not specified, the default factor is 1. |
| poinbc <node_bc_id> <list_of_points> | This command attaches boundary conditions to the nodes that are created at point locations.The first number to be entered is the id of the flag. After that a list of all points to be flagged is entered. |
| linebc <node_bc_id> <list_of_lines> | This command attaches boundary conditions to nodes created along certain lines. The first number entered is the id of the flag. Following the id, all lines to be flagged should be entered. |
| sidebc <side_bc_id> <list_of_lines> | This command attaches boundary conditions to all nodes created on certain lines. The first number entered is the id of the flag. All numbers entered after that point are the ids of the sidesets included in the flag. |
| scheme <region_id> {m\|t\|b\|c\|u} | The letters after the region id indicate the meshing scheme. Schemes specify a meshing algorithm for mesh generation is a regionThe letter 'm' indicates a general rectangle primitive, 't' indicates a triangle primitive, 'b' |

| | |
|---|---|
| | indicates a transition primitive, 'c' indicates a semicircle primitive, and 'u' indicates a pentagon primitive. |

# FullHex vs. NodeHex Representation

CUBIT has two different internal representations of hexes: FullHexes and NodeHexes. The NodeHex is a lighter weight data structure, but occasionally nodeset and sideset shortcomings can be overcome by using FullHexes. The user can select which type of hexes get created when generating or importing a volume mesh with the following command:

**Set FullHex [Use] [on|OFF]**

Using the FullHex representation increases the memory used to store a mesh by a factor of approximately five.

# Generating a Finite Element Mesh from Level-set Data

This documentation will describe how to generate a finite element mesh from an Exodus file that contains level-set data defined as nodal variables. This process was developed to support mesh generation for geometric designs resulting from Adaptive Topological Optimization (ATO). The output format from ATO in this case is an Exodus file containing a tetrahedral mesh with a scalar nodal variable defining a level-set that represents the bounding surfaces of the optimized volume. The process below extracts the bounding surfaces of the optimized volume by evaluating the level-set at a value of zero. These bounding surfaces are in the form of a triangulation that can then be used for generating a finite element mesh. Three methods for generating the finite element mesh will be described using a simple example model.

## Hex Mesh Generation Using Sculpt

This section will describe the process for generating a sculpted hex mesh of the ATO design.

1. This process utilizes beta capabilities in Cubit so activate the use of beta commands.

**set dev on**

2. Import the ATO Exodus file called **"small_bracket.exo"** into Cubit and use the import option that tells Cubit to import the level-set nodal variable called "LSD" (level set data).

**import mesh "small_bracket.exo" nodal_var "LSD" no_geom**

3. Extract the boundary surface triangulations from the level-set data. Because this creates triangles we use a variation of the "create tri …" command. We will use the "iso" option telling the command to generate the iso-surfaces from the level-set data defined by the nodal variable "LSD". We will specify the tets to consider when doing the extraction. In this case we use "tet all" meaning all of the tets in the model. When this command is complete there will be two new blocks defined in Cubit. They will be the two blocks with the highest IDs. One of these blocks (the larger of the two block IDs) contains triangles that represent the optimized portions of the design and the other block (the smaller of the two block IDs) contains triangles on the fixed portions of the design.

**create tri iso tet all nodal_var "LSD"**

4. Look at the extracted iso-surface triangulations by drawing the two new blocks (blocks 3 & 4) created in step 2.

**draw block 3 4**

5. At this point we could smooth the new triangulations if desired but because the sculpting process will have the same effect we will skip that step here. The smoothing will be demonstrated in the next sections. We will now export the new blocks to an STL file that can then be used by the sculpt algorithm. Specify only the triangles in the new blocks. Also specify the "mesh free_mesh" options to tell the command that the triangles are not owned by geometry.

**export stl "small_bracket.stl" tri in block 3 4 mesh free_mesh**

6. Exit Cubit and start a Linux command prompt from which to run sculpt.
7. From the command prompt load the sierra module (this will be used later).

**module load sierra**

8. From the directory where your new stl file is launch the sculpt program. See the sculpt documentation for more details on the options for running sculpt. Here is an example of a simple sculpt command that specifies the sculpt cell size and the number of processors. The number or processors is 8 and the cell size is 0.0007.

**sculpt –j 8 –cs 0.1 --stl_file "small_bracket.stl"**

9. When sculpt finishes the resulting mesh will be spread across 8 files in our case since we used 8 processors. In our example the files will be named something like "small_bracket.stl_results.e.8.0" where the last number in the filename refers to which processor the file came from. To concatenate all of the files into one use the "epu" command from the Sierra suite of tools.

**epu –auto small_bracket.stl_results.e.8.0**

10. Start Cubit up and load the mesh file created by sculpt.

**import mesh "small_bracket.stl_results.e" no_geom**

## Tet Mesh Generation by Remeshing Mesh Based Geometry (MBG)

This section will describe the process for generating a tet mesh by meshing a mesh based geometry (MBG) representation of the optimized part.

1. This process utilizes beta capabilities in Cubit so activate the use of beta commands.

**set dev on**

2. Import the ATO Exodus file called **"small_bracket.exo"** into Cubit and use the import option that tells Cubit to import the level-set nodal variable called "LSD" (level set data).

