

# Mappings for Overture

## A Description of the Mapping Class

### and Documentation for Many Useful Mappings

William D. Henshaw  
Centre for Applied Scientific Computing  
Lawrence Livermore National Laboratory  
Livermore, CA, 94551  
[henshaw@llnl.gov](mailto:henshaw@llnl.gov)  
<http://www.llnl.gov/casc/people/henshaw>  
<http://www.llnl.gov/casc/Overture>

May 20, 2011

UCRL-MA-132239

### Abstract

This document describes the class **Mapping**. The **Mapping** class is used to define transformations. These transformations are used within Overture to define grids and stretching functions and rotations etc. The base class is called **Mapping**. Particular mappings such as a sphere or an annulus are defined by deriving a class from the base class and defining the particular transformation. A number of derived **Mappings** have been written including

- Various Analytical mappings: **LineMapping**, **SquareMapping**, **CircleMapping**, **AnnulusMapping**, **BoxMapping**, **CylinderMapping**, **PlaneMapping**, **QuadraticMapping**, **SphereMapping**
- **AirfoilMapping** : for creating airfoil related grids and curves (including some NACA airfoils).
- **ComposeMapping** : for composing two mappings
- **CompositeSurface** : a mapping that represents a collection of sub-surfaces.
- **CrossSectionMapping** : define a surface by cross-sections
- **DataPointMapping** : mappings defined by data points
- **DepthMapping** : create a 3D grid from a 2D grid by adding a variable depth.
- **EllipticTransform**: smooth a mapping with an elliptic transform.
- **FilletMapping**: create a fillet or collar grid to join two intersecting surfaces.
- **HyperbolicMapping**: create volume grids using hyperbolic grid generation (described else-where).
- **IntersectionMapping**: a mapping that is the intersection between two other mappings, such as the curve of intersection between two surfaces.
- **JoinMapping**: create a mapping that can join two intersecting mappings.
- **LoftedSurfaceMapping**: build a lofted surface such as a wing with a tip.
- **MatrixMapping** : define a matrix transformation by rotations, scaling, shifts etc.
- **MatrixTransform** : apply a matrix transformation to another mapping
- **NormalMapping** : define a new mapping by extending normals
- **NurbsMapping** : define a mapping by a NURBS, non-uniform rational b-spline.
- **OffsetShell** : build offset surfaces and an overlapping edge mapping to join them.
- **OrthographicTransform** : define an orthographic transform
- **ReductionMapping** : make a new Mapping from the face or edge of another mapping.

- ReorientMapping: used by ReparameterizationTransform to reorder the domain coordinate directions.
- ReparameterizationTransform : reparameterize a mapping (e.g. remove singularities, or equidistribute grid lines by arclength and curvature, or reorder the domain coordinates).
- RestrictionMapping : define a restriction to a sub-rectangle.
- RevolutionMapping : create a surface or volume of revolution
- RocketMapping : create curves related to rocket geometries.
- SmoothedPolygon : for polygons with smoothed corners
- SplineMapping: define a cubic spline curve.
- StretchMapping : one-dimensional stretching transformations
- StretchedSquare : stretch grid lines on the unit interval.
- StretchTransform : stretch grid lines along the coordinate directions
- SweepMapping : Sweep a 2D Mapping along a curve in 3D.
- TFIMapping : define a grid from given boundary curves by transfinite-interpolation (Coon's patch).
- TrimmedMapping : define a trimmed surface in 3D, the surface has portions removed from it (“trimmed”).
- UnstructuredMapping : create an unstructured representation for an existing mapping or read in an manipulate and unstructured mesh.

All these classes are described in this document.

## Contents

<b>1</b>	<b>Introduction</b>	<b>20</b>
1.1	Example:	22
<b>2</b>	<b>Class Mapping</b>	<b>22</b>
2.1	Enum Types	22
2.2	Member Functions	23
2.2.1	constructor	24
2.2.2	openDebugFiles	24
2.2.3	closeDebugFiles	24
2.2.4	basicInverse	24
2.2.5	epsilon	24
2.2.6	secondOrderDerivative	25
2.2.7	display	25
2.2.8	getIndex	25
2.2.9	get	25
2.2.10	getID	26
2.2.11	setID	26
2.2.12	getBasicInverseOption	26
2.2.13	getBoundaryCondition	26
2.2.14	getBoundingBox	26
2.2.15	getBoundingBox(indexRange,xBounds)	26
2.2.16	getBoundingBox	27
2.2.17	getCoordinateEvaluationType	27
2.2.18	getDomainBound	27
2.2.19	getDomainCoordinateSystem	27
2.2.20	getDomainCoordinateSystemBound	27

2.2.21	getDomainDimension . . . . .	28
2.2.22	getDomainSpace . . . . .	28
2.2.23	getGridDimensions . . . . .	28
2.2.24	getGrid . . . . .	28
2.2.25	getGrid . . . . .	28
2.2.26	setMinimumNumberOfDistributedGhostLines . . . . .	29
2.3	setNumberOfGhostLines . . . . .	29
2.3.1	setNumberOfDistributedGhostLines . . . . .	29
2.3.2	setGrid . . . . .	29
2.3.3	getInvertible . . . . .	29
2.3.4	getIsPeriodic . . . . .	30
2.3.5	getMappingCoordinateSystem . . . . .	30
2.3.6	getName . . . . .	30
2.3.7	getParameter . . . . .	30
2.3.8	getParameter(int) . . . . .	31
2.3.9	getPeriodVector . . . . .	31
2.3.10	getRangeBound . . . . .	31
2.3.11	getRangeCoordinateSystem . . . . .	31
2.3.12	getRangeCoordinateSystemBound . . . . .	32
2.3.13	getRangeDimension . . . . .	32
2.3.14	getRangeSpace . . . . .	32
2.3.15	getShare . . . . .	32
2.3.16	getShare . . . . .	32
2.3.17	getTypeOfCoordinateSingularity . . . . .	32
2.3.18	hasACoordinateSingularity . . . . .	33
2.3.19	intersects . . . . .	33
2.3.20	inverseMap . . . . .	33
2.3.21	inverseMapC . . . . .	34
2.3.22	inverseMapGrid . . . . .	34
2.3.23	map . . . . .	34
2.3.24	mapC . . . . .	35
2.3.25	mapGrid . . . . .	35
2.3.26	mappingHasChanged . . . . .	36
2.3.27	gridIsValid . . . . .	36
2.3.28	setGridIsValid . . . . .	36
2.3.29	periodicShift . . . . .	36
2.4	project . . . . .	36
2.4.1	put . . . . .	37
2.4.2	getGridIndexRange . . . . .	37
2.4.3	setGridIndexRange . . . . .	37
2.4.4	getNumberOfGhostPoints . . . . .	37
2.4.5	setNumberOfGhostPoints . . . . .	37
2.4.6	reinitialize . . . . .	37
2.4.7	setName . . . . .	37
2.4.8	setCoordinateEvaluationType . . . . .	38
2.4.9	setTypeOfCoordinateSingularity . . . . .	38
2.4.10	topologyMask . . . . .	38
2.4.11	getTopology . . . . .	38
2.4.12	setTopology . . . . .	38
2.4.13	setDomainDimension . . . . .	39

2.4.14	setRangeDimension . . . . .	39
2.4.15	setBasicInverseOption . . . . .	39
2.4.16	setBoundaryCondition . . . . .	39
2.4.17	setShare . . . . .	39
2.4.18	getShare . . . . .	39
2.4.19	setMappingCoordinateSystem . . . . .	39
2.4.20	setIsPeriodic . . . . .	40
2.4.21	setGridDimensions . . . . .	40
2.4.22	setInvertible . . . . .	40
2.4.23	setParameter(real) . . . . .	40
2.4.24	setParameter(int) . . . . .	41
2.4.25	setPeriodVector . . . . .	41
2.4.26	setParition . . . . .	41
2.4.27	setDomainSpace . . . . .	41
2.4.28	setRangeSpace . . . . .	41
2.4.29	setDomainCoordinateSystem . . . . .	41
2.4.30	setRangeCoordinateSystem . . . . .	42
2.4.31	setDomainBound . . . . .	42
2.4.32	setRangeBound . . . . .	42
2.4.33	setDomainCoordinateSystemBound . . . . .	42
2.4.34	setRangeCoordinateSystemBound . . . . .	42
2.4.35	useRobustInverse . . . . .	42
2.4.36	usesDistributedInverse . . . . .	43
2.4.37	usesDistributedMap . . . . .	43
2.4.38	sizeOf . . . . .	43
2.4.39	update . . . . .	43
2.4.40	update . . . . .	43
2.4.41	interactiveUpdate . . . . .	44
2.4.42	Periodic Mappings . . . . .	44
2.5	Member function map . . . . .	44
2.6	Member function getIndex . . . . .	45
2.7	Member functions inverseMap and basicInverse . . . . .	45
2.7.1	Member Functions getName , setName . . . . .	46
2.8	Coordinate singularities . . . . .	47
2.9	Coordinate systems and coordinateEvaluationType . . . . .	47
2.10	Class MappingParams . . . . .	48
2.10.1	Data Members . . . . .	48
2.11	Class ApproximateGlobalInverse . . . . .	49
2.11.1	constructor . . . . .	49
2.11.2	setGrid . . . . .	49
2.11.3	getGrid . . . . .	50
2.11.4	getBoundingBox . . . . .	50
2.11.5	getBoundingBoxTree . . . . .	50
2.11.6	getParameter . . . . .	50
2.11.7	getParameter . . . . .	50
2.11.8	setParameter . . . . .	50
2.11.9	setParameter(int) . . . . .	51
2.11.10	useRobustInverse . . . . .	51
2.11.11	usingRobustInverse . . . . .	51
2.11.12	sizeOf . . . . .	51

2.11.13 get . . . . .	51
2.11.14 put . . . . .	51
2.11.15 inverse . . . . .	52
2.11.16 initializeBoundingBoxTrees . . . . .	52
2.11.17 findNearestGridPoint . . . . .	53
2.11.18 binarySearchOverBoundary . . . . .	53
2.11.19 findNearestCell . . . . .	54
2.11.20 countCrossingsWithPolygon . . . . .	54
2.12 Class ExactLocalInverse . . . . .	55
<b>3 Inverting the Mapping by Newton's Method</b>	<b>55</b>
3.1 The case of a square Jacobian . . . . .	55
3.2 The case of a non-square Jacobian . . . . .	55
3.2.1 Method 1 : Least Squares . . . . .	56
3.2.2 Old way: minimize $l_2$ distance . . . . .	56
3.2.3 constructor . . . . .	57
3.2.4 getParameter . . . . .	57
3.2.5 setParameter . . . . .	57
3.2.6 sizeOf . . . . .	57
3.2.7 reinitialize . . . . .	57
3.2.8 get . . . . .	57
3.2.9 put . . . . .	57
3.2.10 initialize . . . . .	58
3.2.11 compressConvergedPoints . . . . .	58
3.2.12 inverse . . . . .	58
3.3 Registering Mappings and Reading Generic Mappings from the DataBase . . . . .	58
<b>4 AnnulusMapping</b>	<b>61</b>
4.1 Constructor . . . . .	61
4.2 setRadii . . . . .	62
4.3 setOrigin . . . . .	62
4.4 setAngleBounds . . . . .	62
<b>5 AirfoilMapping: create some airfoil related grids or curves</b>	<b>63</b>
5.1 NACA airfoils . . . . .	63
5.2 Joukowsky Airfoil . . . . .	64
5.3 Member function descriptions . . . . .	64
5.3.1 Constructor . . . . .	64
5.3.2 setBoxBounds . . . . .	64
5.3.3 setParameters . . . . .	65
5.3.4 setJoukowskyParameters . . . . .	65
5.4 Examples . . . . .	66
<b>6 BoxMapping</b>	<b>67</b>
6.1 Member functions . . . . .	67
6.2 constructor . . . . .	67
6.3 rotate . . . . .	67
6.4 getVertices . . . . .	67
6.5 setVertices . . . . .	68

