Cubit 15.0 User Documentation

Table of Contents

1. **CUBIT 15.0 User Documentation** 1

2. **Key Features** 3
   - Geometry Creation, Modification, and Healing 3
   - Non-Manifold Topology 3
   - Geometry Decomposition 3
   - Mesh Generation 3
   - Boundary Conditions 3
   - Element Types 4
   - Graphics Display Capabilities 4
   - Graphical User Interface 4
   - Command Line Interface 4

3. **Hardware Requirements** 5

4. **Licensing and Distribution** 7

5. **Trademark Notice** 9

6. **How to Use This Manual** 11

7. **Introduction** 13

8. **CUBIT Mailing Lists** 15

9. **Problem Reports and Enhancement Requests** 17

10. **Starting and Exiting a CUBIT Session** 19
    - Starting the Session 19
    - Windows File Association 19
    - Exiting the Session 19
    - Resetting the Session 19
    - Abort Handling 19

11. **Execution Command Syntax** 21
    - Passing Variables into a CUBIT Session 23

12. **Initialization Files** 25

13. **Environment Variables** 27

14. **Command Syntax** 29

15. **Command Line Help** 31

16. **Environment Commands** 33
Working Directory 33
File Manipulation 33
CPU Time 33
Comment 34
History 34
Error Logging 34
Determining the CUBIT Version 34
Echoing Commands 34
Digits Displayed 34

17. **Saving and Restoring a Cubit Session** 37
CUBIT File Method 37
New 37
Open '<filename>' 37
Save 37
Import 38
Export 38

18. **Interrupting Running Tasks** 39

19. **CUBIT Application Window** 41
Context Sensitive Help in the GUI 42
Customizing the Application Window 42

20. **Command Panel Functionality** 45
ID Input Entry Methods 46
Right-Click Context Menu for ID Input Fields 47
Value Fields 47
Advancing Pickwidgets 48

21. **View Navigation in the GUI** 49
Rotations 49
Zooming 50
Panning 51

22. **Selecting Entities in the GUI** 53
Pre-Selection 53
Polygon and Box Select 54

23. **Key Press Commands for the GUI** 55

24. **Right Click Commands for the GUI Graphics Window** 57
With Entity Selected 57
Without Entity Selected 57

25. **Viewing Curve Valence** 59

26. **Geometry Tree** 61
Drag and Drop 63
Picked Group 63
Right-Click Menu Functions 63

27. **Geometry Power Tools** 65
Geometry Analysis Tools 65
Geometry Repair Tools 67
Right Click Menu 69

28. **Meshing Tools** 71
Right Click Context Menu 71

29. **Mesh Quality Tools** 73
Mesh Quality Tool Buttons 74
Right-Click Context Menu Items 74

30. **Property Editor** 75
Editing Entity Attributes from the Property Editor 76
General Attributes 76
Geometry Attributes 76
Meshing Attributes 76
Boundary Condition Attributes 76
Metadata Attributes 77

31. **Command Line Workspace** 79
Command Window 79
Entering Commands 79
Repeating Commands 79
Focus Follows Cursor 79
Error Window 80
History Window 80
Script Window 80
Docking and Undocking the Input Window 80

32. **Journal File Editor** 81
Journal Editor Toolbar 82
Other Functionality Available in the Journal Editor 82

33. **Toolbars** 83
File 83
Display 83
Select 84

34. **Options Menu** 85
Custom Tools 85
Display Preferences 85
General Preferences 85
Geometry Defaults 86
History Preferences 86
Cubit History Preferences 86
Label Defaults 86
Layout Preferences 86
Cubit Layout Settings 87
Mesh Defaults 87
Mouse Settings 87
Post Processor Settings 87
Quality Defaults 87

35. **Creating Custom Toolbar Buttons** 89

36. **Undo Button** 91
Limitations 91

37. **Journal File Creation and Playback** 93
Recording a Session 93
Replaying a Session 93

38. **Controlling Playback of Journal Files** 95

39. **Automatic Journal File Creation** 97
Controlling Automatic Journal File Creation 97
Recording Graphics Commands 97
Recording Entity IDs and Names 97
Recording APREPRO Commands 97
Recording Errors 97

40. **Idless Journal Files** 99

41. **Command Line View Navigation: Zoom, Pan and Rotate** 101
Rotation 101
Panning 101
Zooming 101

42. **Mouse Based View Navigation: Zoom, Pan and Rotate** 103
Changing the View Transformation Button Bindings 104
Saving and Restoring Views 105

43. **Updating the Display** 107
Prevent Graphics From Updating 107

44. **Graphics Modes** 109
Displaying Using the Element Facets 110
Displaying Composite Surface Lines 110

45. **Drawing, Locating, and Highlighting Entities** 111
Drawing Other Objects 111
Displaying Entity Orientation 111
Volume Sources and Targets 112
Model Axis 112
Surface Isoparameter Lines 112
Surface Overlap 112
Volume Overlap 112
Geometry Preview 113

46. **Mesh Visualization** 115
Notes on Mesh Slicing 115
Mesh Slicing Command 115

47. **Graphics Clipping Plane** 117
Examples 118

48. **Entity Labels** 119

49. **Colors** 121
Specifying Colors in Commands 121
User-Defined Colors 121
Assigning Colors 122
Assigning Global Colors 122

50. **Geometry, Mesh, and BC Entity Visibility** 125

51. **Graphics Camera** 127
Changing Camera Attributes Directly 127

52. **Graphics Lighting Model** 129

53. **Graphics Window Size and Position** 131
Using Multiple Windows 131

54. **Saving Graphics Views** 133

55. **Hardcopy Output** 135
Screen Capture Programs 135

56. **Miscellaneous Graphics Options** 137
Silhouette Lines 137
Line Width 137
Highlight Line Width 137
Text Size 137
Point Size 138
Graphics Status 138
Graphics Scale 138
Model Axis 138
Corner Axis (Triad) 138
Resetting the Graphics 138

Shrink 139
Facet Tolerance 140

57. **Command Line Entity Specification** **143**
Types of Entity Range Input 143
Precedence of "Except" and "In" 144
Placement in CUBIT Commands 145

58. **Entity Selection** **147**

59. **Environment Control** **149**

60. **Extended Command Line Entity Specification** **151**
Extended Parsing Syntax 151
Keywords 151
Functions 152
Precedence 153

61. **Selecting Entities with the Mouse** **155**
Entity Selection 156
Query Selection 156
Multiple Selected Entities 156
Information About the Selection 157
Picked Group 157
Substituting Selection into Other Commands 157

62. **Specifying a Location** **159**
Position (XYZ values) 159
Last Location Used in a Command 159
Node or Vertex 159
On a Curve 159
On a Surface 160
On a Plane 160
Center 160
Extrema 160
Fire Ray 160
Between 160
Move 161
Swing 161
Multiple Location Specification 162
Previewing a Location 162

63. **Specifying a Location on a Curve** **163**
Arc Center 163
Start, Midpoint, or End 163
Fraction 163
Distance 163
{Close_To|At} Location 164
Extrema 164
Segment 164
Crossing 164
Previewing a Location on a Curve 165

64. **Specifying a Direction** 167
Vector (XYZ values) 167
Last Direction Used 167
Positive or Negative X,Y,Z Direction Vectors 167
On Curve Tangent 167
On Surface Normal 168
From Location 168
Rotate 168
Cross 169
Reverse 169
Previewing a Direction 169

65. **Specifying an Axis** 171
Last 171
Specify an origin and a vector 171
Revolve an axis about an axis 171
Previewing an Axis 172

66. **Specifying a Plane** 173
Location and Normal Vector 173
Location and Two Vectors on the Plane 173
Two Locations and Vector on the Plane 174
Three Points on the Plane 174
Plane defined by a Surface 175
Plane Normal to a Curve 175
Plane Defined by a Non-linear curve 176
Plane Defined by a two linear curves 176
Normal Vector and Coefficient 176
Coordinate Plane 177
Last Location Used 177
Previewing a Plane 177
Preview a Cylindrical Plane 177

67. **Drawing a Location, Direction, or Axis** 179

68. **List Model Summary** 181

69. **List Geometry** 183

70. **List Mesh** 185

71. **List Special Entities** 187

72. **List Cubit Environment** 189
Message Output Settings 189
Graphical Display Information 191
Memory Usage Information 191

73. **ACIS Geometry Kernel** 193

74. **Mesh-Based Geometry** 195
Creating Mesh-Based Geometry Models 195

Improving Mesh-Based Geometry Models for Meshing 196
Meshing Mesh-Based Models 197
Exporting Mesh-Based Geometry 197

75. **Importing ACIS Files** **199**
Import Options 199
Importing ACIS files at startup 199

76. **Importing FASTQ Files** **201**

77. **Importing STEP Files** **203**
Import Options 203
Exporting a STEP file from Pro/Engineer 203

78. **Importing IGES Files** **205**
Import Options 205

79. **Importing Facet Files** **207**
Facet File Format 208
Feature Angle 208
Smooth Curves and Surfaces 208
Merge 208
Make elements 208
Stitch 209
Improve 209

80. **Importing Granite Files** 211

81. **Creating Vertices** 213

82. **Creating Curves** 215

83. **Creating Surfaces** 219

84. **Creating Volumes** 227

85. **Creating Bricks** 235

86. **Creating Cylinders** 237

87. **Creating Prisms** 239

88. **Creating Frustums** 241

89. **Creating Pyramids** 243

90. **Creating Spheres** 245

91. **Creating Toruses** 247

92. **Align Command** 249

93. **Copy Command** 251

94. **Move Command** 253
- Moving Other Geometric Entities 253
- Moving Bodies Relative to Other Geometric Entities 253
- Moving Merged Entities 253
- Move Undo 254

95. **Scale Command** 255

96. **Rotate Command** 257
- Rotating Merged Entities 257

97. **Reflect Command** 259

98. **Intersect** 261

99. **Subtract** 263

100. **Unite** 265

101. **Chop Command** 267

102. **Web Cutting by Sweeping Curves or Surfaces** 269
Web Cutting by Sweeping a Surface Along a Trajectory 269
Web Cutting by Sweeping a Surface About an Axis 270
Web Cutting by Sweeping a Curve(s) Along a Trajectory 270
Web Cutting by Sweeping a Curve(s) About an Axis 270

103. **Web Cutting Options** 271

104. **Web Cutting with a Planar or Cylindrical Surface** 273
Coordinate Plane 273
Planar Surface 273
Plane from 3 Points 273
Plane Normal to Curve 273
General Plane Specification 274
Cylindrical Surface 274
Cone Surface 274

105. **Web Cutting using a Tool or Sheet Body** 275

106. **Web Cutting with an Arbitrary Surface** 277

107. **Split Curve** 279

108. **Split Periodic Surfaces** 281

109. **Split Surface** 283
Split Across 283
Split Extend 284
Split (Automatically) 286
Split Skew 297

110. S
ection Command 299

111. S
eparating Multi-Volume Bodies 301

112. S
eparating Surfaces from Bodies 303

113. A
nalyzing Geometry 305
Healer Settings 305

114. H
ealing Attributes 307

115. A
uto Healing 309

116. S
pline Removal 311

117. W
hat if Healing is Unsuccessful? 313

118. T
weaking Vertices 315
Tweaking a Vertex With a Chamfer 315
Tweaking a Vertex With a Non-Equal Chamfer 315
Tweaking a Vertex With a Fillet Radius 316

119. T
weaking Curves 317
Create a Chamfer or Fillet 317
Tweaking a Curve Using an Offset Distance 318
Removing a Curve 318
Tweaking a Curve Using Target Surfaces, Curves, or Plane 319
Tweaking a Pair of Curves to a Corner 320

120. T
weaking Surfaces 323
Tweaking a Surface Using an Offset 323
Tweaking a Surface by Moving 323
Tweaking Surfaces to Target Surfaces 324
Removing a Surface 324
Tweaking a Conical Surface 324
Tweaking Doublers to Target Surfaces 325
Removing Holes and Slots from Sheet Bodies 327
Removing Fillets from Sheet Bodies 328

121. T
weak Remove Topology 331
Example 331

122. T
weak Volume Bend 335

123. R
emoving Vertices 337

124. R
emoving Surfaces 339
Remove Sliver Surface 339

125. A
utomatic Forced Sweepability 341

126. A
utomatic Small Curve Removal 343

127. A
utomatic Small Surface Removal 345

128. A
utomatic Surface Split 347

129. R
egularizing Geometry 349

130. F
inding Surface Overlap 351
Facetted Representation 351
Find Overlap Settings 351

131. V
alidating Geometry 355

132. D
ebugging Geometry 357

133. G
eometry Accuracy 359

134. T
rimming and Extending Curves 361
Trimming a Curve 361
Extending a Curve 362

135. S
titching Sheet Bodies 363

136. I
mprinting Geometry 365
Regular Imprinting 365
Tolerant Imprinting 365
Mesh-Based Imprinting 366

Imprint Settings 366

137. **Merging Geometry** **367**
Merge geometry automatically 367
Test for merging in a specified group of geometry 367
Force merge specified geometry entities 367
Preventing geometry from merging 367
Other Merge Commands 368

138. **Examining Merged Entities** **369**

139. **Merge Tolerance** **371**
Finding Nearly Coincident Entities 371

140. **Unmerging** **373**

141. **Using Geometry Merging to Verify Geometry** **375**

142. **Composite Curves** **377**

143. **Composite Surfaces** **379**
Controlling the Surface Evaluation Method for Composite Surfaces 379
Composite Determination 379

144. **Partitioned Curves** **381**

145. **Partitioned Surfaces** **383**
Partitioning with Vertices and Nodes 383
Partitioning with Hard Points 383
Partitioning with Polylines 383
Partitioning with Curves 384
Partitioning with Mesh Edges 384
Partitioning with Faces or Triangles 384

146. **Partitioned Volumes** 385

147. **Using Mesh Intersections to Partition Surfaces** 387

148. **Removing Partitions** 389

149. **Collapse Angle** 391

150. **Collapse Curve** 395

151. **Collapse Surface** 397

152. **Simplify Geometry** 399
Feature Angle 400
Automatically Compositing Curves 400
Respecting Vertices, Curves and Surfaces 400
Respecting Imprints 400
Using Local Normals 401
Other Options 401

153. **Deleting Virtual Geometry** 403
Removing Virtual Geometry 403
Using The Delete Command With Composites 403
Using the Delete Command With Partitions 403

154. **Geometry Orientation** 405
Adjusting Orientation 405

155. **Basic Group Operations** 407
Geometry Groups 407
Modifying groups by comparing common entities 407
Group Booleans 407
Mesh Groups 408
Group Copy 408
Group Transformations 408
Deleting Groups 409
Cleaning Out Groups 409

156. G
roups in Graphics 411

157. P
ropagated Hex Groups 413
Propagated Hex Group Starting on a Surface 413
Ending at a Surface 413
Number of Times 413
Ending at a Surface with Multiple 413
Number of Times with Multiple 414
Ending at Surface with Direction 414
Number of Times with Direction 414
Propagated Hex Group Starting on a Face 414
Ending at a Surface 415
Ending at a Face 415
Number of Times 415
Ending at a Surface with Multiple 415
Ending at a Face with Multiple 415
Number of Times with Multiple 416
Ending at Face with Direction 416
Ending at Surface with Direction 416
Number of Times with Direction 416
Naming Convention for Propagated Hex Groups 417

158. S
eeded Mesh Groups 419

159. Q
uality Groups 423

160. E
ntity Names 425
Valid and Invalid Names 425
Reconciling Duplicate Names 425
Automatic Name Creation 425
Automatic Name Propagation 426
Naming Merged Entities 426

161. E
ntity IDs 429
Element Ids 429
Gaps in ID space 429
Renumbering IDs 429
Volume ID 429

162. A
ttribute Behavior 431

163. A
ttribute Types 433

164. A
ttribute Commands 435
Control By Attribute Type or Geometric Entity 435

165. U
sing CUBIT Attributes 437

166. E
ntity Measurement 439
Measure Between 439
Measure Small 439
Measure Angle 439
Measure Void 440

167. W
orking With Parts and Assemblies 441
Identifying Parts and Assemblies 441
Creating Parts and Assemblies 441
Deleting Parts and Assemblies 441
Associating Parts with Volumes 442
Viewing All Assembly Information at Once 442
Metadata in the GUI 443

168. M
etadata Attributes 445
Part and Assembly Metadata Attributes 445
Viewing Part and Assembly Metadata Attribute Values 446
Modifying Metadata Attributes 446
Viewing and Modifying Global Metadata 446

169. I
mporting and Exporting Metadata 449
Importing Metadata 449
Exporting Metadata 449
Importing and Exporting DART Artifacts 449

170. E
xporting ACIS Files 451

171. E
xporting STEP Files 453

172. E
xporting IGES Files 455

173. E
xporting Facet Files 457

174. G
eometry Deletion 459

175. M
eshing the Geometry 461
Default Scheme and Interval Selection 461
Continuing Meshing After a Mesh Failure 461

176. I
nterval Firmness 463
Precedence 463

177. E
xplicit Specification of Intervals 465

178. A
utomatic Specification of Interval Size 467
Automatic Interval Size Specification 468
Maximum Spanning Angle on Arcs 468

179. I
nterval Matching 471

180. P
eriodic Intervals 473

181. R
elative Intervals 475

182. M
esh Interval Preview 477

183. B
ias, Dualbias 479

184. C
ircle 481

185. C
urvature 483

186. E
qual 485

187. H
ole 487

188. M
apping 489

189. P
ave 491
Element Shape Improvement 491
Controlling Flattening of Elements 491
Controlling the Grid Search for Intersection Checking 492
Controlling the Paver Sizing Function 492
Controlling Paver Cleanup 492

190. P
entagon 495

191. P
inpoint 497

192. P
olyhedron 499

193. S
phere 501

194. S
Transition 503

195. S
tretch 505

196. S
tride 507

197. S
ubmap 509

198. S
urface Vertex Types 511
Surface Vertex Commands 511
Listing and Drawing Vertex Types 512
Triangle Vertex Types 512
Adjusting the Automatic Vertex Type Selection Algorithm 512
Volume Curve Types 513

199. S
weep 515
Multisweep 517
- Grouping Sweepable Volumes 519

Node Redistribution 519

200. T
etMesh 523
Discussion 523
Tetmesher Options 524
Using tets as the basis of an unstructured hexahedral mesh 525
Conforming the tetmesh to internal features 525
Controling the gradation of mesh size inside the volume 526
Generating a Tetmesh from a Skin of Triangles 526

201. T
etprimitive 529

202. T
riDelaunay 531

203. T
riAdvance 533

204. T
riMap 535

205. T
riMesh 537

206. T
riPave 539

207. T
riPrimitive 541

208. R
adialmesh 543

209. H
Tet 549
Unstructured 549
Structured 549

210. Q
Tri 551

211. T
Hex 553

212. T
Quad 555

213. C
opying a Mesh 557

214. M
irroring a Mesh 559

215. A
utomatic Scheme Selection 561
Default Scheme Selection 561
Auto Scheme Selection General Notes 561
Scheme Firmness 562
Surface Auto Scheme Selection 562
Volume Auto Scheme Selection 563

216. M
etrics for Edge Elements 565
Quality Metric Definitions: 565
Comments on Algebraic Quality Measures 565

217. M
etrics for Triangular Elements 567
Approximate Triangular Quality Definitions: 567
Comments on Algebraic Quality Measures 567
References for Triangular Quality Measures 568

218. M
etrics for Quadrilateral Elements 569
Quadrilateral Quality Definitions 569
Comments on Algebraic Quality Measures 570
References for Quadrilateral Quality Measures 570
Details on Robinson Metrics for Quadrilaterals 570

219. M
etrics for Tetrahedral Elements 573
Tetrahedral Quality Definitions 573
References for Tetrahedral Quality Measures 574

220. M
etrics for Hexahedral Elements 575
Hexahedral Quality Definitions 575
References for Hexahedral Quality Measures 576

221. M
esh Quality Command Syntax 577
Quality Options 577
Scope 577
Draw 577
List 578
Filter 578

222. M
esh Quality Example Output 581

223. A
utomatic Mesh Quality Assessment 583

224. C
ontrolling Mesh Quality 585
Skew Control 585
Propagate Curve Bias 585
Adjust Boundary 585

225. C
oincident Node Check 587

226. M
esh Topology Check 589

227. C
entroid Area Pull 593

228. E
quipotential 595

229. L
aplacian 597

230. S
mart Laplacian 599

231. C
ondition Number 601

232. M
ean Ratio 603

233. W
inslow 605

234. U
ntangle 607

235. E
dge Length 609

236. M
esh Refinement 611
Global Mesh Refinement 611
Refining at a Geometric or Mesh Feature 612
Hexahedral Refinement Using Sheet Insertion 614
Refining at a Geometric Feature 614
Refining along a path 615
Refining a Hex Sheet 615
Directional Refinement 617
Hex Sheet Drawing 619
Local Refinement of Tets, Triangles, and Edges 619
Single Entity Refinement 620
'N' Tets Through the Thickness Refinement 620
Parallel Refinement 622
Scaling the Number of Elements in a Mesh 622

237. M
esh Pillowing 625

238. M
esh Coarsening 627
Hexahedral Coarsening 627
Extracting a Single Hex Sheet 627
Extracting multiple sheets along a curve 627
Uniform hex coarsening 628

239. N
ode and Nodeset Repositioning 629

240. C
ollapsing Mesh Edges 631

241. A
lign Mesh 633

242. C
reating and Merging Mesh Elements 635
Creating Mesh Elements 635
Creating Hex and Tet Elements 635
Creating Wedge Elements 636
Creating Face and Tri Elements 637
Creating Edge Elements 637
Creating Nodes 637
Merging Nodes 637

243. C
leaning Up a Tetrahedral Mesh 639

244. M
esh Validity 641

245. G
eometry Adaptive Sizing Function (Skeleton Sizing) 643
Skeleton Sizing Behaviors 645
Command Line Syntax 645
Basic Arguments 645
Scaling and Accuracy Arguments: 645
Advanced Arguments 646
Lattice Arguments: 646
Source Entity Arguments 646
Skeleton with Other Sizing Controls 646
Limitations 647

246. B
ias Sizing Function 649

247. C
onstant Sizing Function 655

248. C
urvature Sizing Function 657

249. L
inear Sizing Function 659

250. I
nterval Sizing Function 661

251. I
nverse Sizing Function 663

252. E
xodus II-based Field Function 665
Curve Meshing with Exodus II - based Field Functions 666

253. I
mporting Exodus II Files 667
Importing a Free Mesh Without Geometry 667
Importing a Mesh Onto Existing Geometry 668
Importing a Mesh with Nodeset Associativity 668
Importing a Mesh onto Modified Geometry 668
Mesh Import Tolerance 668
Specifying a Portion of the Mesh to be Imported 669
Unique Genesis IDs and Shell Options 669
Nodeset Ordering 669
Creating Mesh-Based Geometry on Import 669
File Name 669
Blocks 669
Unique Genesis IDs 670
Start ID 670
Nodesets/Sidesets 670
Feature Angle 671
Smooth Curves and Surfaces 671
Apply Deformations 672
Merge 673
Merge Nodes 673
Export Facets 673

254. I
mporting Abaqus Files 675

255. I
mporting I-DEAS Files 677

256. I
mporting Patran Files 679

257. I
mporting 2D Exodus Files 681

258. M
esh Deletion 683
Automatic Mesh Deletion 683

259. F
ree Meshes 685
Creating a free mesh 685
Disassociating a mesh from its geometry 685
Creating Mesh-Based Geometry to fit a Free Mesh 685
Merging a free mesh 686
Free Mesh Transformation Operations 686
Extruding Mesh Elements 686
Offsetting Mesh Elements 687
Revolving Mesh Elements 688
Smoothing a free mesh 689
Mesh quality on a free mesh 689
Mesh refinement on a free mesh 690
Cleaning up a free mesh 690
Assigning boundary conditions 690
Skinning a free mesh 691
Deleting free mesh elements 691
Bottom-up element creation 692
Exporting free meshes 692

260. S
kinning a Mesh 693

261. E
lement Block Specification 695
Creating Element Blocks 695
Assigning a Name or Description to an Element Block 696
Defining the Element Type 696
Default Element Blocks 696
Duplicate Block Elements 697
Assigning Attributes to Blocks 697
Displaying Element Blocks 698
Deleting Element Blocks 698
Renumbering Element Blocks 698
Automatically Assigning Mesh Edges to a Block (Rebar) 699
Diagonal and Orthogonal Rebar Blocks 699
Specifying a set of nodes 700
Creating Spider Blocks 700
Creating Beam Blocks 701

Creating Spring Blocks 702
Creating Sphere Blocks 702
2D Elements 703
Mixed Element Output 704
Adding Materials to a Block 704

262. **Nodeset and Sideset Specification** 705
Creating Nodesets and Sidesets 705
Useful hint: 706
Assigning Names and Descriptions to Nodesets and Sidesets 706
Grouping Faces on a Surface into a Sideset 706
Grouping elements in voids and enclosures 707
Deleting Nodesets and Sidesets 707
Renumbering Nodesets and Sidesets 708
Displaying Nodesets and Sidesets 708
Nodeset Associativity Data 708
Equation-Controlled Distribution Factors 709
Nodesets/Sidesets/Blocks Behavior with Geometric Entity Copy 710

263. **Nodeset and Sideset Specification** 711
Creating Nodesets and Sidesets 711
Useful hint: 712
Assigning Names and Descriptions to Nodesets and Sidesets 712
Grouping Faces on a Surface into a Sideset 712
Grouping elements in voids and enclosures 713
Deleting Nodesets and Sidesets 713
Renumbering Nodesets and Sidesets 714
Displaying Nodesets and Sidesets 714
Nodeset Associativity Data 714
Equation-Controlled Distribution Factors 715
Nodesets/Sidesets/Blocks Behavior with Geometric Entity Copy 716

264. **Exodus II File Specification** 717
Exodus II Manual 717
Element Block Definition Examples 717
Multiple Element Blocks 717
Surface Mesh Only 717
Two-dimensional Mesh 717

265. **Exodus II Model Title** 719

266. **Exodus Coordinate Frames** 721

267. **Defining Materials and Media Types** 723

268. **Boundary Condition Sets** 725
*** ABAQUS Parameters *** 725
*** NASTRAN Parameters *** 725

269. U
sing Restraints 727
Displacements/Accelerations/Velocities 727
Fixed or Free 728
Displacement Combinations 728
Temperature 728
Top, Gradient, Middle, Bottom 729

270. U
sing Loads 731
Forces 731
Using Pressure 731
Value 731
Pressure and Total Force 732
Top and Bottom 732
Using Heat Flux 732
Top and Bottom Values 732
Using Convection 732
Surrounding 732
Coefficient 732

271. U
sing Contact Surfaces 733
The Contact Region 733
The Contact Pair 733
Auto-Contact Tool 733

272. U
sing Contact Surfaces 735
The Contact Region 735
The Contact Pair 735
Auto-Contact Tool 735

273. U
sing CFD Boundary Conditions 737
Inlet Velocity 737
Inlet Pressure 737
Inlet Massflow 737
Outlet Pressure 737
Farfield Pressure 737
Symmetry 737

274. M
iscellaneous Boundary Condition Commands 739
Delete 739
List 739
Draw 739
Highlight 739

275. E
xporting Sierra Files 741

276. D
efining PARAMS for NASTRAN 743

277. F
inite Element Model 745

278. E
xporting an Exodus II File 747
Element and Node ID Maps 747
Exporting a Parallel Mesh for pCAMAL 747
Converting an Exodus II file to ASCII 747
Controlling Exodus II Output Precision 748
Large Exodus Format 748

279. I
nstancing Parts with ABAQUS 749

280. E
xporting Fluent Grid Files 751

281. T
ransforming Mesh Coordinates 753

282. A
utomatic Detail Suppression 755
Example 755

283. A
utomatic Geometry Decomposition 757

284. C
ohesive Elements 759
Multiple Curves in FLATQUAD Blocks 759

285. D
eleting Mesh Elements 763

286. F
eatureSize 765

287. M
esh Cutting 767
Coordinate Plane 767
Planar Surface 767
Plane from 3 points 767
Extended Surface 767
Meshcut Options 768
Meshcutting Scope 768
Meshcutting Example 768

288. M
esh Grafting 775
Grafting Options 775
Grafting Scope 775

289. O
ptimize Jacobian 779

290. R
andomize 781

291. R
efine Mesh Boundary 783

292. S
uper Sizing Function 785

293. T
est Sizing Function 787

294. T
ransition 789

295. T
riangle Mesh Coarsening 791

296. W
hisker Weave 793
Whisker Weaving Basic Commands 794
Whisker Weaving Options 794

297. A
vailable Colors 795

298. E
lement Numbering 799
Node Numbering 799
Side Numbering 799
Triangular Shell Element Numbering 799
Node Ordering 800
Side Set Side Ordering 800

299. F
ullHex vs. NodeHex Representation 803

300. A
PREPRO Syntax 805
Immutable Variables 805

301. A
PREPRO Rules 807
1. Functions 807
2. Variables 807

3. Numbers 807
4. Strings 807
5. Operators 807
6. Delimiters 807
7. Expressions 807
8. Algebraic Expressions 807
9. String Expressions 808
10. Relational Expressions 808
11. Conditional Expressions 808
12. White Space 808
13. Switch Statements 808
14. Conditionals and Loops 809

302. A
PREPRO Operators 811
1. Arithmetic Operators 811
2. Assignment Operators 812
3. Relational Operators 812
4. Boolean Operators 813
5. String Operators 813

303. A
PREPRO Predefined Variables 815

304. A
PREPRO Units 817

305. A
PREPRO Functions 821
1. Mathematical Functions 821
2. CUBIT Functions 823
3.String Functions 827

306. A
PREPRO Additional Functionality 831
1. File Inclusion 831
2. Conditionals 831
3. Loops 831

307. A
PREPRO Journaling 833
APREPRO Comments 833
Significant Figures 833
Loops and Journaling 833
Multi-line Strings 833

308. C
ubitInterface 835
Class Member Functions 835
Class Variables 841
Member Function Documentation 841
init(argv) 841
destroy() 841
set_cubit_interrupt(interrupt) 842
set_playback_paused_on_error(pause) 842

Bool is_playback_paused_on_error() 842
Bool developer_commands_are_enabled() 842
str get_version() 842
str get_revision_date() 842
str get_build_number() 842
str get_acis_version() 843
int get_acis_version_as_int() 843
str get_exodus_version() 843
str get_graphics_version() 843
print_cmd_options() 843
Bool is_modified() 843
set_modified() 844
Bool is_undo_save_needed() 844
set_undo_saved() 844
Bool is_command_echoed() 844
Bool is_volume_meshable(volume_id) 844
journal_commands(state) 844
Bool is_command_journaled() 844
str get_current_journal_file() 845
cmd(input_string) 845
silent_cmd(input_string) 845
[int] parse_cubit_list(type, int_list, include_sheet_bodies) 845
print_raw_help(input_line, order_dependent, consecutive_dependent) 845
int get_error_count() 846
[str] get_mesh_error_solutions(error_code) 846
float get_view_distance() 846
[float] get_view_at() 846
[float] get_view_from() 846
reset_camera() 847
unselect_entity(entity_type, entity_id) 847
Bool is_perspective_on() 847
Bool is_occlusion_on() 847
Bool is_scale_visibility_on() 847
Bool is_select_partial_on() 847
int get_rendering_mode() 848
set_rendering_mode(mode) 848
clear_preview() 848
str get_pick_type() 848
float get_mesh_edge_length(edge_id) 848
float get_meshed_volume_or_area(geom_type, entity_ids) 848
int get_mesh_intervals(geom_type, entity_id) 849
float get_mesh_size(geom_type, entity_id) 849
float get_auto_size(volume_id_list, size) 849
float get_quality_value(mesh_type, mesh_id, metric_name) 850
str get_mesh_scheme(geom_type, entity_id) 850
str get_mesh_scheme_firmness(geom_type, entity_id) 850
str get_mesh_interval_firmness(geom_type, entity_id) 851
Bool is_meshed(geom_type, entity_id) 851
Bool is_merged(geom_type, entity_id) 851
str get_smooth_scheme(geom_type, entity_id) 851
int get_hex_count() 852
int get_pyramid_count() 852
int get_tet_count() 852
int get_quad_count() 852
int get_tri_count() 852
int get_edge_count() 853
int get_node_count() 853
int get_element_count() 853

int get_volume_element_count(volume_id) 853
int get_surface_element_count(surface_id) 853
Bool volume_contains_tets(volume_id) 853
[int] get_hex_sheet(node_id_1, node_id_2) 854
Bool is_visible(geom_type, entity_id) 854
ERROR: EOF in multi-line statement 854
Bool is_virtual(geom_type, entity_id) 854
ERROR: EOF in multi-line statement 854
Bool contains_virtual(geom_type, entity_id) 855
ERROR: EOF in multi-line statement 855
[int] get_source_surfaces(volume_id) 855
[int] get_target_surfaces(volume_id) 855
int get_common_curve_id(surface_1_id, surface_2_id) 855
int get_common_vertex_id(curve_1_id, curve_2_id) 856
str get_merge_setting(geom_type, entity_id) 856
str get_curve_type(curve_id) 856
str get_surface_type(surface_id) 856
[float] get_surface_normal(surface_id) 857
[float] get_surface_centroid(surface_id) 857
str get_surface_sense(surface_id) 857
[str] get_entity_modeler_engine(geom_type, entity_id) 857
[float] get_bounding_box(geom_type, entity_id) 858
[float] get_total_bounding_box(geom_type, entity_list) 858
float get_total_volume(volume_list) 858
str get_entity_name(geom_type, entity_id) 859
int get_entity_color_index(entity_type, entity_id) 859
Bool is_multi_volume(body_id) 859
Bool is_sheet_body(volume_id) 859
Bool is_interval_count_odd(surface_id) 860
Bool is_periodic(geom_type, entity_id) 860
Bool is_surface_planar(surface_id) 860
Bool get_undo_enabled() 861
int number_undo_commands() 861
[str] get_aprepro_vars() 861
str get_aprepro_value_as_string(var_name) 861
Bool get_node_constraint() 861
str get_vertex_type(surface_id, vertex_id) 861
[int] get_relatives(source_geom_type, source_id, target_geom_type) 861
[int] get_adjacent_surfaces(geom_type, entity_id) 862
[int] get_adjacent_volumes(geom_type, entity_id) 862
[int] get_entities(geom_type, include_sheet_bodies) 862
[int] get_list_of_free_ref_entities(geom_type) 863
int get_owning_body(geom_type, entity_id) 863
int get_owning_volume(geom_type, entity_id) 863
int get_owning_volume_by_name(entity_name) 864
float get_curve_length(curve_id) 864
float get_arc_length(curve_id) 864
float get_distance_from_curve_start(x, y, z, curve_id) 864
float get_curve_radius(curve_id) 865
[float] get_curve_center(curve_id) 865
float get_surface_area(surface_id) 865
float get_volume_area(volume_id) 865
float get_hydraulic_radius_surface_area(surface_id) 866
float get_hydraulic_radius_volume_area(volume_id) 866
[float] get_center_point(entity_type, entity_id) 866
int get_valence(vertex_id) 866
float get_distance_between(vertex_id_1, vertex_id_2) 866
print_surface_summary_stats() 867

print_volume_summary_stats() 867
int get_body_count() 867
int get_volume_count() 867
int get_surface_count() 867
int get_vertex_count() 867
int get_curve_count() 868
int get_curve_count(target_volume_ids) 868
Bool is_catia_engine_available() 868
[int] evaluate_exterior_angle(curve_list, test_angle) 868
[int] get_small_curves(target_volume_ids, mesh_size) 868
[int] get_smallest_curves(target_volume_ids, num_to_return) 869
[int] get_small_surfaces(target_volume_ids, mesh_size) 869
[int] get_narrow_surfaces(target_volume_ids, mesh_size) 869
[int] get_small_and_narrow_surfaces(target_ids, small_area, small_curve_size) 869
[int] get_surfs_with_narrow_regions(target_ids, narrow_size) 869
[int] get_small_volumes(target_volume_ids, mesh_size) 870
[int] get_blend_surfaces(target_volume_ids) 870
[int] get_small_loops(target_volume_ids, mesh_size) 870
[int] get_tangential_intersections(target_volume_ids, upper_bound, lower_bound) 870
[int] get_coincident_vertices(target_volume_ids, high_tolerance) 871
[[str]] get_solutions_for_near_coincident_vertices(vertex_id1, vertex_id2) 871
[[str]] get_solutions_for_imprint_merge(surface_id1, surface_id2) 871
[[str]] get_solutions_for_small_surfaces(surface_id, small_curve_size, mesh_size) 871
[[str]] get_solutions_for_small_curves(curve_id, small_curve_size, mesh_size) 872
[[str]] get_solutions_for_surfaces_with_narrow_regions(surface_id, small_curve_size, mesh_size) 872
[int] get_overlapping_volumes(target_volume_ids) 872
[[int]] get_mergeable_vertices(target_volume_ids) 872
[[str]] get_solutions_for_blends(surface_id) 873
[[int]] get_blend_chains(surface_id) 873
float get_merge_tolerance() 873
str get_exodus_entity_name(entity_type, entity_id) 873
str get_exodus_entity_description(entity_type, entity_id) 874
[float] get_all_exodus_times(filename) 874
int get_block_id(entity_type, entity_id) 874
[int] get_block_ids(mesh_geom_file_name) 874
[int] get_block_id_list() 875
[int] get_nodeset_id_list() 875
[int] get_sideset_id_list() 875
[int] get_bc_id_list(bc_type_in) 875
str get_bc_name(bc_type_in, bc_id) 875
[int] get_nodeset_id_list_for_bc(bc_type_in, bc_id) 876
[int] get_sideset_id_list_for_bc(bc_type_in, bc_id) 876
int get_next_sideset_id() 876
int get_next_nodeset_id() 876
int get_next_block_id() 876
[int] get_block_volumes(block_id) 876
[int] get_block_surfaces(block_id) 877
[int] get_block_curves(block_id) 877
[int] get_block_vertices(block_id) 877
[int] get_block_nodes(block_id) 877
[int] get_block_edges(block_id) 878
[int] get_block_tris(block_id) 878
[int] get_block_faces(block_id) 878
[int] get_block_pyramids(block_id) 878
[int] get_block_tets(block_id) 878
[int] get_block_hexes(block_id) 879
[int] get_volume_hexes(volume_id) 879
[int] get_volume_tets(volume_id) 879

[int] get_nodeset_volumes(nodeset_id) 879
[int] get_nodeset_surfaces(nodeset_id) 879
[int] get_nodeset_curves(nodeset_id) 880
[int] get_nodeset_vertices(nodeset_id) 880
[int] get_nodeset_nodes(nodeset_id) 880
[int] get_nodeset_nodes_inclusive(nodeset_id) 880
[int] get_sideset_curves(sideset_id) 881
[int] get_curve_edges(curve_id) 881
[int] get_sideset_surfaces(sideset_id) 881
[int] get_sideset_quads(sideset_id) 881
[int] get_surface_quads(surface_id) 881
[int] get_surface_tris(surface_id) 882
str get_entity_sense(source_type, source_id, sideset_id) 882
str get_wrt_entity(source_type, source_id, sideset_id) 882
Bool is_using_shells(sideset_id) 882
[str] get_geometric_owner(mesh_entity_type, mesh_entity_list) 883
[int] get_volume_nodes(vol_id) 883
[int] get_surface_nodes(surf_id) 883
[int] get_curve_nodes(curv_id) 883
int get_vertex_node(vert_id) 884
int get_id_from_name(name) 884
[int] get_group_groups(group_id) 884
[int] get_group_bodies(group_id) 884
[int] get_group_volumes(group_id) 884
[int] get_group_surfaces(group_id) 885
[int] get_group_curves(group_id) 885
[int] get_group_vertices(group_id) 885
[int] get_group_nodes(group_id) 885
[int] get_group_edges(group_id) 885
[int] get_group_quads(group_id) 885
[int] get_group_tris(group_id) 885
[int] get_group_tets(group_id) 886
[int] get_group_hexes(group_id) 886
int get_next_group_id() 886
delete_all_groups() 886
delete_group(group_id) 886
set_max_group_id(max_group_id) 886
int create_new_group() 886
remove_entity_from_group(group_id, entity_id, entity_type) 887
add_entity_to_group(group_id, entity_id, entity_type) 887
[int] get_mesh_group_parent_ids(element_type, element_id) 887
Bool is_mesh_element_in_group(element_type, element_id) 887
Bool is_part_of_list(target_id, id_list) 888
int get_last_id(entity_type) 888
str get_assembly_classification_level() 888
str get_assembly_classification_category() 888
str get_assembly_weapons_category() 889
str get_assembly_metadata(volume_id, data_type) 889
Bool is_assembly_metadata_attached(volume_id) 889
str get_assembly_name(assembly_id) 889
str get_assembly_path(assembly_id) 889
str get_assembly_description(assembly_id) 890
int get_assembly_instance(assembly_id) 890
str get_assembly_file_format(assembly_id) 890
str get_assembly_units(assembly_id) 890
str get_assembly_material_description(assembly_id) 890
str get_assembly_material_specification(assembly_id) 891
int get_exodus_id(entity_type, entity_id) 891

str get_geometry_owner(entity_type, entity_id) 891
[int] get_connectivity(entity_type, entity_id) 891
[int] get_expanded_connectivity(entity_type, entity_id) 892
[int] get_sub_elements(entity_type, entity_id, dimension) 892
[float] get_nodal_coordinates(node_id) 893
Bool get_node_position_fixed(node_id) 893
str get_sideset_element_type(sideset_id) 893
str get_block_element_type(block_id) 893
int get_exodus_element_count(entity_id, entity_type) 893
int get_block_attribute_count(block_id) 894
float get_block_attribute_value(block_id, index) 894
[str] get_valid_block_element_types(block_id) 894
int get_nodeset_node_count(nodeset_id) 894
int get_geometry_node_count(entity_type, entity_id) 895
str get_mesh_element_type(entity_type, entity_id) 895
Bool is_on_thin_shell(bc_type_in, entity_id) 895
Bool temperature_is_on_solid(bc_type_in, entity_id) 895
Bool convection_is_on_solid(entity_id) 895
Bool convection_is_on_shell_area(entity_id, shell_area) 896
float get_convection_coefficient(entity_id, cc_type) 896
float get_bc_temperature(bc_type, entity_id, temp_type) 896
Bool temperature_is_on_shell_area(bc_type, bc_area, entity_id) 896
Bool heatflux_is_on_shell_area(bc_area, entity_id) 896
float get_heatflux_on_area(bc_area, entity_id) 896
int get_cfd_type(entity_id) 897
float get_contact_pair_friction_value(entity_id) 897
float get_contact_pair_tolerance_value(entity_id) 897
Bool get_contact_pair_tied_state(entity_id) 897
Bool get_contact_pair_general_state(entity_id) 897
Bool get_contact_pair_exterior_state(entity_id) 897
int get_displacement_coord_system(entity_id) 898
str get_displacement_combine_type(entity_id) 898
float get_pressure_value(entity_id) 898
str get_pressure_function(entity_id) 898
float get_force_magnitude(entity_id) 898
float get_moment_magnitude(entity_id) 898
[float] get_force_direction_vector(entity_id) 899
[float] get_force_moment_vector(entity_id) 899
str get_constraint_type(constraint_id) 899
str get_constraint_reference_point(constraint_id) 899
str get_constraint_dependent_entity_point(constraint_id) 899
float get_material_property(mp, entity_id) 900
int get_media_property(entity_id) 900
[str] get_material_name_list() 900
[str] get_media_name_list() 900
set_label_type(entity_type, label_flag) 900
int get_label_type(entity_type) 900
Body body(id_in) 900
Volume volume(id_in) 900
Surface surface(id_in) 901
Curve curve(id_in) 901
Vertex vertex(id_in) 901
reset() 901
Body brick(width, depth, height) 901
Body sphere(radius, x_cut, y_cut, z_cut, inner_radius) 902
Body prism(height, sides, major, minor) 902
Body pyramid(height, sides, major, minor, top) 902
Body cylinder(hi, r1, r2, r3) 902

Body torus(r1, r2) 903
Vertex create_vertex(x, y, z) 903
Curve create_curve(v0, v1) 903
Body create_surface(curves) 903
[Body] sweep_curve(curves, along_curves, draft_angle, draft_type, rigid) 904
Body copy_body(init_body) 904
[Body] tweak_surface_offset(surfaces, distances) 904
[Body] tweak_surface_remove(surfaces, extend_ajoining, keep_old, preview) 904
[Body] tweak_curve_remove(curves, keep_old, preview) 905
[Body] tweak_curve_offset(curves, distances, keep_old, preview) 905
[Body] tweak_vertex_fillet(verts, r0, keep_old, preview) 905
[Body] subtract(tool_in, from_in, imprint_in, keep_old_in) 905
[Body] unite(body_in, keep_old_in) 906
move(entity, vector, preview) 906
scale(entity, factor, preview) 906
reflect(entity, axis, preview) 906
[int] get_volumes_for_node(node_name, node_instance) 906
int get_mesh_error_count() 907

309. **PyObservable** **909**
Inheritance 909
Class Member Functions 909
Member Function Documentation 909
notify_observers(event_type) 910

310. **PyObserver** **911**
Class Member Functions 911
Member Function Documentation 911
register_observable(observable) 911
unregister_observable(observable) 911
notify_observers(observable, event_type) 911

311. **Entity** **913**
Inheritance 913
Class Member Functions 913
Member Function Documentation 913
destroy_cubit_entity() 913
[float] bounding_box() 913
[float] center_point() 913
int id() 914
is_visible(visibility_flag) 914
int is_visible() 914
is_transparent(transparency_flag) 915
int is_transparent() 915

312. **GeomEntity** **917**
Inheritance 917
Class Member Functions 917
Member Function Documentation 917
mesh() 917
Bool is_meshed() 917
smooth() 918

remove_mesh() 918
str entity_name() 918
entity_name(name) 919
[str] entity_names() 919
int num_names() 919
remove_entity_name(name) 920
remove_entity_names() 920
int dimension() 920
[Body] bodies() 921
[Volume] volumes() 921
[Surface] surfaces() 921
[Curve] curves() 922
[Vertex] vertices() 922

313. **Body** **923**
Inheritance 923
Class Member Functions 923
Member Function Documentation 923
[float] get_mass_props() 923
int point_containment(loc_in) 923
float volume() 924
Bool is_sheet_body() 924

314. **Surface** **925**
Inheritance 925
Class Member Functions 925
Member Function Documentation 925
color(value) 925
int color() 925
[[Curve]] ordered_loops() 926
[float] normal_at(location) 926
[float] closest_point_trimmed(location) 927
[float] closest_point_trimmed(location) 927
int point_containment(point_in) 927
[float] principal_curvatures(point) 928
[float] position_from_u_v(u, v) 928
[float] u_v_from_position(location) 929
[float] get_param_range_U() 929
[float] get_param_range_V() 929
float area() 930
Bool is_planar() 930
Bool is_cylindrical() 930

315. **Curve** **931**
Inheritance 931
Class Member Functions 931
Member Function Documentation 931
color(value) 931
int color() 932
[float] tangent(point) 932
[float] curvature(point) 932
[float] closest_point(point) 933
[float] closest_point_trimmed(point) 933

float length() 934
[float] curve_center() 934
[float] position_from_fraction(fraction_along_curve) 934
float start_param() 935
float end_param() 935
float u_from_position(position) 935
[float] position_from_u(u_value) 936
float u_from_arc_length(root_param, arc_length) 936
float fraction_from_arc_length(root_vertex, length) 936
[float] point_from_arc_length(root_param, arc_length) 937
float length_from_u(parameter1, parameter2) 937
Bool is_periodic() 938

316. **Vertex** **939**
Inheritance 939
Class Member Functions 939
Member Function Documentation 939
color(value) 939
int color() 939
[float] coordinates() 940

317. **Volume** **941**
Inheritance 941
Class Member Functions 941
Member Function Documentation 941
float volume() 941
color(value) 941
int color() 942
[float] principal_axes() 942
[float] principal_moments() 942
[float] centroid() 943

318. **CubitFailureException** **945**
Class Member Functions 945
Member Function Documentation 945
str what() 945

319. **InvalidEntityException** **947**
Class Member Functions 947
Member Function Documentation 947
str what() 947

320. **InvalidInputException** **949**
Class Member Functions 949
Member Function Documentation 949
str what() 949

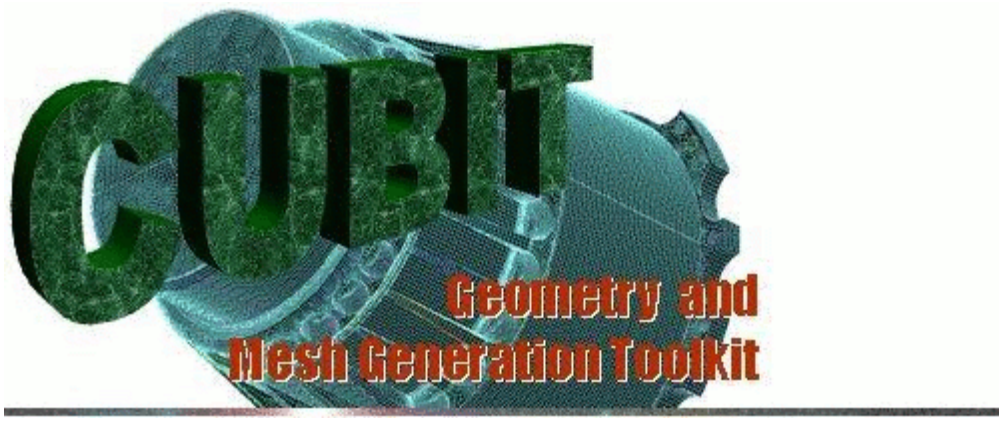
321. F
ASTQ 951

322. P
eriodic Space Filling Models (Tile) 953
Initial setup 953
Creating Nodesets 953
Smoothing 953
Example 954

323. R
eferences 955

324. C
redits 959

325. I
ndex 961



CUBIT 15.0 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.

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

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.

Hardware Requirements

Cubit is available on the following platforms:

- Linux glibc 2.5+ 32 and 64 bit (e.g. Redhat 5, Fedora 6, SUSE 10.2, Debian 5, Ubuntu 7.04)
- Windows Vista/7/8, 32 and 64 bit
- Mac 10.6+, 64 bit only

The Graphical User Interface version is available on all platforms.

For best results, local displays supporting OpenGL 1.5 is recommended.

Licensing and Distribution

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

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.

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.

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

The CUBIT team maintains a couple of mailing lists to help our users.

1) The cubit-announce mailing list is a very low-volume mailing list intended to provide news of new releases and other items of major importance. To subscribe to this list, send a message to: majordomo@sandia.gov
with the body of the message being:

subscribe cubit-announce

2) The cubit users mailing list is a medium-volume mailing list intended for our users to communicate with each other and ask help of the user community. It also contains the same announcements as the cubit-announce mailing list. To send questions or comments to this list, send email to:

cubit@sandia.gov

Users can subscribe to the cubit mailing list by emailing majordomo@scico.sandia.gov with a message body consisting of the single line:

subscribe cubit

An additional mailing list, cubit-help@sandia.gov, has been created for direct communication with the CUBIT developers. These messages won't reach other users. This list should be used for topics that are not of general interest to others, including some bugs.

Note: The recommended use of an electronic mailing list to report bugs and request enhancements is not intended to discourage face-to-face discussion with CUBIT developers, but rather to minimize response time. Users are encouraged to discuss bugs, enhancements or general meshing issues with the CUBIT production meshing and development teams.

Problem Reports and Enhancement Requests

CUBIT bugs, problem reports and enhancement requests should be 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.

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.

Execution Command Syntax

To run CUBIT from the command line:

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

claro [options and args] [journalFile(s)|python historyFile(s)]

Claro is the GUI version of CUBIT, which includes a python interpreter. To run a python script in CUBIT from the command line, run **claro** instead 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 or .cub from file $val)
-fastq <$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)
APREPRO variable pair (Quoted name value pair)

Each of these are 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.
<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.

On Unix or Windows, the following files are played back in order, if they exist, at startup:

<$CUBIT_DIR/.cubit.install
$HOMEDRIVE$HOMEPATH/.cubit
$HOME/.cubit
$(current working directory)/.cubit

Where **$(current working directory)** is determined by the program itself and words starting with '**$**' are environment variables.

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.

The **$CUBIT_DIR** file is installation specific. The **$HOME** file is user specific. The **$PWD** file is run-specific, read when starting up cubit from a particular meshing problem's subdirectory.

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.

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: %a - abbreviated weekday name %A - full weekday name %b - abbreviated month name %B - full month name %d - date of the month [01,31] %H - hour (24-hour clock) [00,23] %I - hour (12-hour clock) [01,12] %j - day of the year [1,366] %m - month number [1,12] %M - minute [00,59] %n - replaced with the next available number between 01 and 999. %p - "a.m." or "p.m." %S - seconds [00,61] %u - weekday [1,7], 1 is Monday %U - week of year [00,53] %w - weekday [0,6], 0 is Sunday %y - year without century [00,99] %Y - year with century (e.g. 1999) %% - a '%' character 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" 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.

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. 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.

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>** . 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> 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]

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.

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

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.

CUBIT Application Window

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

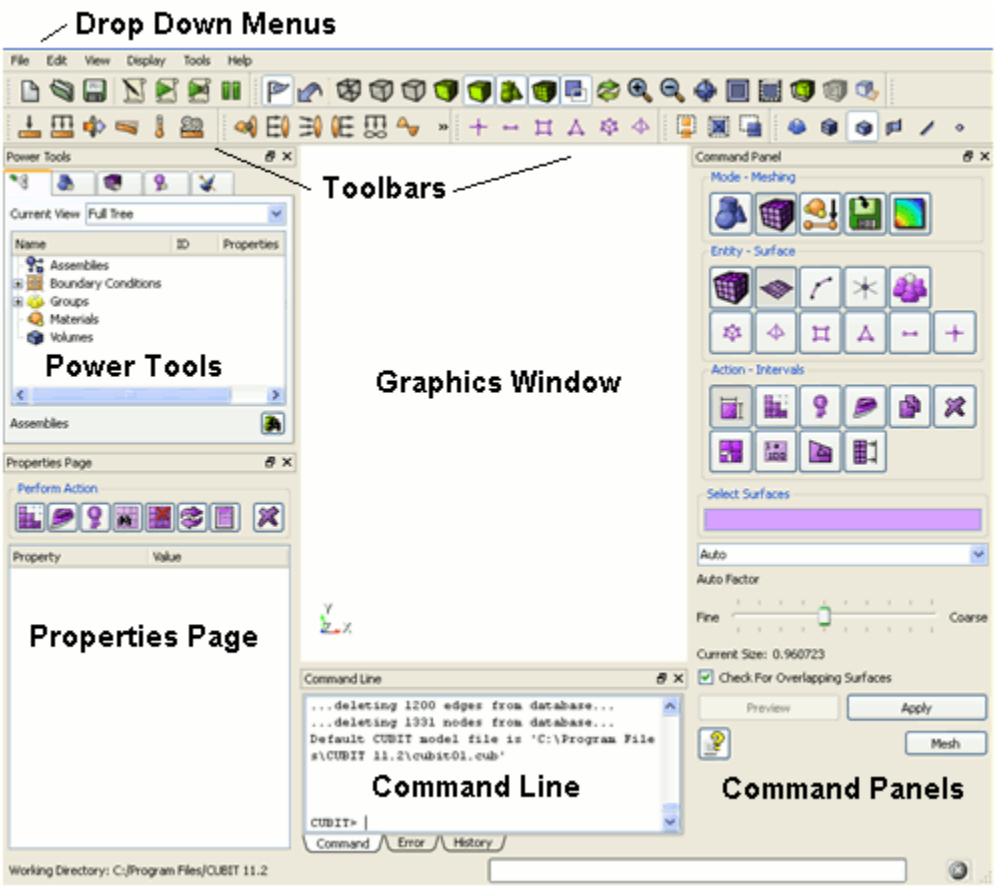


Figure 1. The CUBIT Application Window

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.

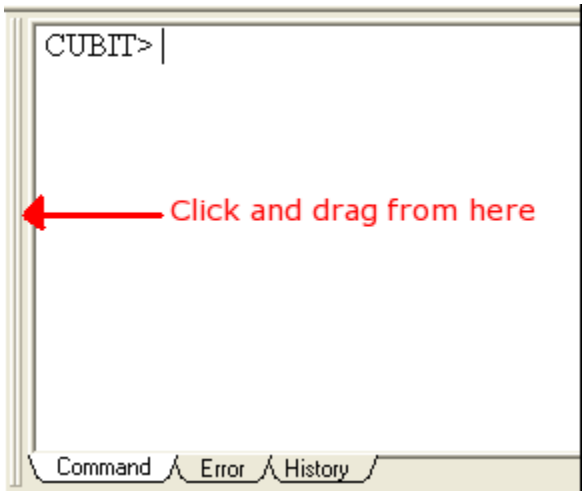


Figure 2. A docked window. Click and drag to float.

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.

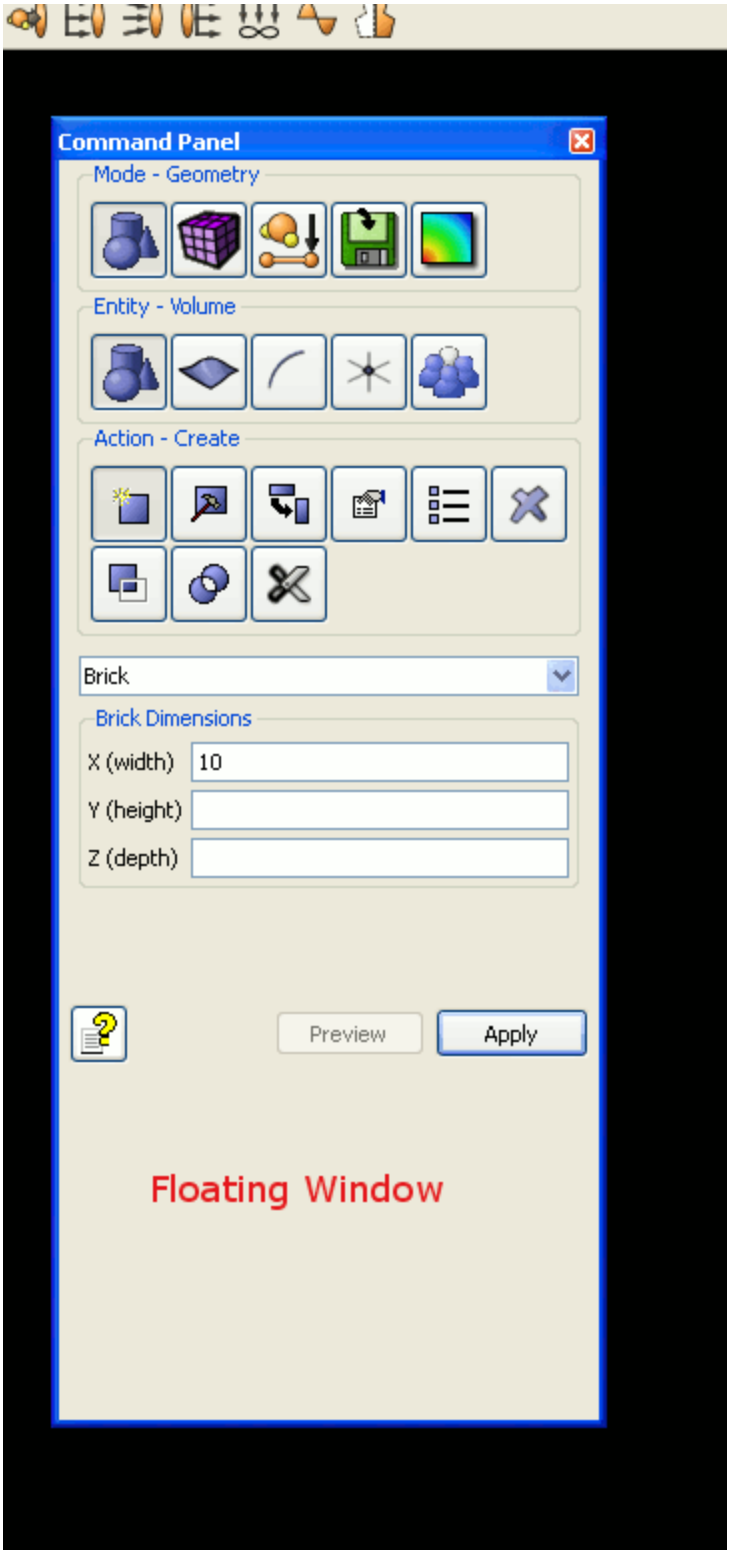


Figure 3. A Floating Window

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 Panel Functionality

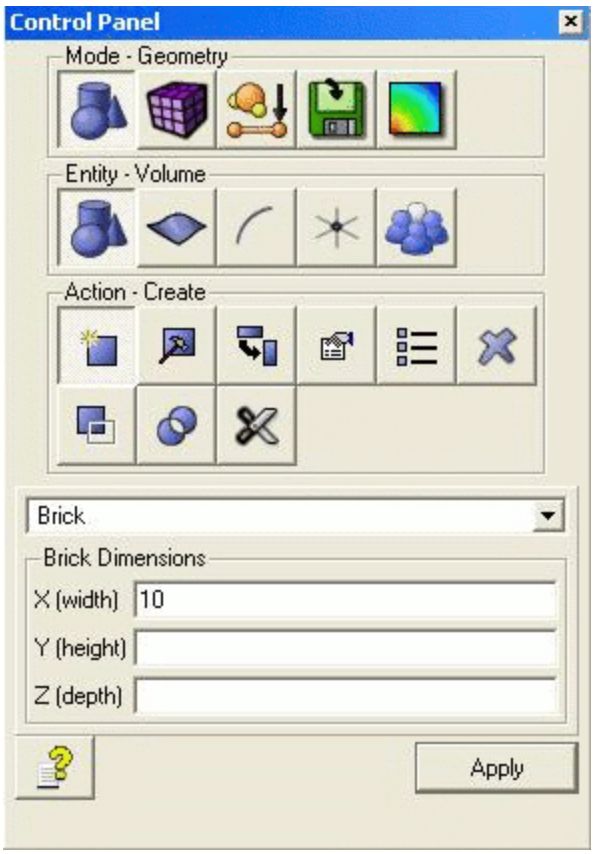
The Command Panel is arranged first by mode on the top row of buttons. Modes are arranged by task. All of the geometry related tasks, for instance, can be found under the Geometry mode. When a mode is selected, a second row of buttons becomes available. The second row of buttons shown depends on the selected mode. For example, if Geometry, Meshing, or Materials and BCs is selected, the second button row will show entity types. Entities are those specific to the mode.

- Geometry panel entity level buttons include Volumes, Surfaces, Curves, Vertices, and Groups.
- Meshing panel entity level buttons include Volumes, Surfaces, Curves, Vertices, Groups, Hexes, Tets, Quads, Tris, Bars, and Nodes.
- Materials and BCs entity level buttons include Exodus Nodesets, Exodus Sidesets, Exodus Blocks, Create Boundary Conditions, Modify Boundary Conditions, List Boundary Conditions, Draw Boundary Conditions, Make a Boundary Condition current, and Delete Boundary Conditions.

The second row of buttons for Analysis Setup and Post Processing are not arranged by entity. Rather, the buttons show specific capabilities.

The third row of buttons contains Actions, such as Create, Delete, Modify, and so forth. The following shows an example of Geometry/Volume actions.

Selecting an Action will display a command panel. The Geometry/Volume/Create command panel is shown below.



All command panels are constructed similarly. Each abstracts a set of Cubit commands. Options are selected using checkboxes, 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.

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).

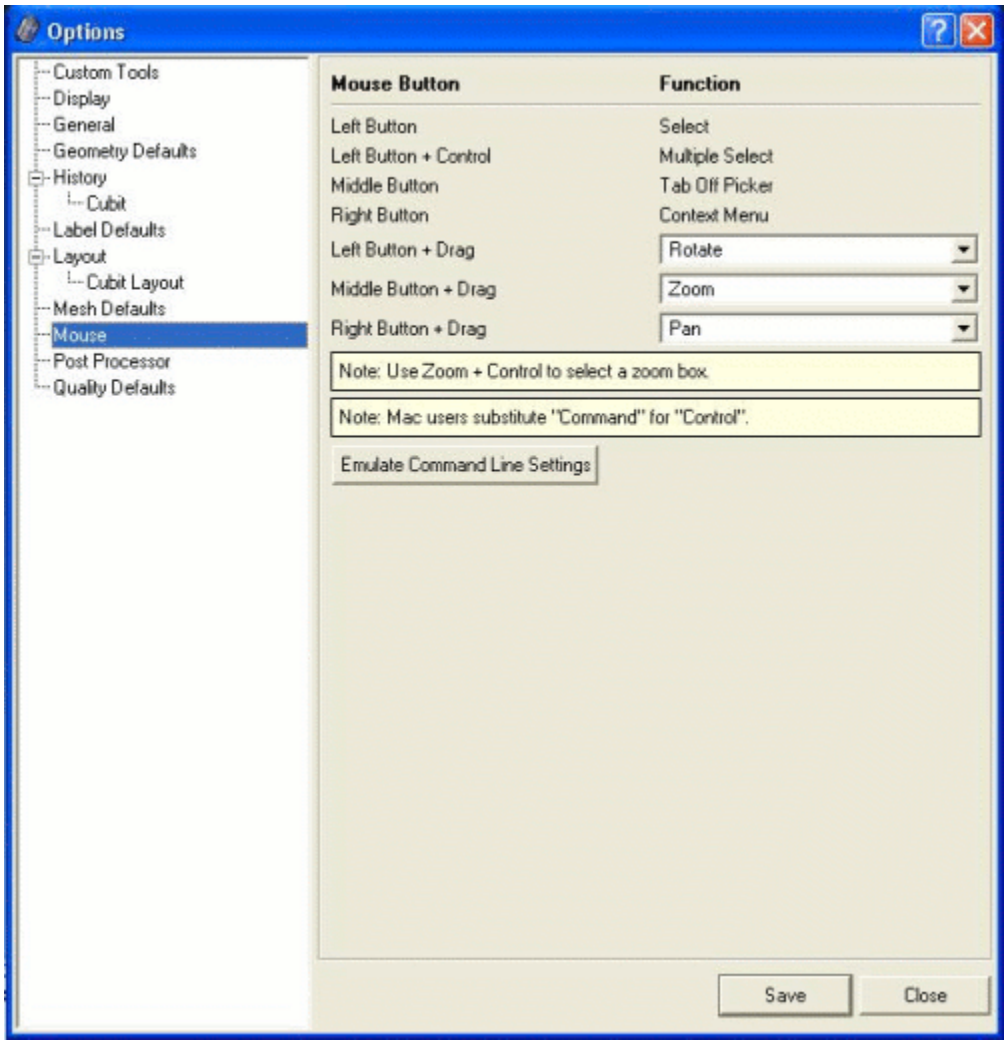


Figure 1. Mouse Settings Dialog

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

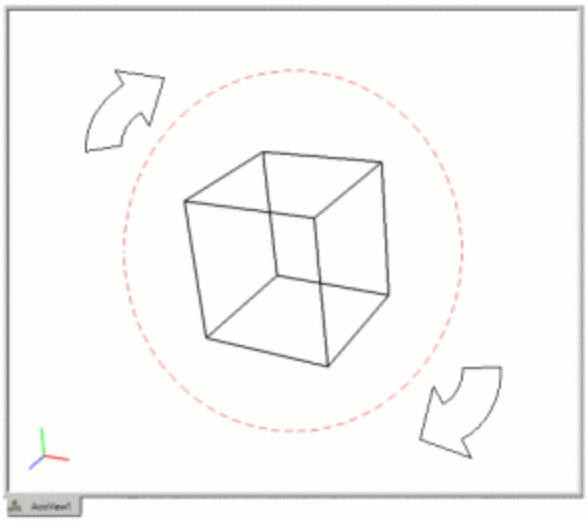


Figure 2. With the mouse pointer outside the circle the view is rotated about an axis normal to the screen

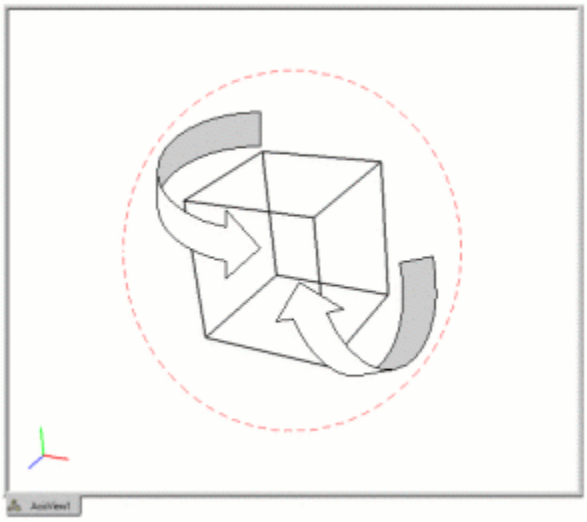


Figure 3. With the mouse pointer inside the circle the view is rotated about the current spin center

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.

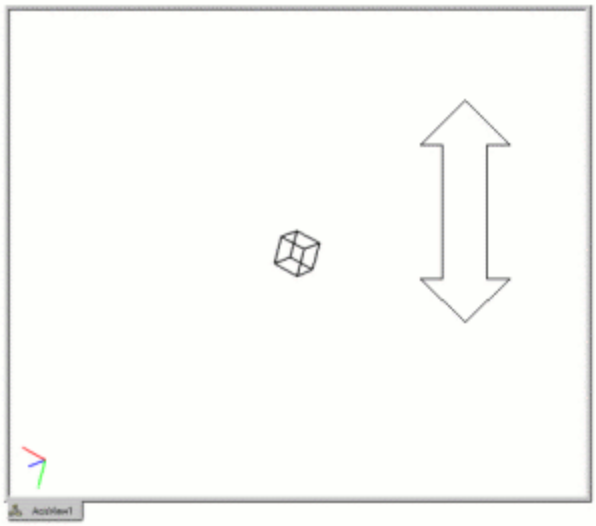


Figure 4. Move the mouse pointer vertically to zoom in and out

Panning

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

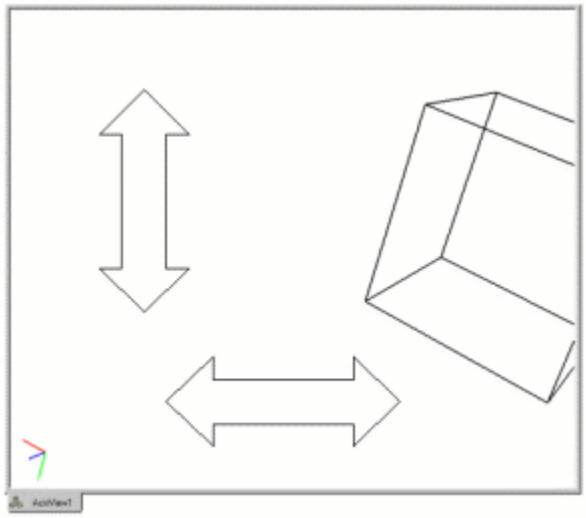


Figure 5. Move the mouse pointer horizontally or vertically to pan the view

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.

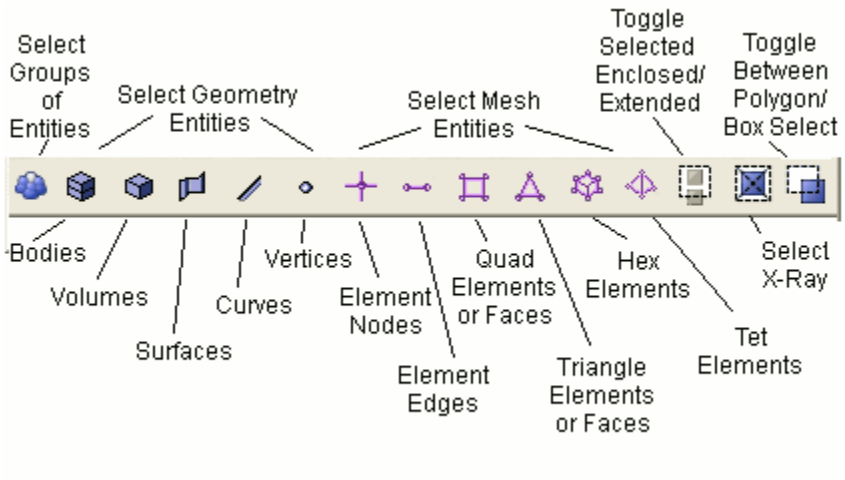


Figure 1. The Selection Toolbar for Geometry and Mesh Entities

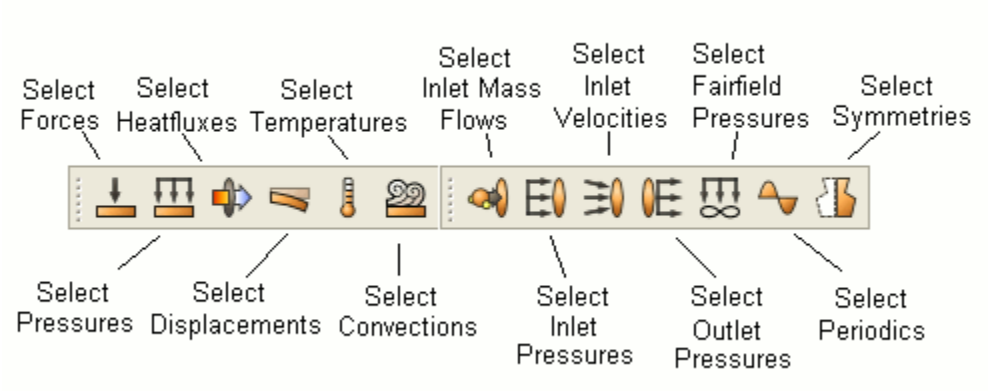


Figure 2. The Selection Toolbar for Boundary Conditions

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 and Box Select

The polygon/box selection feature allows you to select entities by drawing a box or polygon on the screen. To draw box on the screen press and hold the <CTRL> button***** while clicking and dragging the left mouse button. Release the left mouse to complete the box 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 or box method, press the *Toggle Between Polygon/Box 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 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.

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

- **Select Other**- Brings up a dialog with alternate entity selections
- **Zoom To** - Zoom to the selected entity
- **Rotate About** - Changes the center of rotation to the centroid of this entity
- **Draw** - Draw the selected entity
- **Isolate** - Turn all but the selected entities invisible
- **Add to BC/Group/Part** - Opens a dialog box where you can add the selected entity to an existing boundary condition, group, or part.
- **Remove from BC/Group/Part** - Opens a dialog box where you can remove the selected entity from an existing boundary condition, group, or part.
- **Add to Picked Group** - Add this entity to the picked group.
- **Remove from Picked Group** - Remove this entity from the picked group
- **Visibility Off** - Turn selected entities invisible
- **Mesh** - Mesh the selected entities
- **Measure** - Measures between two entities, or two vertices on a curve.
- **Delete Mesh** - Delete the mesh on selected entities (but not interval or scheme information)
- **Reset Entity** - Reset selected entities by deleting mesh and interval information
- **List Info** - Show the menu of additional list commands
- **Delete** - Delete selected entities

Without Entity Selected

- **Reset Zoom** - Reset zoom to original configuration
- **Refresh**- Refresh the graphics display
- **All Visible** - Make all entities visible
- **Display Options** - Opens Options Menu to display options

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.

Geometry Tree

The geometry 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 Assembly Data, active Groups, 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.

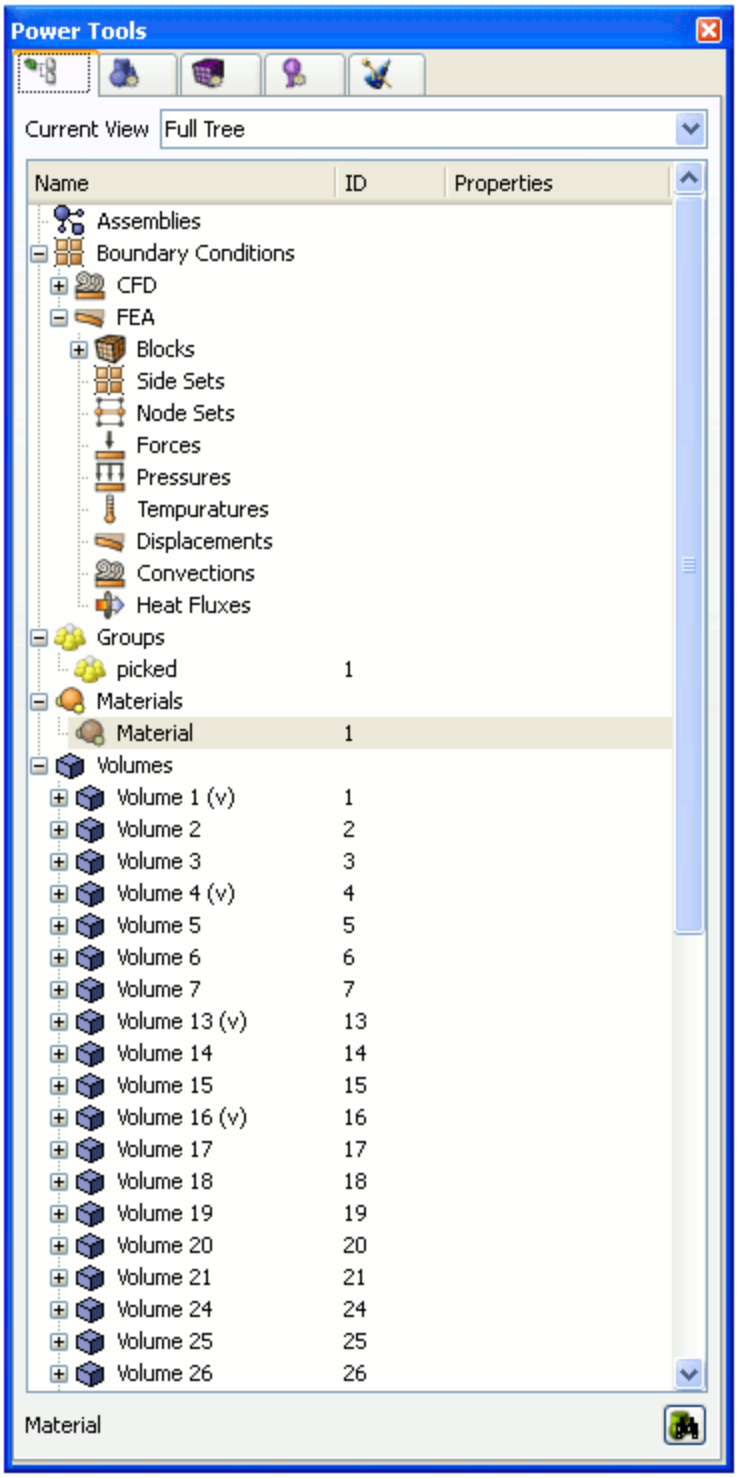


Figure 1. Geometry Tree Window

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. Assembly data may also be organized in the geometry tree window through drag and drop.

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.

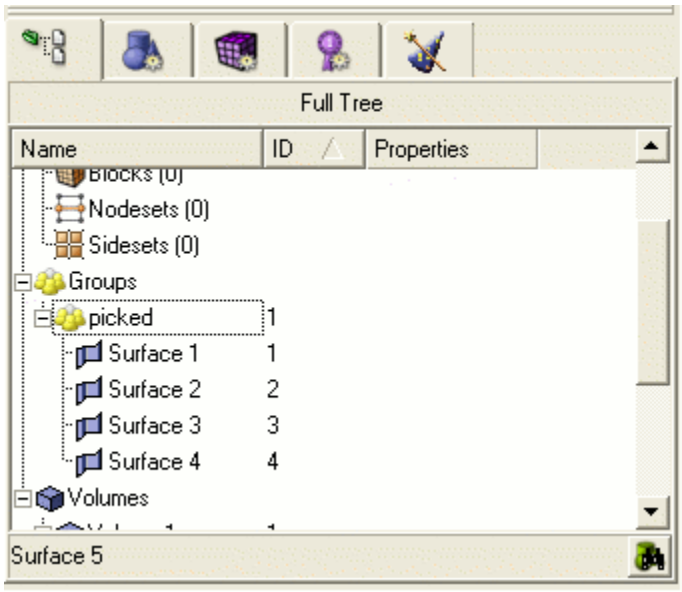


Figure 2. Picked Group

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.
- **Create New Assembly/Sub-assembly/Part** - You must specify the absolute path to create a new assembly, sub-assembly or part (e.g. /a1/p1). It may also be necessary to refresh the full tree before viewing changes.
- **Add Selected to Part**- Add the selected volume in the graphics window to the selected part on the geometry tree.
- **Remove from Metadata** - Deletes the selected part or assembly metadata information. An assembly must be empty to remove it
- **View Metadata** - List metadata in the command line workspace
- **Rename Metadata** - Allows you to rename a part or assembly
- **Clean Metadata** - Removes all parts and assemblies that are not associated with any geometric entities.
- **List Volumes Without Parts** - Lists all volumes that are not associated with a part in the output window
- **Show Part Name/Description** -Toggles the display of the part name/description in the tree.
- **Goto Part** - Finds the associated metadata part when a volume is 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.

Geometry Power Tools

The geometry power tools are located on the Tree View window under the blue geometry tab. In many cases, a model will fail to mesh because of problems with the geometry. Since the range of geometry problems is so wide, and because these problems can be hard to diagnose, the Geometry Power Tool has several built-in tools designed to analyze and repair these problems. The Geometry Repair Tool analyzes geometry for small angles, overlap, small features, bad geometry definition, blend surfaces, close loops, or mergeable entities that may affect meshing capability. It also contains a powerful toolkit of geometry modification methods to fix these problems. All of the common geometry clean-up tools are now in one place on the GUI menu. In addition, there is a window that lists results from geometry analysis in a tree format, making it easier to find, diagnose, and solve geometry problems. And Cubit will save your settings, so you can run the same diagnostic tests each time you use the geometry power tools.

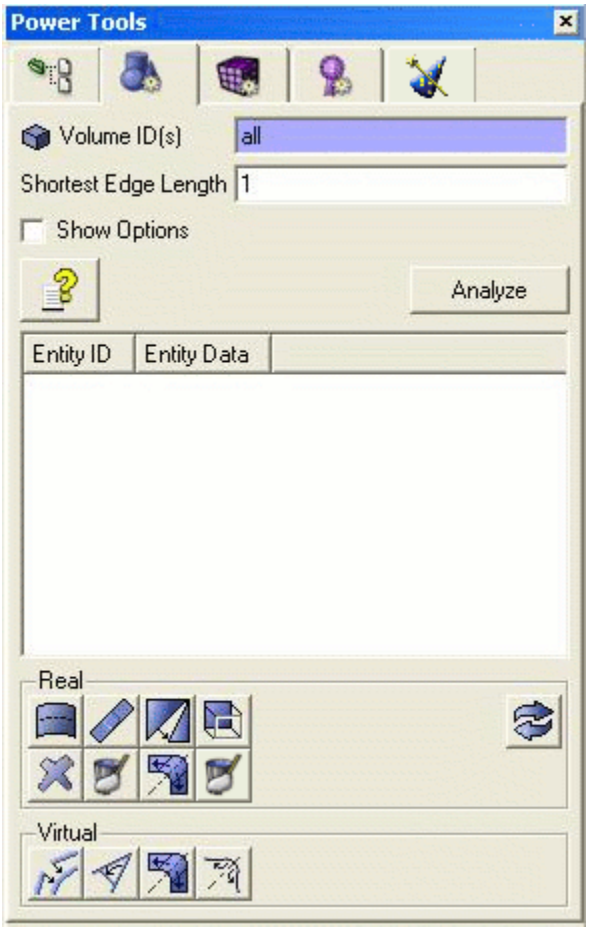


Figure 1. Geometry Power Tools

Geometry Analysis Tools

The geometry power tools contain an array of tests that can be run on geometry to diagnose potential problems for mesh generation. To display a list of tests, click on the **Show Options** check box. By default all tests are selected and run on geometry. Some tests may not apply to specific geometry, or may only need to be run once per geometry (i.e. bad geometry definition test). Clicking on the box by each test will deselect it.

The geometry analysis inputs and tests are summarized below:

Shortest Edge Length -The shortest edge length is a value that is input by the user. It determines the minimum allowable threshold for small features. It is used as an input to test for small curves, small surfaces, small volumes and close loops. The default value for this is 1. This value should be changed relative to the size of the model. In a very broad sense, it represents a desired mesh edge length. Curves and surfaces which are smaller than this size, and which may be troublesome to mesh with the desired granularity, will be flagged and they can be removed or modified.

Bad Angle Upper/Lower Bounds - The bad angle upper/lower bounds are tolerances set by the user to determine the definition of small or large angles. The default values are set at 350 degrees for the large angle and 10 degrees for the small angle. These values are used to test for angles between curves, surfaces, and at tangential intersections.

Bad Angle Check - The bad angle check will test for small angles between curves, surfaces, and at tangential intersections. The test will only look for curves or surfaces that are adjacent.

> **Tangential Intersection** - 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 as shown in the following example. 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.

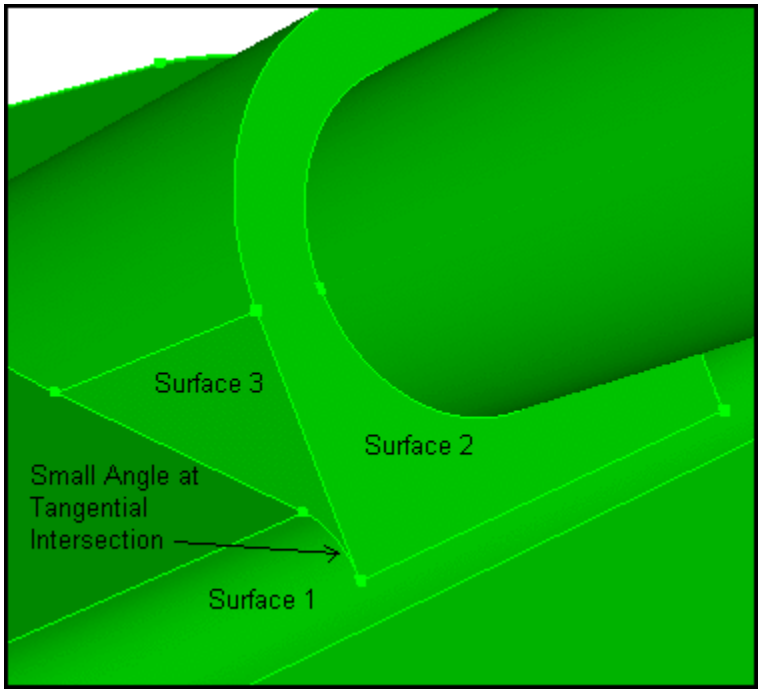


Figure 2. Tangential Intersection

Mergeable Entities Check - As it suggests, this test is looking for entities that overlap and that can be merged. Pressing the "Merge all" button on the Power Tools will automatically merge all entities flagged by the merge test.

Overlap Check - The overlap tests look for geometry that are either overlapping or coincident (exactly on top of each other). Keep in mind that some of these problems may disappear with imprinting and merging.

Small Features Check - Small features may be necessary and desirable in a model, but many times they are the result of poor geometry translation or import, 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 short edge length parameter. Small curves, including zero-length curves such as hardpoints, are compared directly against the defined parameter, and flagged if they less than or equal to the given parameter. Small surfaces and volumes, on the other hand, are compared against their hydraulic radius. For surfaces the hydraulic radius is **4*surface_area/perimeter**. For volumes the hydraulic radius is **6*volume/surface_area**.

Bad Geometry Definition Check - 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. If the third party library does not provide validation capabilities, this check will not return anything. Note: ACIS is a trademark of Spatial.

Blend Surface Check - A blend surface is a transition surface between two orthogonal planes, such as a fillet. The blend surface check identifies the surfaces which meet this criterion. Many times these surfaces are candidates for the split surface command or the remove surface command. The split surface command allows you to split these blend surfaces into two surfaces, making it easier to mesh the volume. The remove surface command removes the surface and extends the adjoining surfaces until they intersect.