**import mesh "small_bracket.exo" nodal_var "LSD" no_geom**

3. Extract the boundary surface triangulations from the level-set data. Because this creates triangles we use a variation of the "create tri …" command. We will use the "iso" option telling the command to generate the iso-surfaces from the level-set data defined by the nodal variable "LSD". We will specify the tets to consider when doing the extraction. In this case we use "tet all" meaning all of the tets in the model. When this command is complete there will be two new blocks defined in Cubit. They will be the two blocks with the highest IDs. One of these blocks (the larger of the two block IDs) contains triangles that represent the optimized portions of the design and the other block (the smaller of the two block IDs) contains triangles on the fixed portions of the design.

**create tri iso tet all nodal_var "LSD"**

4. Look at the extracted iso-surface triangulations by drawing the two new blocks (blocks 3 & 4) created in step 2.

**draw block 3 4**

5. Export the new blocks to an Exodus file so that they are disconnected from the original tet mesh.

**export mesh "small_bracket_iso.e" block 3 4**

6. Reset Cubit.

**reset**

7. Load the new file that just contains the new blocks.

**import mesh “small_bracket_iso.e” no_geom**

8. Smooth the optimized part of the triangulation. This will be the triangles in the block with the larger id (4 in our case). We will do this by first fixing the node positions of all of the nodes in the non-optimized part of the triangulation and then smoothing the triangles in the optimized portion. When smoothing we need to use the “target free mesh” option to tell the command to project the smoothed results back to the original triangulation to try to preserve volume. We use the “iteration” option to limit the number of iterations the smoother does.

**node in tri in block 3 position fixed**
**smooth tri in block 4 target free mesh iteration 5**

9. Create mesh based geometry from the triangulation. For small models this can be done immediately with the commands below (first command creates surfaces and the second command stitches them together to form a closed volume). For larger models it may be faster to export the mesh to a file, reset Cubit, and then re-import the mesh with the “geom” option on so that mesh based geometry is generated on import. Current limitations in Cubit result in this performance difference.

**create mesh geom tri all**
**create vol surf all**

10. Set the size and scheme on the new volume and mesh it.

**volume all size .5**
**volume all scheme tetmesh**
**mesh volume all**

## Tet Mesh Generation Using Level-set Triangulation

This section will describe the process for generating a tet mesh using the triangulation from the level-set extraction.

1. This process utilizes beta capabilities in Cubit so activate the use of beta commands.

**set dev on**

2. Import the ATO Exodus file called **"small_bracket.exo"** into Cubit and use the import option that tells Cubit to import the level-set nodal variable called "LSD" (level set data).

**import mesh "small_bracket.exo" nodal_var "LSD" no_geom**

3. Extract the boundary surface triangulations from the level-set data. Because this creates triangles we use a variation of the "create tri …" command. We will use the "iso" option telling the command to generate the iso-surfaces from the level-set data defined by the nodal variable "LSD". We will specify the tets to consider when doing the extraction. In this case we use "tet all" meaning all of the tets in the model. When this command is complete there will be two new blocks defined in Cubit. They will be the two blocks with the highest IDs. One of these blocks (the larger of the two block IDs) contains triangles that represent the optimized portions of the design and the other block (the smaller of the two block IDs) contains triangles on the fixed portions of the design.

**create tri iso tet all nodal_var "LSD"**

4. Look at the extracted iso-surface triangulations by drawing the two new blocks (blocks 3 & 4) created in step 2.

**draw block 3 4**

5. Export the new blocks to an Exodus file so that they are disconnected from the original tet mesh.

**export mesh "small_bracket_iso.e" block 3 4**

6. Reset Cubit.

**reset**

7. Load the new file that just contains the new blocks.

**import mesh "small_bracket_iso.e" no_geom**

8. Smooth the optimized part of the triangulation. This will be the triangles in the block with the larger id (4 in our case). We will do this by first fixing the node positions of all of the nodes in the non-optimized part of the triangulation and then smoothing the triangles in the optimized portion. When smoothing we need to use the "target free mesh" option to tell the command to project the smoothed results back to the original triangulation to try to preserve volume. We use the "iteration" option to limit the number of iterations the smoother does.

**node in tri in block 3 position fixed**

**smooth tri in block 4 target free mesh iteration 5**

9. Sometimes we will also want to smooth the edges on the "curves" in-between the optimized and non-optimized regions. To do this we will first "un-fix" the node positions we fixed in the previous step. Then we will create a group with all of the edges in the optimized region and a group with all of the edges in the non-optimized region. Then we will intersect these two groups to get the edges that are in-between these two regions. Then we can smooth those edges. Finally, we will smooth the surfaces in the optimized region again. You may also wish to smooth the tris in the non-optimized region but this will require some more sophistication in fixing node positions so as not to lose sharp features in the model. This example shows one specific smoothing sequence. You may prefer other approaches.

**node in tri in block 3 position free**
**group "opt_edges" add edge in tri in block 4**
**group "non_opt_edges" add edge in tri in block 3**
**group "int_edges" intersect opt_edges with non_opt_edges**
**smooth edge in int_edges target free mesh**
**node in edge in int_edges position fixed**
**smooth tri in block 4 target free mesh iteration 5**

10. Once we get a decent surface triangle mesh we can tet mesh the interior.

**tetmesh tri all**

# Credits

## Credits




Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

**Manager**

- Michael Skroch, Manager, Computational Simulation Infrastructure Department (Org. 1543), Sandia National Laboratories

**Project Board**

- **Principal Investigator**: Ryan Viertel, Org. 1543

**Research and Development**

Computational Simulation Infrastructure Department, Org. 1543, Sandia National Laboratories, Albuquerque, NM

- Matthew L. Staten
- Steven J. Owen
- Roshan W. Quadros
- Byron Hanks
- Trevor Hensley
- Clinton Stimpson
- Corey Ernst
- Corey McBride
- Scott Mitchell
- Brian Carnes
- Kevin Copps
- Paul Stallings

External Contributors

- Mark Richardson
- Karl Merkley

**Administrative Assistant**

- Janeane Sanchez, 1545, Sandia National Laboratories

# Quick Reference

## Quick Reference

**Geometry | File Import | Meshing | Genesis | Program | Entity Parsing | Groups | Graphics | Settings**

The following is a brief overview of some of the most used command-line CUBIT commands.