<b>7 CircleMapping (ellipse too)</b>	<b>69</b>
7.1 Constructor(2D) . . . . .	69
7.2 Constructor(3D) . . . . .	69
<b>8 ComposeMapping: compose two mappings</b>	<b>71</b>
8.1 Constructors . . . . .	71
8.2 Member Functions . . . . .	71
<b>9 CompositeSurface: define a surface formed from many sub-surfaces</b>	<b>72</b>
9.1 Projection onto the composite surface . . . . .	72
9.1.1 Moving around sharp corners . . . . .	72
9.2 updateTopology . . . . .	73
9.3 Examples . . . . .	74
<b>10 CrossSectionMapping: define various surfaces by cross-sections</b>	<b>75</b>
10.1 Description . . . . .	75
10.2 General cross-section type . . . . .	75
10.2.1 Notes for generating general cross section mappings . . . . .	76
10.3 Ellipse cross-section type . . . . .	77
10.4 Joukowsky cross-section type . . . . .	77
10.5 Cross section Mappings with polar singularities . . . . .	77
10.6 Constructor . . . . .	77
10.7 setCrossSectionType . . . . .	78
10.8 Constructor . . . . .	78
10.9 Examples . . . . .	79
<b>11 CylinderMapping</b>	<b>81</b>
11.1 Constructor . . . . .	81
11.2 setAngle . . . . .	82
11.3 setAxis . . . . .	82
11.4 setOrientation . . . . .	83
11.5 setOrigin . . . . .	83
11.6 setRadius . . . . .	83
<b>12 DataPointMapping: create a mapping from an array of grid points</b>	<b>85</b>
12.1 Description . . . . .	85
12.2 Fast Approximate Inverse . . . . .	85
12.3 Examples . . . . .	86
12.4 Constructor . . . . .	87
12.5 getDataPoints . . . . .	87
12.6 getGridIndexRange . . . . .	87
12.7 getDimension . . . . .	87
12.8 setDataPoints . . . . .	88
12.9 setDataPoints . . . . .	88
12.10computeGhostPoints . . . . .	89
12.11setNumberOfGhostLines . . . . .	89
12.12projectGhostPoints . . . . .	89
12.13setDataPoints( fileName ) . . . . .	89
12.14setMapping . . . . .	89
12.15setMapping . . . . .	90
12.16setOrderOfInterpolation . . . . .	90

12.17	setOrderOfInterpolation . . . . .	90
12.18	useScalarArrayIndexing . . . . .	90
12.18.1	sizeOf . . . . .	90
12.19	update . . . . .	90
<b>13</b>	<b>DepthMapping: Add a depth to a 2D Mapping</b>	<b>92</b>
13.1	Description . . . . .	92
13.1.1	Quadratic depth profile . . . . .	92
13.2	Examples . . . . .	93
13.3	Constructor . . . . .	93
13.4	setDepthFunction . . . . .	93
13.5	setDepthFunction . . . . .	93
13.6	setSurface . . . . .	94
13.7	setQuadraticParameters . . . . .	94
<b>14</b>	<b>EllipticTransform</b>	<b>96</b>
14.1	Introduction . . . . .	96
14.2	The Governing Equations . . . . .	96
14.3	Control of the Boundary . . . . .	96
14.3.1	Dirichlet Conditions . . . . .	97
14.3.2	Orthogonal Boundary Conditions . . . . .	97
14.3.3	Periodic Boundaries . . . . .	98
14.4	Sources . . . . .	98
14.5	Using the Elliptic Grid Generator With Ogen . . . . .	99
14.5.1	Grid Dimensions . . . . .	99
14.5.2	Boundary Conditions . . . . .	99
14.5.3	Sources and Sinks . . . . .	99
14.5.4	Other Functions . . . . .	100
14.6	In Conclusion . . . . .	100
14.7	Member functions . . . . .	100
14.7.1	Constructor . . . . .	100
14.7.2	get . . . . .	100
14.7.3	put . . . . .	100
14.7.4	generateGrid . . . . .	101
14.8	Examples . . . . .	101
14.8.1	Smoothed out diamond airfoil . . . . .	101
<b>15</b>	<b>FilletMapping</b>	<b>103</b>
15.1	Description of Approach . . . . .	103
15.2	Fillet for two intersecting surfaces . . . . .	103
15.3	setCurves . . . . .	104
15.4	map . . . . .	104
15.5	update . . . . .	104
15.6	examples . . . . .	104
15.6.1	2D Fillet joining two lines . . . . .	104
15.6.2	Fillet to join two cylinders . . . . .	105
15.6.3	Fillet to join two spheres . . . . .	107
15.7	HyperbolicMapping . . . . .	108

<b>16 IntersectionMapping</b>	<b>108</b>
16.1 Constructor . . . . .	108
16.2 Constructor . . . . .	108
16.3 intersect . . . . .	108
16.4 intersect . . . . .	108
16.5 newtonIntersection . . . . .	109
16.6 project . . . . .	109
16.7 determineIntersection . . . . .	109
16.8 map . . . . .	110
16.9 intersectCurves . . . . .	110
16.10map . . . . .	111
16.11get . . . . .	111
16.12put . . . . .	111
16.13update . . . . .	111
<b>17 JoinMapping</b>	<b>115</b>
17.1 A 2D example . . . . .	115
17.2 Intersecting surfaces . . . . .	116
17.3 Intersecting a volume intersector mapping with a surface intersectee mapping. . . . .	117
17.4 Constructor . . . . .	118
17.5 Constructor . . . . .	119
17.6 setCurves . . . . .	119
17.7 setEndOfJoin . . . . .	119
17.8 map . . . . .	119
17.9 update . . . . .	119
17.10Class LineMapping . . . . .	120
17.11Constructor . . . . .	120
17.12Constructor . . . . .	120
17.13Constructor . . . . .	120
17.14getPoints . . . . .	120
17.15getPoints . . . . .	120
17.16getPoints . . . . .	121
17.17setPoints . . . . .	121
17.18setPoints . . . . .	121
17.19setPoints . . . . .	121
<b>18 LoftedSurfaceMapping: build a lofted surface</b>	<b>122</b>
<b>19 MatrixMapping: define a mapping from scalings, shifts and rotations</b>	<b>126</b>
19.1 Constructor . . . . .	126
19.2 rotate . . . . .	126
19.3 rotate . . . . .	127
19.4 scale . . . . .	127
19.5 shift . . . . .	127
19.6 reset . . . . .	127
19.7 matrixMatrixProduct . . . . .	128
19.8 matrixVectorProduct . . . . .	128
19.9 matrixInversion . . . . .	128

<b>20 MatrixTransform: rotate, scale or shift an existing mapping</b>	<b>129</b>
20.1 Description . . . . .	129
20.2 Constructor . . . . .	129
20.3 Constructor(Mapping&)	129
20.4 reset . . . . .	129
20.5 rotate . . . . .	129
20.6 rotate . . . . .	129
20.7 scale . . . . .	130
20.8 shift . . . . .	130
<b>21 NormalMapping: define a new mapping by extending normals</b>	<b>131</b>
21.1 Description . . . . .	131
21.2 Member Functions . . . . .	131
<b>22 NurbsMapping: define a new mapping as a NURBS.</b>	<b>133</b>
22.1 Constructor . . . . .	134
22.2 Constructor . . . . .	134
22.3 intersect3DLines . . . . .	134
22.4 buildCurveOnSurface . . . . .	135
22.5 buildComponentCurve . . . . .	135
22.6 circle . . . . .	135
22.7 conic(a,b,c,...) . . . . .	135
22.8 conic . . . . .	136
22.9 generalCylinder(curve,shift) . . . . .	136
22.10generalCylinder(curve1,curve2) . . . . .	136
22.11plane . . . . .	137
22.12getKnots . . . . .	137
22.13getControlPoints . . . . .	137
22.14insertKnot . . . . .	137
22.15insertKnot . . . . .	138
22.16readFromIgesFile . . . . .	138
22.17parametricCurve . . . . .	138
22.18shift . . . . .	138
22.19scale . . . . .	138
22.20rotate . . . . .	139
22.21rotate . . . . .	139
22.22specify knots and control points . . . . .	139
22.23specify knots and control points . . . . .	139
22.23.1 setDomainInterval . . . . .	140
22.24initialize() . . . . .	140
22.25setBounds . . . . .	140
22.26removeKnot . . . . .	141
22.27getParameterBounds . . . . .	141
22.28reparameterize . . . . .	141
22.29transformKnots . . . . .	141
22.30elevateDegree . . . . .	142
22.31merge . . . . .	142
22.32forcedMerge . . . . .	142
22.33forcedPeriodic . . . . .	142
22.34split . . . . .	143
22.35moveEndpoint . . . . .	143

22.36numberOfSubCurves . . . . .	143
22.37numberOfSubCurvesInList . . . . .	143
22.38subCurve . . . . .	143
22.39subCurveFromList . . . . .	143
22.40interpolate . . . . .	144
22.41interpolate . . . . .	144
22.42put(fileName) . . . . .	145
22.43put(FILE*) . . . . .	145
22.44get(fileName) . . . . .	145
22.45put(FILE *) . . . . .	146
22.46getOrder . . . . .	146
22.47getNumberOfKnots . . . . .	146
22.48getNumberOfControlPoints . . . . .	146
22.49buildSubCurves . . . . .	146
22.50truncateToDomainBounds . . . . .	146
22.51toggleSubCurveVisibility . . . . .	147
22.52isSubCurveHidden . . . . .	147
22.53isSubCurveOriginal . . . . .	147
22.54toggleSubCurveOriginal . . . . .	147
22.55addSubCurve . . . . .	147
22.56deleteSubCurve . . . . .	148
22.57update . . . . .	148
22.58Examples . . . . .	148
<b>23 OffsetShell: Define mappings to build a grid around a shell or plate.</b>	<b>150</b>
23.1 Defining the edge surface : an overlapping round . . . . .	150
23.2 Member function descriptions . . . . .	152
23.3 Constructor . . . . .	152
23.4 buildOffsetMappings . . . . .	152
23.5 generateVolumeGrids . . . . .	153
23.6 createOffsetMappings . . . . .	153
<b>24 OrthographicTransform : define an orthographic transform</b>	<b>153</b>
24.1 Description . . . . .	153
24.1.1 Orthographic transform to reparameterize a spherical-polar singularity . . . . .	153
24.1.2 Orthographic transform to reparameterize a cylindrical polar singularity . . . . .	154
<b>25 Member functions</b>	<b>155</b>
25.1 Default Constructor . . . . .	155
25.2 setAngularAxis . . . . .	155
25.3 setPole . . . . .	155
25.4 setSize . . . . .	156
25.5 Class PlaneMapping . . . . .	157
25.6 Constructor . . . . .	157
25.7 setPoints . . . . .	157
<b>26 QuadraticMapping: define a quadratic curve or surface.</b>	<b>158</b>
26.1 Examples . . . . .	158
26.2 Constructor . . . . .	159
26.3 setQuadraticParameters . . . . .	159
26.4 setParameters . . . . .	159