Close Loops Check - Close loops (pronounced KLOS, not KLOZ) are two loops on a single surface for which the shortest distance between loops is less than a user specified tolerance. The tolerance for close loops is the square of the shortest edge length parameter. Close loops are common around holes and fillets, and are usually found where one loop is entirely within the other loop. These surfaces are often candidates for removal, or tweaking.

Geometry Repair Tools

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.

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.

Right Click Menu

The following right click menu is available from the geometry power tools. Specific options depend on the type of entity selected.

- **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.
- **Highlight** - Highlights selected entities.
- **Draw with Neighbors** - Displays only selected entities with all attached neighbors
- **Clear Highlights** - Clears all highlighted entities and reset graphics
- **Reset Graphics** - Reset graphics window
- **Tweak** - Opens the tweak menu in the main control panel
- **Remove** - Opens the remove menu in the main control panel
- **Remove Slivers** - Opens the remove sliver menu in the main control panel
- **Remove all** - Available when the clicking on an item in the "small surfaces" list. Opens the remove menu in the main control panel with all surfaces in the category as inputs. The individual option will be selected on the panel by default.
- **Split** - Opens the split surface or split curve menu in the main control panel, depending on the type of entity selected.
- **Auto Clean** - Opens the auto clean menu in the main control panel.
- **Regularize** - Issues the regularize command on selected entity.
- **Merge Selected** - Merge selected entity from mergeable entities list
- **Merge All** - Merge all entities listed in the mergeable entities list
- **(Virtual) Composite** - Opens the composite menu in the main control panel
- **(Virtual) Collapse** - Opens the collapse angle menu the main control panel
- **Collapse Surface (Virtual)** - Opens the collapse surface menu on the main control panel

The following right click options are available when category headings are selected.

- **Analyze Geometry** - Similar to pushing the Analyze button.
- **Highlight All** - Highlight all members of this category.
- **Draw All** - Display only members of this category.
- **Locate All** - Label all members of this category.

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 Tools

Entity Specification- The meshing power tool works for volumes or surfaces.

Options Button - Opens the **Tools>Options** dialog to change the visualization colors of surface schemes for the meshing tool

Analyze Button - The Analyze button issues the autoscheme command for all selected volumes and surfaces.

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.

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.

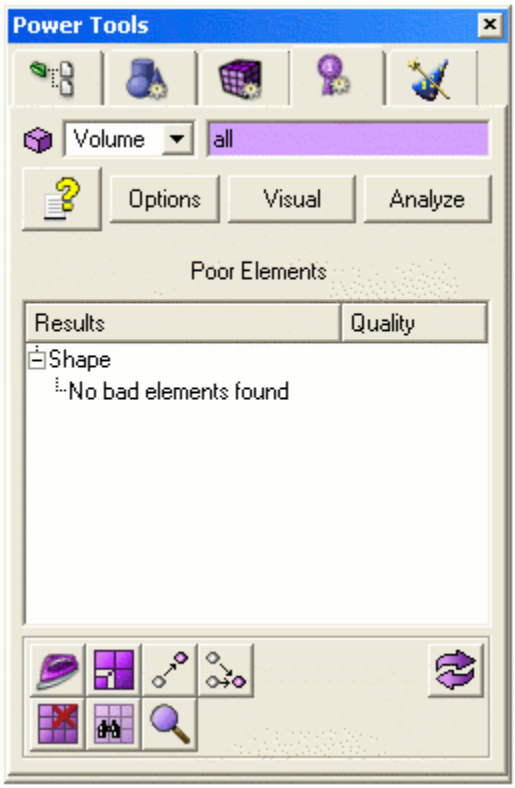


Figure 1. Mesh Quality Tools

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

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.

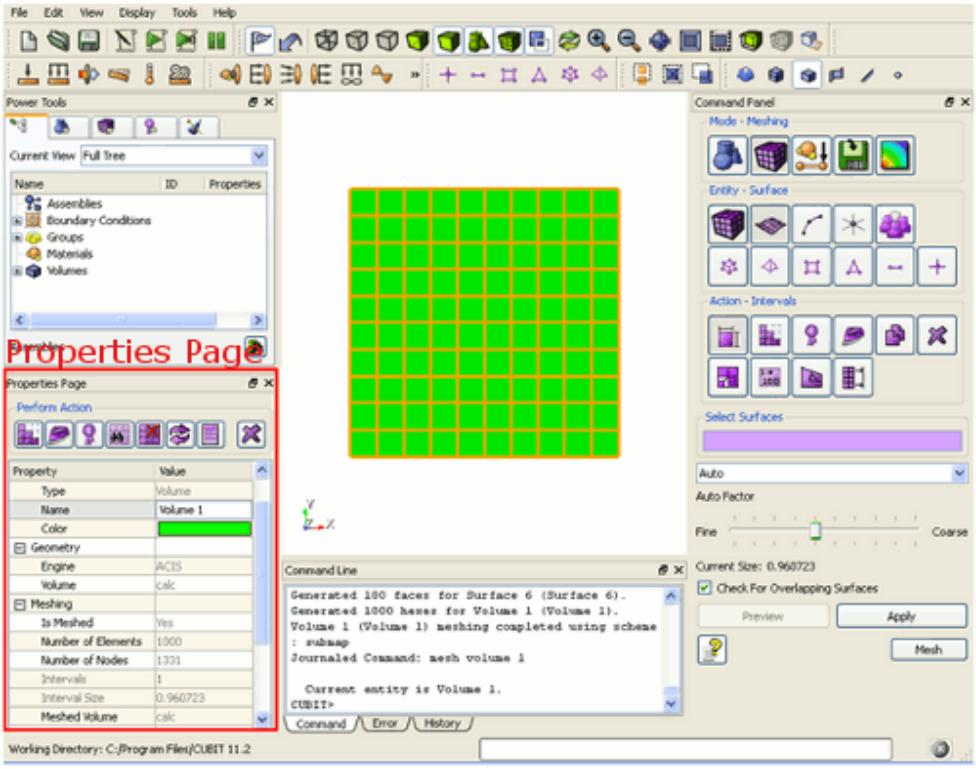


Figure 1. Property Editor Window

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
- **Intervals** - Number of mesh intervals on element. This can be edited from this window. The number must be an integer
- **Interval Size** - 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.

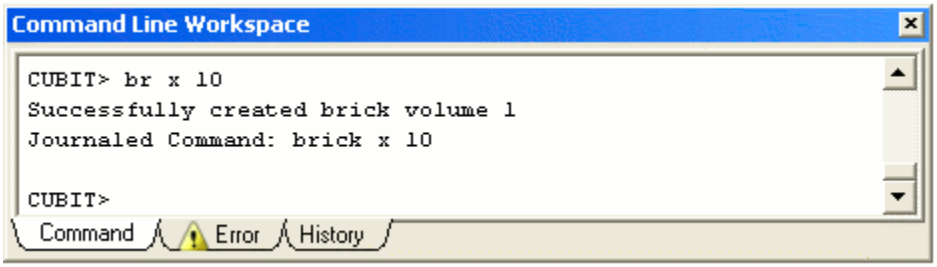
- **Description** - A user-defined description that is included in the metadata for that object. This value can be edited from the property editor.
- **Color** - Opens a dialog box with available colors. A color name can also be input directly into the text field. See Appendix for a list of available 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.

Metadata 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. This can be edited from the property editor
- **Material Description** - The name or description of the material of which this part is composed. Applies only to parts. This can be edited from the property window.
- **Material Specification** - The formal specification number of the material of which this part is composed. This can be edited from the property window.
- **File Format** - The name of the file system containing the original version of this entity. This can be edited from the property editor
- **Units** - The unit system of this part or assembly. This can be edited from the property editor

The part name, description and material description are available when the associated volume is selected, and not just when the part is selected.

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 four tabs:

1. Command
2. Error
3. History
4. Script

The Script window is hidden by default. To turn it on open the Tools-Options dialog and check the "Show Script Tab under Layout/Cubit Layout.

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>** 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>** prompt. If you do not remember the specific 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.

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.

Script Window

CUBIT boasts a robust Python interpreter built right into the graphical user interface. To create a Python script using the Script tab, start typing at the "%>" prompt. At the end of each line, hit **Enter** to move to the next line . To execute the script, press **Enter** at a blank line. Scripts may also be written in the Journal File Editor.

The Claro Python interpreter works as though you were entering lines from the Python command prompt. This means that a blank line is interpreted as the end of a block. If you want to add whitespace for clarity you have to add a # mark for a comment on any white line that is in a loop or a class.

One possible solution to this problem is to create two Python files. The first file can contain the complex set of Python instructions(program.py) including blank lines. The second file will read and execute the first file. An example syntax for the second file is given below.

```
f = file("program.py")
commandText = f.read()
exec(commandText)
```

You can then execute the second program within Cubit.

The interface between cubit and python is the "cubit" object. This object has a method called **cmd** which takes as an argument a command string. Thus, the following command in the script window:

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

will create a cube with sides 10 units long. 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.

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.

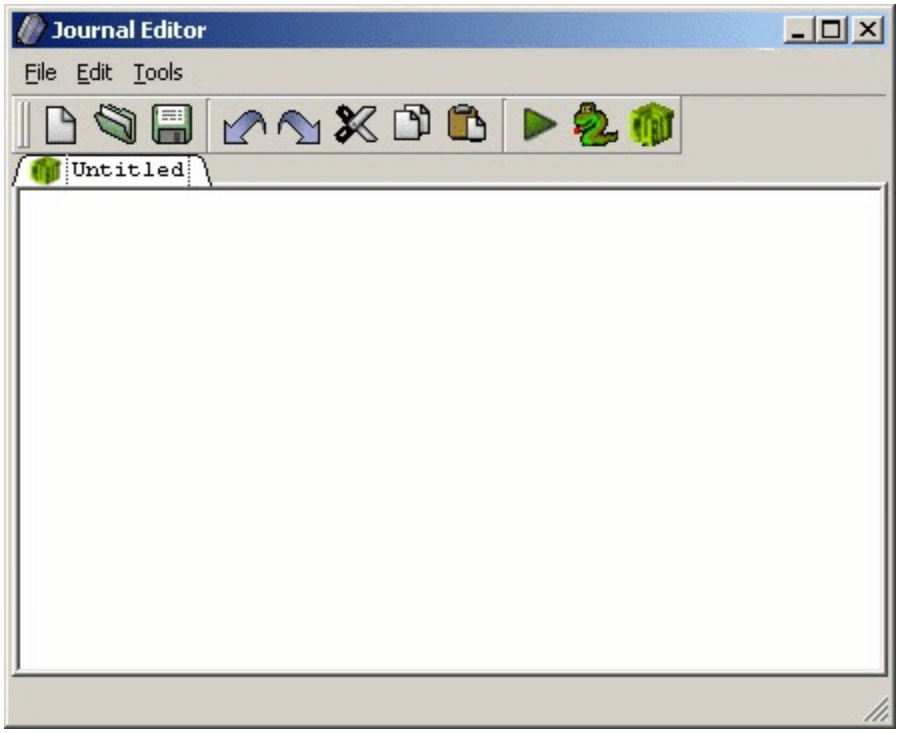


Figure 1. The Journal File Editor

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. You can change the default in the options dialog. On the "General" options page, under the Journal Editor heading, you can select the default syntax. You can change the new journal file's syntax using the translation buttons as well. When you have the correct syntax selected, 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.

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 item contains a list of available imports. Select the tab you want to import from. Only the current commands will be imported from the command line. 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 Python or Cubit command scripts. It can also translate between the two forms. Translating from Python to Cubit commands can cause commands to be lost. The Journal File Editor will warn you when doing so.

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.
- **Translate to Cubit** - Translates the current Python script in the journal file to Cubit commands.

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.

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.

File

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

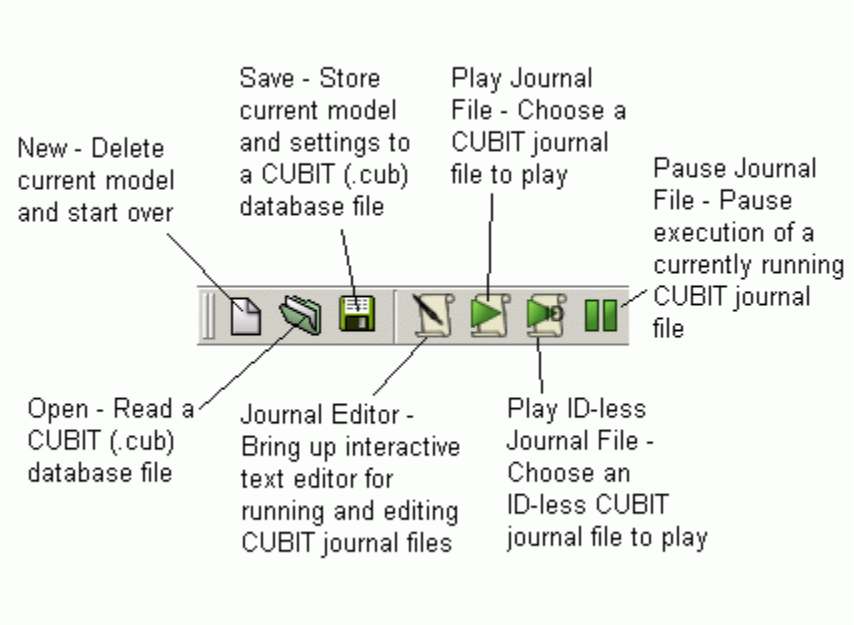


Figure 1. File Toolbar

Display

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

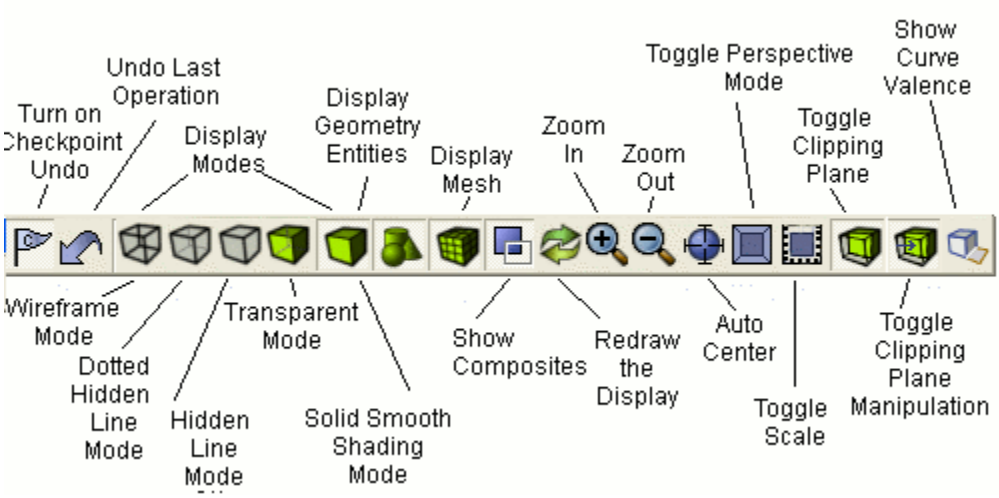
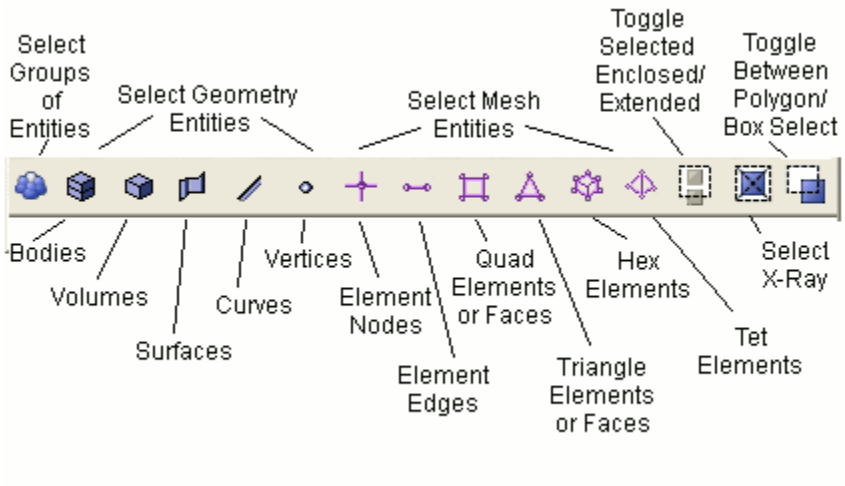


Figure 2. Display Toolbar

Select

Controls the Entity Selection Mode for picking or selecting entities. Also controls options for box/polygon selection.



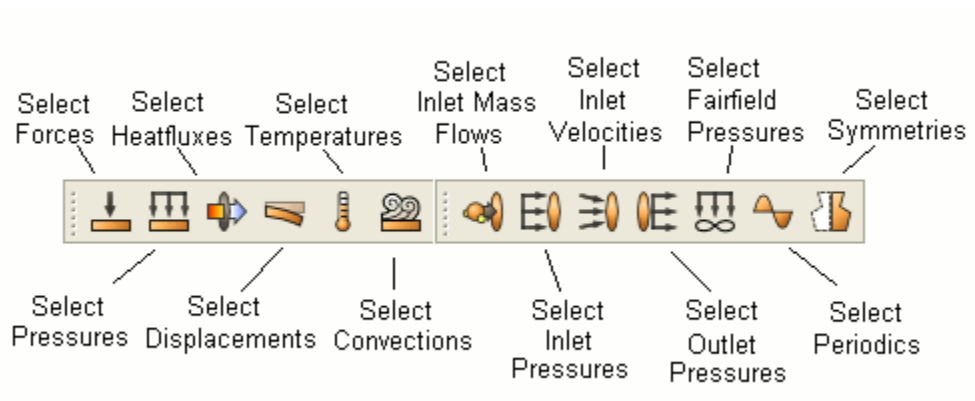


Figure 3. Select Toolbars

Options Menu

To change program preferences in the Graphical User Interface select: **Tools > Options** . The options menu includes:

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

Note: Mac users reach this dialog box by selecting the **Cubit > Preferences** menu.

Custom Tools

This menu controls the creation of Custom Toolbar buttons.

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.
- **Comment Line Filtering** - Whether to count comments in 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.
- **Journal File Directory** - Where the journal file will be saved. Default is the starting directory.
- **Journal File Name** - The name of the journal file. A name will be given by default if one is not specified. The default name for the GUI version of cubit is historyxx.jou with xx as the highest used number between 01 and 999 incremented by 1.

Cubit History Preferences

- **Use Cubit Journaling** - When this option is checked, Cubit journaling will be used. By default it is checked.
- **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

Creating Custom Toolbar Buttons

If you have a string of commands that you use frequently, it can be beneficial to make a custom toolbar button. To create a custom toolbar button open the **Tools->Options** menu. You can create up to 10 custom buttons. See Figure 1 for an example toolbar button.

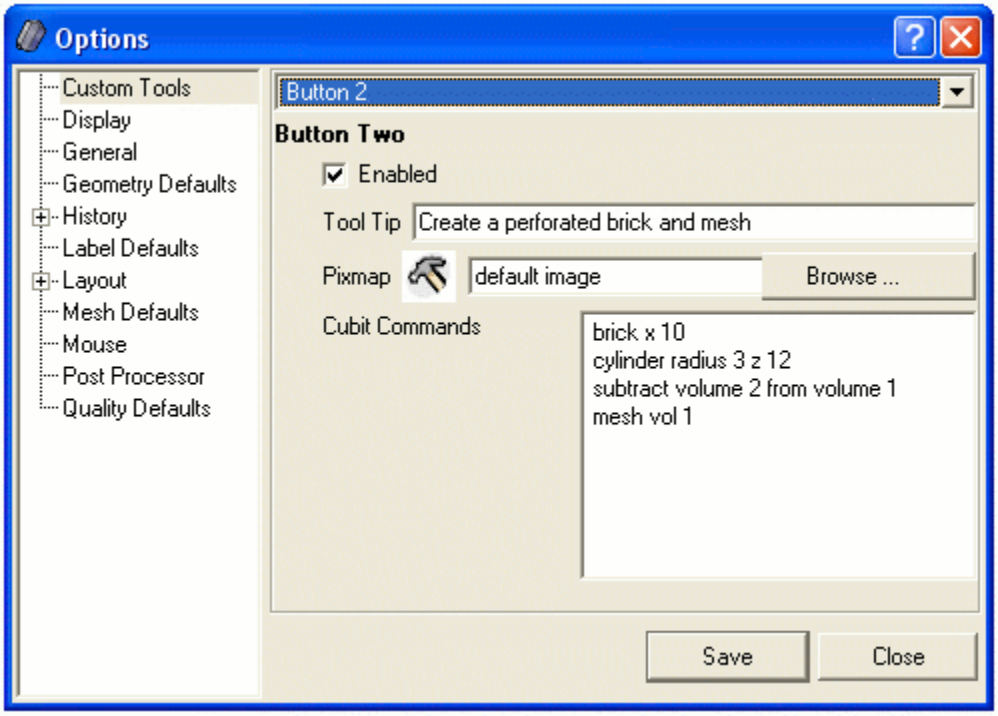


Figure 1. Making a custom toolbar button to create and mesh a perforated brick

The button can have Python or Cubit commands. These commands will be executed in consecutive order when the button is pushed. You must click the **Enabled** check box to activate your custom button.

You can assign a pixmap to your custom buttons or use the default. You can also assign a tool tip.

The buttons are persistent from each run of cubit. To remove a button, uncheck the **Enabled** button.

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

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.

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.

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 " or "history", depending on the version of 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 to be used depends on whether it is creating a historyXX.jou file, or a cubitXX.jou file. For historyXX.jou files, it will look for the highest used number in the current directory and increment it by one. For example, if there are already journal files with names history01.jou, history02.jou, and history04.jou, Cubit will use history05.jou as the current journal file. For 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.

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.

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:

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:

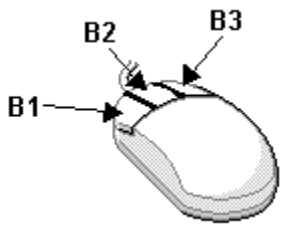


Figure 1. Default Mouse Function Mappings for the Command Line

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.

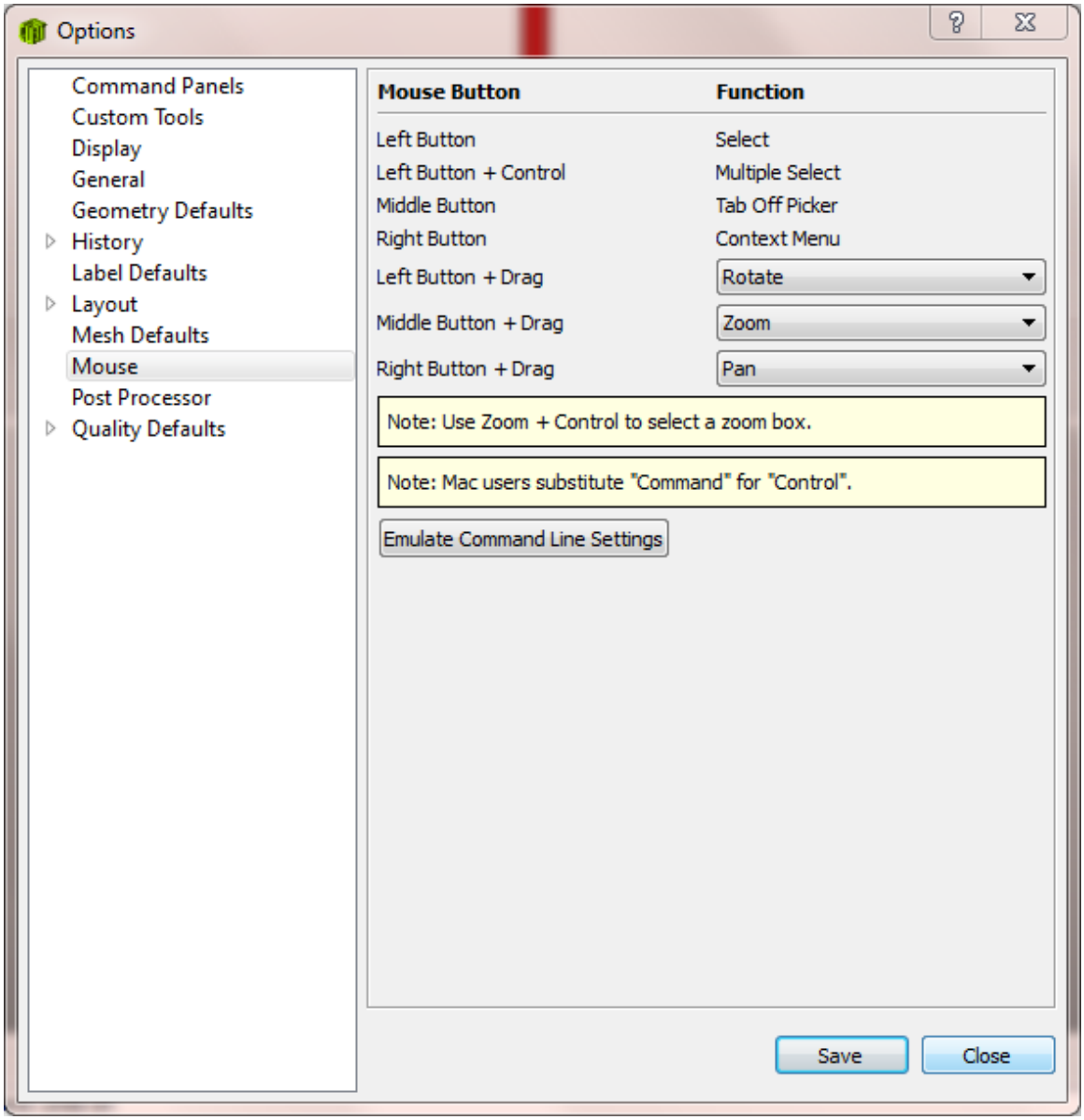
First, you can use the command

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.

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**.**

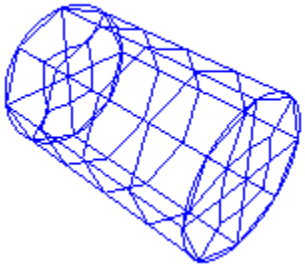
Graphics Modes

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:

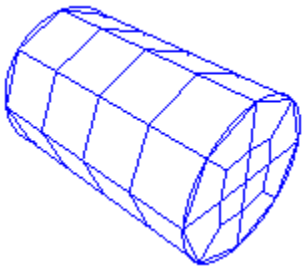
Graphics Mode {Wireframe | Hiddenline | Smoothshade | Transparent } [Geometry | Mesh]

The GUI tool bar buttons for manipulating the graphics modes are as follows:

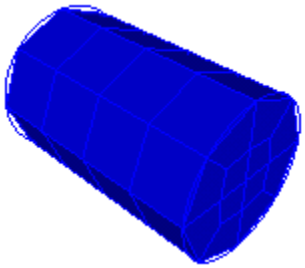
Examples and a brief description of each mode are shown below



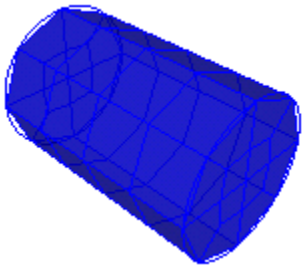
WireFrame - Surfaces are invisible. (This mode can also be accessed by typing **'wireframe'** at the command prompt.)



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.

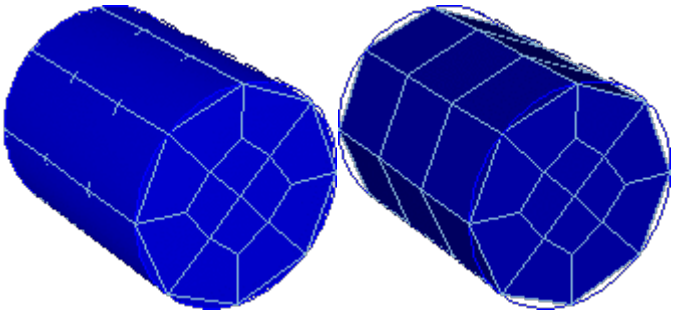


Figure 1. A meshed cylinder shown with graphics facets off (left) and graphics facets on (right); note how geometry facets on the curved surface obscure mesh edges when facets are off.

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}

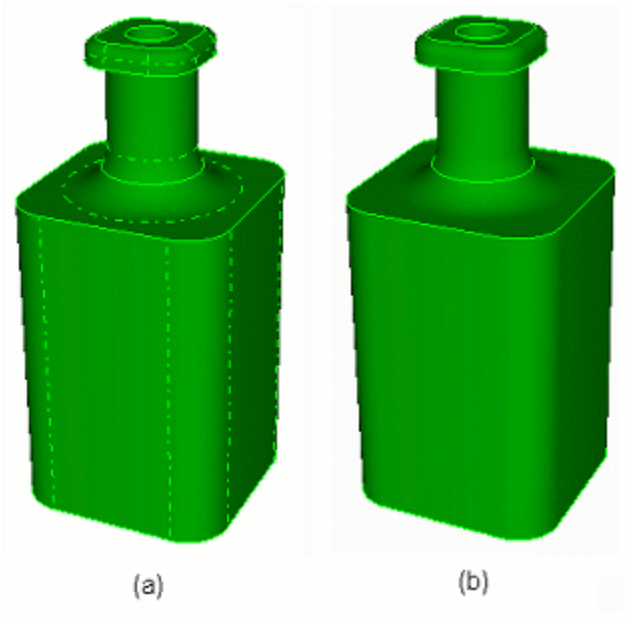


Figure 2. A part shown with (a) composite surfaces displayed (b) composite surfaces not displayed

The GUI tool bar button for toggling the display of graphics composites is as follows:

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>

Locate <location options>

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

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:

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.

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

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.

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} [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

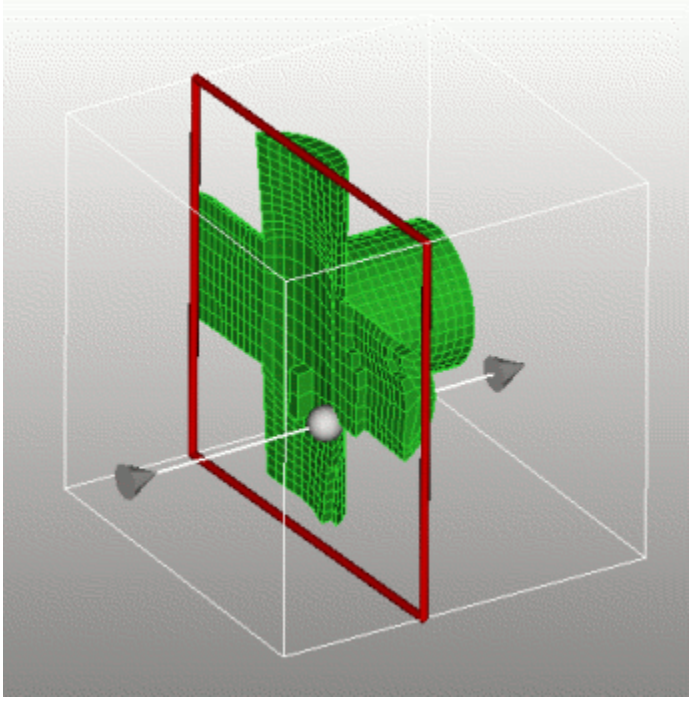


Figure 1. Graphics Clipping Plane

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

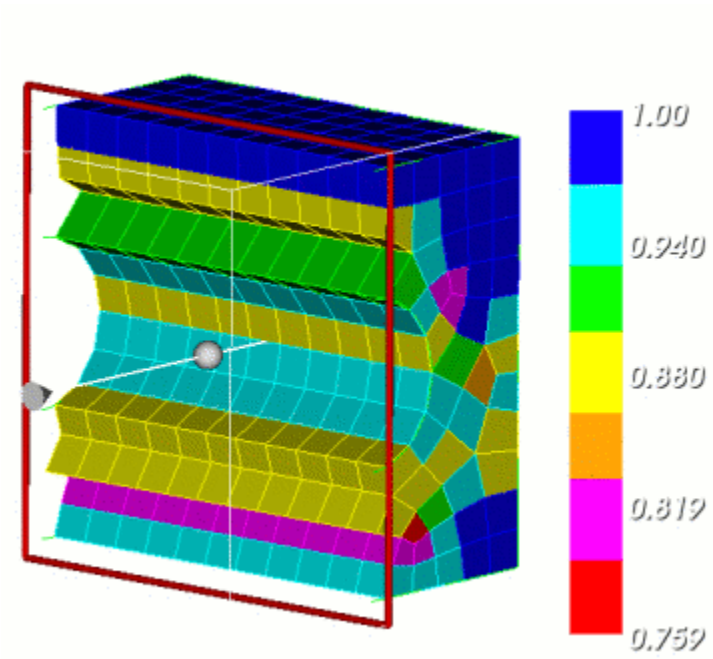


Figure 2. Viewing mesh quality of interior elements

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.

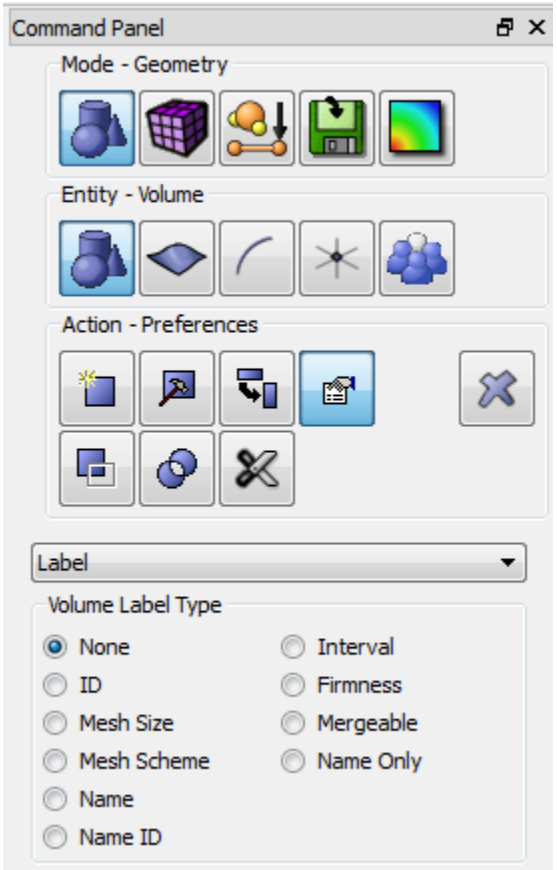
Size - The mesh size set on this entity.

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:



Colors

Specifying Colors in Commands

There are five ways to refer to a color in a command. They are

1. **<Color_Name>**
2. **User "name"**
3. **ID <id>**
4. **Default**
5. **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 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 fifth option refers to the current highlight color.

draw curve 1 tangent color highlight

User-Defined Colors

CUBIT has a palette of 85 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.

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>.

A maximum of 15 user-defined colors may be stored at one time, so it may be necessary to clear a color definition. 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>}

Color Background {<color_name>| id <color_id>} [<color_name2>|id <color_id2>]

Color Block <block_id_range>{<color_name> | id <color_id>}

Color Body <body_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | Default}

Color Curve <curve_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | Default}

Color Group <group_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | Default}

Color Highlight {<color_name>| id <color_id>}

Color Lines <color_name>

Color NodeSet <id_range> { <color_name> | id <color_id> | Default }

Color SideSet <id_range>{ <color_name> | id <color_id> | Default }

Color Surface <surface_id_range> [Geometry|Mesh] {<color_name>|Default}

Color Title {<color_name>|id <color_id>}

Color Volume <volume_id_range> [Geometry|Mesh] {<color_name>| id <color_id> | 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> | default}

Color Global Surface {<color_name>| id <color_id> | default} Curve {<color_name>| id <color_id> | default} Vertex {<color_name>| id <color_id> | 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.

Geometry, Mesh, and BC Entity Visibility

The visibility of geometric, mesh, and bc entities can be turned on or off, either individually, by entity type, by general entity class (mesh, geometry, bc, etc.), or globally. Note that these commands do not refresh automatically. To refresh type **display** or **graphics flush** or click in the display window.

The commands to set the visibility are:

> { {Body|Curve|Surface|Volume} <range> } [Mesh][Geometry] Visibility [On|Off]
>
> Edge Visibility [On | Off]
>
> Vertex [Visibility] [on|off]
>
> {Mesh|Geometry|BC} { [Visibility] [on|off] }

If the **Mesh** keyword is included, only the visibility of the mesh belonging to the specified entity is affected. Similarly, if the **Geometry** keyword is included, only the visibility of the geometry is affected. If the **BC** keyword is included, only the visibility of the BCs and Genesis entities are affected. Including none of these keywords is identical to using all keywords.

BC visibility commands also affect genesis entities such as blocks, sidesets, and nodesets. The commands to set the visibility for genesis entities are:

> **{Block|Nodeset|Sideset} <id-range> visibility [{on|off}]**

Genesis entities and boundary conditions are best viewed with geometry and mesh visibility off and BC visibility on.

Entity visibility is also controlled via context (right-click) menus in the Tree and in the graphics window.

After visibility is turned off, all associated entities (geometry, mesh or bc/genesis) will remain invisible until visibility is turned on again. This is true no matter what other visibility commands are entered.

The GUI tool bar buttons for manipulating geometry, bcs, and mesh visibility are the following:

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 **edge** visibility is off, mesh edges will not be drawn when mesh faces are drawn.

If **vertex** visibility is turned on, the vertices of the geometry become visible. The default for vertex visibility is off.

After turning mesh visibility off, all mesh will remain invisible until mesh visibility is turned on again. This is true no matter what other visibility commands are entered.

Similarly, after turning geometry visibility off, all geometry will remain invisible until geometry visibility is turned on again. This is true no matter what other visibility commands are entered.

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.

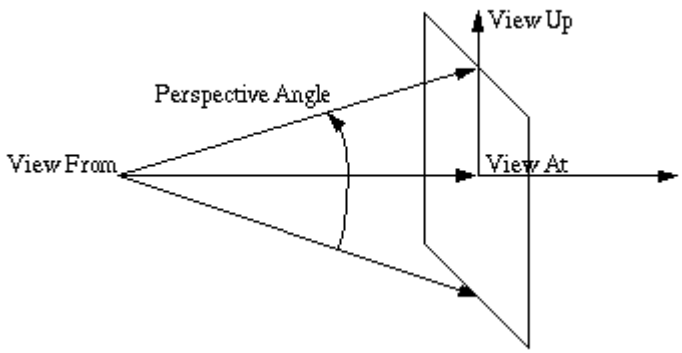


Figure 1: Schematic of From, At, Up, and Perspective Angle

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 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.

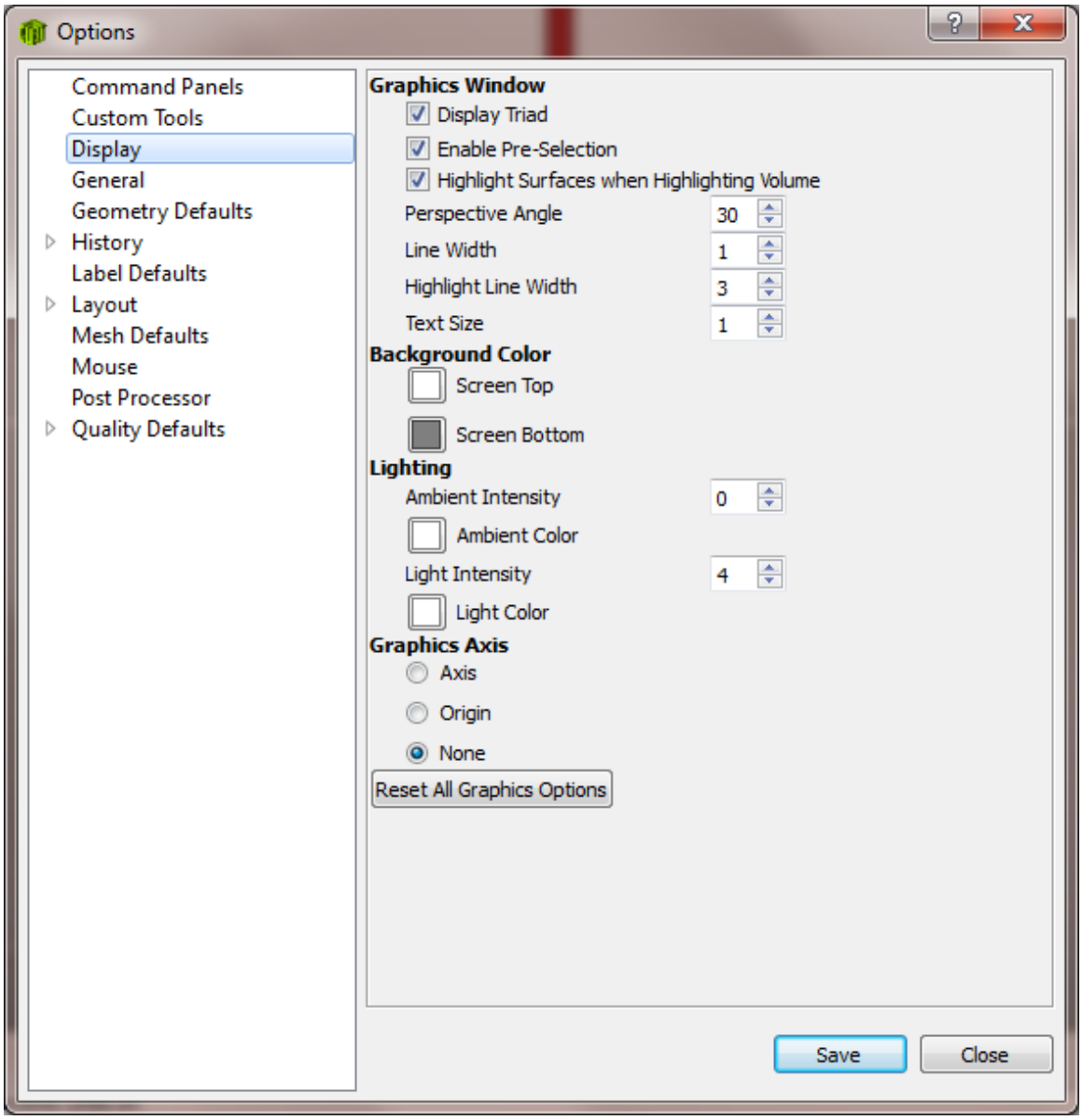
Graphics Ambient Intensity {<intensity> | <r g b>}

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:


Options
Command Panels
Custom Tools
Display
General
Geometry Defaults
History
Label Defaults
Layout
Mesh Defaults
Mouse
Post Processor
Quality Defaults
Graphics Window
Display Triad
Enable Pre-Selection
Highlight Surfaces when Highlighting Volume
Perspective Angle
30
Line Width
1
Highlight Line Width
3
Text Size
1
Background Color
Screen Top
Screen Bottom
Lighting
Ambient Intensity
0
Ambient Color
Light Intensity
4
Light Color
Graphics Axis
Axis
Origin
None
Reset All Graphics Options
Save
Close

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>

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.

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.

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:

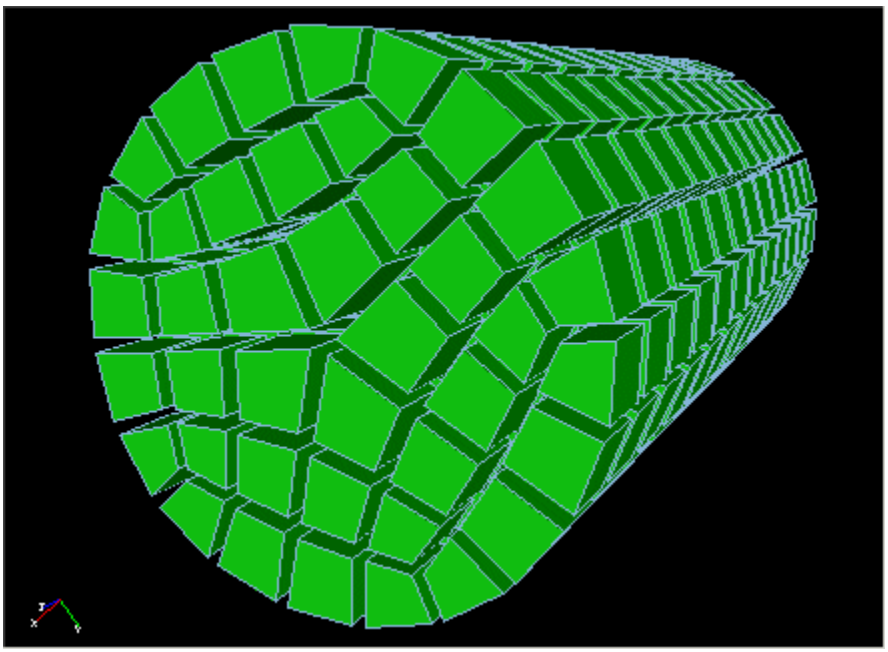
- centers the view on all visible entities (Zoom Reset)
- turns all labeling off
- turns vertex visibility off
- turns mesh and geometry visibility on
- moves the graphics camera back to its original position (View Reset)

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.



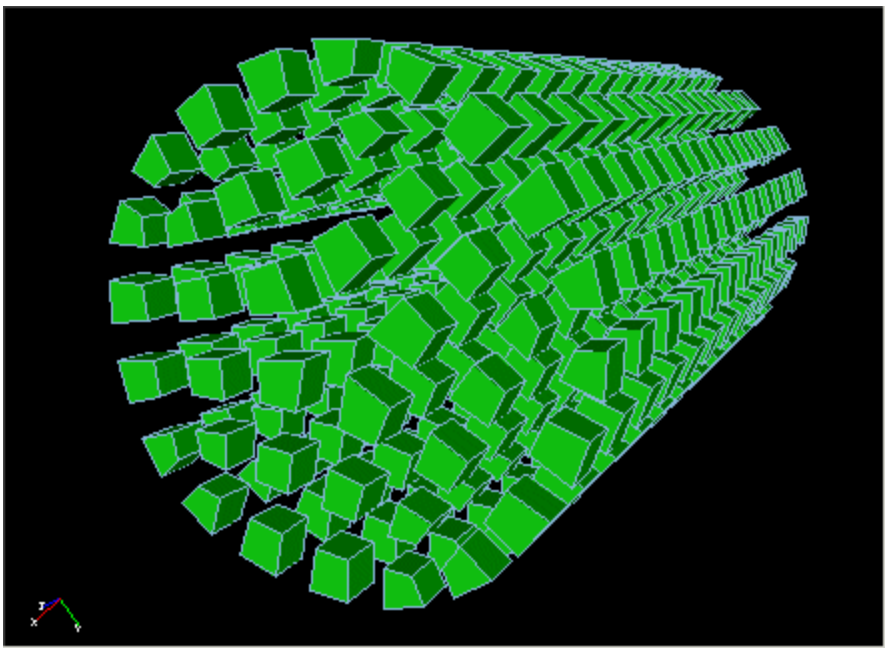


Figure 1. Top: shrink=0.2, Bottom: shrink=0.5

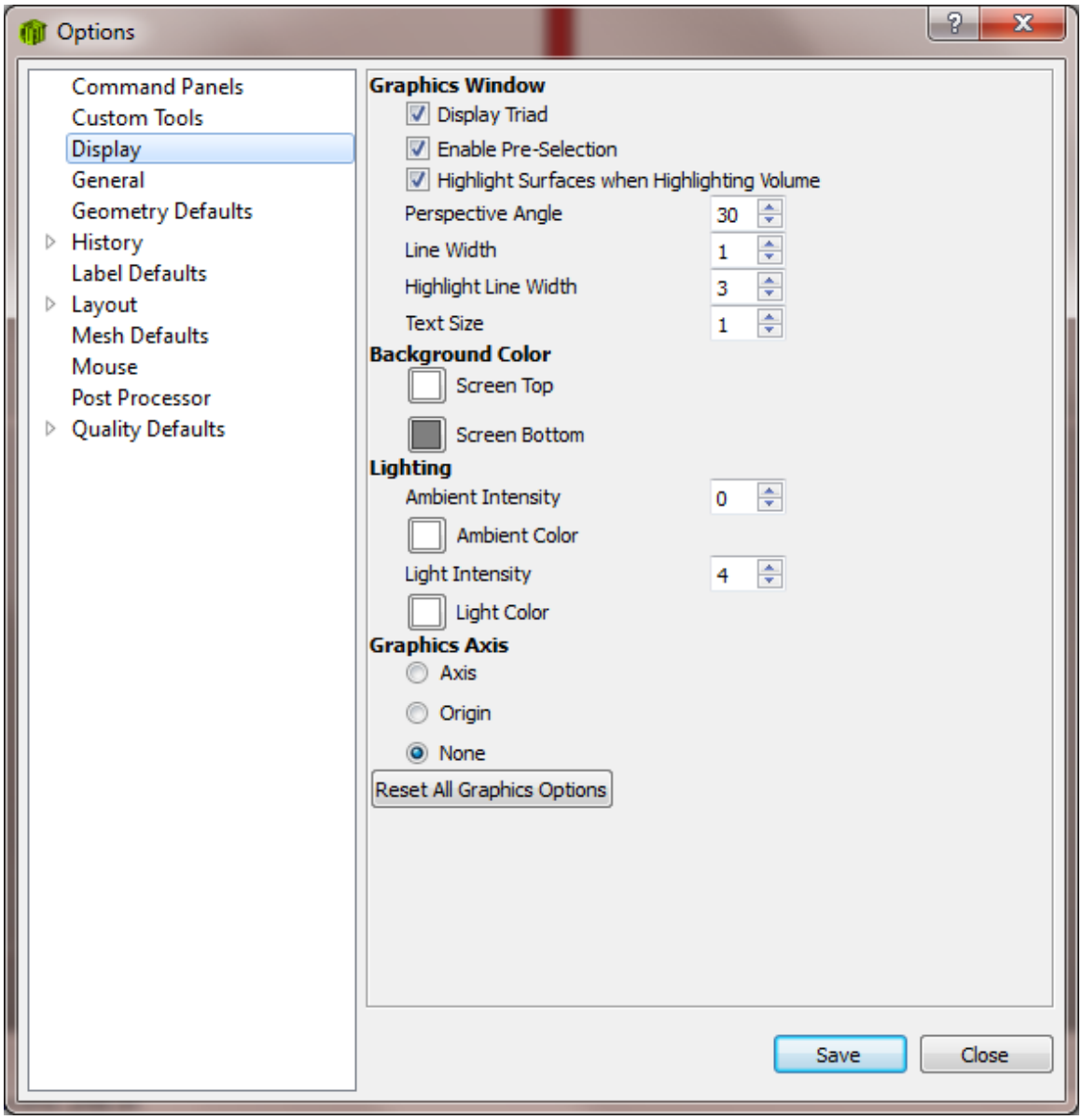
Facet Tolerance

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:


Options
Command Panels
Custom Tools
Display
General
Geometry Defaults
History
Label Defaults
Layout
Mesh Defaults
Mouse
Post Processor
Quality Defaults
Graphics Window
Display Triad
Enable Pre-Selection
Highlight Surfaces when Highlighting Volume
Perspective Angle
30
Line Width
1
Highlight Line Width
3
Text Size
1
Background Color
Screen Top
Screen Bottom
Lighting
Ambient Intensity
0
Ambient Color
Light Intensity
4
Light Color
Graphics Axis
Axis
Origin
None
Reset All Graphics Options
Save
Close

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

***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" identifier, 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

   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

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!)

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.

Entity Selection

- Command Line Entity Specification
- Extended Command Line Entity Specification
- Selecting Entities With the Mouse

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.

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.

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, Expand	These keywords are used the same way as in basic entity specification. For example: **draw surface all** **draw surface 1 to 5 step 2 curve 1 to 3 in body 4 to 8 by 2** **draw hex in face in surface 2** **draw node in hex in face in surface 2 curve 1 2 5 to 50 except 2 3 4**
Not	Not flips the logical sense of an expression - it changes true to false and false to true. For example: **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: **list curve with length < length of curve 5 ids**
And, Or	These logic operators determine how multiple criteria are combined. **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: **draw surface with x_max <= 3**

	draw volume with z_max <>12.3
+ - * /	These arithmetic operators work in the traditional manner. **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. **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.
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.
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_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 entity. Only elements of the same dimension as the entity are counted (number of hexes in a volume, number of faces on a surface, etc.)
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 point at the center of the entity's bounding box.
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.
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.

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.

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

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, hold down the control key and 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 shift and control keys 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.

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.