**GEOMETRY**

**Primitives**

Brick X <> [Y <> Z <>]
Cylinder Radius <> Height <>
Frustum Z <> Radius <> [Top <>]
Frustum Z <> Maj Rad <> Min Rad <>
Prism Z <> Sides <> Rad <> [Maj <> Min <>]
Pyramid Height <> Sides <> Radius <>
Sphere Rad <> [Xpos] [Ypos] [Zpos] [Inn <>]
Torus Major Rad <> Minor Rad <>

**Booleans**

Unite <> [With <>] [keep]
Subtract <> From <> [keep]
Intersect <> [With <>] [keep]

**Transformations**

Body <> [Copy] Move <dx> <dy> <dz>
Move {} <> location {} <> [except [x] [y] [z]]
Rotate {} <> About {x| y| z|<> <> <>} Angle <>
Rotate {} <> About Vert <> Vert <> Angle <>
Rotate {} <> About Nor Of Surf <> Angle <> Body <> [Copy] Scale <> Body <> [Copy]
Reflect {x| y| z|< x> <y> <z>}

**Decomposition**

Webcut {} <> Pla Vert <> [Vert]<> [Vert]<> ()
Webcut {} <> Plane Surf <> ()
Webcut {} <> Plane {xpla| ypla| zpla} [offs <>]
Webcut {} <> Tool [Body] <>
Webcut {} <> With Sheet {Body| Surf} <>
Webcut {} <> With Sheet Ext Fr Surf <>
Webcut {} <> Cyl Rad <> Axis {x| y| z| Vert <> Vert <>| <x><y><z>} [cent]
Options: [Noimprint| Imprint( default)], [Nomerge( default)| Merge], [group_ results]
Section {} <> {{ xpla| ypla| zpla} [offs <>]} | Surf <>} [keep] [normal( default)| reverse]

**FILE IMPORT**

Import Acis 'filename'
Export Acis 'filename' [Body <>]
Import Mesh Geometry 'filename' (options)

**MESHING**

Mesh {} <>
Delete Mesh {} <> [Propagate]

**Intervals**
{} <> Interval {<> | Hard | Soft | Default}
{} <> Size {<> | Auto}
Match Intervals {} <> [Ass Grou [Onl| Infea]] [Seed Cur <>] [Map| Pave]
**Mesh schemes**
{} <> Scheme ...
**Curve**: bias, copy, curvature, equal, stretch
**Surface**: auto, circle, copy, hole, map, mirror, pave, pentagon, qtri, submap, triprimitive, trimap, trimesh, tripave
**Volume**: auto, copy, map, sphere, submap, sweep, tetmesh, tetprimitive, thex
Smooth {} <>
{} <> Smooth Scheme ...
**Smooth schemes**
**Curves**: laplacian, randomize
**Surface**: centroid area pull, equipotential, laplacian, condition number, randomize, untangle, winslow
**Volume**: equipotential, laplacian, condition number, untangle, randomize
**GENESIS**
Block <> {Group| Vol| Surf| Curv} <> [Remove]
SideSet <> {Group| Curve} <> [Remove]
NodeSet <> {} <> [Remove]
Export Genesis 'filename'
Block <> Attribute <>
Block <> Element Type <type_>
**Curves**: bar[| 2| 3]| beam[| 2| 3]| truss[| 2| 3]
**Surfaces**: quad[| 4| 8| 9]| shell[| 4| 8| 9]| tri[| 3| 6| 7]
**Volumes**: hex[| 8| 20| 27]| pyr| tetra[| 4| 8| 10| 14] hexshell
SideSet <> Surf <> [Rem|[ She][ For| Rev| Both]]
SideSet <> Surf <> wrt Volume <>
Reset {Genesis | Nodesets | Sidesets | Blocks}
**PROGRAM**
Play 'filename'
Record {' filename' | stop}
Logging {off|on file <'filename'> [resume]}
Reset
Reset Genesis
Quit
**ENTITY PARSING**
**Examples**
Surface 1 2 3 4 to 6 by 2 ...
Curve all in Volume 2 ...
Draw Edge all in Hex 32
List Curve 1 to 50 except 2 4 6
Draw Sideset 1 2 3 Curve 3 to 5 Hex 2 4 6
**GROUPS**

Group <> {add| equals| remove| xor} {} <>
Group <> {inters| unite} grou <> with grou <>
Group <> subtract group <> from group <>

**GRAPHICS**

**Default mouse buttons (command line)**

B1 - rotate; B2 - zoom; B3 - pan
Control-B1: pick entity (In graph win: 0,1,2,3,4 - Pick vert, curv, surf, vol, body)