<b>27 ReductionMapping: create a Mapping from the face or edge of an existing Mapping</b>	<b>161</b>
27.1 Description . . . . .	161
27.2 Constructor . . . . .	161
27.3 Constructor . . . . .	161
27.4 Constructor . . . . .	161
27.5 set . . . . .	162
27.6 set . . . . .	162
27.7 setInActiveAxes . . . . .	162
27.8 setInActiveAxes . . . . .	163
<b>28 ReorientMapping: reorder the domain coordinates of an existing mapping</b>	<b>164</b>
28.1 Description . . . . .	164
28.2 Default Constructor . . . . .	164
28.3 setOrientation . . . . .	164
28.4 getOrientation . . . . .	164
<b>29 ReparameterizationTransform: reparameterize an existing mapping (e.g. remove a polar singularity)</b>	<b>165</b>
29.1 Description . . . . .	165
29.2 Reparameterizing a spherical-polar or cylindrical-polar singularity . . . . .	165
29.3 Default Constructor . . . . .	166
29.4 Constructor(Mapping,ReparameterizationTypes) . . . . .	166
29.5 Constructor(MappingRC,ReparameterizationTypes) . . . . .	166
29.6 constructor(MappingRC,ReparameterizationTypes) . . . . .	167
29.7 constructorForMultipleReparams . . . . .	167
29.8 scaleBound . . . . .	167
29.9 getBounds . . . . .	167
29.10 setBounds . . . . .	167
29.10.1 getBoundsForMultipleReparameterizations . . . . .	168
29.10.2 setBoundsForMultipleReparameterizations . . . . .	168
29.10.3 setEquidistributionParameters . . . . .	168
<b>30 RestrictionMapping: define a restriction to a sub-rectangle of the unit cube</b>	<b>169</b>
30.1 Description . . . . .	169
30.2 Default Constructor . . . . .	169
30.3 scaleBounds . . . . .	169
30.4 getBounds . . . . .	170
30.5 setBounds . . . . .	170
30.6 setSpaceIsPeriodic . . . . .	170
<b>31 RevolutionMapping: create a surface or volume of revolution</b>	<b>171</b>
31.1 Description . . . . .	171
31.2 Inverse of the mapping . . . . .	172
31.3 Reparameterized to spherical-like coordinates . . . . .	173
31.4 Examples . . . . .	174
31.5 Constructor . . . . .	175
31.6 Constructor . . . . .	175
31.7 setRevolutionAngle . . . . .	175
31.8 getRevolutionAngle . . . . .	176
31.9 setParameterAxes . . . . .	176
31.10 setRevolutionary . . . . .	176
31.11 setLineOfRevolution . . . . .	176

<b>32 RocketMapping: create rocket geometry curves</b>	<b>178</b>
32.1 Slot . . . . .	178
32.2 Star . . . . .	179
32.3 circle . . . . .	180
32.4 Member functions . . . . .	180
32.4.1 Constructor . . . . .	180
32.4.2 computePoints . . . . .	180
32.4.3 computeSlotPoints . . . . .	180
32.4.4 computePoints . . . . .	180
32.4.5 computeCirclePoints . . . . .	180
32.4.6 update . . . . .	180
<b>33 SmoothedPolygon</b>	<b>181</b>
33.1 update(MappingInformation &) . . . . .	182
33.2 Examples . . . . .	183
<b>34 SphereMapping</b>	<b>185</b>
34.1 Examples . . . . .	185
34.2 Constructor . . . . .	186
34.3 setOrigin . . . . .	186
34.4 setPhi . . . . .	187
34.5 setRadii . . . . .	187
34.6 setTheta . . . . .	187
<b>35 SplineMapping: create a spline curve</b>	<b>188</b>
35.1 Member functions . . . . .	188
35.1.1 Constructor . . . . .	188
35.2 shift . . . . .	189
35.3 scale . . . . .	189
35.4 rotate . . . . .	189
35.4.1 setParameterizationType . . . . .	189
35.4.2 getParameterization . . . . .	189
35.4.3 getNumberOfKnots . . . . .	189
35.4.4 setParameterization . . . . .	190
35.4.5 parameterize . . . . .	190
35.4.6 setEndConditions . . . . .	190
35.4.7 setPoints . . . . .	191
35.4.8 setPoints . . . . .	191
35.4.9 setPoints . . . . .	191
35.4.10 setShapePreserving . . . . .	191
35.4.11 setTension . . . . .	191
35.4.12 setDomainInterval . . . . .	192
35.4.13 getDomainInterval . . . . .	192
35.4.14 setIsPeriodic . . . . .	192
35.4.15 useOldSpline . . . . .	192
35.4.16 map . . . . .	192
35.4.17 update . . . . .	193
35.5 Examples . . . . .	193

<b>36 SquareMapping (rectangles too)</b>	<b>195</b>
36.1 Constructor . . . . .	195
36.2 getVertices . . . . .	195
36.3 setVertices . . . . .	195
<b>37 StretchMapping: create 1D stretching functions</b>	<b>196</b>
37.1 Inverse hyperbolic tangent stretching function . . . . .	196
37.2 Hyperbolic tangent stretching function . . . . .	197
37.3 Exponential stretching function . . . . .	199
37.4 Exponential blending function . . . . .	199
37.5 Exponential to Linear function . . . . .	199
37.6 Member function descriptions . . . . .	200
37.6.1 Constructor . . . . .	200
37.6.2 Constructor . . . . .	202
37.6.3 setStretchingType . . . . .	202
37.6.4 setNumberOfLayers . . . . .	202
37.6.5 setNumberOfIntervals . . . . .	202
37.6.6 setNumberOfSplinePoints . . . . .	203
37.6.7 setLayerParameters . . . . .	203
37.6.8 setIntervalParameters . . . . .	203
37.6.9 setEndPoints . . . . .	203
37.6.10 setIsNormalized . . . . .	203
37.6.11 setScaleParameters . . . . .	204
37.6.12 setIsPeriodic . . . . .	204
37.6.13 setHyperbolicTangentParameters . . . . .	204
37.6.14 setExponentialParameters . . . . .	204
37.6.15 setLinearSpacingParameters . . . . .	205
37.6.16 setExponentialToLinearParameters . . . . .	205
37.7 Examples . . . . .	205
<b>38 StretchedSquare: stretch grid lines on the unit interval</b>	<b>208</b>
38.1 Description . . . . .	208
<b>39 StretchTransform: stretch grid lines of an existing mapping</b>	<b>209</b>
39.1 Description . . . . .	209
39.2 Constructors . . . . .	209
39.3 Data Members . . . . .	209
39.4 Member Functions . . . . .	209
<b>40 Sweep Mapping</b>	<b>210</b>
40.1 Sweep . . . . .	210
40.2 Constructor . . . . .	211
40.3 SetSweepSurface . . . . .	212
40.4 setCentering . . . . .	212
40.5 setOrientation . . . . .	212
40.6 setExtrudeBounds . . . . .	213
40.7 setStraightLine . . . . .	213
40.8 SetSweepCurve . . . . .	213
40.9 SetScaleSpline . . . . .	213
40.10 setMappingProperties . . . . .	213
40.11 FindRowSplines . . . . .	213

40.12map . . . . .	214
40.13Examples . . . . .	214
<b>41 TFIMapping: Transfinite-Interpolation</b>	<b>215</b>
41.1 Compatibility conditions . . . . .	216
41.2 Examples . . . . .	217
41.2.1 2D linear TFI mapping with 2 sides specified . . . . .	217
41.2.2 2D hermite TFI mapping with 2 sides specified . . . . .	218
41.2.3 2D linear TFI mapping with 4 sides specified . . . . .	219
41.2.4 3D linear TFI mapping with 2 sides specified . . . . .	220
41.3 setSides . . . . .	220
41.4 flipper . . . . .	220
41.5 map . . . . .	221
41.6 update . . . . .	221
<b>42 TrimmedMapping: define a trimmed surface in 3D</b>	<b>222</b>
42.1 Description . . . . .	222
42.2 Constructor . . . . .	223
42.3 Constructor . . . . .	223
42.4 Constructor . . . . .	224
42.5 getTriangulationParameters . . . . .	224
42.6 setMaxAreaForTriangulation . . . . .	224
42.7 setMinAngleForTriangulation . . . . .	224
42.8 setElementDensityTolerance . . . . .	224
42.9 setCurves . . . . .	225
42.10setUnInitialized . . . . .	225
42.11initializeTrimCurves . . . . .	225
42.12addCurve . . . . .	225
42.13deleteTrimCurve . . . . .	225
42.14deleteTrimCurve . . . . .	226
42.15undoDelete . . . . .	226
42.16initializeQuadTree (protected) . . . . .	226
42.17getOuterCurve . . . . .	226
42.18getInnerCurve . . . . .	226
42.19curveGoesThrough . . . . .	227
42.20insideOrOutside . . . . .	227
42.21insideOrOutside . . . . .	227
42.22findClosestCurve . . . . .	228
42.23findDistanceToACurve . . . . .	228
42.24map . . . . .	229
42.25map . . . . .	229
42.26hasTriangulation . . . . .	229
42.27getTriangulation . . . . .	229
42.28triangulate . . . . .	229
42.29update . . . . .	229
42.30reportTrimCurveInfo . . . . .	230
42.31reportTrimmingInfo . . . . .	230
42.32editTrimCurve . . . . .	230
42.33editNurbsTrimCurve . . . . .	230