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
```

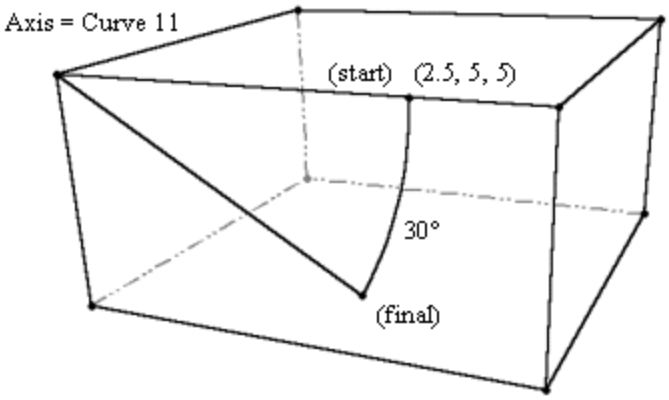


Figure 1 - Swinging a Location

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}
- Arc 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

Arc Center

arc_center

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

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

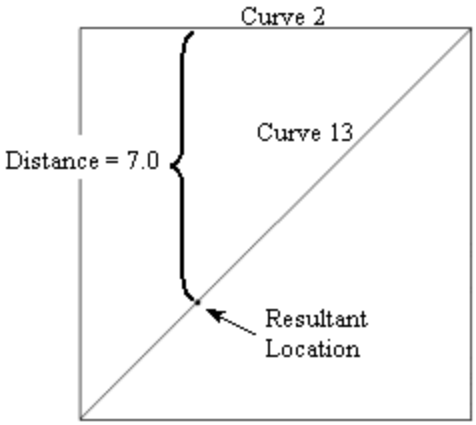


Figure 1 - Location on a Curve a Distance from Another Curve

{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 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

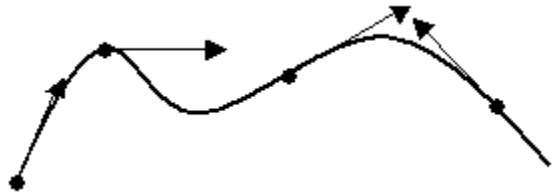


Figure 1 - Tangents to a Curve

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 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
- Specify a direction and a location
- Revolve an axis about an axis

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 an origin and a vector

{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

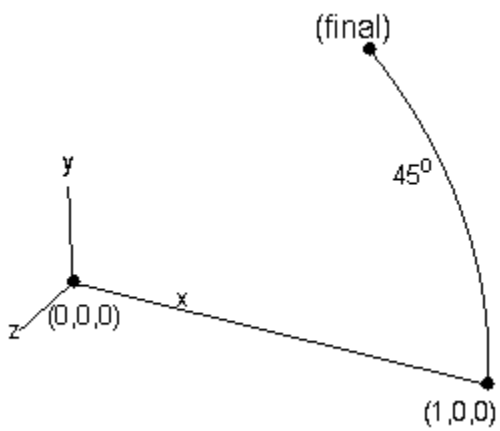


Figure 1 - Swinging a point about the z-axis

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.

Revolve an axis about an axis

[Axis {options} 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

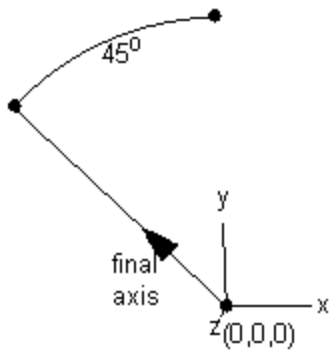


Figure 2 - Revolving an axis about another axis

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 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

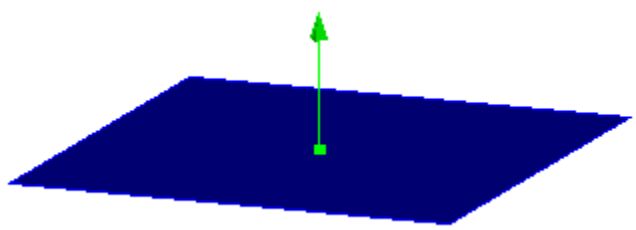


Figure 1. Specifying a plane with a location and surface normal

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.

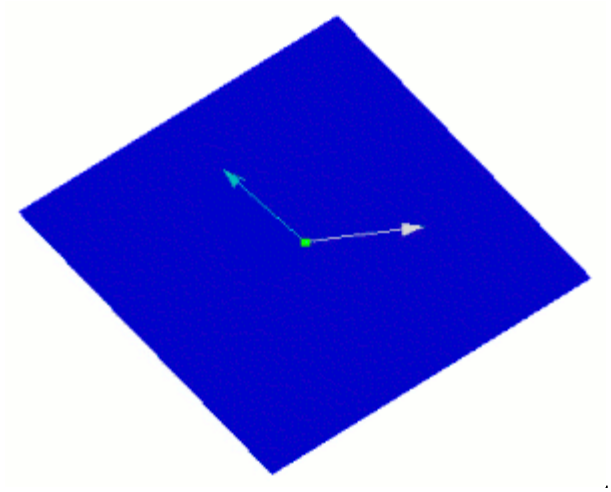


Figure 2. Specifying a plane with a point and 2 in-plane vectors

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

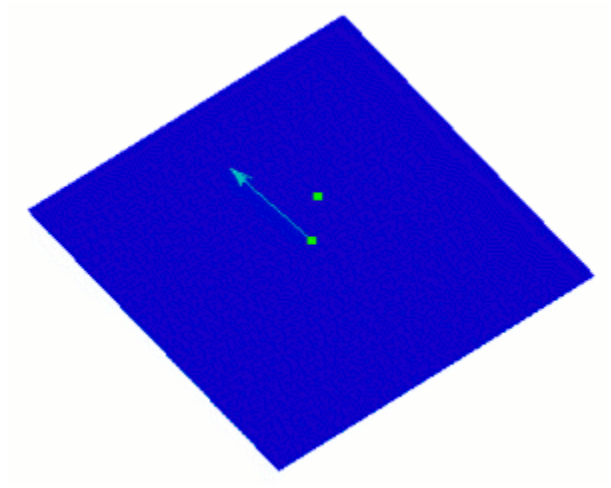


Figure 3. Specifying 2 locations and 1 direction on the plane

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

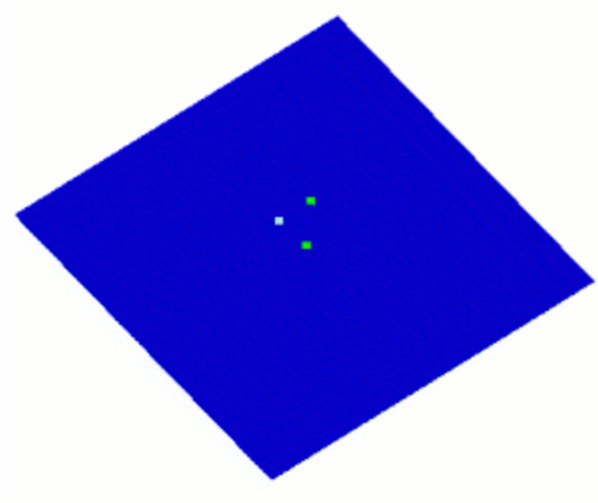


Figure 4. A plane specified by three points

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

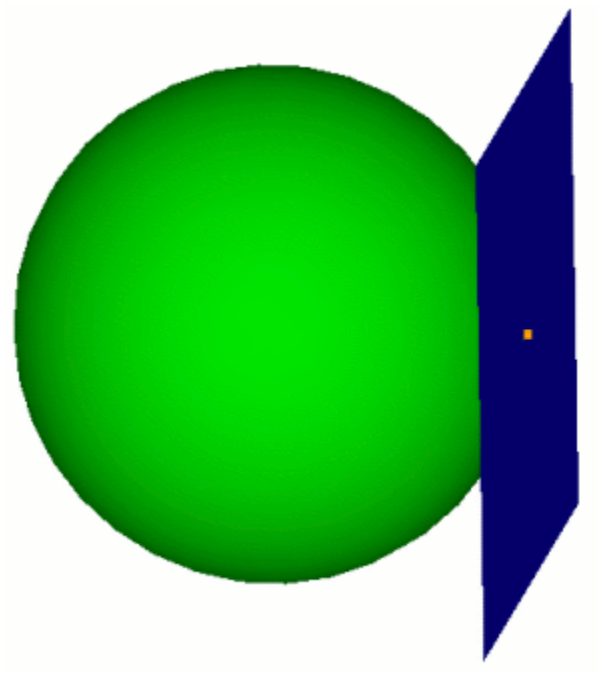


Figure 5. Specifying a Tangent plane to a Surface

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

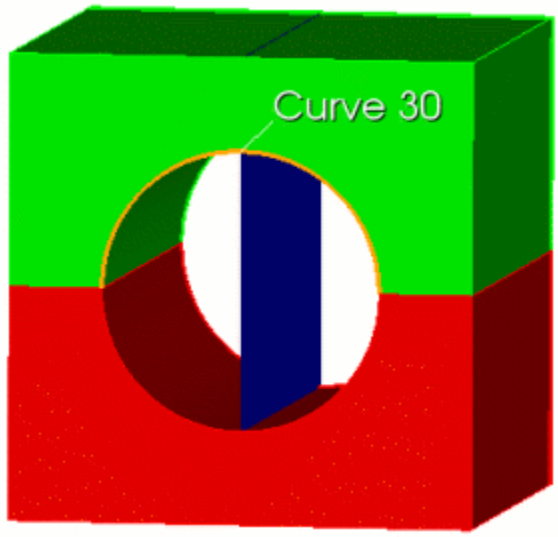


Figure 6. Draw Plane Normal to Curve

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.

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]

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

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 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.

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 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.)

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 and Debug message printing can be turned on or off (or toggled) with a set command; error messages are always printed. Debugging output can be redirected to a file. Current message printing settings can be listed.

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

Set {Echo|Info|Warning} [On|Off]

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

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

[Set] Debug <index> Terminal

Message flags can also be set using command line options, e.g. -**warning={on|off}** and -**information={on|off}.** Debug flags can be set on 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**.

In addition to the major categories, there are some special purpose output settings.

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

List Logging

If logging is enabled, all echo, info, warning, and error messages will be output both to the terminal and to the logging file. The **resume** option will append to the logfile, if it exists, instead of writing over it. If the logfile doesn't already exist, it will be created.

List Journal Title "<title_string>"

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

List Title

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

List Default Block

Set Default Block {ON|off}

The List Default Block command lists which type of geometric entities for which blocks will automatically be generated at export if no other blocks have been specified. The Set Default Block command will toggle whether these default blocks are written, or not, during the export operation when no other blocks have been specified.

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. Next come the other message settings, 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.

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.

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.

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.

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 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 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.

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.

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 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 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|STL] "<filename>" [Feature_Angle] [LINEAR||Spline] [MERGE|No_merge] [Make_elements] [Stitch] [Improve]

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.

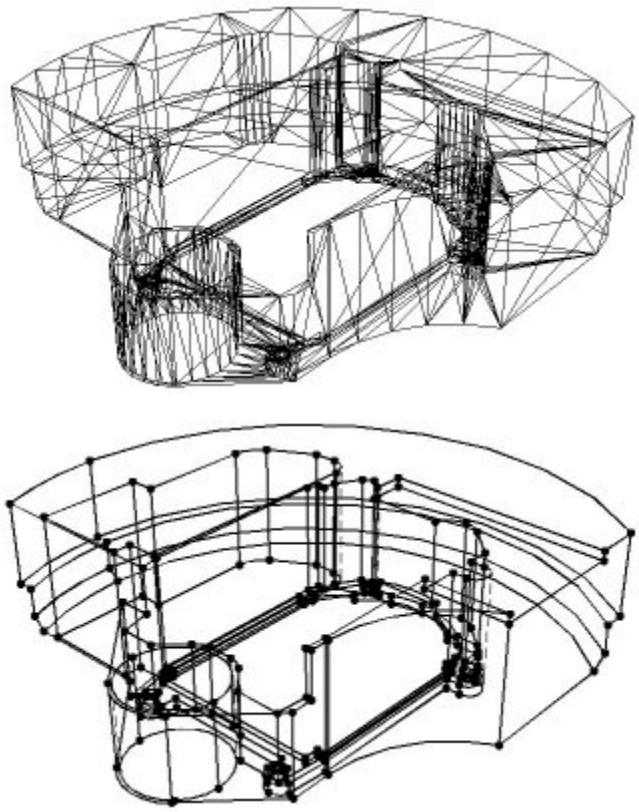


Figure 1. Example of faceted model and the resulting solid model created in CUBIT from the facets.

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.

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 if vertex i
x<i> y<i> z<i> = location of vertex i
fid<j> = facet ID if 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.

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.

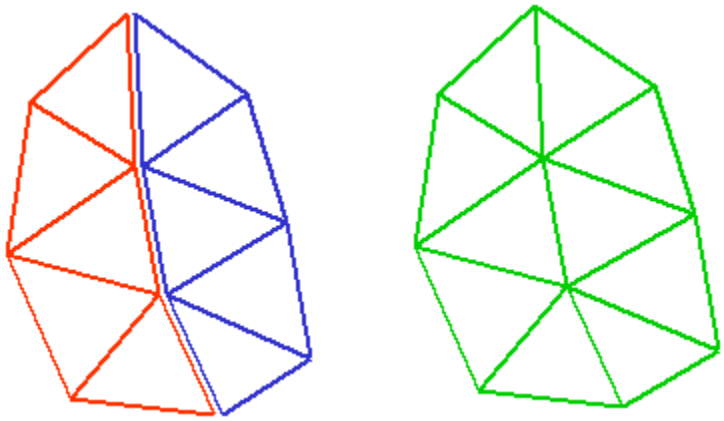
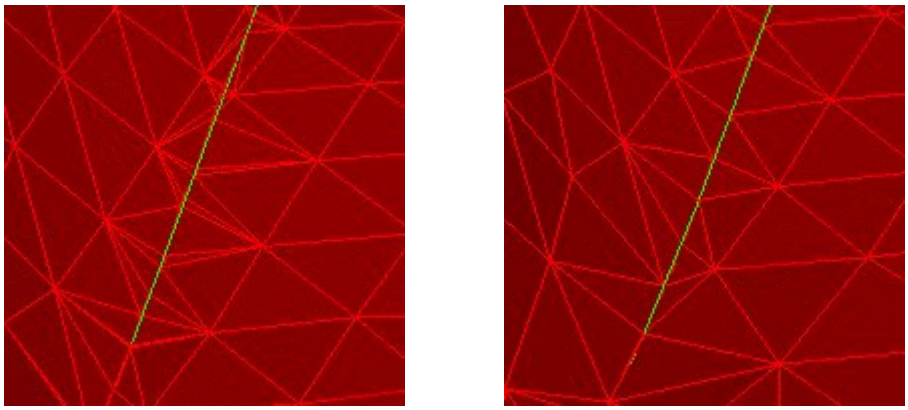


Figure 2. Example use of the stitch option on import.

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 Granite Files

As of version 13.0, native Granite models are no longer supported.

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

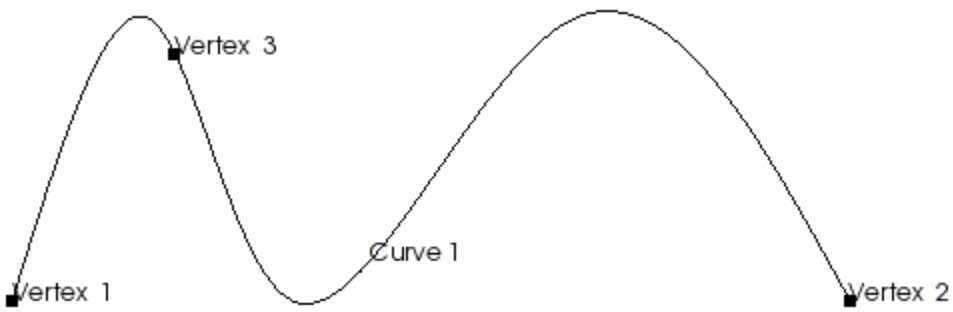


Figure 1. Create Vertex a Fraction of the length of a Curve

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>]

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
- 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
- Projecting onto Surface
- Helix

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 [first angle <val=0> last 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, **first angle** will default to 0 and **last 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. **First angle** tells Cubit where to start the ellipse -- the angle from the first axis (v1 - v3) specified. **Last 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.

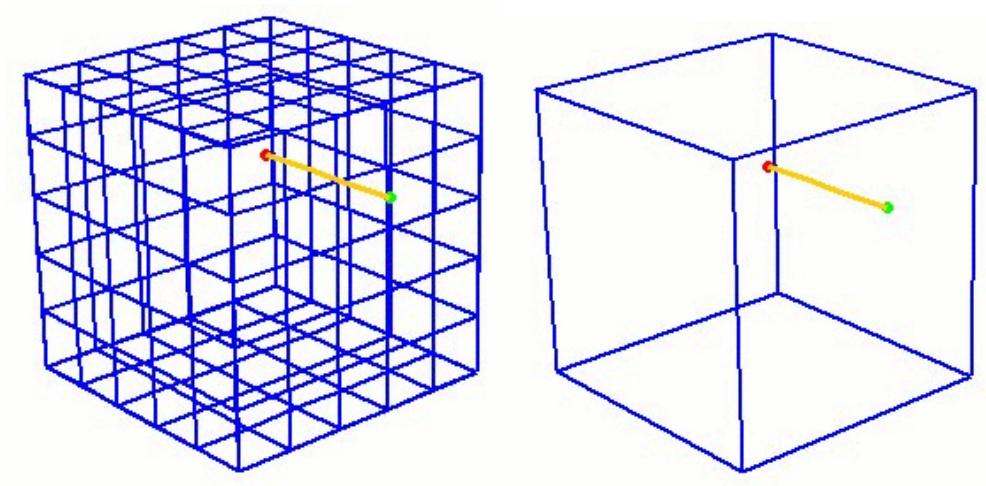


Figure 1. Example of curve created from mesh

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. Projecting onto a Surface The project command allows you to make an imprint of a surface or set of curves onto another surface. The command syntax is as follows:

Project Curve <id_list> Onto Surface <surface_id> [Imprint [Keepcurve] [Keepbody]] [Trim]

Project Surface <id_list> Onto Surface <surface_id> [Imprint [Keepcurve] [Keepbody]]

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 that bound the surface (i.e. a group of free curves on the projecting surface).

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

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.

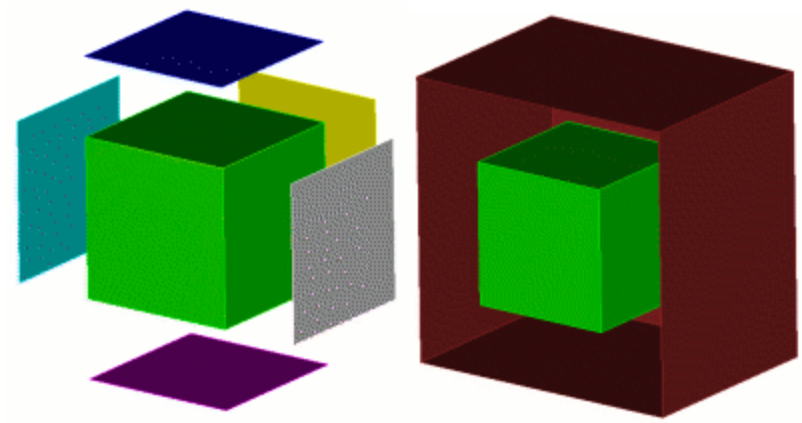


Figure 1. Offsetting surfaces to form individual surfaces or sheet bodies

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]

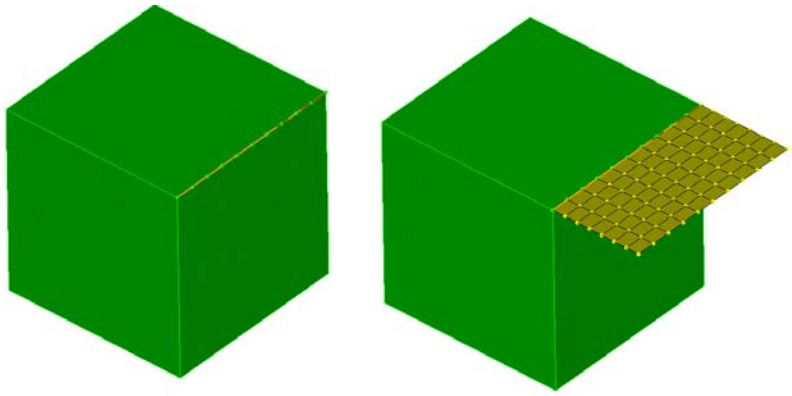
Sweep Curve <curve_id_range> Along Curve <refcurve_id_range> [Draft_angle <degrees>] [Draft_type <integer>] [Include_mesh] [Keep] [Rigid]

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.

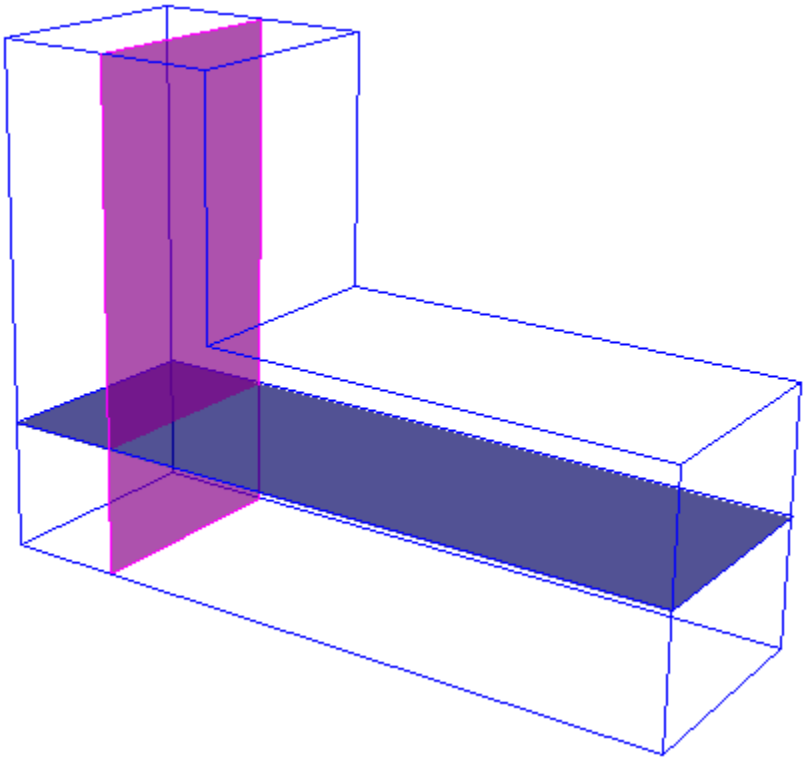


Figure 2. Multisurface created with the Create Midsurface command

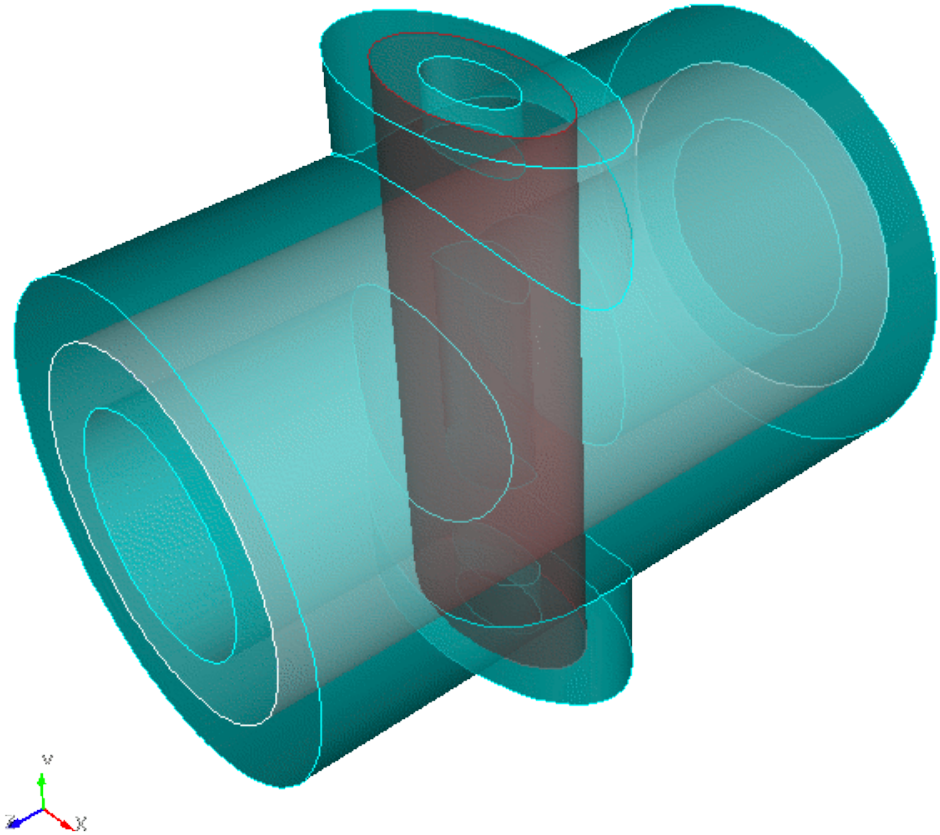


Figure 3. Midsurface created from 2 pairs of cylindrical surfaces

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

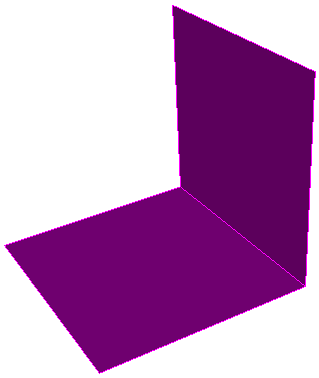


Figure 4. Midsurface created from a volume

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

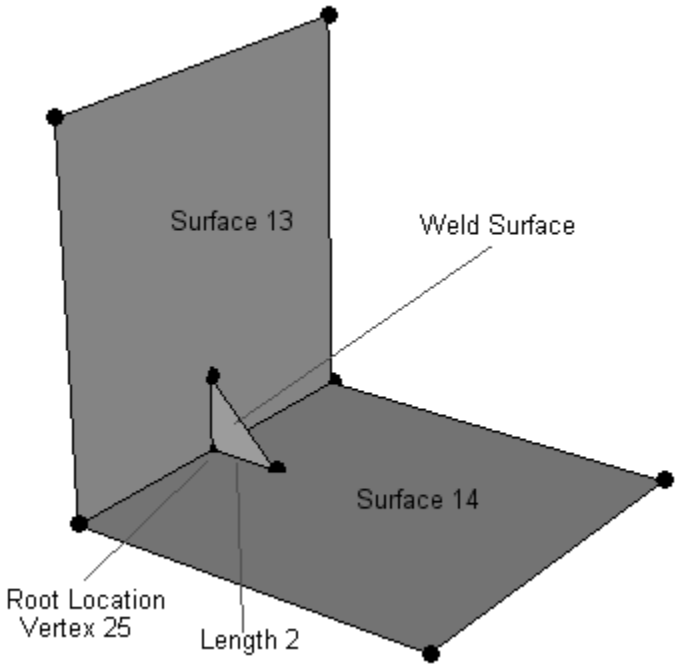


Figure 5. Weld Profile surface with length and root specifications

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>]

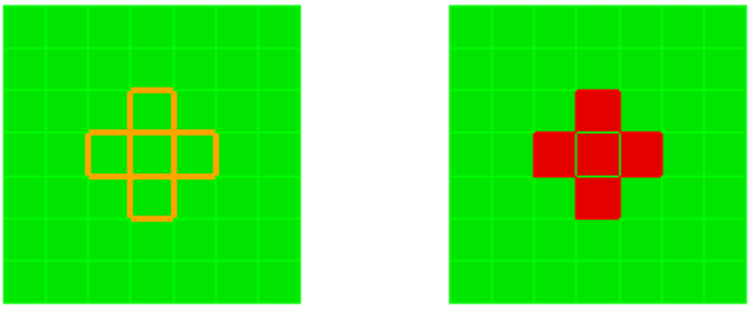


Figure 6. Acis Surface created from a Set of Quadrilaterals

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 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.

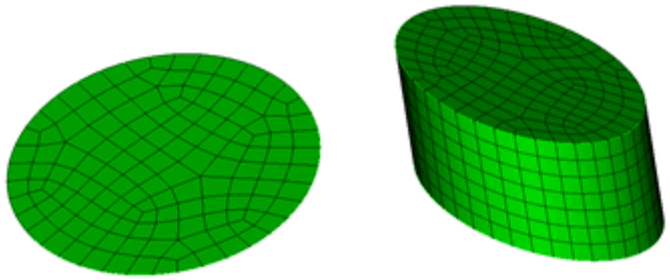
anchor_entity: The default behavior for the sweep command is to move the source surface along a path to create a new 3D solid. The anchor_entity option instructs the sweep to leave the source surface in its original location.

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.

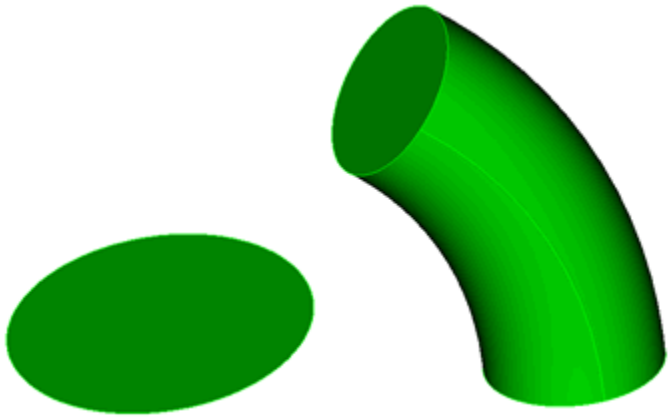
Sweep Surface {<surface_id_range>} Vector <x_vector y_vector z_vector> [Distance <distance_value>] [switchside] [Draft_angle <degrees>] [Draft_type <0|1>][rigid][anchor_entity][include_mesh] [keep] [merge]



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

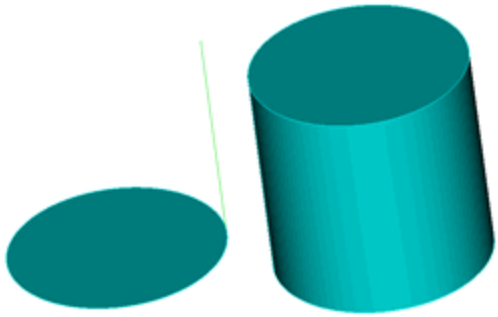
Specifying multiple surfaces that belong to the same body will not work as expected, as ACIS performs the sweep operation in place. Hence, if a range of surfaces is provided, they ought to each belong to different bodies.

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. 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.

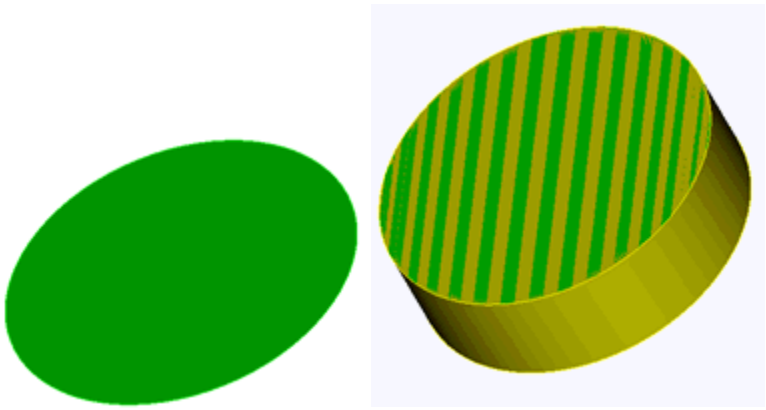


Volume generated by sweeping a surface along a reference curve

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.

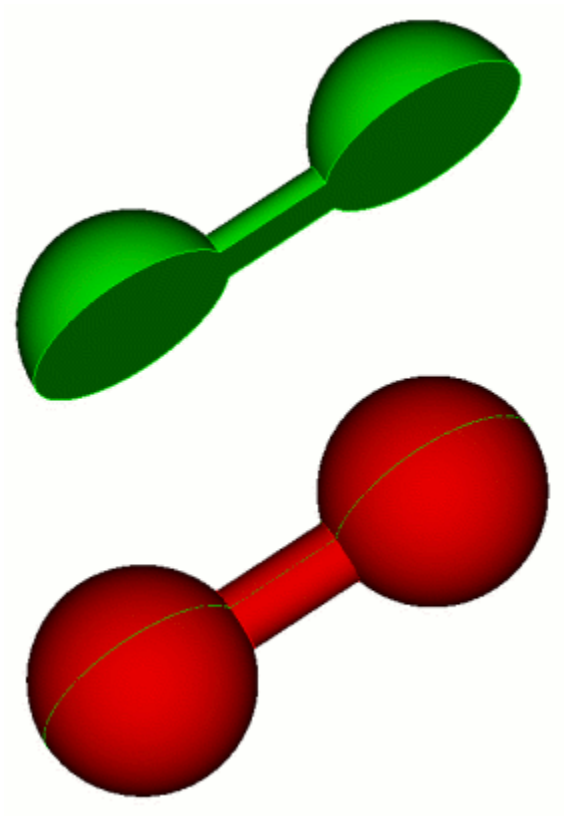


Figure 1. Sheet created from extending multiple surfaces

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 copied surfaces. 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 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 direction of the loft for each surface. The default takeoff vector for each surface is the normal at the surface centroid. One takeoff vector may 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.

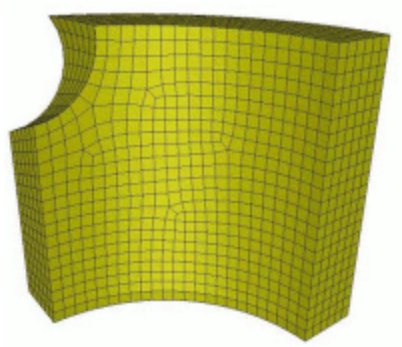


Figure 2. Mesh before loft. Single swept volume with a large paved face.

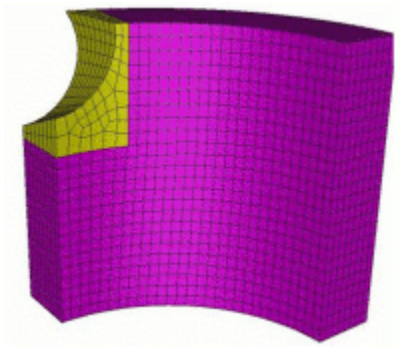


Figure 3. Mesh after loft. The yellow volume is paved and the purple volume is submapped.

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.

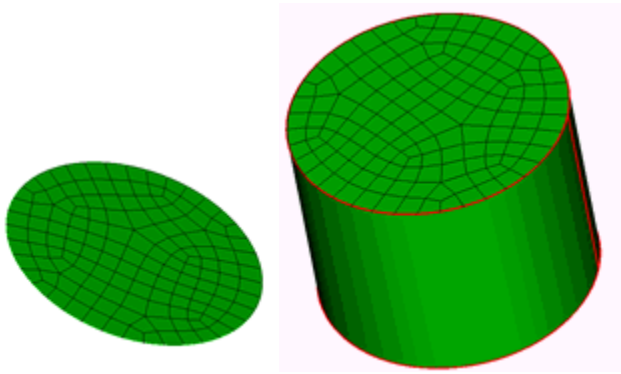
Thicken [Volume|BODY] <id> Depth <depth> [Both]

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]**

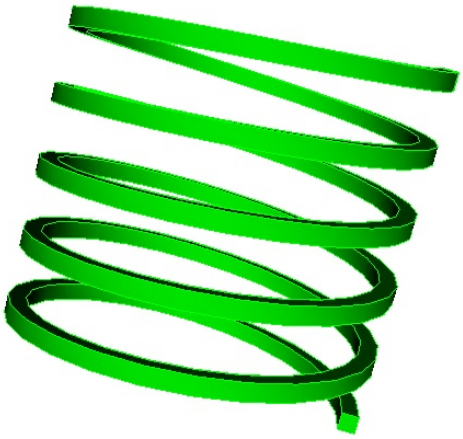
***** Specifying multiple Surfaces that belong to the same Body can cause the creation of invalid Bodies and is discouraged. *****

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 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 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>

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.

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 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>]

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.

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 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.

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**.

Align Command

The align command is a combination of the rotate and move commands.

The first align command below will transform the specified volumes by computing a transformation that would align the first surface with the second surface such that the surface centroids are coincident and the normals are pointing either in the same or opposite direction (depending on their initial alignment). The first surface need not be in the specified volumes.

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.

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.

The syntax of the align commands are:

> Align Volume <id_range> Surface <surface_id> with Surface <surface_id> [reverse] [include_merged] [preview]
>
> Align Volume <id_range> {Surface <surface_id>| Vertex <vertex_id>} {{X|Y|Z Axis}|{XY|XZ|YZ plane}} [reverse] [include_merged] [preview]
>
> Align Volume <id_range> Location {options} with Location {options} about Axis {options} [include_merged] [preview]

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.

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

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.

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.

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}

Intersect

The intersect command generates a new body composed of the space that is shared by the two bodies being intersected. Both of the original bodies will be deleted and the new body will be given the next highest body ID available. The command is:

Intersect {Volume|[Body]} <range> [With {Volume|[Body]} <range>] [Keep] [Preview]

The **keep** option results in the original bodies used in the intersect being kept.

If the **Preview** option is included in the command, the input bodies 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.

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}

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 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:

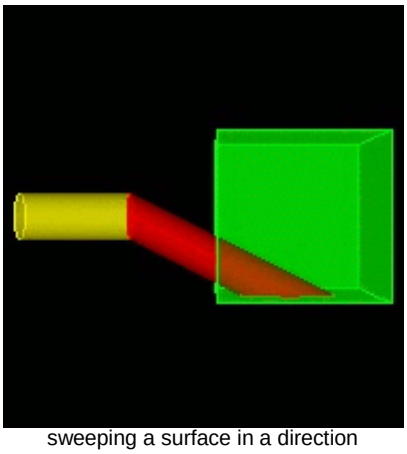
- Web Cutting by Sweeping a Surface Along a Trajectory
- Web Cutting by Sweeping a Surface About an Axis
- Web Cutting by Sweeping a Curve(s) Along a Trajectory
- Web Cutting by Sweeping a Curve(s) About an Axis

Web Cutting by Sweeping a Surface Along a Trajectory

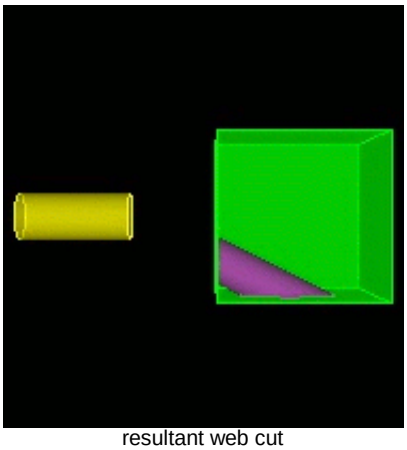
This command allows one or more surfaces to be swept, creating a volume that is used for the web cut. If more than one surface is specified, the surfaces must contain coincident curves. The surfaces are swept along a direction and some distance or perpendicular and some distance or along a curve. For 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**]

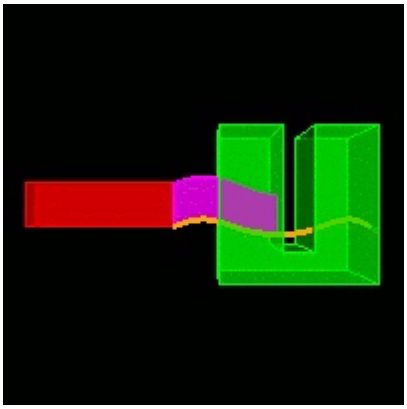
Webcut {Volume|Body|Group} <id> Sweep Surface <id_range> Perpendicular {Distance <distance> | Through_all | Stop Surface <id>} [OUTWARD|Inward] [**webcut_options**]

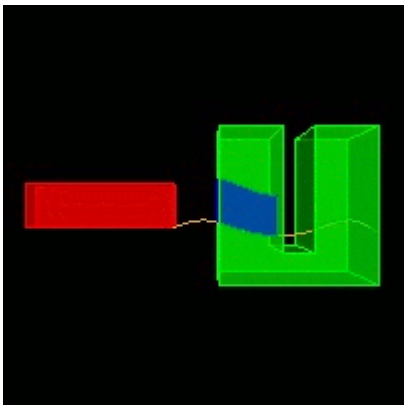


sweeping a surface in a direction



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 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.

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 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.

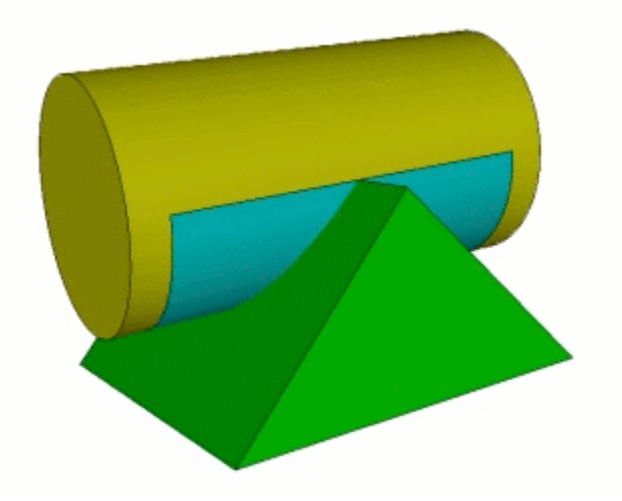


Figure 1. Cylinder cut with bounding box of prism.

Web Cutting 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.

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

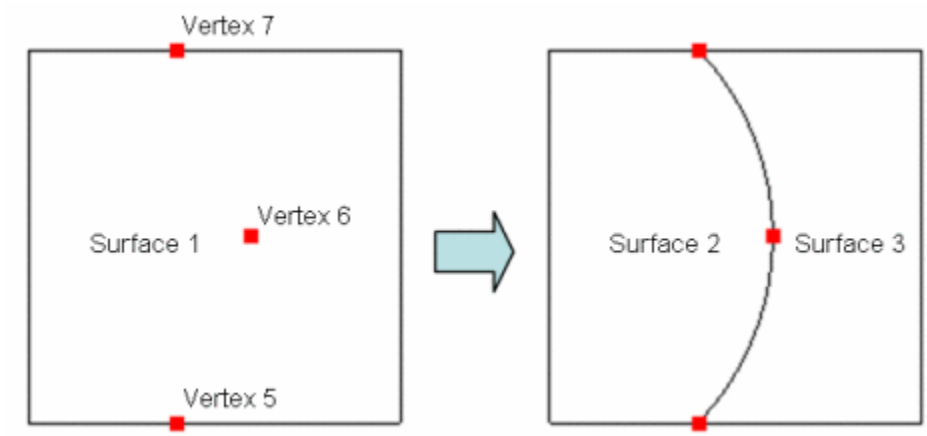


Figure 1 - Splitting Across with Multiple Locations

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

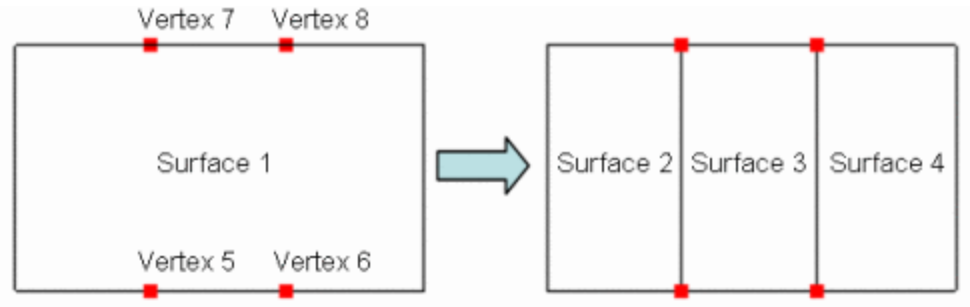


Figure 2 - Splitting Across with Pair Option

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

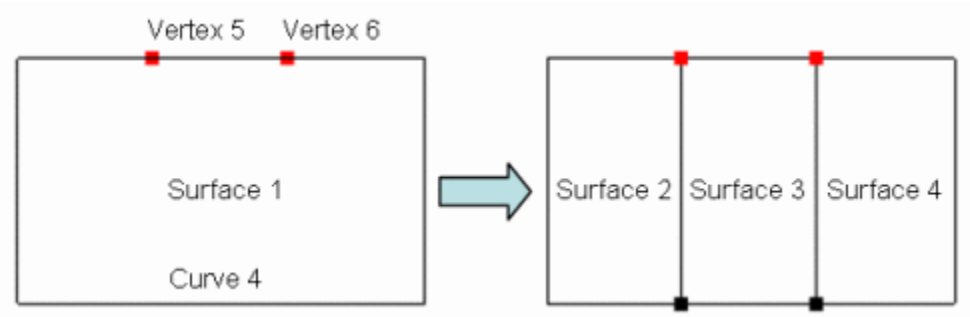


Figure 3 - Splitting Across with Onto Curve

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

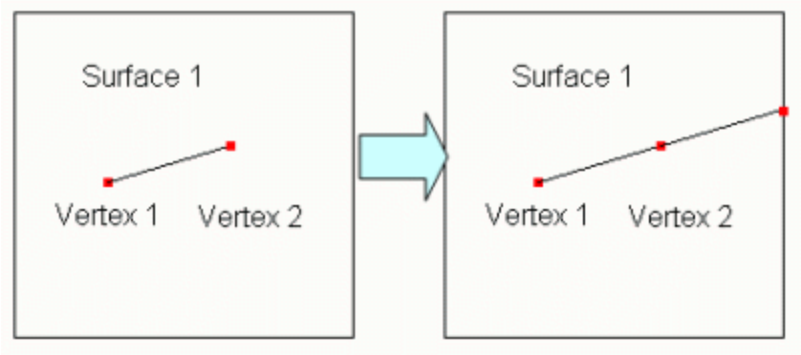


Figure 4 - Splitting by Extending Hard-line

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

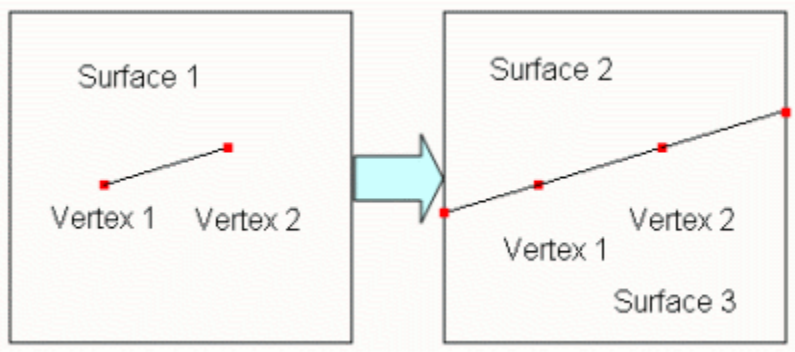


Figure 5 - Splitting by Extending with Auto Option

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

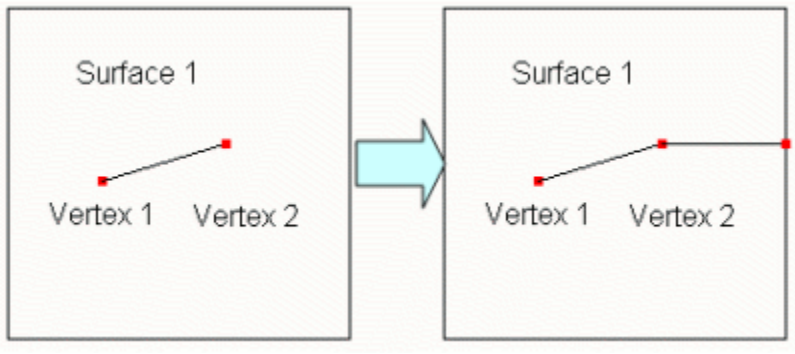


Figure 6 - Splitting by Extending a Hard Line with Normal Setting ON

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

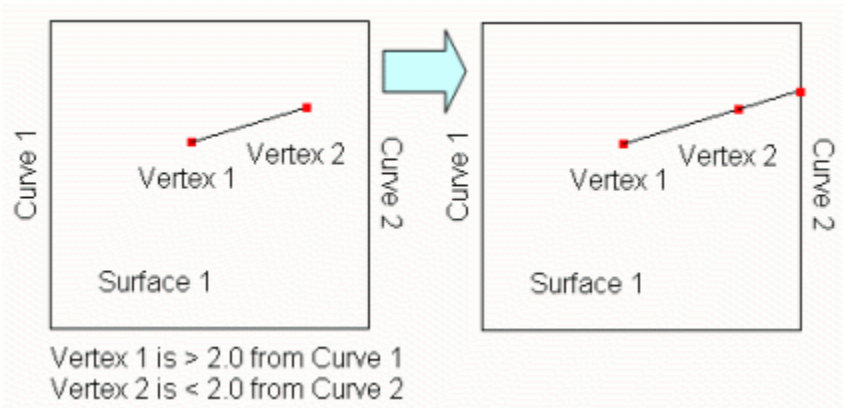


Figure 7 - Extending Hard-lines with Gap Threshold = 2.0.
(Notice Vertex 1 was not extended because it exceeded the gap threshold)

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

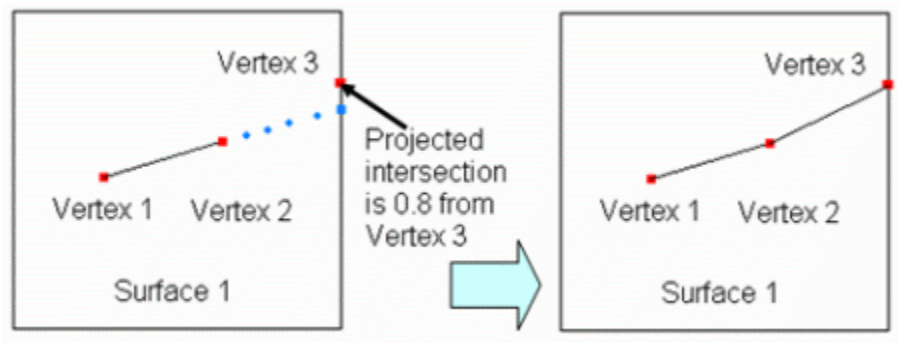


Figure 8 - Extending Hard-lines with Tolerance
(Notice the extension snapped to Vertex 3)

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.

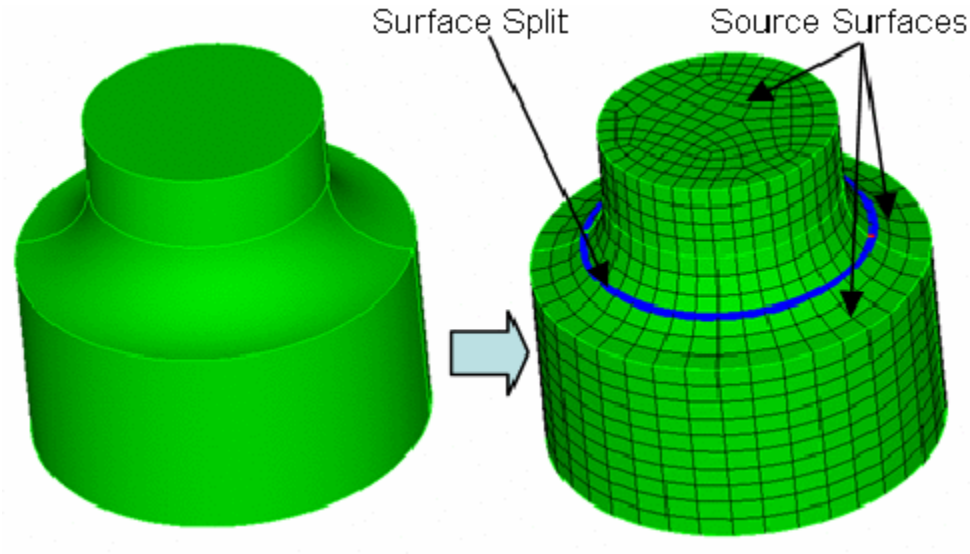


Figure 9 - Splitting Fillets to Facilitate Sweeping

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.

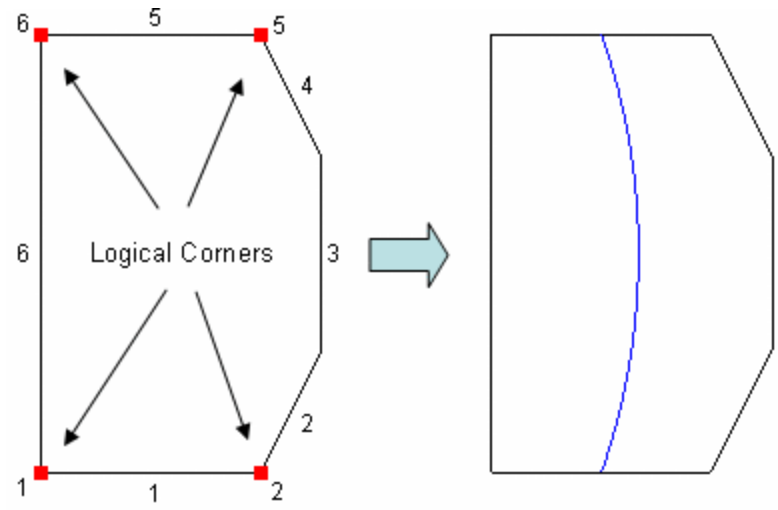


Figure 10 - Split Surface Logical Properties

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.

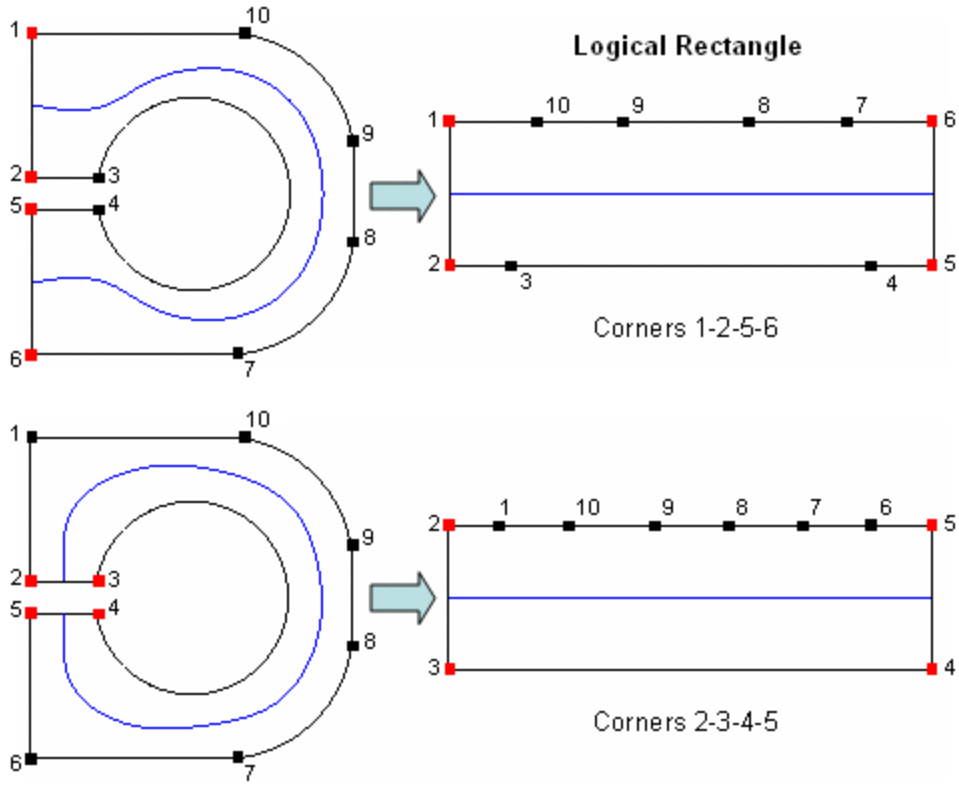
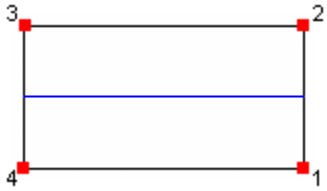
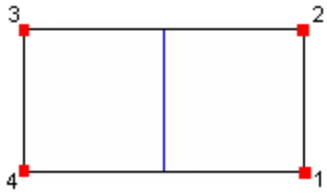
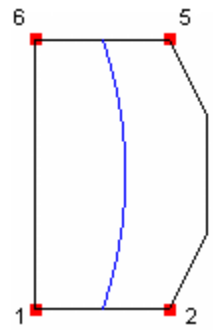
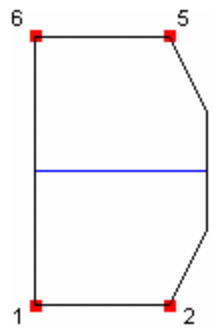
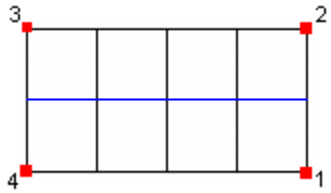
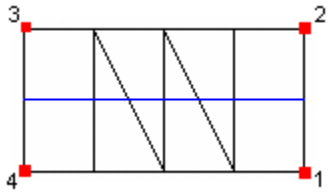
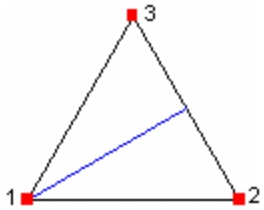


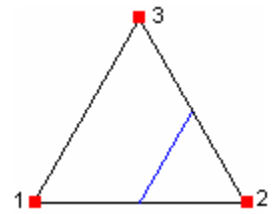
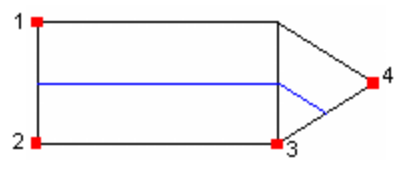
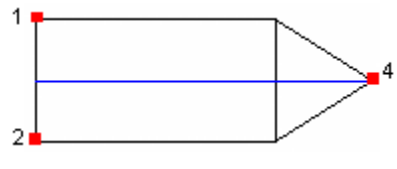
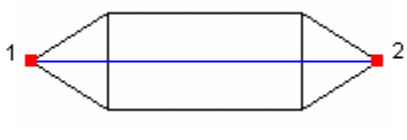
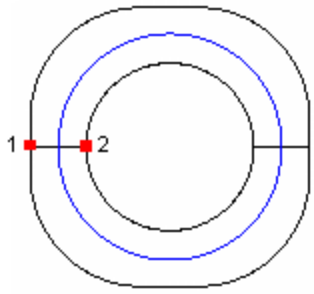
Figure 11 - Different Possible Logical Rectangles for Same Surface

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 (using aspect ratio)
	4-1-2-3 (user selected)

	1-2-5-6
	2-5-6-1
	1-2-3-4 (split is always along the chain)
	1-2-3-4 (notice triangular surfaces along the chain)
	1-1-2-3 (note side 1 of the logical rectangle is collapsed; side 3 is from vertex 2 to 3)

	1-2-2-3 (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 (selected automatically)

Split Orientation

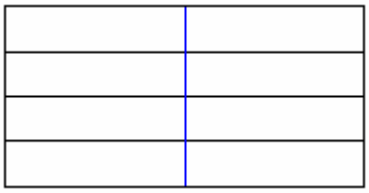
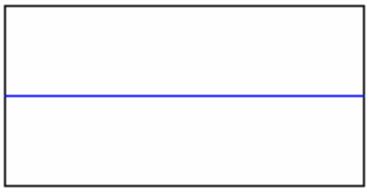
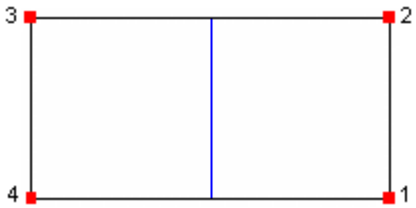
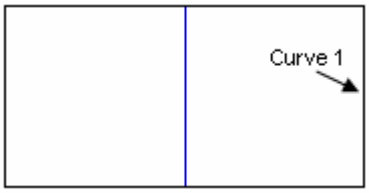
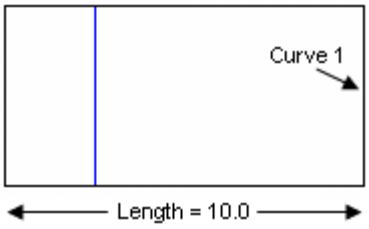
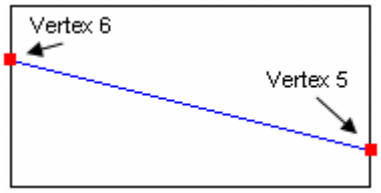
If a chain of surfaces are split, the surfaces will always be split along the chain. The command will not allow disconnected surfaces.

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 (split always starts on side 1 of the logical rectangle)
	Direction Curve 1
	From Curve 1 Fraction .75 or 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.

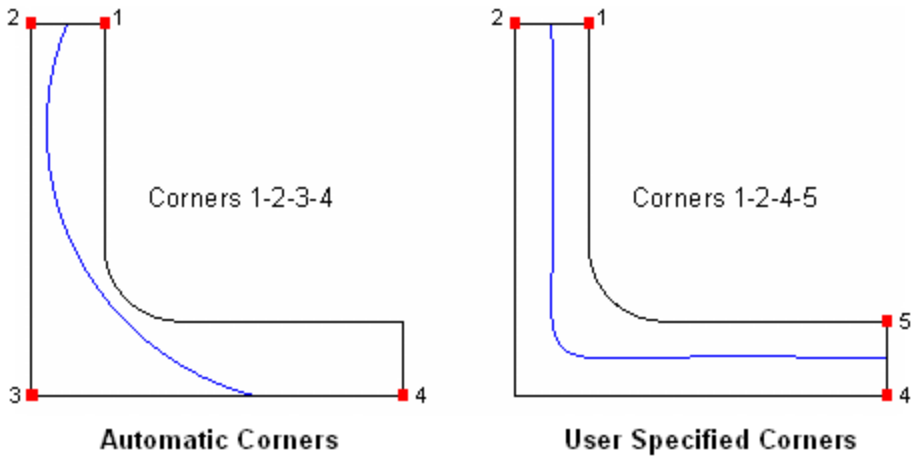
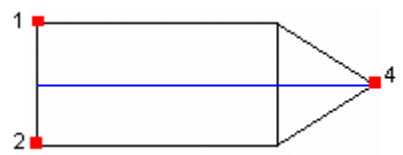
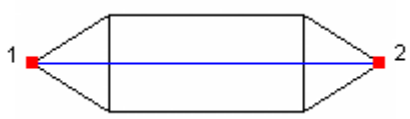


Figure 12 - Selecting the Desired Corners

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 or 4-1-2 (shortcut method)
	1-1-2-2 or 2-2-1-1 or 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.

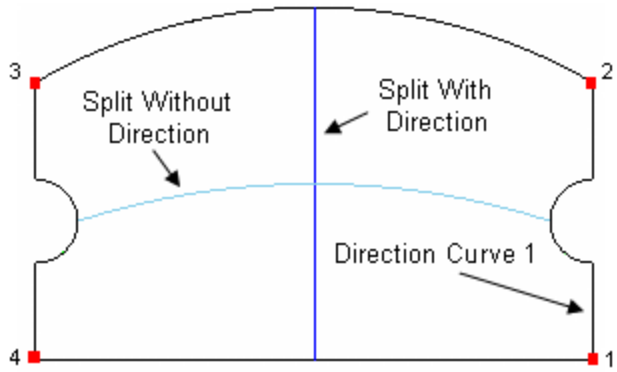
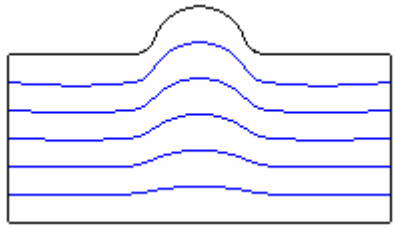
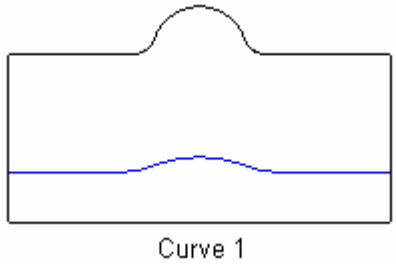
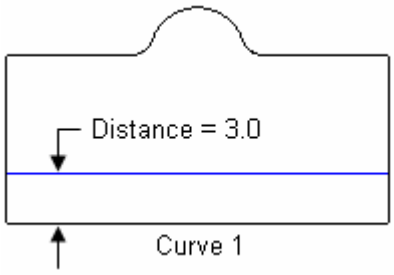


Figure 13 - Direction Specification Overrides Corner Order

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.

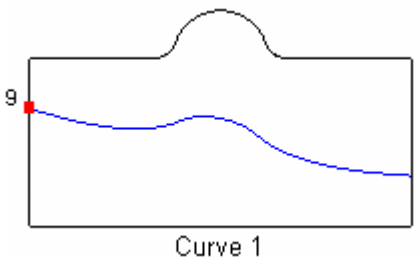
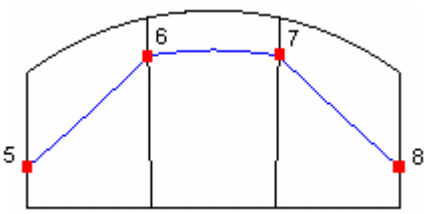
Table 5 - Segment, Fraction, Distance Examples

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

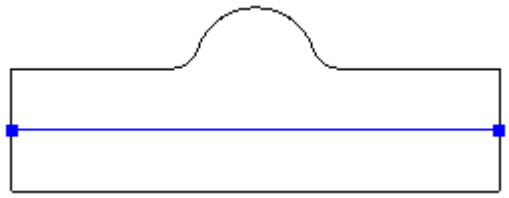
By default, split locations are calculated in 3D space and projected to the surface. As an alternative, split locations can be calculated directly in the surface parametric space. In rare instances, this can result in a smoother or more desirable split. The command option **Parametric {on|Off}** can 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.

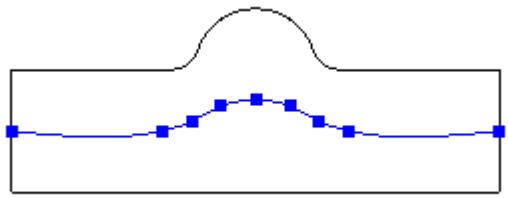
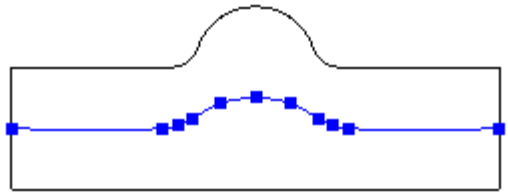
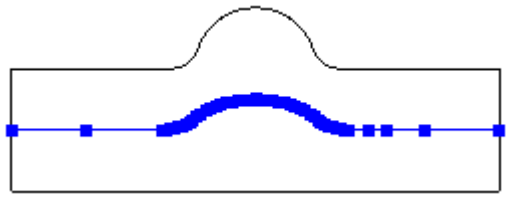
Tolerance

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

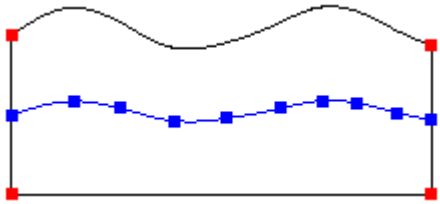
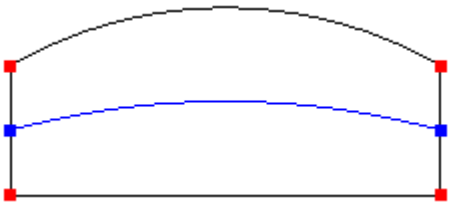
Surface	Tolerance
	2.0

	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.

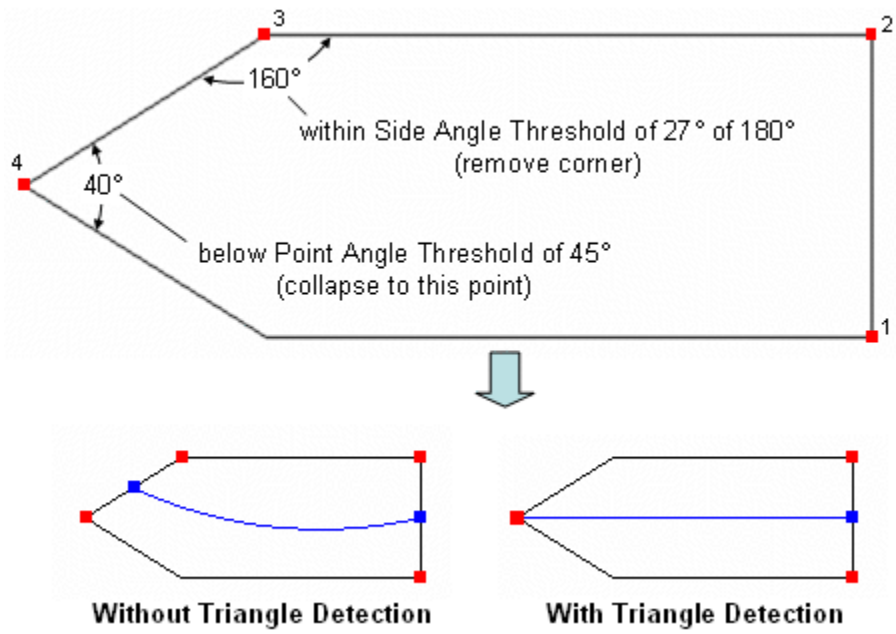


Figure 14 - Triangle Detection Settings

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

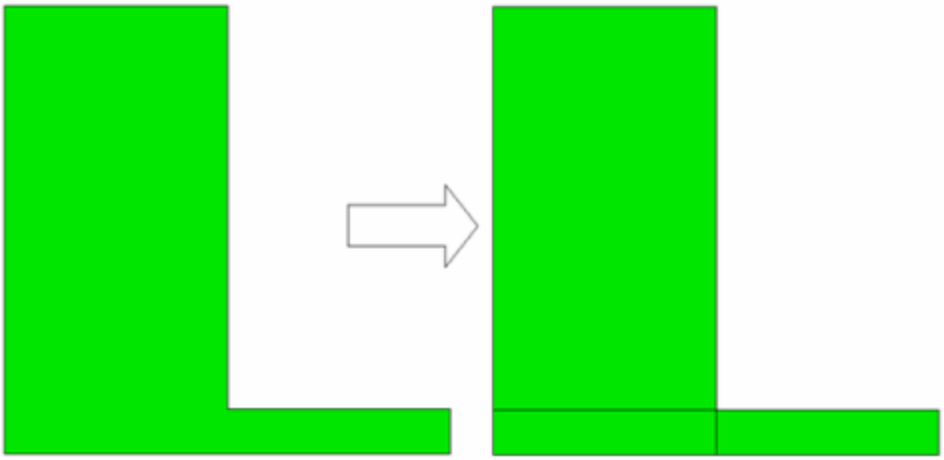


Figure 15. Split Skew applied to an L-shaped surface

The **Preview** keyword will show a graphics preview of the splitting curves. If the **Create** keyword is also specified, free curves will be created.

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 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]

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.

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>

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}

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.

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 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.

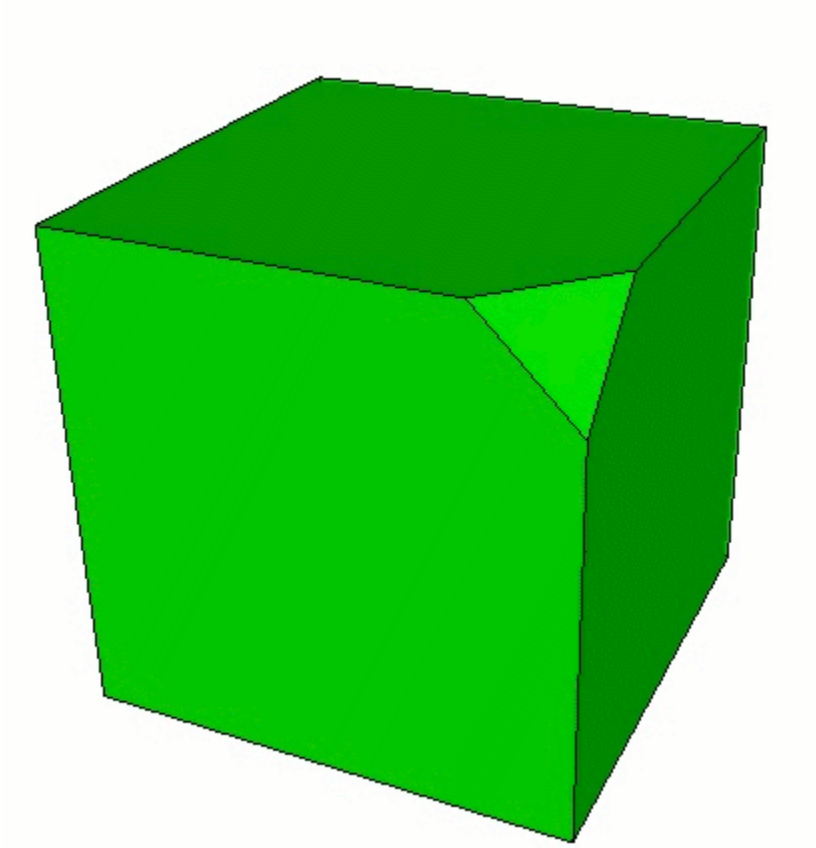


Figure 1. Tweak Vertex Chamfer

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.

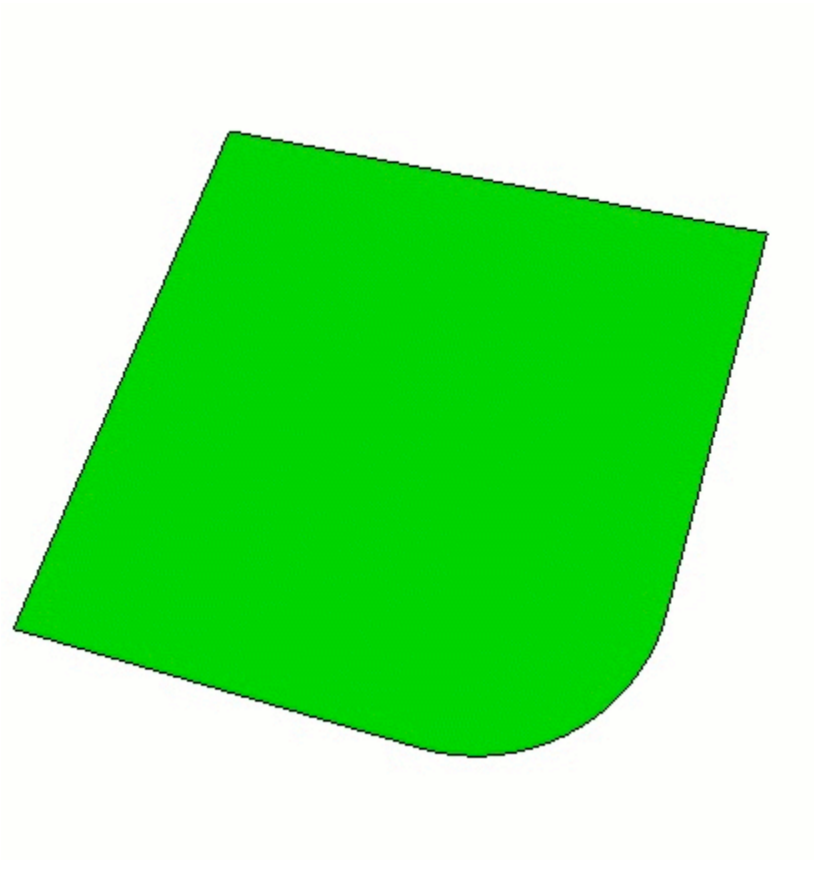


Figure 2. Tweak Vertex Fillet

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').

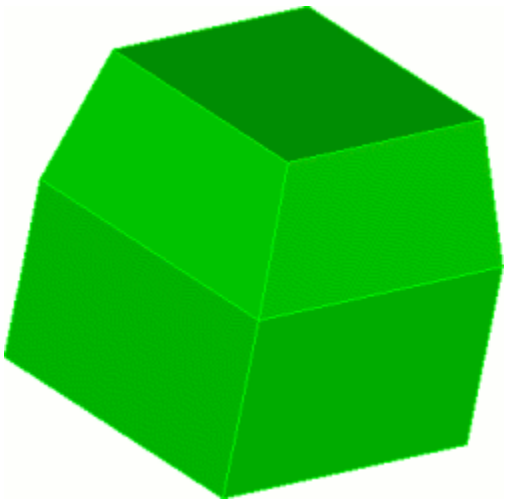


Figure 1 Chamfer with two different distances

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').

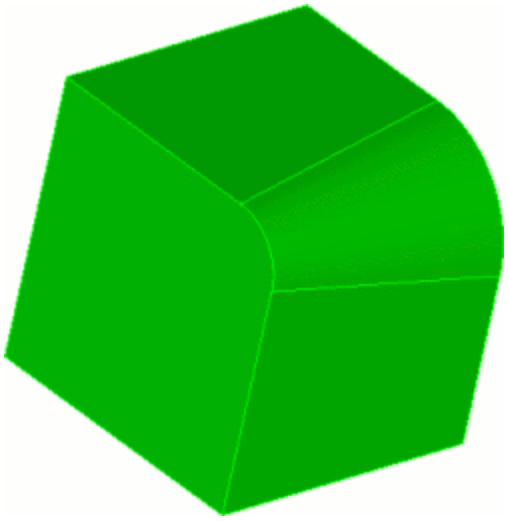


Figure 2. Fillet with two different radii

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.

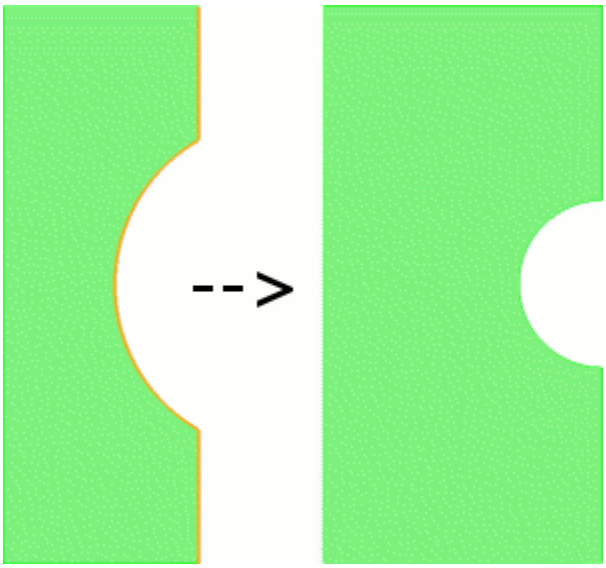


Figure 3 Offsetting a set of curves a specified distance

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.

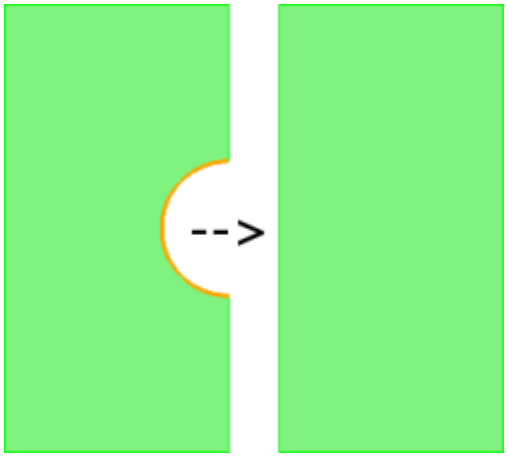


Figure 4. Removing a curve from a sheet body

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.

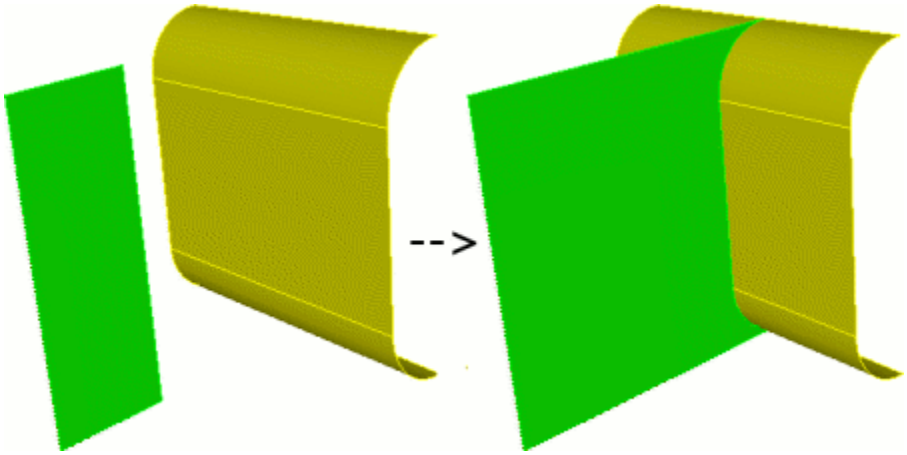


Figure 5 Tweaking a curve to multiple target surfaces

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.

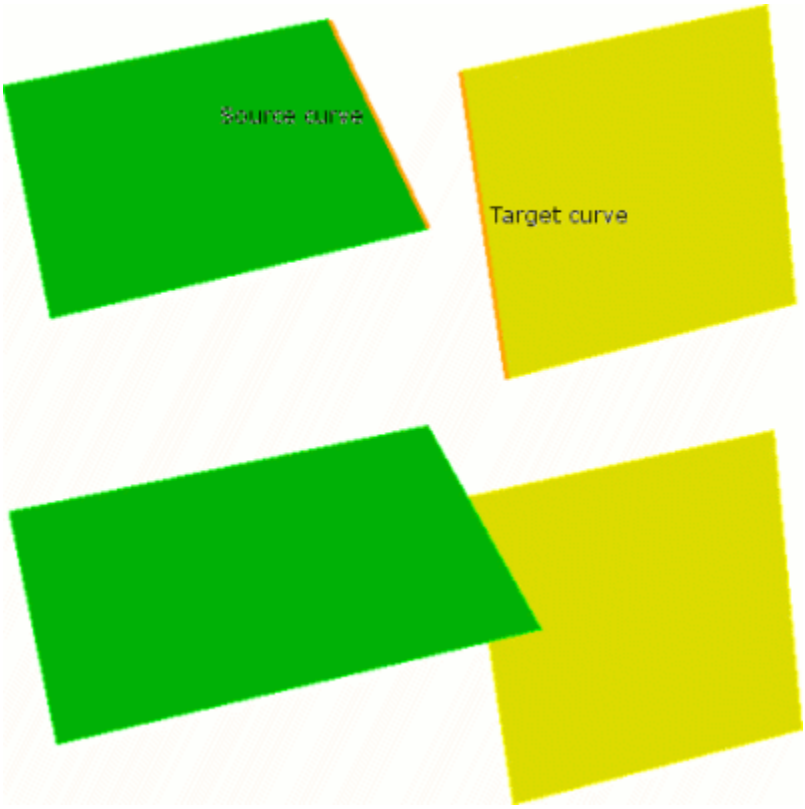


Figure 6 Tweaking a curve to a target curve

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.

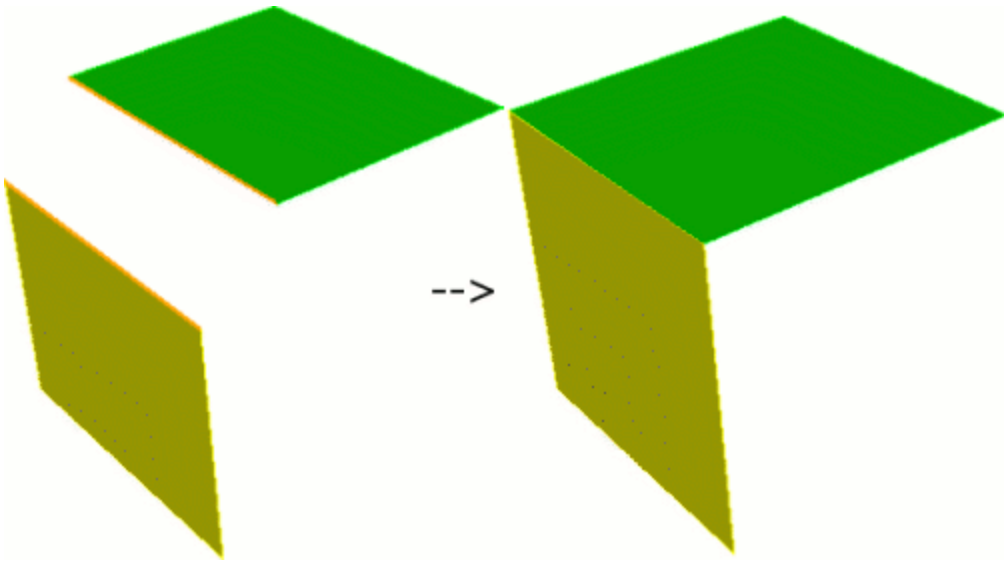


Figure 7. Tweaking two curves to a corner

The preview option temporarily displays the new geometry configuration without actually changing the geometry.

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

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.

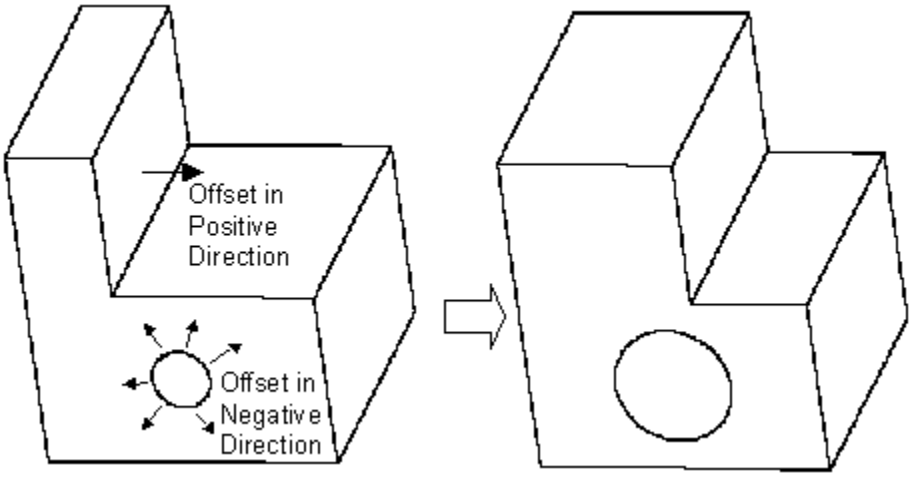


Figure 1. Tweak Offset

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.

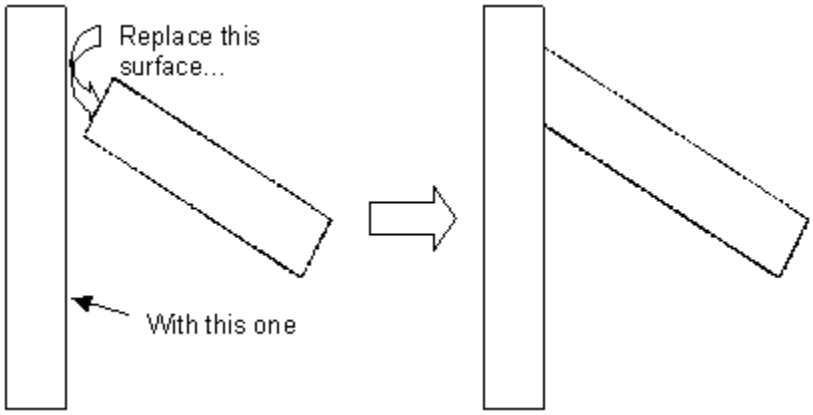


Figure 2. Tweak Surface Target (Viewed directly from the side)

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 [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.

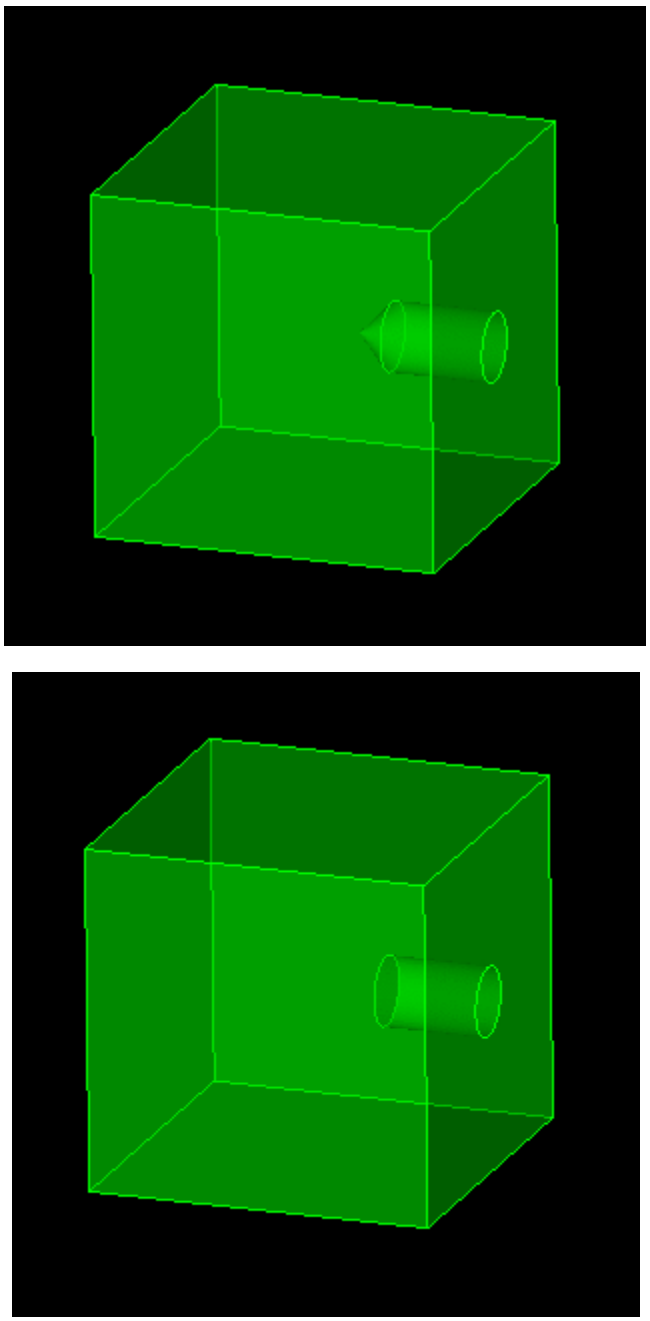


Figure 3. Conical bolt hole before and after tweaking

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]

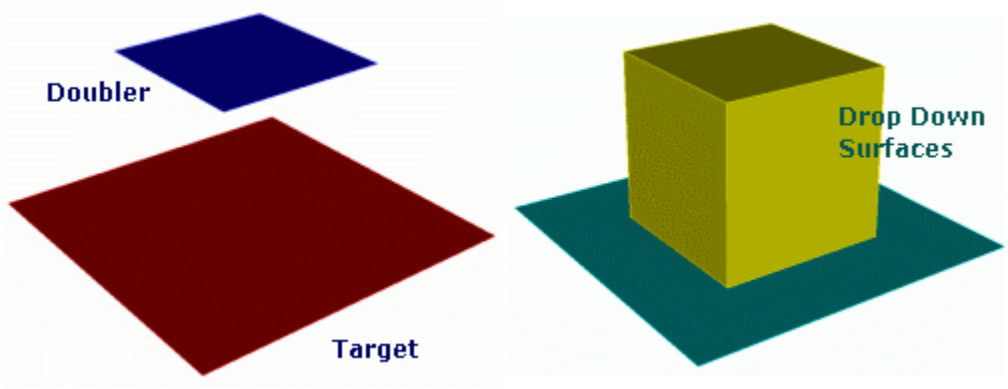
The **plane** option allows a plane to be specified instead of target surface(s). If a target surface is supplied, the user can also use a **limit plane** if he wishes. A limit plane is a plane that the tweak will stop at if the tweaked surface does not completely intersect the target surface. The limit 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.

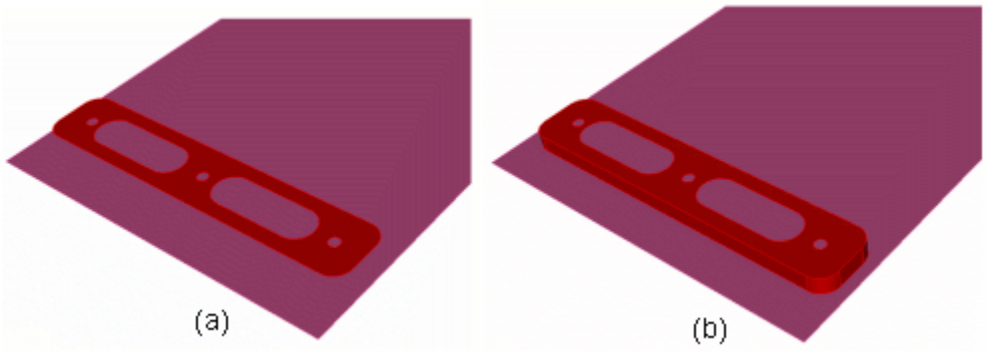
NOTE: This function only works for ACIS geometry.



Geometry **Output**

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).



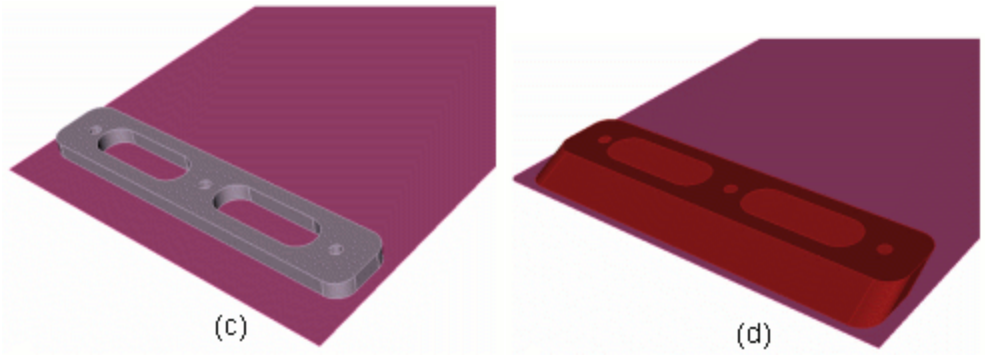


Figure 4. Explanation of tweak doubler options (a) Original surfaces (b) No option flags used (c) Internal option used - notice internal curves dropped down (d) Direction flag - notice skew

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.

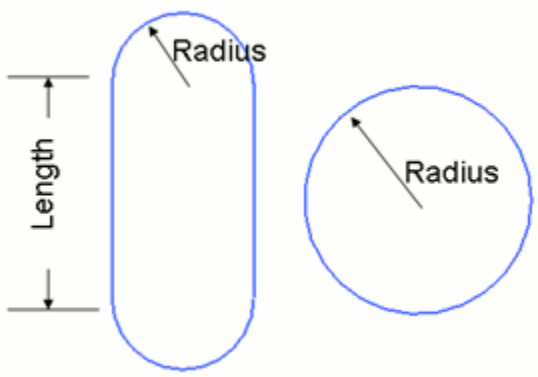


Figure 5. Input parameters for tweak surface idealize command

#Hole Removal Example
tweak surface 13 idealize hole radius 6

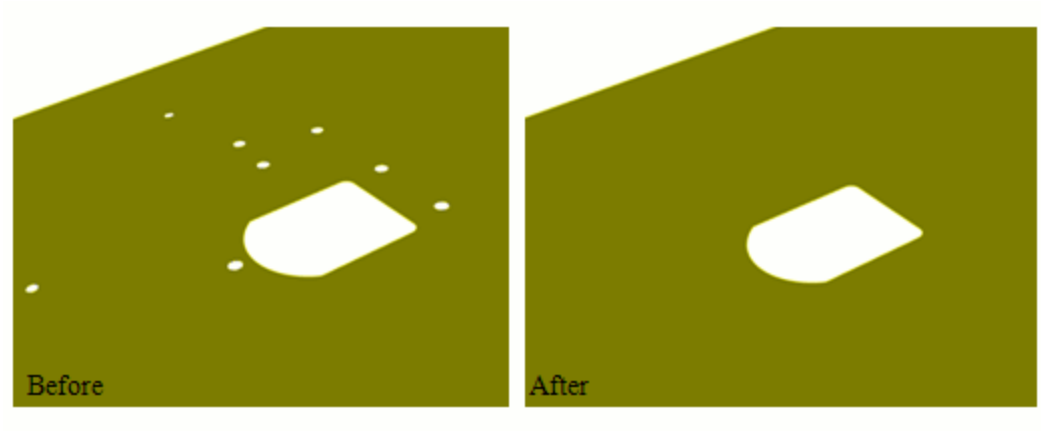


Figure 6. Example of hole removal using tweak surface idealize command

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.

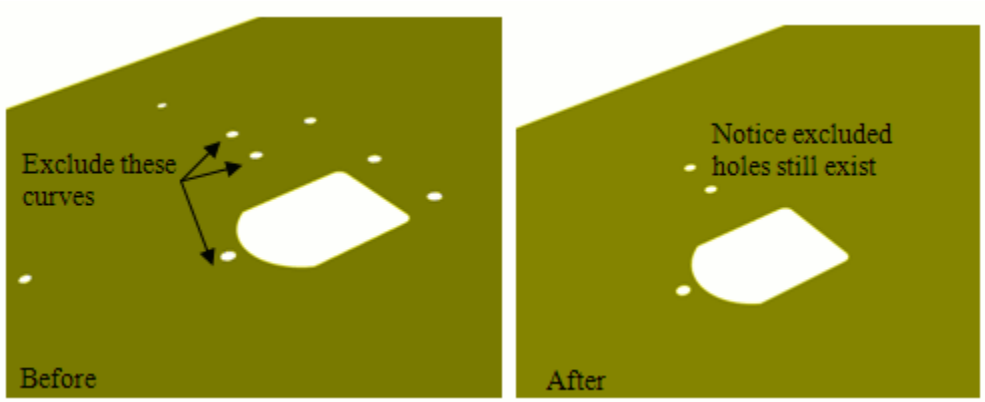


Figure 7. Example of hole removal using exclude option

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

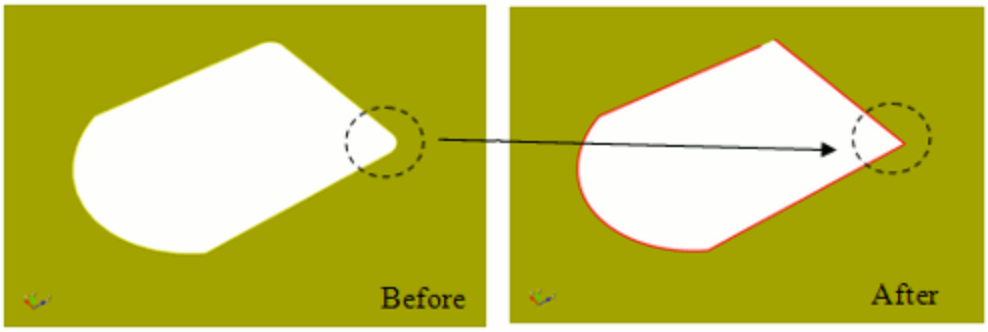


Figure 8. Example of fillet removal using tweak surface idealize command

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.

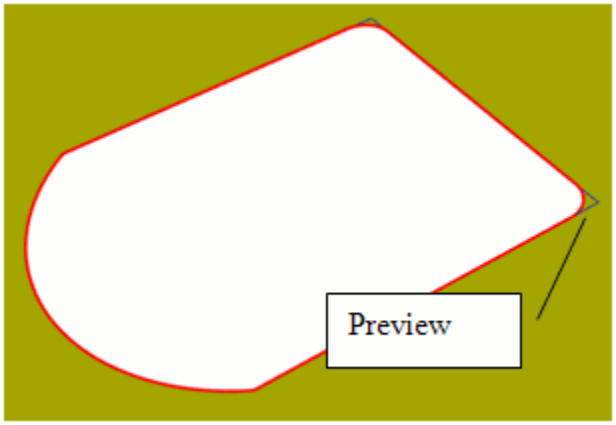


Figure 9. Preview of the tweak surface idealize command

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.

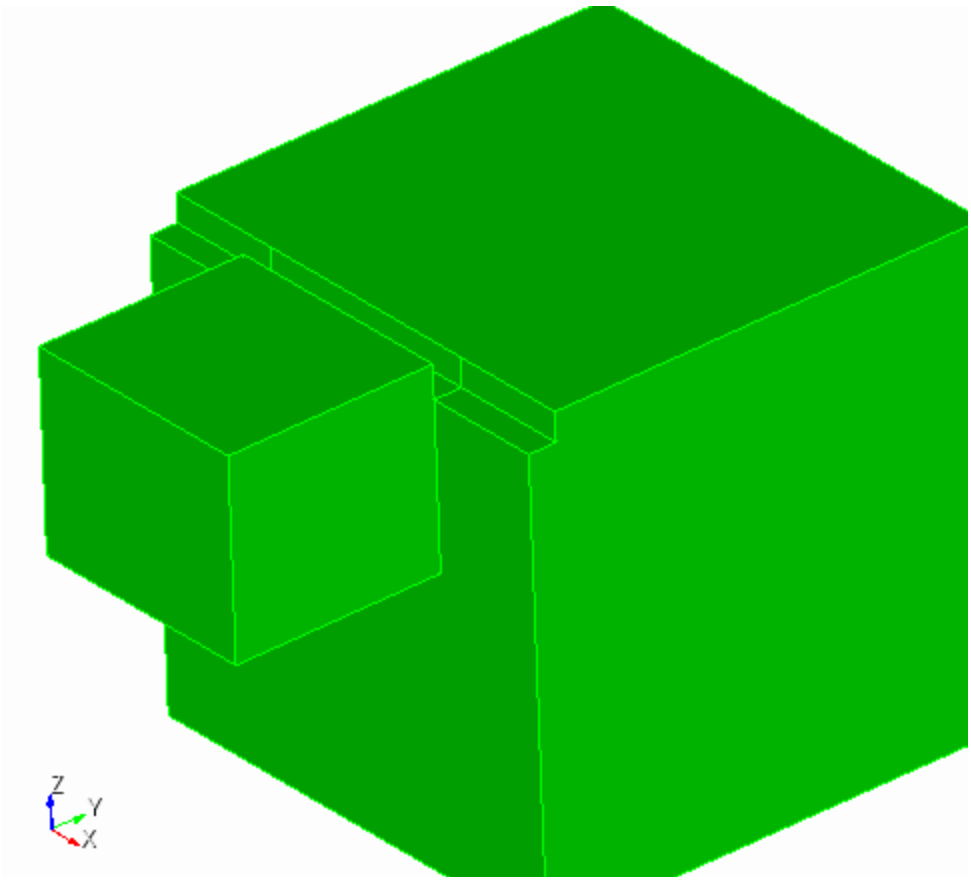
Tweak Remove_Topology {Surface <id_range> | Curve <id_range> | Surface <id_range> Curve <id_range>} Small_curve_size <val> Backoff_distance <val>

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

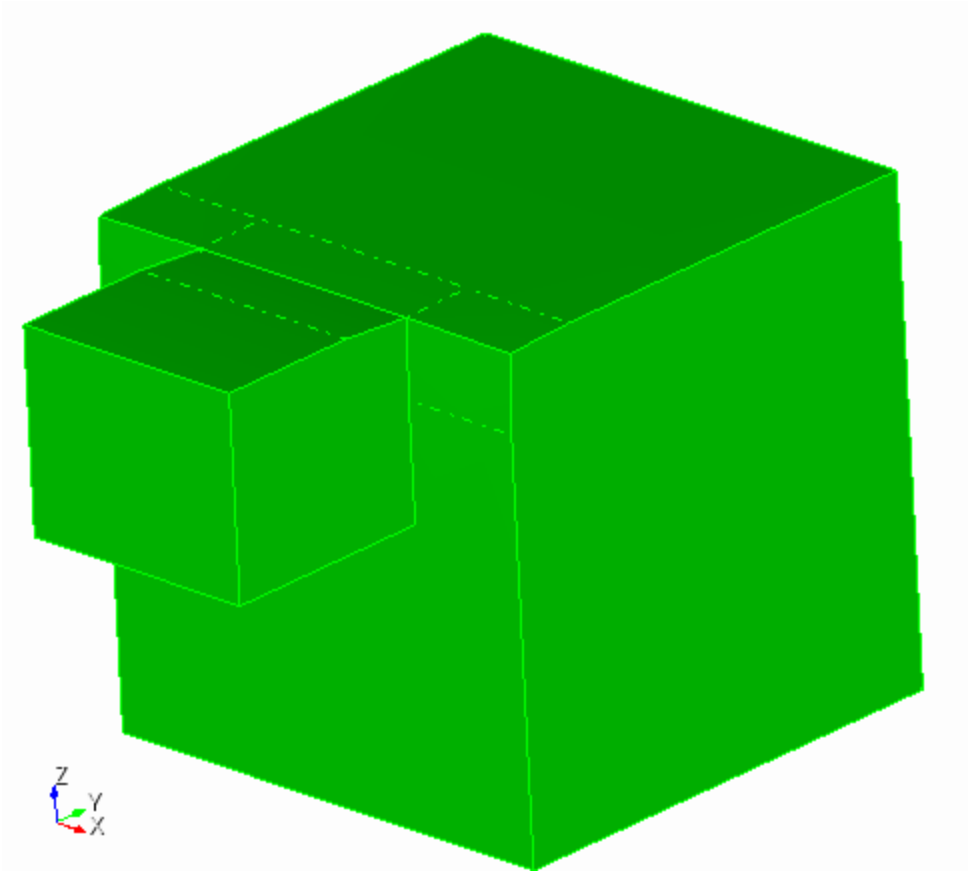


Figure 1. Tweak Remove Topology command

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.

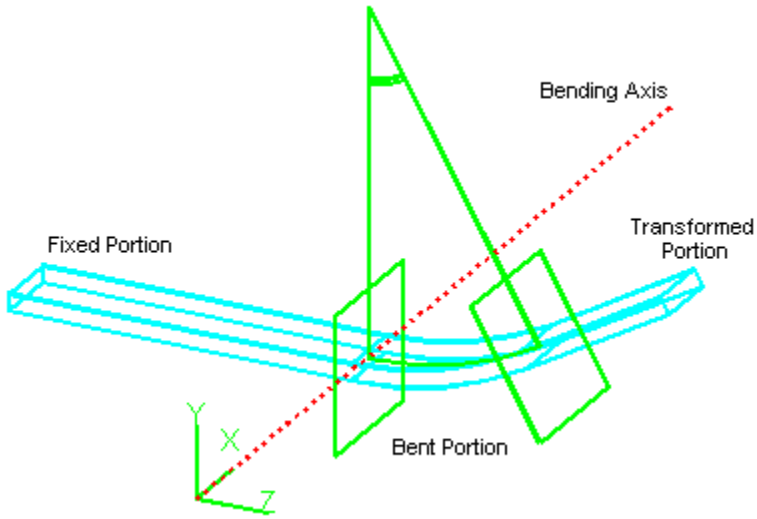


Figure 1. Bending a volume

#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

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 edges. 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 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> [EXTEND|Noextend] [Keepsurface] [Keep] [Individual]

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.

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.

This command is identical to the Tweak Surface Remove command.

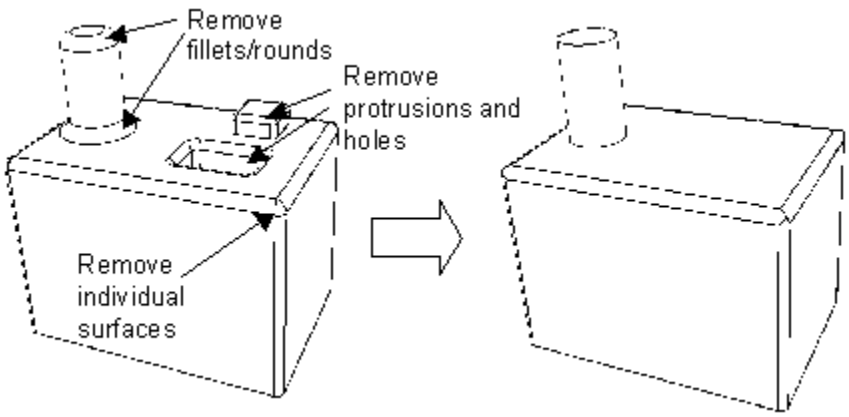


Figure 1. Remove Surface Example

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.

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

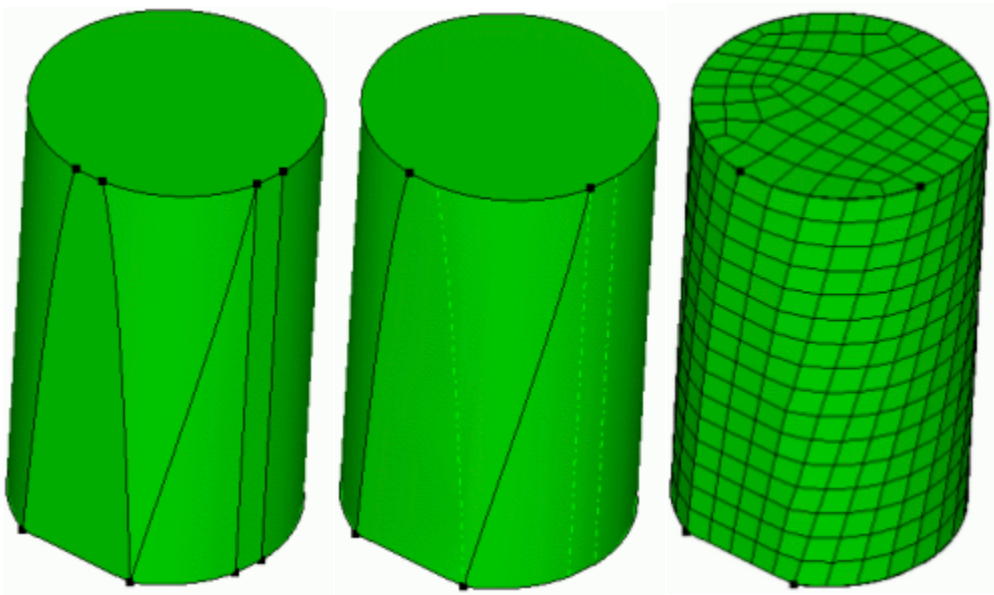


Figure 1. Linking surfaces are composited to force a sweepable volume topology

Automatic 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

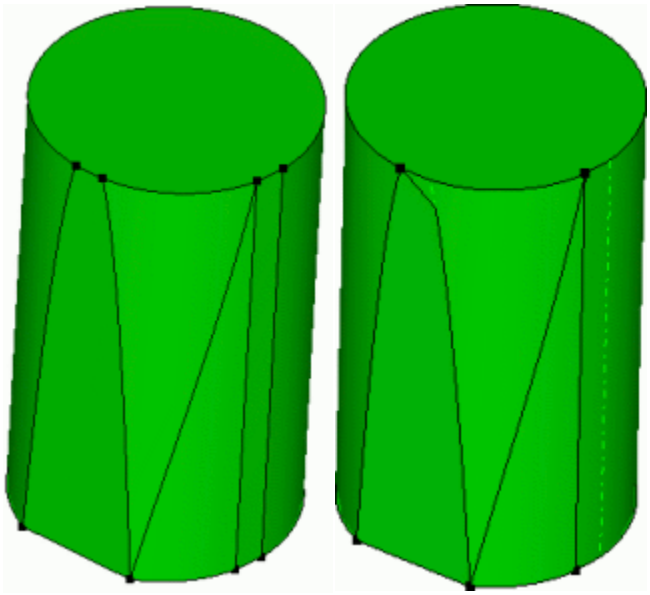


Figure 1. Automatic small curve removal on a cylinder

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.

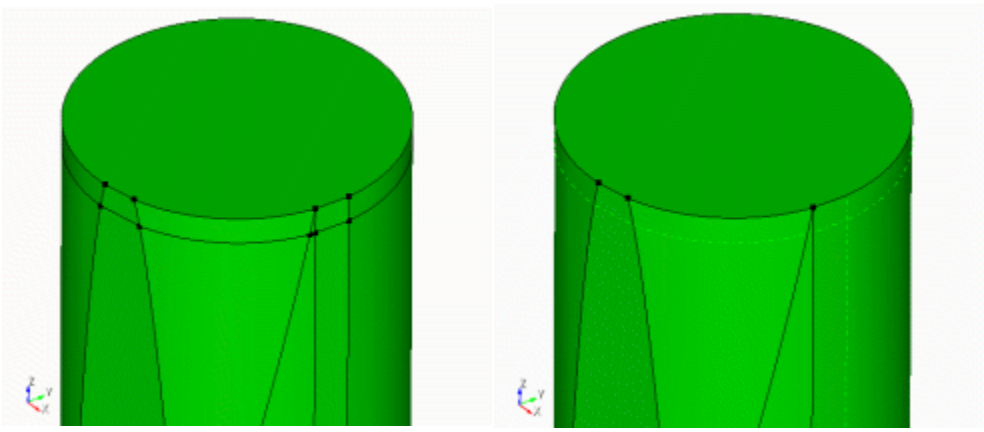


Figure 1. Automatic small and narrow surface removal on a cylinder

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.

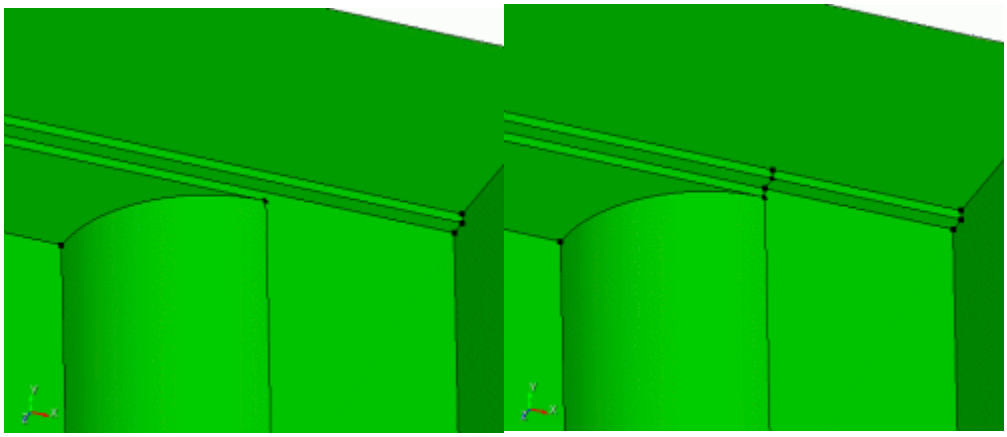


Figure 1. Automatic small and narrow surface removal on a cylinder

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>

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]

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.

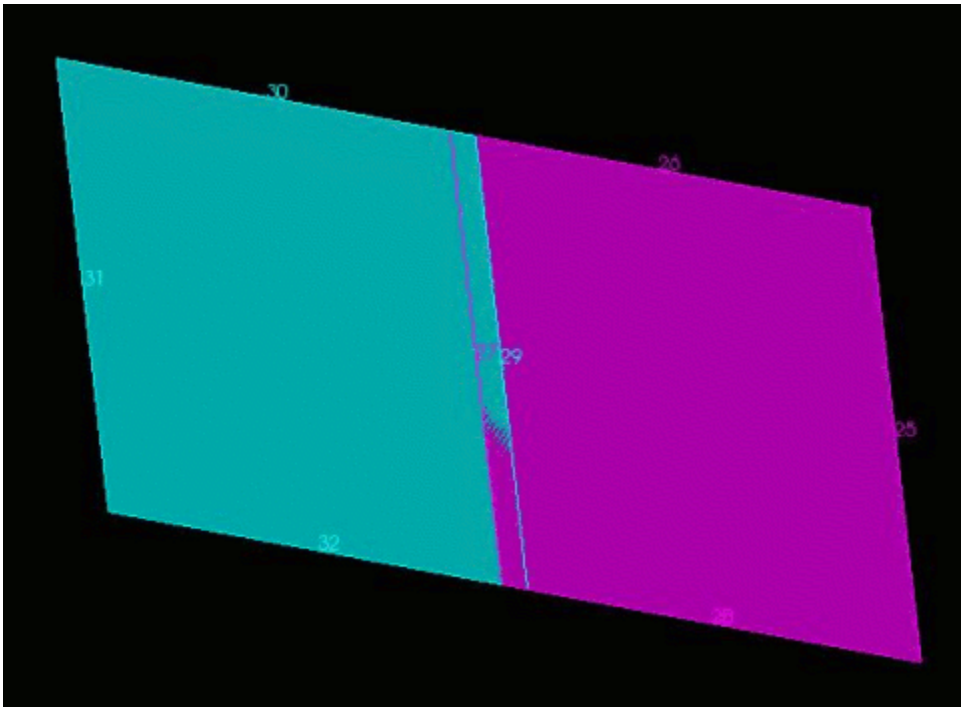


Figure 1. Example of a sliver surface

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>

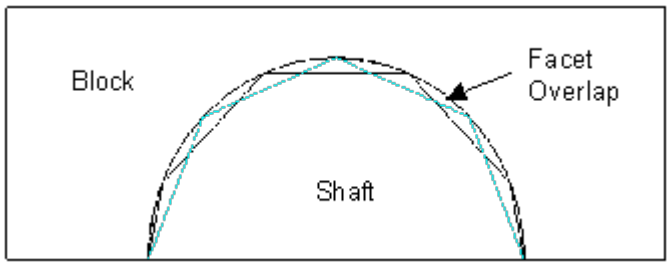


Figure 2. Possible false find due to overlap (tolerance will prevent finding match)

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}

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.

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>

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***.

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

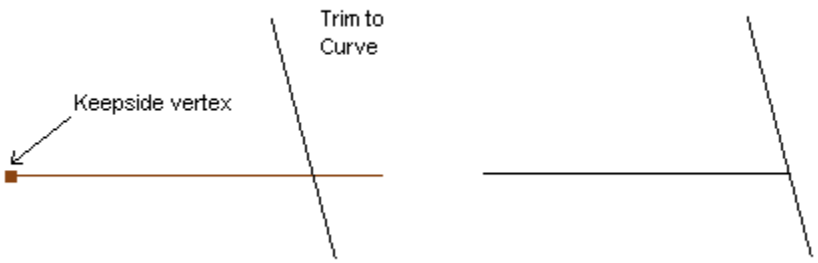


Figure 1. Trimming a Curve to an Intersecting Curve

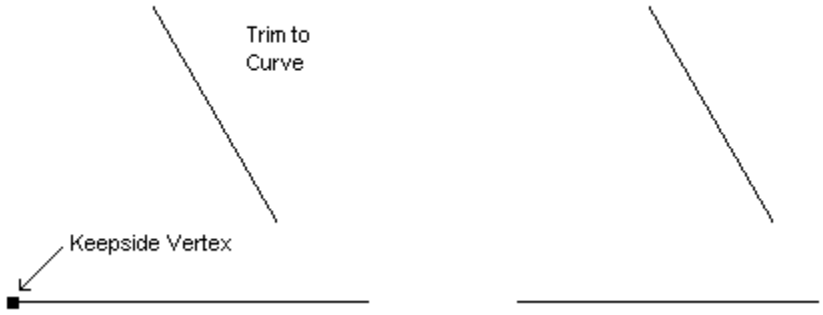


Figure 2. Trimming a Curve to a Non-Intersecting Curve Using the Near Option

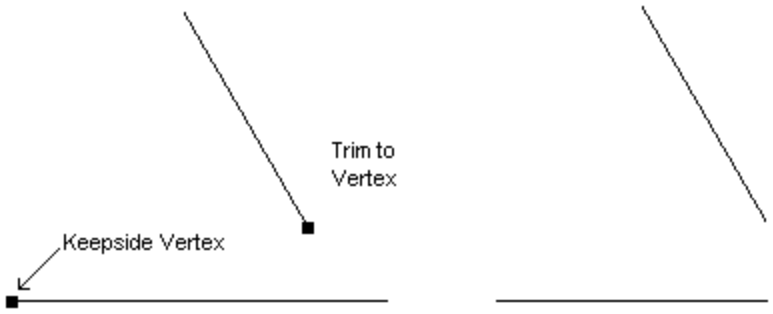


Figure 3. Trimming a Curve to a Vertex

Extending a Curve

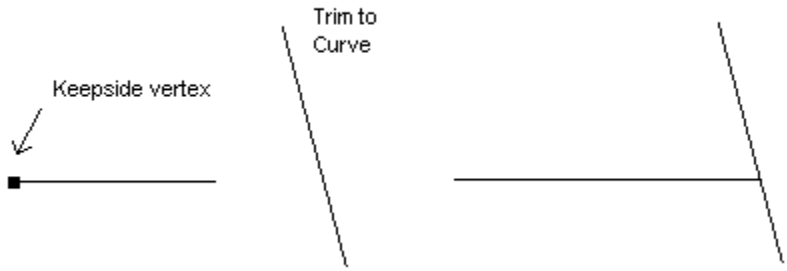


Figure 4. Extending a Curve to An Intersecting Curve

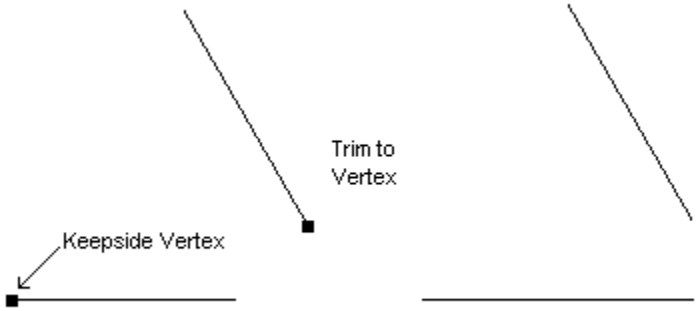


Figure 5. Extending a Curve to a Non-Intersecting Vertex Using the Near Option

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_tighten_gaps]

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.

If the **no_tighten_gaps** option is included, the third step of the stitching process is excluded. This may be necessary in very large or complex models, where the regular approach fails.

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.

Imprint Body <body_id_range> [with] Curve <curve_id_range> [Keep]

Imprint Body <body_id_range> [with] Vertex <vertex_id_range> [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>

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.

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 or unmerged entities of a given type into a specified group, use the command:

Group {<`name'>|<id>} [Surface | Curve | Vertex] [Merged | Unmerged]

If the entity type is unspecified, surfaces will be assumed.

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

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>]

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.

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.

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 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.

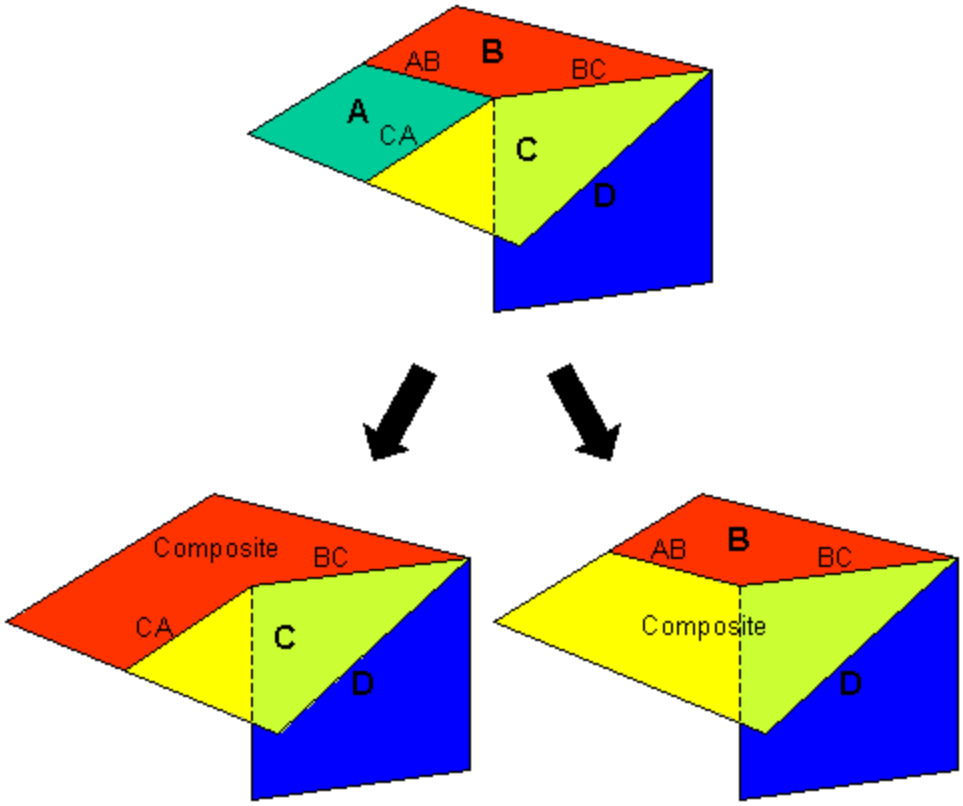


Figure 1. In some cases, the program will make a determination of which surfaces to composite.

Partitioned 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]

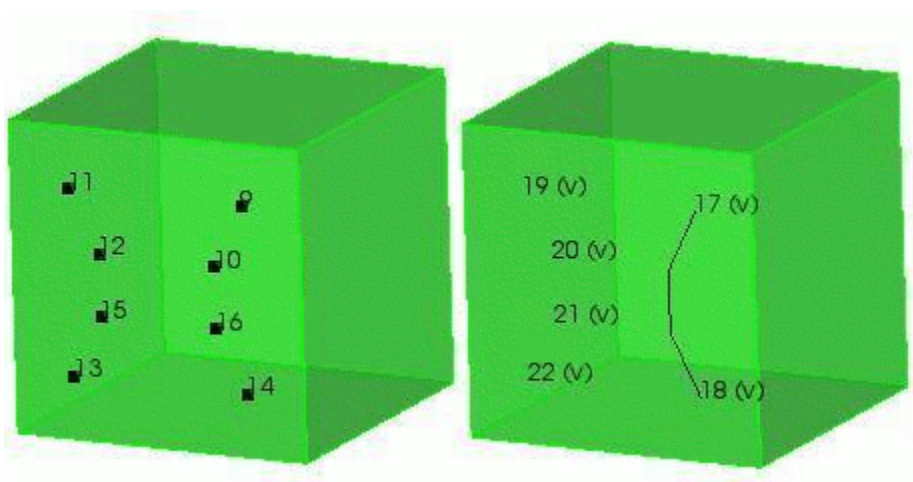
Partitioning with Polylines

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.



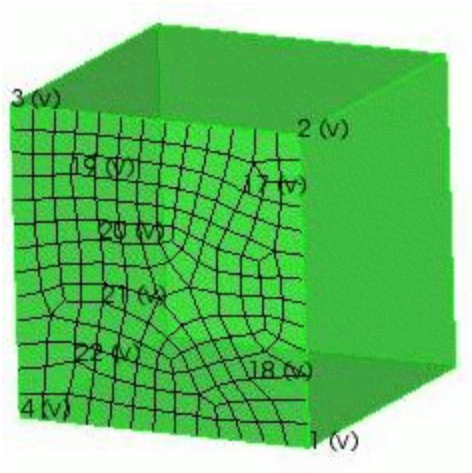


Figure 1. Partitioning a Surface Using Vertices

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.

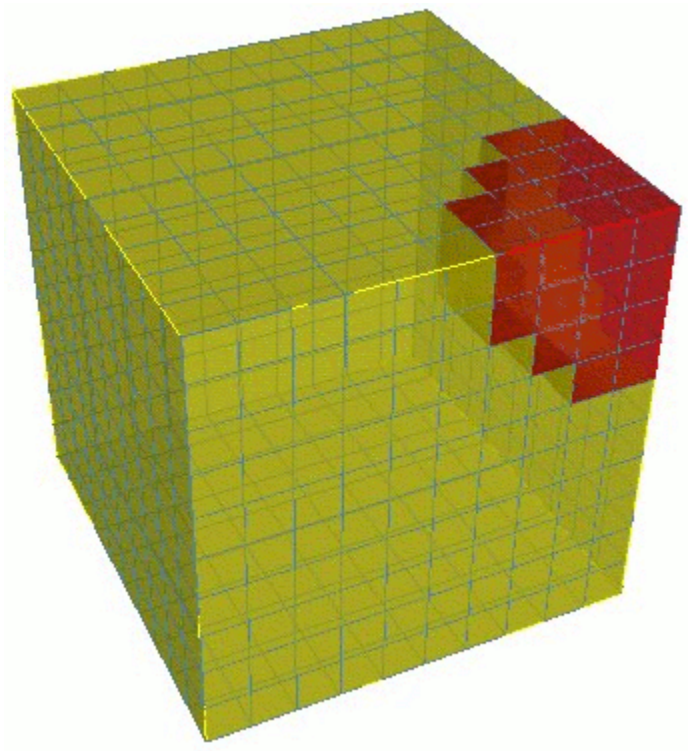


Figure 1. A partitioned volume

This command may be useful for separating small regions of a meshed volume so that remeshing or mesh improvement may be performed locally.

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.

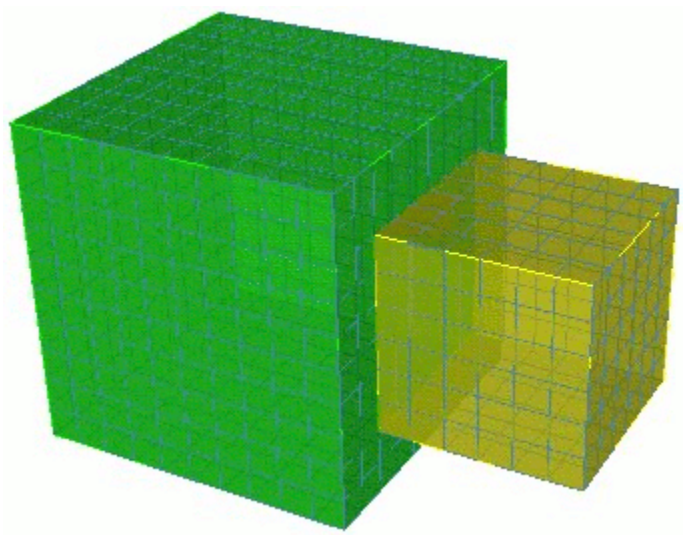


Figure 1. Two adjacent meshed volumes. The coincident meshes will form the basis of the imprint operation.

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.

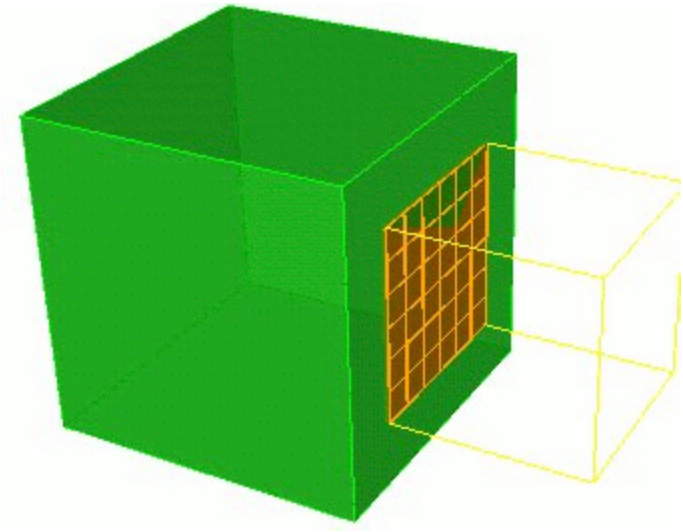


Figure 2. The imprinted surface. Adjacent volume meshes joined at the interface surface.

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.

Collapse Angle

The collapse command allows the user to collapse small angles using virtual geometry. The command syntax is:

> Collapse Angle at Vertex <id> Curve <id1> [Arc_length1 <length>] Curve <id2> [Arc_length2 <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. The options of the command allow you to specify 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.

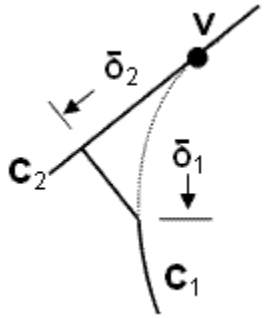


Figure 1. Collapse angle syntax

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.

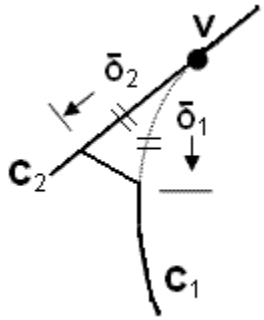


Figure 2. Collapse angle using the same_size option

The **perpendicular** option will split curve 2 so it is perpendicular to the split location on curve 1, as shown in Figure 3.

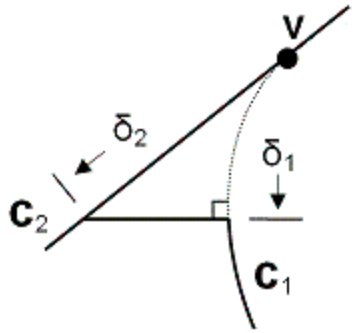


Figure 3. Collapse angle using the perpendicular option

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.

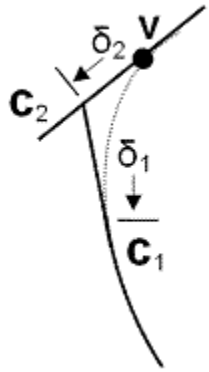


Figure 4. Collapse angle using the tangent option

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.

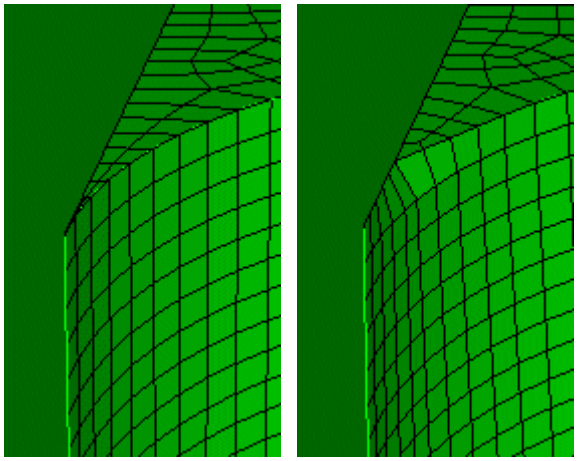


Figure 5. An example of a meshed surface that is generated after using the collapse angle command.

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.

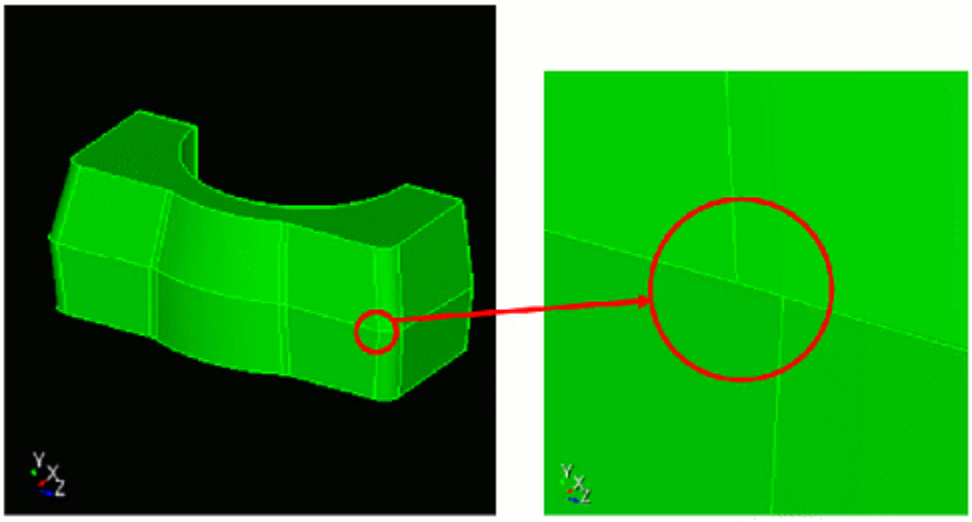


Figure 1. Example where the collapse curve operation is needed.