**Shortcuts (focus in Graphics Window)**

**a** Add to selection group
**b** Toggle Bounding Box on Click
**c** Clear "picked" Group
**d** Display 'picked' group, make it the selection
**e** Echo ID of selection to command line
**f** Assign function to mouse button
**g** List geometry of selection
**h** Print help
**i** Toggle visibility of selection
**j/k** Move slicing plane down/up
**l** List current selection (as if you typed 'list ...')
**control-l** Give focus to the command prompt
**m/n** List picked group/selection contents
**p** Toggle Persistent Wireframe
**q** Quit Current Mode (Exit slicing if slicing)
**r** Remove from 'picked' Group
**s** Toggle save-mesh on slice move
**u** Toggle mouse circle visibility
**v** Reset view
**w** Toggle Wireframe on click
**x/y/z** Slice along x/y/z-axis
**Shift-Z** Zoom on current selection
**F1** Save view 1 Numbers: set what you're picking.
**ESC** Cancel current Action
**Tab** Next possible selection
**Shift-Tab** Previous possible selection
**Shift-S**Activates graphics clipping plane controls

**SETTINGS**

Set AcisOption {string|double|integer} 'OptionName' <value>
Set Attribute <attrib_type> Auto {actuate|update} {on|off}
[Set] Auto Size Default
[Set] Auto Size Function [1|2]
Set AutoUniqueId {on|OFF}
Set Auto Sweep Scheme {Sw|Proj|Trans|Rot}
Set Boolean Regularize [ON|off]
Set Block Mixed Element Output {offset|degenerate|explicit}
Set Block Triangle Offset <value>
Set Block Tetrahedron Offset <value>

Set Block Pyramid Offset <value>
Set Catch Interrupt [on|off]
Set Cleanup Angle <val> (default = 179.0)
Set {curve|surface} Imprint Cleanup Tolerance <value>
Set Continue Meshing [ON|off]
Set Core [on|off]
set {Corner|End} Angle <degrees>
set Corner Weight <value>
Set Crash Save [on|off]
[set] Diagnostic {on|off}
[set] Geometry Version <> (1400, 1500, 1600, 1700, 1800, 1900)
[set] Debug <index> {on|off}
[set] Debug <index> File <'filename'>
[set] Debug <index> Terminal
set Default Blocks {on|off|Volumes|Surfaces}
set Default Names {on|off}
Set Default Element [tri|tet|QUAD|HEX|None]
Set Default Autosize [ON|off]
Set Digits [<number_to_list = -1>]
Set Deletion Off
Set Developer [commands] [on|off]
Set Detail Periodic Fraction <value>
Set Duplicate Block Elements {on|OFF}
[set] Echo [on|off]
Set Exodus Single Precision [on|off]
[Set] [Export Mesh] Nodeset Associativity [on|OFF]
[Set] [Export Mesh] Nodeset Associativity Complete [on|OFF]
[Set] Facet BBox [ON|off]
[Set] Facet_modify [ON|off]
Set Fastq {on|off]
Set File Overwrite [Check] [ON|off]
set FPE {divbyzero|invalid|underflow|overflow|all} [<toggle>]
set Fix Duplicate Names {on|off}
set FullHex [Use] [on|OFF]
[Set] Geometry Accuracy <value>
Set Geometry Engine {acis|catia|facet}
Set Group Edge Visibility [on|OFF]
Set Hex Relative Size Metric <value>
[set] Info {on|off}
set Interval Weight <value>
Set Import Mesh [vertex] [curve] [surface] Tolerance <distance>
[Set] Import Mesh NodeSet Associativity [ON|off]
Set Import Mesh NodeSet Order [On|Off]
Set Imprint Groups {ON|off}
Set Keep Invalid Mesh [on|off]
[set] Journal {on|off}}

[set] Journal [Graphics|Names|Aprepro|Errors] [on|off]}
[set] Journal idless [on|off|reverse]}
set Keep Invalid Mesh {on|off}
[Set] Laminate Tolerance <double>
set Large Angle Weight <value>
Set Large Exodus [ON|Off]
Set Exodus NetCDF4 [On|OFF]
[set] Logging {off|on file <'filename'> [resume]}
[Set] Logging Errors {off|on file <'filename'> [resume]}
Set Mapping Constraint [ON|off]
set Match Intervals Rounding {on|off}
set Match Intervals Fast {on|off}
Set Match Intervals Delta <interval_difference = 0.0>
Set Maximum Arc_span {<angle>|default}
Set Maximum Interval <int>
Set Maximum Memory [on|off|value(in MB)]
Set Merge Test BBox {on|OFF}
Set Merge Test InternalSurf {on|OFF|Spline}
Set Merge Base Names [on|off]
Set Measure Small Tolerance <value>
Set Metrics [on|OFF]
Set Mesh Autodelete [ON|off]
[Set] Morph Smooth [ON|off]
Set Multisweep [ON|off]
Set Nastran Exporter Params Add '<param_string>'
Set Nastran Exporter Params Remove '<param_string>'
Set Nastran Exporter Params Clear
Set New Ids [on|off]
Set Node Coincident Tolerance [<value>]
set Node Constraint [ON|off]
Set Overlap [Facet] {Angle|Absolute} <value>
Set Overlap {Minimum|Maximum} {Gap|Angle} <value>
Set Overlap Normal {ANY|opposite|same}
Set Overlap Tolerance <value>
Set Overlap Group {on|OFF}
Set Overlap {List|Display} {ON|off}
Set Overlap [Within] {Body|Volume} {on|OFF}
Set Overlap Imprnt {on|OFF}
Set Parallel Meshing [on|OFF}
[Set] Paver Cleanup {ON|off|extend}
[Set] Paver Diagonal Scale <factor> (default = 0.9)
[Set] Paver Grid Cell <factor> (default = 2.5)
[Set] Paver Size Limits {default|minimum <value>|maximum<value>}
[set] Paver Smooth Method { Default | Smooth Scheme|Old}
[set] Paver Linearsizing {off|on}
Set Persistent Ids {off|ON}