<b>43 UnstructuredMapping</b>	<b>231</b>
43.1 Connectivity Iterator Interface . . . . .	231
43.2 Introduction . . . . .	231
43.3 Additions to the <b>UnstructuredMapping</b> class . . . . .	232
43.3.1 Generic size and data access methods . . . . .	232
43.3.2 Iterator related methods . . . . .	233
43.3.3 Entity and connectivity manipulations . . . . .	234
43.4 Iterator Classes . . . . .	235
43.4.1 Common methods and operators . . . . .	235
43.4.2 <b>UnstructuredMappingIterator</b> constructor . . . . .	235
43.4.3 <b>UnstructuredMappingAdjacencyIterator</b> constructor . . . . .	236
43.4.4 <b>UnstructuredMappingAdjacencyIterator</b> orientation . . . . .	236
43.5 The tagging interface . . . . .	236
43.6 Other useful stuff . . . . .	238
43.7 Examples . . . . .	239
43.7.1 Iterating through entities of a given type . . . . .	239
43.7.2 Adjacency iterations . . . . .	240
43.7.3 Tag all vertices on the boundary of the mesh . . . . .	240
43.7.4 Computing 2D Face areas . . . . .	241
43.7.5 A generic finite volume gradient operator . . . . .	242
43.8 Implementation Details . . . . .	245
43.9 Iterations using the old connectivity interface . . . . .	245
43.9.1 Element iteration . . . . .	245
43.9.2 Vertex iteration . . . . .	245
43.9.3 Iteration through the vertices in an element . . . . .	245
43.9.4 Iteration through the faces . . . . .	246
43.9.5 Iteration through the vertices in a face . . . . .	246
43.10Enum Types . . . . .	246
43.11File Formats . . . . .	247
43.12Relationship to Normal Overture Mappings . . . . .	247
43.13New Connectivity Interface . . . . .	247
43.14Member Function Descriptions . . . . .	247
43.15Constructor . . . . .	247
43.16Constructor . . . . .	248
43.17addGhostElements . . . . .	248
43.18getBoundaryFace . . . . .	248
43.19getGhostElements . . . . .	248
43.20getMask . . . . .	248
43.21getBoundaryFaceTags . . . . .	248
43.22getNumberOfNodes . . . . .	248
43.23getMaxNumberOfNodesPerElement . . . . .	249
43.24getMaxNumberOfNodesPerElement . . . . .	249
43.25getMaxNumberOfNodesPerFace . . . . .	249
43.26getNumberOfElements . . . . .	249
43.27getNumberOfFaces . . . . .	249
43.28getNumberOfBoundaryFaces . . . . .	249
43.29getNumberOfEdges . . . . .	249
43.30getNodes . . . . .	250
43.31getElements . . . . .	250
43.32getFaces . . . . .	250

43.33getFaces . . . . .	250
43.34getEdges . . . . .	250
43.35getElementFaces . . . . .	251
43.36getTags . . . . .	251
43.37setElementDensityTolerance . . . . .	251
43.38setTags . . . . .	251
43.39setNodesAndConnectivity . . . . .	251
43.40setNodesElementsAndNeighbours . . . . .	252
43.41setNodesAndConnectivity . . . . .	252
43.42buildFromAMapping . . . . .	253
43.43printConnectivity . . . . .	253
43.44printConnectivity . . . . .	253
43.45printStatistics . . . . .	253
43.46get from an ascii file . . . . .	254
43.47put to an ascii file . . . . .	254
43.48findBoundaryCurves . . . . .	254
43.49Constructor . . . . .	254
43.50Constructor . . . . .	254
43.51getNumberOfNodes . . . . .	254
43.52getMaxNumberOfNodesPerElement . . . . .	255
43.53getMaxNumberOfNodesPerElement . . . . .	255
43.54getMaxNumberOfNodesPerFace . . . . .	255
43.55getNumberOfElements . . . . .	255
43.56getNumberOfFaces . . . . .	255
43.57getNumberOfBoundaryFaces . . . . .	255
43.58getNumberOfEdges . . . . .	255
43.59getNodes . . . . .	256
43.60getElements . . . . .	256
43.61getFaces . . . . .	256
43.62getFaces . . . . .	256
43.63getEdges . . . . .	256
43.64getTags . . . . .	257
43.65setTags . . . . .	257
43.66setNodesAndConnectivity . . . . .	257
43.67project . . . . .	257
43.68buildFromAMapping . . . . .	257
43.69get from an ascii file . . . . .	258
43.70put to an ascii file . . . . .	258
43.71Constructor . . . . .	258
43.72getColour . . . . .	258
43.73setColour . . . . .	259
43.74eraseUnstructuredMapping . . . . .	259
43.75getColour . . . . .	259
43.76setColour . . . . .	259
43.77eraseUnstructuredMapping . . . . .	259
43.78addTag . . . . .	259
43.79deleteTag . . . . .	260
43.80deleteTag . . . . .	260
43.81hasTag . . . . .	260
43.82getTag . . . . .	261

43.83getTagData . . . . .	261
43.84setData . . . . .	261
43.85maintainTagToEntityMap . . . . .	262
43.86maintainsTagToEntityMap . . . . .	262
43.87getColour . . . . .	262
43.88setColour . . . . .	262
43.89eraseUnstructuredMapping . . . . .	263
43.90addTag . . . . .	263
43.91deleteTag . . . . .	263
43.92deleteTag . . . . .	263
43.93hasTag . . . . .	264
43.94getTag . . . . .	264
43.95getTagData . . . . .	264
43.96setData . . . . .	264
43.97maintainTagToEntityMap . . . . .	265
43.98maintainsTagToEntityMap . . . . .	265
43.99getColour . . . . .	266
43.10setColour . . . . .	266
43.10eraseUnstructuredMapping . . . . .	266
43.10addTag . . . . .	266
43.10deleteTag . . . . .	266
43.10deleteTag . . . . .	267
43.10hasTag . . . . .	267
43.10getTag . . . . .	267
43.10getTagData . . . . .	268
43.10setData . . . . .	268
43.10maintainTagToEntityMap . . . . .	268
43.10maintainsTagToEntityMap . . . . .	269
43.11getColour . . . . .	269
43.11setColour . . . . .	270
43.11eraseUnstructuredMapping . . . . .	270
43.11addTag . . . . .	270
43.11deleteTag . . . . .	270
43.11deleteTag . . . . .	271
43.11hasTag . . . . .	271
43.11getTag . . . . .	271
43.11getTagData . . . . .	271
43.12setData . . . . .	272
43.12maintainTagToEntityMap . . . . .	272
43.12maintainsTagToEntityMap . . . . .	272
43.12getColour . . . . .	273
43.12setColour . . . . .	273
43.12eraseUnstructuredMapping . . . . .	273
43.12addTag . . . . .	274
43.12deleteTag . . . . .	274
43.12deleteTag . . . . .	274
43.12hasTag . . . . .	275
43.13getTag . . . . .	275
43.13getTagData . . . . .	275
43.13setData . . . . .	275

43.13 <del>3</del> maintainTagToEntityMap . . . . .	276
43.13 <del>4</del> maintainsTagToEntityMap . . . . .	276
43.13 <del>5</del> getColour . . . . .	276
43.13 <del>6</del> setColour . . . . .	276
43.13 <del>7</del> eraseUnstructuredMapping . . . . .	276
43.13 <del>8</del> addTag . . . . .	277
43.13 <del>9</del> deleteTag . . . . .	277
43.14 <del>10</del> deleteTag . . . . .	277
43.14 <del>11</del> hasTag . . . . .	278
43.14 <del>12</del> getTag . . . . .	278
43.14 <del>13</del> getTagData . . . . .	278
43.14 <del>14</del> setTagData . . . . .	278
43.14 <del>15</del> maintainTagToEntityMap . . . . .	279
43.14 <del>16</del> maintainsTagToEntityMap . . . . .	279
43.14 <del>17</del> entitiesAreEquivalent . . . . .	279
43.14 <del>18</del> getColour . . . . .	279
43.14 <del>19</del> setColour . . . . .	279
43.15 <del>20</del> eraseUnstructuredMapping . . . . .	280
43.15 <del>21</del> addTag . . . . .	280
43.15 <del>22</del> deleteTag . . . . .	280
43.15 <del>23</del> deleteTag . . . . .	280
43.15 <del>24</del> hasTag . . . . .	281
43.15 <del>25</del> getTag . . . . .	281
43.15 <del>26</del> getTagData . . . . .	281
43.15 <del>27</del> setTagData . . . . .	281
43.15 <del>28</del> maintainTagToEntityMap . . . . .	282
43.15 <del>29</del> maintainsTagToEntityMap . . . . .	282
43.16 <del>30</del> entitiesAreEquivalent . . . . .	282
43.16 <del>31</del> getColour . . . . .	282
43.16 <del>32</del> setColour . . . . .	282
43.16 <del>33</del> eraseUnstructuredMapping . . . . .	283
43.16 <del>34</del> addTag . . . . .	283
43.16 <del>35</del> deleteTag . . . . .	283
43.16 <del>36</del> deleteTag . . . . .	283
43.16 <del>37</del> hasTag . . . . .	284
43.16 <del>38</del> getTag . . . . .	284
43.16 <del>39</del> getTagData . . . . .	284
43.17 <del>40</del> setTagData . . . . .	284
43.17 <del>41</del> maintainTagToEntityMap . . . . .	285
43.17 <del>42</del> maintainsTagToEntityMap . . . . .	285
43.17 <del>43</del> entitiesAreEquivalent . . . . .	285
43.17 <del>44</del> getColour . . . . .	285
43.17 <del>45</del> setColour . . . . .	285
43.17 <del>46</del> eraseUnstructuredMapping . . . . .	286
43.17 <del>47</del> addTag . . . . .	286
43.17 <del>48</del> deleteTag . . . . .	286
43.17 <del>49</del> deleteTag . . . . .	286
43.18 <del>50</del> hasTag . . . . .	287
43.18 <del>51</del> getTag . . . . .	287
43.18 <del>52</del> getTagData . . . . .	287

43.183setTagData . . . . .	287
43.184maintainTagToEntityMap . . . . .	288
43.185maintainsTagToEntityMap . . . . .	288
43.186entitiesAreEquivalent . . . . .	288
43.187getColour . . . . .	288
43.188setColour . . . . .	288
43.189eraseUnstructuredMapping . . . . .	289
43.190addTag . . . . .	289
43.191deleteTag . . . . .	289
43.192deleteTag . . . . .	289
43.193hasTag . . . . .	290
43.194getTag . . . . .	290
43.195getTagData . . . . .	290
43.196setTagData . . . . .	290
43.197maintainTagToEntityMap . . . . .	291
43.198maintainsTagToEntityMap . . . . .	291
43.199entitiesAreEquivalent . . . . .	291
<b>44 Class Fraction</b>	<b>293</b>
44.1 Constructors . . . . .	293
44.2 Member Functions . . . . .	293
<b>45 Class Bound</b>	<b>293</b>
45.1 enum types . . . . .	293
45.2 Constructors . . . . .	293
45.3 Member Functions . . . . .	294
<b>46 Class Triangle</b>	<b>294</b>
46.1 Constructor . . . . .	294
46.2 Constructor( const real x1[],x2[],x3[] ) . . . . .	294
46.3 Constructor( const RealArray & x1,x2,x3 ) . . . . .	294
46.4 Constructor(grid) . . . . .	294
46.5 setVertices(const real x1,x2,x3) . . . . .	295
46.6 setVertices( const RealArray & x1,x2,x3 ) . . . . .	295
46.7 setVertices . . . . .	295
46.8 area . . . . .	296
46.9 display . . . . .	296
46.10tetrahedralVolume . . . . .	296
46.11intersects . . . . .	296
46.12intersects . . . . .	296
46.13intersects . . . . .	297
46.14intersects . . . . .	297
46.15intersects . . . . .	297
46.16intersects . . . . .	297
46.17getRelativeCoordinates . . . . .	298

# 1 Introduction

The C++ class “Mapping” can be used to define the “mappings” (transformations) and their properties. For example, each component grid in an overlapping grid will contain a member function that defines the mapping from the unit square (or unit cube) onto the domain covered by the grid. This mapping may in turn be defined in terms of the curves (or surfaces) that form its boundaries. Stretching functions as well as rotations and scalings are all defined by mappings.