Figure 2 shows the above example after collapsing the small curve

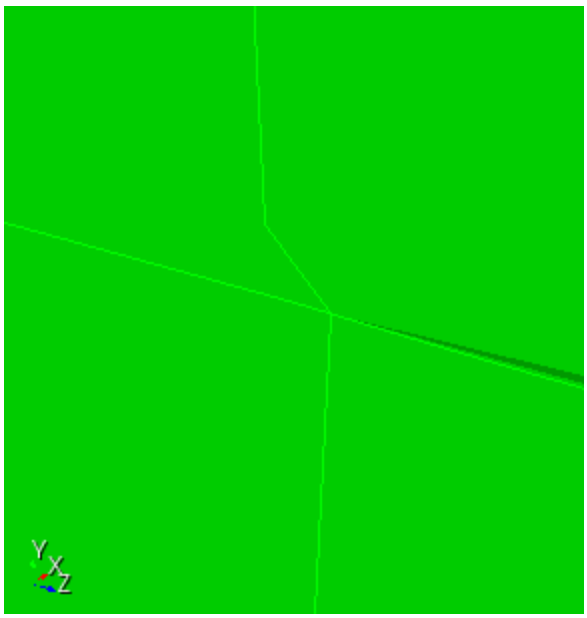


Figure 2. Above example after collapsing the small curve.

Collapse 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.

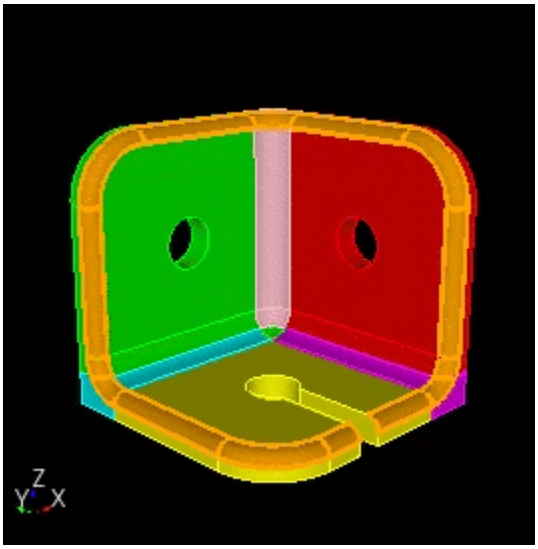


Figure 1. Bracket with chamfered edges.

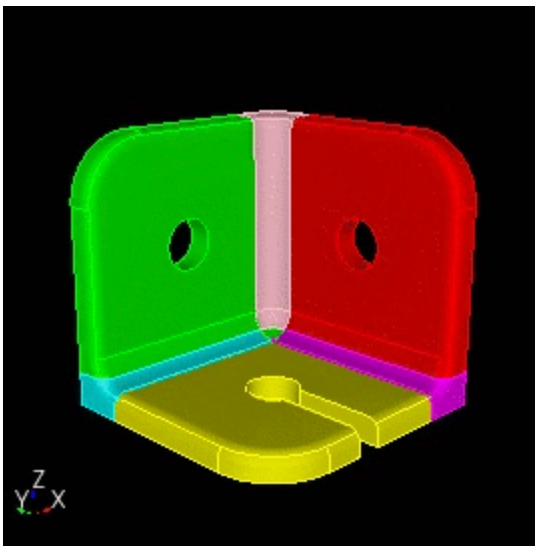


Figure 2. Bracket after highlighted edges have been collapsed

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').

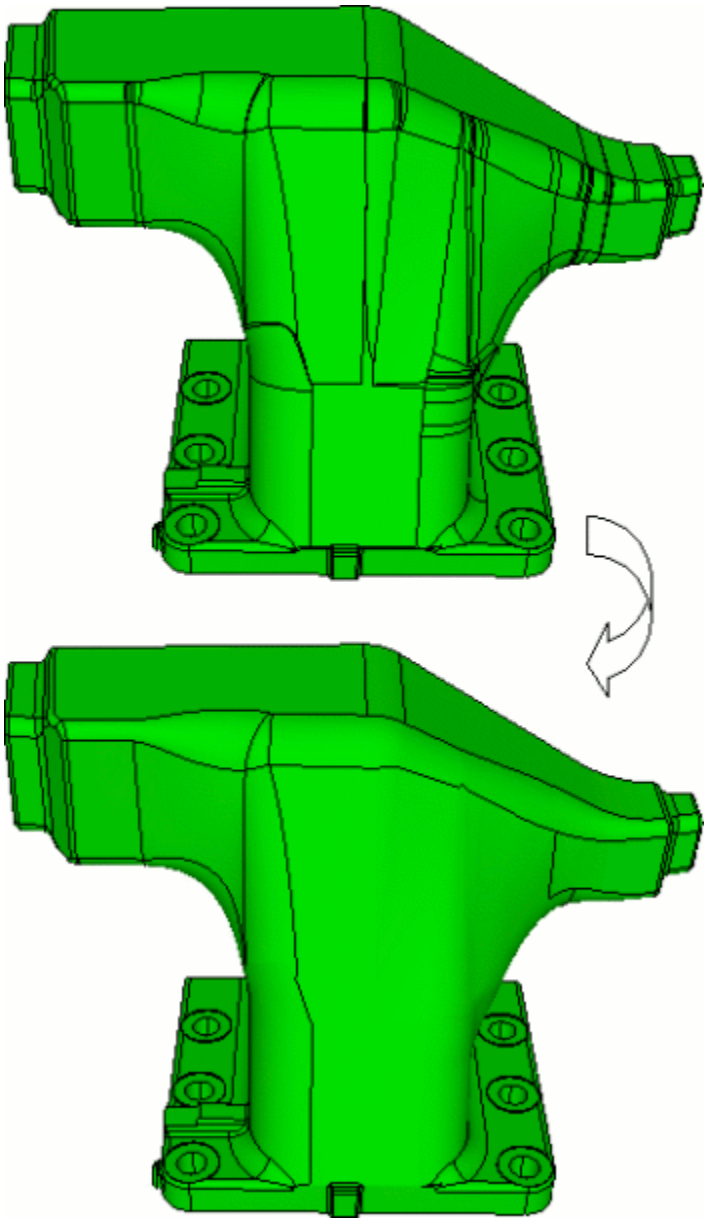


Figure 1. Typical Simplify command usage

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’).

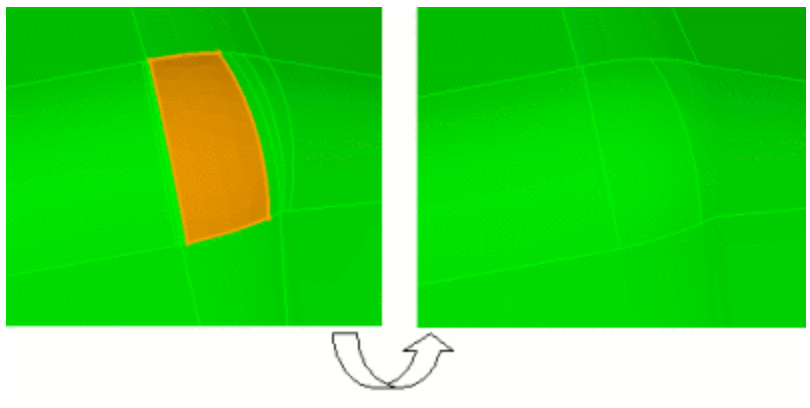


Figure 2 Respecting a surface

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.

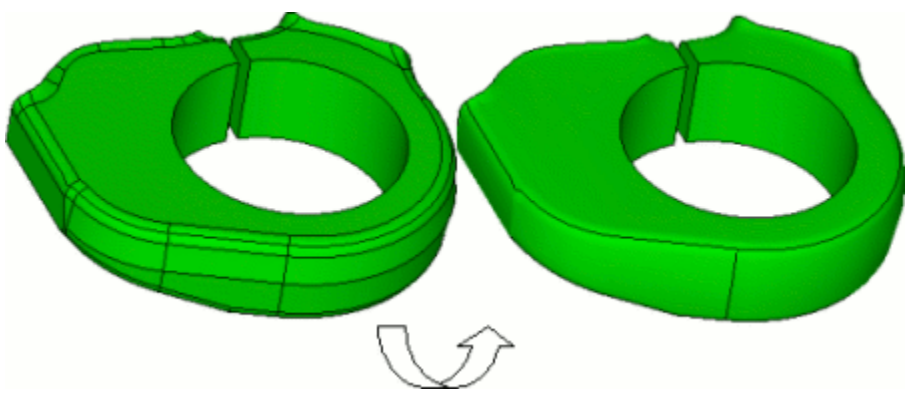


Figure 3 Respecting imprint geometry

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.

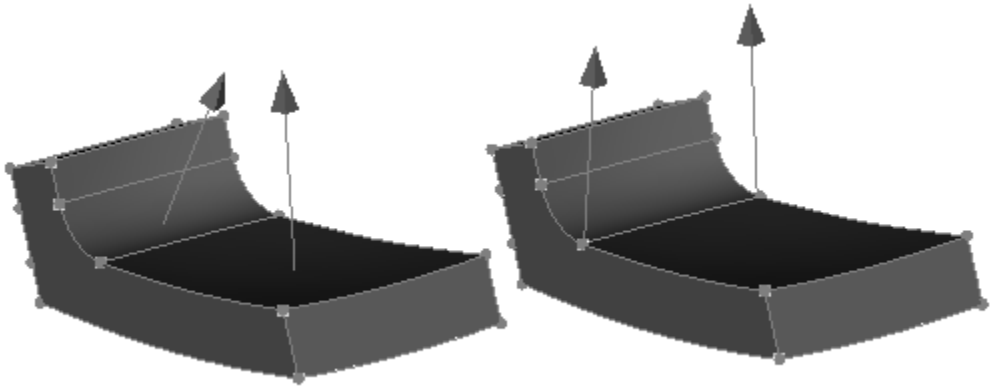


Figure 4. Comparison of surface normals using the average surface normal method (on the left) and local normal method (on the right).

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

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.

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.

Basic Group Operations

Geometry Groups

The command syntax to create or modify a group is:

Group ["name" | <id>] Add <list of topology entities>

For example, the command,

group "exterior" add surface 1 to 2, curve 3 to 5

will create the group named Exterior consisting of the listed topological entities. Any of the commands that can be applied to the "regular" topological entities can also be applied to groups. For example, **mesh Exterior** , **list Exterior** , or **draw Exterior** .

Elements may specified by name as well. For example, the command

group 'interior' add surface name 'bill' 'john' 'fred'

will add the surfaces named 'bill' 'john' and 'fred' to the group 'interior'. A topological entity can be removed from a group using the command:

Group ["name" | <id>] Remove <entity list>

The *Xor* operation can also be performed on entities in group. Xor means if an entity is already in the group, the command will delete this entity from the group. If it is not in the group, the entity is then added to the group.

Group ["name" | <id>] Xor <entity list>

The *Equals* operation assigns the group to be exactly the same as the list given. All other existing members of the group will be removed.

Group ["name" | <id>] Equals <entity list>

Modifying groups by comparing common entities

The *Common_To* operation looks for geometry entities that are related to the input elements, either as parents or children. For example, specifying all curves common to two surfaces will give all of the curves that are attached to both of the specified surfaces. The elements must be specified by name (specifying by id will not work), and the name must be enclosed in single quotation marks. This option works for all of the group operators given above. The command syntax is:

Group ["name" | <id>] {Add|Equals|Remove|Xor} <entity_type> Common_to <entity_type> Name 'pattern' ['pattern'...]

The following is an example of the common_to operator.

bri x 10
curve 2 name 'joe'
curve 3 name 'alf'
group 'mygroup' add surf common_to curve name 'joe' 'alf'

Group Booleans

Groups may also be created from existing groups by using boolean operations. Each of these commands will create a new group that contains entities from two existing groups. The **intersect** command will create a new group that contains elements common to both existing groups. The **unite** command will contain entities that exist in either group. The **subtract** command will remove entities that are common to both groups and create a new group from entities that exist in exactly one of the groups.

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>

Mesh Groups

Groups may also contain mesh entities. The commands for adding and removing mesh entities are analogous to those for geometric entities.

Group ["name" | <id>] Add {Hex|Face|Edge|Node <id_list>}

Group ["name" | <id>] Remove {Hex|Face|Edge|Node <id_list>}

Group ["name" | <id>] Xor {Hex|Face|Edge|Node <id_list>}

Group Copy

Groups may be copied as groups using the group transform commands. Child entities cannot be moved using this command. If a child entity is in the group, its parent entity must be specified as well. In addition, all merge partners must be specified. Only groups containing geometric entities can be copied with these commands. If a geometry entity is meshed, the mesh will be copied as well, unless the [nomesh] option is given. Copied entities can be moved, rotated, reflected, or scaled as well.

Group {<'name'>|<id>} Copy [Move <x> <y> <z>] [nomesh]

Group {<'name'>|<id>} Copy [Move {x|y|z} <distance>...] [nomesh]

Group {<'name'>|<id>} Copy [Move <direction> [distance]] [nomesh]

Group {<'name'>|<id>} Copy [Reflect {x|y|z}] [nomesh]

Group {<'name'>|<id>} Copy [Reflect <x> <y> <z>] [nomesh]

Group {<'name'>|<id>} Copy [Rotate <angle> About {x|y|z}] [nomesh]

Group {<'name'>|<id>} Copy [Rotate <angle> About <x> <y> <z>] [nomesh]

Group {<'name'>|<id>} Copy [Scale <scale> | x <val> y <val> z <val>] [nomesh]

Group Transformations

Groups may be transformed as groups using the group transform commands. This is especially helpful for transforming groups of free mesh elements, where no geometry exists. The command syntax is shown below.

Group {<'name'>|<id>} [Copy [nomesh]] Move <dx> <dy> <dz>

Group {<'name'>|<id>} [Copy [nomesh]] Move {x|y|z} <distance>...

Group {<'name'>|<id>} [Copy [nomesh]] Reflect {x|y|z}

Group {<'name'>|<id>} [Copy [nomesh]] Reflect <x> <y> <z>

Group {<'name'>|<id>} [Copy [nomesh]] Reflect {x|y|z}

Group {<'name'>|<id>} [Copy [nomesh]] Reflect <x> <y> <z>

Group {<'name'>|<id>} [Copy [nomesh]] Rotate <angle> About {x|y|z}

Group {<'name'>|<id>} [Copy [nomesh]] Rotate <angle> About <x> <y> <z>

Group {<'name'>|<id>} [Copy [nomesh]] Rotate <angle> About Vertex <Vertex-1> [Vertex] <Vertex-2>

Group {<'name'>|<id>} [Copy [nomesh]] Scale <scale> | x <val> y <val> z <val>

The nomesh option applies to the copy part of the command. If the no_mesh option is specify, the mesh will not be copied.

Deleting Groups

Groups can be deleted with the following command:

Delete Group <id range> [Propagate]

The option **propagate** will delete the group specified and all of its contained groups recursively.

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

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

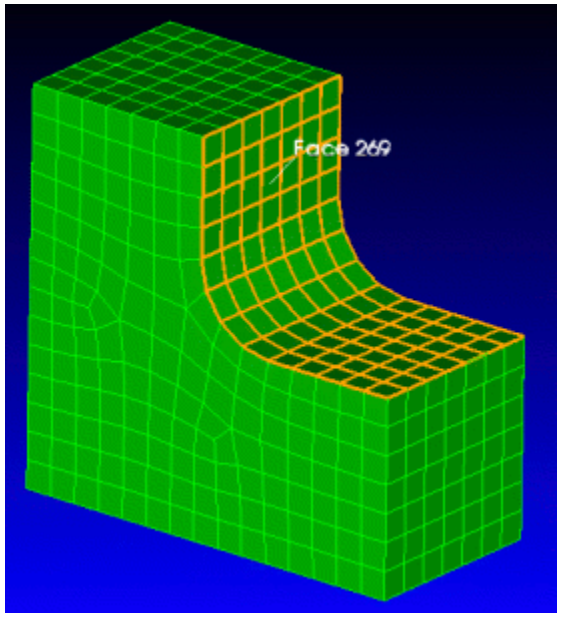
Seeded Mesh Groups

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:

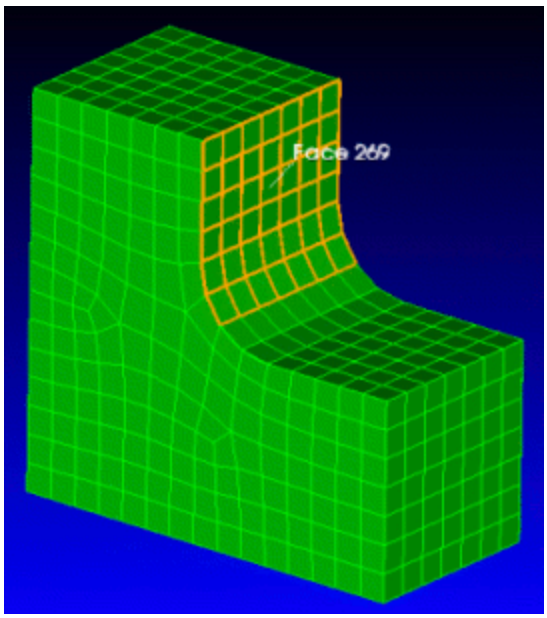
> Group {<'name'>|<id>} {Add|Equals|Remove|Xor} Seed <mesh_entities> {Feature_angle <angle> [Divergence]|Depth <number>}

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

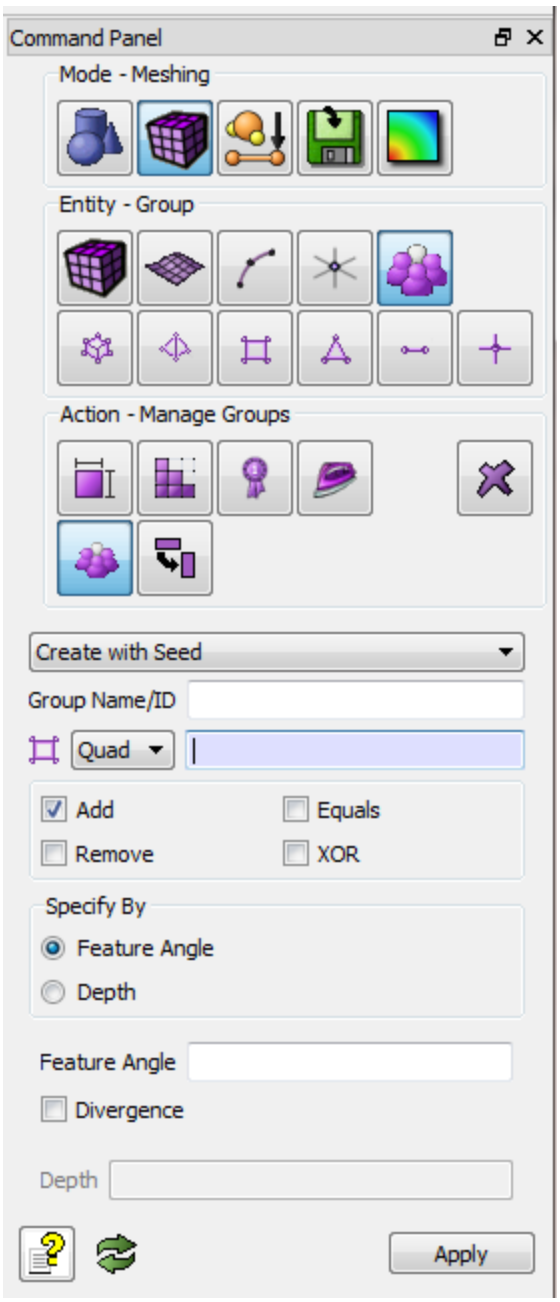


CUBIT> group 'mygroup2' add seed face 269 feature_angle 45 divergence

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:


Command Panel
Mode - Meshing
Entity - Group
Action - Manage Groups
Create with Seed
Group Name/ID
Quad
Add
Equals
Remove
XOR
Specify By
Feature Angle
Depth
Feature Angle
Divergence
Depth
Apply

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.

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.

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

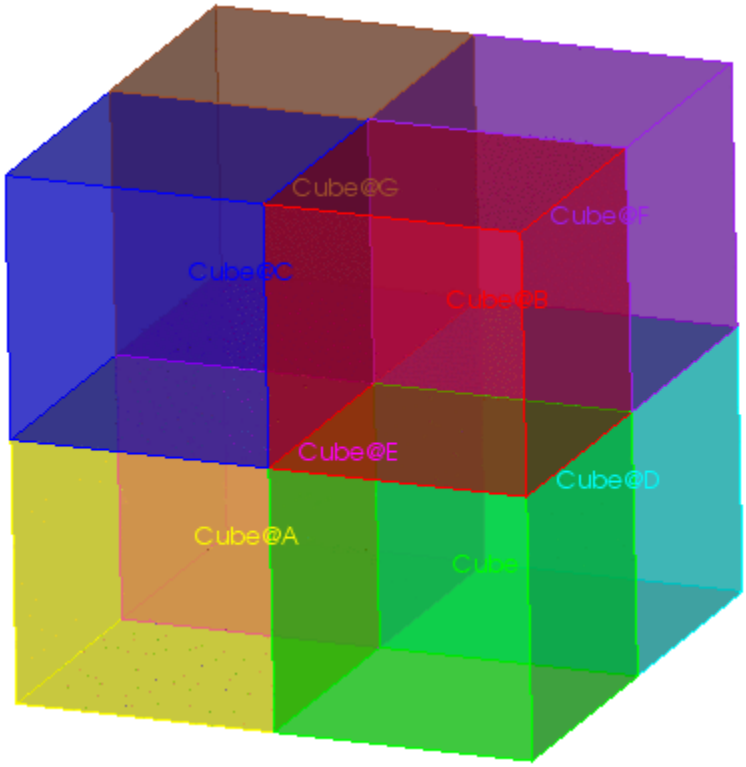


Figure 1. Name Propagation through Webcuts

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.

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** command can be used to eliminate these gaps:

> Compress [Ids] [All] [Group|Body|Volume|Surface|Curve|Vertex|Element|Hex|Tet|Face|Edge|Node] [Retainmax] [Sort]

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, y, and z coordinates leading to a lower id. 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.

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 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

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.

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 Measurement

To output various properties of entities, the following **Measure** command options are available.

- Measure Between
- Measure Small
- Measure Angle
- Measure Void

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).

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_check** 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.

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

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|Assembly} "<path>"

This will remove the specified part or assembly. Assemblies can only be removed if they have no contents. All contained parts and subassemblies must be removed before removing the parent assembly.

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.

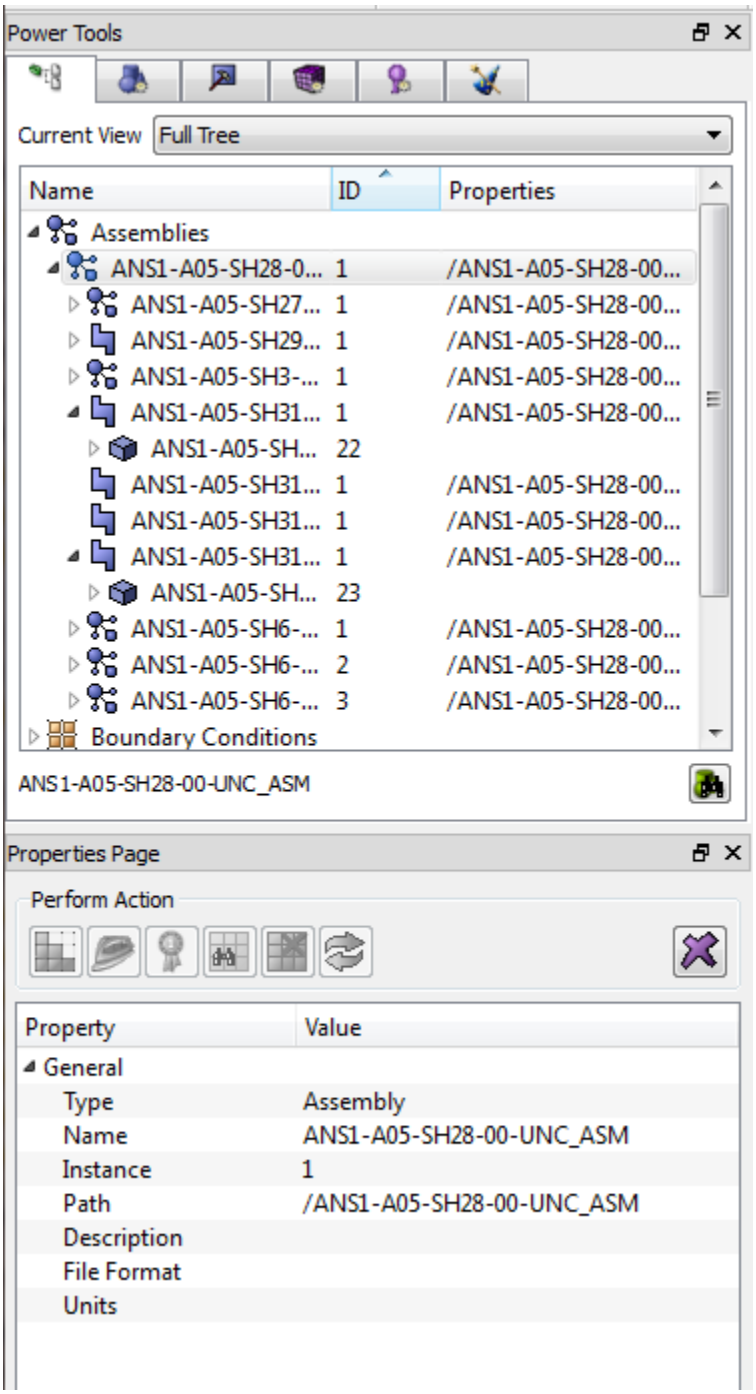
The **Export Part_List** command is particularly useful for identifying parts which are not correctly associated with parts. Among the fields that can be filtered is the **is-part** field. This field is FALSE for each volume that is not associated with a part. Filtering on this value will show a list 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.



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 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.	**x**	
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: • Secret • Confidential • Unclassified
Classification_Category	The classification category. Usually one of the following: • Not Restricted • Restricted Data (RD) • Formerly Restricted Data (FRD) • 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.

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 http://www-irn.sandia.gov/schema/dart/3.0/DARTMetadata.xsd, 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 an add-in called eMatrix (a tool under the umbrella of the DART project, see the Analyst Home 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.

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. 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.

See also Importing ACIS Models.

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.

See Importing STEP Files for information on setting up the STEP import and export functionality.

Note that the STEP import and export functionality might not be available on all 64-bit platforms.

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 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.

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.

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 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 an interval firmness of default , and 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.

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.

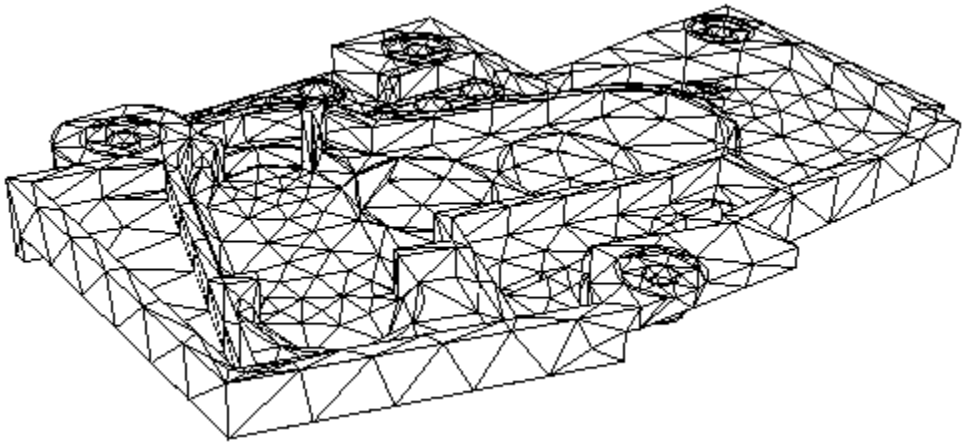
{Curve|Surface|Volume|Body|Group} <range> Interval Factor <factor>

Automatic Specification of Interval Size

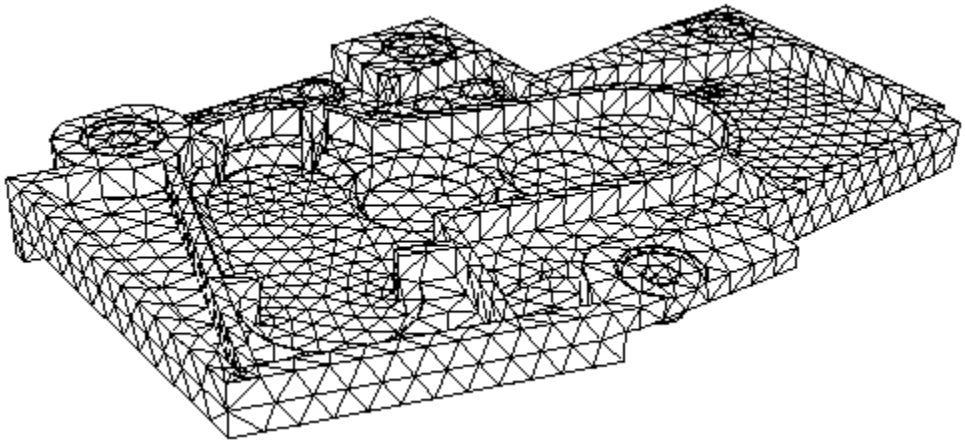
In addition to specifying intervals explicitly based on a known count or size, CUBIT is able to compute interval sizes automatically based on characteristics of the model geometry. The following automatic interval size setting command can be used:

{geom_list} Size Auto [Factor <factor>] [Individual] [Propagate]

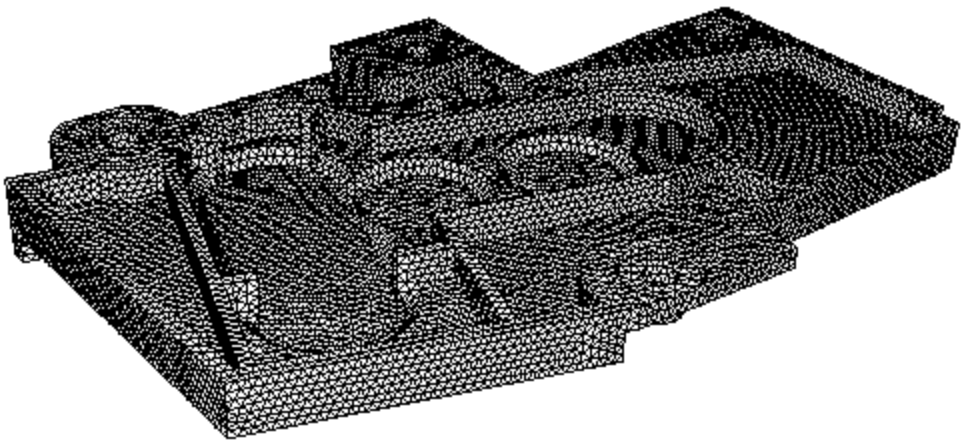
Vertices are not valid in the **geom_list** for this command. Automatic interval size assignment works by examining the geometric characteristics of the entities in the **geom_list** and assigning a heuristic size to the entities and their child entities. The factor may be a floating point number between 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.

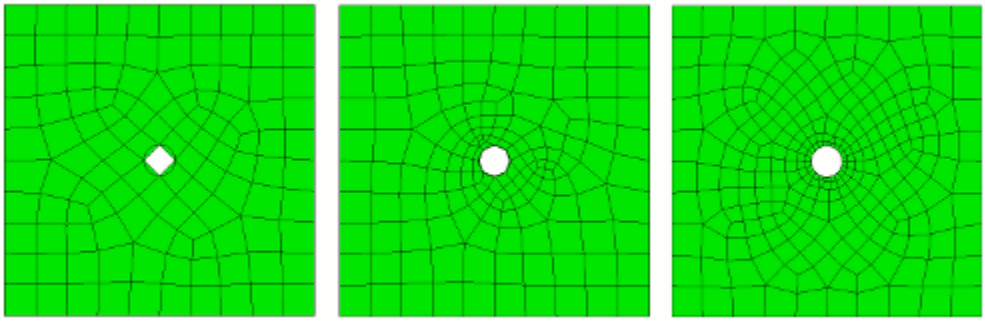


Figure 2. Maximum arc_span settings of 90, 45 and 15 degrees respectively.

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 a set of constraints on the intervals assigned to the curves bounding the entity being meshed. For example, meshing any surface with quadrilaterals requires that the surface be bounded by an even number of mesh edges. This constrains the intervals on the bounding curves to sum to an even number. For a collection of connected surfaces and volumes, these interval constraints must be resolved globally to ensure that each surface will be meshable with the assigned scheme. The global solution technique implemented in CUBIT is referred to as interval matching.

When meshing a surface or volume, matching intervals is performed automatically. In some cases, interval matching needs to be invoked manually, for example when meshing a collection of volumes, or a collection of surfaces not in a common volume. Interval matching can also be called to check whether the assigned intervals and schemes 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.

The interval matcher assigns intervals as close as possible to the user-specified intervals, while satisfying global interval constraints. The goal is to minimize the relative change in pre-assigned intervals on all entities. Interval matching only changes curves with interval firmness of soft or default .

Extra constraints can be added by the user to improve mesh quality locally; in particular, curves can be constrained to have the same intervals using the command

Curve <range> Interval {Same|Different}

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."

The interval assignment algorithm tries to find one good interval solution from among the possibly infinite set of solutions. However, if many curves are hard-set or already meshed, there may be no solution. To improve the chances of finding a solution, it is suggested that curves are soft-set whenever possible. Also, a solution might not exist due to the way the local selections of corners and sides of mapped surfaces interact globally. If there is no solution, the following command may help in determining the cause:

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.

Advanced users may also wish to experiment with setting the following, which may change the interval solution slightly:

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 iteration. Note in rare cases this may produce better meshes. If set match intervals fast is set to **on** multiple curves will be fixed per iteration.

Set match intervals delta allows the number of intervals assigned to a curve to be delta intervals away from optimal unexpectedly. A larger value makes matching intervals faster, but the quality of the solution may be worse; Hint: try delta = 1.0. Default is 0.0.

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.

Rather than specifying a specific size or interval for a curve or surface, which may overconstrain the interval matcher, you can specify an upper and lower bound that is acceptable. This would typically be used in a complex assembly where there may be multiple intervals that may interact in order to get a compatible mapped/swept mesh through the assembly.

Surface <surface_id_range> {Interval|Size|Periodic Interval} {Lower|Upper} Bound {On|Off|<bound>}

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.

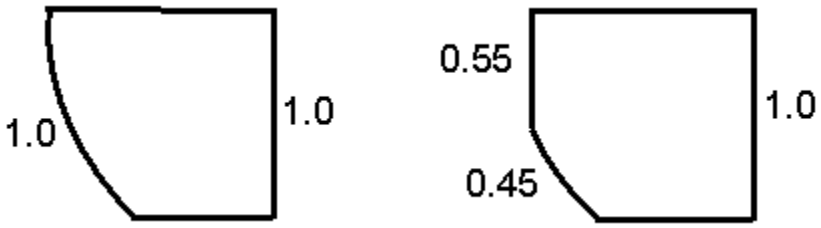
However, periodic intervals and firmness can be specified explicitly by the following commands:

Surface <range> Periodic Interval <intervals>

Surface <range> Periodic Interval {Default|Soft|Hard}

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]

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.

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]

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.

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.

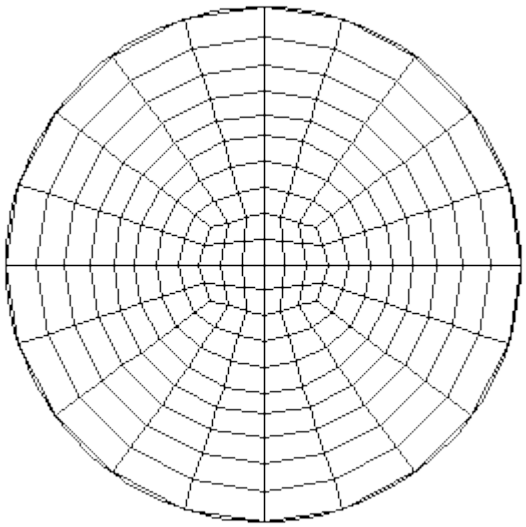


Figure 1. Circle Primitive Mesh

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.

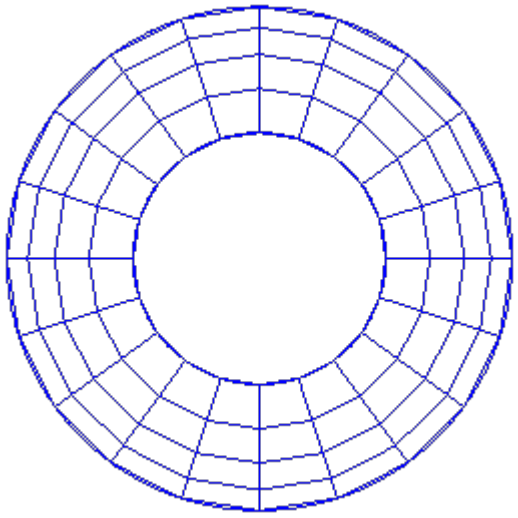


Figure 1. Example of Hole Scheme

Mapping

Applies to: Surfaces, Volumes

Summary: Meshes a surface/volume with a structured mesh of quadrilaterals/hexahedra.

Syntax:

{Volume|Surface} <range> Scheme Map

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.

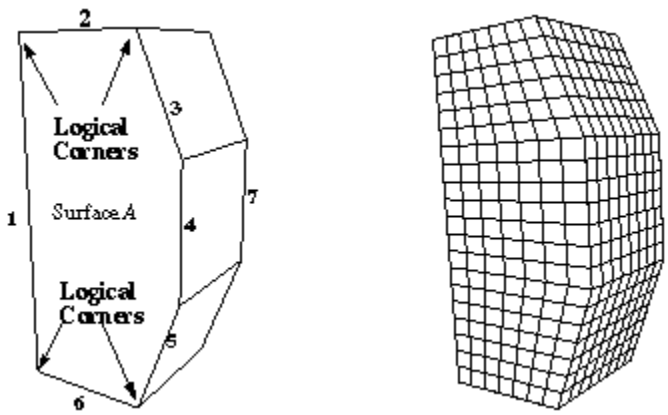


Figure 1. Scheme Map Logical Properties

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.

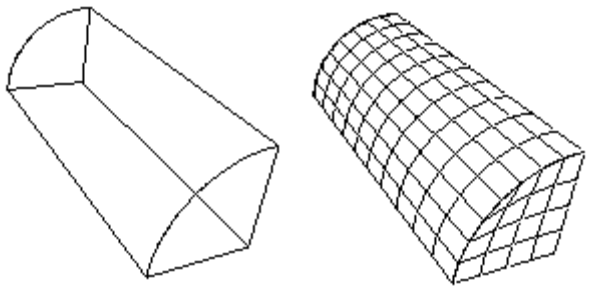


Figure 2. Volume Mapping of a 5-surfaced volume

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}

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.

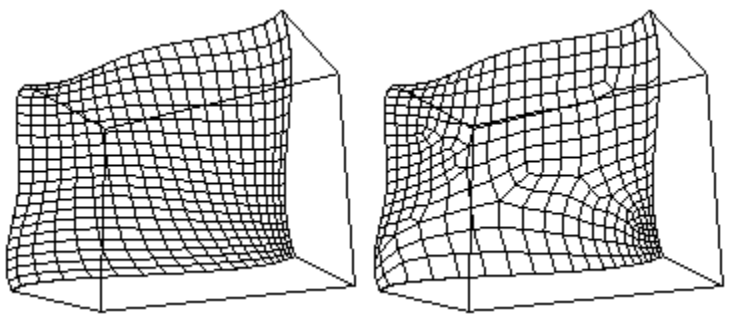


Figure 1. Map (left) and Paved (right) Surface Meshes

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.

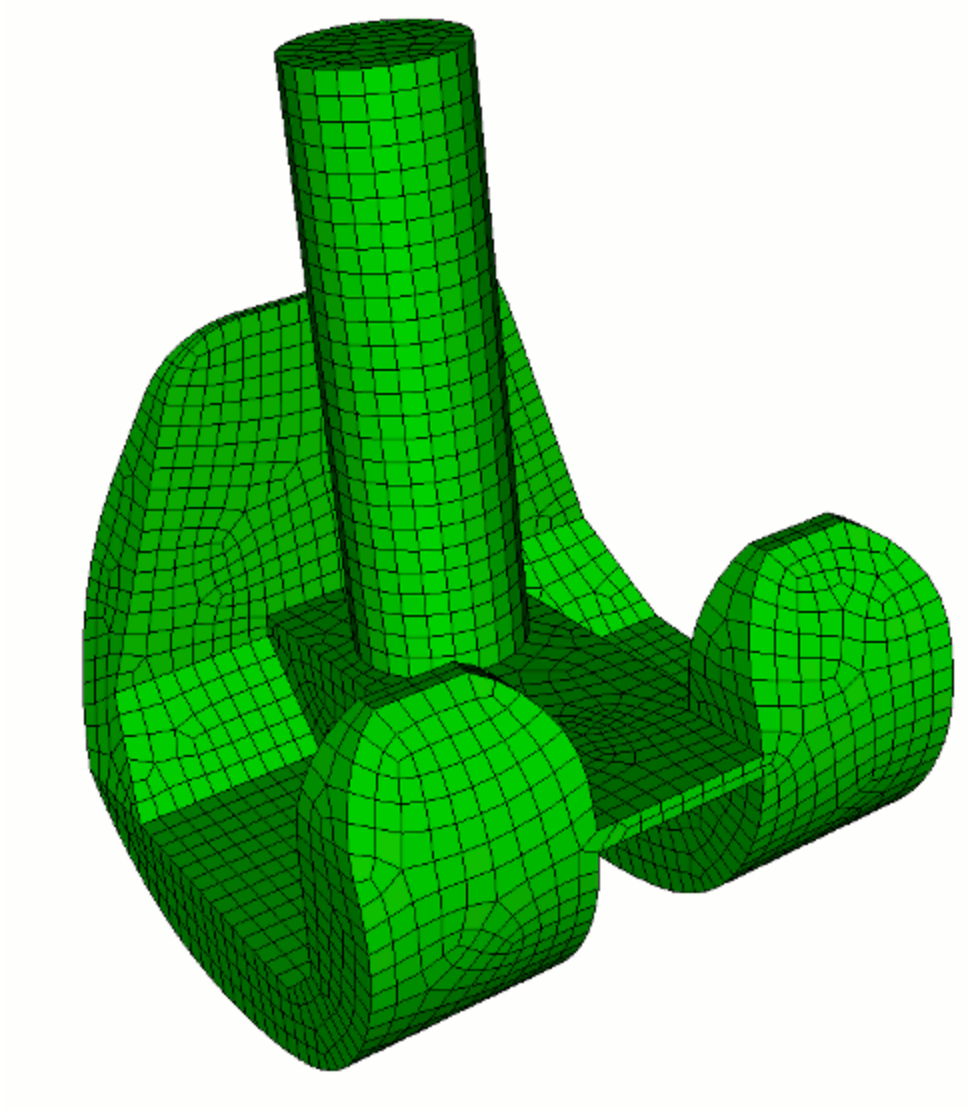


Figure 2. Paved mesh before using cleanup extend

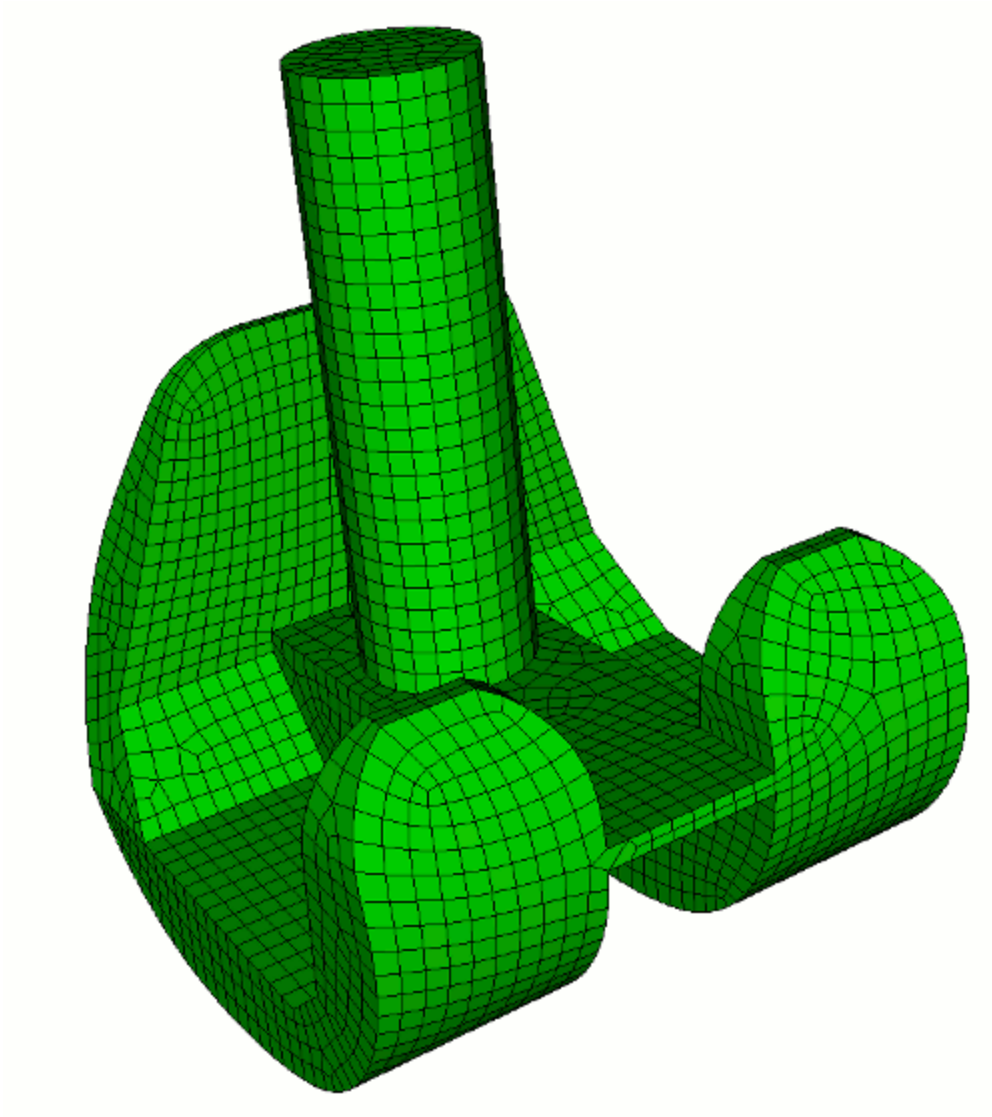


Figure 3. Paved mesh after using cleanup extend

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.

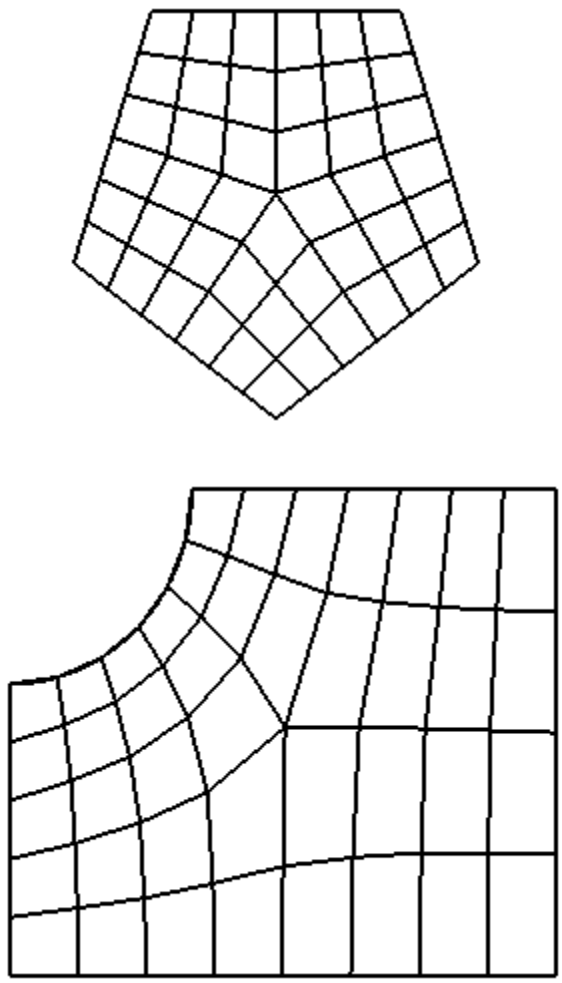


Figure 1. Examples of Pentagon Scheme Meshes

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

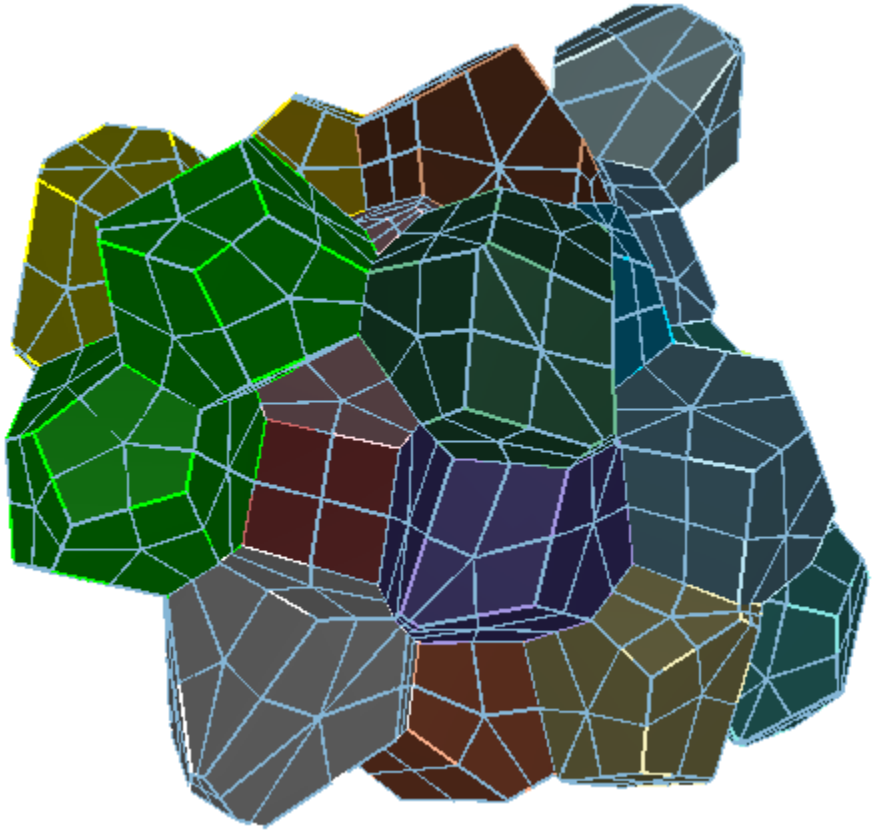
Surface <range> Scheme Polyhedron

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.



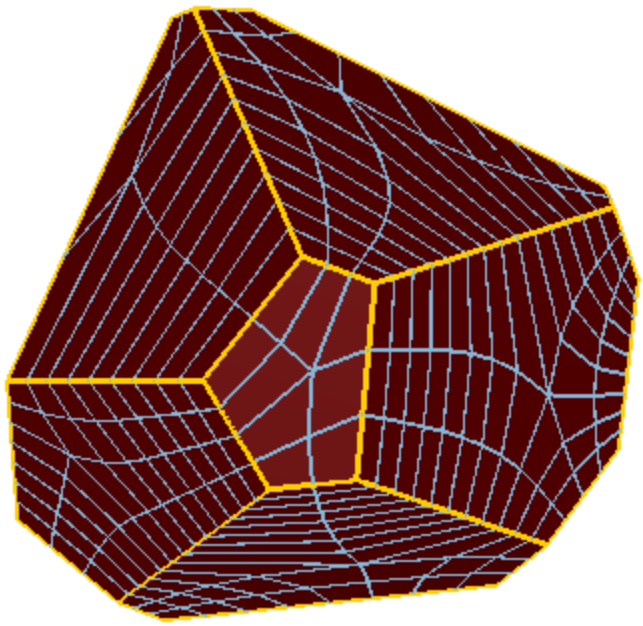


Figure 1. Examples of Polyhedron Scheme Meshes

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 inner region and outer region. The inner region is a mesh of a cube and the outer region contains fronts that trasition from cube surface to sphere surface. The number of azimuthal intervals around the equator is controlled by the az_interval input parameter. The number of radial intervals in the outer region is controlled by the graded_interval input parameter. Azimuthal mesh lines in the outer portion of the sphere have constant radius. The bias parameter controls the amount of radial grading in the outer region of the mesh (default=1 gives a uniform mesh). The fraction parameter (between 0 and 1) determines what fraction of the sphere is occupied by the inner cube. The fraction is defined as ratio of cube's body diagonal to sphere's diameter. The Max_smooth_iterations determines the number of smoothing iterations. By default number of smoothing interations is set to 2. Note that even though smoothing improves the quality of the mesh, it disturbs the bias and fraction. When bias and fraction are very 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:

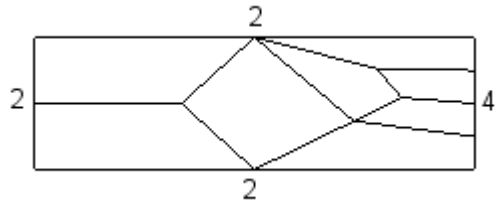
Produces a simple transitional mapped mesh.

Syntax:

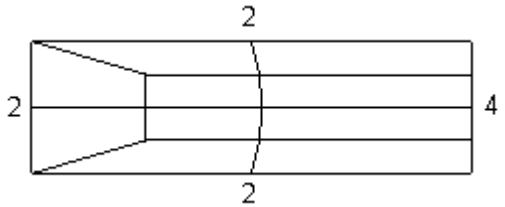
Surface <surface_id_range> Scheme STransition [Triangle] [Coarse]

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.

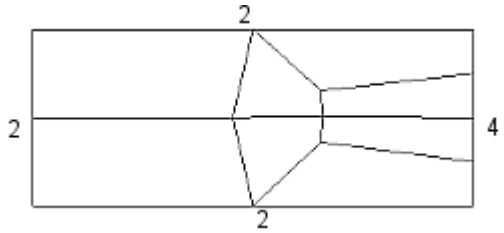


Pave scheme



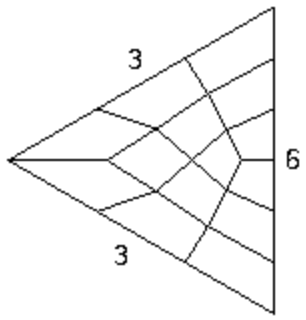
STransition scheme

The **coarse** option forces the mesh to transition to a coarser mesh in the first layer.



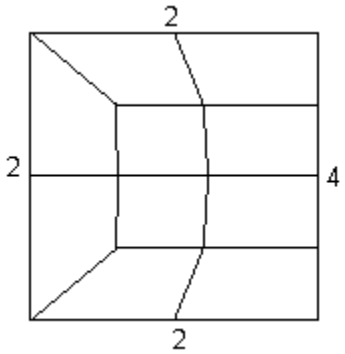
STransition scheme with coarse option

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.

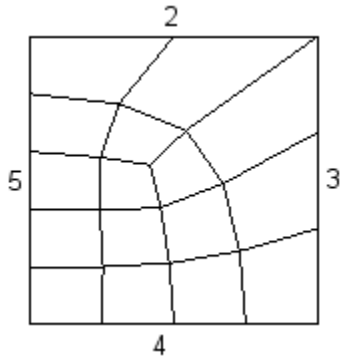


STransition scheme on a triangular surface with intervals set to 3, 3, and 6.

The figures below show the STransition meshing scheme response to different shapes and interval settings.

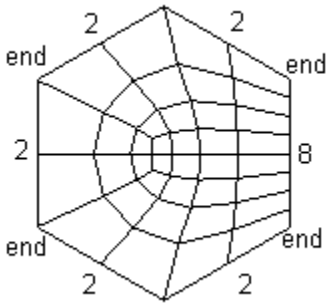


STransition scheme on a rectangular surface with three intervals set to 2 and one set to 4.



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.

Stride

Applies to: Curves

Summary: Mesh a curve with node spacing based on a general field function.

Syntax:

Curve <range> Scheme Stride

Discussion:

The ability to specify the number and location of nodes based on a general field function is also available in CUBIT. With this capability the node locations along a curve can be determined by some field variable (e.g. an error measure). This provides a means of using CUBIT in adaptive analyses. To use this capability, a sizing function must have been read in and associated to the geometry (See Exodus II -based field function for more information on this process). After a sizing function is made available, the *stride* scheme can be used to mesh the 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.

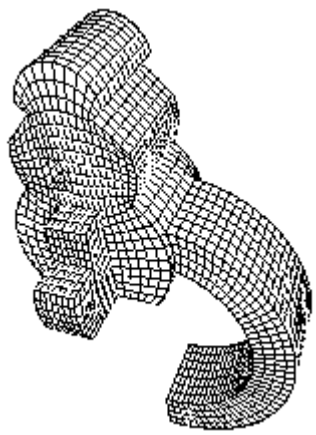


Figure 1. Quadrilateral and Hexahedral meshes generated by submapping

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.

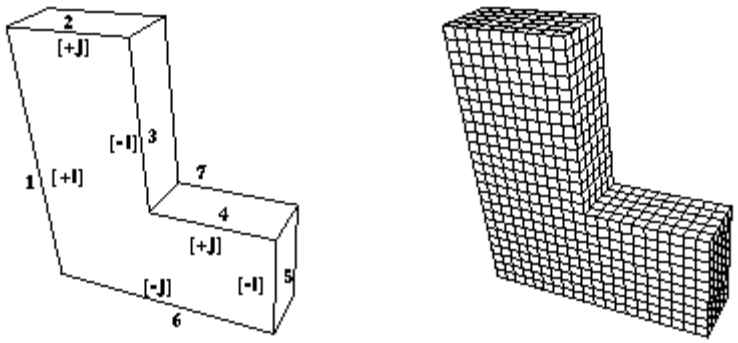


Figure 2. Scheme Submap Logical Properties

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}

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.

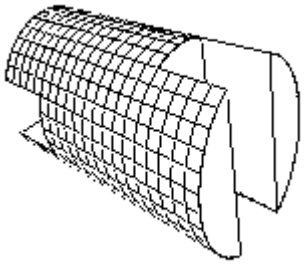


Figure 3. Periodic Surface Meshing with Submapping

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.

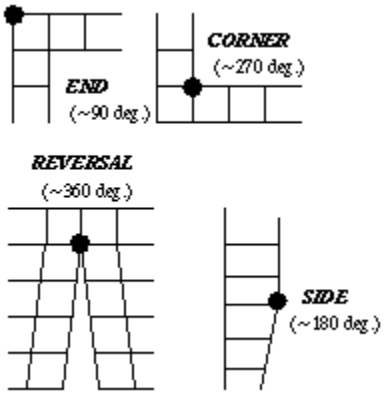


Figure 1. Angle Types for Mapped and Submapped Surfaces: An End vertex is contained in one element, a Side vertex two, a Corner three, and a Reversal four.

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 have a firmness, just as meshing schemes do. Automatically selected vertex types are **soft**, while user-set vertex types are **hard**. Instead of a type, an angle in degrees can be specified instead.

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. The **loop_index** is used only for vertices that are on the boundary of a single surface more than once.

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.

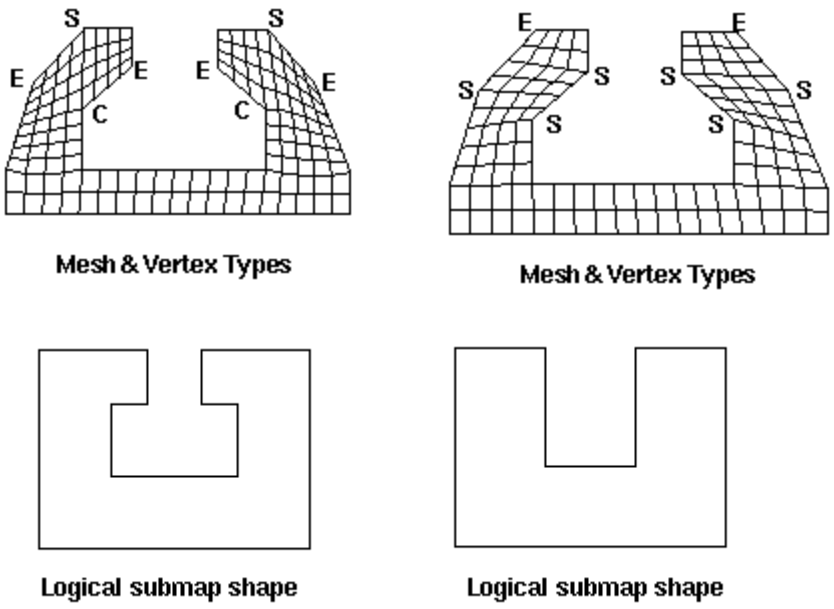


Figure 2. Influence of vertex types on submap meshes; vertices whose types are changed are indicated above, along with the mesh produced; logical submap shape shown below.

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> Set [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.

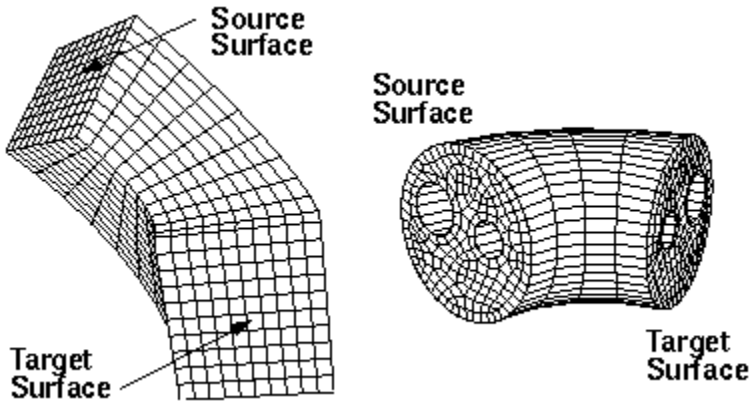


Figure 1. Sweep Volume Meshing

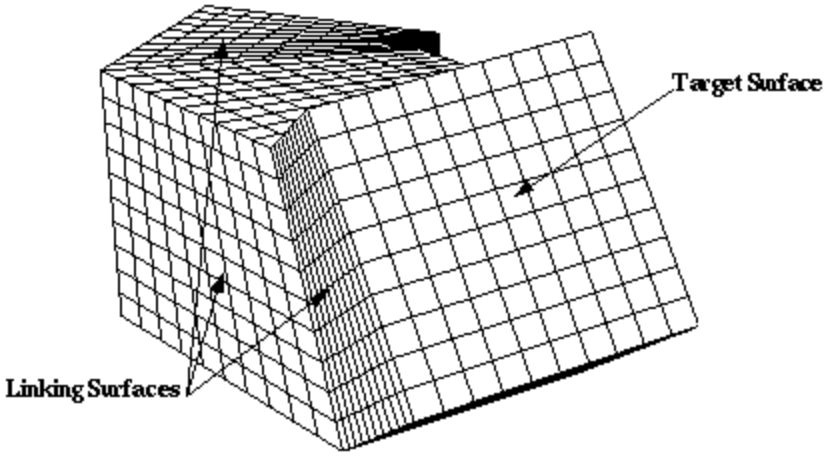


Figure 2. Multiple Linking Surface Volume Meshing with Scheme Sweep

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.

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.

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 controls 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.

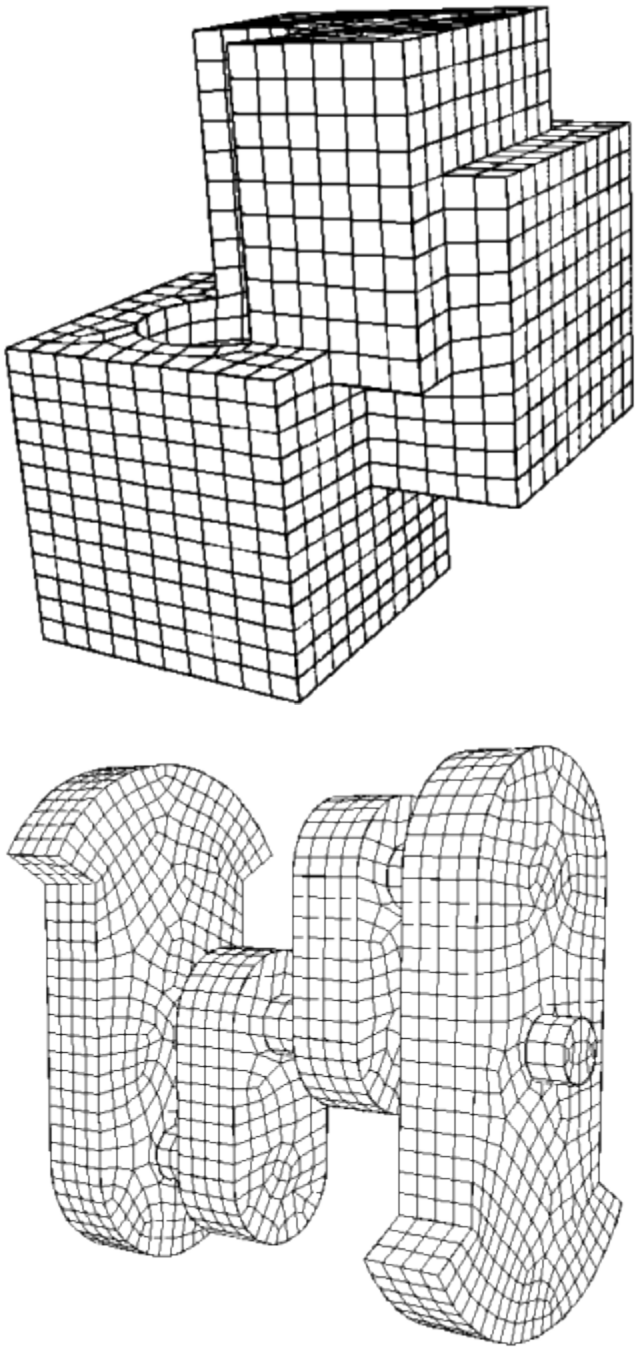


Figure 4. Examples of Multisweep meshes.

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.

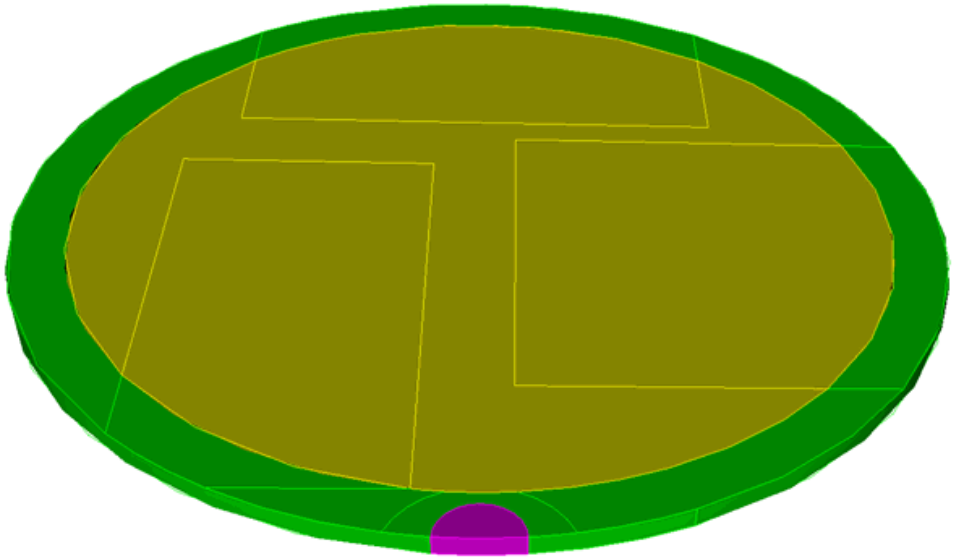


Figure 1 - Linking surfaces of a many-to-one sweepable solid (shown in green) is mappable

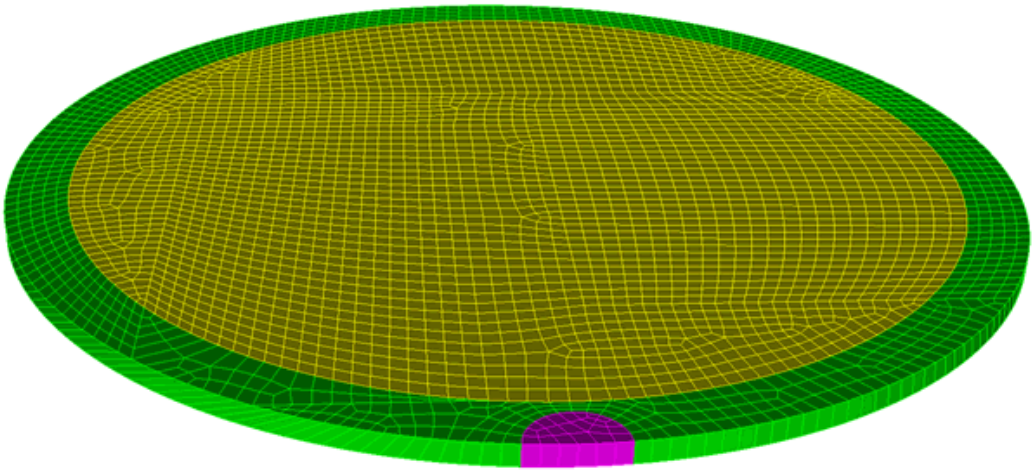


Figure 2 - Highly skewed elements on the linking mapped surface with 'redistribute nodes OFF'

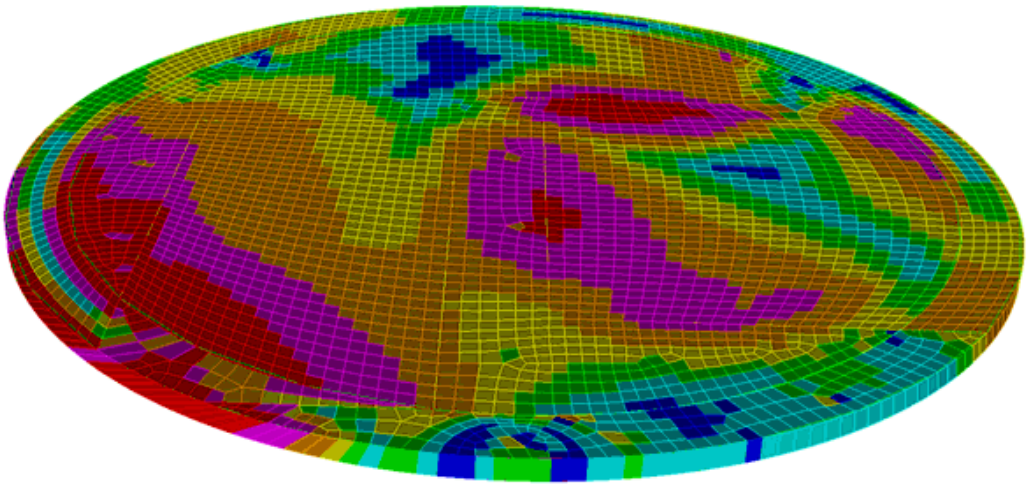


Figure 3 - Quality of mesh with 'Redistribute Nodes OFF'

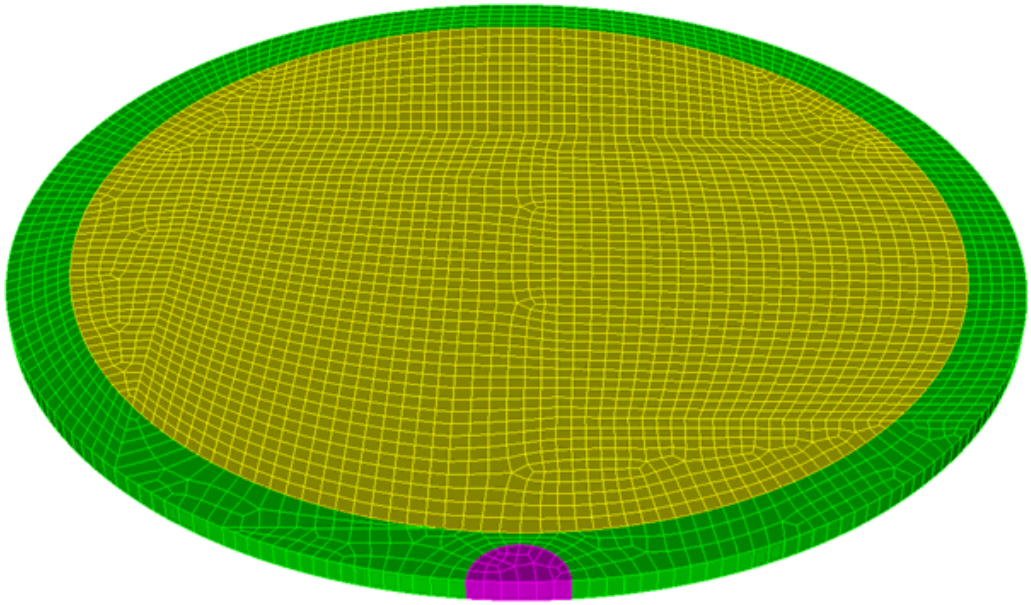


Figure 4 - High skew on the linking mapped surface can be avoided with 'Redistribute Nodes ON'

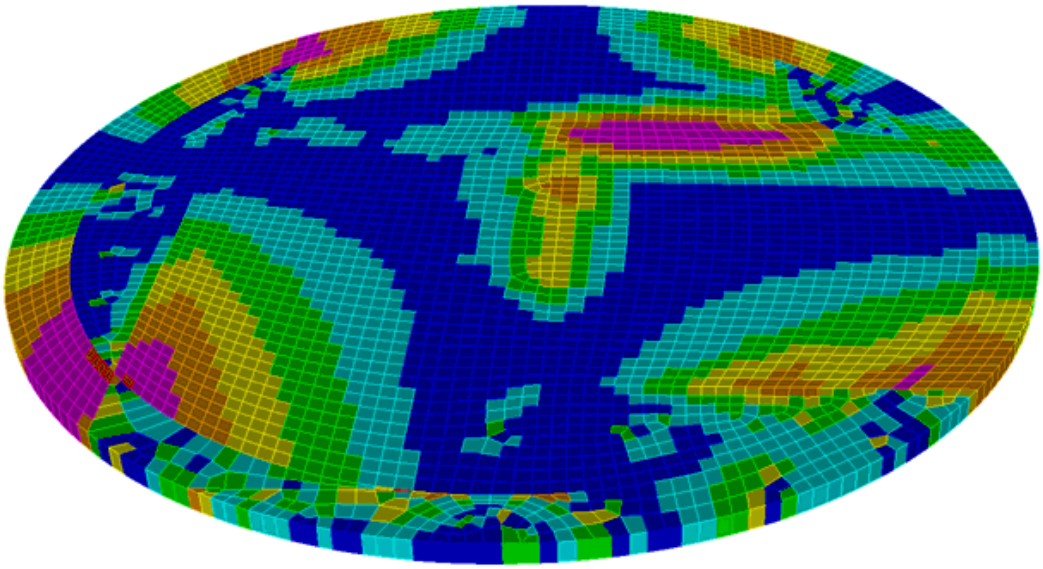


Figure 5 - Quality of mesh with 'Redistribute Nodes ON'

TetMesh

Applies to: Volumes

Summary: Automatically meshes a volume with an unstructured tetrahedral mesh.

Syntax:

Volume <range> Scheme TetMesh

Related Commands:

[Set] Tetmesher **Optimize Level** <level>

[Set] Tetmesher **Optimize Overconstrained** {on|OFF}

[Set] Tetmesher **Optimize Sliver** {on|OFF}

[Set] Tetmesher **Optimize Default**

[Set] Tetmesher **Boundary Recovery** {on|OFF}

[Set] Tetmesher **Interior Points** {ON|off}

THex Volume All

Volume <volume_id> **Tetmesh Respect** {Face|Tri|Edge|Node} <range>

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 = 1.0>

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 or TriAdvance) or a quadrilateral scheme with the quadrilaterals being split into two triangles.

The Distene/INRIA tet-mesher is included in CUBIT. This is a robust and fast tetrahedral mesher developed in France at INRIA and distributed and maintained by Distene. Figure 1 shows a volume filled with tetrahedra by this algorithm.

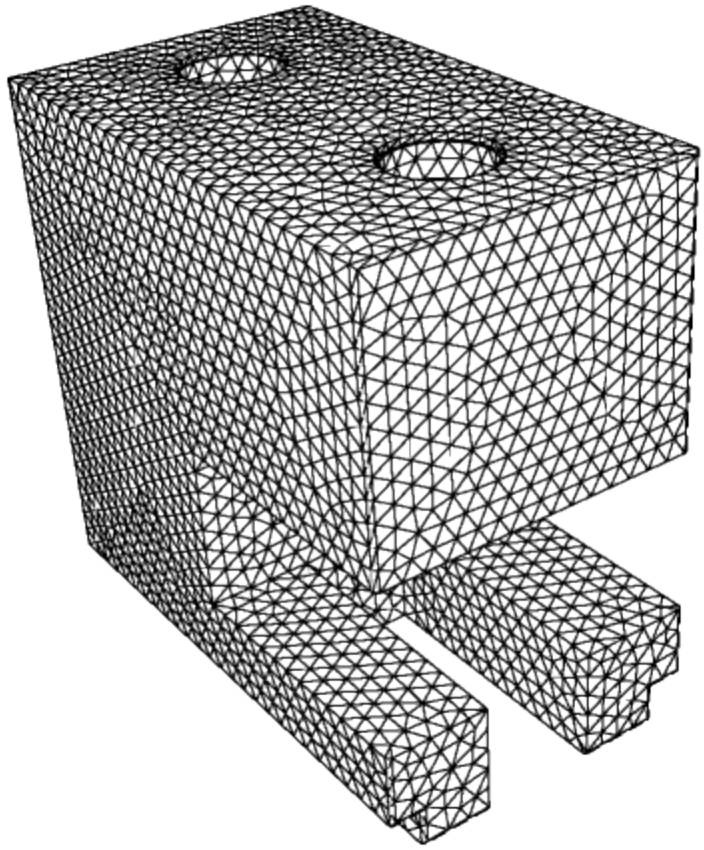


Figure 1. Tetrahedral Mesh generated with the TetMesh scheme. Surface meshing was performed with the TriAdvance scheme.

Cubit uses three of the four Distene/INRIA tetmesher modules: standard, boundary recovery and anisotropic-scalar. The fourth module, anisotropic-tensor is not available to the Cubit user at this time. If you need element sizing based on a tensor field, please contact cubit-help@sandia.gov.

The boundary recovery module differs from the standard one in that it allows the surface mesh constraints to be relaxed during tet-meshing. Once tet-meshing is done, the surface mesh constraints are reapplied. This allows tet-meshing of volumes bounded by poor quality surface triangularizations. The anisotropic-scalar module allows the user to apply a scalar sizing function to the volume, and this size is respected, as far as possible, throughout the mesh.

The standard and boundary recovery modules are at revision 4.2, while the anisotripic modules are at revision 3.5.

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 until the option is changed by the user.

The *Tetmesher Optimize* options listed below are only available in the standard module and will be ignored by the others.

[Set] Tetmesher Optimize Level <level>

The *Tetmesher Optimize Level* command allows the user to control the degree of optimization. The optimization level is an integer in the range 0 to 6, which represent optimization levels of *none, light, medium, standard, strong, heavy, and extreme*. Greater values will result in greater computation time. The default is 3 or standard optimization.

[Set] Tetmesher Optimize Overconstrained {on|OFF}

The *Tetmesher Optimize Overconstrained* command enables or disables an additional optimization step that attempts to eliminated tetrahedral elements that share multiple surface triangles. This optimization may improve the mesh for finite element analysis. The default setting is *off*.

[Set] Tetmesher Optimize Sliver {on|OFF}

The *Tetmesher Optimize Sliver* command enables or disables an additional optimization step that attempts to eliminate tetrahedral sliver elements. The default setting is *off*.

[Set] Tetmesher Optimize Default

The *Tetmesher Optimize Default* command restores the default optimizations values: level = 3 (standard), overconstrained = off, and sliver = off.

[Set] Tetmesher Boundary Recovery {on|OFF}

The *Tetmesher Boundary Recovery* command allows the user to enable or disable the use of the boundary recovery module. By default, the standard module is used if no sizing function is declared for the volume; the anisotropic-scalar module is used if one is declared. Cubit selects the boundary recovery module if the minimum surface element condition number is less than 0.2. This option allows the user to force Cubit to select the boundary recovery module instead of the standard module even if the minimum surface element condition number is greater than 0.2. The default setting is *off*.

[Set] Tetmesher Interior Points {ON|off}

Infrequently, the user desires a model with as few interior points as possible. The *Tetmesher Interior Points* command allows the user to enbale or disable the insertion of interior points. If interior points are disable, 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*.

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|Node} <range>

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.

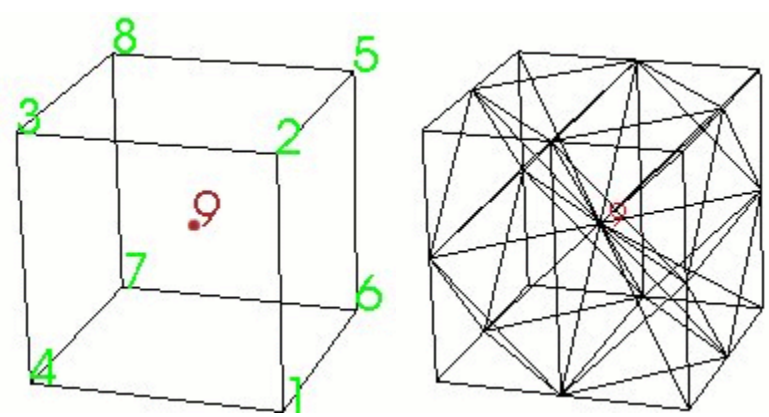


Figure 2. Example of using tetmesh respect to ensure node 9 is captured in the tetmesh.

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 node12 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

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 mesh size inside the volume

Volume <id_range> Tetmesh growth_factor <value 1.0 to 10.0 = 1.0>

The Tetmesh growth_factor command controls the mesh gradation inside the volume. The default value of growth_factor is 1.0. Therefore, the tetmesh scheme, by default, generates a uniform mesh. Note that the default value of growth_factor was 1.05 in CUBIT 14.0 and older releases. Therefore, the default mesh generated in CUBIT 14.0 and older releases contain a lower number of nodes and elements. Specifying growth_factor > 1.0 increases the mesh size in the interior layers compared to the boundary mesh size. This reduces the total number of elements and nodes, thus reducing the total degrees of freedom.

Generating a Tetmesh from a Skin of Triangles

Tetmesh Tri <range> [Make {Block|Group} [<id>]]

Tetmesh Tri <range> {Add|Replace} {Block|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 object receives the tet mesh. The command fails if an object with the optional identifier exists. If the object identifier is omitted, the identifier is set to the next available block.

The second 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 tet mesh to the object. The **replace** option removes any existing mesh from the object before adding the tet mesh.

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.

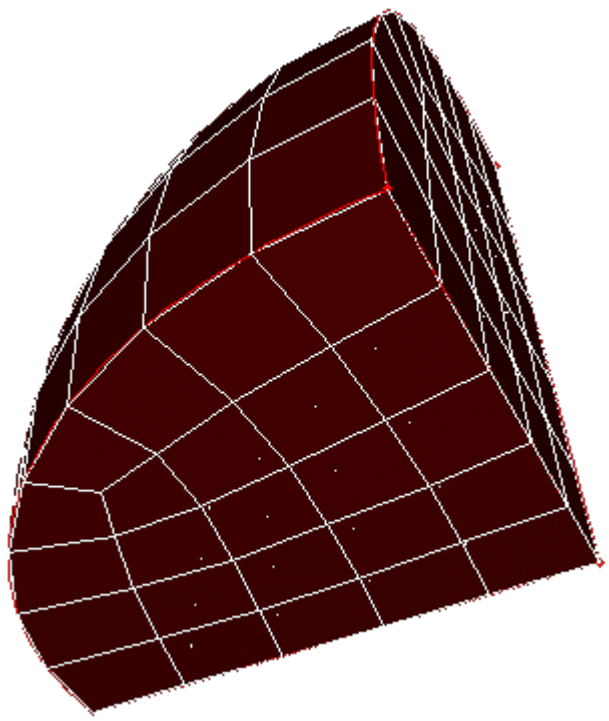


Figure 1. Sphere octant hex meshed with scheme Tetprimitive, surfaces meshed using scheme Triprimitive

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.

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:

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 all available triangle meshing schemes.

Syntax:

Surface <range> Scheme TriMesh

Discussion:

The scheme **TriMesh** automatically switches between TriDelaunay, TriAdvance, and the QTri schemes. First it tries the **TriDelaunay** scheme if the surface is parametric and non-periodic. If that fails, it tries the **TriAdvance** scheme; and if that fails it tries the QTri scheme. The QTri scheme first paves the surface and then cuts the quadrilateral elements in half to form triangles. Figure 1 shows a surface meshed with the TriAdvance scheme, and Figure 2 shows the same surface meshed using the Qtri method. The TriMesh scheme is the preferred triangle meshing scheme unless there is an overriding reason to select one of the others. You specify this scheme for a surface by giving the command:

Surface <range> Scheme TriMesh

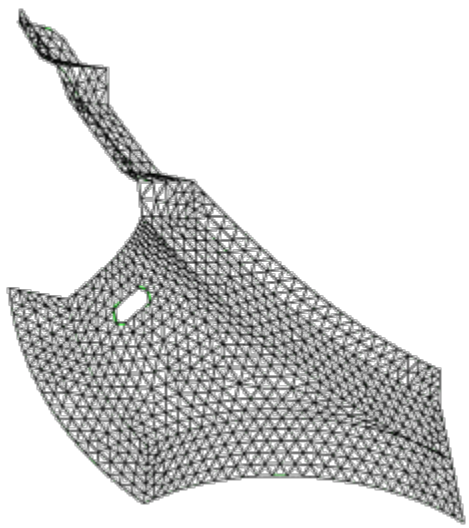


Figure 1. Triangle mesh generated with scheme TriAdvance

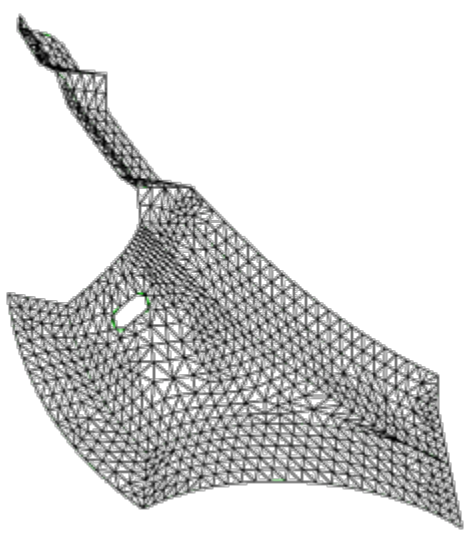


Figure 2. Triangle mesh generated with QTri scheme

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.

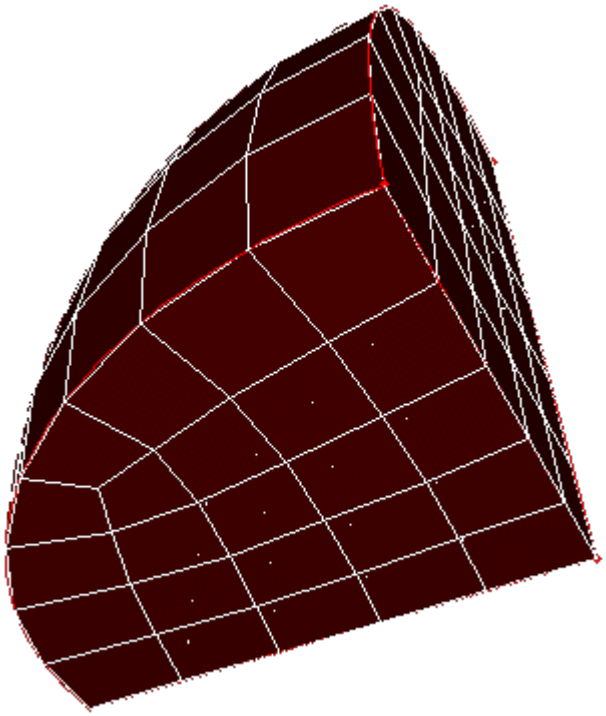


Figure 1. Surfaces meshed with scheme Triprimitive

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).

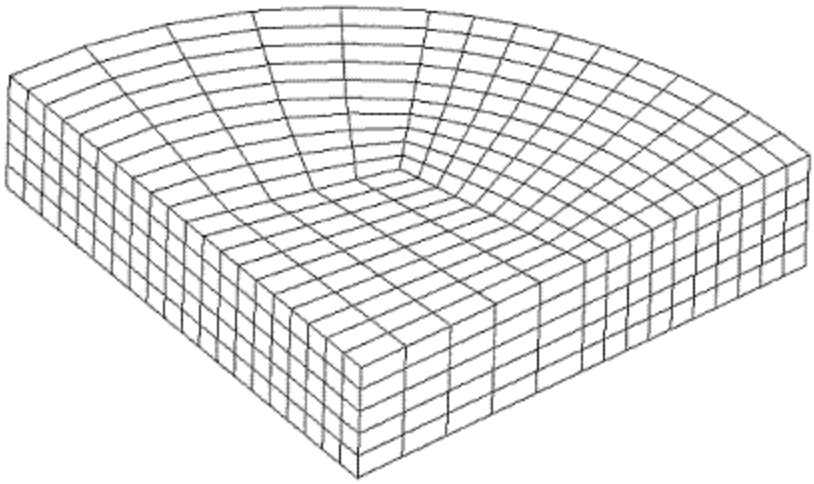


Figure 1. Tri-section Radialmesh

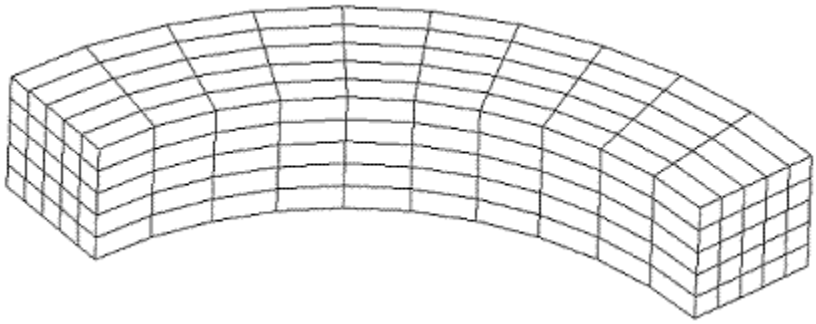


Figure 2. Non-tri-section Radialmesh

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
```