set Patran Export Autogroups [on|OFF]
Set Patran Export Groups {ON|off]
Set Play History {on|OFF}
[set] Project Smooth {on|off}
Set Push Attribs {on|off}
Set Print Quality {WARNING|error|off}
Set QTri Test {angle|diagonal}
Set Qtri Split <2|4> (default = 2)
Set Quad Relative Size Metric <value>
Set Quality Threshold <double> (default = 0.2)
set Replacement character '.|_|@'
Set ReverseZoom [on|off]
Set Save [Exodus|Cubit] [backups <number>]
[set] Scheme Auto Fuzzy [Tolerance] <degrees>
Set Sculpt Refine {on|OFF}[set] Smooth Iterations {default|<value>}
Set Separate After Webcut [ON|off]
[set] Smooth Method {laplacian | isoparametric}
[set] Smooth Tol <value> (Default = 0.05)
set {source|target} surface pattern '<pattern>'
Set Split Surface Tolerance <value>
Set Split Surface Parametric {on|OFF}
Set Split Surface Auto Detect Triange {ON|off}
Set Split Surface Point Angle Threshold <value>
Set Split Surface Side Angle Threshold <value>
Set Split Surface Extend Gap Threshold <value>
Set Split Surface Extend Tolerance <value>
Set Split Surface Extend Normal {on|OFF}
Set Stop Error {on|OFF}
Set Submap CornerPicking {ON|off}
set Suffix character '.|_|@'
Set Tight [[Bounding] [Box] [{Surface|Curve|Vertex} {on|off}]]
[Set] Tridelaunay Point Placement [{asp|gq}] (Advancing Steiner Point,Guaranteed Quality)
[Set] Trimesher Advancing Front
[Set] Tolerant Mesh Feature Size <value>
[Set] Tolerant Mesh MBG {OFF|on|only}
Set Tri Relative Size Metric <value>
Set Tet Relative Size Metric <value>
set Turn Weight <value>
Set Unite Mixed [ON|off]
[Set] Unmerge Duplicate_mesh {on|off}
[Set] Unmerge New Ids [{on|off}]
Set Verbose Errors [on|off]
Set Verbose Mesh [on|off]
[set] Warning {on|off}
Set WorkingDirectory 'directory_path'

# Index


#
#{abortloop} 1197
.
.cub 24
.sab 237
.sat 237
A
abaqus 820, 875
abort management 26
absolute value 1201
Acceleration 151
accuracy 414
acis 239, 480, 489
adaptive 834
adjust boundary 734
advancing front 528, 571
align 278
align mesh 771
alpha commands 1167
ambient intensity 190
analyze geometry 66, 313
angle 184
  calculate 1201
  mesh quality 743, 751
  perspective 184
  vertex type 545
appendix 1167
apply button 133
aprepro 1197
  functions 1201
  journal file 29
  journaling 1210
  loops 1197
  variable 19
arc 259, 275
arc span 503
area 56, 743, 751
aspect ratio 735, 743, 745
Aspect Ratio Beta 745
Aspect Ratio Gamma 745
assembly 59, 471, 472, 475
Assembly Tool 85
associativity 821
attributes 462, 463, 464
  block 886
  metadata 472
Auto Bias 521
AutoCAD 239
autocenter 197
automatic curve biasing 501
automatic forced sweepability 341
automatic geometry cleanup 340
automatic scheme selection 723
  general notes 724
  surfaces 726
  vertex types 723
  volumes 726
automatic size assignment 503
autosmooth 548
axis 33, 151, 176, 192
B
background color 173
bar 903
batch 19
beam 894, 903
bend 328
bias 521, 733, 786, 834
biasing along curves 501
bitmap 189
blend surfaces 330, 1146
block 886
  attribute 886
  curve 886
  element type 886
  repositioning 810
  surface 886
  volume 886
body 250
  align 278
  auto heal 314
  copy 280
  healer analyze 313
  imprint 422
  intersect 284
  list 56
  merge 426

move 278, 281
reflect 282
rotate 278, 283
scale 283
section 311
separate 312
split 292
subtract 284
unite 285
webcut 292
boolean 284
intersect 284
subtract 284
unite 285
border 192
boundary condition 918
boundary condition sets 928
boundary conditions 7, 903
feature 7
Boundary Layer 931
brick 244
bug reports 8
buttons 131
C
CALCULATED 1213
camera 184
cancel 23
cd command 14
ceiling 1201
centroid area pull 762
cfd 919
CFD Boundary Conditions 151
cgm 189
chamfer 327, 330
changing preferences 128
chop 286
circle 523
cleanup 312, 530, 777, 1146
clear 202
clip 171
closestpt 437, 1174
coarsening 784
cohesive element 1168
coincident nodes 732
collapse 434
angle 430
curve 432
mesh edges 772
surface 434
colors 173, 1557
Column Deletion 813
Column Groups 813
Column Insertion 813
Column Swapping 813
command 12
command echo 16
command line 11, 12
command syntax 12
command window 144
comment 27
Compare 88
Compare volume 88
component 126
composite 436
curves 436
surfaces 437
Condition No 745
condition number 735, 743, 745, 762
conductivity 900
Cone Surface 287
conformal 1148
Constraint 886
Contact Pairs 151
Contact Regions 151
contact set 928
contact surface 920, 1152
control skew 733
convection 151, 923
coordinates 1201
copy 280
body 280
mesh 512
scale 283
scheme 512
corner 545
cosine 1201
create 243
bottom-up 249
brick 244
curve 259
cylinder 246
primitives 247