New mappings are defined by derivation from the base class “Mapping”. For example, the class “MatrixMapping” is a derived class that defines transformations such as rotations, scalings and translations.

The mapping class can be used to define mapping functions for curves, areas, surfaces, volumes etc.:

$$f : \mathbf{R}^{domainDimension} \rightarrow \mathbf{R}^{rangeDimension} \quad domainDimension \leq rangeDimension , \quad rangeDimension = 0, 1, 2, 3$$

For example a curve in 2D would have  $(domainDimension, rangeDimension) = (1, 2)$  and a volume in 3D would have  $(domainDimension, rangeDimension) = (3, 3)$

$\mathbf{R}^{domainDimension}$  is called the **domain** of the mapping while  $\mathbf{R}^{rangeDimension}$  is called the **range**.

The domain will either be **parameter space** (i.e. unit line, unit square, or unit cube) or **cartesian space** (i.e. physical space with coordinates  $(x_1, x_2, x_3)$ ). Similarly, the range is either parameter space or cartesian space.



Figure 1: Class diagram for a Mapping

## 1.1 Example:

Here is a simple example of creating and evaluating a mapping and its inverse. (file `example1.C`)

```
1 #include "Mapping.h"
2 #include "BoxMapping.h"
3
4 int
5 main()
6 {
7     int axis1=0;  int axis2=1;  int axis3=2;
8
9 // -- Define a box in 3D
10
11 BoxMapping cube(1.,2.,1.,2.,1.,2.) ;           // create a cube: [1,2]x[1,2]x[1,2]
12
13 cube.setName(Mapping::mappingName,"cube");      // give the mapping a name
14
15 cube.setIsPeriodic(axis1,Mapping::derivativePeriodic); // periodic in x direction
16
17 RealArray r(1,3),x(1,3),xr(1,3,3),rx(1,3,3);    // evaluate only 1 point
18
19 r(0,axis1)=.25; r(0,axis2)=.5; r(0,axis3)=.75;
20 r.display("here is r");
21 cube.map( r,x,xr );                         // evaluate the mapping and derivatives: r --> (x,xr)
22 x.display("here is x after map");
23
24 r=0;
25 cube.inverseMap( x,r,rx );                  // evaluate the inverse mapping: x --> (r,rx)
26 r.display("here is r after inverseMap");
27
28 return 0;
29 }
30 }
```

## 2 Class Mapping

The base class for mappings is the class `Mapping`.

### 2.1 Enum Types

The following enum types are members of Class `Mapping`.

Here are the enumerators for the possible spaces for the domain and range

```
enum mappingSpace{
    parameterSpace,    // bounds are [0,1]
    cartesianSpace }   // default (-infinity,infinity)
};
```

For example, a stretching function will normally map from `parameterSpace` to `parameterSpace`. A square grid will usually be a mapping from `parameterSpace` to `cartesianSpace` (i.e. physical space with coordinates  $(x_1, x_2, x_3)$ ). A rotation will normally be a mapping from `cartesian space` to `cartesian space`.

Here are the enumerators used to define the periodicity of the mapping (i.e. possible values for `getIsPeriodic`)

```
enum periodicType
{
    notPeriodic,
```

```

derivativePeriodic,    // Derivative is periodic but not the function
functionPeriodic      // Function is periodic
};

```

Here are the enumerators for the coordinate systems that we can use for the domain or the range

```

enum coordinateSystem{
    cartesian,           // x,y,z
    spherical,           // phi/pi, theta/2pi, r
    cylindrical,         // theta/2pi, z, r
    polar,               // r, theta/2pi, z
    toroidal             // theta1/2pi, theta2/2pi, theta3/2pi
};

```

Coordinate systems are discussed in greater detail later.

Here are the enumerators for the items that we save names for in the form of character strings,

```

enum mappingItemName
{
    mappingName,          // mapping name
    domainName,           // domain name
    rangeName,
    domainAxis1Name,      // names for coordinate axes in domain
    domainAxis2Name,
    domainAxis3Name,
    rangeAxis1Name,        // names for coordinate axes in range
    rangeAxis2Name,
    rangeAxis3Name
};

```

The names are assigned and retrieved with the member functions `setName` and `getName`.

Here are the enumerators used to supply options to `setBasicInverseOption`

```

enum basicInverseOptions // options for basicInverse
{
    canDoNothing,
    canDetermineOutside,
    canInvert
};

```

Use the `setBasicInverseOption` or `getBasicInverseOption` functions to retrieve or change these values.

Here are enumerators for the types of Mapping coordinate systems, these are used to optimize the computation of difference approximations to functions defined on grids derived from this mapping.

```

enum mappingCoordinateSystem
{
    rectangular,          // rectangular mapping
    conformal,            // conformal : metric tensor is diagonal and ...
    orthogonal,           // orthogonal mapping : metric tensor is diagonal
    general               // general transformation : no special properties
};

```

Use the `setMappingCoordinateSystem` and `getMappingCoordinateSystem` functions to retrieve or change the mapping coordinate system.

## 2.2 Member Functions

In the following `real` will denote either `float` or `double`.

### 2.2.1 constructor

```
Mapping(int domainDimension_ =3,  
        int rangeDimension_ =3,  
        mappingSpace domainSpace_ =parameterSpace,  
        mappingSpace rangeSpace_ =cartesianSpace,  
        coordinateSystem domainCoordinateSystem_ =cartesian,  
        coordinateSystem rangeCoordinateSystem_ =cartesian)
```

**Description:** Default Constructor.

**domainDimension\_ (input):**

**rangeDimension\_ (input):**

**domainSpace\_ (input):**

**rangeSpace\_ (input):**

**domainCoordinateSystem\_ (input):**

**rangeCoordinateSystem\_ (input):**

### 2.2.2 openDebugFiles

```
void  
openDebugFiles()
```

**Description:** Open the Mapping debug files.

### 2.2.3 closeDebugFiles

```
void  
closeDebugFiles()
```

**Description:** Close the Mapping debug files.

### 2.2.4 basicInverse

```
void  
basicInverse(const realArray & x,  
            realArray & r,  
            realArray & rx =nullDistributedArray,  
            MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** A derived class may optionally define this function if the class knows how to rapidly compute the inverse of the mapping (by an analytic formula for example).

### 2.2.5 epsilon

```
real  
epsilon()
```

**Description:** Return the tolerance used by the Mappings.

## 2.2.6 secondOrderDerivative

```
void  
secondOrderDerivative(const Index & I,  
                      const realArray & r,  
                      realArray & xrr,  
                      const int axis,  
                      const int & rAxis )
```

**Description:** compute second derivatives of the mapping by finite differences

**I (input) :**

**r (input) :** evaluate at these points, r(I,0:domainDimension-1).

**xrr (output):**

**axis (input):** compute the derivative of x(axis,I)

**rAxis (input):** compute the second derivative along the direction rAxis.

## 2.2.7 display

```
void  
display( const aString & label ) const
```

**Description:** Write the values of the Mapping parameters to standard output.

## 2.2.8 getIndex

**Index**

```
getIndex(const realArray & r,  
        realArray & x,  
        const realArray & xr,  
        int & base0,  
        int & bound0,  
        int & computeMap0,  
        int & computeMapDerivative0 )
```

**Description:** Return an Index operator for loops in the map and inverseMap functions Also compute the members:

**computeMapping :** TRUE or FALSE

**computeMappingDerivative :** TRUE or FALSE

**base :** base for Index

**bound :** bound for the Index

**NOTE:** do note make x "const" so we check that this routine is called correctly from map and inverseMap

## 2.2.9 get

```
int  
get( const GenericDataBase & dir, const aString & name)
```

**Description:** Get this object from a sub-directory called "name"

### **2.2.10 getID**

```
int  
getID() const
```

**Description:** Get the current value for the Mapping identifier, a unique number to use when saving the Mapping in a database file. This value is used to avoid having multiple copies of a Mapping saved in a data base file.

### **2.2.11 setID**

```
void  
setID()
```

**Description:** Set a new value for the Mapping identifier, a unique number to use when saving the Mapping in a database file. This value is used to avoid having multiple copies of a Mapping saved in a data base file.

### **2.2.12 getBasicInverseOption**

```
basicInverseOptions  
getBasicInverseOption() const
```

**Description:**

### **2.2.13 getBoundaryCondition**

```
int  
getBoundaryCondition( const int side, const int axis ) const
```

**Description:** Return the boundary condition code for a side of the mapping. A positive value denotes a physical boundary, 0 an interpolation boundary and a negative value a periodic direction.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### **2.2.14 getBoundingBox**

```
RealArray  
getBoundingBox(const int & side = -1,  
              const int & axis = -1) const
```

**Description:** Return the bounding box for the Mapping (if side $\neq$ 0 and axis $\neq$ 0) or the bounding box for a particular side.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### **2.2.15 getBoundingBox(indexRange,xBounds)**

```
int  
getBoundingBox( const IntegerArray & indexRange, const IntegerArray &  
               gridIndexRange_,  
               RealArray & xBounds, bool local = false) const
```

**Description:** Return the bounding box, xBounds, for the set of grid points spanned by indexRange.

**indexRange(0: 1,0:2) (input)** : range of indices, i\_m=indexRange(0,m),...,indexRange(1,m)

**gridIndexRange\_(0: 1,0:2) (input)** : Normally these should match the gridIndexRange of the Mapping.  
This argument is used to double check that this is true.

**xBounds(0: 1,0:2) :** bounds

**local (input)** : if local=true then only compute the min and max over points on this processor, otherwise compute the min and max over all points on all processors

**Return values:** 0=success, 1=indexRange values are invalid, 2=cannot compute bounds with local=true since the indexRange values do not lie within the local array.

## 2.2.16 getBoundingBox

```
const BoundingBox &
getBoundingBoxTree( const int & side,
                    const int & axis ) const
```

**Description:** Return the BoundingBox (tree) for a side of a Mapping.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

## 2.2.17 getCoordinateEvaluationType

```
int
getCoordinateEvaluationType( const coordinateSystem type ) const
```

**Description:**

## 2.2.18 getDomainBound

```
Bound
getDomainBound( const int side, const int axis ) const
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

## 2.2.19 getDomainCoordinateSystem

```
coordinateSystem
getDomainCoordinateSystem() const
```

**Description:**

## 2.2.20 getDomainCoordinateSystemBound

```
Bound
getDomainCoordinateSystemBound( const int side, const int axis ) const
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

### 2.2.21 getDomainDimension

```
int  
getDomainDimension() const
```

**Description:**

### 2.2.22 getDomainSpace

```
mappingSpace  
getDomainSpace() const
```

**Description:**

### 2.2.23 getGridDimensions

```
int  
getGridDimensions( const int axis ) const
```

**Description:**

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### 2.2.24 getGrid