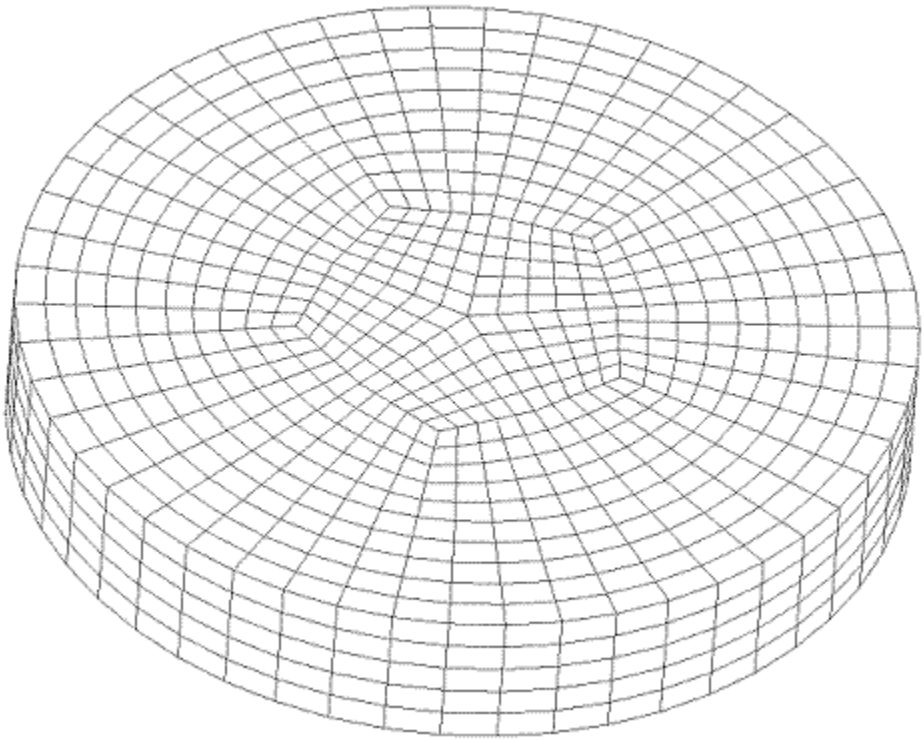


Figure 4. Radialmesh using full360 option

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
```

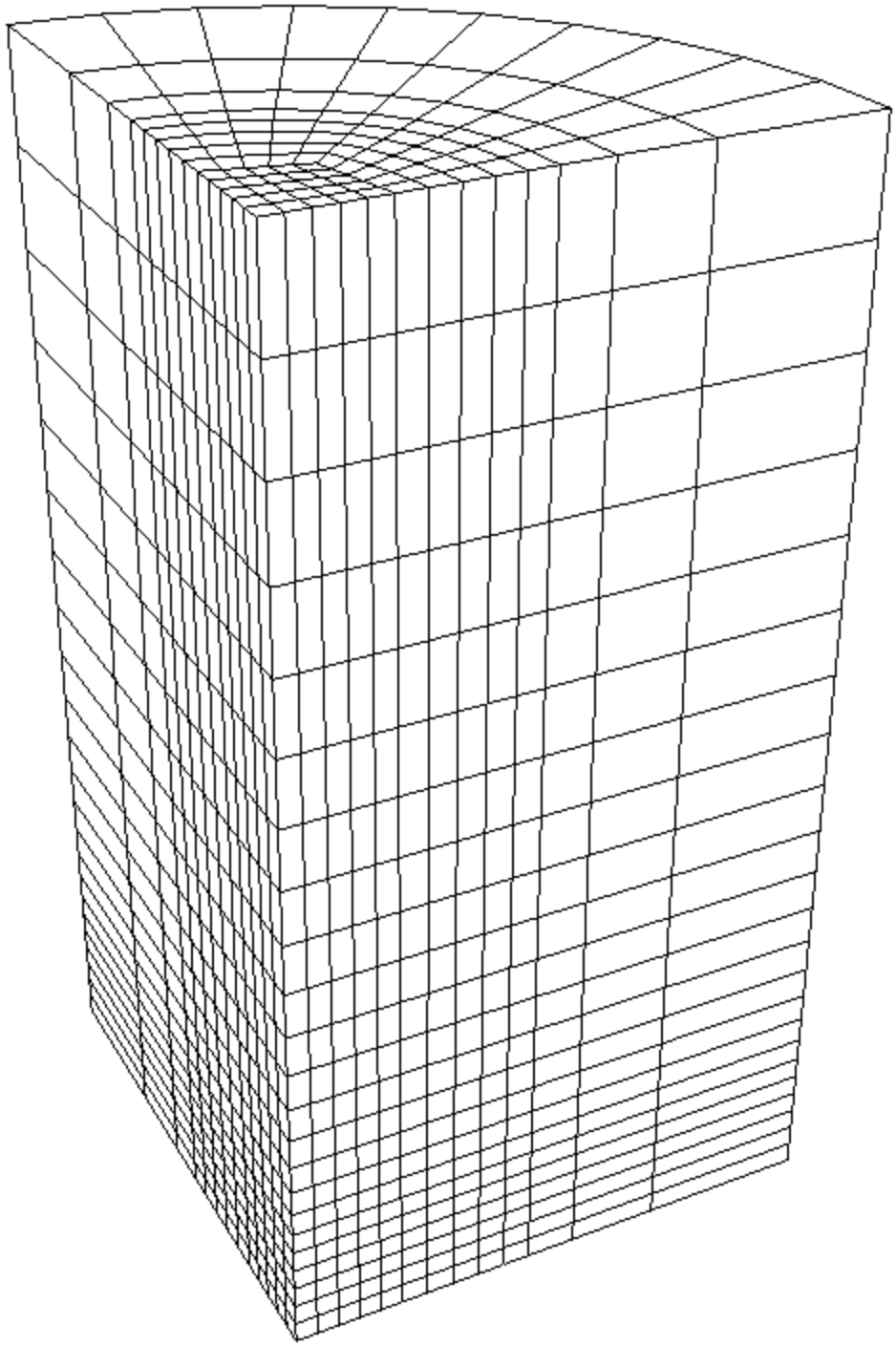


Figure 5. Radialmesh created with biased spacing

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:

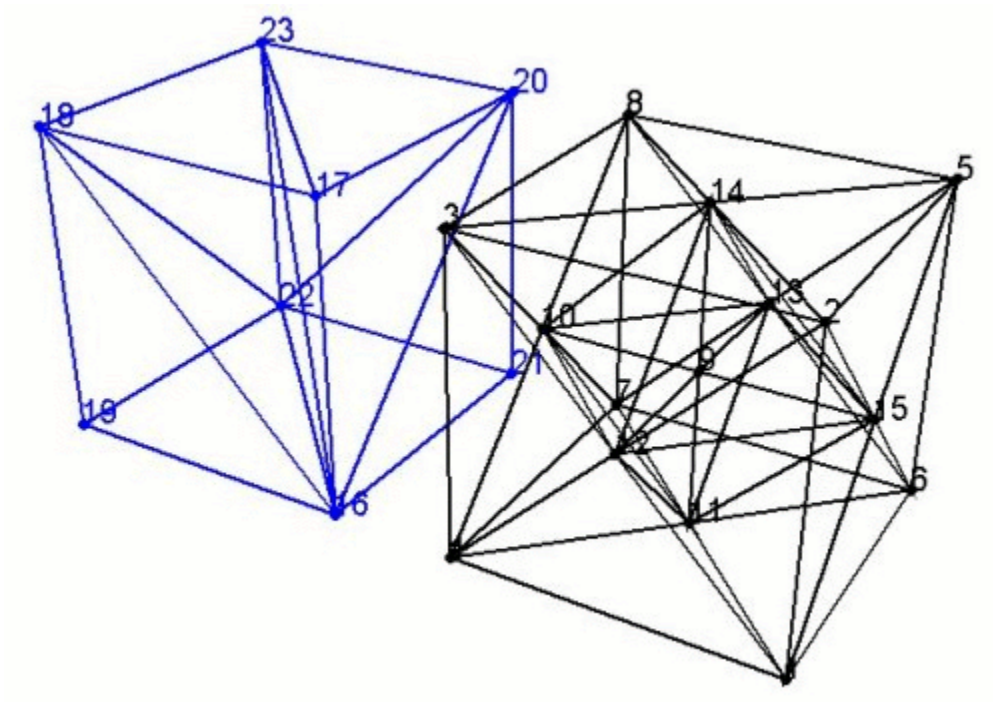


Figure 1. Left: Unstructured method creates 6 tets per hex. Right: Structured method creates 28 tets per hex

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>

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).

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.

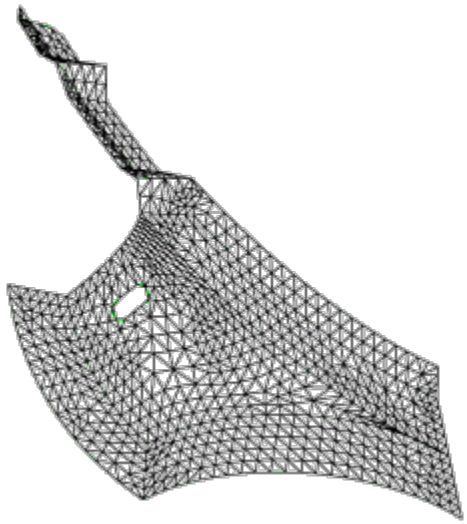


Figure 1. Surface meshed with scheme QTri

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.

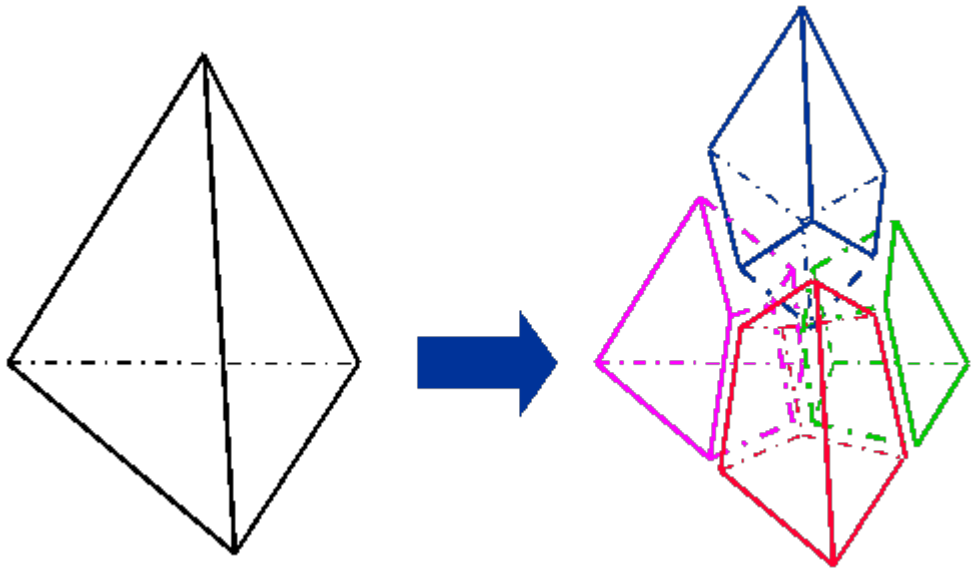


Figure 1. Conversion of a tetrahedron to four hexahedra, as performed by the THex algorithm.

.

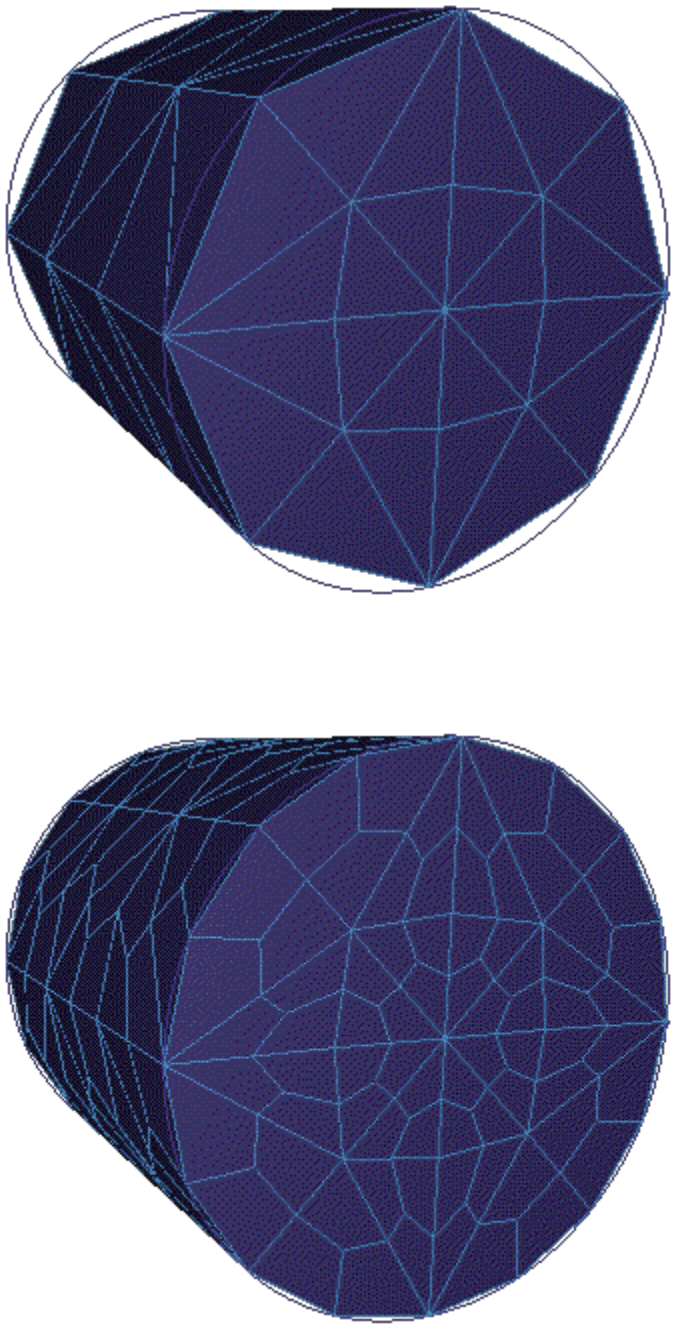


Figure 2. A cylinder before and after the THex algorithm is applied.

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.

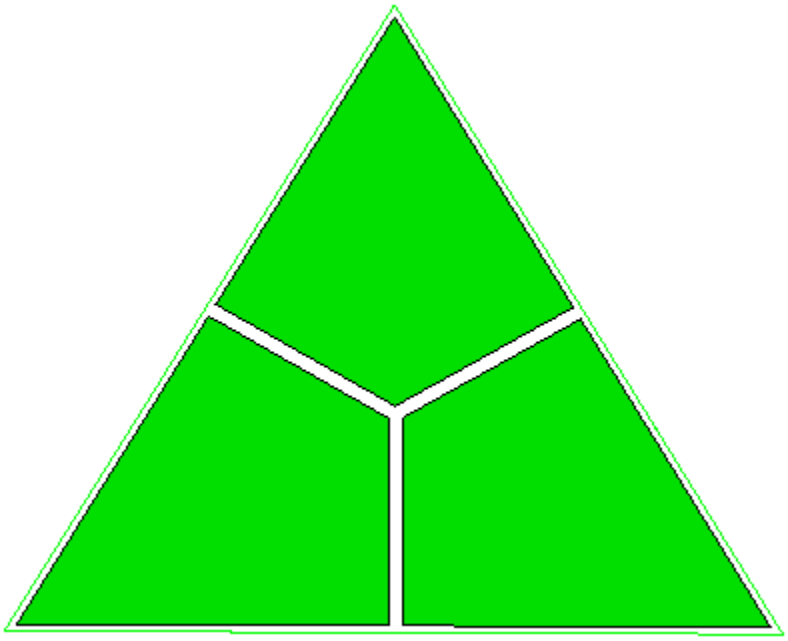


Figure 1. A triangle split into 3 quads using the TQuad scheme

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 <range> Scheme Copy [Source Surface] <id> [[Source Curve <id> Target Curve <id>] [Source Vertex <id> Target Vertex <id>] [Nosmoothing]

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 <curve_id_range> Onto Curve <curve_id_range> [Source Node <starting node id> <ending node id>] [Source Percent [<percentage>|auto]] [Source [combine|SEPARATE]] [Target [combine|SEPARATE]] [Source Vertex <id_range>] [Target Vertex <id_range>]

Copy Mesh Surface <surface_id> Onto Surface <surface_id> [Source Face <id_range>] [Source Node <id> Target Node <id>] [Source Edge <id> Target Edge <id>] [Source Vertex <id> Target Vertex <id>] [Source Curve <id> Target Curve <id>] [Nosmoothing]

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.

Volume mesh copying depends on the surface copying scheme. Because of this, the target volume must not have any of its surfaces meshed already.

Because of how the copying algorithm works, the target mesh might not be an exact copy of the source mesh. This happens because of the effects of smoothing. If an exact copy is required, there are two possible solutions. The first option is useful when the source and target surfaces or volumes are exact matches. If this criterion is met, the user may specify the **Nosmoothing** option. That will disable any smoothing of the mesh on the target surface and thereby providing an exact copy of the mesh. The second option is useful if the source and target surfaces are not identical. In this case 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.

Mirroring a Mesh

Applies to: Surfaces

Summary: Mirrors the mesh from one surface to another

Syntax:

> Surface <range> Scheme Mirror [Source Surface <id> [Source Vertex <id> Target Vertex <id>]] [Nosmoothing]
>
> Mirror Mesh Surface <surface_id> Onto Surface <surface_id> [Source Vertex <id> Target Vertex <id> Source Curve <id> Target Curve <id> Source Node <id> Target Node <id>] [Nosmoothing]

Discussion:

The mirror scheme is very similar to the copy scheme. In order to understand what is changed, a discussion of the copy command is in order. Depending on what the user enters for the copy scheme, the resulting mesh might be oriented one of two ways. For example, if the user entered:

Surface 1 scheme copy source surface 2 source vertex 5 target vertex 1

then the algorithm would match vertex 1 with vertex 5, but then would have to make a guess about how to match the curves. Lacking other pertinent data, the match will be a direct match, as is shown in the following figure:

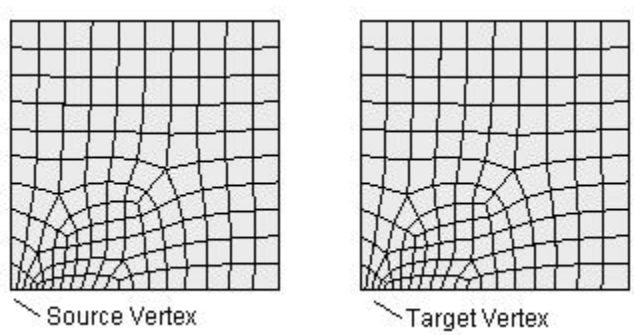


Figure 1. Surface 1 copied onto surface 2

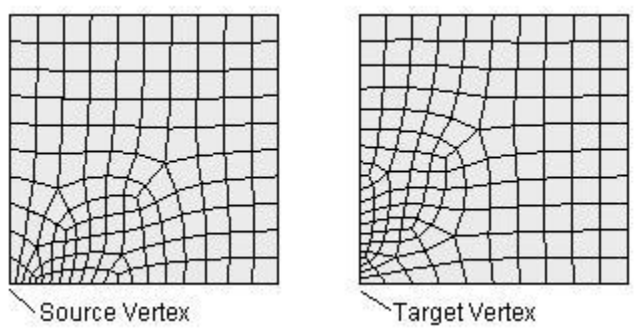


Figure 2. Surface 1 mirrored onto surface 2

This default matching can be changed by specifying more information for matching, or the user can specify scheme mirror. The mirror scheme sets up the copying information in such a way as to reverse the default orientation of the target mesh, as is shown in the above figure (right).

There are times when the resulting mesh may not match the original mesh exactly due to smoothing. Using the nosmoothing option will ensure that the resulting mesh matches the original mesh exactly.

The alternate form of the command copies the mesh immediately instead of setting a scheme first. This form of the command can also use curves and mesh entities as references.

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).

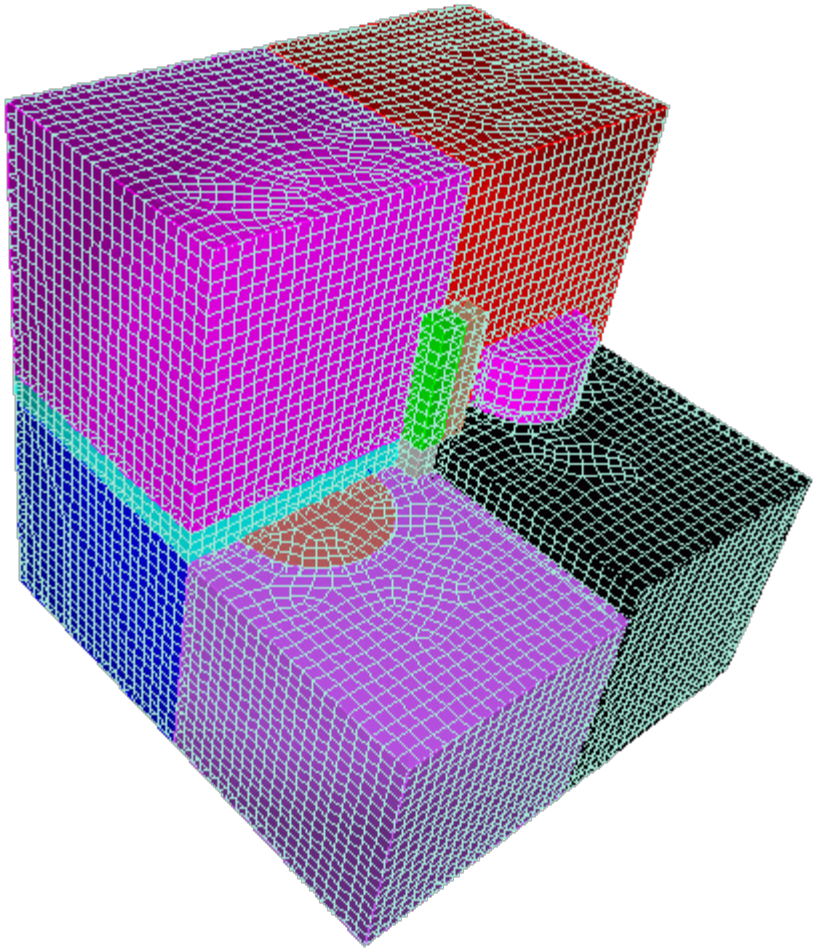


Figure 1. Non-trivial model meshed using automatic scheme selection

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.

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 Triangular Elements

The metrics used for triangular elements in CUBIT are summarized in the following table:

Function Name	Dimension	Full Range	Acceptable Range	Reference
Element Area	L^2	0 to inf	None	1
Maximum Angle	degrees	60 to 180	60 to 90	1
Minimum Angle	degrees	0 to 60	30 to 60	1
Condition No	L^0	1 to inf	1 to 1.3	2
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
Distortion	L^2	-1 to 1	0.6 to 1	4

Approximate Triangular Quality Definitions:

Element Area: (1/2) * Jacobian at corner node

Maximum Angle: Maximum included angle in triangle

Minimum Angle: Minimum included angle in triangle

Condition No. Condition number of the Jacobian matrix

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

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 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
Element Area	L^2	-inf to inf	None	1
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
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
Distortion	L^2	-1 to 1	0.6 to 1	6

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: 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

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.

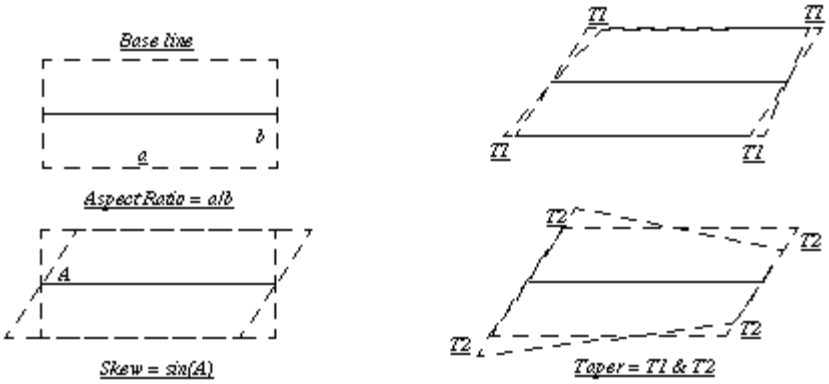


Figure 1. Illustration of Quadrilateral Shape Parameters (Quality Metrics)

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
Element Volume	L^3	-inf to inf	None	1
Condition No	L^0	1 to inf	1 to 3	2
Jacobian	L^3	-inf to inf	None	2
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
Distortion	L^0	-1 to 1	0.6 to 1	4

Tetrahedral Quality Definitions

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

Element Volume: (1/6) * Jacobian at corner node

Condition No.: Condition number of the Jacobian matrix at any corner

Jacobian: Minimum pointwise volume at any corner

Scaled Jacobian: 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

Distortion: {min(|J|)/actual volume}*parent volume, parent volume = 1/6 for tet

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.

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. SDRC/IDEAS Simulation: Finite Element Modeling - User's Guide

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
Element Volume	L^3	-inf to inf	None	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
Jacobian	L^3	-inf to inf	None	5
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
Distortion	L^0	0 to 1	0.6 to 1	6

Hexahedral Quality Definitions

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.

Element Volume: Jacobian at hex center.

Stretch: Sqrt(3) * minimum edge length / maximum diagonal length.

Diagonal Ratio: Minimum diagonal length / maximum diagonal length.

Dimension: Pronto-specific characteristic length for stable time step calculation. Char_length = Volume / 2 grad Volume.

Condition No. Maximum condition number of the Jacobian matrix at 8 corners.

Jacobian: Minimum pointwise volume of local map at 8 corners & center of hex.

Scaled Jacobian: 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

Distortion: {min(|J|)/actual volume}*parent volume, parent volume = 8 for hex

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. SDRC/IDEAS Simulation: Finite Element Modeling - User's Guide

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.

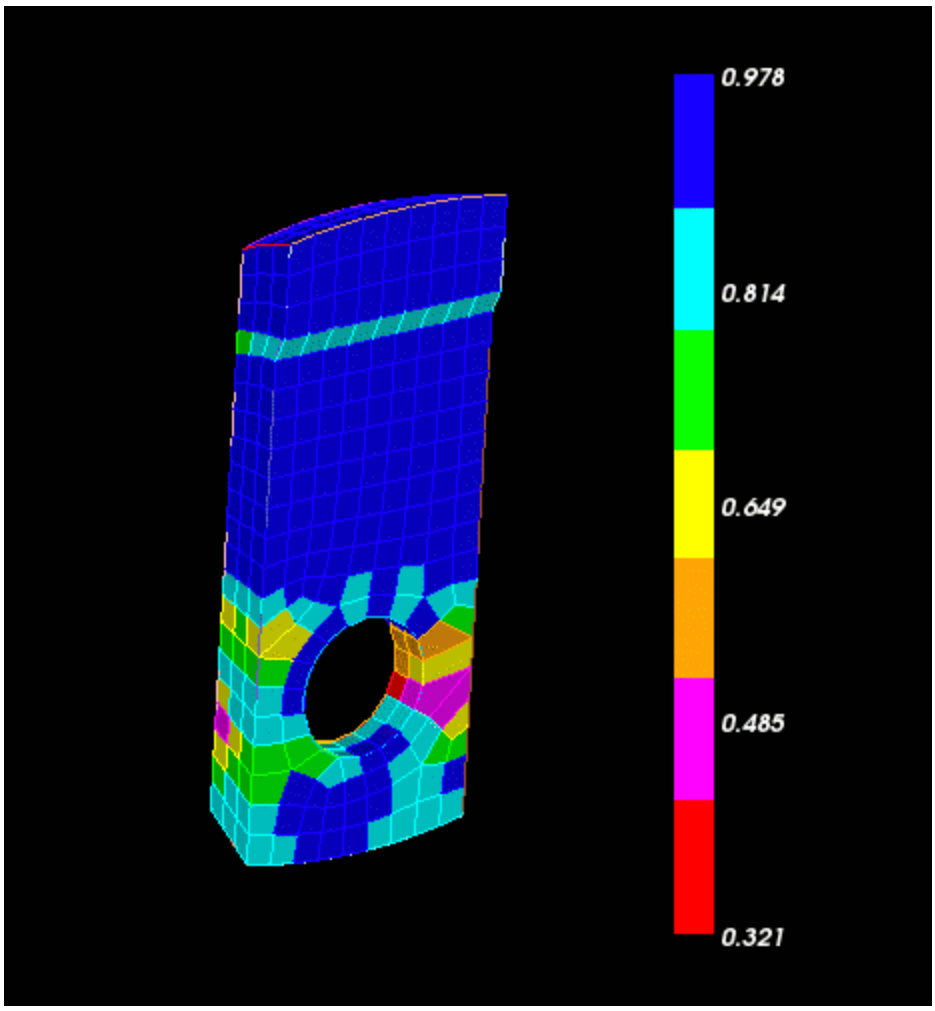


Figure 1. Quality Scale

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.

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

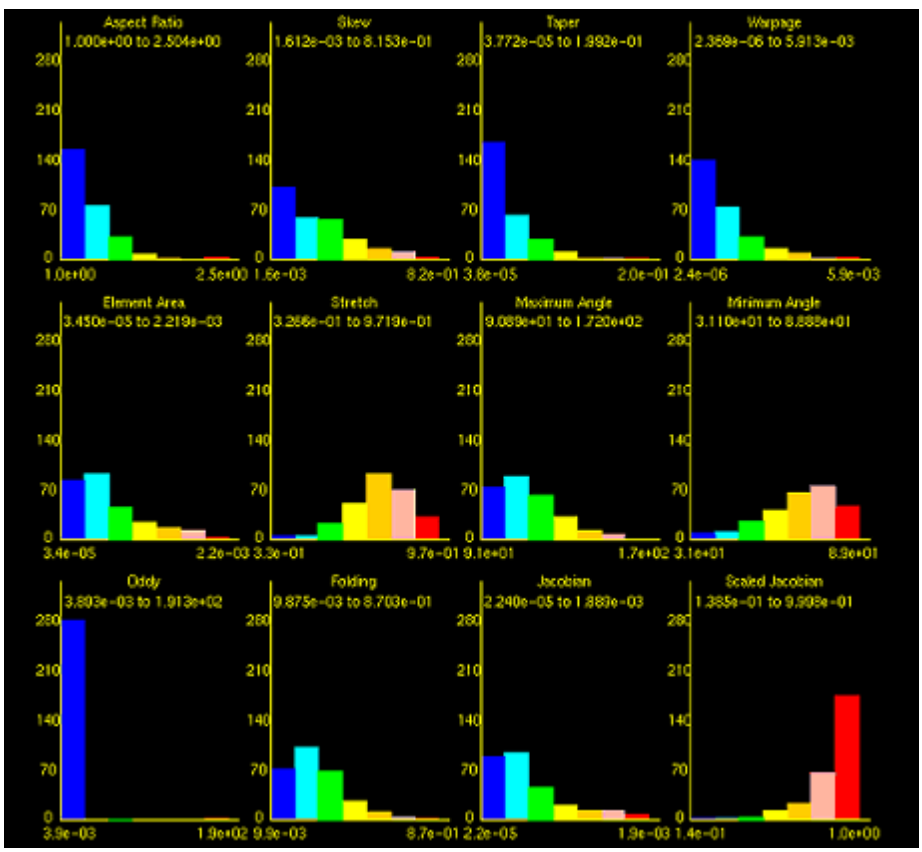


Figure 2. Histogram output from command "Quality Surface 24 Draw Histogram"

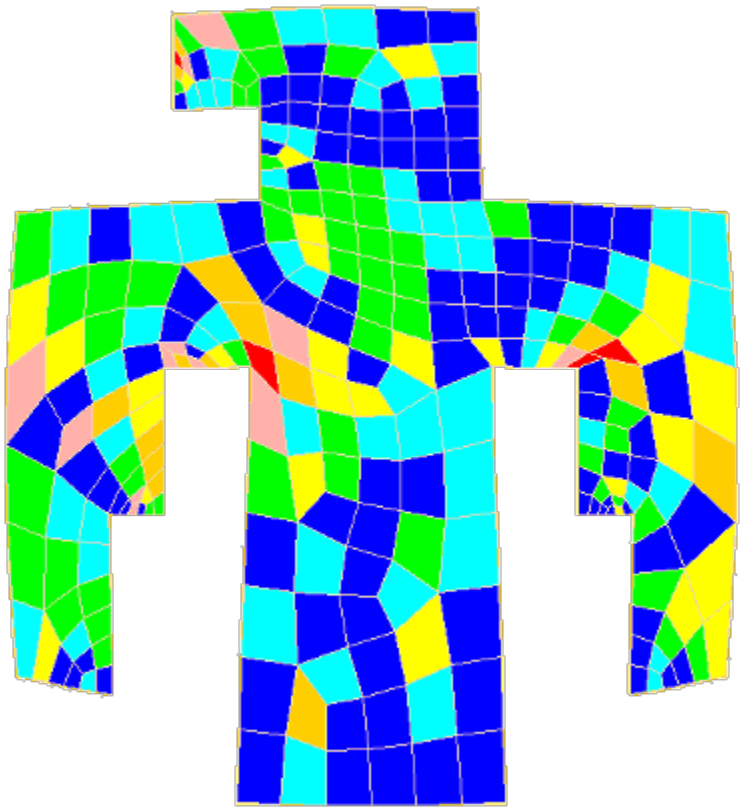


Figure 3. Graphical output of quality metric for command "Quality Surface 24 Skew Draw Mesh"

```
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"

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).

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.

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>]

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.

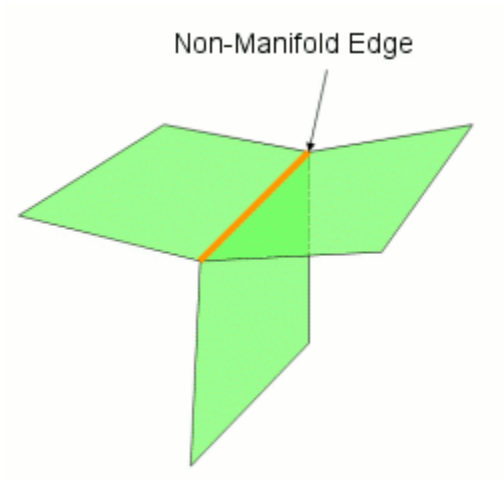


Figure 1. Topology check for quads and tris

For hexes and tets it looks for edges with two or more elements connected that do not share common faces.

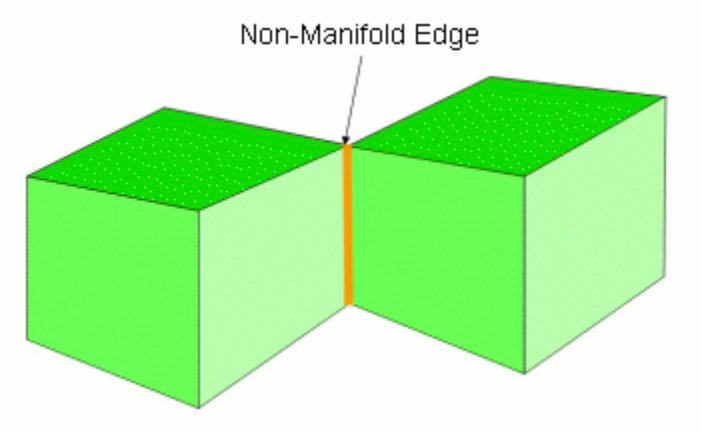


Figure 2. Topology check for hexes and tets

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.

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).

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.

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.

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**.

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.

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.

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.

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.

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
- Scale Mesh

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.

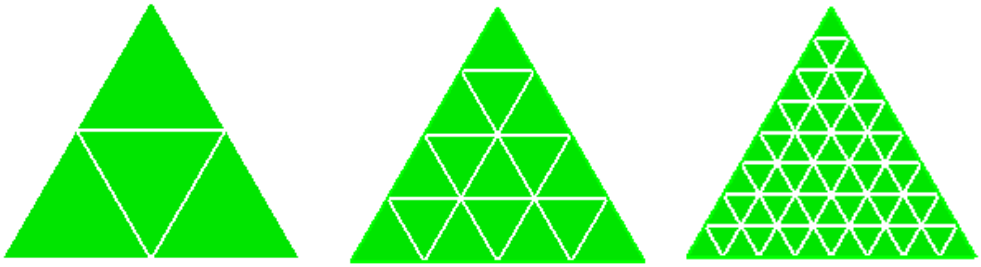
Global Mesh Refinement

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:

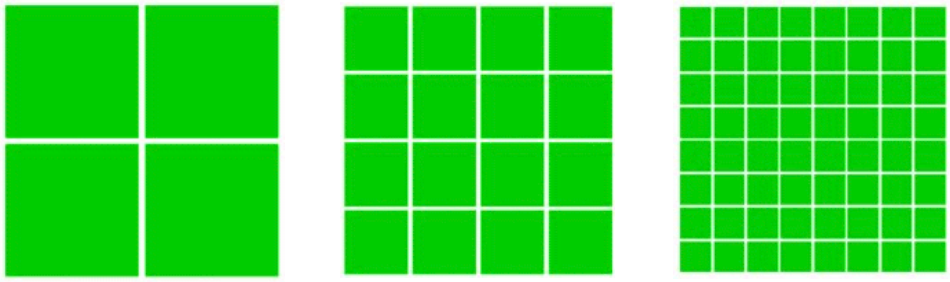
Refine Volume <range>numsplit<int>

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.



original mesh | NumSplit = 1 | NumSplit = 2



original mesh | NumSplit = 1 | NumSplit = 2

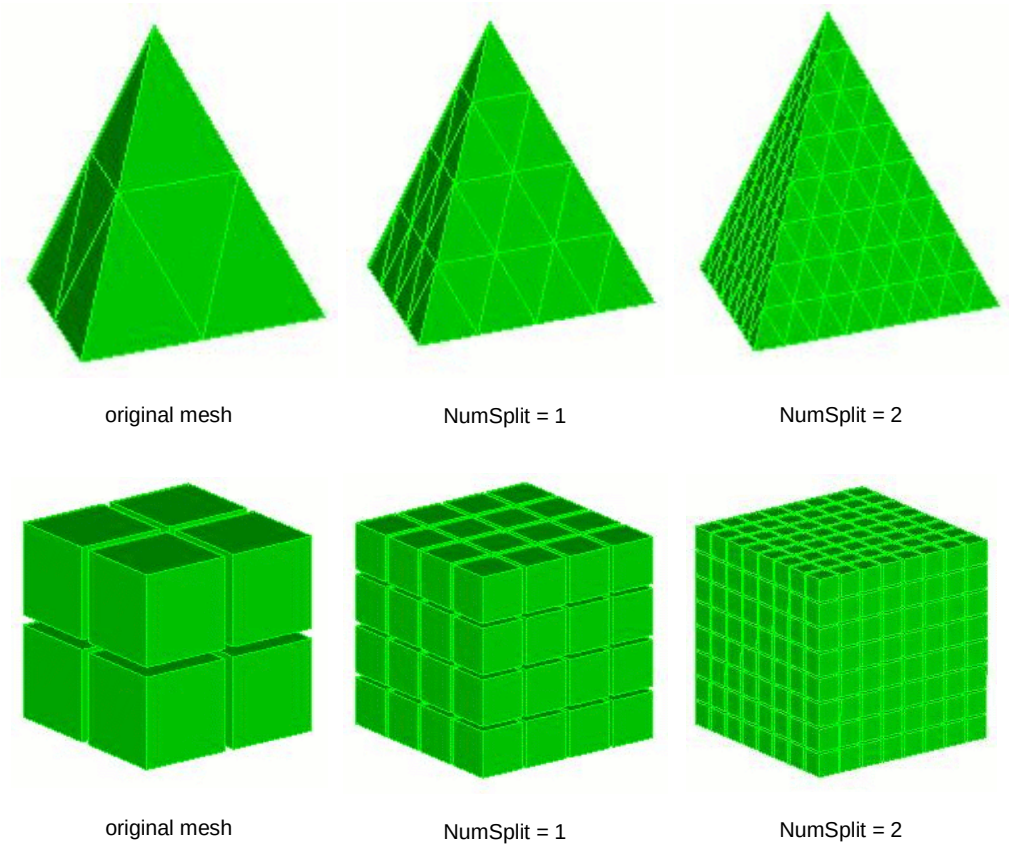


Figure 1. Example of uniform refinement for each of the mesh entities

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 16 and 64 elements respectively. For uniform refinement, the total number of elements obeys the following:

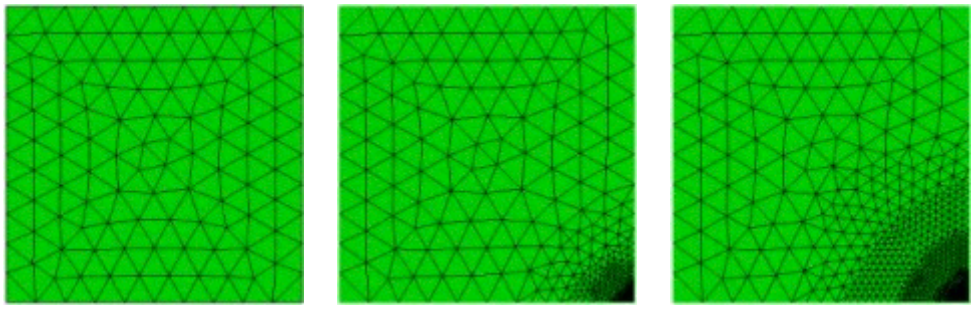
1. **NE = NI * E^NumSplit**

where **NE** is the final number of elements, **NI** is the initial number of elements and **E** is 4 or 8 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 (**E=**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.

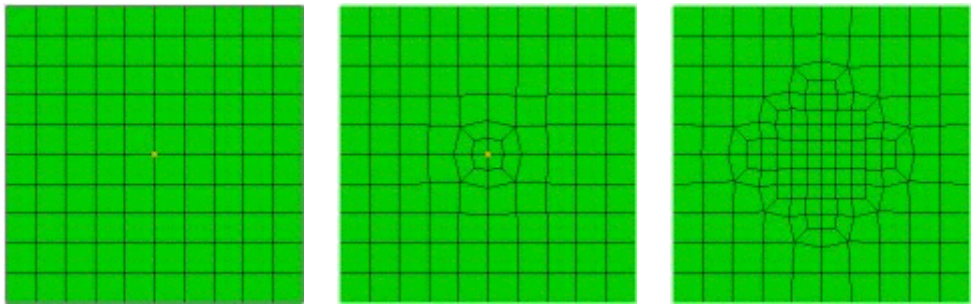


original mesh Bias=2.0 Bias=1.5

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.



original mesh Depth=1 Depth=2

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.

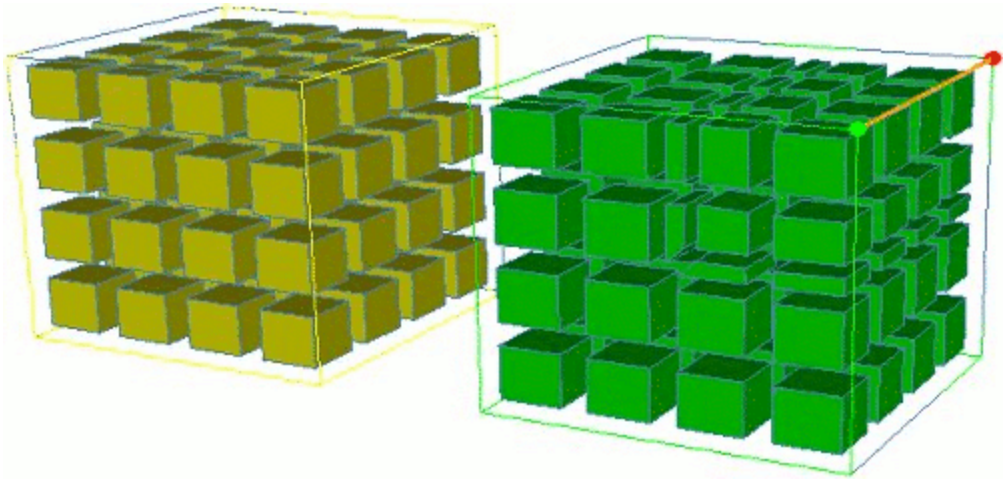


Figure 4. Example of Refinement at a curve

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 it's 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 volumeâ€™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.

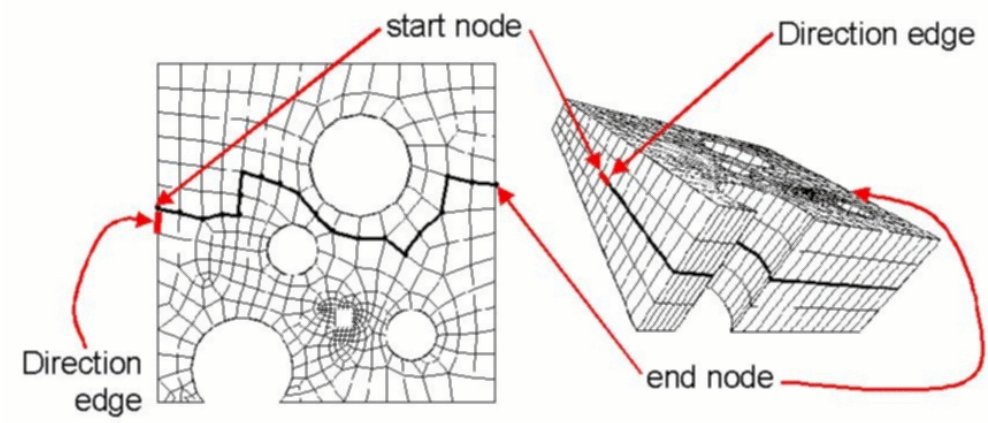


Figure 5. Refining a Mesh Along a Path

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).

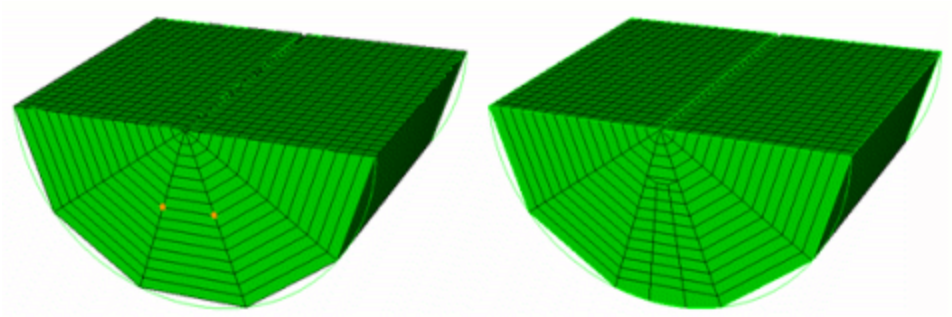


Figure 6. Refine mesh sheet node 796 782 greater_than 6

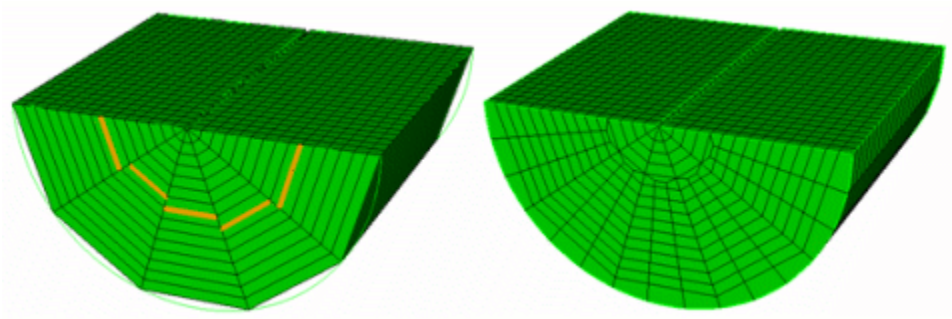


Figure 7. Refine mesh sheet edge 1584 1564 1533 1502 1471 greater_than 6

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.

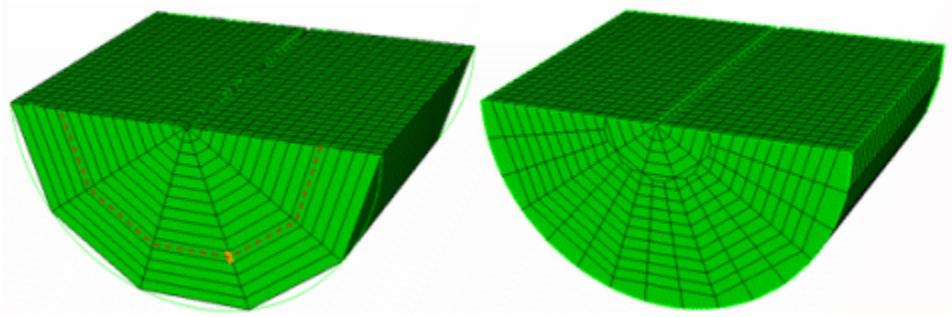


Figure 8. Refine mesh sheet intersect edge 1499 greater_than 6

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.

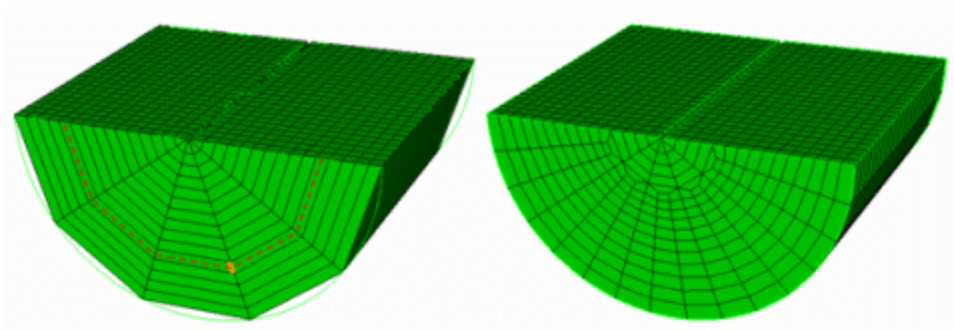


Figure 9. Refine mesh sheet intersect edge 1499 greater_than 6 smooth

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

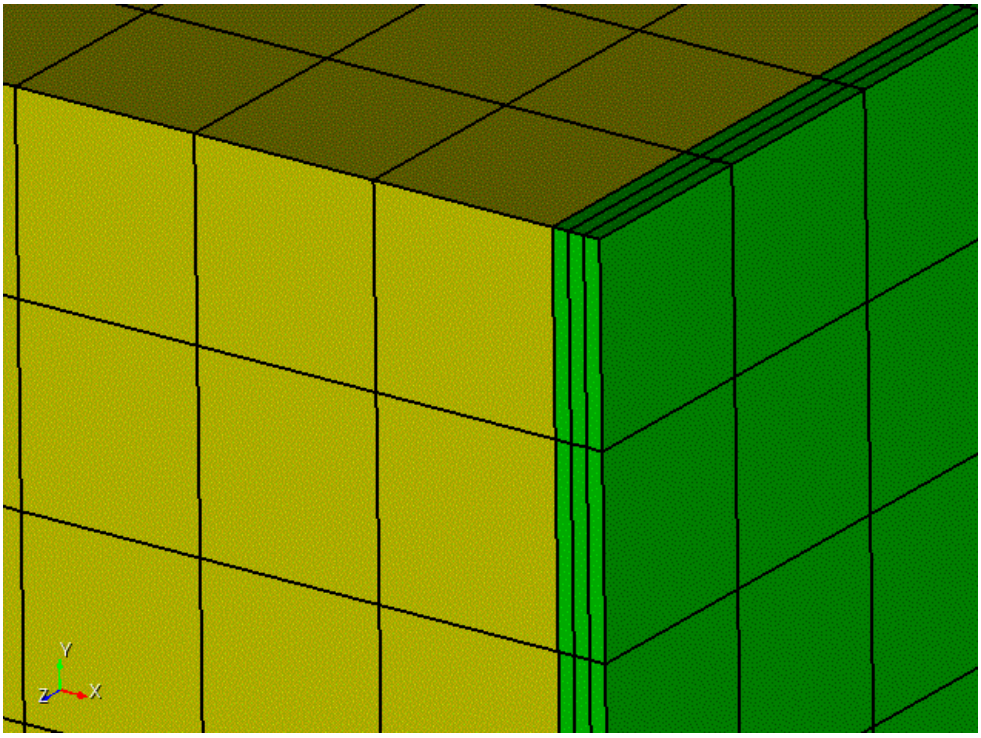


Figure 10. Starting mesh

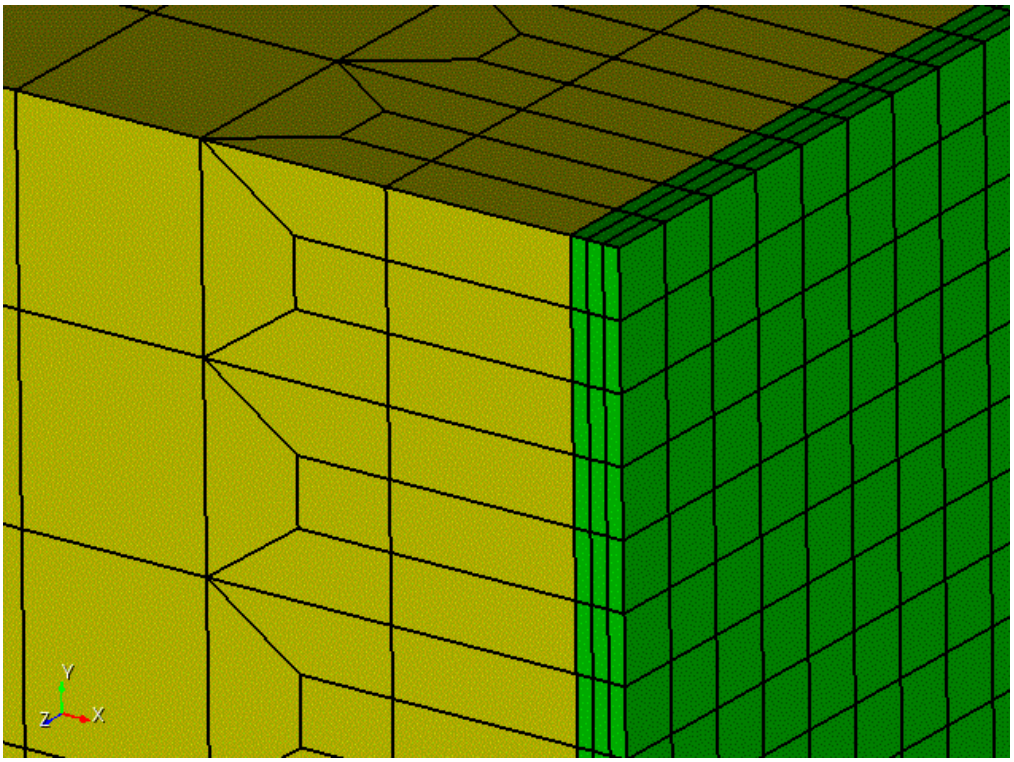


Figure 11. Directional result of refinement resulting from using the plane option on the refinement command.

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
```

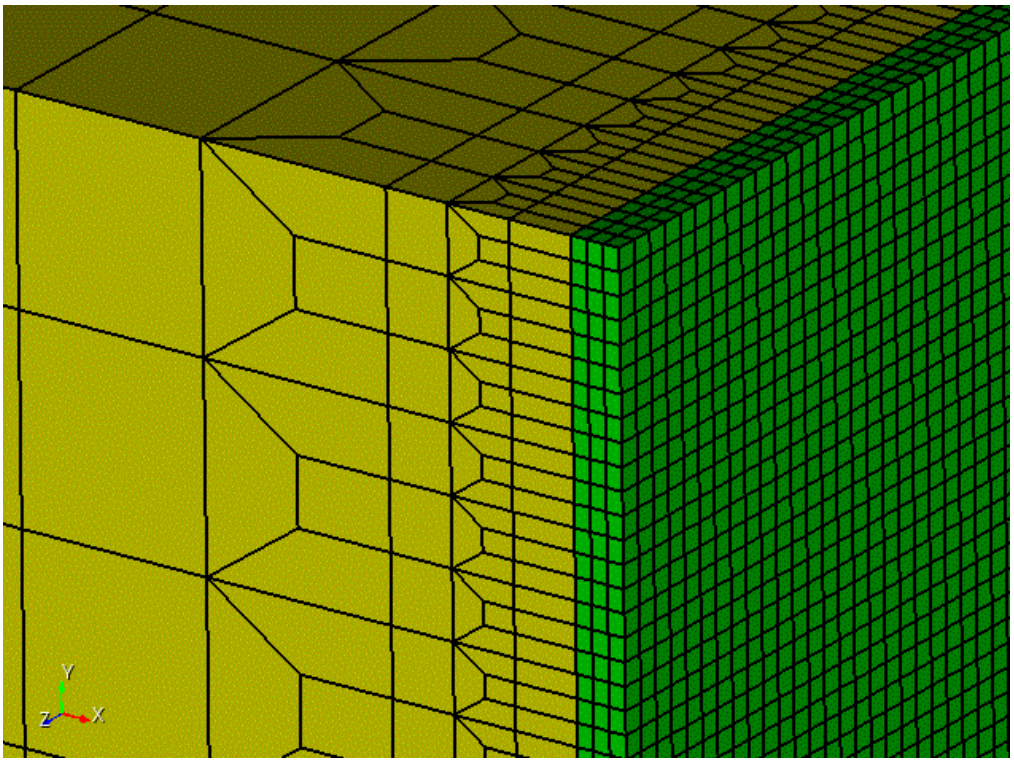


Figure 12. A 2nd iteration of direction refinement is applied.

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 â€˜anisotropicâ€™ 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.

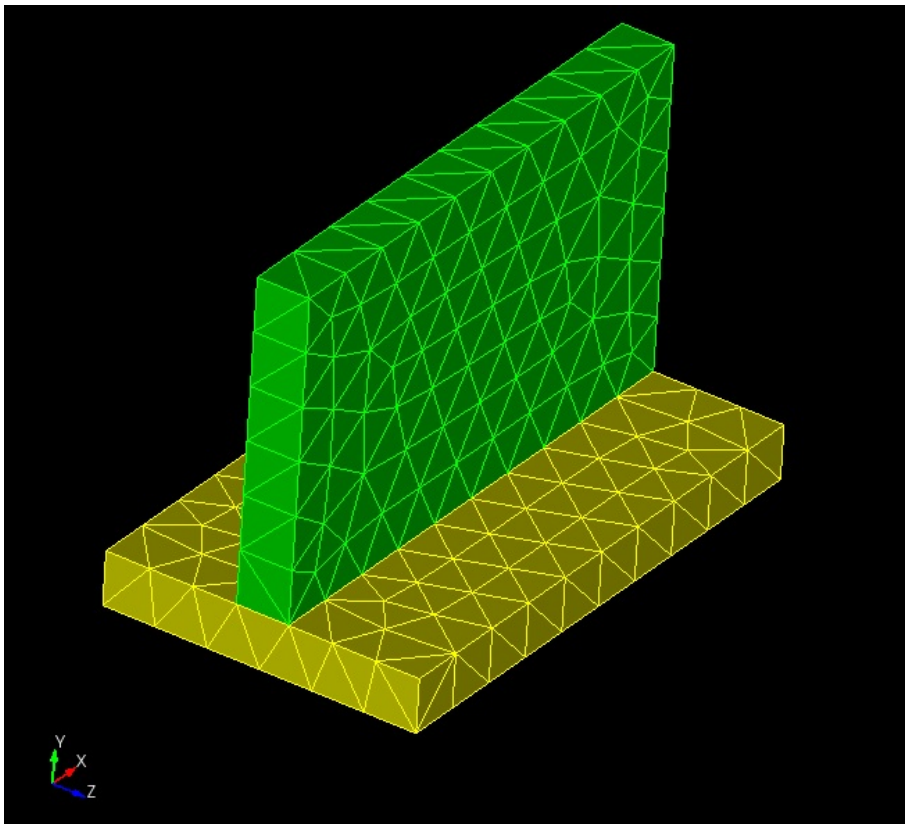


Figure 13. Before N through the thickness refinement.

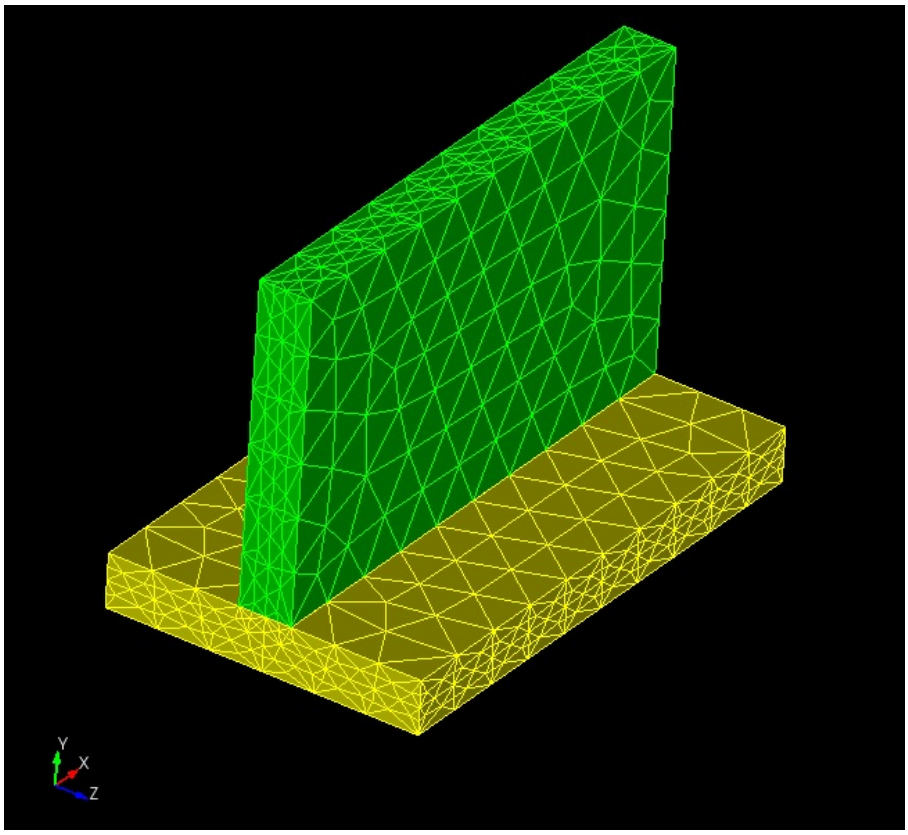


Figure 14. After N through the thickness refinement.

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 â€˜headâ€™, â€˜4.22.0â€™, 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 *â€˜somemeshâ€™* 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/**.**

Scaling the Number of Elements in a Mesh

For hexahedral meshes only, Mesh Scaling is another tool to globally increase the number of elements in your mesh. Mesh Scaling gives the user more control than the refinement commands over how much denser the resulting mesh will be. The refinement commands described above require a *numsplit* argument. During refinement, each edge in the mesh will get split either by 1/2 or 1/3 times the *numsplit* parameter. The result is that when refining globally, the element count of the mesh will increase by 8X, 64X, etc. for a dramatic increase in the element count.

In contrast, Mesh Scaling will allow the user to increase the element count of mesh by any floating point multiplier. The command syntax is:

Scale Mesh {multiplier <double> [minimum <int>] | add_interval <int>}

Given a mesh composed of hexahedral elements, Scale Mesh will increase the number of elements in the mesh by an amount equal to or slightly greater than **multiplier**. For example:

bri x 10

mesh vol all

will result in a mesh with 1000 elements. The command:

scale mesh multiplier 2.5

will increase the number of elements in the mesh to just over 2500. If instead of scaling the mesh, the user issued the command:

refine vol all

the result would have been a mesh ith 8000 elements. Thus, we see that mesh scaling allows significantly more control over the element count increase in the mesh.

Internal to Cubit, Mesh Scaling works by extracting the block decomposition of the hexahedral mesh. The block decomposition is a decomposition of the mesh into groups of elements such that each group is a local mapped block of elements. For example, Figure 15 illustrates an example input mesh for Mesh Scaling. Figure 16 illustrates the resulting block decomposition, and Figure 17 illustrates the final scaled mesh with a multiplier of 3.0.

Figure 15. Input mesh for Mesh Scaling. The mesh contains 84 hex elements.

Figure 16. The block decomposition of the mesh from Figure 15.

Figure 17. The mesh resulting from scaling the mesh in Figure 15 with a multiplier of 3.0. The mesh contains 260 elements (84 X 3.0 = 252 <= 260).

Multiplier values greater than 1.0 result in elements added to the mesh. A multiplier of 1.0 results in now change to the mesh. A multiplier of less than 1.0 results in global coarsening of the mesh.

Each block in the decomposition is locally a mapped mesh of elements. The number of elements in each local block in the decomposition is NxMxP, where N, M, and P are the interval counts for the 3 dimensions of the mapped block. The **minimum** argument on the Scale Mesh command specifies a minimum increase in N, M, & P for each block. Scale mesh will refine the mesh as close to the input multiplier, but with a minimum increase of {N+minimum}x{O+minimum}x{P+minimum} in each block. Optionally the **add_interval** argument can be specified resulting in an exact element count increase of {N+add_interval}x{O+add_interval}x{P+add_interval} in each block in the decomposition.

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.

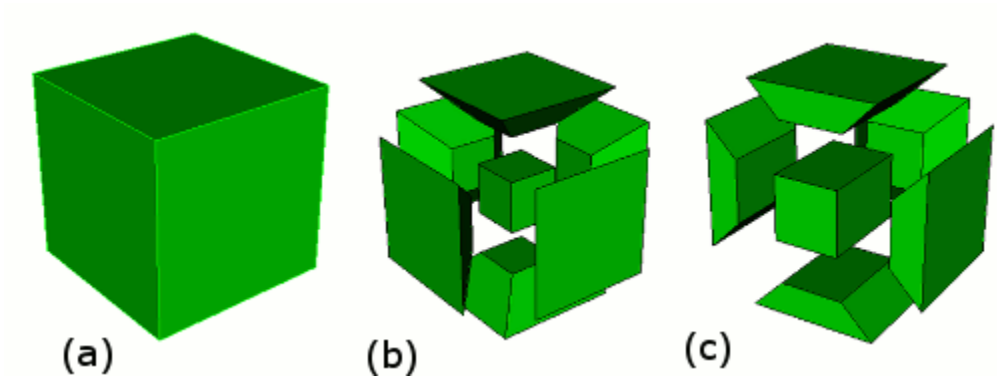


Figure 1: A single hex before (a) and after (b) a pillow operation. The far right (c) depicts a pillow operation with the front surface designated as a 'through' surface.

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]

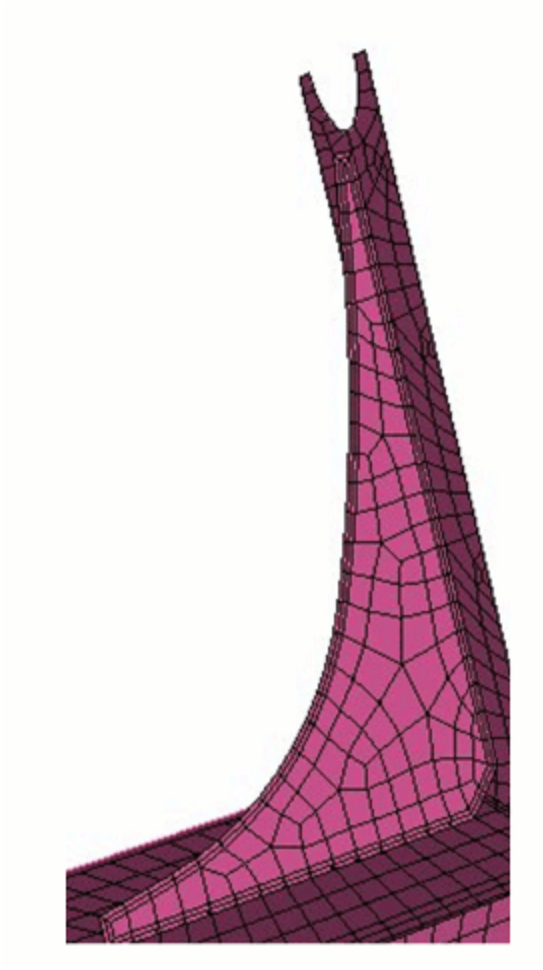


Figure 2: Example model using pillow operations to create ordered nodes a specified distance around the boundary of a mesh.

Mesh 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.

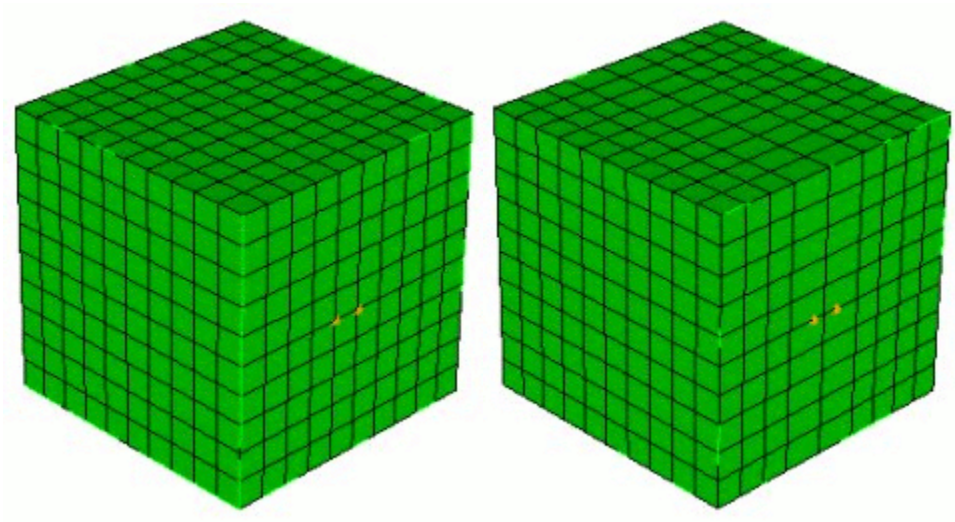


Figure 3. Example of Hex Sheet Extraction

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]

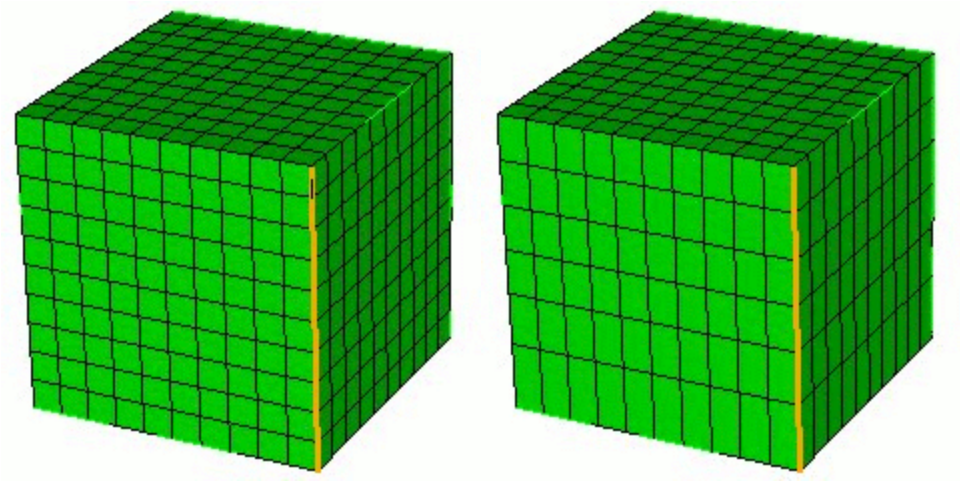


Figure 4. Coarsening a mesh by extracting sheets perpendicular to a curve

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.

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>

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.

Collapsing Mesh Edges

CUBIT currently offers several options for modifying an existing finite element mesh. In addition to providing for coarsening and refining of hexahedral and triangle meshes, CUBIT can also reposition nodes by smoothing or by moving individual nodes.

The collapse edge command is also provided for making small modifications to an existing triangle mesh.

Meshedit Collapse Edge <id>

This command will collapse the two triangles associated with the given edge, effectively removing the triangles from the mesh. This command only works on surface meshes, and only with triangles. If volumetric elements, or quads, are attached to the edge, the command does nothing to the mesh.

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
```

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

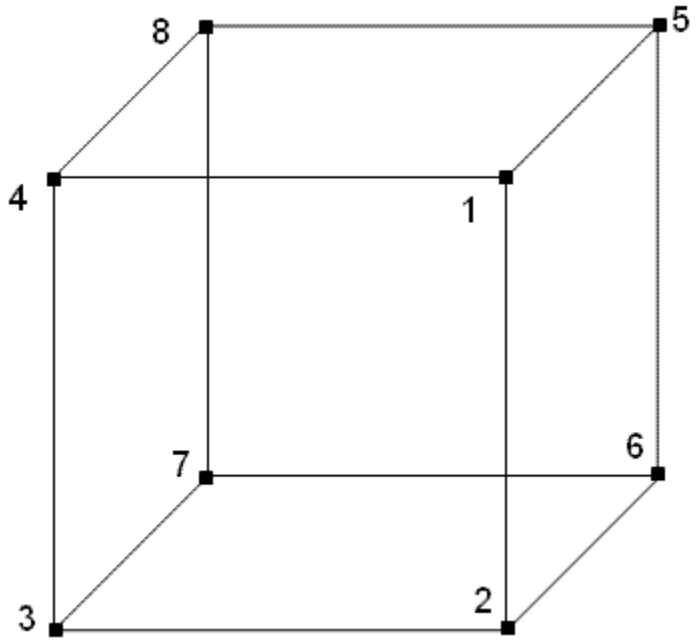


Figure 1. Node Numbering for the Create Hex command

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

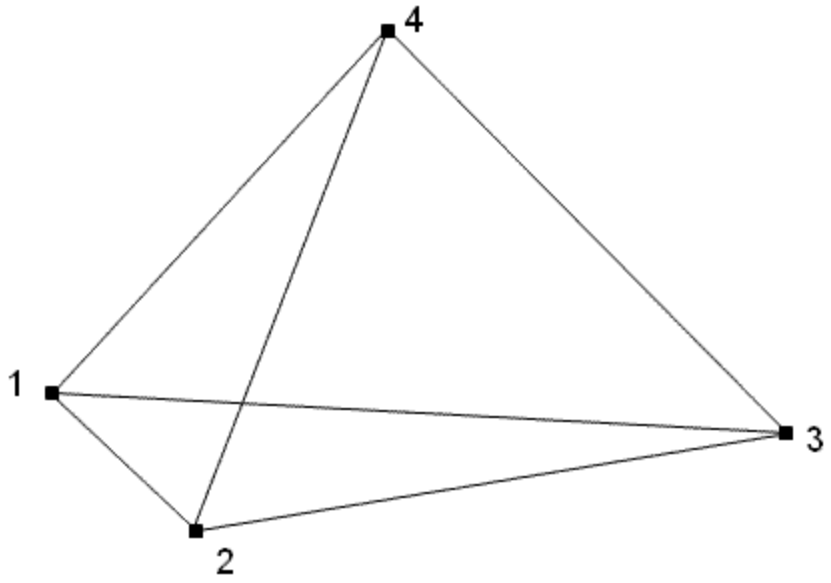


Figure 2. Node ordering for Create Tet Command

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

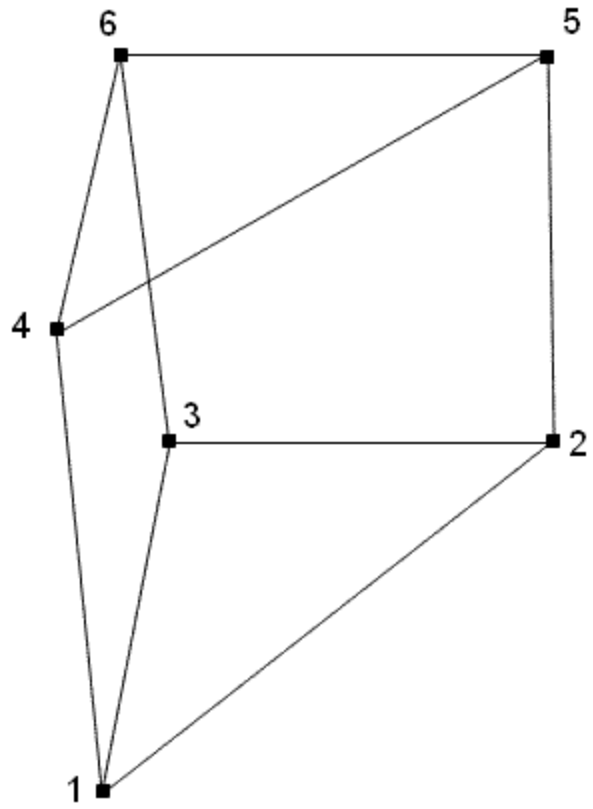


Figure 3. Node ordering for Create Wedge Command

Note: The wedge command is still under development. To enable this feature, use the developer command features by issuing the command **Set Developer Commands On**.