pyramid 245
sheet 267
sphere 249
surface 266
torus 246
vertex 275
volume 250
credits 1573
ctrl-c 23
cubit file 24
cubit file method 24
cubit_geom.save.g 24
cubit_geom.save.sat 24
CUBIT_OPT 17
cubit-dev 5
curvature 524, 834
sizing function 843
curve 259
bias 521
block 886
copy mesh 512
create 259
extrude 266
intervals 506
list 56
nodeset 905
partitioning 440
plane normal to 288
sideset 905
split 294
tangent 492
trim or extend 415
type 545
valence 119
vertex on 275
customize 128, 140
cut
mesh 1174
cylinder 246
D
DART 470, 471, 472, 475
data filters 217
data structure 1567
date 1201
debug 19
decomposition 1152
geometry 1152
partitioning 444
split periodic 294
web cutting 292
Default Block 886
defeaturing
compositing 436
defeature tool 90
surface removal 338
tweaking geometry 329
deformations 821
degrees 1201
Delaunay 570
delete 492
density 900, 1172
development requests 8
diagonal ratio 735
dialog 133
command 133
options 128
property editor 148
tree view 59, 64
digits 17
dimension 876
direction 35
displacement 151, 925
display 202
display toolbar 152
distance 494
distortion 735, 743, 745, 751
Distributing 917
distribution 8
distribution factor 915
divergence 459
doubler 321
draw 176
color table 173
cylinder 293
drawing, locating, and highlighting 176
edge 214
group 449
histogram 740
location 32
nodeset 214, 905
normal 176
picked 223

plane 51
vertex types 545
Drawing and Highlighting Entities 176
Drawing Columns 813
dualbias 521, 733
Duplicate Block Elements 886
duplicating 512
E
echo 53
edge collapse 772
edge length 763, 1201
elastic modulus 900
element block 886, 903
element numbering 1561
element types 8, 497, 903
Element Volume 745
enclosure 909
end 545
enhancement requests 8
entity 249
curve 259
drawing 176
highlighting 176
labels 182
names 467
picking 223
selecting 122, 223
selection mode 122
specification 214
surface 266
tree 59
vertex 275
virtual 429
visibility 202
volume 250
Entity ID Spaces 873
environment 11
user settings 17
equal 525
equal_to 500
equipotential 764
error count 1201
error logging 16
errors 16, 1201
examples 939, 1021
execute 1201
execute button 133
execution 19
command syntax 19
Exhaust Fan 151
exit 25
exodus file method 899
element numbering 1561
exporting 879
file specification 899
importing 821
model title 900
sizing function 844
exodus II 821, 876
exotxt 876
expand 214
export 879, 884
extend 415
Extended Selection 205
Extended Selection Dialog 223
extraneous 340
F
facets 186, 490
factor 786
Fan 151
farfield pressure 151, 919
fastq 19, 1563
importing 486
FEA Boundary Conditions 151
feature angle 459, 484, 829
feature size 834, 1172
features 6
field function 844
file 23
acis 489
exodus II 899, 900
fastq 486
iges 491
initialization 23
input 19
journal 19, 27, 29
filename 12
step 491
fillet 324, 327, 330
find mesh intersection 755
find surface overlap 411
finite element model definition 903

fire ray 40
firmness 508
  interval 508
  scheme 723
  vertex type 545
Fixed_Imprints 548
flatquad 1168
flatshade 186
floor 1201
fluent 819, 883, 919
flush 202
fly-in 79
fonts 126
force 151, 341, 921, 1154
free elements 821
free mesh 727, 819, 824, 856
from 184, 197
frustum 245
fullhex 1567
fuzzy 723
G
gamma 1201
geometric entities 249
  curve 259
  surface 266
  vertex 275
  virtual 429
  volume 250
geometry 237
  analyzing 313
  attributes 462
  boolean 284
  bottom-up creation 249
  clean up 312
  creation 243
  debug 410
  decomposition 1152
  exporting 489
  healing 315
  importing 479
  merging 426
  modification 312
  primitives 247
  transformations 277
  validating 417
  virtual 429
  visibility 202
geometry adaptive 847
geometry associativity 819, 824
geometry deletion 492
geometry groups 449
geometry representation 239
Geometry Selection Filters 151
geometry tweaking 317, 326, 329, 330
Geometry/Mesh Comparison 88
Global Element ID 873
Global Element IDs 873
GlobalElement IDs 873
grafting 1181
graphical user interface 59
graphics 8, 170
  camera 184
  clipping plane 171
  colors 173
  display 202
  draw 176
  hardcopy 189
  labels 182
  mesh slicing 191
  modes 186
  no graphics option 19
  options 192
  selection 223
  views 200
  window size 188
graphics clip 171
graphics lighting 190
group 449
  graphical selection 451
  operations 449
  picked 226
  propagated hex 451
  quality 458
groups 449, 554
  sweep 554
  xor 449
H
hammer icon 6
hard interval 508
hardcopy 189
hardware platforms 6
healing 315