```
const realArray&  
getGrid(MappingParameters & params = nullMappingParameters(),  
        bool includeGhost =false)
```

**Description:** Return an array that holds the values of this Mapping evaluated on an array of equally spaced points. Note that this array may or may not contain ghost points. If  $x$  denotes the array that is returned, then the values that are guaranteed to be there are

$$x(0 : n_0, 0 : n_1, 0 : n_2, 0 : rangeDimension - 1)$$

where  $n_i = getGridDimensions(i) - 1$ . Thus the valid values will always start with base 0 in the array. The array  $x$  may have ghost points in which case the base will be less than 0 and the bound greater than  $n_i$ .

**Return value:** An array  $x$

**Note:** For efficiency the array is returned by reference. Thus **you should not alter the array that is returned by this routine.**

### 2.2.25 getGrid

```
const RealArray&  
getGridSerial(MappingParameters & params = nullMappingParameters(),  
              bool includeGhost =false)
```

**Description:** Return an array that holds the values of this Mapping evaluated on an array of equally spaced points. Note that this array may or may not contain ghost points. If  $x$  denotes the array that is returned, then the values that are guaranteed to be there are

$$x(0 : n_0, 0 : n_1, 0 : n_2, 0 : rangeDimension - 1)$$

where  $n_i = getGridDimensions(i) - 1$ . Thus the valid values will always start with base 0 in the array. The array x may have ghost points in which case the base will be less than 0 and the bound greater than  $n_i$ .

**Return value:** An array x

**Note:** gridSerial is used for the parallel inverse. If the inverse is distributed then gridSerial will equal the local-array-with-ghost-boundaries of grid. Otherwise grid-serial will hold the entire grid on each processor. In a serial computation, gridSerial is the same as grid.

**Note:** For efficiency the array is returned by reference. Thus **you should not alter the array that is returned by this routine.**

### 2.2.26 setMinimumNumberOfDistributedGhostLines

**void**

**setMinimumNumberOfDistributedGhostLines( int numGhost )**

**Description:** On Parallel machines always add at least this many parallel ghost lines on the grid array.

## 2.3 setNumberOfGhostLines

**int**

**setNumberOfGhostLines( IndexRangeType & numberOfGhostLinesNew )**

**Description:** Specify the number of ghost lines.

**numberOfGhostLinesNew(side, axis) :** specify the number of ghostlines.

### 2.3.1 setNumberOfDistributedGhostLines

**void**

**setNumberOfDistributedGhostLines( int numGhost )**

**Description:** Specify the number of parallel ghost lines to use for this mapping (on the "grid" array)

### 2.3.2 setGrid

**void**

**setGrid(realArray & grid\_, IntegerArray & gridIndexRange\_ )**

**Description:** Provide the grid to be used for plotting and the inverse. The MappedGrid, for example, can provide the vertex array to use and thus this array can be shared (to save space).

**grid\_ (input) :** grid points with optional ghost points.

**gridIndexRange\_(0: 1,0:2) :** grid index range – this array indicates the index values in the array grid that corresponding to the domain boundaries. Values outside of these index bounds are this ghost points. For now gridIndexRange\_(0,0:2) should always be zero.

### 2.3.3 getInvertible

**int**

**getInvertible() const**

**Description:**

### 2.3.4 getIsPeriodic

**periodicType**  
**getIsPeriodic( const int axis ) const**

**Description:**

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### 2.3.5 getMappingCoordinateSystem

**mappingCoordinateSystem**  
**getMappingCoordinateSystem() const**

**Description:**

### 2.3.6 getName

**aString**  
**getName( const mappingItemName item ) const**

**Description:** Return a name from enum mappingItemName:

**mappingName** : mapping name  
**domainName** : domain name  
**rangeName** :  
**domainAxis1Name** : names for coordinate axes in domain  
**domainAxis2Name** :  
**domainAxis3Name** :  
**rangeAxis1Name** : names for coordinate axes in range  
**rangeAxis2Name** :  
**rangeAxis3Name** :

**item (input):** return the name of this item.

### 2.3.7 getParameter

**real**  
**getParameter( const realParameter & param ) const**

**Description:** Return the value of a parameter used by the Mapping or the ApproximateGlobalInverse or the ExactLocalInverse.

**THEnonConvergenceValue** : value given to "r" value of the inverse when there is no convergence.  
This is currently equal to 10. and cannot be changed.

**THEnewtonToleranceFactor** : convergence tolerance is this times the machine epsilon. Default=100. ?

**THEnewtonDivergenceValue** : newton is deemed to have diverged if the r value is this much outside [0,1]. The default value is .1 and so Newton is deemed to have diverged when the r value is outside the range [-.1,1.1]

**THEnewtonL2Factor** : extra factor for finding the closest point to a curve or surface, default=.1.

This factor allows a less strict convergence factor if the target point is far from the mapping. Decrease this value if you want a more accurate answer. You may also have to decrease this value for mappings that have poor parameterizations.

**THEboundingBoxExtensionFactor** : relative amount to increase the bounding box each direction. The bounding box can be increased in size to allow the inverse function to still converge for nearby points. The default value is .01. \*\*\*Actually\*\*\* only the bounding boxes for the highest leaves in the bounding box tree are extended by this factor. The bounding boxes for all other nodes (and the root) are just computed from the size of the bounding boxes of the two leaves of the node.

**THEstencilWalkBoundingBoxExtensionFactor** : The stencil walk routine that finds the closest point before inversion by Newton's method will only find the closest point if the point lies in a box that is equal to the bounding box extended by this factor in each direction. Default =.2

### 2.3.8 getParameter(int)

```
int  
getParameter( const intParameter & param ) const
```

**Description:** Set the value of a parameter used by the Mapping or the ApproximateGlobalInverse or the ExactLocalInverse.

**THEfindBestGuess** : if true, always find the closest point, even if the point to be inverted is outside the bounding box. Default value is false.

### 2.3.9 getPeriodVector

```
real  
getPeriodVector(const int axis, const int direction ) const
```

**Description:** For a mapping with `getIsPeriodic(direction)==derivativePeriodic` this routine returns the vector that determines the shift from the ‘left’ edge to the ‘right’ edge.

**axis (input):**  $axis = (0,1,2)$  (or  $axis = (axis1,axis2,axis3)$ ) with  $axis < rangeDimension$ , are the components of the vector.

**direction (input) :**  $direction = 0,1,\dots, domainDimension$

### 2.3.10 getRangeBound

```
Bound  
getRangeBound( const int side, const int axis ) const
```

**Description:**

**side, axis (input):** indicates the side of the mapping,  $side=(0,1)$  (or  $side=(Start,End)$ ) and  $axis = (0,1,2)$  (or  $axis = (axis1,axis2,axis3)$ ) with  $axis < domainDimension$ .

### 2.3.11 getRangeCoordinateSystem

```
coordinateSystem  
getRangeCoordinateSystem() const
```

**Description:**

### 2.3.12 getRangeCoordinateSystemBound

Bound

getRangeCoordinateSystemBound( const int side, const int axis ) const

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### 2.3.13 getRangeDimension

int

getRangeDimension() const

**Description:**

### 2.3.14 getRangeSpace

mappingSpace

getRangeSpace() const

**Description:**

### 2.3.15 getShare

int

getShare( const int side, const int axis ) const

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### 2.3.16 getShare

real

getSignForJacobian() const

**Description:** Return the sign of the jacobian, 1 (right handed coordinate system) or -1 (left handed).  
This may only make sense for some mappings.

### 2.3.17 getTypeOfCoordinateSingularity

coordinateSingularity

getTypeOfCoordinateSingularity( const int side, const int axis ) const

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

### 2.3.18 hasACoordinateSingularity

```
int  
hasACoordinateSingularity() const
```

**Description:** return true if the Mapping has a coordinate singularity

### 2.3.19 intersects

```
int  
intersects(Mapping & map2,  
           const int & side1 = -1,  
           const int & axis1 = -1,  
           const int & side2 = -1,  
           const int & axis2 = -1,  
           const real & tol = 0.) const
```

**Description:** Determine one mapping (or a face of the mapping) intersects another mapping (or the face of another mapping).

**map2 (input):** check intersect with this Mapping.

**side1, axis1 (input):** Check this face of this mapping (by default check all faces).

**side2, axis2 (input):** Check this face of map2 (by default check all faces).

**tol (input) :** increase the size of the bounding boxes by tol\*(box size) when determining whether the mappings intersect. Thus choosing a value of .1 will cause the Mappings to intersect provided they are close to each other while a value of -.1 will cause the mappings to intersect only if they overlap sufficiently. Return value : TRUE if the face (side, axis) of map intersects this mapping.

### 2.3.20 inverseMap

```
void  
inverseMap(const realArray & x_,  
           realArray & r_,  
           realArray & rx_ = nullDistributedArray,  
           MappingParameters & params = Overture::nullMappingParameters())
```

**Description:** — Here is the generic inverse —

**x (input) :** invert these points. The dimensions of this array will determine which points are inverted.

**r (input/output) :** On input this is an initial guess. If you know a reasonable initial guess then supply it, If you don't know an initial guess then set r=-1. for those points that you do not know a guess. If you do not know a guess then do NOT specify some valid value like .5 since this will probably be slower than allowing the value to be automatically generated.

**rx (output):** the derivatives of the inverse mapping.

**params (input) : params.computeGlobalInverse :** TRUE means compute a full global inverse, FALSE means only compute a local inverse using the initial guess supplied in r

**params.periodicityOfSpace :**

**params.periodVector :**

### 2.3.21 inverseMapC

```
void  
inverseMapC(const realArray & x,  
            const realArray & r,  
            const realArray & rx =nullDistributedArray,  
MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** This version of inverseMap defines x and xr to be const (even though they really aren't).

It can be used for some compilers (IBM:xlC) that don't like passing views of arrays to non-const references, as in mapping.inverseMapC(r(I),x(I),xr(I))

### 2.3.22 inverseMapGrid

```
void  
inverseMapGrid(const realArray & x,  
               realArray & r,  
               realArray & rx =nullDistributedArray,  
MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** inverseMap a grid of points.

This version of inverseMap assumes that the input array is of the form of a grid of points:

```
if rangeDimension==1 then x can be of the form  
  x(a1:a2,0:d-1)  
  x(a1:a2,0:0,0:d-1)  
  x(a1:a2,0:0,0:0,0:d-1)  
if rangeDimension==2 then x can be of the form  
  x(a1:a2,b1:b2,0:d-1)  
  x(a1:a2,b1:b2,0:0,0:d-1)  
if rangeDimension==3 then x can be of the form  
  x(a1:a2,b1:b2,c1:c2,0:d-1)
```

The output is in a similar form

**x (input)** : evaluate the inverse mapping at these points, where

**r (input/output)** : if r has enough space, then compute the inverse mapping. You must supply an initial guess. Choose r=-1. if you don't know a good guess.

**rx (output)** : if rx has enough space, then compute the derivatives of the inverse mapping.

**params (input/output)** : holds parameters for the mapping.

### 2.3.23 map

```
void  
map(const realArray & r,  
     realArray & x,  
     realArray & xr =nullDistributedArray,  
MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** Here is the transformation that defines the mapping.