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.

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.

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.

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.

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

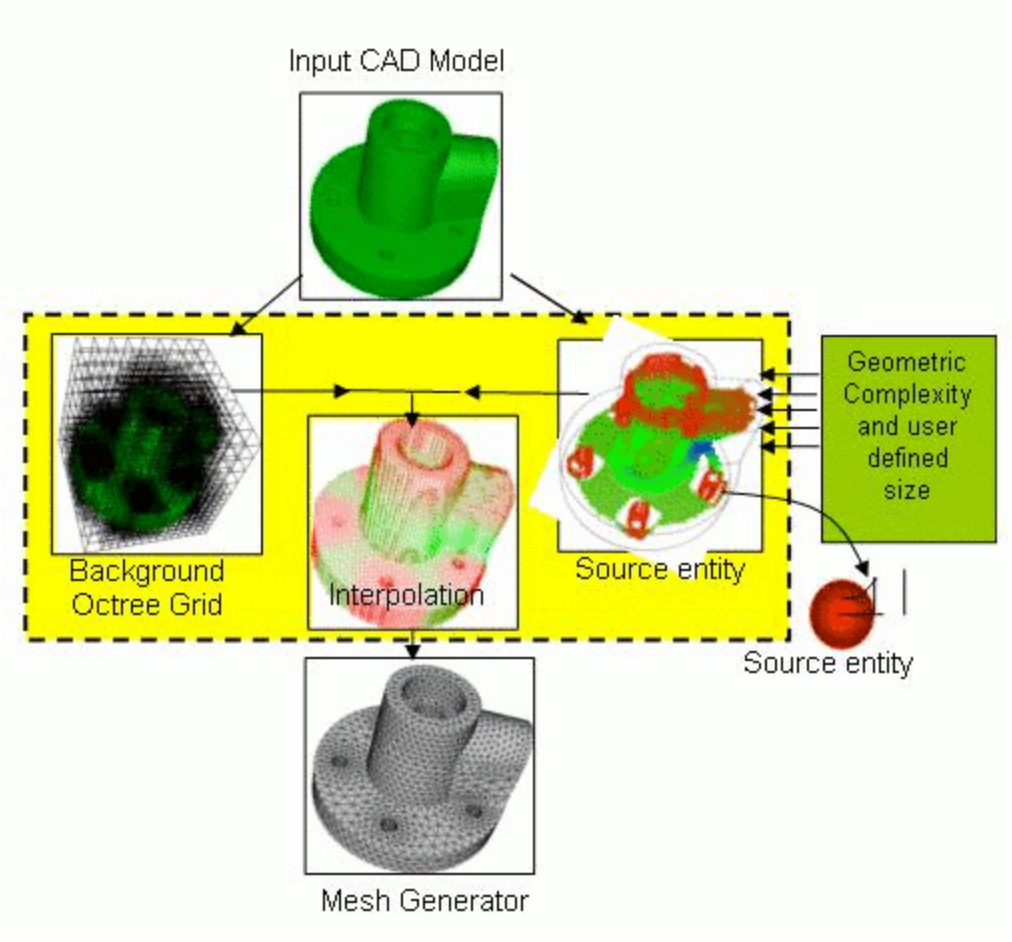


Figure 1: Overview of Computational Framework

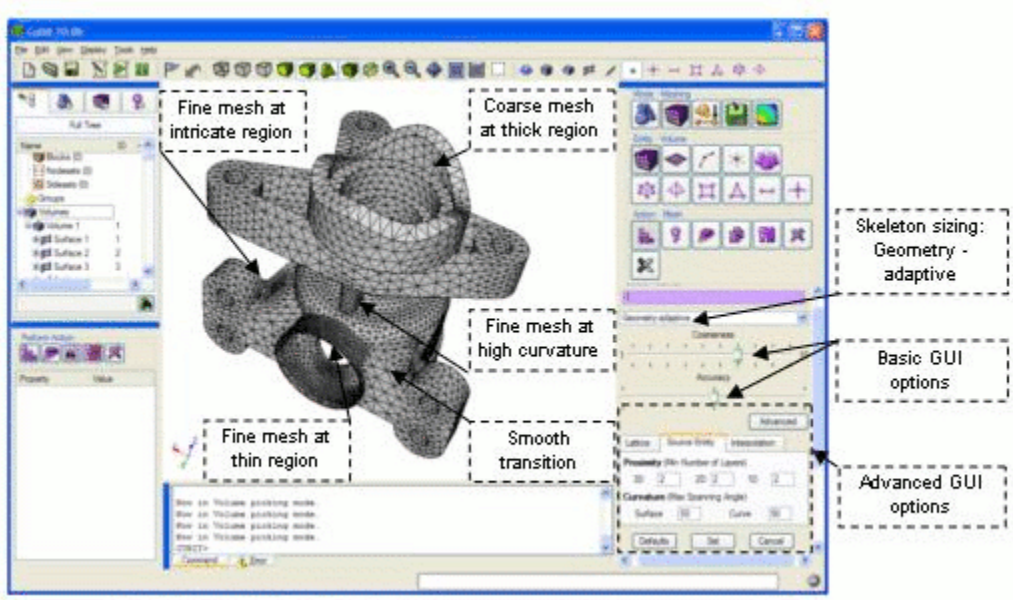


Figure 2: Skeleton Sizing Function example in the GUI

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.

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.

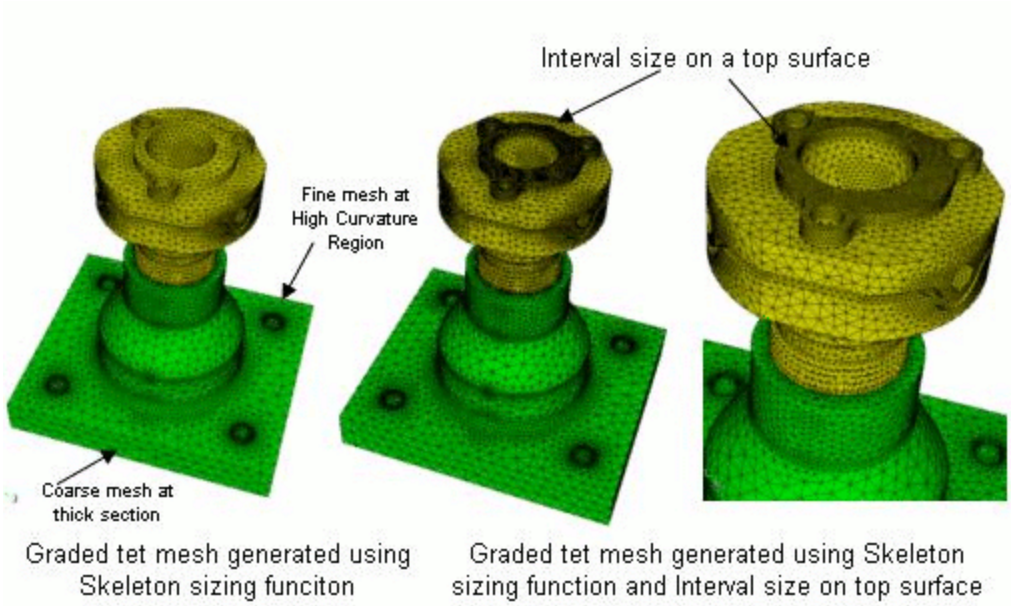


Figure 3: Skeleton sizing function with other sizing controls

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”

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
```

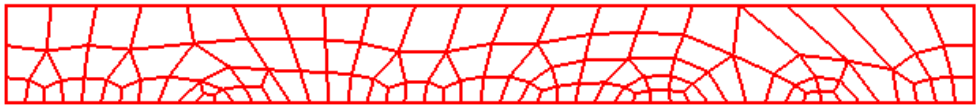


Figure 1. Surface with bias sizing function factor > 1.

```
# mesh 2
delete mesh
surface 1 sizing function type bias start curve 4 factor {1/1.1}
mesh surface 1
# see figure 2
```

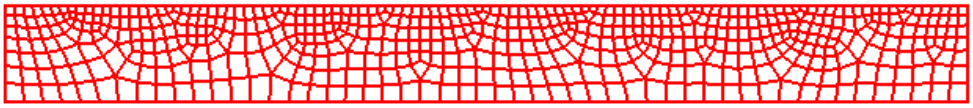


Figure 2. Surface with bias sizing function factor < 1

```
# 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
```

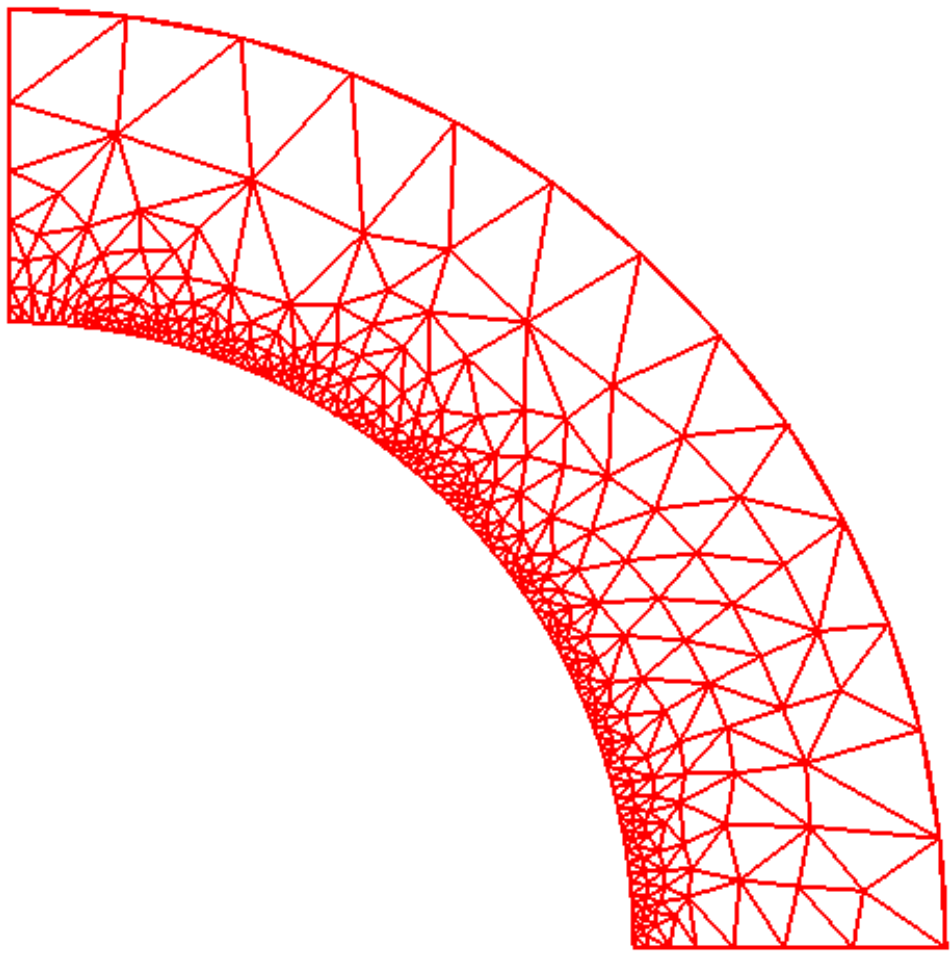


Figure 3. Surface with bias sizing function start and finish curve. Scheme qtri, base scheme pave.

```
# 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
```

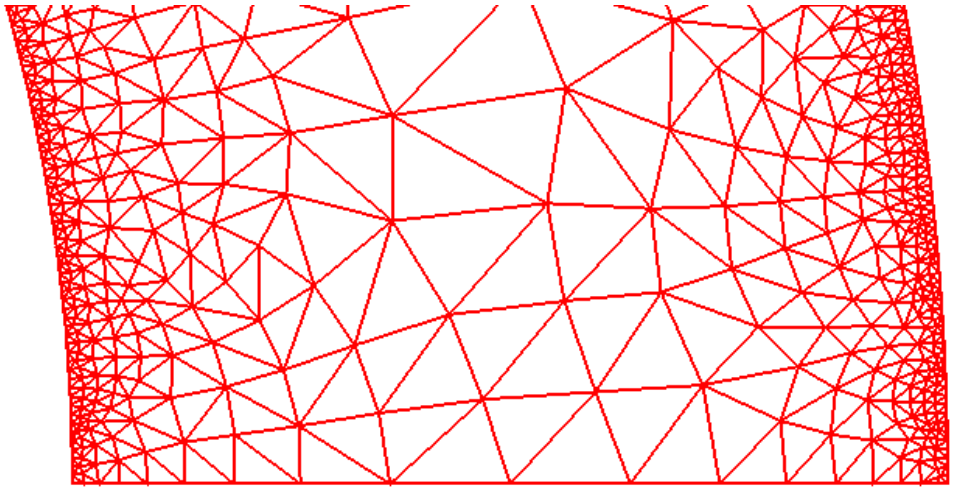


Figure 4. Close up of surface with dual bias sizing function start and finish curve. Scheme qtri, base scheme pave.

```
# 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
```

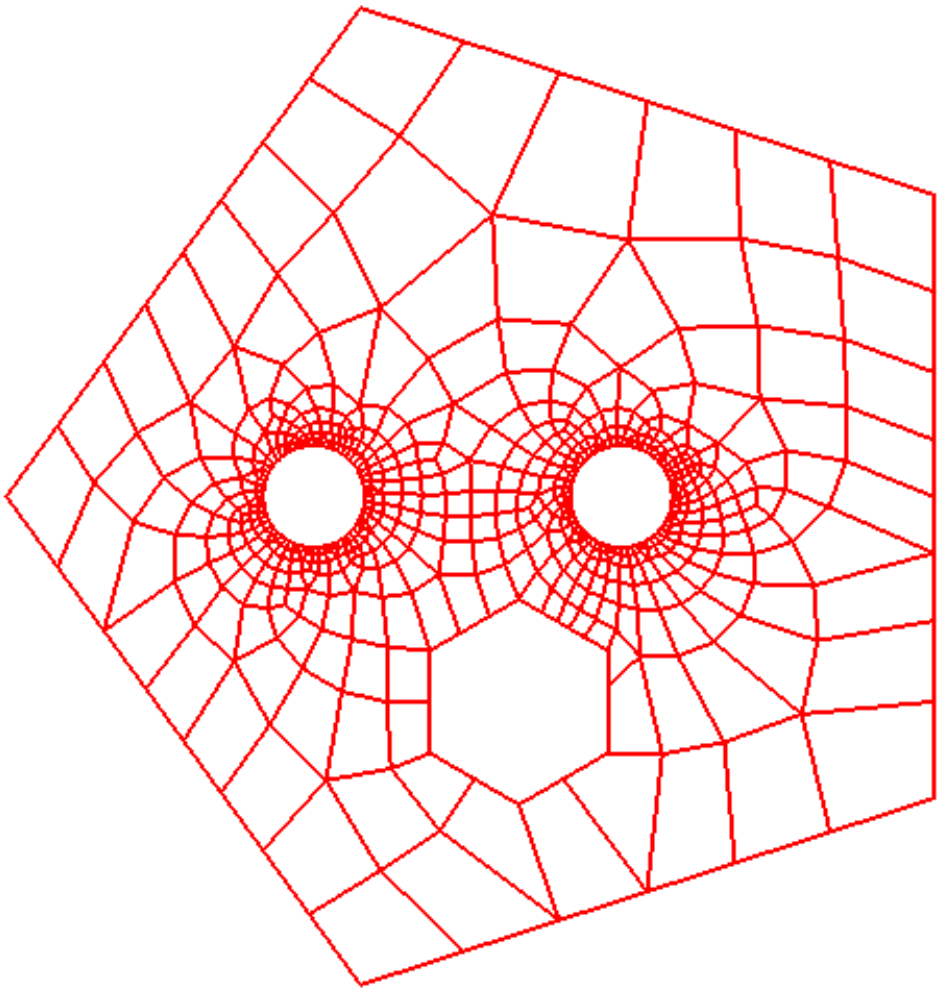


Figure 5. Bias away from two round holes.

```
# 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
```

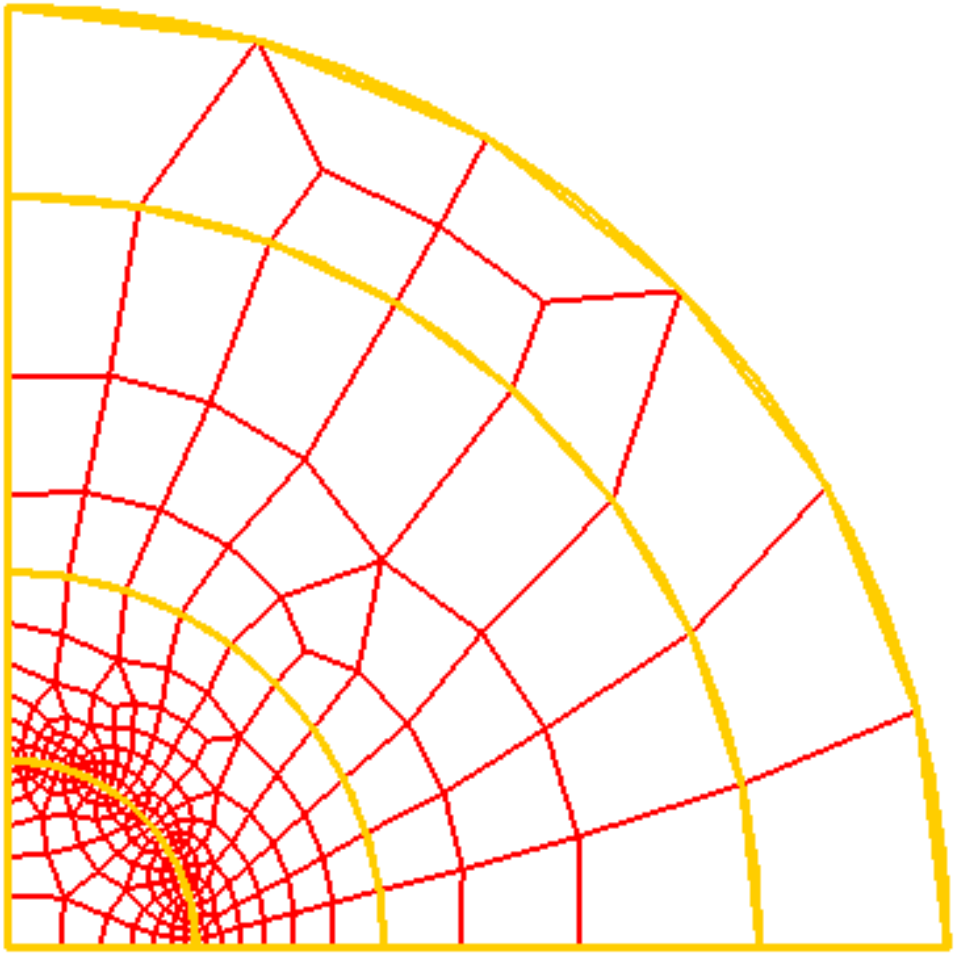


Figure 6. A chain of biased surfaces. Only one curve's intervals were explicitly set.

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

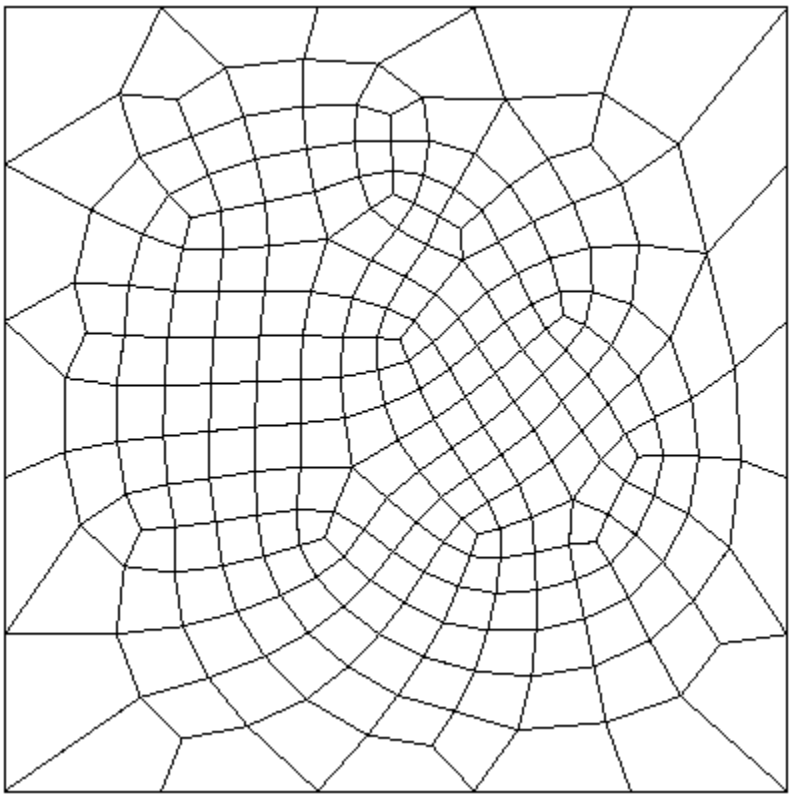


Figure 1. Constant Sizing Function

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.

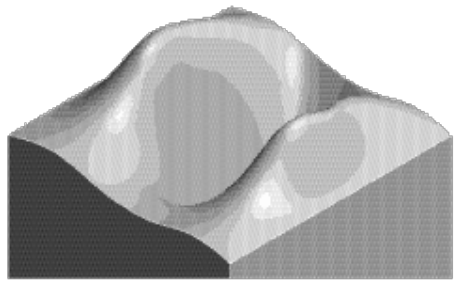


Figure 1. NURB solid with high surface curvature change

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.

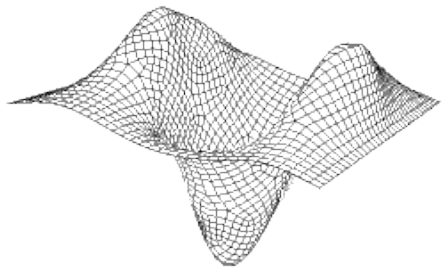


Figure 2. NURB mesh with no interior sizing function

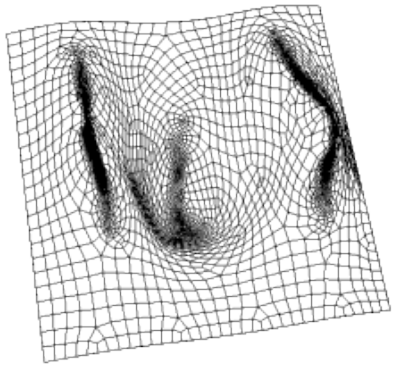


Figure 3. NURB mesh with curvature sizing function

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.

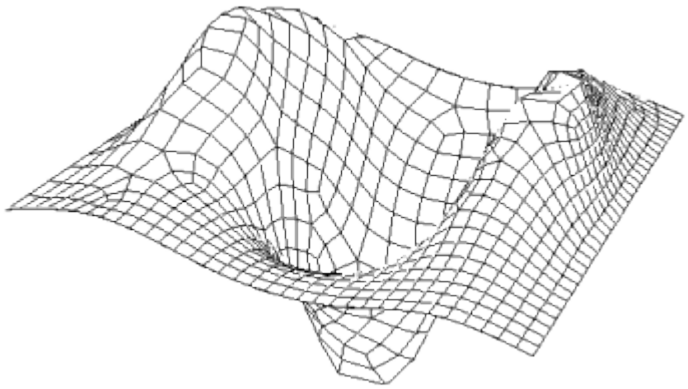
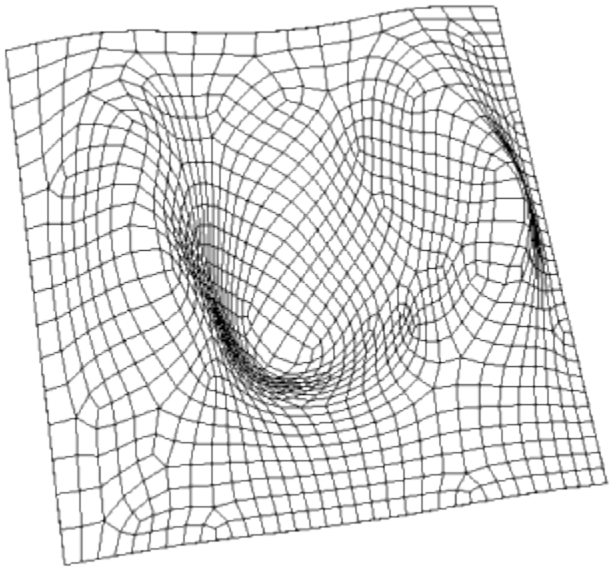


Figure 1. NURB mesh with no sizing function, 34 by 16 density



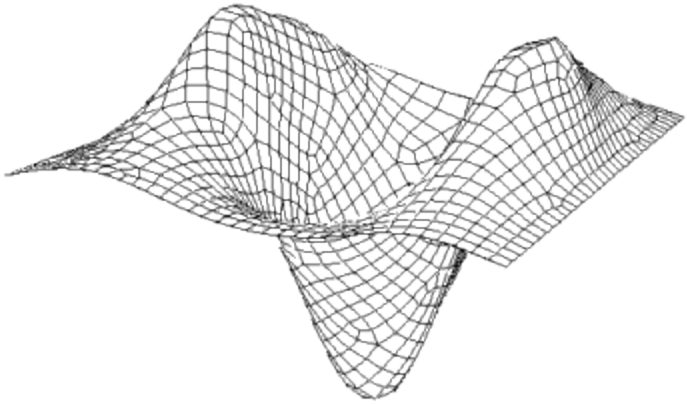


Figure 2. NURB mesh with linear sizing function, 34 by 16 density

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.

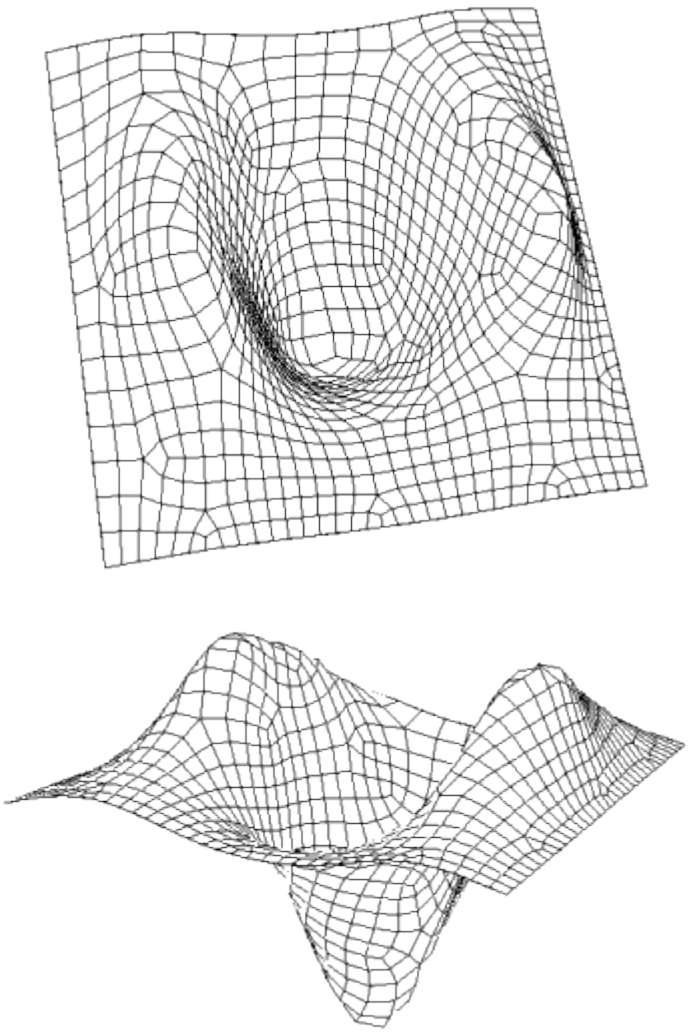


Figure 1. NURB mesh with interval sizing function, 34 by 16 density

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.

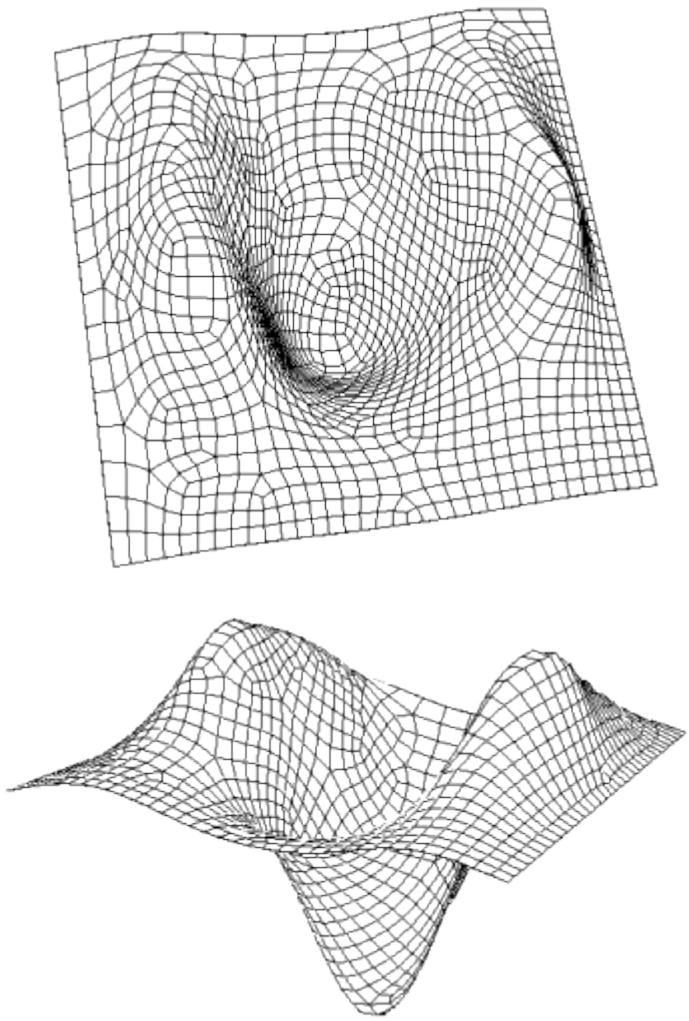


Figure 1. NURB mesh with inverse sizing function, 34 by 16 density

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, but only tetrahedral elements are supported for volumes at this time.

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 TriAdvance, and/or volume mesh scheme set to Tetmesh. Bounding curve mesh schemes can also optionally be set to Stride (see comments below.)
2. An Exodus mesh and time-dependent variable for that mesh is read into CUBIT.
3. The mesh and variable data are associated to geometry.
4. The Exodus variable is normalized to give localized size measures, and the surface/volume sizing function type is designated.
5. Geometry is meshed

Importing a field function and associating it with its geometry, and normalizing that function are done in two separate steps to allow renormalization. The following command is used to read in a field function and its associated mesh:

> Import Sizing Function '<exodusII_filename>' Block <block_id> Variable `<variable_name>' Time <time_val> [Deformed]

where **block_id** is the element block to be read, **variable_name** is the Exodus time-dependent variable name (either element-based or nodal-based),and **time_val** is the problem time at which the data is to be read. The **Deformed** keyword indicates whether deformation has been accounted for on the new model (for information on creating deformed 2D geometry from EXODUSII data, see Importing 2D EXODUSII Files) and needs to be accounted for in the sizing function data. When this command is given, the nodes and elements for that element block are read in and associated to geometry already initialized in CUBIT.

Note that when a sizing function is read in, the mesh is stored in an ExodusMesh object for the corresponding geometry, and therefore the geometry is not considered meshed. Also note that if deformation is not being modeled, the geometry to which the mesh is being associated must 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 and assigned to geometry, it can be normalized before being used to generate a mesh. The normalization parameters are specified in the same command that is used to specify the sizing function type for the surface or volume. The syntax of this command is:

> Surface <id> Sizing Function Type Exodus [Min <min_val> Max <max_val>]

> Volume <id> Sizing Function Type Exodus [Min <min_val> Max <max_val>]

If normalization parameters are specified, the field function will be normalized so that its range falls between the minimum and maximum values input. Subsequent normalizations operate on the normalized data and not on the original data. If an element-based variable is used for the sizing function, each node is assigned a sizing function that is the average of variables on all elements connected to that node. Nodal variables are used directly.

After the sizing function normalization, the geometry may be meshed using the normal meshing command.

For example, the left image in Figure 1 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.

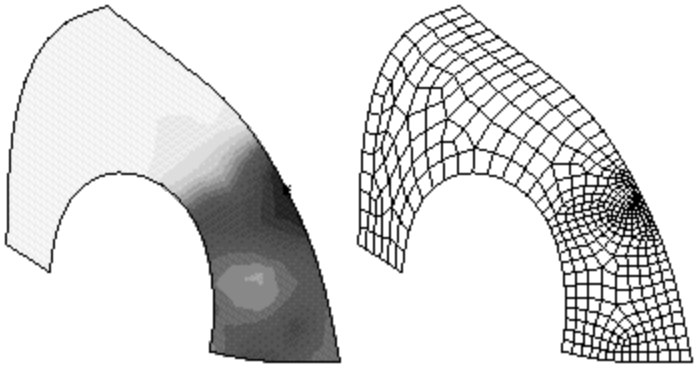


Figure 1. Plastic strain metric and the adaptively generated mesh

Curve Meshing with Exodus II - based Field Functions

In addition to the capability to adaptively mesh surface using a field function, curves may also be meshed separately using the Exodus II information. The Stride scheme for meshing curves is used for this purpose. If the user does not specify a mesh scheme for the curve, Cubit will default to scheme Stride when the Exodus sizing function is used for surfaces and volumes defined by that curve.

Importing Exodus II Files

- Importing a Free Mesh without Geometry
- Importing a Free Mesh onto Existing Geometry
- Creating Mesh-based Geometry on Import

The commands to import meshes from an Exodus II format file are:

> Import Mesh '<exodusII_filename>' [Block <block_ids>] [Unique Genesis IDs] [Shell] No_Geom [group_name '<free_mesh_group_name>']] [[Time <time>|Step <step>|Last] [Scale <value>]]
>
> Import Mesh '<exodusII_filename>' [Block <block_ids>] [Unique Genesis IDs] [Shell] [{Group|Body|Volume|Surface|Curve|Vertex} <id_range> | Preview]
>
> 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] [Deformed {Time <time>|Step <step>|Last} [Scale <value>]] [MERGE|No_Merge] [Export_facets <1|2|3>] [Merge_nodes <tolerance>]

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

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>' [Block <block_ids>] [Unique Genesis IDs] [Shell] No_Geom [group_name '<free_mesh_group_name>']] [[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.

Deformation information can be read in via the **Time/Step/Last** and **Scale** parameters.

Note: The **Import Mesh [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 [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>' [Block <block_ids>] [Unique Genesis IDs] [Shell]
[{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.

Unique Genesis IDs and Shell Options

The Unique Genesis IDs option is used to preserve ids in the genesis file in the case that id overlap exists when importing into CUBIT. This can occur when importing into an active session where CUBIT ids have already been assigned.

The Shell Option is used as a flag to alert the program that there are shell elements in the file. Shell elements can not always be detected by the import program, and this ensures that the shell elements will be included in the model.

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] [Unique Genesis IDs] [Start_id <id>] [Use [NODESET|no_nodeset]
[SIDESET|no_sideset] [Feature_Angle <angle>] [LINEAR|Gradient|Quadratic|Spline] [Deformed
{Time <time>|Step <step>|Last} [Scale <value>] ] [MERGE|No_Merge] [Export_facets <1|2|3>]
[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.

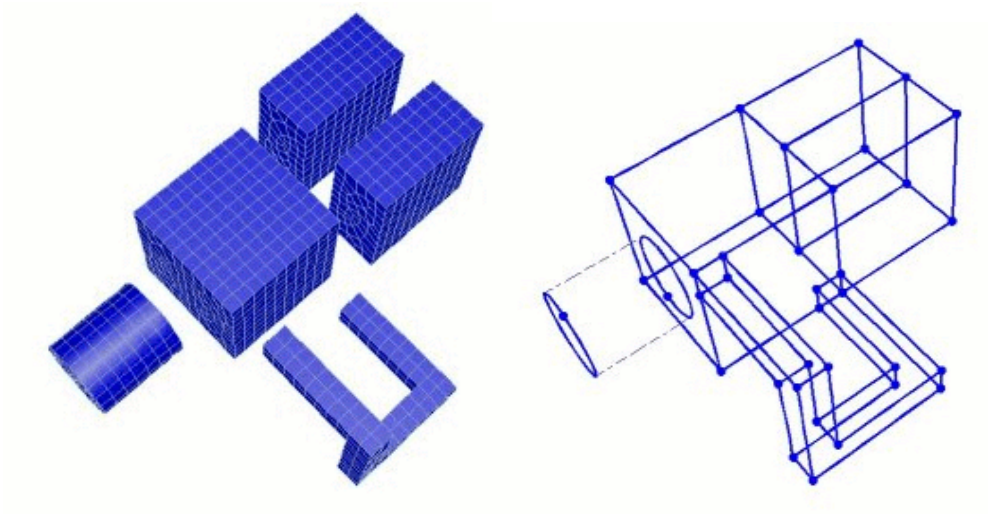


Figure 1. Example of mesh based geometry (right) created from a finite element mesh (left)

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.

Unique Genesis IDs

The **Unique Genesis IDs** option is used to preserve ids in the Cubit session in the case that id overlap exists when importing an Exodus II file. This can occur when importing into an active session where ids for blocks, nodesets or sidesets have already been assigned. The default behavior, when ID collisions occur, is to include any new entity into the existing block nodeset or sideset. If the **Unique Genesis IDs** option is used, Cubit will automatically generate a unique ID for any block, nodeset or sideset imported. A report of the collisions and their new IDs will be displayed on import in the command window.

Start ID

Use this option to specify an alternate ID value for imported mesh entities. The specified value will be used as the starting ID for BOTH nodes and mesh elements. The new IDs will be assigned consecutively from the starting value. If the new ID values for any of the imported entities would conflict with existing IDs, the command does not abort but moves the starting ID for all element types to the same useable starting ID value.

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.

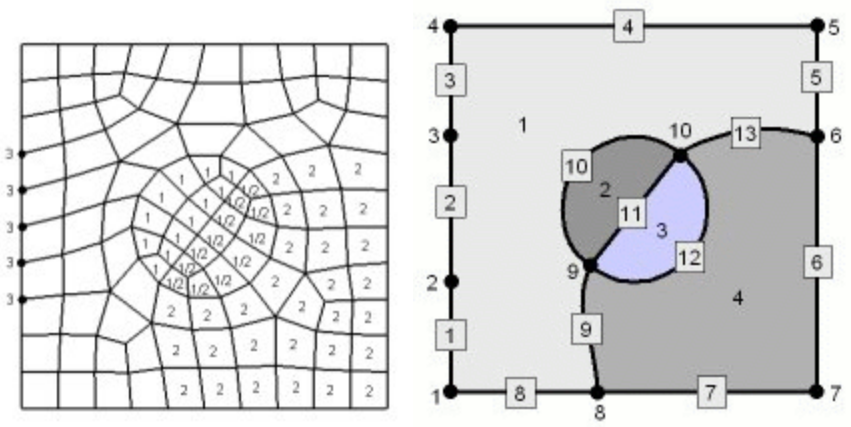


Figure 2. Example of geometry created from mesh entities assigned to nodesets (3) and sidesets (1 and 2).

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.

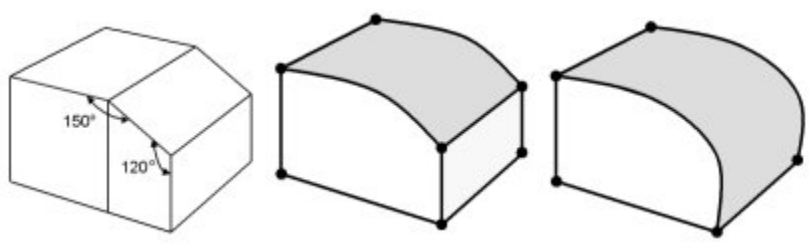


Figure 3. Example use of Feature Angle

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.

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.

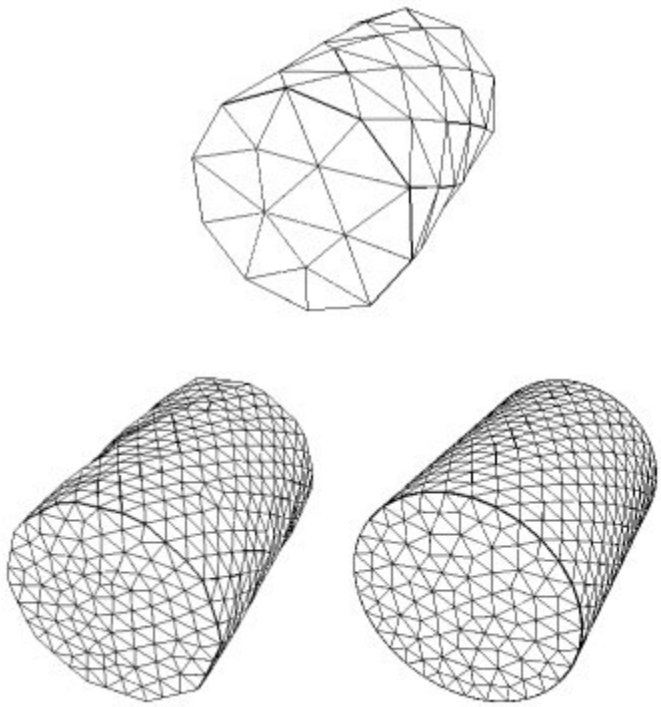


Figure 4. Effect of Smooth Curve and Surface Option for remeshing of mesh-based geometry

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]

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.

Apply Deformations

This 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.

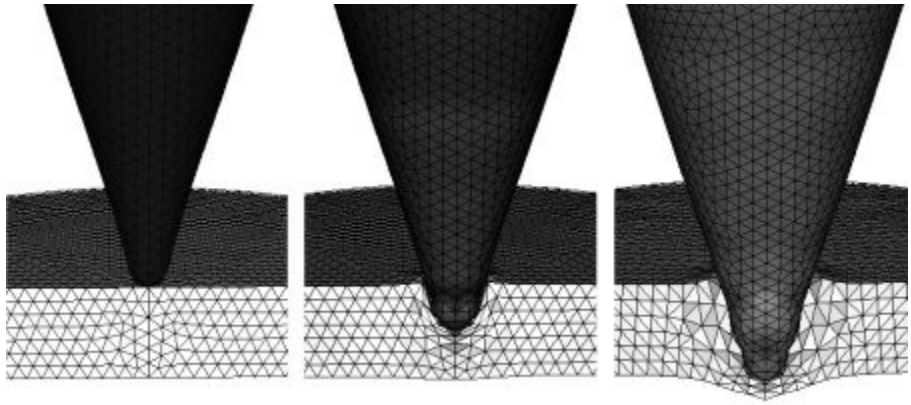


Figure 5. Example of remeshing of a deformed finite element mesh

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.

Export Facets

[export_facets <1|2|3>]

This is primarily a debug option available only from the command line. This option will export the shell of the Exodus mesh to an ASCII file in the form of facets. The resulting file can be imported to Cubit using the "Import Facets" command. Export options: 1 = export only the exterior facets to file "facets.shell"; 2 = export only the interior facets between element blocks to file "facets.inter"; 3 = export all boundary facets to file "facets.all".

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 CAX3 CPE3	STRI65 CAX6 CPE6
Quadrilateral	S4 CAX4 CPE4	S8 CAX8 CPE8
Tetrahedron	C3D4	C3D10
Hexahedron	C3D8	C3D20
Line Element	B21 B31 T2D2 T3D2 SPRINGA SPRING1 SPRING2	B22 B32 T2D3 T3D3

See http://www.simulia.com/ for more information on the ABAQUS file format.

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 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.

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.

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]

Free Meshes

A free mesh is a mesh that is not associated with any underlying geometric entities. A free mesh contains only mesh elements (hexahedrons, 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 not require the extra import/export step. So 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> [Feature_Angle <angle=135>] [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 warning and enable the keep option if the mesh is owned. The keep option is not specified by default.

Also note that any genesis entites defined on the free mesh will be maintained with this option. The genesis entities will not however be transfered to the new MBG entities and will not be used as criteria for building the new MBG geometry. Other options such as creating a spline representation and building geometry from genesis entities are not supported in this command. Exporting the free mesh and reimporting using "import mesh geometry" may be an option if these features are desired.

Merging a free mesh

To merge two free meshes, the equivalence command may be used. The command syntax is:

Equivalence Node <range> [Tolerance <value>]

All nodes in the given range that are within the specified tolerance will be merged. 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. A direction or curve can be used to specify how the elements are created. The **distance** parameter is optional and if not specified the length of the given direction will be used instead. Specifying a value for the **layers** option determines how many elements will be created in the given distance. **Twist** can also be specified and requires an angle of twist and a twist axis.

Create Element Extrude {Node|Edge|Face|Tri} <element_list> {Direction **<options>**|Along Curve <curve_list>} [Distance <value>] [Layers <num_layers] [Twist <angle> Axis <**axis_options**>]

#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

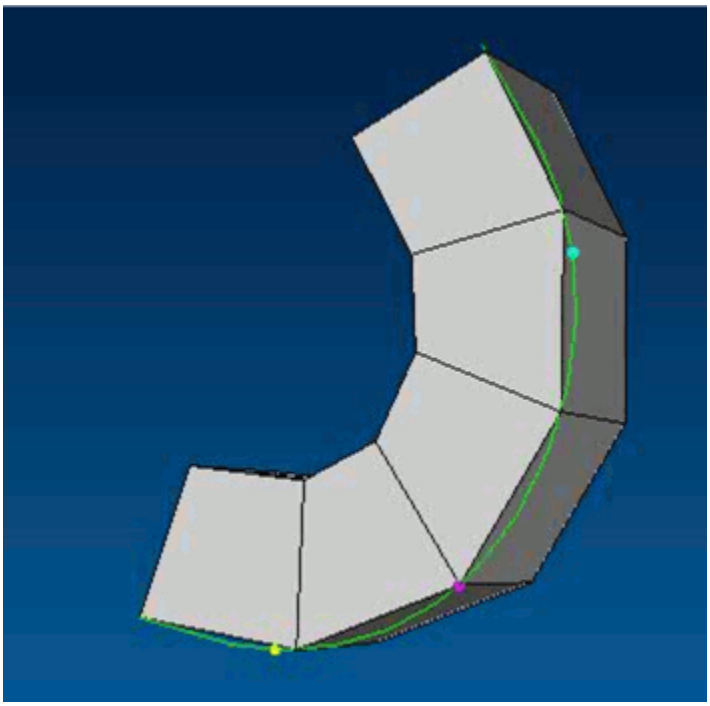


Figure 1. Extruding mesh elements along a spline

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.

> Create Element Revolve {Edge|Face|Tri} <element_list> Axis <**axis_options**> Angle <angle> [Layers <num_layers>] [Attemp_fix]

#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 0 1 0 angle 180 layers 4 attempt_fix

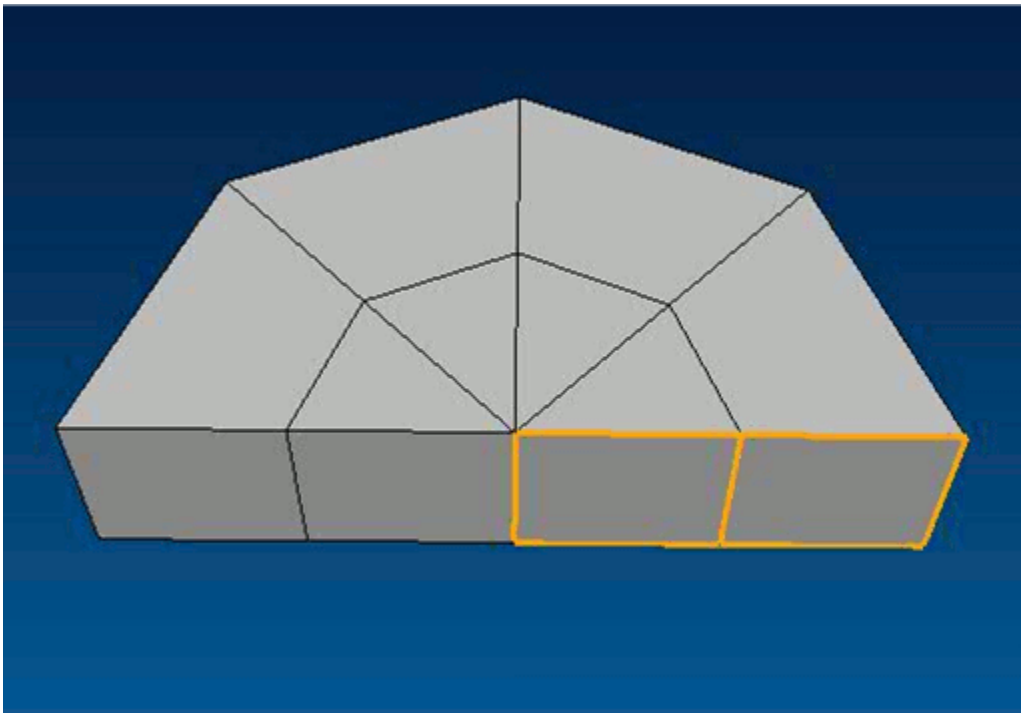


Figure 2. Revolving free mesh elements to create hex and wedge elements

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

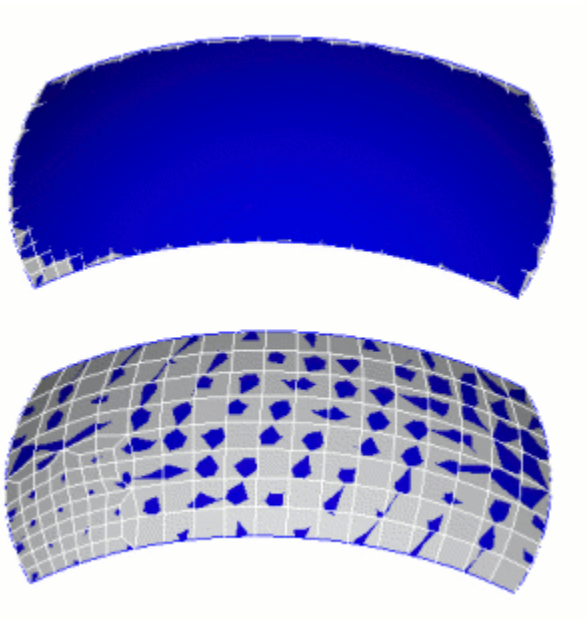


Figure 3. Smoothing without a target (above) and smoothing to a target surface (below).

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

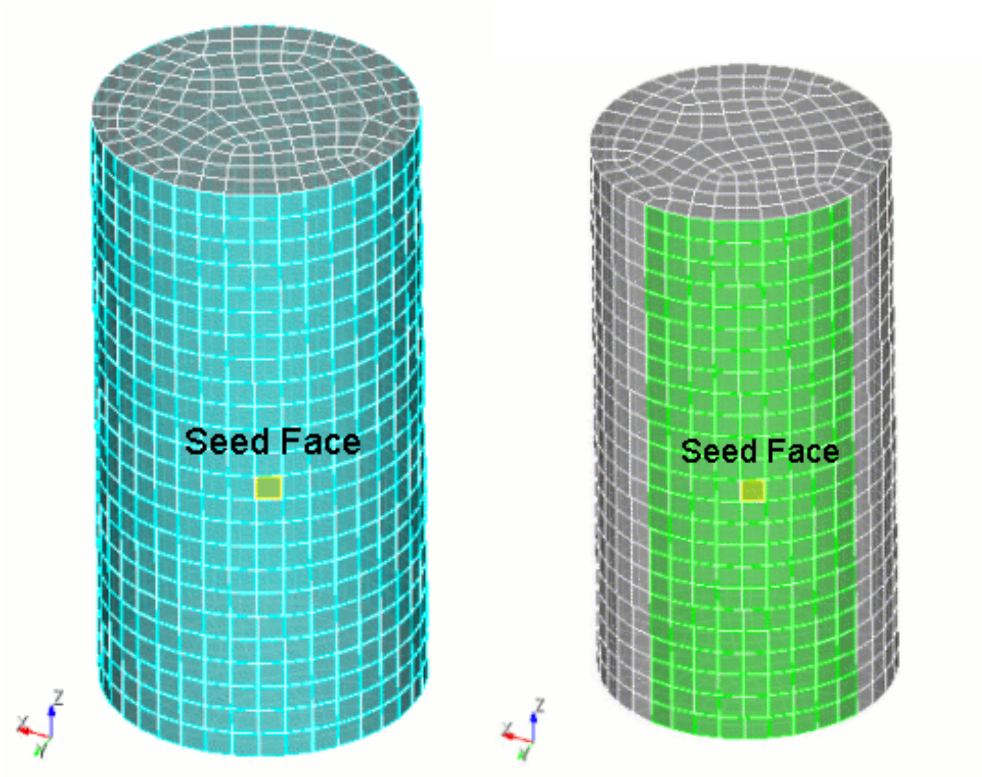


Figure 4. Grouping faces on free meshes using the seed method. The feature angle method is used on the left with a feature angle of 45 degrees. On the right is the result if using the divergence method.

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).

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.

Element Block Specification

- Creating Blocks
- Assigning a Name or Description to an Element Block
- Defining the Element Type
- 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

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> {Vertex | Curve | Surface | Volume} <range> [Remove]

Block <block_id> {Hex|Tet|Pyramid|Face|Tri|Edge|Node} <range> [Remove]

Block <block_id> Group <range> [Remove]

The first command defines the block based on a list of geometric entities, while the second uses specific lists of mesh entities. Since a block can only contain a single element type, usually entities of the same type are defined on the same block. The third option provides for assigning **groups** of entities to a single block. This is useful, for example, when several entities of the same type can be grouped together. The **Block Group** command simplifies the specification of the block.

By using the **Remove** argument to the **block** command, the specified geometry or mesh entity can be removed from the block definition.

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 PYRAMID TETRA TETRA4 TETRA8 TETRA10 TETRA14

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.

Higher order nodes are moved to curved geometry by default. To change this, use the following command:

set Node Constraint [ON|off|smart]

On means higher order nodes snap to curved geometry. **Off** means the nodes retain their positions. **"smart"** means higher order 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. Several examples of specifying various types of element blocks are given in the Appendix.

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]

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** argument indicates that only surfaces containing elements will use default blocks.

When default blocks are used, the IDs for the resulting blocks will be defined as follows based upon the type of geometry:

Volume: The default block ID will be set to the Volume ID

Surface: The block ID will be set to 0

Curve: The block ID will be set to

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 an attribute to an element block, use the following command:

Block <block_id_range> Attribute <value>

The default number of attributes of an element block is dependent on the element type of the element block. Except for the element blocks of the element types below, all element blocks contain zero attributes by default.

Element Type	Number Default Attributes
SPHERE	1
BAR	3
BEAM	7
TRUSS	1
SPRING	1
SHELL	1
TRISHELL	1

To assign more attributes than the number of default attributes use the following command:

Block <id_range> Attribute Count <1-20>

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.

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, nodeset, and sideset renumber commands give the user the ability to renumber these entities to fit the user's needs. The command is:

{Block|Nodeset|Sideset} <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 id space for the entity. 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.

Examples:

Assume:

block ids: 100, 105, 106, 109
nodeset ids: 201, 202, 300
sideset ids: 1, 2, 4, 6, 10

block all renumber start 20 uniqueids
nodeset all renumber start 20
sideset all renumber start 30
block 20 renumber start 24
After commands:

block ids: 21, 22, 23, 24
nodeset ids: 20, 21, 22, 23

sideset ids: 30, 31, 32, 33, 34

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.

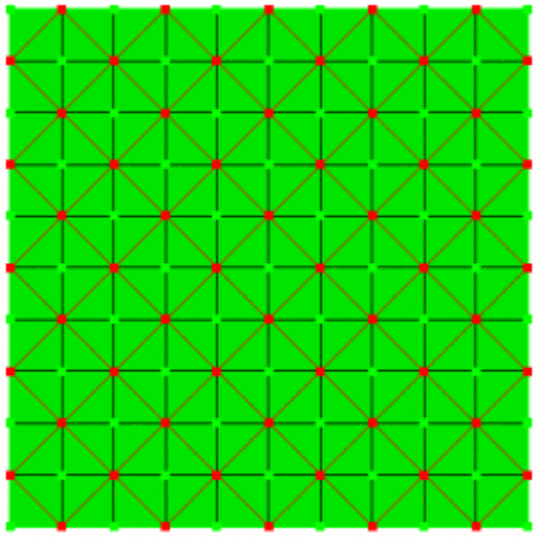
Similarly, you can use the following command which will use the 30 degree cone described above to gather edges from a surface into a single block using the Cartesian x, y, and/or z vectors.

> Rebar Surface <range> [x] [y] [z] Block <id> [Element Type {bar|bar2|bar3|BEAM|beam2|beam3|truss|truss2|truss3}] [Propagate]

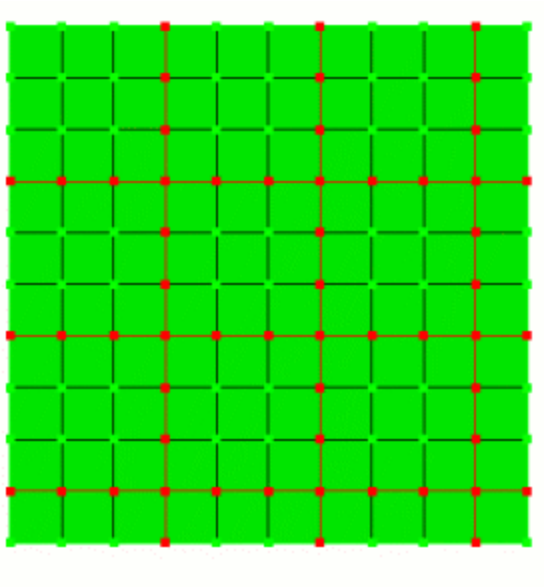
Diagonal and Orthogonal Rebar Blocks

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

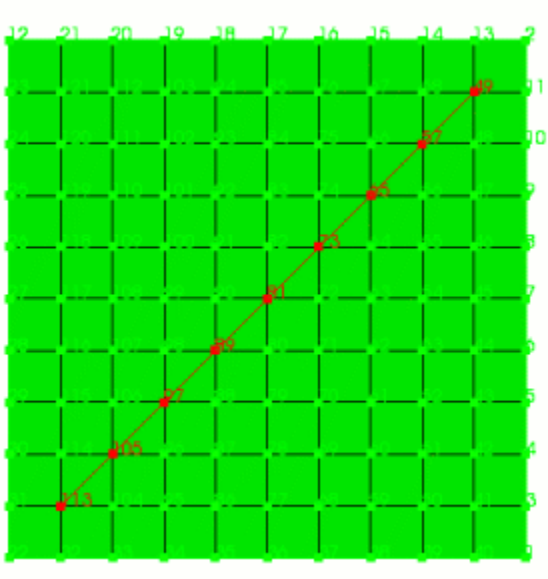


CUBIT> rebar surf 1 orthogonal spacing 3 block 3

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|VertexFace|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 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.

When specifying vertex ids, please know the bar elements will be tied to the nodes associated with the vertex, not the vertex itself.

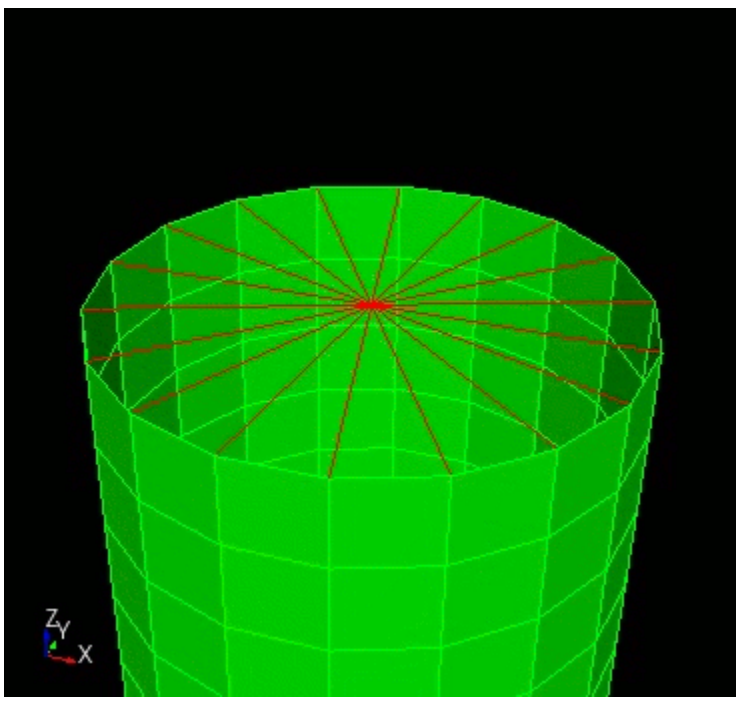


Figure 1. Beam elements created with the Spider command

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.

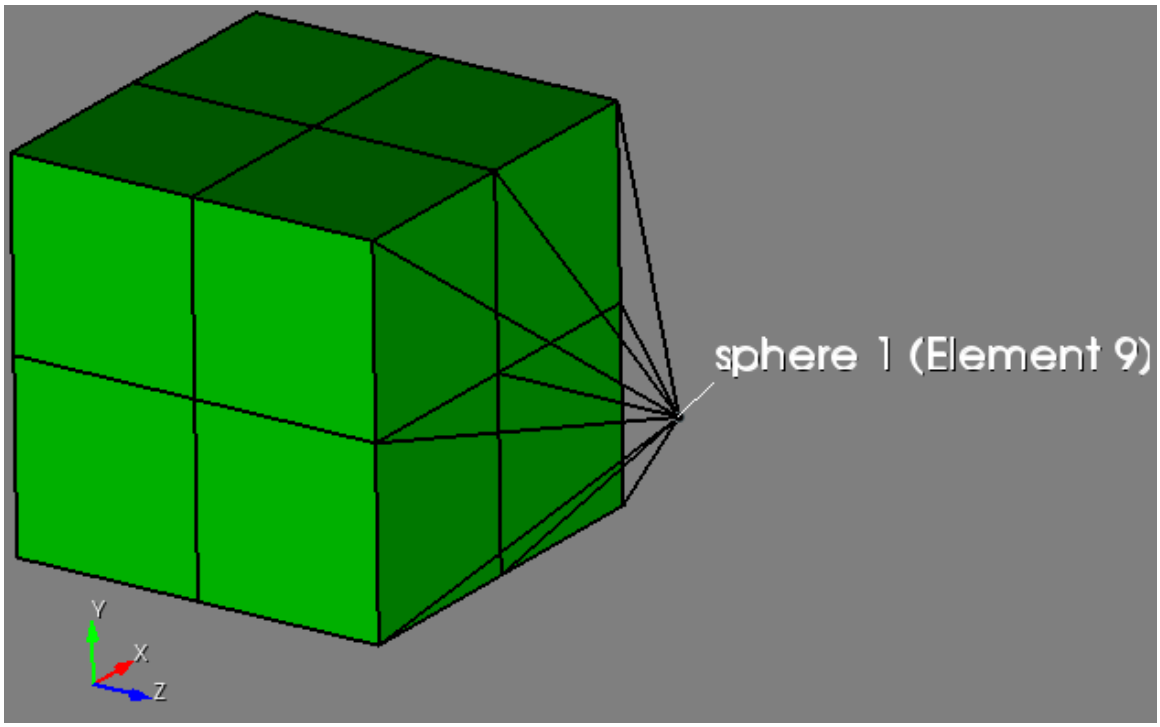


Figure 2. A sphere element created and connected to a solid mesh with 2d elements.

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.

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
- 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> {Curve | Surface | Volume | Vertex | Node} <range> [Remove]

Sideset <sideset_id> Group <id_range> [remove]

Sideset <sideset_id> {Curve|Surface|Edge|Face|Tri} <id_range> Remove

Sideset <sideset_id> Edge <id_range> [wrt {{Tri|Face} <id_range> | all } ]

Sideset <sideset_id> Face <id_range> [wrt {Hex <id_range> | all} ]

Sideset <sideset_id> Tri <id_range> [wrt {Tet <id_range> | all} ]

Sideset <sideset_id> Surface <id_range> [wrt {{Volume|Surface} <id_range> | all} ]
[FORWARD|Reverse|Both]

Sideset <sideset_id> 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 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 surface 2 remove
# Remove nodes from the nodeset
# that belong to the curves that bound surface 2
Nodeset 1 node in curve in surface 2 remove
```

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 node in surface all with z_coord = 15 remove
```

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

*T*he block, nodeset, and sideset renumber commands give the user the ability to renumber these entities to fit the user's needs. The command is:

{Block|Nodeset|Sideset} <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 id space for the entity. 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.

Examples:

Assume:

block ids: 100, 105, 106, 109
nodeset ids: 201, 202, 300
sideset ids: 1, 2, 4, 6, 10

block all renumber start 20 uniqueids
nodeset all renumber start 20
sideset all renumber start 30
block 20 renumber start 24
After commands:

block ids: 21, 22, 23, 24
nodeset ids: 20, 21, 22, 23
sideset ids: 30, 31, 32, 33, 34

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|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 set the behavior of nodesets/sideset/blocks when a copy command is applied on geometric entities. The default OFF option states that the nodesets/sideset/blocks will not be copied to new geometric entity if the original geometric entity contains nodesets/sideset/blocks. The "on" option implies that the nodesets/sideset/blocks present in the original geometric entity will be copied to new geometric entity. The use_original option indicates that the new geometric entities created after the copy command will have the nodesets/sidesets/blocks present in the original geometric entities.

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]

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
- 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> {Curve | Surface | Volume | Vertex | Node} <range> [Remove]

Sideset <sideset_id> Group <id_range> [remove]

Sideset <sideset_id> {Curve|Surface|Edge|Face|Tri} <id_range> Remove

Sideset <sideset_id> Edge <id_range> [wrt {{Tri|Face} <id_range> | all } ]

Sideset <sideset_id> Face <id_range> [wrt {Hex <id_range> | all} ]

Sideset <sideset_id> Tri <id_range> [wrt {Tet <id_range> | all} ]

Sideset <sideset_id> Surface <id_range> [wrt {{Volume|Surface} <id_range> | all} ]
[FORWARD|Reverse|Both]

Sideset <sideset_id> 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 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 surface 2 remove
# Remove nodes from the nodeset
# that belong to the curves that bound surface 2
Nodeset 1 node in curve in surface 2 remove
```

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 node in surface all with z_coord = 15 remove
```

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

*T*he block, nodeset, and sideset renumber commands give the user the ability to renumber these entities to fit the user's needs. The command is:

{Block|Nodeset|Sideset} <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 id space for the entity. 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.

Examples:

Assume:

block ids: 100, 105, 106, 109
nodeset ids: 201, 202, 300
sideset ids: 1, 2, 4, 6, 10

block all renumber start 20 uniqueids
nodeset all renumber start 20
sideset all renumber start 30
block 20 renumber start 24
After commands:

block ids: 21, 22, 23, 24
nodeset ids: 20, 21, 22, 23
sideset ids: 30, 31, 32, 33, 34

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|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 set the behavior of nodesets/sideset/blocks when a copy command is applied on geometric entities. The default OFF option states that the nodesets/sideset/blocks will not be copied to new geometric entity if the original geometric entity contains nodesets/sideset/blocks. The "on" option implies that the nodesets/sideset/blocks present in the original geometric entity will be copied to new geometric entity. The use_original option indicates that the new geometric entities created after the copy command will have the nodesets/sidesets/blocks present in the original geometric entities.

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]

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>'

Exodus Coordinate Frames

CUBIT allows the user to define coordinate systems (frames) that are written to an Exodus II 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. Exodus coordinate frames can be created using x-y-z coordinates, nodes or vertices with the following commands:

```
Exodus Create Coordinate Frame
<xval> <yval> <zval>//origin
<xval> <yval> <zval> //z-axis
<xval> <yval> <zval> //xz-plane
[tag { 'R' | 'C' | 'S' } ]

Exodus Create Coordinate Frame Node
<node_origin_id>
<node_zaxis_id>
<node_xzplane_id>
[tag { 'R' | 'C' | 'S' } ]

Exodus 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 Exodus Coordinate Frame [ids] [ <frame_id>]