analyzing geometry 313
attributes 316
automatic 314
failure 316
heat flux 923
Heatflux 151
help 11, 19, 126, 140
hiddenline 186
highlight 176, 192
drawing, locating, and highlighting 176
histogram 738, 740
history command 14
hole 322, 525
htet 515
hypotenuse 1201
I
id input field 133
id maps 877
idealize 317
ideas 879
i-deas 832
i-deas 879
idless journal file 30
ids 465, 1201
iges 414, 486, 491
import 479, 480, 482, 486, 487, 818, 819, 820, 821, 832, 833, 1138
imprint 423
mesh 439
improve 485
info 53, 1575
initialization file 19, 23
Inlet Mass Flow 151
inlet massflow 919
inlet pressure 151, 919
Inlet Velocities 151
inlet velocity 919
Inlet Vent 151
input 884
input file 19
input window 144
Inradius 745
inria 558
installation 8
Intake Fan 151
Interface 151
interior mesh elements 171
Interiors 151
interrupt 23, 140
intersect 284
intersecting 755
interval 499
automatic specification 503
explicit specification 502
firmness 508
matching 506
periodic 510
relative 509
Interval equal_to 500
Interval same 500
isoparameter 176
isoparametric 176
J
jacobian 735, 743, 745
journal file 19
APREPRO 1210
automatic creation 27
creation and playback 29
editor 146
playback 27
recording 27
K
key icon 6
key press commands 119
Kinematic 917
L
label 182
laplacian smoothing 764
length 1201
license 8, 17
light intensity 190
lighting model 190
limit plane 317, 330
line width 192
listing information 52
environment 53
geometry 56
mesh 57
model summary 58
special entities 58
vertex types 545
load set 928

loads 921
Local refinement 786
locate 79
  drawing, locating, and highlighting 176
Locate command 176
location 38
location on curve 43
**loft** 256
logarithm 1201
logging 53
lowercase 1201
ls command 14
ls-dyna 879
M
Machine Learning 66, 105
magic mesh 1129
mailing lists 5
make solid 250
mapping 526
material 472, 898, 900
mathematical functions 1201
mean ratio smoothing 765
measurement 494
memory usage 1567
menu 126, 133
merge nodes 831
merge tolerance 424, 1156
merging 426
  examining merged entities 423
  tolerance 425
  using to verify geometry 428
mesh 497
  and BC Entity Visibility 202
  collapse element 772
  copy 512
  creation 497, 1171
  deletion 1171
  density 834
  import 818
  interval 499
  modification 759
  procedure 498
  quality 83, 171, 737, 738, 740
  scheme 510
  tools 80
mesh based geometry 239
  adaptive 512, 834
  algorithms 510
  deletion 867
  export 876
  feature 7
  import 821
  interval assignment 499
  meshedit 772
  preview 509
  process 497
  quality 737
  remesh 498
  sizing function 834
  slicing 191
  smoothing 766
  transform coordinates 884
  validity 867
  visibility 202
Mesh Column Operations 813
Mesh Scaling 802
Mesh Selection Filters 151
mesh topology 748
meshing in item 1139
metadata 470
Metadata Attributes 85
metric 737
metric name 740
  algebraic 740
  allmetrics 740
  robinson 740
  traditional 740
middle mouse button 122, 124, 128
midplane 266
min_through_thickness 786
mod 1201
model axis 176, 192
morph smooth 512
mouse 124, 197
  customization 128
  right click 120
  selecting entities with 122, 223
  view navigation 124, 197
move 278, 281
msc 558, 571
multisweep 548

N
name 467
narrow regions 342
nastran 879
navigation 124, 197
ncdump 876
negative Jacobians 769
Neighbor 834
netcdf 876
new 24
next 28
node 810
  coincident 732
  fix position 766
  nodehex 1567
  nodeset 905
  numbering 1561
  repositioning 810
  selection 223
Node Constraint 886
Node Redistribution 548
nodeset 151, 905
  importing 821
  repositioning 810
  size 192
  smoothing 766
  visibility 202
nogui 19
non-manifold topology 7, 748
normal 176, 492
NOT_SET 1213
notation 12
numbering 1561
numeric 12
NumInGrp 1201
O
offset 250, 262, 266, 317, 331
open 24
openGL 126
optimize jacobian smoothing 1185
options 128
orthogonal 759
Outflows 151
outlet pressure 151, 919
Outlet Vents 151
output 884
output window 144
Overconstrained 571
Overconstrained Edges 571
overlap 411
P
painters 186
pan 124, 197, 204
parallel 786, 877
Parallel Meshing 614
parse 38
part 470, 471, 472, 475
partition 438, 444
  curves 440
  surfaces 441
  volumes 443
patch 908
patran 833, 879
pause 23
pave 528
pentagon 532
periodic 919, 1551
Periodic Shadows 151
Periodics 151
perspective 184
Pick Extended 205
pick toolbar 122
picked group 62, 226
picking 223
pict 189
pillow 811
pinpoint 533
planar 217
plane 46
playback 29
point 192
poisson ratio 900
polygonfill 186
polyhedron 534
Porous Jump 151
postscript 189
ppm 189
preselection 122
pressure 151, 922
preview 32, 293, 509
  axis 32
  direction 32, 38