**r (input):** r(base:bound,0:d) - evaluate the mapping at these points, where d=domainDimension-1

**x (output) :** - if x has enough space, x(base:bound,0:r), then compute the mapping. Here r=rangeDimension-1. Do not compute the mapping if x is not large enough

**xr (output) :** - if xr has enough space, xr(base:bound,0:r,0:d), then compute the derivatives of the mapping.

**params (input):** - holds parameters for the mapping.

### 2.3.24 mapC

```
void  
mapC(const realArray & r,  
      const realArray & x,  
      const realArray & xr =nullDistributedArray,  
      MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** This version of map defines x and xr to be const (even though they really aren't). It can be used for some compilers (IBM:xlC) that don't like passing views of arrays to non-const references, as in mapping.mapC(r(I),x(I),xr(I))

### 2.3.25 mapGrid

```
void  
mapGrid(const realArray & r,  
        realArray & x,  
        realArray & xr =nullDistributedArray,  
        MappingParameters & params =Overture::nullMappingParameters())
```

**Description:** Map a grid of points.

This version of map assumes that the input array is of the form of a grid of points:

```
if domainDimension==1 then r can be of the form  
  r(a1:a2,0:d-1)  
  r(a1:a2,0:0,0:d-1)  
  r(a1:a2,0:0,0:0,0:d-1)  
if domainDimension==2 then r can be of the form  
  r(a1:a2,b1:b2,0:d-1)  
  r(a1:a2,b1:b2,0:0,0:d-1)  
if domainDimension==3 then r can be of the form  
  r(a1:a2,b1:b2,c1:c2,0:d-1)
```

The output is in a similar form

**r (input) :** evaluate the mapping at these points, where

**x (output) :** if x has enough space, then compute the mapping.

**xr (output) :** if xr has enough space, then compute the derivatives of the mapping.

**params (input/output) :** holds parameters for the mapping.

### 2.3.26 mappingHasChanged

```
int  
mappingHasChanged()
```

**Access:** protected

**Description:** Call this function when the mapping has changed

### 2.3.27 gridIsValid

```
bool  
gridIsValid() const
```

**Description:** Return true if remakeGrid=false

### 2.3.28 setGridIsValid

```
void  
setGridIsValid()
```

**Description:** Indicate that the grid is valid.

### 2.3.29 periodicShift

```
void  
periodicShift( realArray & r, const Index & I )
```

**Description:** Shift r into the interval [0.,1] if the mapping is periodic (derivative or function)

## 2.4 project

```
int  
project( realArray & x,  
          MappingProjectionParameters & mpParams )
```

**Purpose:** Project the points  $x(i,0:2)$  onto the Mapping. This is normally used to project points onto a curve in 2D or surface in 3D (i.e. domainDimension=rangeDimension-1, aka a hyperspace of co-dimension 1).

**x (input) :** project these points.

**mpParams (input) :** This class holds parameters used by the projection algorithm.

**Notes:** The inverse unit square coordinates will be held in the array mpParams.getRealArray(r). If you have a good guess for these values then you should supply this array.

If you want the derivatives you should dimension mpParams.getRealArray(xr) to be big enough and then they will be computed.

**Note:** If you want the normal (or tangent to a curve) you should dimension mpParams.getRealArray(normal) to be big enough. For curves (domainDimension==1) the normal is actually the tangent to the curve. Otherwise the normal will only make sense if the Mapping is a curve in 2D or a surface in 3D, i.e. domainDimension=rangeDimension-1.

#### **2.4.1 put**

```
int  
put( GenericDataBase & dir, const aString & name) const
```

**Description:** save this object to a sub-directory called "name"

#### **2.4.2 getGridIndexRange**

```
int  
getGridIndexRange(int side, int axis )
```

**Description:** Get the number of getGridIndexRange

#### **2.4.3 setGridIndexRange**

```
void  
setGridIndexRange(int side, int axis, int num )
```

**Description:** Set the getGridIndexRange

#### **2.4.4 getNumberOfGhostPoints**

```
int  
getNumberOfGhostPoints(int side, int axis )
```

**Description:** Get the number of ghost points to use on the grid associated with the mapping.

#### **2.4.5 setNumberOfGhostPoints**

```
void  
setNumberOfGhostPoints(int side, int axis, int numGhost )
```

**Description:** Set the number of ghost points to use on the grid associated with the mapping.

#### **2.4.6 reinitialize**

```
void  
reinitialize()
```

**Description:** Re-initialize a mapping that has changed (this will re-initialize the inverse)

#### **2.4.7 setName**

```
void  
setName( const mappingItemName item, const aString & itemName )
```

**Description:** Assign a name from enum mappingItemName:

```
mappingName : mapping name  
domainName : domain name  
rangeName :  
domainAxis1Name : names for coordinate axes in domain
```

```
domainAxis2Name :  
domainAxis3Name :  
rangeAxis1Name : names for coordinate axes in range  
rangeAxis2Name :  
rangeAxis3Name :  
item (input): assign this item.  
itemName (input) : name to give the item.
```

#### 2.4.8 setCoordinateEvaluationType

```
void  
setCoordinateEvaluationType( const coordinateSystem type, const int trueOrFalse )
```

**Description:**

#### 2.4.9 setTypeOfCoordinateSingularity

```
void  
setTypeOfCoordinateSingularity( const int side, const int axis,  
                                  const coordinateSingularity type )
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

#### 2.4.10 topologyMask

```
intArray &  
topologyMask()
```

**Description:** Return the mask that represents a partial periodicity, such as a C-grid.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

#### 2.4.11 getTopology

```
topologyEnum  
getTopology( const int side, const int axis ) const
```

**Description:** Return the topology. This is primarily used to represent C-grids.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

#### 2.4.12 setTopology

```
void  
setTopology( const int side, const int axis, const topologyEnum topo )
```

**Description:** Specify the topology. This is primarily used to represent C-grids.

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis < domainDimension*.

#### 2.4.13 setDomainDimension

```
void  
setDomainDimension( const int domainDimension0 )
```

Description:

#### 2.4.14 setRangeDimension

```
void  
setRangeDimension( const int rangeDimension0 )
```

Description:

#### 2.4.15 setBasicInverseOption

```
void  
setBasicInverseOption( const basicInverseOptions option )
```

Description:

#### 2.4.16 setBoundaryCondition

```
void  
setBoundaryCondition( const int side, const int axis, const int bc0 )
```

Description:

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.17 setShare

```
void  
setShare( const int side, const int axis, const int share0 )
```

Description:

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.18 getShare

```
void  
setSignForJacobian( const real signForJac )
```

**Description:** Set the sign of the jacobian, 1 (right handed coordinate system) or -1 (left handed). This may only make sense for some mappings.

**signForJac (input) :** should be 1. or -1.

#### 2.4.19 setMappingCoordinateSystem

```
void  
setMappingCoordinateSystem( const mappingCoordinateSystem  
mappingCoordinateSystem1 )
```

Description:

#### 2.4.20 setIsPeriodic

```
void  
setIsPeriodic( const int axis, const periodicType isPeriodic0 )
```

**Description:**

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

**Notes:** This routine has some side effects. It will change the boundaryConditions to be consistent with the periodicity (if necessary).

#### 2.4.21 setGridDimensions

```
void  
setGridDimensions( const int axis, const int dim )
```

**Description:**

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.22 setInvertible

```
void  
setInvertible( const int invertible0 )
```

**Description:**

#### 2.4.23 setParameter(real)

```
void  
setParameter( const realParameter & param, const real & value )
```

**Description:** Set the value of a parameter used by the Mapping or the ApproximateGlobalInverse or the ExactLocalInverse.

**THEnonConvergenceValue** : value given to "r" value of the inverse when there is no convergence.  
This is currently equal to 10. and cannot be changed.

**THEnewtonToleranceFactor** : convergence tolerance is this times the machine epsilon. Default=100. ?

**THEnewtonDivergenceValue** : newton is deemed to have diverged if the r value is this much outside [0,1]. The default value is .1 and so Newton is deemed to have diverged when the r value is outside the range [-.1,1.1]

**THEnewtonL2Factor** : extra factor for finding the closest point to a curve or surface, default=.1. This factor allows a less strict convergence factor if the target point is far from the mapping. Decrease this value if you want a more accurate answer. You may also have to decrease this value for mappings that have poor parameterizations.

**THEboundingBoxExtensionFactor** : relative amount to increase the bounding box each direction. The bounding box can be increased in size to allow the inverse function to still converge for nearby points. The default value is .01. \*\*\*Actually\*\*\* only the bounding boxes for the highest leaves in the bounding box tree are extended by this factor. The bounding boxes for all other nodes (and the root) are just computed from the size of the bounding boxes of the two leaves of the node.

**THEstencilWalkBoundingBoxExtensionFactor** : The stencil walk routine that finds the closest point before inversion by Newton's method will only find the closest point if the point lies in a box that is equal to the bounding box extended by this factor in each direction. Default =2

#### 2.4.24 setParameter(int)

```
void  
setParameter( const intParameter & param, const int & value )
```

**Description:** Set the value of a parameter used by the Mapping or the ApproximateGlobalInverse or the ExactLocalInverse.

**THEfindBestGuess** : if true, always find the closest point, even if the point to be inverted is outside the bounding box. Default value is false.

#### 2.4.25 setPeriodVector

```
void  
setPeriodVector( const int axis, const int direction, const real periodVectorComponent )
```

For a mapping with `getIsPeriodic(direction)==derivativePeriodic` this routine sets the vector that determines the shift from the 'left' edge to the 'right' edge.

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < rangeDimension$  are the components of the vector

**direction (input) :** direction =0,1,...,domainDimension

#### 2.4.26 setPartition

```
void  
setPartition( Partitioning_Type & partition_ )
```

**Description:** Supply a parallel partition to use when building the "grid" array

#### 2.4.27 setDomainSpace

```
void  
setDomainSpace( const mappingSpace domainSpace0 )
```

**Description:**

#### 2.4.28 setRangeSpace

```
void  
setRangeSpace( const mappingSpace rangeSpace0 )
```

**Description:**

#### 2.4.29 setDomainCoordinateSystem

```
void  
setDomainCoordinateSystem( const coordinateSystem domainCoordinateSystem0 )
```

**Description:**

#### 2.4.30 setRangeCoordinateSystem

```
void  
setRangeCoordinateSystem( const coordinateSystem rangeCoordinateSystem0 )
```

**Description:**

#### 2.4.31 setDomainBound

```
void  
setDomainBound( const int side, const int axis, const Bound domainBound0 )
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.32 setRangeBound

```
void  
setRangeBound( const int side, const int axis, const Bound rangeBound0 )
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.33 setDomainCoordinateSystemBound

```
void  
setDomainCoordinateSystemBound(const int side,  
                               const int axis,  
                               const Bound domainCoordinateSystemBound0 )
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.34 setRangeCoordinateSystemBound

```
void  
setRangeCoordinateSystemBound(const int side,  
                            const int axis,  
                            const Bound rangeCoordinateSystemBound0 )
```

**Description:**

**side, axis (input):** indicates the side of the mapping, side=(0,1) (or side=(Start,End)) and axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with  $axis < domainDimension$ .

#### 2.4.35 useRobustInverse

```
void  
useRobustInverse(const bool trueOrFalse =TRUE)
```

**Description:** Use the robust form of the inverse.

#### **2.4.36 usesDistributedInverse**