Delete Exodus Coordinate Frame [ids] [ <frame_id>| all]
```

Any exodus coordinate frames that exist at the time the exodus file is exported will be written out in the exodus file.

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'>

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.

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.

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])).

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 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 CFD Boundary Conditions

- Inlet Velocity
- Inlet Pressure
- Inlet Massflow
- Outlet Pressure
- Farfield Pressure
- Symmetry

CUBIT can export models to the Fluent mesh format and supports defining the above 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) is not defined within CUBIT and must be assigned using a CFD model editor, such as Fluent.

The following shows the commands for creating and modifying CFD boundary conditions. To delete them, use the delete command (see Miscellaneous Commands).

Inlet Velocity

Create Inletvelocity [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Inletvelocity [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

Inlet Pressure

Create Inletpressure [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Inletpressure [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

Inlet Massflow

Create Inletmassflow [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Inletmassflow [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

Outlet Pressure

Create Outletpressure [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Outletpressure [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

Farfield Pressure

Create Farfieldpressure [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Farfieldpressure [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

Symmetry

Create Symmetry [id] [name <'name'>] [{Add|On} {Surface} <entity_list>]

Modify Symmetry [id] [name <'name'>] [{Add|Remove} {Surface} <entity_list>]

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.

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>”.

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.

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>] [cubitids] [XML '<filename>']

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 portion of the blocks in the model. The **<id_list>** may be any valid set of integers corresponding to the Blocks in the current model.

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]

Instancing Parts with ABAQUS

The ABAQUS file format allows users to instance a mesh multiple times. An example of this would be to create a mesh of a single bolt, but instance the bolt mesh several times in the ABAQUS model file to generate multiple bolts.

To create an ABAQUS file with instanced parts, use the following syntax:

> Export Abaqus <'filename'> [Block <id_list>] [Sideset <id_list>] [Nodeset <id_list>] [BCSet <id_list>]
> [Instance Block <id_list> [Source_csys <id>] [Target_csys <id_list>] [Overwrite] [Cubitids] [Everything]

Any block defined in Cubit can be instanced n number of times in the ABAQUS file. 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 instance keyword can be used as many times as needed.

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 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.'

The exporter can handle higher-order elements, although Fluent will convert the elements to first-order upon import.

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

Automatic Detail Suppression

Note: This feature is under development. The command to enable or disable features under development is:

Set Developer Commands {On|OFF}

Geometry models often have small features, which can be difficult to resolve in a mesh. In fact, these features are sometimes too small to see, and are revealed only when the user attempts to mesh the geometry. Automatic detail suppression identifies and removes the following types of features from the geometric model:

- valence-2 vertices
- short edges
- small faces

Details are removed using virtual geometry , which means they can be restored later if desired.

There are several stages to the automatic detail suppression process, all of which can be controlled separately by the user. Small details are identified using the command:

Detail <ref entity list> [identify] [dimension <dim> [only]]

The results are placed in a series of groups named "detail_vertices", "detail_edges", "detail_faces" and "detail_volumes". These details can be drawn or highlighted using the normal group commands:

Draw {detail_vertices | detail_edges | detail_faces | detail_volumes}

Highlight {detail_vertices | detail_edges | detail_faces | detail_volumes}

Or by using the following command:

Detail <ref entity list> draw [dimension <dim> [only]]

Details are removed automatically from the model using the command:

Detail <ref entity list> remove [dimension <dim> [only]]

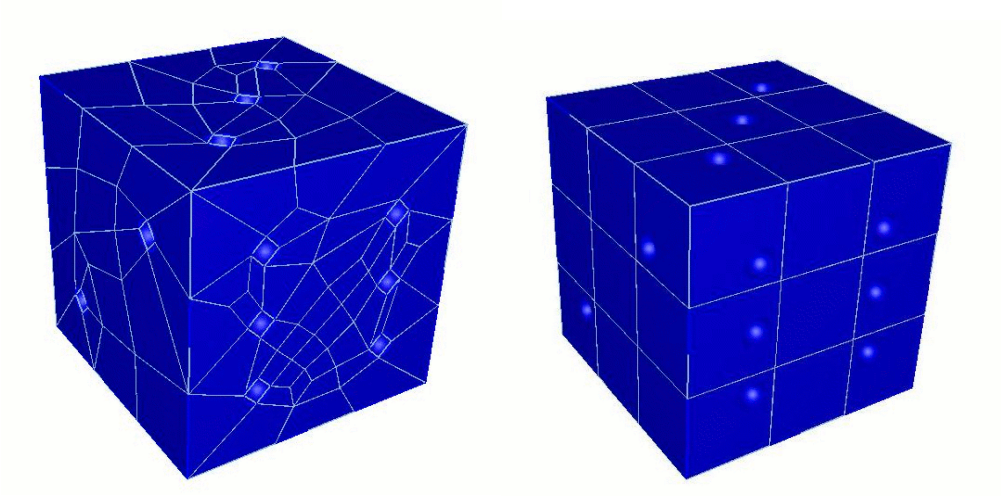
The **dimension** option is used to identify the maximum dimension of entities examined for small detail identification (<**dim**> is **3**, **2**, **1** for volumes, surface, and curves, respectively). If the **only** identifier is specified, only entities of the specified dimension are examined, otherwise that dimension and all lower dimensions are examined.

In some cases, details are identified which the user would like to retain in the model; likewise, the algorithm used to identify small details sometimes misses small details the user would like removed from the model. To include or exclude geometric entities from the list of small details to be removed, the following command is used:

Detail <ref entity list> [include | exclude]

Example

Shown below is a model of a game die meshed with identical mesh size, with details included (left) and removed (right).



Note: "Small" Measurement

Automatic detail suppression identifies "small" geometric entities by comparing their "size" to the mesh size assigned by the user to the entity. Anything smaller than that size is identified as being a detail and put in the appropriate detail group (e.g. detail_faces, detail_edges, etc.). The size of an edge is simply its arc length; surfaces and volumes are measured using the "hydraulic diameter" (see next note).

Note: Hydraulic Diameter

The hydraulic diameter of a surface is computed as 4.0*A/P, where A is the surface area and P is the summed arc lengths of all bounding curves. For circles, the hydraulic diameter is the circle diameter; for squares, it is the length of the bounding curves. Similarly, for volumes, the hydraulic diameter is computed as 6.0*V/A, which evaluates to the diameter and bounding curve length for perfect spheres and cubes, respectively.

Automatic Geometry Decomposition

Note: This feature is under development. The command to enable or disable features under development is:

Set Developer Commands {On|OFF}

In many cases, model geometry includes protrusions which, when cut off using geometry decomposition, are easily meshable with existing algorithms. CUBIT includes a feature-based decomposition capability, which automates this process. This algorithm operates by finding concave curves in the model, grouping them into closed loops, then forming cutting surfaces based on those loops. Although this algorithm is still in the research stage, it can be useful for automating some of the decomposition required for typical models.

To automatically decompose a model, use the command

Cut Body <body_id_range> [Trace {on|off}] [Depth <cut_depth>]

If the **Trace** option is used, the algorithm prints progress information as decomposition progresses. The **Depth** option controls how many cuts are made before the algorithm returns; by default, the algorithm cuts the model wherever it can.

Automatic decomposition is used to decompose the model shown in Figure 1 (left), with the results shown in Figure 1 (right). In this case, automatic decomposition performs all but one of the required cuts.

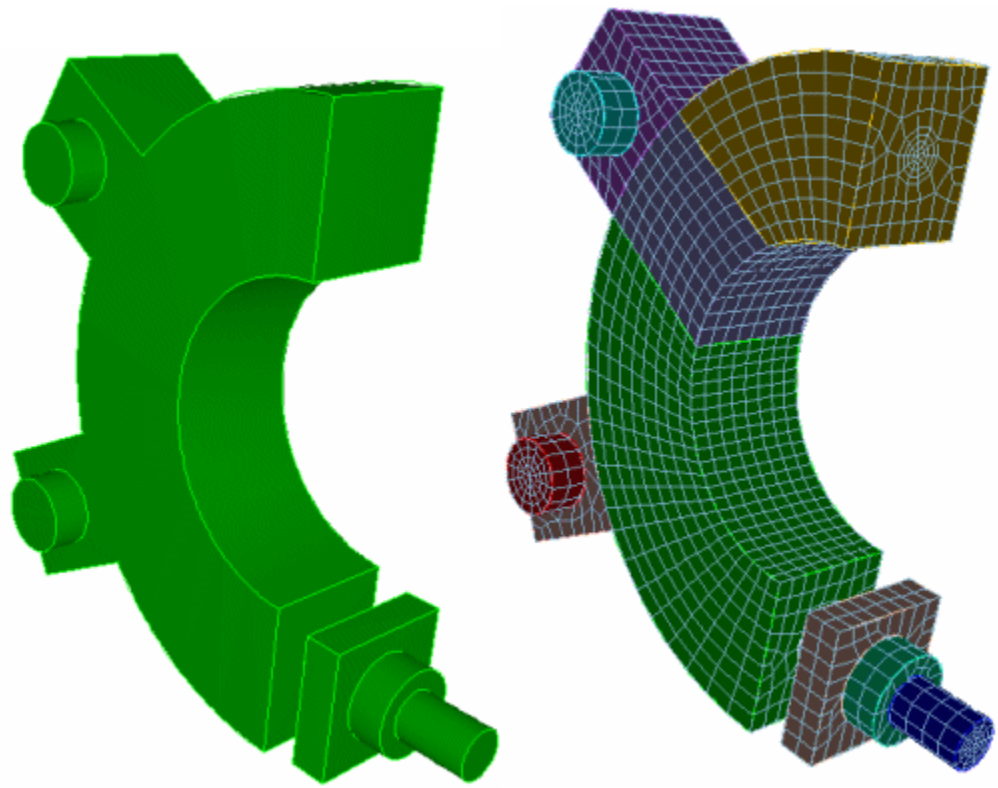


Figure 1. Model where automatic decomposition was utilized.

Cohesive Elements

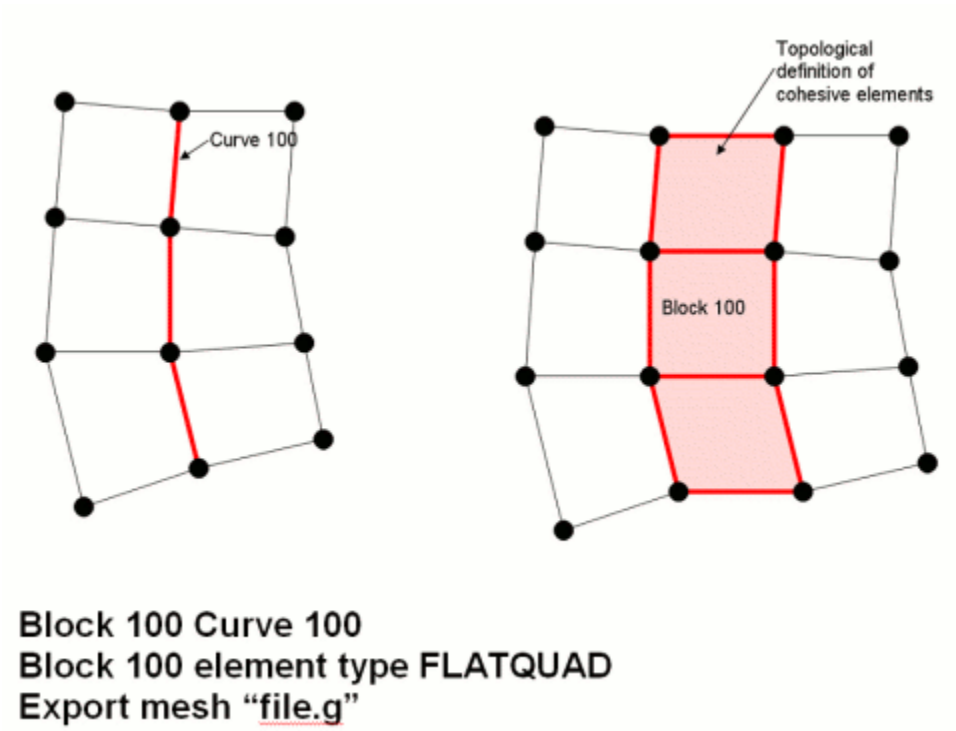
Note: This feature is under development. The command to enable or disable features under development is:

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.

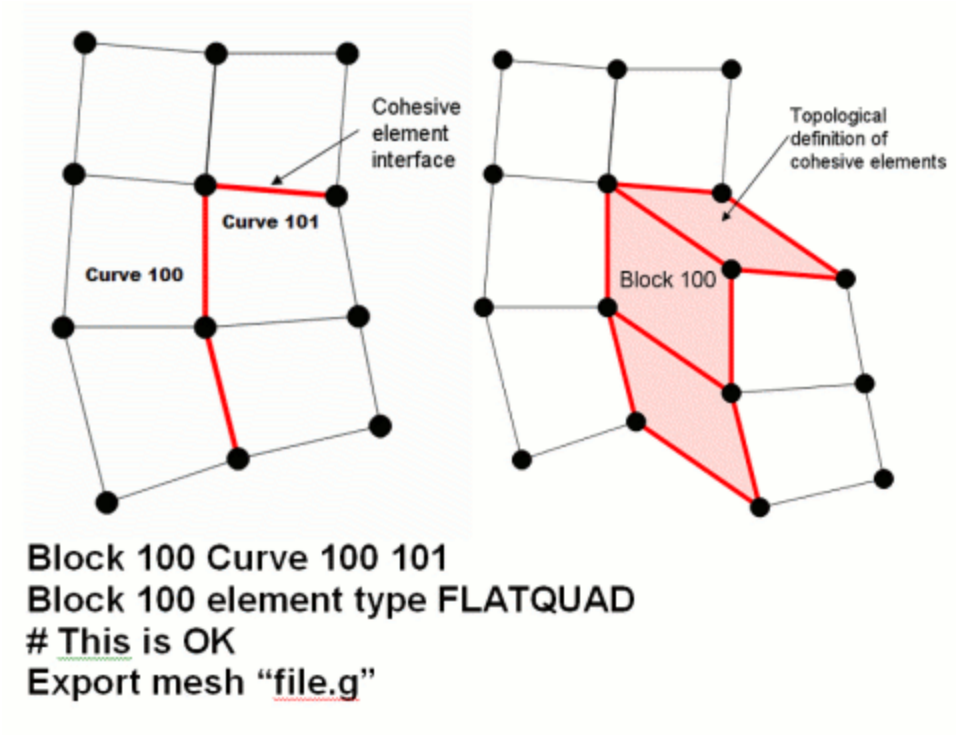
Cubit supports 2D cohesive regions. Cohesive elements are implemented in Cubit as element blocks with an element type of FLATQUAD. The cohesive region is identified by assigning geometric curves to the FLATQUAD element block. When the element block is exported, each edge on the specified curves is represented 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.

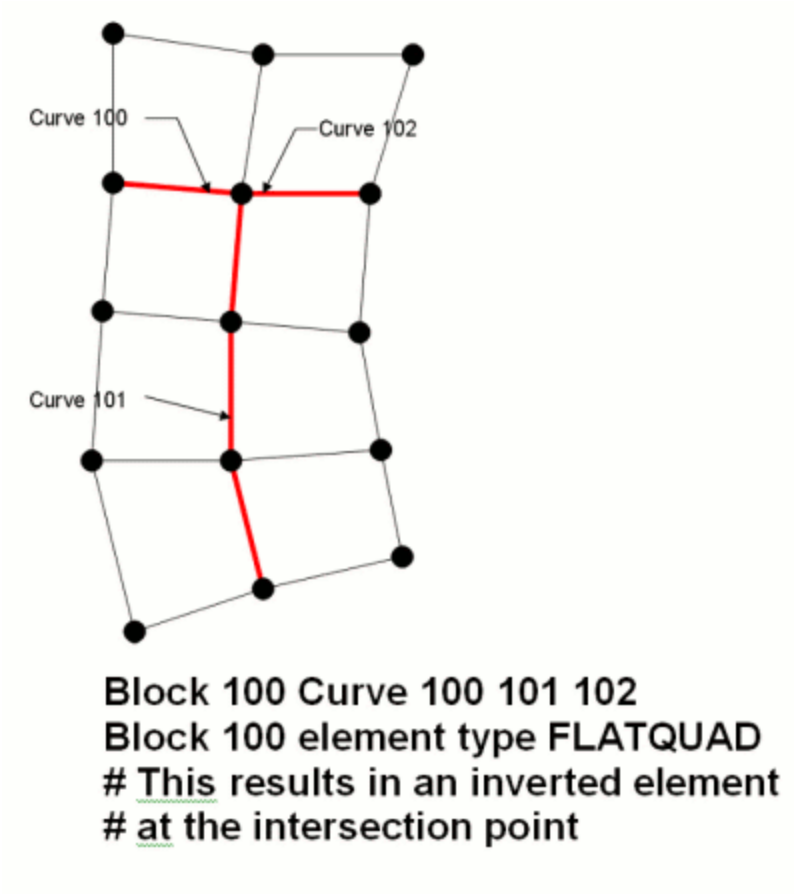


Multiple Curves in FLATQUAD Blocks

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.



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.


Curve 100
Curve 102
Curve 101
Block 100 Curve 100 101 102
Block 100 element type FLATQUAD
This results in an inverted element
at the intersection point

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.

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.

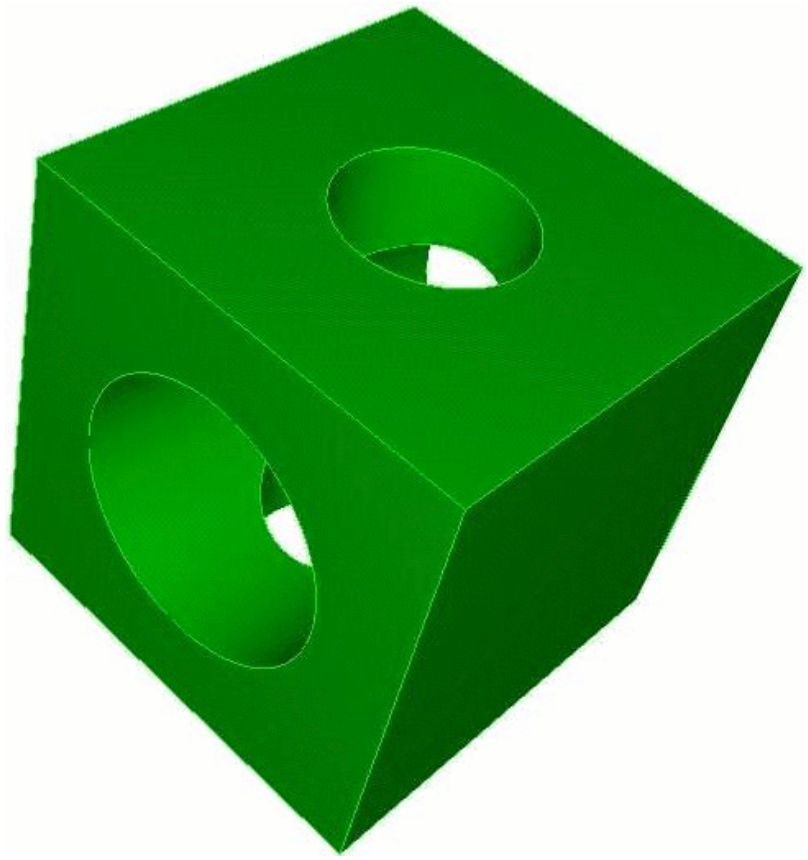


Figure 1: The original, unmeshed body

Step 1: Create a starting mesh. Figure 2 below shows the starting mesh for this problem. The commands for this mesh are:

cubit> 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> mesh volume 1

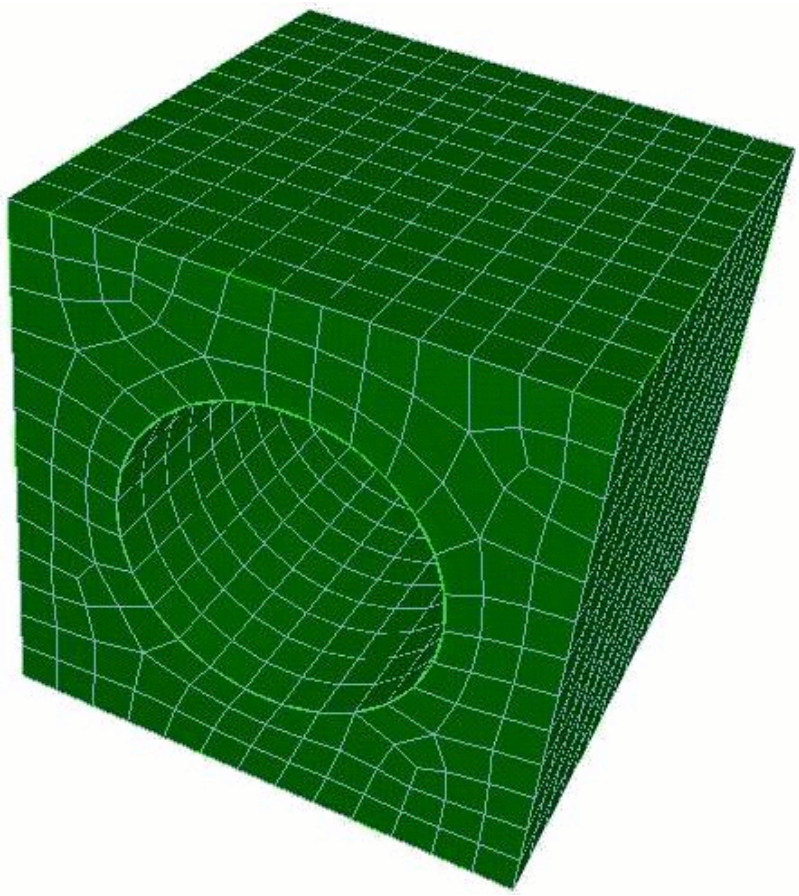


Figure 2: The starting mesh

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 body 3 about x angle 90

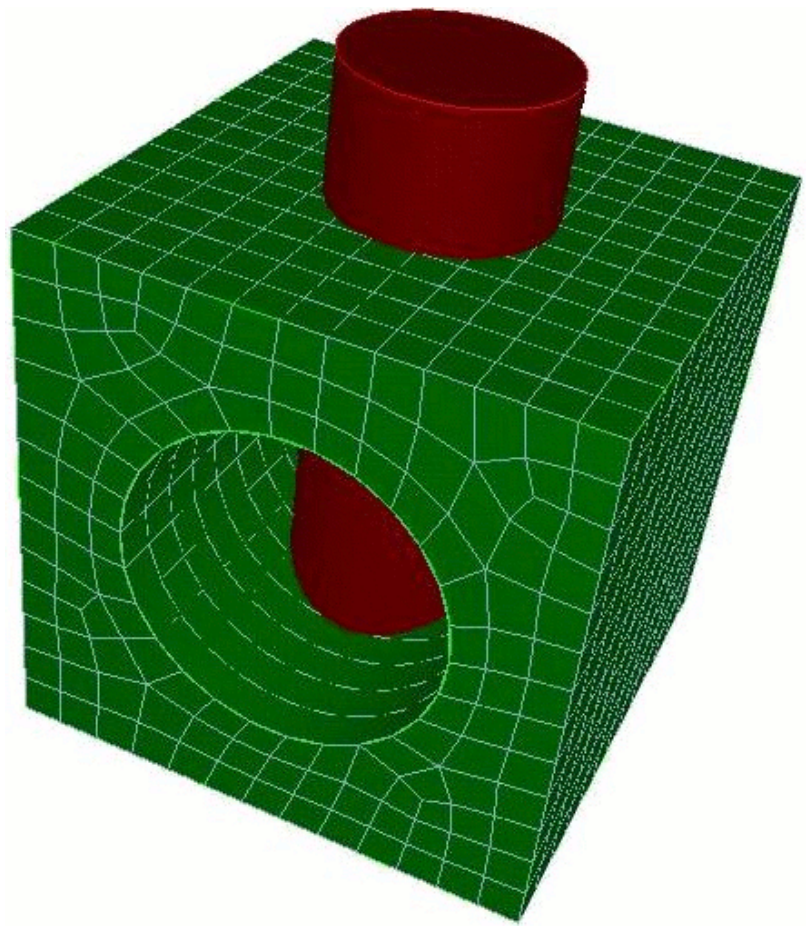


Figure 3: The starting mesh and cutting entity

Step 3: Cut the mesh. Figure 4 shows the new mesh after the original mesh has been cut. At this point we have 3 meshed volumes. The commands for this step are:

cubit> meshcut vol 1 sheet surface 13
cubit> draw volume 1 4 5

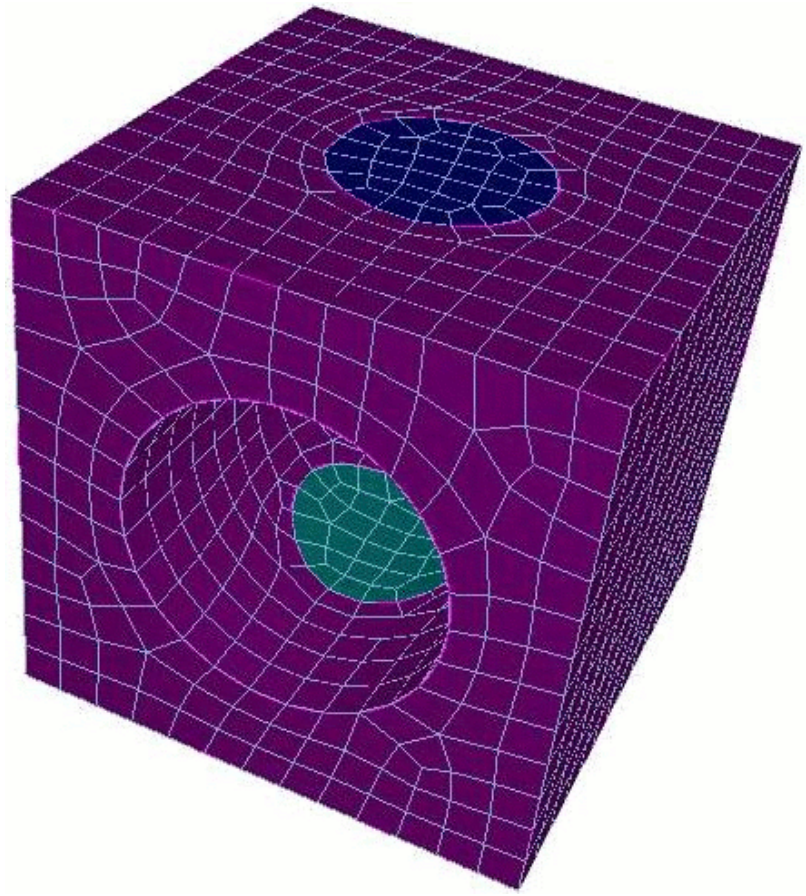


Figure 4: The mesh after meshcutting

Step 4: Final step. Figure 5 shows the final mesh after the mesh of the mesh of the two extra volumes is deleted. The commands are:

cubit> delete mesh vol 4 5 propagate
cubit> draw volume 1

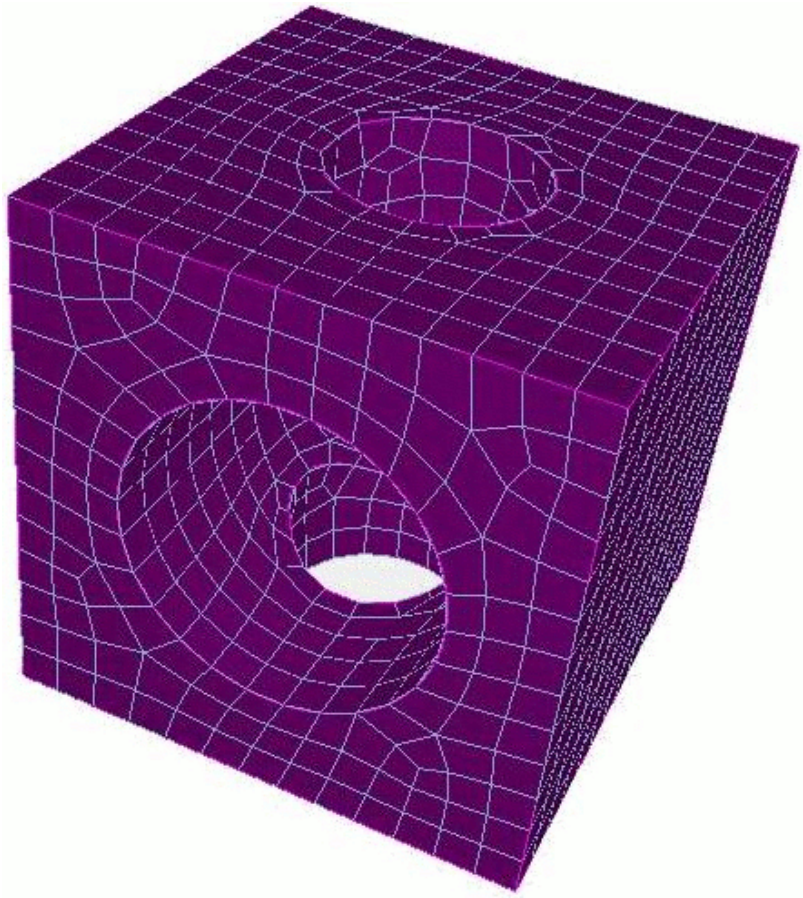


Figure 5: Final mesh after deleting unneeded elements

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.

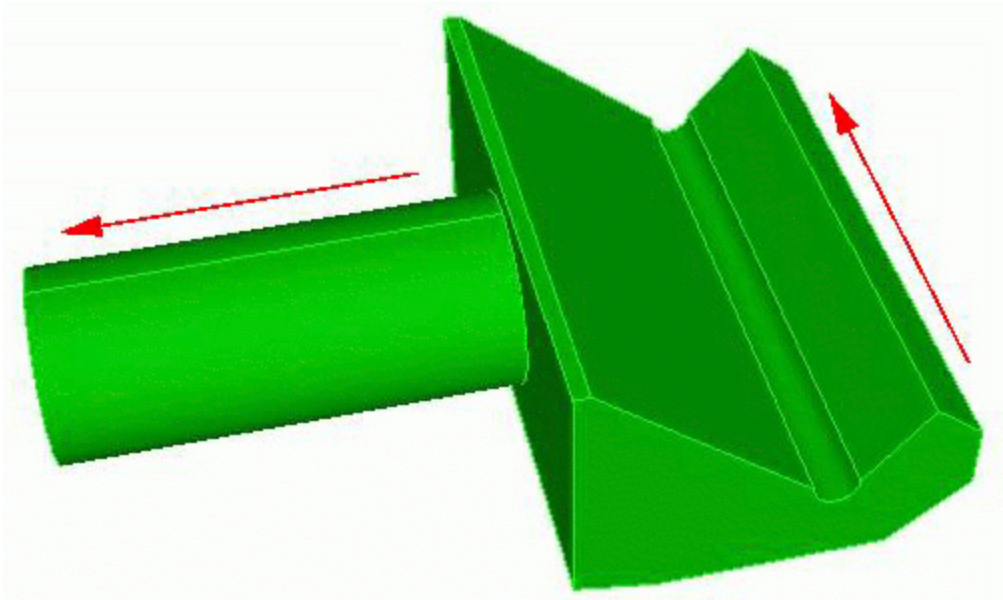


Figure 1. A model with two intersecting sweep directions.

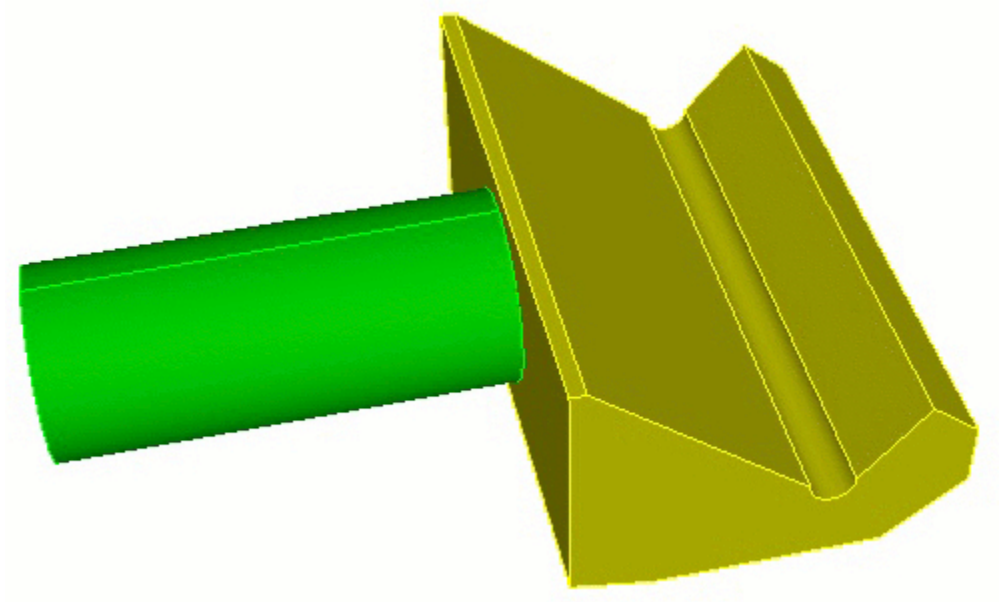


Figure 2. The model decomposed for grafting

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.

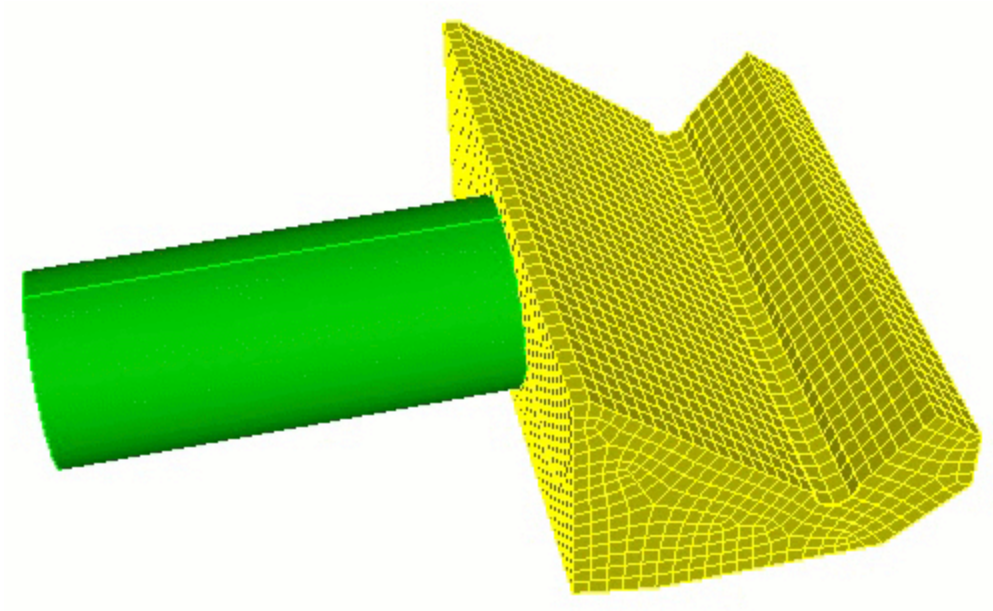


Figure 3. Meshed trunk volume.

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.

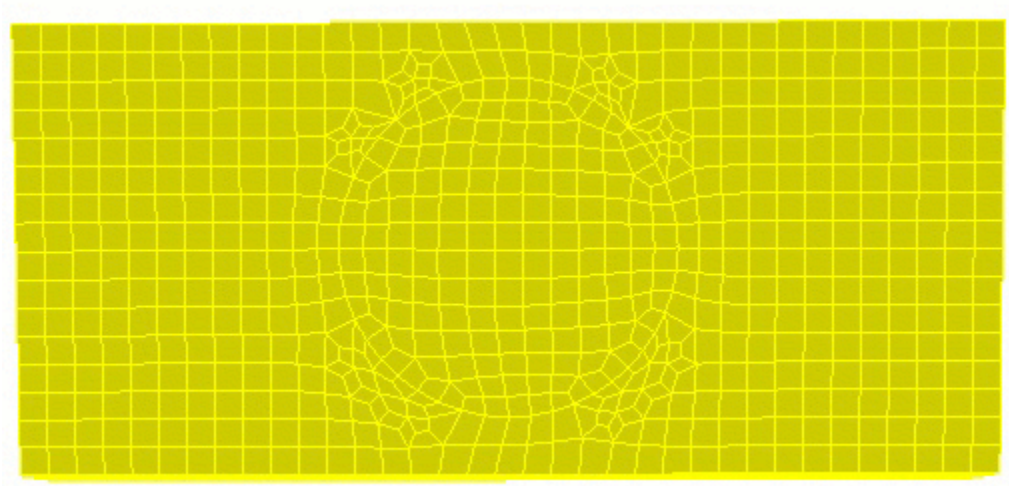


Figure 4. Trunk surface after grafting.

Step 4: Mesh the branch volume.

The final mesh is shown in Figure 5.

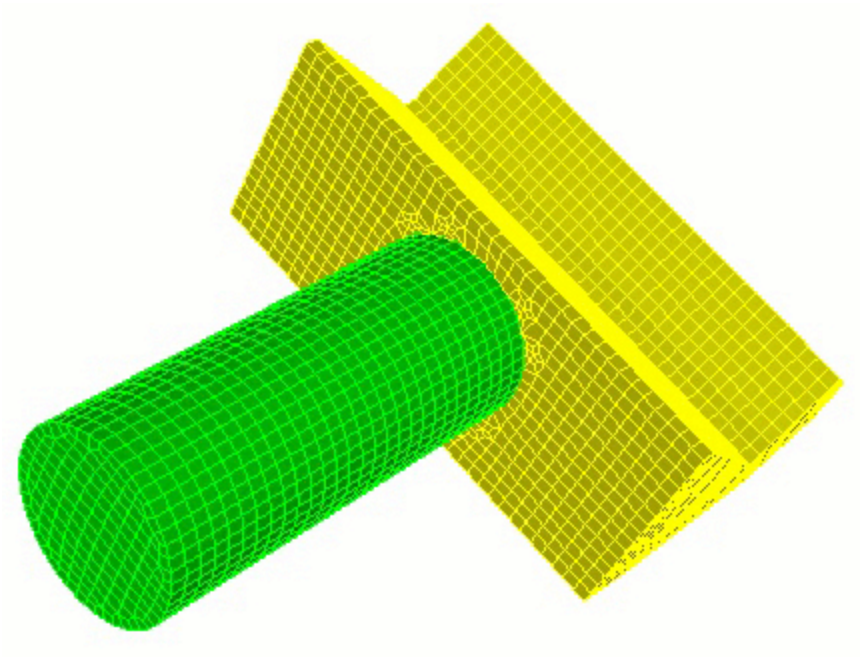


Figure 5. Final mesh

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.

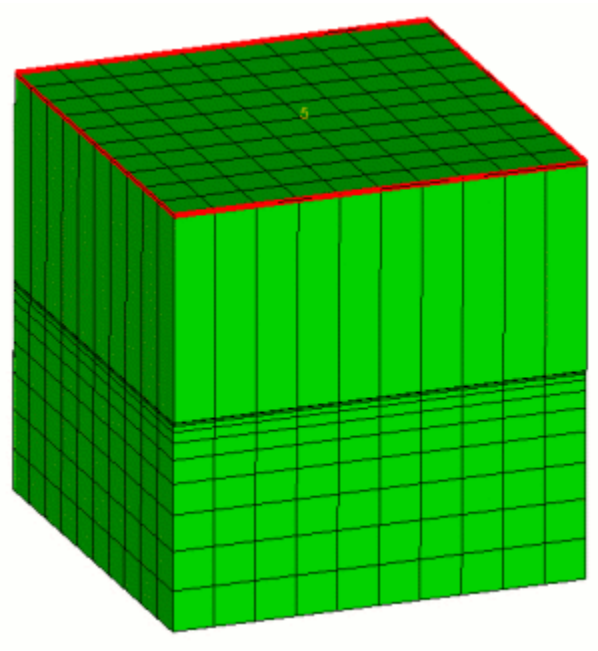


Figure 5. Example of Boundary Surface Refinement

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.

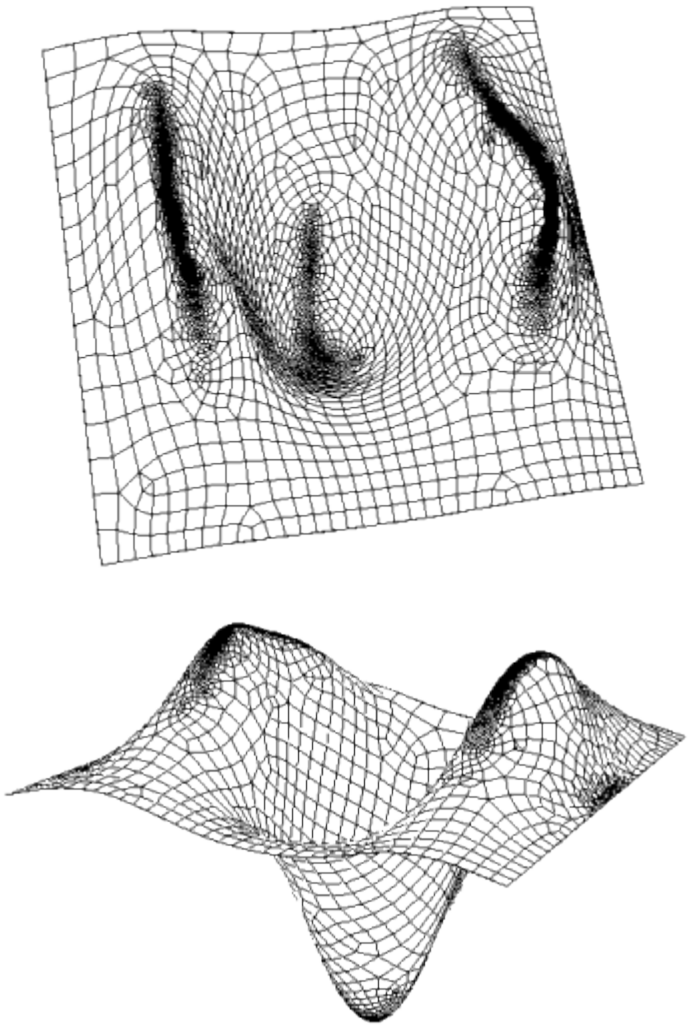


Figure 1. NURB mesh with super sizing function, 34 by 16 density

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.

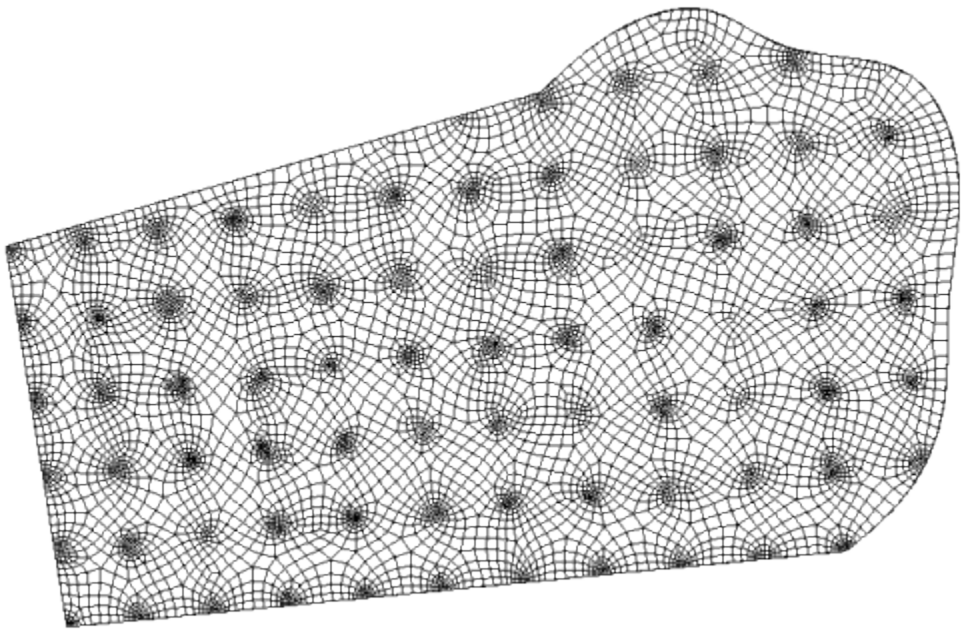


Figure 1. Test sizing function for spline geometry

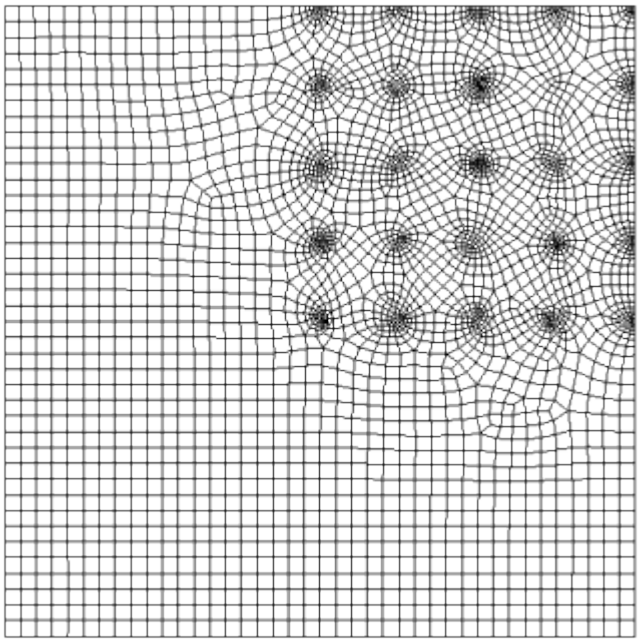


Figure 2. Test sizing function for square geometry

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

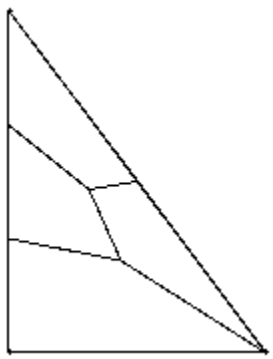
Summary: Produces a specified transition mesh for specific situations

Syntax:

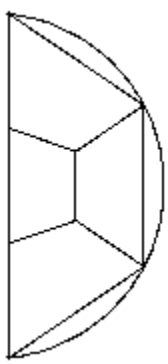
Surface <range> Scheme Transition
{Triangle|Half_circle|Three_to_one|Two_to_one|Convex_corner|Four_to_two} [Source Curve <id>]
[Source Vertex <id>]

Discussion:

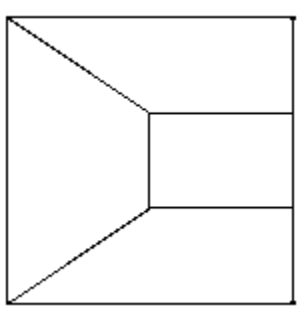
The *transition* scheme supplies a set of transition primitives which serve to transition a mesh from one density to another across a given surface. The six transition sub-types are demonstrated here.



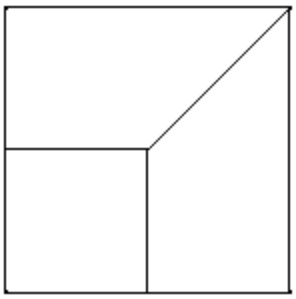
Scheme Transition **Triangle** creates four quads in a triangle that has sides of three, two, and one intervals.



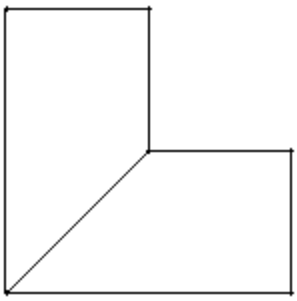
Scheme Transition **Half_circle** creates three intervals on the flat and three on the curved part of the half-circle, then creates four quads in the surface.



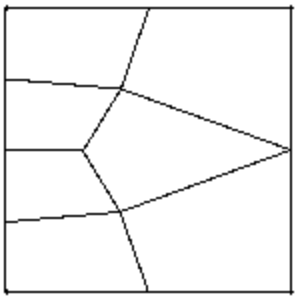
Scheme Transition **Three_to_one** creates four quads on a rectangular surface that has intervals of three, one, one, and one on its sides.



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

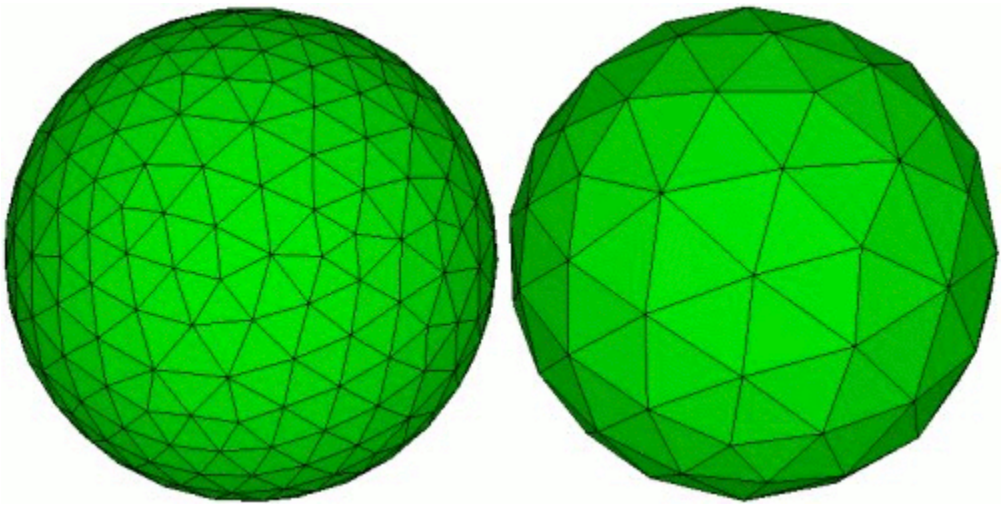


Figure 1. Example of coarsening all triangles with a factor of 2.

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.

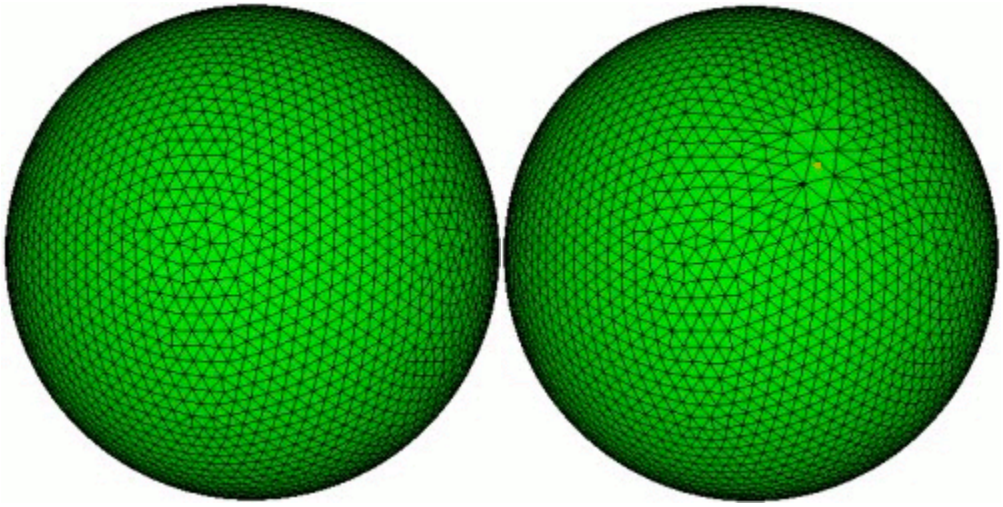


Figure 2. Coarsening performed at a node with factor = 3 and depth = 3

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.

Whisker Weave

Note: This feature is under development. The command to enable or disable features under development is:

Set Developer Commands {On|OFF}

Applies to: Volumes

Summary: Research algorithm for all-hexahedral meshing of arbitrary 3D volumes

Syntax:

Volume <range> Scheme Weave

Related Commands:

Pillow Volume <range>

{Volume|Surface|Curve} <range> Mesh [Fixed|Free]

Set AutoWeaveShrink [on|off]

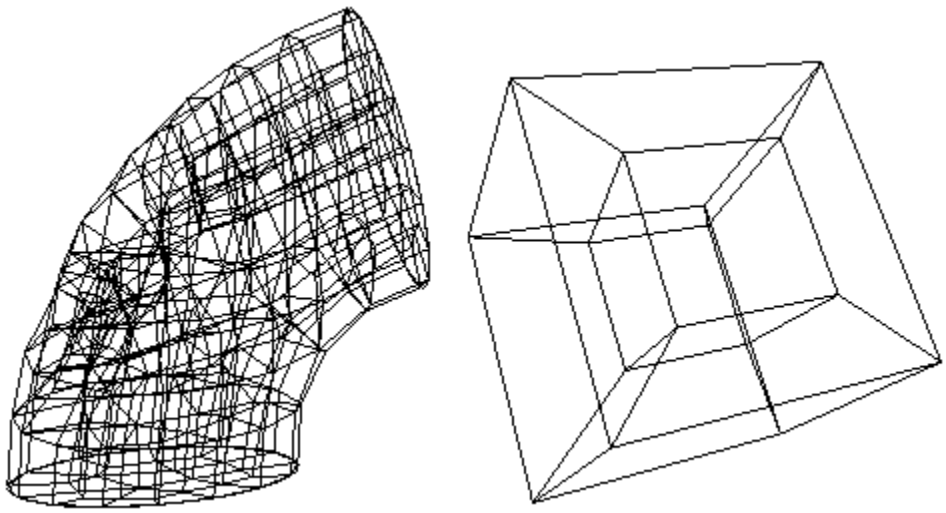
Set Statelist [on|off]

Discussion:

Whisker Weaving (Tautges, 96; Tautges, 95; Folwell, 98) is a volume meshing algorithm currently being researched and is not released for general use. However, daring users may find the current form of the algorithm useful for mostly-convex geometries.

Whisker Weaving holds the promise of being able to fill arbitrary geometries with hexahedra that conform to a fixed surface mesh. The algorithm is based on the rich information contained in the Spatial Twist Continuum (STC) (Murdoch, 95), which is the grouping of the dual of an all-hexahedral mesh into an arrangement of surfaces called sheets. Given a bounding quadrilateral surface mesh, Whisker Weaving constructs sheets advancing from the boundary inward. The sheets are then modified so that the arrangement dualizes to a well defined hexahedral mesh. Once the primal hex-mesh is generated, interior node positions are generated by smoothing.

Examples of meshes generated using the whisker weaving algorithm are shown in the following figure.



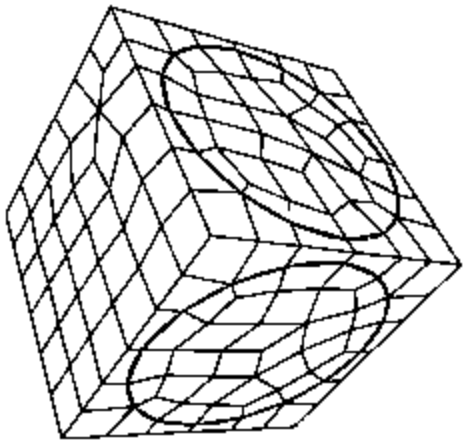


Figure 1. Some simple Whisker Weaving meshes with good quality

Whisker Weaving Basic Commands

The basic steps for meshing a volume with Whisker Weaving are the following:

Set the meshing scheme for the volume to weave

Volume <range> Scheme Weave

Mesh the volume, which generates hexes

Mesh Volume <range>

Pillow the volume to remove certain additional degenerate hexes

Pillow Volume <range>

and typically, smooth the mesh to improve quality, e.g.

Volume <range> Smooth Scheme Condition Number

Smooth Volume <range>

Whisker Weaving Options

Currently, Whisker Weaving relies on being able to perturb the bounding quadrilateral mesh. However, a bounding surface's mesh will not be changed if it is contained in another volume that is already meshed.

The user may also explicitly prevent Whisker Weaving from changing a bounding mesh by fixing it with the following command:

{Volume|Surface|Curve} <range> Mesh [Fixed|Free]

The user may select an optional control strategy that doesn't change the surface mesh by setting **AutoWeaveShrink** off, and setting **Statelist** on with the following commands:

Set AutoWeaveShrink [on|off]

Set Statelist [on|off]

Numerous developer commands are available for stepping through the algorithm, examining results, and toggling options. These are available via the command line help but are not detailed here.

Available Colors

All color commands in CUBIT require the specification of a color name. The following table lists the colors available in CUBIT at this time. 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	green	0.000	1.000	0.000
3	yellow	1.000	1.000	0.000
4	red	1.000	0.000	0.000
5	magenta	1.000	0.000	1.000
6	cyan	0.000	1.000	1.000
7	blue	0.000	0.000	1.000
8	white	1.000	1.000	1.000
9	orange	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

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.

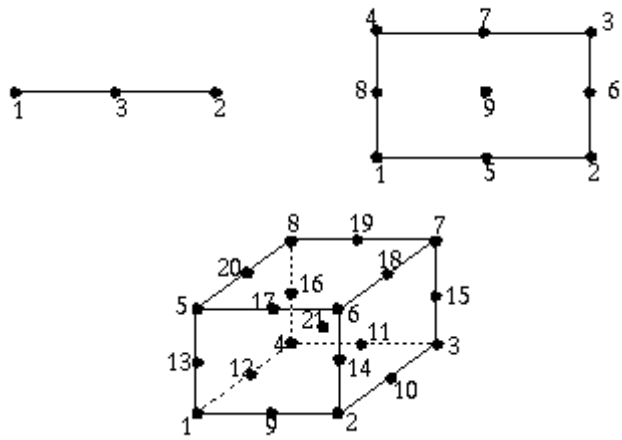


Figure 1. Local Node Numbering for CUBIT element types

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.

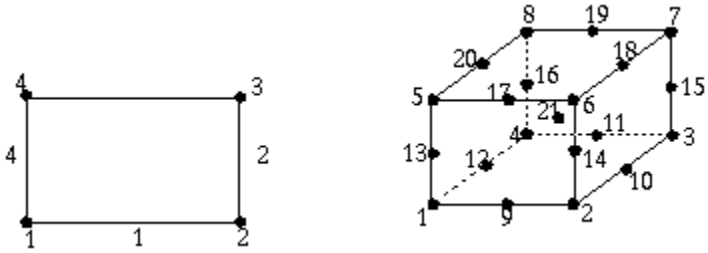


Figure 2. Local side numbering for CUBIT element types

Triangular Shell Element Numbering

A three-dimensional shell element with triangular topology will have the element type 'TRIANGLE'. 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 'TRIANGLE6'. However, any element whose type begins with the 8 letters 'TRIANGLE' 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.

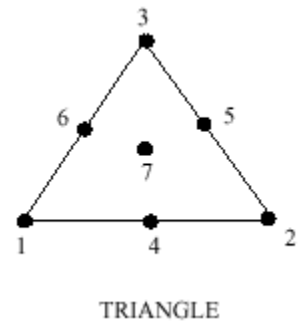


Figure 3. Local Node Numbering for CUBIT triangular element types

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.

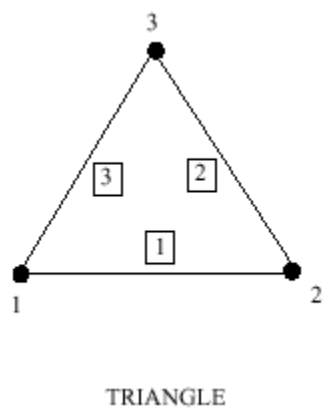


Figure 4. Local sideset numbering for CUBIT triangular element types

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}

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.

APREPRO Syntax

Within CUBIT, *APREPRO* expressions must be written inside of curly braces {}. For example, the following is a valid CUBIT command:

```
Curve 1 Size {sqrt(2.0)}
```

- this will set the mesh size on curve 1 to 1.414214....(the square root of 2)

APREPRO expressions can also exist on separate lines as follows:

```
#{_numSeat=30}
```

- this will set the variable _numSeat to be equal to 30
- instead of a # you can use $ (i.e., ${_numSeat=30}

As in the example, separate line expressions must exist within commented lines. There is an exception though - looping expressions must exist on non-commented lines. See Additional Functionality .

Immutable Variables

As of Cubit 15.0, immutable variables are supported. Immutable variables are created by putting the variable definition in an IMMUTABLE block. For example:

```
{ IMMUTABLE(ON) }

{ x = 2 }

{ y = 3 }

{ IMMUTABLE(OFF) }
```

APREPRO Rules

The rules that *APREPRO* uses when identifying functions, variables, numbers, operators, delimiters, and expressions are described below:

1. Functions

Function names are sequences of letters and digits and underscores (_) that begin with a letter. The function's arguments are enclosed in parentheses. For example, in the line **atan2(a,1.0), atan2** is the function name, and **a** and **1.0** are the arguments. See APREPRO Functions for a list of the available functions and their arguments.

2. Variables

A variable is a name that references a numeric or string value. A variable is defined by giving it a name and assigning it a value. For example, the expression **a = 1.0** defines the variable **a** with the numeric value **1.0**; the expression **b= "A string"** defines the variable **b** with the value **"A string"**. Variable names are sequences of letters, digits, and underscores (_) that begin with either a letter or an underscore. Variable names cannot match any function name and they are case-sensitive, that is, **abc_de** and **AbC_dE** are two distinct variable names. A few variables are predefined, these are listed in APREPRO Predefined Variables. Any variable that is not defined is equal to 0. A warning message is output to the terminal if an undefined variable is used, or if a previously defined variable is redefined. To see a list of all of the current APREPRO variables use the DUMP() command.

3. Numbers

Numbers can be integers like **1234**, decimal numbers like **1.234**, or in scientific notation like **1.234E-26**. All numbers are stored internally as floating point numbers.

4. Strings

Strings are sequences of numbers, characters, and symbols that are delimited by either single quotes (**'this is a string'**) or double quotes (**"this is another string"**). Strings that are delimited by one type of quote can include the other type of quote. For example, {**'This is a valid "string"'**}. Strings delimited by single quotes can span multiple lines; strings delimited by double quotes must terminate on a single line or a parsing error message will be issued.

5. Operators

Operators are any of the symbols defined in APREPRO Operators. Examples are + (addition), - (subtraction), * (multiplication), / (division), = (assignment), and ^ (exponentiation).

6. Delimiters

The delimiters recognized by APREPRO are: the comma (,) which separates arguments in function lists, the left curly brace (**{**) which begins an expression, the right curly brace (**}**) which ends an expression, the left parenthesis (which begins a function argument list, the right parenthesis) which ends a function argument list, the single quote (') which delimits a multi-line string, and the double quote (") which delimits a single-line string.

7. Expressions

An expression consists of any combination of numeric and string constants, variables, operators, and functions. Four types of expressions are recognized in APREPRO: algebraic, string, relational, and conditional.

8. Algebraic Expressions

Almost any valid FORTRAN or C algebraic expression can be recognized and evaluated by APREPRO. An expression of the form **a=b+10/37.5** will evaluate the expression on the right-hand-side of the equals sign and assign the value to the variable a. An expression of the form **b+10/37.5** will simply evaluate the expression. Variables can also be set on the command line prior to playing any journal files using the 'var=val' syntax. Only a single expression is allowed within the **{ }** delimiters. For example, **{x = sqrt(y^2 + sin(z))}, {x=y=z}**, and **{x=y} {a=z}** are valid expressions, but **{x=y a=z}** is invalid because it contains two expressions within a single set of delimiters.

9. String Expressions

APREPRO has very limited string support. The only supported operations are assigning a variable equal to a string (**a = "This is a string"**) or a function that returns a string, and concatenating two strings into another string (**a = "Hello" // " " // "World"**).

10. Relational Expressions

Relational expressions are expressions that return the result of comparing two expressions. A relational expression is either true or false. Relational expressions can only be used on the left-hand side of a conditional expression. A relational expression is simply two expressions of any kind separated by a relational operator. See Relational Operators.

11. Conditional Expressions

APREPRO recognizes a conditional expression of the form::

relational_expression ? true_exp : false_exp

where **relational_expression** can be any valid relational expression, and **true_exp** and **false_exp** are two algebraic expressions. If the relational expression is true, then the result of **true_exp** is returned, otherwise the result of **false_exp** is returned. For example, if the following command were entered:

#{a = (sind(20.0) > cosd(20.0) ? 1 : -1)}

then, a would be assigned the value **-1** since the relational expression to the left of the question mark is false. Both **true_exp** and **false_exp** are always evaluated prior to valuating the relational expression. Therefore, you should not write an equation such as

#{sind(20.0*a)>cosd(20.0*a) ? a=sind(20.0) : a=cosd(20.0)}

since the value of a can change during the evaluation of the expression. Instead, this equation should be written as:

#{a = (sind(20.0*a)>cosd(20.0*a) ? sind(20.0) : cosd(20.0))}

12. White Space

White space in APREPRO can be important. For example, the following expressions will parse correctly:

x_coord-1 and **x_coord -1**

The following expressions will not parse:

x_coord- 1 and **x_coord - 1**

The operator must not be separated from the operand by white space.

13. Switch Statements

Please refer to the current APREPRO documentation for details.

```
a = 10*PI
switch(10*PI + sin(0))
        ... This is ignored since it is after the switch, but before any case() statements
case(1)
        ... This is not executed since 1 is not equal to 10*PI+sin(0)
case(a)
        ... This is executed since a matches the value of 10*PI+sin(0)
case(10*PI+sin(0))
        ... This is not executed since a previous case was executed.
default
        ... This is not executed since a previous case was executed.
endswitch
        ... This is executed since the switch construct
        is finished.
```

14. Conditionals and Loops

- As of Cubit 15.0, **'ifdef'** and **'ifndef'** are deprecated. Use **if(expression)** and if **(!expression)** instead
- As of Cubit 15.0, **elseif(expression)** is valid
- **{abortloop}** will terminate all nested loops without expanding the loop . This is useful when typing in a loop at the command line and you make a mistake.
- A warning is issued to indicate to the user that they have started an APREPRO loop and that all subsequent commands will have no effect until the corresponding {**endloop**} or {**abortloop**} has been reached.
- A Cubit **'pause'** is not valid in a loop

APREPRO Operators

The operators recognized by APREPRO are listed below.

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Boolean Operators
- String Operators

In the following table, the letters **a** and **b** can represent variables, numbers, functions, or expressions unless otherwise noted. The tables below also list the precedence and associativity of the operators. Precedence defines the order in which operations should be performed. For example, in the expression:

3 * 4 + 6 / 2

the multiplications and divisions are performed first, followed by the addition because multiplication and division have higher precedence than addition. The precedence is listed from 1 to 14 with 1 being the lowest precedence and 14 being the highest.

Associativity defines which side of the expressions should be simplified first. For example the expression: **3 + 4 + 5** would be evaluated as **(3 + 4) + 5** for left associativity, the expression **a = b / c** would be evaluated as **a = (b / c)** for right associativity.

Note: Operators must not be separated from operands by white space. For example, the expressions **x_coord-1** and **x_coord -1** will parse correctly, while the expressions **x_coord - 1** and **x_coord- 1** will not parse correctly.

1. Arithmetic Operators

Arithmetic operators combine two or more algebraic expressions into a single algebraic expression. These have obvious meanings except for the pre- and post- increment and decrement operators. The pre-increment and pre-decrement operators first increment or decrement the value of the variable and then return the value. For example, if **a = 1**, then **b=++a** will set both **b** and **a** equal to **2**. The post-increment and post-decrement operators first return the value of the variable and then increment or decrement the variable. For example, if **a = 1**, then **b=a++** will set **b** equal to **1** and a equal to **2**. The modulus operator **%** calculates the integer remainder. That is both expressions are truncated an integer value and then the remainder calculated. See the **fmod** function in Mathematical Functions, for the calculation of the floating point remainder. The tilde character ~ is used as a synonym for multiplication to improve the aesthetics of the *APREPRO* unit conversion system (however, the unit conversions system is not supported in CUBIT). It is more natural for some users to type **12~metre** than **12*metre**

Table 1. Arithmetic Operators

Syntax	Description	Precedence	Associativity
a+b	Addition	9	left
a-b	Subtraction	9	left
a*b, a~b	Multiplication	10	left
a/b	Division	10	left
a^b, ab**	Exponentiation	12	right
a%b	Modulus (remainder)	10	left
++a, a++	Pre-, post-increment	13	left

--a, a--	Pre-, post-decrement	13	left

2. Assignment Operators

Assignment operators combine a variable and an algebraic expression into a single algebraic expression, and also set the variable equal to the algebraic expression. Only variables can be specified on the left-hand-side of the equal sign.

Table 2. Assignment Operators

Syntax	Description	Precedence	Associativity
a=b	The value of 'a' is set equal to 'b'	1	right
a+=b	The value of 'a' is set equal to a + b	2	right
a-=b	The value of 'a' is set equal to a - b	2	right
a*=b	The value of 'a' is set equal to a * b	3	right
a/=b	The value of 'a' is set equal to a / b	3	right
a^=b	The value of 'a' is set equal to a^b	4	right
a=b**	The value of 'a' is set equal to a^b	4	right

3. Relational Operators

Relational operators combine two algebraic expressions into a single relational expression. Relational expressions and operators can only be used before the question mark (?) in a conditional expression.

Table 3. Relational Operators

Syntax	Description	Precedence	Associativity
a<b	true if 'a' is less than 'b'	8	left
a>b	true if 'a' is greater than 'b'	8	left
a<=b	true if 'a' is less than or equal to 'b'	8	left
a>=b	true if 'a' is greater than or equal to 'b'	8	left
a==b	true if 'a' is equal to 'b'	8	left
a!=b	true if 'a' is not equal to 'b'	8	left

4. Boolean Operators

Boolean operators combine one or more relational expressions into a single relational expression. If **la** and **lb** are two relational expressions, then:

Table 4. Boolean Operators

Syntax	Description	Precedence	Associativity
1a || 1b	true if either 'la' or 'lb' are true.	6	left
1a && 1b	true if both 'la' and 'lb' are true.	7	left
!1a	true if 'la' is false.	11	left

5. String Operators

The only supported string operator at this time is string concatenation, which is denoted by //. If **a = "Hello"** and **b = "World"**, then:

c = a // " " // b

sets **c** equal to **"Hello World"**. Concatenation has precedence 14 and left associativity. Also see String Functions

APREPRO Predefined Variables

A few commonly used variables are predefined in *APREPRO*. These are listed below. The default output format is specified as a C language format string, see your C language documentation for more information. The default format and comment variables are defined with a leading underscore in their name so they can be redefined without generating an error message.

Table 1. Predefined Variables

Name	Value	Description
PI	3.14159265358979323846	pi
PI_2	1.57079632679489661923	pi/2
SQRT2	1.41421356237309504880	$\sqrt{2}$
DEG	57.2957795130823208768	180 /pi degrees per radian
RAD	0.01745329251994329576	pi/180 radians per degree
E	2.71828182845904523536	base of natural logarithm
GAMMA	0.57721566490153286060	euler-mascheroni constant[1]
PHI	1.61803398874989484820	golden ratio ($\sqrt{5}$ + 1)/2
CUBIT_VERSION	Varies, string value	current version of Cubit
VERSION	Varies, string value	current version of APREPRO
_ FORMAT	"%.10g"	default output format
_C_	"#"	default comment character

1 The euler-mascheroni constant is defined as the limit of 1 + 1/2 + ... + 1/s - log(s) as s approaches infinity.

Note that the output format is used to output both integers and floating point numbers. Therefore, it should use the %g format descriptor which will use either the decimal (%d), exponential (%e), or float (%f) format, whichever is shorter, with insignificant zeros suppressed. The table below illustrates the effect of different format specifications on the output of the variable PI and the value 1.0 . See the documentation of your C compiler for

Table 2. Effect of Various Output Format Specifications more information. For most cases, the default value is sufficient.

Format	PI Output	1.0 Output
%.10g	3.141592654	1
%.10e	3.1415926536e+00	1.0000000000e+00
%.10f	3.1415926536	1.0000000000

%.10d	1413754136	0000000000

APREPRO Units

Cubit uses a unitless coordinate system. For example, the command **brick x 10** creates a cube 10 units wide, but Cubit does not know whether those 10 units are 10 inches, 10 meters, 10 microns or 10 miles. The Aprepro Units() function facilitates the use of a unit system in Cubit's unitless environment.

The Aprepro **Units(svar)** function takes a single string parameter which identifies the desired unit system. If the specified unit system is recognized, then a set of variables are defined to facilitate working in that unit system. Dimensions can be multiplied by an appropriate unit variable to convert between various dimensions. For example, the statement **#{Units("in-lbf-s")}** defines variables useful when working in a coordinate system where one Cubit unit is one inch. The command **brick x {1*ft}** will create a cube 12 units wide, and the command **brick x {1*m}** will create a cube 39.37 units wide, the number of inches in 1 meter.

It is important to note that the Units() function does nothing more than define a set of useful variables. The function does not change the dimensions of existing geometry, nor does it change the scale of geometry imported from a file. For example, the following commands create two cubes, the first being 12 units wide and the second being 1 unit wide. The first cube remains 12 units wide, even after the second call to the Units() function:

```
#{Units("in-lbf-s")}
brick x 1*ft
#{Units("ft-lbf-s")}
brick x 1*ft
```

The Units() function returns a zero-length string if it is successful. If the Units() function fails (usually because the specified unit system was not recognized), a non-zero-length string containing an error message is returned.

The unit systems currently supported by the Units() function are: si, cgs, cgs-ev, shock, swap, ft-lbf-s, ft-lbm-s, in-lbf-s. For each of these unit systems, the following variables are defined by the Units() function:

Table 1. String Variables

Name	Value
Tout	Base Time Unit
lout	Base Length Unit
Aout	Base Acceleration Unit
Mout	Base Mass Unit
fout	Base Force Unit
vout	Base Velocity Unit
Vout	Base Volume Unit
dout	Base Density Unit
eout	Base Energy Unit
Pout	Base Power Unit
pout	Base Pressure Unit
Tout	Base Temperature Unit

Aout	Base Angle Unit

Time Variables

- sec
- second
- usec
- microsecond
- msec
- millisecond
- minute
- hr
- hour
- day
- yr
- year
- decade
- century

Length Variables

- m
- meter
- metre
- cm
- centimeter
- centimetre
- mm
- millimeter
- millimetre
- um
- micrometer
- micrometre
- km
- kilometer
- kilometre
- ft
- foot
- mi
- mile
- yd
- yard
- in
- inch
- mil

Acceleration Variables

- ga

Force Variables

- newton
- N

- dyne
- lbf
- kip
- kgf
- gf
- pdl
- poundal
- ounce

Mass Variables

- kg
- gram
- g
- lbm
- slug
- lbfs2pin

Velocity Variables

- mps
- fps
- mph
- ips
- kph
- kps

Volume Variables

- liter
- gal
- gallon

Density Variables

- gpcc
- kgpm3
- lbfs2pin4
- lbmpin3
- lbmpft3
- slugpft3

Power Variables

- W
- watt
- Hp

Energy Variables

- joule
- J
- ftlbf
- Btu

- erg
- calorie
- kwh
- therm
- tonTNT

Pressure Variables

- Pa
- pascal
- MPa
- GPa
- bar
- kbar
- Mbar
- psi
- ksi
- psf
- atm
- torr
- mHg
- mmHg
- inHg
- inH2O
- ftH20

Temperature Variables

- degK
- kelvin
- degC
- degF
- degR
- rankine
- eV

Angular Variables

- rad
- rev
- deg
- degree
- arcmin
- arcsec
- grade

APREPRO Functions

Several mathematical, CUBIT and string functions are implemented in *APREPRO*.

- Mathematical Functions
- CUBIT Functions
- String Functions

To cause a function to be used, you enter the name of the function followed by a list of zero or more arguments in parentheses. For example

sqrt(min(a,b*3))

uses the two functions **sqrt()** and **min()**. The arguments **a** and **b*3** are passed to **min()**. The result is then passed as an argument to **sqrt()**. The functions in *APREPRO* are listed below along with the number of arguments and a short description of their effect.

1. Mathematical Functions

The following mathematical functions are available in APREPRO. Check current APREPRO documentation for any updates to this list of functions.

Table 1. Mathematical Functions

Syntax	Description
abs(x)	Calculates the absolute value of **x**. |**x**|
acos(x)	Calculates the inverse cosine of **x**, returns radians
acosd(x)	Calculates the inverse cosine of **x**, returns degrees
acosh(x)	Calculates the inverse hyperbolic cosine of **x**
asin(x)	Calculates the inverse sine of **x**, returns radians
asind(x)	Calculates the inverse sine of **x**, returns degrees
asinh(x)	Calculates the inverse hyperbolic sine of **x**
atan(x)	Calculates the inverse tangent of **x**, returns radians
atan2(y,x)	Calculates the inverse tangent of **y/x**, returns radians
atan2d(y,x)	Calculates the inverse tangent of **y/x**, returns degrees
atand(x)	Calculates the inverse tangent of **x**, returns degrees
atanh(x)	Calculates the inverse hyperbolic tangent of **x**
ceil(x)	Calculates the smallest integer not less than **x**
cos(x)	Calculates the cosine of **x**, with **x** in radians

cosd(x)	Calculates the cosine of **x**, with **x** in degrees
cosh(x)	Calculates the hyperbolic cosine of **x**
d2r(x)	Converts degrees to radians.
dim(x,y)	Calculates x - min(**x,y**).
dist(x_1,y_1, x_2,y_2)	Calculates distance from **x_1,y_1** to **x_2,y_2**
exp(x)	Calculates e^x (Exponential)
floor(x)	Calculates the largest integer not greater than x.
fmod(x,y)	Calculates the floating-point remainder of x/y.
hypot(x,y)	Calculates **sqrt**(x^2+y^2)
int(x), [x]	Calculates the integer part of **x** truncated toward 0.
julday(mm, dd, yy)	Calculates the Julian day corresponding to **mm**/**dd**/**yy**.
juldayhms (mm, dd, yy, hh, mm, ss)	Calculates the Julian day corresponding to **mm**/**dd**/**yy** at **hh**:**mm**:**ss**
lgamma(x)	Calculates log(G(***x****))
ln(x), log(x)	Calculates the natural (base e) logarithm of **x**.
log1p(x)	Calculates log(1+**x**)
log10(x)	Calculates the base 10 logarithm of **x**.
max(x,y)	Calculates the maximum of **x** and **y**.
min(x,y)	Calculates the minimum of **x** and **y**.
polarX(r,a)	Calculates ***r*** ´ **cos**(***a***), **a** is in degrees
polarY(r,a)	Calculates ***r*** ´ **sin**(***a***), **a** is in degrees
r2d(x)	Converts radians to degrees.
rand(xl,xh)	Calculates a random number between **xl** and **xh**.
sign(x,y)	Calculates **x** ´ sgn(***y***)
sin(x)	Calculates the sine of **x**, with **x** in radians.
sind(x)	Calculates the sine of **x**, with **x** in degrees.

sinh(x)	Calculates the hyperbolic sine of **x**
sqrt(x)	Calculates the square root of **x**.
tan(x)	Calculates the tangent of **x**, with **x** in radians.
tand(x)	Calculates the tangent of **x**, with **x** in degrees.
tanh(x)	Calculates the hyperbolic tangent of **x**.
Vangle(x1,y1, x2,y2)	Calculates the angle between the vector $x_1 i + y_1 j$ and $x_2 i + y_2 j$. Returns radians.
Vangled(x1,y1, x2,y2)	Calculates the angle between the vector $x_1 i + y_1 j$ and $x_2 i + y_2 j$. Returns degrees.

2. CUBIT Functions

The following CUBIT Functions are available:

Table 2. CUBIT 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
FileExists(filename)	Checks if the given file exists. Returns non-zero if it does
GeometryEngineVersion(engine name)	Get the geometry version for the specified geometry engine
get_error_count()	Gets the current error count in CUBIT
set_error_count(val)	Sets the error count in CUBIT to given value
get_warning_count()	Gets the current warning count in CUBIT
HasFeature(feature_name)	Checks if the feature "feature_name" is available and returns nonzero if the feature is enabled
set_warning_count(val)	Sets the warning count in CUBIT to value
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".
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.
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.
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.
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.

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.
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.
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 relative size scaled jacobian shape and size shear and size shear skew stretch taper
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. Acceptable metrics include: shape aspect ration bet aspect ratio gam aspect ratio condition no diagonal ratio dimension distortion element volume jacobian relative size scaled jacobian shape and size shear and size shear skew stretch taper
MinSurfaceMeshQuality(id, "metric")	Returns the worst value of the specified element quality metric of all elements on the given surface. 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
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
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.
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.
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.
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.
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. .
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.
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.
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.
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.
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.
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".
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.
NumVolumes()	Returns the number of volumes in the model.
NumSurfaces()	Returns the number of surfaces in the model.
NumCurves()	Returns the number of curves in the model.
NumVertices()	Returns the number of vertices in the model.
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.
SessionId()	Returns a unique ID for each Cubit session.
DUMP()	Returns a list of all APREPRO variables with their values.
delete("var")	Deletes the APREPRO variable with the name var.
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.
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.

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.
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.
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.
TriAt(x, y, z, ordinal)	Returns the id of the triangle with the idless reference, x,y,z,ordinal.
HexAt(x, y, z, ordinal)	Returns the id of the hexahedra element 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.
WedgeAt(x, y, z, ordinal)	Returns the id of the wedge element 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.
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.

3.String Functions

A few useful string functions are available:

Table 3. String Functions

Syntax	Description
tolower(svar)	Translates all uppercase characters in **svar** to lowercase. It modifies **svar** and

	returns the resulting string.
toupper(svar)	Translates all lowercase character in **svar** to uppercase. It modifies **svar** and returns the resulting string.
execute(svar)	**svar** is parsed and executed as if it were a line read from the input file. For example, if **svar="b=sqrt(25.0)"**, then **#{execute(svar)}** returns the value **5** and sets **b = 5**. The expression **svar** is enclosed in delimiters prior to being executed, and it must be a valid expression or an error message will be printed.
rescan(svar)	Similar to **execute(svar)**, except that **svar** is not enclosed in delimiters prior to being executed. For example, if **svar = "Create Vertex {1+5} {sqrt(5)} {sqrt(6)}"**, then **#{rescan(svar)}** would print: **Create Vertex 6 2.236067977 2.449489743.** The difference between **execute(sv1)** and **rescan(sv2)** is that **sv1** must be a valid expression, but **sv2** can contain zero or more expressions.
getenv(svar)	Returns a string containing the value of the environment variable **svar**. If the environment variable is not defined, an empty string is returned.
get_word(n,svar,del)	Returns a string containing the nth word of svar. The words are separated by one or more of the characters in the string variable del
word_count(svar,del)	Returns the number of words in **svar**. Words are separated by one or more of the characters in the string variable del
strtod(svar)	Returns a double-precision floating-point number equal to the value represented by the character string pointed to by **svar**.
Print(msg)	Prints msg
PrintError(svar)	Outputs the string **svar** to stderr.
error(svar)	Outputs the string **svar** to stderr and then terminates the code with an error exit status
Quote(svar)	Returns the string **svar**, enclosed in single quotes.
TimerStart()	Starts the CPU timer
TimerStop()	Stops the CPU timer
Type(entity name)	Returns the type of the entity with the given name
Units(svar)	Sets variables useful for working in a unit system. See APREPRO Units.

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 (3 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}

Output:

Create Vertex 10 0

Create Vertex 9.510565163 3.090169944

Create Vertex 8.090169944 5.877852523

Create Vertex 5.877852523 8.090169944

Create Vertex 3.090169944 9.510565163

Create Vertex 6.123233765e-16 10

Create Vertex -3.090169944 9.510565163

Create Vertex -5.877852523 8.090169944

Create Vertex -8.090169944 5.877852523

Create Vertex -9.510565163 3.090169944

Create Vertex -10 1.224646753e-15

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 Additional Functionality

Additional APREPRO Functionality includes the following:

- File Inclusion
- Conditionals
- Loops

1. File Inclusion

APREPRO can read input from multiple files using the **include()** and **cinclude()** functions. If a line of the form:

{include(" filename")}

{include(string_variable)}

is read, *APREPRO* will open and begin reading from the file **filename**. A string variable can be used as the argument instead of a literal string value. When the end of the file is reached, it will be closed and *APREPRO* will continue reading from the previous file. The difference between **include** and **cinclude** is that if **filename** does not exist, **include** will terminate *APREPRO* with a fatal error, but **cinclude** will just write a warning message and continue with the current file. The **cinclude** function can be thought of as a conditional include, that is, include the file if it exists. Multiple include files are allowed and an included file can also include additional files. Approximately 16 levels of file inclusion can be used. This option can be used to set variables globally in several files. For example, if two or more input files share common points or dimensions, those dimensions can be set in one file that is included in the other files.

2. Conditionals

Portions of an input file can be conditionally processed through the use of the **{Ifdef(variable)}** or **Ifndef(variable)}** constructs. The syntax is:

#{Ifdef(variable)}

...Lines processed if **'variable'** is not equal to 0

#{Else}

...Lines processed if **'variable'** is equal to 0 or undefined

#{Endif}

#{Ifndef(variable)}

...Lines processed if **'variable'** is equal to 0 or undefined

#{Else}

...Lines processed if **'variable'** is not equal to 0

#{Endif}

The **{Else}** is optional. Note that if **variable** is undefined, its value is equal to zero. **Ifdef** constructs can be nested up to approximately 16 levels. A warning message will be printed if improper nesting is detected. **Ifdef(variable)}**, **{Ifndef(variable)}**, **{Else}**, and **{Endif}** are the only text parsed on a line. Text following these on the same line is ignored.

3. Loops

Repeated processing of a group of lines can be controlled with the **{loop(control)}**, **{endloop}** commands. The syntax is:

{loop(variable)}

...Process these lines **'variable'** times

{endloop}

Loops can be nested. A numerical variable or constant must be specified as the loop control specifier. You currently cannot use an algebraic expression such as **{loop(3+5)}**.

A loop may also be exited 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++}
   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 loop, but also terminates the most recent **#{If}** statement exactly as **#{EndIf}** would do. **#{Break}** statements should not be used outside of **#{If}** statements.

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}**

CubitInterface

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.

Class Member Functions

	init	Use init to initialize Cubit. Using a blank list as the input parameter is acceptable.
	destroy	Closes the current journal file.
	set_cubit_interrupt	This sets the global flag in Cubit that stops all interruptable processes.
	set_playback_paused_on_error	Sets whether or not playback is paused when an error occurs.
Bool	is_playback_paused_on_error	Gets whether or not playback is paused when an error occurs.
Bool	developer_commands_are_enabled	This checks to see whether developer commands are enabled.
str	get_version	Get the Cubit version.
str	get_revision_date	Get the Cubit revision date.
str	get_build_number	Get the Cubit build number.
str	get_acis_version	Get the Acis version number.
int	get_acis_version_as_int	Get the Acis version number as an int.
str	get_exodus_version	Get the Exodus version number.
str	get_graphics_version	Get the VTK version number.
	print_cmd_options	Used to print the command line options.
Bool	is_modified	Get the modified status of the model.
	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.
Bool	is_undo_save_needed	Get the status of the model relative to undo checkpointing.
	set_undo_saved	Set the status of the model relative to undo checkpointin.
Bool	is_command_echoed	Check the echo flag in cubit.
Bool	is_volume_meshable	Check if volume is meshable with current scheme.
	journal_commands	Set the journaling flag in cubit.
Bool	is_command_journaled	Check the journaling flag in cubit.
str	get_current_journal_file	Gets the current journal file name.
	cmd	Pass a command string into Cubit.
	silent_cmd	Pass a command string into Cubit and have it executed without being verbose at the command prompt.
[int]	parse_cubit_list	Parse a Cubit style list of IDs (1,2,4 to 19 by 3 or all) into a list of integers.
	print_raw_help	Used to print out help when a ?, & or ! is pressed.
int	get_error_count	Get the number of errors in the current Cubit session.
[str]	get_mesh_error_solutions	Get the paired list of mesh error solutions and help context cues.
float	get_view_distance	Get the distance from the camera to the model (from - at).
[float]	get_view_at	Get the camera 'at' point.
[float]	get_view_from	Get the camera 'from' point.
	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.
	unselect_entity	Unselect an entity that is currently selected.
Bool	is_perspective_on	Get the current perspective mode.
Bool	is_occlusion_on	Get the current occlusion mode.
Bool	is_scale_visibility_on	Get the current scale visibility setting.
Bool	is_select_partial_on	Get the current select partial setting.
int	get_rendering_mode	Get the current rendering mode.
	set_rendering_mode	Set the current rendering mode.

	clear_preview	Clear preview graphics without affecting other display settings.
str	get_pick_type	Get the current pick type.
float	get_mesh_edge_length	Get the length of a mesh edge.
float	get_meshed_volume_or_area	Get the total volume/area of a entity's mesh.
int	get_mesh_intervals	Get the interval count for a specified entity.
float	get_mesh_size	Get the mesh size for a specified entity.
float	get_auto_size	Get the auto size for a given set of volumes. 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.
float	get_quality_value	Get the metric value for a specified mesh entity.
str	get_mesh_scheme	Get the mesh scheme for the specified entity.
str	get_mesh_scheme_firmness	Get the mesh scheme firmness for the specified entity.
str	get_mesh_interval_firmness	Get the mesh interval firmness for the specified entity.
Bool	is_meshed	Determines whether a specified entity is meshed.
Bool	is_merged	Determines whether a specified entity is merged.
str	get_smooth_scheme	Get the smooth scheme for a specified entity.
int	get_hex_count	Get the count of hexes in the model.
int	get_pyramid_count	Get the count of pyramids in the model.
int	get_tet_count	Get the count of tets in the model.
int	get_quad_count	Get the count of quads in the model.
int	get_tri_count	Get the count of tris in the model.
int	get_edge_count	Get the count of edges in the model.
int	get_node_count	Get the count of nodes in the model.
str	get_element_count	Get the number of elements which have been assigned to a block, given a global element id, and will be exported.
int	get_volume_element_count	Get the count 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.
int	get_surface_element_count	Get the count of quads, and triangles in a surface. NOTE: This count does not distinguish between elements which have been put into a block or not.
Bool	volume_contains_tets	Determine whether a volume contains tets.
[int]	get_hex_sheet	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.
Bool	is_visible	Query visibility for a specific entity.
Bool	is_virtual	Query virtualality for a specific entity.
Bool	contains_virtual	Query virtualality of an entity's children.
[int]	get_source_surfaces	Get a list of a volume's sweep source surfaces.
[int]	get_target_surfaces	Get a list of a volume's sweep target surfaces.
int	get_common_curve_id	Given 2 surfaces, get the common curve id.
int	get_common_vertex_id	Given 2 curves, get the common vertex id.
str	get_merge_setting	Get the merge setting for a specified entity.
str	get_curve_type	Get the curve type for a specified curve.
str	get_surface_type	Get the surface type for a specified surface.
[float]	get_surface_normal	Get the surface normal for a specified surface.
[float]	get_surface_centroid	Get the surface centroid for a specified surface.
str	get_surface_sense	Get the surface sense for a specified surface.
[str]	get_entity_modeler_engine	Get the modeler engine type for a specified entity.
[float]	get_bounding_box	Get the bounding box for a specified entity.
[float]	get_total_bounding_box	Get the bounding box for a list of entities.
float	get_total_volume	Get the total volume for a list of volume ids.
str	get_entity_name	Get the name of a specified entity.
int	get_entity_color_index	Get the color of a specified entity.
Bool	is_multi_volume	Query whether a specified body is a multi volume body.
Bool	is_sheet_body	Query whether a specified volume is a sheet body.
Bool	is_interval_count_odd	Query whether a specified surface has an odd loop.

Bool	is_periodic	Query whether a specified surface or curve is periodic.
Bool	is_surface_planar	Query whether a specified surface is planer.
Bool	get_undo_enabled	
int	number_undo_commands	
[str]	get_aprepro_vars	Gets the current aprepro variable names.
str	get_aprepro_value_as_string	Gets the string value of an aprepro variable.
Bool	get_node_constraint	Query current setting for node constraint (move nodes to geometry).
str	get_vertex_type	Get the Vertex Types for a specified vertex on a specified surface. Vertex types include "side", "end", "reverse", "unknown".
[int]	get_relatives	Get the relatives (parents/children) of a specified entity.
[int]	get_adjacent_surfaces	Get a list of adjacent surfaces to a specified entity.
[int]	get_adjacent_volumes	Get a list of adjacent volumes to a specified entity.
[int]	get_entities	Get all geometry entities of a specified type.
[int]	get_list_of_free_ref_entities	Get all free entities of a given geometry type.
int	get_owning_body	Get the owning body for a specified entity.
int	get_owning_volume	Get the owning volume for a specified entity.
int	get_owning_volume_by_name	Get the owning volume for a specified entity.
float	get_curve_length	Get the length of a specified curve.
float	get_arc_length	Get the arc length of a specified curve.
float	get_distance_from_curve_start	Get the distance from a point on a curve to the curve's start point.
float	get_curve_radius	Get the radius of a specified arc.
[float]	get_curve_center	Get the center point of the arc.
float	get_surface_area	Get the area of a surface.
float	get_volume_area	Get the area of a volume.
float	get_hydraulic_radius_surface_area	Get the area of a hydraulic surface.
float	get_hydraulic_radius_volume_area	Get the area of a hydraulic volume.
[float]	get_center_point	Get the center point of a specified entity.
int	get_valence	Get the valence for a specific vertex.
float	get_distance_between	Get the distance between two vertices.
	print_surface_summary_stats	Print the surface summary stats to the console.
	print_volume_summary_stats	Print the volume summary stats to the console.
int	get_volume_count	Get the current number of volumes.
int	get_body_count	Get the current number of bodies.
int	get_surface_count	Get the current number of surfaces.
int	get_vertex_count	Get the current number of vertices.
int	get_curve_count	Get the current number of curves.
int	get_curve_count	Get the current number of curves in the passed-in volumes.
Bool	is_catia_engine_available	Determine whether catia engine is available.
[int]	evaluate_exterior_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)
[int]	get_small_curves	Get the list of small curves for a list of volumes.
[int]	get_smallest_curves	Get a list of the smallest curves in the list of volumes. The number returned is specified by 'num_to_return'.
[int]	get_small_surfaces	Get the list of small surfaces for a list of volumes.
[int]	get_narrow_surfaces	Get the list of narrow surfaces for a list of volumes.
[int]	get_small_and_narrow_surfaces	Get the list of small or narrow surfaces from a list of volumes.
[int]	get_surfs_with_narrow_regions	Get the list of surfaces with narrow regions.
[int]	get_small_volumes	Get the list of small volumes from a list of volumes.
[int]	get_blend_surfaces	Get the list of blend surfaces for a list of volumes.
[int]	get_small_loops	Get the list of close loops (surfaces) for a list of volumes.
[int]	get_tangential_intersections	Get the list of bad tangential intersections for a list of volumes.
[int]	get_coincident_vertices	
[[str]]	get_solutions_for_near_coincident_vertices	Get lists of display strings and command strings for near

		coincident vertices.
[[str]]	get_solutions_for_imprint_merge	Get lists of display strings and command strings for imprint/merge solutions.
[[str]]	get_solutions_for_small_surfaces	Get lists of display, preview and command strings for small surface solutions.
[[str]]	get_solutions_for_small_curves	Get lists of display, preview and command strings for small curve solutions.
[[str]]	get_solutions_for_surfaces_with_narrow_regions	Get lists of display, preview and command strings for surfaces with narrow regions solutions.
[int]	get_overlapping_volumes	Get the list of overlapping volumes for a list of volumes.
[[int]]	get_mergeable_vertices	Get the list of mergeable vertices from a list of volumes/bodies.
[[str]]	get_solutions_for_blends	Get the solution list for a given blend surface.
[[int]]	get_blend_chains	Queries the blend chains for a surface.
float	get_merge_tolerance	Get the current merge tolerance value.
str	get_exodus_entity_name	Get the name associated with an exodus entity.
str	get_exodus_entity_description	Get the description associated with an exodus entity.
[float]	get_all_exodus_times	Open an exodus file and get a vector of all stored time stamps.
int	get_block_id	Get the associated block id for a specific curve, surface, or volume.
[int]	get_block_ids	Get list of block ids from a mesh geometry file.
[int]	get_block_id_list	Get a list of all blocks.
[int]	get_nodeset_id_list	Get a list of all nodesets.
[int]	get_sideset_id_list	Get a list of all sidesets.
[int]	get_bc_id_list	Get a list of all bcs of a specified type.
str	get_bc_name	Get the name for the specified bc.
[int]	get_nodeset_id_list_for_bc	Get a list of all nodesets the specified bc is applied to.
[int]	get_sideset_id_list_for_bc	Get a list of all sidesets the specified bc is applied to.
int	get_next_sideset_id	Get a next available sideset id.
int	get_next_nodeset_id	Get a next available nodeset id.
int	get_next_block_id	Get a next available block id.
[int]	get_block_volumes	Get a list of volume ids associated with a specific block.
[int]	get_block_surfaces	Get a list of surface associated with a specific block.
[int]	get_block_curves	Get a list of curve associated with a specific block.
[int]	get_block_vertices	Get a list of vertices associated with a specific block.
[int]	get_block_nodes	Get a list of nodes associated with a specific block.
[int]	get_block_edges	Get a list of edges associated with a specific block.
[int]	get_block_tris	Get a list of tris associated with a specific block.
[int]	get_block_faces	Get a list of faces associated with a specific block.
[int]	get_block_pyramids	Get a list of pyramids associated with a specific block.
[int]	get_block_tets	Get a list of tets associated with a specific block.
[int]	get_block_hexes	Get a list of hexes associated with a specific block.
[int]	get_volume_hexes	get the list of any hex elements in a given volume
[int]	get_volume_tets	get the list of any tet elements in a given volume
[int]	get_nodeset_volumes	Get a list of volume ids associated with a specific nodeset.
[int]	get_nodeset_surfaces	Get a list of surface ids associated with a specific nodeset.
[int]	get_nodeset_curves	Get a list of curve ids associated with a specific nodeset.
[int]	get_nodeset_vertices	Get a list of vertex ids associated with a specific nodeset.
[int]	get_nodeset_nodes	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.
[int]	get_nodeset_nodes_inclusive	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.
[int]	get_sideset_curves	Get a list of curve ids associated with a specific sideset.
[int]	get_curve_edges	get the list of any edge elements on a given curve

[int]	get_sideset_surfaces	Get a list of any surfaces in a sideset.
[int]	get_sideset_quads	Get a list of any quads in a sideset.
[int]	get_surface_quads	get the list of any quad elements on a given surface
[int]	get_surface_tris	get the list of any tri elements on a given surface
str	get_entity_sense	Get the sense of a sideset item.
str	get_wrt_entity	Get the with-respect-to entity.
Bool	is_using_shells	Get the shell use for a sideset.
[str]	get_geometric_owner	Get a list of geometric owners given a list of mesh entities.
[int]	get_volume_nodes	Get list of node ids owned by a volume. Excludes nodes owned by bounding surfs, curves and verts.
[int]	get_surface_nodes	Get list of node ids owned by a surface. Excludes nodes owned by bounding curves and verts.
[int]	get_curve_nodes	Get list of node ids owned by a curve. Excludes nodes owned by bounding vertices.
int	get_vertex_node	Get the node owned by a vertex.
int	get_id_from_name	Get id for a named entity.
[int]	get_group_groups	Get group groups (groups that are children of another group).
[int]	get_group_bodies	Get group bodies (bodies that are children of a group).
[int]	get_group_volumes	Get group volumes (volumes that are children of a group).
[int]	get_group_surfaces	Get group surfaces (surfaces that are children of a group).
[int]	get_group_curves	Get group curves (curves that are children of a group).
[int]	get_group_vertices	Get group vertices (vertices that are children of a group).
[int]	get_group_nodes	Get group nodes (nodes that are children of a group).
[int]	get_group_edges	Get group edges (edges that are children of a group).
[int]	get_group_quads	Get group quads (quads that are children of a group).
[int]	get_group_tris	Get group tris (tris that are children of a group).
[int]	get_group_tets	Get group tets (tets that are children of a group).
[int]	get_group_hexes	Get group hexes (hexes that are children of a group).
int	get_next_group_id	Get the next available group id from Cubit.
	delete_all_groups	Delete all groups.
	delete_group	Delete a specific group.
	set_max_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	Create a new group.
	remove_entity_from_group	Remove a specific entity from a specific group.
	add_entity_to_group	Add a specific entity to a specific group.
[int]	get_mesh_group_parent_ids	Get the group ids which are parents to the indicated mesh element.
Bool	is_mesh_element_in_group	Indicates whether a mesh element is in a group.
Bool	is_part_of_list	Routine to check for the presence of an id in a list of ids.
int	get_last_id	Get the id of the last created entity of the given type.
str	get_assembly_classification_level	Get Classification Level for metadata.
str	get_assembly_classification_category	Get Classification Category for metadata.
str	get_assembly_weapons_category	Get Weapons Category for metadata.
str	get_assembly_metadata	Get metadata for a specified volume id.
Bool	is_assembly_metadata_attached	Determine whether metadata is attached to a specified volume.
str	get_assembly_name	Get the stored name of an assembly node.
str	get_assembly_path	Get the stored path of an assembly node.
str	get_assembly_description	Get the stored description of an assembly node.
int	get_assembly_instance	Get the stored instance number of an assembly node.
str	get_assembly_file_format	Get the stored file format of an assembly node.

str	get_assembly_units	Get the stored units measure of an assembly node.
str	get_assembly_material_description	Get the stored material description of an assembly part.
str	get_assembly_material_specification	Get the stored material specification of an assembly part.
int	get_exodus_id	Get the exodus/genesis id for this element.
str	get_geometry_owner	Get the geometric owner of this mesh element.
[int]	get_connectivity	Get the list of node ids contained within a mesh entity.
[int]	get_expanded_connectivity	Get the list of node ids contained within a mesh entity, including interior nodes.
[int]	get_sub_elements	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.
[float]	get_nodal_coordinates	Get the nodal coordinates for a given node id.
Bool	get_node_position_fixed	Query "fixedness" state of node. A fixed node is not affecting by smoothing.
str	get_sideset_element_type	Get the element type of a sideset.
str	get_block_element_type	Get the element type of a block.
int	get_exodus_element_count	Get the number of elements in a exodus entity.
int	get_block_attribute_count	Get the number of attributes in a block.
float	get_block_attribute_value	Get a specific block attribute value.
[str]	get_valid_block_element_types	Get a list of potential element types for a block.
int	get_nodeset_node_count	Get the number of nodes in a nodeset.
int	get_geometry_node_count	
str	get_mesh_element_type	Get the mesh element type contained in the specified geometry.
Bool	is_on_thin_shell	Determine whether a BC is on a thin shell. Valid for temperature, convection and heatflux.
Bool	temperature_is_on_solid	Determine whether a BC temperature is on a solid. Valid for convection and temperature.
Bool	convection_is_on_solid	Determine whether a BC convection is on a solid. Valid for convection.
Bool	convection_is_on_shell_area	Determine whether a BC convection is on a shell top or bottom. Valid for convection.
float	get_convection_coefficient	Get the convection coefficient.
float	get_bc_temperature	Get the temperature. Valid for convection, temperature.
Bool	temperature_is_on_shell_area	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	Determine whether a BC heatflux is on a shell area.
float	get_heatflux_on_area	Get the heatflux on a specified area.
int	get_cfd_type	Get the cfd subtype for a specified cfd BC.
float	get_contact_pair_friction_value	Get the contact pair's friction value.
float	get_contact_pair_tolerance_value	Get the contact pair's tolerance value.
Bool	get_contact_pair_tied_state	Get the contact pair's tied state.
Bool	get_contact_pair_general_state	Get the contact pair's general state.
Bool	get_contact_pair_exterior_state	Get the contact pair's exterior state.
int	get_displacement_coord_system	Get the displacement's coordinate system id.
str	get_displacement_combine_type	Get the displacement's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine".
float	get_pressure_value	Get the pressure value.
str	get_pressure_function	Get the pressure function.
float	get_force_magnitude	Get the force magnitude from a force.
float	get_moment_magnitude	Get the moment magnitude from a force.
[float]	get_force_direction_vector	Get the direction vector from a force.
[float]	get_force_moment_vector	Get the moment vector from a force.
str	get_constraint_type	Get the type of a specified constraint.
str	get_constraint_reference_point	Get the reference point of a specified constraint.
str	get_constraint_dependent_entity_point	Get the dependent entity of a specified constraint.
float	get_material_property	

int	get_media_property	
[str]	get_material_name_list	
[str]	get_media_name_list	
	set_label_type	
int	get_label_type	
Body	body	Gets the body object from an ID.
Volume	volume	Gets the volume object from an ID.
Surface	surface	Gets the surface object from an ID.
Curve	curve	Gets the curve object from an ID.
Vertex	vertex	Gets the vertex object from an ID.
	reset	Executes a reset within cubit.
Body	brick	Creates a brick of specified width, depth, and height.
Body	sphere	Creates all or part of a sphere.
Body	prism	Creates a prism of the specified dimensions.
Body	pyramid	Creates a pyramid of the specified dimensions.
Body	cylinder	creates a cylinder of the specified dimensions
Body	torus	creates a torus of the specified dimensions
Vertex	create_vertex	Creates a vertex at a x,y,z.
Curve	create_curve	Creates a curve between two vertices.
Body	create_surface	Creates a surface from boundary curves.
[Body]	sweep_curve	Create a Body or a set of Bodies from a swept curve.
Body	copy_body	Creates a copy of the input Body.
[Body]	tweak_surface_offset	Performs a tweak surface offset command.
[Body]	tweak_surface_remove	Removes a surface from a body and extends the surrounding surfaces if extend_ajoining is true.
[Body]	tweak_curve_remove	Removes a curve from a body and extends the surrounding surface to fill the gap.
[Body]	tweak_curve_offset	Performs a tweak curve offset command.
[Body]	tweak_vertex_fillet	Performs a tweak vertex fillet command.
[Body]	subtract	Performs a boolean subtract operation.
[Body]	unite	Performs a boolean unite operation.
	move	Moves the Entity the specified vector.
	scale	Scales the Entity according to the specified factor.
	reflect	Reflect the Entity about the specified axis.
[int]	get_volumes_for_node	
int	get_mesh_error_count	

Class Variables

const int CI_ERROR -

Member Function Documentation

---

init(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

---

destroy()

Closes the current journal file.

---

set_cubit_interrupt(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_playback_paused_on_error(pause)

Sets whether or not playback is paused when an error occurs.

Parameters

pause True if playback should be paused when an error occurs.

Bool is_playback_paused_on_error()

Gets whether or not playback is paused when an error occurs.

Return

True if playback should be paused when an error occurs.

Bool developer_commands_are_enabled()

This checks to see whether developer commands are enabled.

Return

True if developer commands are enabled, otherwise False

---

str get_version()

Get the Cubit version.

Return

A string containing the current version of Cubit

---

str get_revision_date()

Get the Cubit revision date.

Return

A string containing Cubit's last date of revision

---

str get_build_number()

Get the Cubit build number.

Return

A string containing the current Cubit build number

---

str get_acis_version()

Get the Acis version number.

Return

A string containing the Acis version number

---

int get_acis_version_as_int()

Get the Acis version number as an int.

Return

An integer containing the Acis version number

---

str get_exodus_version()

Get the Exodus version number.

Return

A string containing the Exodus version number

---

str get_graphics_version()

Get the VTK version number.

Return

A string containing the VTK version number

---

print_cmd_options()

Used to print the command line options.

Bool is_modified()

Get the modified status of the model.

Return

A boolean indicating whether the model has been modified

---

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.

Bool is_undo_save_needed()

Get the status of the model relative to undo checkpointing.

Return

A boolean indicating whether the model has been modified

---

set_undo_saved()

Set the status of the model relative to undo checkpointin.

Bool is_command_echoed()

Check the echo flag in cubit.

Return

A boolean indicating whether commands should be echoed in Cubit

---

Bool is_volume_meshable(volume_id)

Check if volume is meshable with current scheme.

Parameters

volume_id

Return

A boolean indicating whether volume is meshable with current scheme

---

journal_commands(state)

Set the journaling flag in cubit.

Parameters

state A boolean that turns journaling on (1) and off (0)

Bool is_command_journaled()

Check the journaling flag in cubit.

Return

A boolean indicating whether commands are journaled by Cubit

str get_current_journal_file()

Gets the current journal file name.

Return

The current journal file name.

cmd(input_string)

Pass a command string into Cubit.

Example

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

Parameters

input_string Pointer to a string containing a complete Cubit command

silent_cmd(input_string)

Pass a command string into Cubit and have it executed without being verbose at the command prompt.

Example

```
cubit.silent_cmd("display")
```

Parameters

input_string Pointer to a string containing a complete Cubit command

[int] parse_cubit_list(type, int_list, include_sheet_bodies)

Parse a Cubit style list of IDs (1,2,4 to 19 by 3 or all) into a list of integers.

Parameters

type The specific entity type represented by the list of IDs
int_list The string that contains the user's ID list
include_sheet_bodies - include sheet bodies in the integer list (1 if yes, 0 if no)

Return

A vector of validated integers

print_raw_help(input_line, order_dependent, 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'

---

int get_error_count()

Get the number of errors in the current Cubit session.

Return

The number of errors in the Cubit session.

[str] get_mesh_error_solutions(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

Return

List of 'married' strings. First string is solution text. Second string is help context cue. Third string is command_panel cue.

---

float get_view_distance()

Get the distance from the camera to the model (from - at).

Return

Distance from the camera to the model

[float] get_view_at()

Get the camera 'at' point.

Return

The xyz coordinates of the camera's current position

[float] get_view_from()

Get the camera 'from' point.

Return

The xyz coordinates of the camera's from position

---

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.

---

unselect_entity(entity_type, entity_id)

Unselect an entity that is currently selected.

Example

```
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

Bool is_perspective_on()

Get the current perspective mode.