imported mesh 1146
location 32
mesh 509
plane 51
primitive 247
brick 244
cylinder 246
prism 248
pyramid 245
sphere 249
torus 246
print error 1201
prism 248
problem reports 8
propagate curve bias 734
property editor 148
pwd 14
pyramid 245
python 1212
Q
QA_Records 25
qtri 517
quality 737
controlling skew 733
groups 458
hexahedral 735
higher order 1194
quadrilaterals 743
tetrahedral 745
tools 83
triangles 751
wedge 753
quick reference 1575
quit 25
quote 1201
R
radialmesh 727
radians 1201
Radiator 151
radius 1201
random 1201
randomize 1185
ray 40
rebar 892
record 29
references 1554
refine 786, 1186
Refine Mesh Sheet 786
Refine Mesh Volume 786
Refine min_through_thickness 786
Refine Tet_edge 786
reflect 282
regularize 414
relative size 735, 743, 745, 751
remesh 498
removal 316, 317, 338, 340
remove 444, 548
Remove Tiny Edge Length 1187
remove topology 335
renumber ids 466
Renumbering Element Blocks 886
Renumbering Nodesets and Sidesets 905
repositioning nodes 810
rescan 1197, 1201
reset 25
respect tetmesh 558
restart 24
restore 24
restraint 925
restraint set 928
resume 23
reversal 545
Reverse 492
right click options 120
rotate 124, 197, 204, 278, 283
rotation 283
S
same 500
save 24
save as 24
scale 283
Scale Mesh 802
scaled jacobian 735, 743, 745, 751
scheme 510
automatic selection 723
bias 521
circle 523
curvature 524
delaunay 571
dualbias 521
equal 525

featuresize 1172
firmness 723
hole 525
htet 515
mapping 526
multisweep 548
parallel 614
pave 528
pentagon 532
pinpoint 533
polyhedron 534
qtri 517
sculpt 583
selection 723
sphere 536
stransition 540
stretch 542
submap 542
sweep 548
tetinria 558
tetmesh 558
tetmsc 558
tetprimitive 568
thex 518
transition 1190
triadvance 571
tridelaunay 570
trimap 570
trimesh 571
tripave 580
triprimitive 581
Sculpt 583
Sculpt Application 610
Sculpt Installation 583
Sculpt Parallel Command 583
Sculpt Technical Description 605
section 311
seed 459
selection 122, 214
separate 312
session control 11
session id 1201
Set Default Block 886
Set Duplicate Block Elements 886
set Node Constraint 886
shape 735, 743, 745, 751
shear modulus 900
sheet 250
sheet body 217, 266, 312, 414
shell 904
side 545
sideset 151, 905
Sierra 879
silhouette 192
simplify 335, 446
simulog 558
sine 1201
size 192, 786
auto 503
feature 1172
interval 502
sizing function 733, 834
bias 836
constant 842
curvature 843
exodus II 844
field 844
interval 853
inverse 854
linear 855
super 1188
test 1189
skeleton sizing 834, 847
skew 310, 733, 735, 743
skew control 733
skinning 269, 868
sliver surface 339, 424
slot 322
small curves 343, 1158
small feature 1158, 1163
small surfaces 344, 1158
smart laplacian 768
Smart_Smooth 548
smoothing 766
centroid area pull 762
edge length 763
equipotential 764
facets 241
laplacian 764
optimize condition number 762
optimize jacobian 1185
optimize untangle 769

randomize 1185
winslow 771
smoothshade 186
soft interval 508
solid model 19
SolidDesigner 239
SolidWorks 239
sort 465
specific heat 900
sphere 249, 536
spider 894
spline 217, 260
split 311
body 292
curve 294
periodic 294
surface 294
Split Overconstrained 571
Split Overconstrained Edges 571
sqrt 1201
step 414, 487, 491
stitch 414
stop 23
stransition 540
stray 340
stretch 542, 735, 743
string 12, 1201
sub-assembly 470, 475
submap 542
subtract 284
SUPERELEMENT_TOPOLOGY_XX 886
SUPERELEMENT_TOPOLOGY_XX. 886
surface 250, 266
creation 266
normal 492
overlap 411
removal 338
vertex type 545
surface area 1201
sweep 250, 266, 548
sweep group 554
sweep surface 250
Symmetries 151
symmetry 919
syntax 12
T
tangent 492
taper 735, 743
target 319
temperature 151, 928
Tet_edge 786
tetdice 558
tetmesh 558
tetmsc 558
tetprimitive 568
text size 192
thex 518
thicken 250
threshold 731
Tie 917
tile 1551
Timestep 745
Tiny Edge 1187
title 900
toggle 12
tolerance 414
tolerant imprinting 424
toolbars 151
topology 7
torus 246
tquad 520
transform 884
transformations 277
transition 1190
transition map 540
translation 281
translators 879
transparent 186
triad 192
triadvance 571
triangle coarsening 1192
triangle visibility 176
tridelaunay 570
trim 415
trimap 570
trimesh 571
tripave 580
triprimitive 581
tutorial 939
gui 1078

non-gui 1106
power tools 969
tweak 329
curve 330
remove topology 335
surface 317
vertex 326
volume bend 328
U
unite 285
units 1201
unmerge 428
untangle 769
up command 184, 197
uppercase 1201
usage 19
user environment settings 17, 19
USER_SET 1213
users manual 1
V
valence 119
validation 417
variable 19, 1197
Velocity 151
verify 428
version 16, 489
vertex 275
vertex size 501
Vertex Sizing 501
vertex type 545
view 184, 191, 192, 197
virtual geometry 429
collapse 434
composite 436
deleting 444
partition 444
simplify 446
visibility 202
void 495
volume 250
curve type 545
draw 176
in volume 1201
measurement 494
partitioning 443
quality metrics 735, 745
VRML 879
W
Walls 151
warning 19, 53
warning count 1201
webcut 292
chop 286
options 293
sweep 289
with arbitrary surface 286
with planar or cylindrical surface 287
with tool body 291
where 28
window 140
application 140
command 144
control panel 133
drop-down menu 126
entity tree 59
graphics 118, 170
input 144
journal file editor 146
output 144
property 148
query select 122
toolbar 151
windowlocation 188
winslow smoothing 771
wireframe 186
word count 1201
workbook mode 126
working directory 14
Z
zoom 197, 204