Return

True if perspective is on, otherwise false

Bool is_occlusion_on()

Get the current occlusion mode.

Return

True if occlusion is on, otherwise false

Bool is_scale_visibility_on()

Get the current scale visibility setting.

Return

True if scale is visible, otherwise false

Bool is_select_partial_on()

Get the current select partial setting.

Return

True if partial select is on, otherwise false

---

int get_rendering_mode()

Get the current rendering mode.

Return

The current rendering mode of the graphics subsystem

---

set_rendering_mode(mode)

Set the current rendering mode.

Parameters

mode Integer associated with the rendering mode. Options are 1,7,2,8, or 5

---

clear_preview()

Clear preview graphics without affecting other display settings.

---

str get_pick_type()

Get the current pick type.

Return

The current pick type of the graphics system

---

float get_mesh_edge_length(edge_id)

Get the length of a mesh edge.

Parameters

edge_id Specifies the id of the edge

Return

The length of the mesh edge

---

float get_meshed_volume_or_area(geom_type, entity_ids)

Get the total volume/area of a entity's mesh.

Example

```
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

Return

The entity's meshed volume or area

---

int get_mesh_intervals(geom_type, entity_id)

Get the interval count for a specified entity.

Example

```
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

Return

The entity's interval count

---

float get_mesh_size(geom_type, entity_id)

Get the mesh size for a specified entity.

Example

```
mesh_size = cubit.get_meshed_size("volume", 2)
```

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Return

The entity's mesh size

---

float get_auto_size(volume_id_list, size)

Get the auto size for a given set of volumes. 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.

Example

```
double get_auto_size(volume_list)
```

Parameters

volume_id_list
size The auto factor for the AutoSizeTool

Return

The interval size from the AutoSizeTool

---

float get_quality_value(mesh_type, mesh_id, 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)

Return

The value of the quality metric

---

str get_mesh_scheme(geom_type, entity_id)

Get the mesh scheme for the specified entity.

Example

```
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

Return

The entity's meshing scheme

---

str get_mesh_scheme_firmness(geom_type, entity_id)

Get the mesh scheme firmness for the specified entity.

Example

```
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

Return

The entity's meshing firmness (HARD, LIMP, SOFT, etc)

---

str get_mesh_interval_firmness(geom_type, entity_id)

Get the mesh interval firmness for the specified entity.

Example

```
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

Return

The entity's meshing firmness (HARD, LIMP, SOFT, etc)

Bool is_meshed(geom_type, entity_id)

Determines whether a specified entity is meshed.

Example

```
if cubit.is_meshed("surface", 137):
```

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Bool is_merged(geom_type, entity_id)

Determines whether a specified entity is merged.

Example

```
if cubit.is_merged("surface", 137):
```

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

---

str get_smooth_scheme(geom_type, entity_id)

Get the smooth scheme for a specified entity.

Example

```
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

Return

The smooth scheme associated with the entity

---

int get_hex_count()

Get the count of hexes in the model.

Return

The number of hexes in the model

---

int get_pyramid_count()

Get the count of pyramids in the model.

Return

The number of pyramids in the model

---

int get_tet_count()

Get the count of tets in the model.

Return

The number of tets in the model

---

int get_quad_count()

Get the count of quads in the model.

Return

The number of quads in the model

---

int get_tri_count()

Get the count of tris in the model.

Return

The number of tris in the model

---

int get_edge_count()

Get the count of edges in the model.

Return

The number of edges in the model

---

int get_node_count()

Get the count of nodes in the model.

Return

The number of nodes in the model

---

int get_element_count()

Get the number of elements which have been assigned to a block, given a global element id, and will be exported.

Return

The number of elements in the model

---

int get_volume_element_count(volume_id)

Get the count 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.

Parameters

volume_id

Return

The number of elements (hexes, tets, wedges, and pyramids) in a volume

---

int get_surface_element_count(surface_id)

Get the count of quads, and triangles in a surface. NOTE: This count does not distinguish between elements which have been put into a block or not.

Parameters

surface_id

Return

The number of elements (both quads and tris) in a surface

Bool volume_contains_tets(volume_id)

Determine whether a volume contains tets.

Parameters

volume_id

Return

bool

[int] get_hex_sheet(node_id_1, 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.

Parameters

node_id_1
node_id_2

Return

A list of hex ids in the hex sheet

Bool is_visible(geom_type, entity_id)

Query visibility for a specific entity.

Example

```
if cubit.is_visible("volume", 4)):
```

ERROR: EOF in multi-line statement

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Bool is_virtual(geom_type, entity_id)

Query virtualality for a specific entity.

Example

```
if cubit.is_virtual("surface", 134)):
```

ERROR: EOF in multi-line statement

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Bool contains_virtual(geom_type, entity_id)

Query virtualality of an entity's children.

Example

```
if cubit.contains_virtual("surface", 134)):
```

ERROR: EOF in multi-line statement

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

[int] get_source_surfaces(volume_id)

Get a list of a volume's sweep source surfaces.

Parameters

volume_id Specifies the volume id

Return

List of surface ids

[int] get_target_surfaces(volume_id)

Get a list of a volume's sweep target surfaces.

Parameters

volume_id Specifies the volume id

Return

List of surface ids

---

int get_common_curve_id(surface_1_id, 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

Return

The id of the curve common to the two surfaces

---

int get_common_vertex_id(curve_1_id, 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

Return

The id of the vertex common to the two curves, 0 if there is none

---

str get_merge_setting(geom_type, entity_id)

Get the merge setting for a specified entity.

Example

```
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

Return

A text string that indicates the merge setting for the entity

---

str get_curve_type(curve_id)

Get the curve type for a specified curve.

Parameters

curve_id ID of the curve

Return

Type of curve

---

str get_surface_type(surface_id)

Get the surface type for a specified surface.

Parameters

surface_id ID of the surface

Return

Type of surface

[float] get_surface_normal(surface_id)

Get the surface normal for a specified surface.

Parameters

surface_id ID of the surface

Return

surface normal at the center

[float] get_surface_centroid(surface_id)

Get the surface centroid for a specified surface.

Parameters

surface_id ID of the surface

Return

surface centroid

---

str get_surface_sense(surface_id)

Get the surface sense for a specified surface.

Parameters

surface_id ID of the surface

Return

surface sense as "Reversed" or "Forward" or "Both"

[str] get_entity_modeler_engine(geom_type, entity_id)

Get the modeler engine type for a specified entity.

Example

```
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

Return

A vector of modeler engines associated with this entity

[float] get_bounding_box(geom_type, entity_id)

Get the bounding box for a specified entity.

Example

```
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

Return

A vector of coordinates describing the entity's bounding box. Ten (10) values will be returned in axis-min, axis-max, and axis-range order, repeated for x-axis, y-axis, and z-axis and ending with the total diagonal measure.

[float] get_total_bounding_box(geom_type, entity_list)

Get the bounding box for a list of entities.

Example

```
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

Return

A vector of coordinates for the entity's bounding box. Twelve (12) values will be returned in xyz set order repeated four (4) times per set.

---

float get_total_volume(volume_list)

Get the total volume for a list of volume ids.

Parameters

volume_list List of volume ids

Return

The total volume of all volumes indicated in the id list

---

str get_entity_name(geom_type, entity_id)

Get the name of a specified entity.

Example

```
name = cubit.get_entity_name("vertex", 22)
```

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Return

The name of the entity

---

int get_entity_color_index(entity_type, entity_id)

Get the color of a specified entity.

Example

```
color_index = cubit.get_entity_color_index("curve", 33)
```

Parameters

entity_type Specifies the type of the entity
entity_id Specifies the id of the entity

Return

The color of the entity

Bool is_multi_volume(body_id)

Query whether a specified body is a multi volume body.

Parameters

body_id Id of the body

Return

True if body contains multiple volumes, otherwise false.

Bool is_sheet_body(volume_id)

Query whether a specified volume is a sheet body.

Parameters

volume_id Id of the volume

Return

True if volume is a sheet body, otherwise false

Bool is_interval_count_odd(surface_id)

Query whether a specified surface has an odd loop.

Parameters

surface_id Id of the surface

Return

True if surface is/contains an odd looop, otherwise false.

Bool is_periodic(geom_type, entity_id)

Query whether a specified surface or curve is periodic.

Example

```
if cubit.is_periodic("surface", 22):
```

Parameters

geom_type Specifies the geometry type of the entity
entity_id Specifies the id of the entity

Return

True is entity is periodic, otherwise false

Bool is_surface_planar(surface_id)

Query whether a specified surface is planar.

Example

```
if cubit.is_surface_planar(22):
```

Parameters

surface_id Specifies the id of the surface

Return

True is surface is planar, otherwise false

Bool get_undo_enabled()

---

int number_undo_commands()

[str] get_aprepro_vars()

Gets the current aprepro variable names.

---

str get_aprepro_value_as_string(var_name)

Gets the string value of an aprepro variable.

Parameters

var_name

Bool get_node_constraint()

Query current setting for node constraint (move nodes to geometry).

Return

True if constrained, otherwise false

---

str get_vertex_type(surface_id, 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

Return

The type -- "side", "end", "reverse", or "unknown"

[int] get_relatives(source_geom_type, source_id, target_geom_type)

Get the relatives (parents/children) of a specified entity.

Example

```
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

Return

A list of ids of the target geometry type

[int] get_adjacent_surfaces(geom_type, entity_id)

Get a list of adjacent surfaces to a specified entity.

Example

```
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

Return

A list of surfaces ids

[int] get_adjacent_volumes(geom_type, entity_id)

Get a list of adjacent volumes to a specified entity.

Example

```
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

Return

A list of volume ids

[int] get_entities(geom_type, include_sheet_bodies)

Get all geometry entities of a specified type.

Example

```
entity_id_list = cubit.get_entities("volume")
```

Parameters

geom_type Specifies the geometry type of the entity

include_sheet_bodies If true, then those routines requesting volumes or bodies will have sheet bodies returned. Normally, when requesting volume lists, sheet bodies are specifically excluded. Some parts of the Cubit interface need to see sheet bodies when requesting volumes, hence, the parameter.

Return

A list of ids of the specified geometry type

[int] get_list_of_free_ref_entities(geom_type)

Get all free entities of a given geometry type.

Example

```
free_curve_id_list =
cubit.get_list_of_free_ref_entities("curve")
```

Parameters

geom_type Specifies the geometry type of the free entity

Return

A list of ids of the specified geometry type

---

int get_owning_body(geom_type, entity_id)

Get the owning body for a specified entity.

Example

```
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

Return

ID of the specified entity's owning body

---

int get_owning_volume(geom_type, entity_id)

Get the owning volume for a specified entity.

Example

```
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

Return

ID of the specified entity's owning volume

---

int get_owning_volume_by_name(entity_name)

Get the owning volume for a specified entity.

Example

```
volume_id = cubit.get_owning_volume_by_name("TipSurface")
```

Parameters

entity_name Specifies the name (supplied by Cubit) of the entity

Return

ID of the specified entity's owning volume or 0 if name is unknown

---

float get_curve_length(curve_id)

Get the length of a specified curve.

Parameters

curve_id ID of the curve

Return

Length of the curve

---

float get_arc_length(curve_id)

Get the arc length of a specified curve.

Parameters

curve_id ID of the curve

Return

Arc length of the curve

---

float get_distance_from_curve_start(x, y, z, 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

Return

Distance from the xyz to the curve start

---

float get_curve_radius(curve_id)

Get the radius of a specified arc.

Parameters

curve_id ID of the curve

Return

Radius of the curve

[float] get_curve_center(curve_id)

Get the center point of the arc.

Parameters

curve_id ID of the curve

Return

x, y, z center point of the curve in a vector

---

float get_surface_area(surface_id)

Get the area of a surface.

Parameters

surface_id ID of the surface

Return

Area of the surface

---

float get_volume_area(volume_id)

Get the area of a volume.

Parameters

volume_id ID of the volume

Return

Area of the volume

---

float get_hydraulic_radius_surface_area(surface_id)

Get the area of a hydraulic surface.

Parameters

surface_id ID of the surface

Return

Hydraulic area of the surface

---

float get_hydraulic_radius_volume_area(volume_id)

Get the area of a hydraulic volume.

Parameters

volume_id ID of the volume

Return

Hydraulic area of the volume

[float] get_center_point(entity_type, entity_id)

Get the center point of a specified entity.

Example

```
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

Return

Vector of doubles representing x y z

---

int get_valence(vertex_id)

Get the valence for a specific vertex.

Parameters

vertex_id ID of vertex

---

float get_distance_between(vertex_id_1, vertex_id_2)

Get the distance between two vertices.

Parameters

vertex_id_1 ID of vertex 1 vertex_id_2 ID of vertex 2 /return distance
vertex_id_2

---

print_surface_summary_stats()

Print the surface summary stats to the console.

---

print_volume_summary_stats()

Print the volume summary stats to the console.

The number of bodies in the current model, if any

---

int get_body_count()

Get the current number of bodies.

Return

The number of volumes in the current model, if any

---

int get_volume_count()

Get the current number of volumes.

Return

The number of volumes in the current model, if any

---

int get_surface_count()

Get the current number of surfaces.

Return

The number of surfaces in the current model, if any

---

int get_vertex_count()

Get the current number of vertices.

Return

The number of vertices in the current model, if any

---

int get_curve_count()

Get the current number of curves.

Return

The number of curves in the current model, if any

---

int get_curve_count(target_volume_ids)

Get the current number of curves in the passed-in volumes.

Parameters

target_volume_ids

Return

The number of curves in the volumes

Bool is_catia_engine_available()

Determine whether catia engine is available.

Return

True if catia engine is available, otherwise false

[int] evaluate_exterior_angle(curve_list, 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

Return

a list of curve ids that meet the angle test.

[int] get_small_curves(target_volume_ids, mesh_size)

Get the list of small curves for a list of volumes.

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

Return

List of small curve ids

[int] get_smallest_curves(target_volume_ids, num_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

Return

List of smallest curve ids

[int] get_small_surfaces(target_volume_ids, mesh_size)

Get the list of small surfaces for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.
mesh_size Indicate the mesh size used as the threshold

Return

List of small surface ids

[int] get_narrow_surfaces(target_volume_ids, mesh_size)

Get the list of narrow surfaces for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.
mesh_size Indicate the mesh size used as the threshold

Return

List of small surface ids

[int] get_small_and_narrow_surfaces(target_ids, small_area, small_curve_size)

Get the list of small or narrow surfaces from a list of volumes.

Parameters

target_ids
small_area Indicate the area threshold
small_curve_size Indicate size for 'narrowness'

Return

List of small or narrow surface ids

[int] get_surfs_with_narrow_regions(target_ids, narrow_size)

Get the list of surfaces with narrow regions.

Parameters

target_ids
narrow_size Indicate the size that defines 'narrowness'

Return

List of surface ids

[int] get_small_volumes(target_volume_ids, mesh_size)

Get the list of small volumes from a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.
mesh_size Indicate the mesh size used as the threshold

Return

List of small volume ids

[int] get_blend_surfaces(target_volume_ids)

Get the list of blend surfaces for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine. List of blend surface ids

---

[int] get_small_loops(target_volume_ids, mesh_size)

Get the list of close loops (surfaces) for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.
mesh_size Indicate the mesh size used as the threshold

Return

List of close loop (surface) ids

[int] get_tangential_intersections(target_volume_ids, upper_bound, lower_bound)

Get the list of bad tangential intersections for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.
upper_bound Upper threshold angle
lower_bound Lower threshold angle

Return

List of surface ids associated with bad tangential angles

[int] get_coincident_vertices(target_volume_ids, high_tolerance)

Parameters

target_volume_ids
high_tolerance

Return

Paired list of vertex ids considered coincident

[[str]] get_solutions_for_near_coincident_vertices(vertex_id1, vertex_id2)

Get lists of display strings and command strings for near coincident vertices.

Parameters

vertex_id1
vertex_id2

Return

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.

[[str]] get_solutions_for_imprint_merge(surface_id1, 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
surface_id2

Return

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.

[[str]] get_solutions_for_small_surfaces(surface_id, small_curve_size, 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

Return

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.

[[str]] get_solutions_for_small_curves(curve_id, small_curve_size, 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

Return

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.

[[str]] get_solutions_for_surfaces_with_narrow_regions(surface_id, small_curve_size, 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

Return

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.

[int] get_overlapping_volumes(target_volume_ids)

Get the list of overlapping volumes for a list of volumes.

Parameters

target_volume_ids List of volume ids to examine.

Return

volume_list List of overlapping volumes ids

[[int]] get_mergeable_vertices(target_volume_ids)

Get the list of mergeable vertices from a list of volumes/bodies.

Parameters

target_volume_ids List of volume ids to examine.

Return

list of lists of mergeable vertices (potentially more than a pair)

[[str]] get_solutions_for_blends(surface_id)

Get the solution list for a given blend surface.

Parameters

surface_id the surface being queried

Return

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.

[[int]] get_blend_chains(surface_id)

Queries the blend chains for a surface.

Parameters

surface_id surface to retrieve the blend chains from

Return

A set of lists of id's in each blend chain

---

float get_merge_tolerance()

Get the current merge tolerance value.

Return

The value of the current merge tolerance

---

str get_exodus_entity_name(entity_type, entity_id)

Get the name associated with an exodus entity.

Example

```
entity_name = cubit.get_exodus_entity_name("sideset", 33)
```

Parameters

entity_type "block", "sideset", nodeset"
entity_id Id of the entity in question

Return

Name of the entity or "" if none

---

str get_exodus_entity_description(entity_type, entity_id)

Get the description associated with an exodus entity.

Example

```
entity_description =
cubit.get_exodus_entity_description("sideset", 33)
```

Parameters

entity_type "block", "sideset", nodeset"
entity_id Id of the entity in question

Return

Description of the entity or "" if none

[float] get_all_exodus_times(filename)

Open an exodus file and get a vector of all stored time stamps.

Parameters

filename Fully qualified exodus file name

Return

List of time stamps in the exodus file

---

int get_block_id(entity_type, entity_id)

Get the associated block id for a specific curve, surface, or volume.

Example

```
block_id = cubit.get_block_id("surface", 33)
```

Parameters

entity_type Type of entity
entity_id Id of entity in question

Return

Block id associated with this entity or zero (0) if none

[int] get_block_ids(mesh_geom_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

Return

List of block ids in the mesh geometry file

[int] get_block_id_list()

Get a list of all blocks.

Return

List of all active block ids

[int] get_nodeset_id_list()

Get a list of all nodesets.

Return

List of all active nodeset ids

[int] get_sideset_id_list()

Get a list of all sidesets.

Return

List of all active sideset ids

[int] get_bc_id_list(bc_type_in)

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

Return

List of all active bc ids

---

str get_bc_name(bc_type_in, 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.

Return

The bc name

[int] get_nodeset_id_list_for_bc(bc_type_in, 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

Return

A list of nodeset ID's associated with that bc

[int] get_sideset_id_list_for_bc(bc_type_in, 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

Return

A list of sideset ID's associated with that bc

---

int get_next_sideset_id()

Get a next available sideset id.

Return

Next available sideset id

---

int get_next_nodeset_id()

Get a next available nodeset id.

Return

Next available nodeset id

---

int get_next_block_id()

Get a next available block id.

Return

Next available block id

[int] get_block_volumes(block_id)

Get a list of volume ids associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of volume ids contained in the block

[int] get_block_surfaces(block_id)

Get a list of surface associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of surface ids contained in the block

[int] get_block_curves(block_id)

Get a list of curve associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of curve ids contained in the block

[int] get_block_vertices(block_id)

Get a list of vertices associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of vertex ids contained in the block

[int] get_block_nodes(block_id)

Get a list of nodes associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of node ids contained in the block

[int] get_block_edges(block_id)

Get a list of edges associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of edge ids contained in the block

[int] get_block_tris(block_id)

Get a list of tris associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of tri ids contained in the block

[int] get_block_faces(block_id)

Get a list of faces associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of face ids contained in the block

[int] get_block_pyramids(block_id)

Get a list of pyramids associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of pyramid ids contained in the block

[int] get_block_tets(block_id)

Get a list of tets associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of tet ids contained in the block

[int] get_block_hexes(block_id)

Get a list of hexes associated with a specific block.

Parameters

block_id User specified id of the desired block

Return

A list of hex ids contained in the block

[int] get_volume_hexes(volume_id)

get the list of any hex elements in a given volume

Parameters

volume_id User specified id of the desired volume

Return

A list of the hex ids in the volume

[int] get_volume_tets(volume_id)

get the list of any tet elements in a given volume

Parameters

volume_id User specified id of the desired volume

Return

A list of the tet ids in the volume

[int] get_nodeset_volumes(nodeset_id)

Get a list of volume ids associated with a specific nodeset.

Parameters

nodeset_id User specified id of the desired nodeset

Return

A list of volume ids contained in the nodeset

[int] get_nodeset_surfaces(nodeset_id)

Get a list of surface ids associated with a specific nodeset.

Parameters

nodeset_id User specified id of the desired nodeset

Return

A list of surface ids contained in the nodeset

[int] get_nodeset_curves(nodeset_id)

Get a list of curve ids associated with a specific nodeset.

Parameters

nodeset_id User specified id of the desired nodeset

Return

A list of curve ids contained in the nodeset

[int] get_nodeset_vertices(nodeset_id)

Get a list of vertex ids associated with a specific nodeset.

Parameters

nodeset_id User specified id of the desired nodeset

Return

A list of vertex ids contained in the nodeset

[int] get_nodeset_nodes(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

Return

A list of node ids contained in the nodeset

[int] get_nodeset_nodes_inclusive(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

Return

A list of node ids contained in the nodeset

[int] get_sideset_curves(sideset_id)

Get a list of curve ids associated with a specific sideset.

Parameters

sideset_id User specified id of the desired sideset

Return

A list of curve ids contained in the sideset

[int] get_curve_edges(curve_id)

get the list of any edge elements on a given curve

Parameters

curve_id User specified id of the desired curve

Return

A list of the edge element ids on the curve

[int] get_sideset_surfaces(sideset_id)

Get a list of any surfaces in a sideset.

Parameters

sideset_id User specified id of the desired sideset

Return

A list of the surfaces defining the sideset

[int] get_sideset_quads(sideset_id)

Get a list of any quads in a sideset.

Parameters

sideset_id User specified id of the desired sideset

Return

A list of the quads in the sideset

[int] get_surface_quads(surface_id)

get the list of any quad elements on a given surface

Parameters

surface_id User specified id of the desired surface

Return

A list of the quad ids on the surface

[int] get_surface_tris(surface_id)

get the list of any tri elements on a given surface

Parameters

surface_id User specified id of the desired surface

Return

A list of the tri ids on the surface

str get_entity_sense(source_type, source_id, sideset_id)

Get the sense of a sideset item.

Example

```
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

Return

Sense of the source_type/source_id in specified sideset

str get_wrt_entity(source_type, source_id, sideset_id)

Get the with-respect-to entity.

Example

```
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

Return

'with-respect-to' entity of the source_type/source_id in specified sideset

Bool is_using_shells(sideset_id)

Get the shell use for a sideset.

Parameters

sideset_id ID of the sideset

Return

True if the sideset uses shells, otherwise false

[str] get_geometric_owner(mesh_entity_type, mesh_entity_list)

Get a list of geometric owners given a list of mesh entities.

Example

```
owner_list = cubit.get_geometric_owner("quad", id_list)
```

Parameters

mesh_entity_type The type of mesh entity. Only works for 'quad, 'face', or 'tri'
mesh_entity_list A string containing space delimited ids, Cubit command form (i.e. 'all', '1 to 8', '1 2 3', etc)

Return

A list of geometry owners in the form of 'surface x', 'curve y', etc.

[int] get_volume_nodes(vol_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

Return

vector of IDs of nodes owned by the volume

[int] get_surface_nodes(surf_id)

Get list of node ids owned by a surface. Excludes nodes owned by bounding curves and verts.

Parameters

surf_id id of surface

Return

vector of IDs of nodes owned by the surface

[int] get_curve_nodes(curv_id)

Get list of node ids owned by a curve. Excludes nodes owned by bounding vertices.

Parameters

curv_id id of curve

Return

vector of IDs of nodes owned by the curve

---

int get_vertex_node(vert_id)

Get the node owned by a vertex.

Parameters

vert_id id of vertex

Return

ID of node owned by the vertex. returns -1 of doesn't exist

---

int get_id_from_name(name)

Get id for a named entity.

Example

```
entity_id = cubit.get_id_from_name("member_2")
```

Parameters

name Name of the entity to examine return Integer representing the entity

---

[int] get_group_groups(group_id)

Get group groups (groups that are children of another group).

Parameters

group_id ID of the group to examine return List of group ids contained in the specified group

---

[int] get_group_bodies(group_id)

Get group bodies (bodies that are children of a group).

Parameters

group_id ID of the group to examine return List of body ids contained in the specified group

---

[int] get_group_volumes(group_id)

Get group volumes (volumes that are children of a group).

Parameters

group_id ID of the group to examine return List of volume ids contained in the specified group

---

[int] get_group_surfaces(group_id)

Get group surfaces (surfaces that are children of a group).

Parameters

group_id ID of the group to examine return List of surface ids contained in the specified group

[int] get_group_curves(group_id)

Get group curves (curves that are children of a group).

Parameters

group_id ID of the group to examine return List of curve ids contained in the specified group

[int] get_group_vertices(group_id)

Get group vertices (vertices that are children of a group).

Parameters

group_id ID of the group to examine return List of vertex ids contained in the specified group

[int] get_group_nodes(group_id)

Get group nodes (nodes that are children of a group).

Parameters

group_id ID of the group to examine return List of node ids contained in the specified group

[int] get_group_edges(group_id)

Get group edges (edges that are children of a group).

Parameters

group_id ID of the group to examine return List of edge ids contained in the specified group

[int] get_group_quads(group_id)

Get group quads (quads that are children of a group).

Parameters

group_id ID of the group to examine return List of quad ids contained in the specified group

[int] get_group_tris(group_id)

Get group tris (tris that are children of a group).

Parameters

group_id ID of the group to examine return List of tri ids contained in the specified group

[int] get_group_tets(group_id)

Get group tets (tets that are children of a group).

Parameters

group_id ID of the group to examine return List of tet ids contained in the specified group

[int] get_group_hexes(group_id)

Get group hexes (hexes that are children of a group).

Parameters

group_id ID of the group to examine return List of hex ids contained in the specified group

int get_next_group_id()

Get the next available group id from Cubit.

delete_all_groups()

Delete all groups.

delete_group(group_id)

Delete a specific group.

Parameters

group_id ID of group to delete

set_max_group_id(max_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.

Parameters

max_group_id

int create_new_group()

Create a new group.

Return

group_id ID of new group

remove_entity_from_group(group_id, entity_id, entity_type)

Remove a specific entity from a specific group.

Example

```
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

add_entity_to_group(group_id, entity_id, entity_type)

Add a specific entity to a specific group.

Example

```
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

[int] get_mesh_group_parent_ids(element_type, element_id)

Get the group ids which are parents to the indicated mesh element.

Example

```
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 of group ids that contain this mesh element

Bool is_mesh_element_in_group(element_type, element_id)

Indicates whether a mesh element is in a group.

Example

```
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

---

Bool is_part_of_list(target_id, 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

Return

True if target_id is member of id_list, otherwise false

---

int get_last_id(entity_type)

Get the id of the last created entity of the given type.

Example

```
last_id = cubit.get_last_id("surface")
```

Parameters

entity_type Type of the entity being queried

Return

Integer id of last created entity

---

str get_assembly_classification_level()

Get Classification Level for metadata.

Return

Requested data

---

str get_assembly_classification_category()

Get Classification Category for metadata.

Return

Requested data

---

str get_assembly_weapons_category()

Get Weapons Category for metadata.

Return

Requested data

---

str get_assembly_metadata(volume_id, 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

Return

Requested data

Bool is_assembly_metadata_attached(volume_id)

Determine whether metadata is attached to a specified volume.

Parameters

volume_id ID of the volume

Return

True if metadata exists, otherwise false

---

str get_assembly_name(assembly_id)

Get the stored name of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

Name of the assembly node

---

str get_assembly_path(assembly_id)

Get the stored path of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

Path of the assembly node

str get_assembly_description(assembly_id)

Get the stored description of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

Description of the assembly node

int get_assembly_instance(assembly_id)

Get the stored instance number of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

Instance of the assembly node

str get_assembly_file_format(assembly_id)

Get the stored file format of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

File Format of the assembly node

str get_assembly_units(assembly_id)

Get the stored units measure of an assembly node.

Parameters

assembly_id Id that identifies the assembly node

Return

Units of the assembly node

str get_assembly_material_description(assembly_id)

Get the stored material description of an assembly part.

Parameters

assembly_id Id that identifies the assembly node

Return

Material Description of the assembly part

---

str get_assembly_material_specification(assembly_id)

Get the stored material specification of an assembly part.

Parameters

assembly_id Id that identifies the assembly node

Return

Material Specification of the assembly part

---

int get_exodus_id(entity_type, entity_id)

Get the exodus/genesis id for this element.

Example

```
exodus_id = cubit.get_exodus_id("hex", 221)
```

Parameters

entity_type The mesh element type
entity_id The mesh element id

Return

Exodus id of the element if element has been written out, otherwise 0

---

str get_geometry_owner(entity_type, entity_id)

Get the geometric owner of this mesh element.

Example

```
geom_owner = cubit.get_geometry_owner("hex", 221)
```

Parameters

entity_type The mesh element type
entity_id The mesh element id

Return

Name of owner

[int] get_connectivity(entity_type, entity_id)

Get the list of node ids contained within a mesh entity.

Example

```
node_id_list = cubit.get_connectivity("hex", 221)
```

Parameters

entity_type The mesh element type
entity_id The mesh element id

Return

List of node ids

[int] get_expanded_connectivity(entity_type, entity_id)

Get the list of node ids contained within a mesh entity, including interior nodes.

Example

```
node_id_list = cubit.get__expanded_connectivity("hex",
221)
```

Parameters

entity_type The mesh element type
entity_id The mesh element id

Return

List of all node ids associated with the element, including interior nodes

---

[int] get_sub_elements(entity_type, entity_id, 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.

Example

```
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

Return

List of ids of the desired dimension

[float] get_nodal_coordinates(node_id)

Get the nodal coordinates for a given node id.

Parameters

node_id The node id

Return

a triple containing the x, y, and z coordinates

Bool get_node_position_fixed(node_id)

Query "fixedness" state of node. A fixed node is not affecting by smoothing.

Parameters

node_id The node id

Return

True if constrained, otherwise false

---

str get_sideset_element_type(sideset_id)

Get the element type of a sideset.

Parameters

sideset_id The id of the sideset to be queried

Return

Element type

---

str get_block_element_type(block_id)

Get the element type of a block.

Parameters

block_id The block id

Return

Element type

---

int get_exodus_element_count(entity_id, entity_type)

Get the number of elements in a exodus entity.

Example

```
element_count = cubit.get_exodus_element_count(2,
"sideset")
```

Parameters

entity_id The id of the entity
entity_type The type of the entity

Return

Number of Elements

---

int get_block_attribute_count(block_id)

Get the number of attributes in a block.

Parameters

block_id The block id

Return

Number of attributes in the block

---

float get_block_attribute_value(block_id, index)

Get a specific block attribute value.

Parameters

block_id The block id
index The index of the attribute

Return

List of attributes

[str] get_valid_block_element_types(block_id)

Get a list of potential element types for a block.

Parameters

block_id The block id

Return

List of potential element types

---

int get_nodeset_node_count(nodeset_id)

Get the number of nodes in a nodeset.

Parameters

nodeset_id The nodeset id

Return

Number of nodes in the nodeset

---

int get_geometry_node_count(entity_type, entity_id)

Parameters

entity_type
entity_id

---

str get_mesh_element_type(entity_type, entity_id)

Get the mesh element type contained in the specified geometry.

Example

```
element_type = cubit.get_mesh_element_type("surface", 2)
```

Parameters

entity_type The type of entity
entity_id The id of the entity

Return

Mesh element type for that entity

Bool is_on_thin_shell(bc_type_in, entity_id)

Determine whether a BC is on a thin shell. Valid for temperature, convection and heatflux.

Parameters

bc_type_in
entity_id

Bool temperature_is_on_solid(bc_type_in, entity_id)

Determine whether a BC temperature is on a solid. Valid for convection and temperature.

Parameters

bc_type_in
entity_id

Bool convection_is_on_solid(entity_id)

Determine whether a BC convection is on a solid. Valid for convection.

Parameters

entity_id

Bool convection_is_on_shell_area(entity_id, shell_area)

Determine whether a BC convection is on a shell top or bottom. Valid for convection.

Parameters

entity_id
shell_area

float get_convection_coefficient(entity_id, cc_type)

Get the convection coefficient.

Parameters

entity_id
cc_type

float get_bc_temperature(bc_type, entity_id, temp_type)

Get the temperature. Valid for convection, temperature.

Parameters

bc_type
entity_id
temp_type

Bool temperature_is_on_shell_area(bc_type, bc_area, 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
bc_area
entity_id

Bool heatflux_is_on_shell_area(bc_area, entity_id)

Determine whether a BC heatflux is on a shell area.

Parameters

bc_area
entity_id

float get_heatflux_on_area(bc_area, entity_id)

Get the heatflux on a specified area.

Parameters

bc_area
entity_id

---

int get_cfd_type(entity_id)

Get the cfd subtype for a specified cfd BC.

Parameters

entity_id ID of the cfd BC

Return

Integer corresponding to the type of cfd, as defined by CI_BCTypes

---

float get_contact_pair_friction_value(entity_id)

Get the contact pair's friction value.

Parameters

entity_id

---

float get_contact_pair_tolerance_value(entity_id)

Get the contact pair's tolerance value.

Parameters

entity_id

Bool get_contact_pair_tied_state(entity_id)

Get the contact pair's tied state.

Parameters

entity_id

Bool get_contact_pair_general_state(entity_id)

Get the contact pair's general state.

Parameters

entity_id

Bool get_contact_pair_exterior_state(entity_id)

Get the contact pair's exterior state.

Parameters

entity_id

---

int get_displacement_coord_system(entity_id)

Get the displacement's coordinate system id.

Parameters

entity_id

---

str get_displacement_combine_type(entity_id)

Get the displacement's combine type which is "Overwrite", "Average", "SmallestCombine", or "LargestCombine".

Parameters

entity_id

---

float get_pressure_value(entity_id)

Get the pressure value.

Parameters

entity_id

---

str get_pressure_function(entity_id)

Get the pressure function.

Parameters

entity_id

---

float get_force_magnitude(entity_id)

Get the force magnitude from a force.

Parameters

entity_id

---

float get_moment_magnitude(entity_id)

Get the moment magnitude from a force.

Parameters

entity_id

[float] get_force_direction_vector(entity_id)

Get the direction vector from a force.

Parameters

entity_id

[float] get_force_moment_vector(entity_id)

Get the moment vector from a force.

Parameters

entity_id

---

str get_constraint_type(constraint_id)

Get the type of a specified constraint.

Parameters

constraint_id ID of the constraint

Return

A std::string indicating the type -- Kinematic, Distributing, Rigidbody

---

str get_constraint_reference_point(constraint_id)

Get the reference point of a specified constraint.

Parameters

constraint_id ID of the constraint

Return

A std::string indicating the reference point

---

str get_constraint_dependent_entity_point(constraint_id)

Get the dependent entity of a specified constraint.

Parameters

constraint_id ID of the constraint

Return

A std::string indicating the dependent entity

---

float get_material_property(mp, entity_id)

Parameters

mp
entity_id

---

int get_media_property(entity_id)

Parameters

entity_id

[str] get_material_name_list()

[str] get_media_name_list()

---

set_label_type(entity_type, label_flag)

Parameters

entity_type
label_flag

---

int get_label_type(entity_type)

Parameters

entity_type

Body body(id_in)

Gets the body object from an ID.

Parameters

id_in The ID of the body

Return

The body object

Volume volume(id_in)

Gets the volume object from an ID.

Parameters

id_in The ID of the volume

Return

The volume object

Surface surface(id_in)

Gets the surface object from an ID.

Parameters

id_in The ID of the surface

Return

The surface object

Curve curve(id_in)

Gets the curve object from an ID.

Parameters

id_in The ID of the curve

Return

The curve object

Vertex vertex(id_in)

Gets the vertex object from an ID.

Parameters

id_in The ID of the vertex

Return

The vertex object

---

reset()

Executes a reset within cubit.

Body brick(width, depth, height)

Creates a brick of specified width, depth, and height.

Parameters

width The width of the brick being created
depth The depth of the brick being created
height The height of the brick being created

Return

A Bodyobject of the newly created brick

Body sphere(radius, x_cut, y_cut, z_cut, inner_radius)

Creates all or part of a sphere.

Parameters

radius The radius of the sphere
x_cut If 1, cuts sphere by yz plane (default to 0)
y_cut If 1, cuts sphere by xz plane (default to 0)
z_cut If 1, cuts sphere by xy plane (default to 0)
inner_radius The inside radius if the sphere is hollow (default to 0)

Return

A Bodyobject of the newly created sphere

Body prism(height, sides, major, minor)

Creates a prism of the specified dimensions.

Parameters

height The height of the prism
sides The number of sides of the prism
major The major radius
minor The minor radius

Return

A Bodyobject of the newly created prism

Body pyramid(height, sides, major, minor, top)

Creates a pyramid of the specified dimensions.

Parameters

height The height of the pyramid
sides The number of sides of the pyramid
major The major radius
minor The minor radius
top determines size for the top of the pyramid. Defaults to 0, meaning it will go to a point

Return

A Bodyobject of the newly created pyramid

Body cylinder(hi, r1, r2, r3)

creates a cylinder of the specified dimensions

Parameters

hi the height of the cylinder
r1 radius in the x direction
r2 radius in the y direction
r3 used to adjust the top. If set to 0, will produce a cone. Set to r1/r2 to get a straight cylinder

Return

A body object of the newly created cylinder

Body torus(r1, r2)

creates a torus of the specified dimensions

Parameters

r1 radius from center to center of circle to be swept (r1>r2)
r2 radius of circle swept to create torus (r1>r2)

Return

A Bodyobject of the newly created torus

Vertex create_vertex(x, y, z)

Creates a vertex at a x,y,z.

Parameters

x The x location of the vertex (default to 0)
y The y location of the vertex (default to 0)
z The z location of the vertex (default to 0)

Return

A Vertexobject

Curve create_curve(v0, v1)

Creates a curve between two vertices.

Parameters

v0 The start vertex
v1 The end vertex

Return

A Curveobject

Body create_surface(curves)

Creates a surface from boundary curves.

Parameters

curves A list of curve objects from which to make the surface

Return

A Bodyobject of the newly created Surface

[Body] sweep_curve(curves, along_curves, draft_angle, draft_type, rigid)

Create a Body or a set of Bodies from a swept curve.

Parameters

curves A list of curves to sweep
along_curves A list of curves to sweep along
draft_angle The sweep draft angle (default to 0)
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) ***
rigid The inside radius if the sphere is hollow (default to False)

Return

A List of newly created Bodies

Body copy_body(init_body)

Creates a copy of the input Body.

Parameters

init_body The Bodyto be copied

Return

A Bodyidentical to the input Body

[Body] tweak_surface_offset(surfaces, 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

Return

A list of the body objects of the modified bodies

[Body] tweak_surface_remove(surfaces, extend_ajoining, keep_old, preview)

Removes a surface from a body and extends the surrounding surfaces if extend_ajoining is true.

Parameters

surfaces The surfaces to be removed
extend_ajoining Extend the ajoining surfaces (default to true)
keep_old Keep the old body (default to false)

preview Flag to show the preview or not (default to false)

Return

A list of changed body objects

[Body] tweak_curve_remove(curves, keep_old, preview)

Removes a curve from a body and extends the surrounding surface to fill the gap.

Parameters

curves
keep_old Keep the old body (defaults to false)
preview Flag to show the preview (defaults to false)

Return

A list of changed body objects

[Body] tweak_curve_offset(curves, distances, keep_old, preview)

Performs a tweak curve offset command.

Parameters

curves A list of curve objects to offset
distances A list of distances associated with the offset for each curve
keep_old Keep the old body (defaults to false)
preview Flag to show the preview (defaults to false)

Return

A list of changed body objects

[Body] tweak_vertex_fillet(verts, r0, keep_old, preview)

Performs a tweak vertex fillet command.

Parameters

verts A list of vertex objects to fillet
r0 radius of the fillet
keep_old Keep the old body (defaults to false)
preview Flag to show the preview (defaults to false)

Return

A list of changed body objects

[Body] subtract(tool_in, from_in, imprint_in, keep_old_in)

Performs a boolean subtract operation.

Parameters

tool_in List of Bodyobjects to subtract
from_in List of Bodyobjects to be subtracted from

imprint_in Flag to set the imprint (defaults to false)
keep_old_in Flag to keep the old volume (defaults to false)

Return

A list of changed body objects

[Body] unite(body_in, keep_old_in)

Performs a boolean unite operation.

Parameters

body_in A list of body objects to unite
keep_old_in Flag to keep old bodies (defaults to false)

Return

A list of changed bodies

---

move(entity, vector, preview)

Moves the Entity the specified vector.

Parameters

entity The Entityto be moved
vector The vector the Entitywill be moved
preview Flag to show the preview or not, default is false

---

scale(entity, factor, preview)

Scales the Entity according to the specified factor.

Parameters

entity The Entityto be scaled
factor The scale factor
preview Flag to show the preview or not, default is false

---

reflect(entity, axis, preview)

Reflect the Entity about the specified axis.

Parameters

entity The Entityto be reflected
axis The axis to be reflected about
preview Flag to show the preview or not, default is false

[int] get_volumes_for_node(node_name, node_instance)

Parameters

node_name
node_instance

int get_mesh_error_count()

PyObservable

The base class of everything in the CubitInterface.

The PyObservableclass allows a user to be able to 'observe' any entity in the CubitInterface. Thus, a user would be able to handle events within Cubit appropriately.

Example

```
import cubit
class TestObserver(cubit.PyObserver):
  def notify_observers(self, obsvd, evt ):
    if evt == 2:
      print 'Entity destroyed!'
    elif evt == 11 or evt == 12 or evt == 13:
      print 'Volume changed!'
    else:
      print 'Unknown event! '

testobs = TestObserver()
br = cubit.brick(1,1,1)
testobs.register_observable(br)
cubit.scale(br,2)
cubit.cmd('delete body 1')
```

Inheritance

PyObservable
Entity
GeomEntity
Body | Curve | Surface | Vertex | Volume

Class Member Functions

notify_observers	Notify the observer of a event.

Member Function Documentation

notify_observers(event_type)

Notify the observer of a event.

Example

```
import cubit
class TestObserver(cubit.PyObserver):
  def notify_observers(self, obsvd, evt ):
    if evt == 2:
      print 'Entity destroyed!'
    elif evt == 11 or evt == 12 or evt == 13:
      print 'Volume changed!'
    else:
      print 'Unknown event! '
```

Parameters

event_type The type of event

PyObserver

A base class to be extended to perform custom actions on Cubit events.

Class Member Functions

register_observable	Register a PyObservable to be watched by this PyObserver.
unregister_observable	Unregister a PyObservable to be watched by this PyObserver.
notify_observers	The function called when an event happens.

Member Function Documentation

---

register_observable(observable)

Register a PyObservable to be watched by this PyObserver.

Parameters

observable The PyObservableto be observed by this PyObserver

---

unregister_observable(observable)

Unregister a PyObservable to be watched by this PyObserver.

Parameters

observable The PyObservableto stop being observed by this PyObserver

---

notify_observers(observable, event_type)

The function called when an event happens.

Parameters

observable The PyObservableon/to which the event is occuring
event_type An integer representing a specific event type

Entity

The base class of all the geometry and mesh types.

Inheritance

PyObservable
Entity
GeomEntity
Body | Curve | Surface | Vertex | Volume

Class Member Functions

	destroy_cubit_entity	
[float]	bounding_box	Get the bounding box of the Entity.
[float]	center_point	Get the center point of the Entity.
int	id	Get the id of the Entity.
	is_visible	Set the visibility state of the Entity.
int	is_visible	Get the visibility state of the Entity.
	is_transparent	Set the tranparency state of the Entity.
int	is_transparent	Get the tranparency state of the Entity.

Member Function Documentation

---

destroy_cubit_entity()

[float] bounding_box()

Get the bounding box of the Entity.

Example

```
b_box = entity.bounding_box()
```

Return

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

---

[float] center_point()

Get the center point of the Entity.

Example

```
center = entity.center_point()
```

Return

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

---

int id()

Get the id of the Entity.

Example

```
id = entity.id()
```

Return

The id of the Entity

---

is_visible(visibility_flag)

Set the visibility state of the Entity.

Example

```
entity.is_visible(1)
```

Parameters

visibility_flag The flag that sets whether the Entityis visible (1) or not (0)

---

int is_visible()

Get the visibility state of the Entity.

Example

```
vis = entity.is_visible()
```

Return

The current visiblity state of the Entity(1 if visible, 0 if not)

---

is_transparent(transparency_flag)

Set the tranparency state of the Entity.

Example

```
entity.is_transparent(1)
```

Parameters

transparency_flag The flag that sets whether the Entityis transparent (1) or not (0)

---

int is_transparent()

Get the tranparency state of the Entity.

Example

```
trans = entity.is_transparent()
```

Return

The current transparency state of the Entity(1 if transparent, 0 if not)

GeomEntity

The base class for specifically the Geometry types (Body, Surface, etc.).

Inheritance

PyObservable
Entity
GeomEntity
Body | Curve | Surface | Vertex | Volume

Class Member Functions

	mesh	Mesh the GeomEntity.
Bool	is_meshed	Return the current mesh state of the GeomEntity.
	smooth	Smooths the mesh on the GeomEntity.
	remove_mesh	Removes the mesh on the GeomEntity.
str	entity_name	Return the first name of the GeomEntity.
	entity_name	Assign a name to the GeomEntity.
[str]	entity_names	Return the all the names of the GeomEntity.
int	num_names	Return the number of names for the GeomEntity.
	remove_entity_name	Remove a specific name from the list of names assigned to the GeomEntity.
	remove_entity_names	Remove all the names assigned to the GeomEntity.
int	dimension	Get the dimensions of the GeomEntity.
[Body]	bodies	Get the bodies in the GeomEntity.
[Volume]	volumes	Get the volumes in the GeomEntity.
[Surface]	surfaces	Get the surfaces in the GeomEntity.
[Curve]	curves	Get the curves in the GeomEntity.
[Vertex]	vertices	Get the vertices in the GeomEntity.

Member Function Documentation

---

mesh()

Mesh the GeomEntity.

Example

```
geomEntity.mesh()
```

Bool is_meshed()

Return the current mesh state of the GeomEntity.

Example

```
mesh = geomEntity.is_meshed()
```

Return

Whether the GeomEntityis meshed or not

---

smooth()

Smooths the mesh on the GeomEntity.

Example

```
geomEntity.smooth()
```

---

remove_mesh()

Removes the mesh on the GeomEntity.

Example

```
geomEntity.remove_mesh()
```

---

str entity_name()

Return the first name of the GeomEntity.

Example

```
name = geomEntity.entity_name()
```

Return

The first name of the GeomEntity

---

entity_name(name)

Assign a name to the GeomEntity.

Example

```
geomEntity.entity_name("Brick1")
```

Parameters

name The name to be assigned to the GeomEntity

[str] entity_names()

Return the all the names of the GeomEntity.

Example

```
names = geomEntity.entity_names()
```

Return

A vector (or list) of all the names of the GeomEntity

---

int num_names()

Return the number of names for the GeomEntity.

Example

```
num = geomEntity.num_names()
```

Return

The number of names for the GeomEntity

---

remove_entity_name(name)

Remove a specific name from the list of names assigned to the GeomEntity.

Example

```
geomEntity.remove_entity_name("Brick1")
```

Parameters

name The name to be removed from the list of names assigned to the GeomEntity

---

remove_entity_names()

Remove all the names assigned to the GeomEntity.

Example

```
geomEntity.remove_entity_names()
```

---

int dimension()

Get the dimensions of the GeomEntity.

Example

```
dim = geomEntity.dimension()
```

Return

The dimension of the GeomEntity

[Body] bodies()

Get the bodies in the GeomEntity.

Example

```
bodies = geomEntity.bodies()
```

Return

A vector (or list) of bodies contained within the GeomEntity

[Volume] volumes()

Get the volumes in the GeomEntity.

Example

```
volumes = geomEntity.volumes()
```

Return

A vector (or list) of volumes contained within the GeomEntity

---

[Surface] surfaces()

Get the surfaces in the GeomEntity.

Example

```
surfaces = geomEntity.surfaces()
```

Return

A vector (or list) of surfaces contained within the GeomEntity

---

[Curve] curves()

Get the curves in the GeomEntity.

Example

```
curves = geomEntity.curves()
```

Return

A vector (or list) of curves contained within the GeomEntity

[Vertex] vertices()

Get the vertices in the GeomEntity.

Example

```
vertices = geomEntity.vertices()
```

Return

A vector (or list) of vertices contained within the GeomEntity

---

Body

Defines a body object that mostly parallels Cubit's Body class.

Inheritance

PyObservable
Entity
GeomEntity
Body

Class Member Functions

[float]	get_mass_props	Get the mass properties of the Body, specifically the center of gravity.
int	point_containment	Get whether a point is in, on, or outside the Body.
float	volume	Get the volume of the Body.
Bool	is_sheet_body	Get whether the Body is a sheet body or not.

Member Function Documentation

[float] get_mass_props()

Get the mass properties of the Body, specifically the center of gravity.

Example

```
props = body.get_mass_props()
```

Return

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

---

int point_containment(loc_in)

Get whether a point is in, on, or outside the Body.

Example

```
on_out_in = body.point_containment([0,0,0])
```

Parameters

loc_in

Return

Whether a point is unknown (-1), outside (0), in (1), or on (2) the Body

---

float volume()

Get the volume of the Body.

Example

```
vol = body.volume()
```

Return

The volume of the Body

Bool is_sheet_body()

Get whether the Body is a sheet body or not.

Example

```
is_sheet = body.is_sheet_body()
```

Return

Whether the Bodyis a sheet body or not

Surface

Defines a surface object that mostly parallels Cubit's RefFace class.

Inheritance

PyObservable
Entity
GeomEntity
Surface

Class Member Functions

	color	Set the color of the surface.
int	color	Get the color of the surface.
[[Curve]]	ordered_loops	Get the ordered loops of the Surface.
[float]	normal_at	Get the normal at a particular point on the Surface.
[float]	closest_point_trimmed	Get the nearest point on the Surface to point specified.
[float]	closest_point_trimmed	Get the nearest point on the Surface to point specified.
int	point_containment	Get whether a point is on or off of the Surface.
[float]	principal_curvatures	Get the principal curvatures of the Surface.
[float]	position_from_u_v	Get the Cartesian coordinates from the uv coordinates on the Surface.
[float]	u_v_from_position	Get the uv coordinates from the supplied Cartesian coordinates on the Surface.
[float]	get_param_range_U	Get range of u for the Surface.
[float]	get_param_range_V	Get range of v for the Surface.
float	area	Get area of the Surface.
Bool	is_planar	Get whether the Surface is planar or not.
Bool	is_cylindrical	Get whether the Surface is cylindrical or not.

Member Function Documentation

---

color(value)

Set the color of the surface.

Example

```
surface.color(0)
```

Parameters

value The color value that the surface will have

---

int color()

Get the color of the surface.

Example

```
col = surface.color()
```

Return

The color value associated with the surface's current color

[[Curve]] ordered_loops()

Get the ordered loops of the Surface.

Example

```
loops = surface.ordered_loops()
```

Return

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...

[float] normal_at(location)

Get the normal at a particular point on the Surface.

Example

```
norm = surface.normal_at([0,0,0])
```

Parameters

location A vector containing three values that are the coordinates of a point

Return

A vector (or list) of doubles representing values of normal vector as follows: 0 - x value 1 - y value 2 - z value

---

[float] closest_point_trimmed(location)

Get the nearest point on the Surface to point specified.

Example

```
nearest = surface.closest_point_trimmed([0,0,0])
```

Parameters

location A vector containing three values that are the coordinates of a point

Return

A vector (or list) of doubles representing values of nearest point as follows: 0 - x coordinate 1 - y coordinate 2 - z coordinate

---

[float] closest_point_trimmed(location)

Get the nearest point on the Surface to point specified.

Example

```
nearest = surface.closest_point_trimmed([0,0,0])
```

Parameters

location A vector containing three values that are the coordinates of a point

Return

A vector (or list) of doubles representing values of nearest point as follows: 0 - x coordinate 1 - y coordinate 2 - z coordinate

---

int point_containment(point_in)

Get whether a point is on or off of the Surface.

Example

```
on_off = surface.point_containment([0,0,0])
```

Parameters

point_in A vector containing three values that are the coordinates of a point

Return

A python boolean representing whether the point is off (0) or on (1) the Surface

---

[float] principal_curvatures(point)

Get the principal curvatures of the Surface.

Example

```
curvatures = surface.principal_curvatures([0,0,0])
```

Parameters

point A vector containing three values that are the coordinates of a point

Return

A list of two floats representing the curvatures 0 - curvature 1 1 - curvature 2

[float] position_from_u_v(u, v)

Get the Cartesian coordinates from the uv coordinates on the Surface.

Example

```
pos = surface.position_from_u_v(0, 0)
```

Parameters

u The u parameter
v The v parameter

Return

The Cartesian coordinates of the supplied uv coordinates as a vector: 0 - x coordinate 1 - y coordinate 2 - z coordinate

---

[float] u_v_from_position(location)

Get the uv coordinates from the supplied Cartesian coordinates on the Surface.

Example

```
uv = surface.position_from_u_v([0,0,0])
```

Parameters

location A vector containing the Cartesian coordinates

Return

The curvature values: 0 - The u parameter 1 - The v parameter

---

[float] get_param_range_U()

Get range of u for the Surface.

Example

```
bounds = surface.get_param_range_U()
```

Return

The curvature values: 0 - The lowest value in the u direction 1 - The highest value in the u direction

---

[float] get_param_range_V()

Get range of v for the Surface.

Example

```
lower_bound, upper_bound = surface.get_param_range_V()
```

Return

The curvature values: 0 - The lowest value in the v direction 1 - The highest value in the v direction

float area()

Get area of the Surface.

Example

```
area = surface.area()
```

Return

The area of the Surface

Bool is_planar()

Get whether the Surface is planar or not.

Example

```
planar = surface.is_planar()
```

Return

Whether the Surfaceis planar or not

Bool is_cylindrical()

Get whether the Surface is cylindrical or not.

Example

```
cyl = surface.is_cylindrical()
```

Return

Whether the Surfaceis cylindrical or not

Curve

Defines a curve object that mostly parallels Cubit's RefEdge class.

Inheritance

PyObservable
Entity
GeomEntity
Curve

Class Member Functions

	color	Set the color of the Curve.
int	color	Get the color of the Curve.
[float]	tangent	Get the tangent to the Curve at a particular point.
[float]	curvature	Get the curvature of the Curve at a particular point.
[float]	closest_point	Get the curvature of the Curve at a particular point.
[float]	closest_point_trimmed	Get the curvature of the Curve at a particular point.
float	length	Get the length of the Curve.
[float]	curve_center	Get the center point of the Curve.
[float]	position_from_fraction	Get the position of the point a specified fraction along the Curve.
float	start_param	Get the lowest value of the Curve in uv space.
float	end_param	Get the highest value of the Curve in uv space.
float	u_from_position	Get the u value of a particular position on the Curve.
[float]	position_from_u	Get the position of a particular u value for the Curve.
float	u_from_arc_length	Get the u value for a point a specified arc length away from a specified root parameter on the Curve.
float	fraction_from_arc_length	Get the fraction along the Curve a specified arc length is away from a given Vertex.
[float]	point_from_arc_length	Get the position on a Curve that is a specified arc length away from the specified root parameter.
float	length_from_u	Get the length between two specified parameters on a Curve.
Bool	is_periodic	Get whether the Curve is periodic or not.

Member Function Documentation

---

color(value)

Set the color of the Curve.

Example

```
curve.color(0)
```

Parameters

value The color value that the curve will have

---

int color()

Get the color of the Curve.

Example

```
col = curve.color()
```

Return

The color value associated with the curve's current color

[float] tangent(point)

Get the tangent to the Curve at a particular point.

Example

```
tan = curve.tangent([0,0,0])
```

Parameters

point A vector containing 3 doubles representing coordinates of a location on the Curve

Return

The tangent to the Curveat the location specified

[float] curvature(point)

Get the curvature of the Curve at a particular point.

Example

```
curvature = curve.curvature([0,0,0])
```

Parameters

point A vector containing 3 doubles representing coordinates of a location on the Curve

Return

The curvature of the Curveat the location specified

[float] closest_point(point)

Get the curvature of the Curve at a particular point.

Example

```
close = curve.closest_point([0,0,0])
```

Parameters

point A vector containing 3 doubles representing coordinates of a location on the Curve

Return

The closest point to the Curvefrom the location specified

[float] closest_point_trimmed(point)

Get the curvature of the Curve at a particular point.

Example

```
close = curve.closest_point([0,0,0])
```

Parameters

point A vector containing 3 doubles representing coordinates of a location on the Curve

Return

The closest point to the Curvefrom the location specified

---

float length()

Get the length of the Curve.

Example

```
len = curve.length()
```

Return

The length of the Curve

[float] curve_center()

Get the center point of the Curve.

Example

```
center = curve.curve_center()
```

Return

A vector containing the coordinates of the Curve's center according to the following: 0 - x coordinate 1 - y coordinate 2 - z coordinate

---

[float] position_from_fraction(fraction_along_curve)

Get the position of the point a specified fraction along the Curve.

Example

```
pos = curve.position_from_fraction(0.5)
```

Parameters

fraction_along_curve A decimal value between 0 and 1 to determine a particular position along the Curve

Return

A vector containing the coordinates of the position a specified fraction along the Curve: 0 - x coordinate 1 - y coordinate 2 - z coordinate

---

float start_param()

Get the lowest value of the Curve in uv space.

Example

```
start = curve.start_param()
```

Return

The beginning value of the parameter

---

float end_param()

Get the highest value of the Curve in uv space.

Example

```
end = curve.end_param()
```

Return

The ending value of the parameter

---

float u_from_position(position)

Get the u value of a particular position on the Curve.

Example

```
u = curve.u_from_position([0,0,0])
```

Parameters

position A vector containing the coordinates of the input position

Return

The u value of the position along the Curve

[float] position_from_u(u_value)

Get the position of a particular u value for the Curve.

Example

```
position = curve.position_from_u(0.5)
```

Parameters

u_value The u value of the position along the Curve

Return

A vector containing the coordinates of the output position

---

float u_from_arc_length(root_param, arc_length)

Get the u value for a point a specified arc length away from a specified root parameter on the Curve.

Example

```
u = curve.u_from_arc_length(0, 0.5)
```

Parameters

root_param The beginning parameter from which the arc length is added to
arc_length The length away from the root parameter of the output parameter

Return

The u value of the Curvethe arc length away from the root parameter

---

float fraction_from_arc_length(root_vertex, length)

Get the fraction along the Curve a specified arc length is away from a given Vertex.

Example

```
fraction = curve.fraction_from_arc_length(vertex, 0.5)
```

Parameters

root_vertex The Vertexto start from (vertex object)
length The length along the Curveaway from the root Vertex

Return

The fraction of the Curvethat is the specified length away from the specified Vertex

---

[float] point_from_arc_length(root_param, arc_length)

Get the position on a Curve that is a specified arc length away from the specified root parameter.

Example

```
position = curve.point_from_arc_length(0, 0.5)
```

Parameters

root_param The root parameter from which the arc length is added to
arc_length The arc length along the Curveaway from the root parameter

Return

A vector that contains the coordinates of a position a specified arc length away from the root parameter

---

float length_from_u(parameter1, parameter2)

Get the length between two specified parameters on a Curve.

Example

```
length = curve.length_from_u(0, 0.5)
```

Parameters

parameter1 The beginning parameter
parameter2 The ending parameter

Return

The length between the two specified paramters along the Curve

---

Bool is_periodic()

Get whether the Curve is periodic or not.

Example

```
periodic = curve.is_periodic()
```

Return

Whether the Curveis periodic or not

Vertex

Defines a vertex object that mostly parallels Cubit's RefVertex class.

Inheritance

PyObservable
Entity
GeomEntity
Vertex

Class Member Functions

	color	Set the color of the Vertex.
int	color	Get the color of the Vertex.
[float]	coordinates	Get the Cartesian coordinates of the Vertex.

Member Function Documentation

---

color(value)

Set the color of the Vertex.

Example

```
vertex.color(0)
```

Parameters

value The color value that the vertex will have

---

int color()

Get the color of the Vertex.

Example

```
col = vertex.color()
```

Return

The color value associated with the vertex's current color

[float] coordinates()

Get the Cartesian coordinates of the Vertex.

Example

```
position = vertex.coordinates()
```

Return

A vector containing the coordinates of the Vertexwith indices corresponding to the coordinates as follows: 0 - x coordinate 1 - y coordinate 2 - z coordinate

Volume

Defines a volume object that mostly parallels Cubit's RefVolume class.

Inheritance

PyObservable
Entity
GeomEntity
Volume

Class Member Functions

float	volume	Get the volume of the Volume.
	color	Set the color of the Volume.
int	color	Get the color of the Volume.
[float]	principal_axes	Get the principal axes of the Volume.
[float]	principal_moments	Get the principal moments of the Volume.
[float]	centroid	Get the centroid of the Volume.

Member Function Documentation

---

float volume()

Get the volume of the Volume.

Example

```
vol = volume.volume()
```

Return

The volume of the Volume

---

color(value)

Set the color of the Volume.

Example

```
volume.color(0)
```

Parameters

value The color value that the volume will have

---

int color()

Get the color of the Volume.

Example

```
col = volume.color()
```

Return

The color value associated with the volume's current color

[float] principal_axes()

Get the principal axes of the Volume.

Example

```
axes = volume.principal_axes()
```

Return

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

---

[float] principal_moments()

Get the principal moments of the Volume.

Example

```
moments = volume.principal_moments()
```

Return

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

---

[float] centroid()

Get the centroid of the Volume.

Example

```
centroid = volume.centroid()
```

Return

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

---

CubitFailureException

An exception class to alert the caller when the underlying Cubit function fails.

Class Member Functions

str	what

Member Function Documentation

---

str what()

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.

Class Member Functions

str	what

Member Function Documentation

---

str what()

InvalidInputException

An exception class to alert the caller of a function that invalid inputs were entered.

Class Member Functions

str	what

Member Function Documentation

---

str what()

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.

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

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

Credits

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Manager

- Ted Blacker, Manager, Computational Simulation Infrastructure Department (Org. 1543), Sandia National Laboratories

Project Board

- **Principal Investigator**: Byron Hanks, Org. 1543
- **SNL Support Manager**: Robert Kerr, Org. 9513
- **SNL Product Manager**: Kristin Dion, Org. 1525

Research and Development

Computational Simulation Infrastructure Department, Org. 1543, Sandia National Laboratories, Albuquerque, NM

- Brett Clark
- Steven J. Owen
- Matthew L. Staten
- Roshan W. Quadros
- Clifton Dudley (BYU - Intern)

Elemental Technologies Inc., American Fork, UT

- Ray J. Meyers
- Corey Ernst
- Karl Merkley
- Randy Morris
- Corey McBride
- Mark Richardson
- Clinton Stimpson
- Ernie Perry
- James Perry

- Matthew Meyers

Caterpillar, Inc., Peoria, IL

- Sam Showman
- Alex Hays

Carnegie Mellon University, PA

- Kenji Shimada (PI)

- Jean Lu
- Karthik Srinivasan
- Inho Song

Testing Staff

- Daniel Merkley, ETI, UT
- Nathaniel Merkley, ETI, UT
- Kyle Merkley, csimsoft, UT
- Michael Meyers, ETI, UT
- Matthew Meyers, csimsoft, UT

Documentation

- Randy Morris, ETI, UT

Administrative Assistant

- Anne Gigante, Org. 1543, Sandia National Laboratories

Index

.

.cub 41

A

abaqus 761, 853
abort management 22
abortloop 919
absolute value 935
acceleration 929
accuracy 411
acis 225, 233, 515
adjust boundary 666
advancing front 555, 613
align 291
align mesh 717
ambient intensity 153
analyze geometry 73, 355
angle 151
  calculate 935
  mesh quality 645, 647
  perspective 151
  units 929
  vertex type 583
apply button 49
aprepro
  conditional statement 951
  file inclusion 951
  functions 935
  if statement 951
  journal file 111
  journaling 955
  loops 951
  operators 923
  predefined variables 927
  rules 919
  syntax 917
  units 929
  variable 23
arc 249, 251
arc span 531
area 215, 645, 647
aspect ratio 647, 651, 653
assembly 67, 503, 509, 513
associativity 751
attributes 493, 495, 497
  block 783
  metadata 509
Auto Bias 543
AutoCAD 225
autocenter 121
automatic forced sweepability 393
automatic geometry decomposition 861
automatic scheme selection 637
  general notes 638
  surfaces 640
  vertex types 637
  volumes 640
automatic size assignment 531
autosmooth 587
axis 131, 161, 201

B

background color 145
batch 23
beam 791
bend 387
bias 543, 665, 691
bitmap 159
blend surfaces 367
block 783
  attribute 783
  curve 783
  element type 783
  surface 783
  volume 783
body 267
  align 291
  auto heal 359
  copy 293
  cut 861
  healer analyze 355
  intersect 303
  list 215
  merge 423
  move 291, 295
  reflect 301
  rotate 291, 299
  scale 297
  section 349
  separate 351
  subtract 305
  unite 307
boolean
  intersect 303
  subtract 305
  unite 307
border 161
boundary condition sets 825
boundary conditions 4
  feature 4
brick 277
bug reports 19
buttons 106

C

CALCULATED 957
camera 151
cancel 43
cd command 37
ceiling 935
centroid area pull 673
cfd 839
cgm 159

chamfer 365, 367
changing preferences 103
chop 309
circle 545
cleanup 557, 723
clear 125
clip 137
closestpt 437, 871
coarsening 711
cohesive element 863
coincident nodes 667
collapse
  angle 451
  curve 455
  mesh edges 715
  surface 457
colors 145, 905
command 33
command echo 39
command line 33, 35
command syntax 33
command window 91
comment 115
composite
  curves 435
  surfaces 437
condition number 647, 651, 653, 681
conductivity 823
Cone Surface 317
contact set 825
contact surface 835, 837
control skew 665
convection 833
coordinates 935
copy 293
  body 293
  mesh 633
  scale 297
  scheme 633
corner 583
cosine 935
create
  brick 277
  curve 251
  cylinder 279
  frustum 283
  pyramid 285
  sheet 258
  sphere 287
  surface 257
  torus 289
  vertex 249
  volume 267
credits 1135
ctrl-c 43
cubit file 41
cubit file method 41
cubit_geom.save.g 41
cubit_geom.save.sat 41
CUBIT_OPT 31
cubit-dev 17
curvature 547
  sizing function 741
curve 251
  bias 543
  block 783
  copy mesh 633
  create 251
  extrude 257
  intervals 535
  list 215
  nodeset 797, 807
  partitioning 439
  plane normal to 317
  sideset 797, 807
  split 325
  tangent 465
  trim or extend 413
  type 583
  valence 65
  vertex on 249
customize 45, 103, 107
cut 861
  mesh 871
cylinder 279

D

DART 503, 509, 513
data filters 175
data structure 915
date 935
debug 23
decomposition
  automatic 861
  split periodic 327
defeaturing 859
  detail suppression 859
  surface removal 391
deformations 751
degrees 935
Delaunay 607
delete 523
density 823, 869, 929
detail suppression 859
development requests 19
diagonal ratio 653
dialog 49
  command 49
  options 103
  property editor 87
  tree view 67
digits 39
dimension 849

direction....197
displacement....827
display....125
display toolbar....99
distance....501
distortion....645, 647, 651, 653
distribution....9
distribution factor....803, 813
divergence....481
doubler....376
draw....131
  color table....145
  cylinder....315
  detail....859
  drawing, locating, and highlighting....131
  edge....167
  group....467
  histogram....655
  location....211
  nodeset....167, 797, 807
  normal....131
  picked....181
  plane....208
  vertex types....583
Drawing
  Locating
    and Highlighting Entities....131
dualbias....543, 665
Duplicate Block Elements....783
duplicating....633

E

echo....221
edge collapse....715
edge length....689, 935
elastic modulus....823
element block....783
element numbering....911
element types....4
enclosure....800, 810
end....583
energy....929
enhancement requests....19
entity
  curve....251
  drawing....131
  highlighting....131
  labels....141
  names....487
  picking....181
  selecting....59, 181
  selection mode....59
  specification....167
  surface....257
  tree....67
  vertex....249
  visibility....149
  volume....267
environment....173
  user settings....31
equal....549
equipotential....675
error count....935
error logging....39
errors....39, 935
execute....935
execute button....49
execution....23
  command syntax....23
exit....21
exodus file method....817
  element numbering....911
  file specification....817
  importing....751
  model title....819
  sizing function....749
exodus II....751, 849
exotxt....849
expand....167
export....857
extend....413
extraneous....389

F

facets....127, 521, 760
factor....691
farfield pressure....839
fastq....23, 1123
  importing....235
feature angle....243, 481, 757
feature size....869
features....3
field function....749
file....29
  acis....515
  exodus II....817, 819
  fastq....235
  iges....519
  initialization....29
  input....23
  journal....23, 111, 115
filename....33
  step....517
fillet....365, 367, 380
find surface overlap....403
fire ray....189
firmness....527
  interval....527
  scheme....637
  vertex type....583
Fixed_Imprints....587
flatquad....863
flatshade....127

floor 935
fluent 839, 855
flush 125
fly-in 79
force 393, 831, 929
free elements 751
free mesh 619, 751, 767, 771
from 121, 151
frustum 283
fullhex 915
fuzzy 637

G

gamma 935
geometric entities
  curve 251
  surface 257
  vertex 249
  volume 267
geometry
  analyzing 355
  debug 409
  merging 423
  validating 407
  visibility 149
geometry accuracy 951
geometry associativity 751, 767
geometry deletion 523
geometry groups 467
geometry representation 225
geometry tweaking 365, 367, 373
grafting 879
graphics 4
  camera 151
  clipping plane 137
  colors 145
  display 125
  draw 131
  hardcopy 159
  labels 141
  mesh slicing 135
  modes 127
  no graphics option 23
  options 161
  selection 181
  views 157
  window size 155
graphics clip 137
graphics lighting 153
group
  graphical selection 471
  operations 467
  picked 184
  propagated hex 473
  quality 485
groups 593
  sweep 593
  xor 467

H

hammer icon 13
hard interval 527
hardcopy 159
hardware platforms 7
healing
  analyzing geometry 355
  attributes 357
  automatic 359
  failure 363
heat flux 832
help 23, 35, 45
hiddenline 127
highlight 131, 161
  drawing, locating, and highlighting 131
histogram 655, 659
history command 37
hole 378, 551
htet 625
hypotenuse 935

I

id input field 49
id maps 849
idealize 373
i-deas 763
idless journal file 117
ids 491, 935

'

'ifdef' 919
'ifndef' 919

I

iges 239, 417, 519
IMMUTABLE block 917
Immutable Variables 917
import233, 235, 237, 239, 241, 247, 751, 761, 763, 765, 767
imprint 419
  mesh 447
improve 244
info 221
initialization file 23, 29
inlet massflow 839
inlet pressure 839
inlet velocity 839
input 857
input file 23
input window 91
inria 599
installation 9

interior mesh elements 137

interrupt 43, 45

intersect 303

interval
- *automatic specification* 531
- *explicit specification* 529
- *firmness* 527
- *matching* 535
- *periodic* 537
- *relative* 539

isoparameter 131

isoparametric 131

J

jacobian 647, 651, 653

journal file 23
- *APREPRO* 955
- *automatic creation* 115
- *creation and playback* 111
- *editor* 95

K

key icon 13

key press commands 61

L

label 141

laplacian smoothing 677

length 935

license 9, 31

light intensity 153

lighting model 153

limit plane 367, 373

line width 161

listing information
- *environment* 221
- *geometry* 215
- *mesh* 217
- *model summary* 213
- *special entities* 219
- *vertex types* 583

load set 825

loads 831

Local refinement 691

locate 79
- *drawing, locating, and highlighting* 131

Locate command 131

location 187

location on curve 193

loft 272

logarithm 935

logging 221

lowercase 935

ls command 37

M

mailing lists 17

make solid 267

mapping 553

mass 929

material 509, 795, 823

mathematical functions 935

mean ratio smoothing 683

measurement 501

memory usage 915

merge nodes 759

merge tolerance 419

merging 423
- *examining merged entities* 427
- *tolerance* 429
- *using to verify geometry* 433

mesh
- *and BC Entity Visibility* 149
- *collapse element* 715
- *copy* 633, 635
- *creation* 867
- *deletion* 867
- *mirror* 635
- *procedure* 525
- *quality* 83, 137, 655, 659
- *tools* 81

mesh based geometry 227
- *adaptive* 633
- *deletion* 769
- *export* 849
- *feature* 3
- *import* 751
- *meshedit* 715
- *preview* 541
- *remesh* 525
- *slicing* 135
- *transform coordinates* 857
- *validity* 725
- *visibility* 149

Mesh Scaling 691

mesh topology 669

metric name 655
- *algebraic* 655
- *allmetrics* 655
- *robinson* 655
- *traditional* 655

middle mouse button 55, 59, 103

midplane 257

min_through_thickness 691

mirror 635

mod 935

model axis 131, 161

morph smooth 633

mouse 55, 121
- *customization* 103
- *right click* 63
- *selecting entities with* 59, 181
- *view navigation* 55, 121

move 291, 295

msc 599, 613

multisweep 587

N

name 487
narrow regions 399
navigation 55, 121
ncdump 849
negative Jacobians 687
netcdf 849
new 41
next 113
node 713
  coincident 667
  nodehex 915
  nodeset 797, 807
  numbering 911
  repositioning 713
  selection 181
Node Redistribution 587
nodeset 797, 807
  importing 751
  repositioning 713
  size 161
  visibility 149
nogui 23
non-manifold topology 3, 669
normal 131, 465
NOT_SET 957
notation 33
numbering 911
numeric 33
NumInGrp 935

O

offset 253, 257, 267, 368, 373
open 41
optimize jacobian smoothing 883
options 103
outlet pressure 839
output 857
output window 91
overlap 403

P

painters 127
pan 55, 119, 121
parallel 849
parse 187
part 503, 509, 513
partition 449
  curves 439
  surfaces 441
  volumes 445
patch 799, 809
patran 765
pause 43
pave 555
pentagon 561
periodic 839, 1127
perspective 151
pick toolbar 59
picked group 69, 184
picking 181
pict 159
pillow 709, 901
pinpoint 563
planar 175
plane 203
playback 111
point 161
poisson ratio 823
polygonfill 127
polyhedron 565
postscript 159
power 929
ppm 159
preselection 59
pressure 831, 929
preview 211, 315, 541
  axis 211
  direction 200, 211
  location 211
  mesh 541
  plane 208
primitive
  brick 277
  cylinder 279
  frustum 283
  prism 281
  pyramid 285
  sphere 287
  torus 289
print error 935
prism 281
problem reports 19
project 255
propagate curve bias 665
property editor 87
pwd 37
pyramid 285

Q

QA_Records 21
qtri 627
quality
  controlling skew 665
  groups 485
  hexahedral 653

quadrilaterals ... 647
tetrahedral ... 651
tools ... 83
triangles ... 645
quit ... 21
quote ... 935

R

radialmesh ... 619
radians ... 935
radius ... 935
random ... 935
randomize ... 885
ray ... 189
rebar ... 788
record ... 111
references ... 1131
refine ... 691, 887
Refine Mesh Sheet ... 691
Refine Mesh Volume ... 691
Refine min_through_thickness ... 691
Refine Tet_edge ... 691
reflect ... 301
regularize ... 401
relative size ... 645, 647, 651, 653
remesh ... 525
removal ... 361, 373, 389, 391
remove ... 463, 587
remove topology ... 383
renumber ids ... 491
Renumbering Element Blocks ... 783
Renumbering Nodesets and Sidesets ... 797, 807
repositioning nodes ... 713
rescan ... 935
reset ... 21
respect tetmesh ... 599
restart ... 41
restore ... 41
restraint ... 827
restraint set ... 825
resume ... 43
reversal ... 583
Reverse ... 465
right click options ... 63
rotate ... 55, 119, 121, 291, 299
rotation ... 299

S

save ... 41
save as ... 41
scale ... 297
Scale Mesh ... 691
scaled jacobian ... 645, 647, 651, 653
scheme
automatic selection ... 637
bias ... 543
circle ... 545
curvature ... 547
delaunay ... 613
dualbias ... 543
equal ... 549
featuresize ... 869
firmness ... 637
hole ... 551
htet ... 625
mapping ... 553
mirror ... 635
multisweep ... 587
pave ... 555
pentagon ... 561
pinpoint ... 563
polyhedron ... 565
qtri ... 627
selection ... 637
sphere ... 569
stransition ... 571
stretch ... 575
stride ... 577
submap ... 579
sweep ... 587
tetinria ... 599
tetmesh ... 599
tetmsc ... 599
tetprimitive ... 605
thex ... 629
transition ... 893
triadvance ... 613
tridelaunay ... 607
trimap ... 611
trimesh ... 613
tripave ... 615
triprimitive ... 617
whisker weave ... 901
section ... 349
seed ... 481
selection ... 59, 167
separate ... 351, 353
session id ... 935
shape ... 645, 647, 651, 653
shear modulus ... 823
sheet ... 267
sheet body ... 175, 257, 353, 417
side ... 583
sideset ... 797, 807
silhouette ... 161
simplify ... 383, 459
simulog ... 599
sine ... 935
size ... 161, 691
auto ... 531
feature ... 869
interval ... 529
sizing function ... 665

bias ... 733
constant ... 739
curvature ... 741
exodus II ... 749
field ... 749
interval ... 745
inverse ... 747
linear ... 743
super ... 889
test ... 891

skeleton sizing ... 727

skew ... 346, 647, 653, 665

skew control ... 665

skinning ... 260, 781

sliver surface ... 391, 419

slot ... 378

small curves ... 395

small surfaces ... 397

smart laplacian ... 679

Smart_Smooth ... 587

smoothing
centroid area pull ... 673
edge length ... 689
equipotential ... 675
facets ... 229
laplacian ... 677
optimize condition number ... 681
optimize jacobian ... 883
optimize untangle ... 687
randomize ... 885
winslow ... 685

smoothshade ... 127

soft interval ... 527

solid model ... 23

SolidDesigner ... 225

SolidWorks ... 225

sort ... 491

specific heat ... 823

sphere ... 287, 569

spider ... 791

spline ... 175, 252

split
curve ... 325
periodic ... 327
surface ... 329

sqrt ... 935

start_id ... 756

statelist ... 901

step ... 237, 417, 517

stitch ... 417

stop ... 43

stransition ... 571

stray ... 389

stretch ... 575, 647, 653

stride ... 577

string ... 33, 935

sub-assembly ... 503

submap ... 579

subtract ... 305

suppression ... 859

surface ... 257, 267
creation ... 257
normal ... 465
overlap ... 403
removal ... 391
split ... 951
vertex type ... 583

surface area ... 935

sweep ... 257, 267, 587

sweep group ... 593

sweep surface ... 267

symmetry ... 839

syntax ... 33

T

tangent ... 465

taper ... 647, 653

target ... 374

temperature ... 829, 929

Tet_edge ... 691

tetdice ... 599

tetinria ... 599

tetmesh ... 599

tetmsc ... 599

tetprimitive ... 605

text size ... 161

thex ... 629

thicken ... 267

threshold ... 663

tile ... 1127

time ... 929

title ... 819

toggle ... 33

tolerance ... 411

tolerant imprinting ... 419

toolbars ... 99

topology ... 3

torus ... 289

tquad ... 631

transform ... 857

transition ... 893

transition map ... 571

translation ... 295

transparent ... 127

triad ... 161

triadvance ... 613

triangle coarsening ... 897

triangle visibility 131
tridelaunay 607
trim 413
trimap 611
trimesh 613
tripave 615
triprimitive 617
truehiddenline 127
tweak
  curve 367
  remove topology 383
  surface 373
  vertex 365
  volume bend 387

U

unite 307
units 929, 935
unmerge 431
untangle 687
up command 121, 151
uppercase 935
usage 23
user environment settings 23, 31
USER_SET 957
users manual 1

V

valence 65
validation 407
variable 23
velocity 929
verify 433
version 39, 515
vertex 249
vertex type 583
view 121, 135, 151, 161
virtual geometry
  deleting 463
  simplify 459
visibility 149
void 502
volume 267
  curve type 583
  draw 131
  in volume 935
  measurement 501
  partitioning 445
  quality metrics 651, 653
  units 929

W

warning 23, 221
warning count 935
webcut
  chop 309
  options 315
  sweep 311
  with arbitrary surface 323
  with planar or cylindrical surface 317
  with tool body 321
where 113
whisker weave 901
white space 919, 923
window 45
  application 45
  command 91
  entity tree 67
  input 91
  journal file editor 95
  output 91
  property 87
  query select 59
  toolbar 99
windowlocation 155
winslow smoothing 685
wireframe 127
word count 935
working directory 37

Z

zoom 119, 121