```
bool  
usesDistributedInverse() const
```

**Description:** Indicate whether in parallel this Mapping uses the distributed array, grid, for the inverse or whether the Mapping uses the serial array, gridSerial, (duplicated across all processors) for the inverse. If true, then the inverseMap function is a distributed (i.e. collective) parallel operation.

Normally only Mapping's with a small amount of data would duplicate the grid across all processors.

**Return value:** If true, then the inverseMap function is a distributed (collective) operation. The Mapping uses the distributed parallel array, grid, for the inverse. In this case, the array gridSerial points to the local array with ghost boundaries of grid.

If false then the inverseMap function is a serial (non-collective) operation with no communication. The Mapping uses the serial array, gridSerial, for the inverse and this grid is duplicated across all processors. In this case the array, grid, will remain empty.

#### **2.4.37 usesDistributedMap**

```
bool  
usesDistributedMap() const
```

**Description:** Indicate whether in parallel the map function is a distributed (i.e. collective) parallel operation.

Most Mapping's the map function is NOT distributed. The DataPointMapping, however, will normally use a distributed map function since the data-points that define the mapping are usually stored in a distributed array.

**Return value:** If true, then the map function is a distributed (collective) operation. If false, then the map function can be evaluated with no communication.

#### **2.4.38 sizeOf**

```
real  
sizeOf(FILE *file = NULL) const
```

**Description:** Return size of this object

#### **2.4.39 update**

```
bool  
updateWithCommand(MappingInformation &mapInfo, const aString & command)
```

**Description:** Update one of the parameters common to all Mappings. This function is usually called by the update function for the derived class.

**returns :** true if the command was understood, false otherwise

#### **2.4.40 update**

```
int  
update( MappingInformation & mapInfo )
```

**Description:** Update parameters common to all Mappings. This function is usually called by the update function for the derived class.

#### 2.4.41 interactiveUpdate

```
int  
interactiveUpdate( GenericGraphicsInterface & gi )
```

**Description:** Update Mapping parameters. This virtual function will call the update function for the derived Mapping. Use this function if you don't need to pass other Mapping information.

**gi (input) :** use this graphics interface.

#### 2.4.42 Periodic Mappings

The possible values returned by the function `getIsPeriodic` or passed to the function `setIsPeriodic` are given by the enumerator `periodicType`:

```
enum periodicType  
{  
    notPeriodic,  
    derivativePeriodic,    // Derivative is periodic but not the function  
    functionPeriodic       // Function is periodic  
};
```

### 2.5 Member function map

Here is an example of an implementation of the `map` member function. The mapping function is implemented so that it can evaluate the mapping for an array of points.

```
const int axis1 = 0;  
const int axis2 = 1;  
  
void map( realArray & r, realArray & x, realArray & xr = nullArray, MappingParams & params = nullParams)  
{  
    Index I = getIndex( r,x,xr,base,bound,computeMap,computeMapDerivative );  
  
    if( computeMap )  
    {  
        x(I,axis1)=2.*r(I,axis1)+xa;  
        x(I,axis2)=2.*r(I,axis2)+ya;  
    }  
    if( computeMapDerivative )  
    {  
        xr(I,axis1,axis1)=2.;  
        xr(I,axis1,axis2)=0.;  
        xr(I,axis2,axis1)=0.;  
        xr(I,axis2,axis2)=2.;  
    }  
}
```

The function `getIndex` returns an A++ index object that can be used when evaluating the mapping. Alternatively the variables `base` and `bound` can be used; note that `I.getBase(axis1)=base` and `I.getBound(axis1)=bound`. `getIndex` is described later in this section.

The default argument for the `xr` array is `nullArray` which is a static member function of the Mapping Class.

The argument `params` is not used in this example. It is used, for example, to indicate whether the derivatives should be returned in a different coordinate system, such as sphericalPolar. The default argument for `params` is `nullparms` which is a static member function of the `MappingParams` Class.

## 2.6 Member function getIndex

The function `getIndex` returns an `Index` object that can be used for A++ operations. The function `getIndex` also assigns values to the variables `base`, `bound`, `computeMap` and `computeMapDerivative`. Note that `base` and `bound` are consistent with the base and bound of the `Index` object returned by `getIndex`. The base and bound and `Index` object are determined by the first dimension of `r`. For example if `r` is dimensioned `r(0:9,3)` then `base=0`, and `bound=9` and `getIndex(...)=Index(base,bound-base+1)`. The variable `computeMap` is set to TRUE if dimensions of the array `x` can hold the index object. The variable `computeMapDerivative` is set to TRUE if dimensions of the array `xr` can hold the index object.

Thus, calling `map` with a null array (such as `Mapping::nullArray;`) in the place of `x` (or `xr`) will cause the mapping function not to evaluate `x` (or `xr`).

## 2.7 Member functions inverseMap and basicInverse

These member functions evaluate the inverse of the mapping for an array of points. The derivatives of the inverse mapping can also be obtained. By default an inverse is defined for all mappings whose `domainDimension ≤ rangeDimension`. This inverse uses Newton's method to invert the mapping. If the mapping can be inverted more quickly using another method then you can supply your own inverse.

- **inverseMap** : This is the primary function to call if you want to invert the mapping. This function will call `basicInverse` if it has been supplied. If the mapping is a transformation from parameter space to cartesian space (such as a mapping that defines a grid) then one normally should NOT supply this function but instead supply the `basicInverse` function. The reason for this is that the `inverseMap` must in general be able to invert the mapping when space is periodic. However, if the mapping is a transformation from parameter space to parameter space (such as a stretching transformation) or a transformation from cartesian space to cartesian space (such as a rotation) then one can supply the `inverseMap` function.
- **basicInverse** : This routine should be supplied if the mapping can be inverted quickly and the mapping is a transformation from parameter space to cartesian space. The `basicInverse` function does not need to take into account the fact that space may be periodic. To indicate that a `basicInverse` has been supplied you should use `setBasicInverseOption(canInvert)`.

The `inverseMap` function is automatically defined for all mappings whose `domainDimension ≤ rangeDimension`. When `domainDimension < rangeDimension` (for example, a curve in 2D) the inverse is defined as the closest point in the least squares sense ( $L_2$  norm).

If the mapping is a transformation from parameter space to cartesian space and the mapping can be inverted with an analytic formula then you should write the `basicInverse` member function. Here is an example

```
const int axis1 = 0;
const int axis2 = 1;
//=====
// Here is the basic Inverse (this is an inverse that does not know how
// to deal with space being periodic)
//=====
void SquareMapping::basicInverse( const realArray & x, realArray & r, realArray & rx )
{
    Index I = getIndex( x,r,rx,base,bound,computeMap,computeMapDerivative );

    if( computeMap )
    {
        r(I,axis1)=(x(I,axis1)-xa)/(xb-xa);
        r(I,axis2)=(x(I,axis2)-ya)/(yb-ya);
    }
    if( computeMapDerivative )
```

```

    {
        rx(I, axis1, axis1)=1./(xb-xa);
        rx(I, axis1, axis2)=0.;
        rx(I, axis2, axis1)=0.;
        rx(I, axis2, axis2)=1./(yb-ya);
    }
}

```

If the mapping is a transformation from parameter space to parameter space or from cartesian space to cartesian space and the mapping can be inverted easily then you should write an `inverseMap` function. Here is an example from the `MatrixMapping` class:

```

void MatrixMapping::  

inverseMap( const realArray & x, realArray & r, realArray & rx, MappingParameters & params )  

{  

    Index I = getIndex( x,r,rx,base,bound,computeMap,computeMapDerivative );  

    if( (Mapping::debug/64) % 2 ==1 )  

        cout << "MatrixMapping::inverseMap - params.isNull =" << params.isNull << endl;  

    if( computeMap )  

        for( int i=axis1; i<domainDimension; i++ )  

        {  

            r(I,i)=matrixInverse(i,3);           // holds shift  

            for( int j=axis1; j<rangeDimension; j++ )  

            {  

                r(I,i)=r(I,i)+matrixInverse(i,j)*x(I,j);  

            }  

        }  

    if( computeMapDerivative )  

        for( int i=axis1; i<domainDimension; i++ )  

        {  

            for( int j=axis1; j<rangeDimension; j++ )  

            {  

                rx(I,i,j)=matrixInverse(i,j);  

            }  

        }  

}

```

### 2.7.1 Member Functions `getName` , `setName`

These functions get or set the name for any of the items defined in the enum `mappingItemName`:

```

item = mappingClassName
    mappingName
    domainName
    rangeName
    domainAxis1Name
    domainAxis2Name
    domainAxis3Name
    rangeAxis1Name
    rangeAxis2Name
    rangeAxis3Name

```

These names can be used for plotting labels, for example. For example

```

StretchMapping stretch;                                // create a mapping
stretch.setName( mappingName, "myStretchMapping" );   // assign the mapping name

```

```

...
cout << " Mapping name = " << stretch.getName( Mapping::mappingName ) << endl;

```

## 2.8 Coordinate singularities

The `getTypeOfCoordinateSingularity` function can be used to determine if a given side of a mapping has a singularity. The possible types of singularities are

```

enum coordinateSingularity
{
    noCoordinateSingularity,    // no coordinate singularity
    polarSingularity           // grid lines go to a point along the side
};

```

A `polarSingularity` means that the grids lines converge to a point. For example, the standard representation for a sphere would have a `polarSingularity` on the two sides corresponding to  $\phi = 0$  and  $\phi = \pi$ .

Information about singularities is used by the `inverseMap`.

## 2.9 Coordinate systems and coordinateEvaluationType

Some mappings will have the capability to return the mapping derivatives in different forms, corresponding to different coordinate systems. Use the `setCoordinateEvaluationType` function to indicate that a mapping can return the derivatives in the specified form. These alternative forms of the derivatives can be used by a grid generator to remove coordinate singularities. Here is an example taken from the `SphereMapping` Class:

```

void SphereMapping::
map( const realArray & r, realArray & x, realArray & xr, MappingParameters & params )
{
    Index I = getIndex( r,x,xr,base,bound,computeMap,computeMapDerivative );

    int i;
    switch (params.coordinateType)
    {
        case cartesian: // mapping returned in cartesian form

            if( computeMap )
            {
                x(I,axis1)=radius(r(I,axis3))*cos(twoPi*r(I,axis2))*sin(Pi*r(I,axis1))+x0;
                x(I,axis2)=radius(r(I,axis3))*sin(twoPi*r(I,axis2))*sin(Pi*r(I,axis1))+y0;
                x(I,axis3)=radius(r(I,axis3))*cos(Pi*r(I,axis1))+z0;
            }
            if( computeMapDerivative )
            {
                xr(I,axis1,axis1)=radius(r(I,axis3))*cos(twoPi*r(I,axis2))*Pi*cos(Pi*r(I,axis1));
                ...
            }
            break;

        case spherical: // Mapping returned in spherical form : (phi,theta,r)
                        // derivatives: ( d/d(phi), (1/sin(phi))d/d(theta), d/d(r) )

            if( computeMap )
            {
                x(I,axis1)=radius(r(I,axis3))*cos(twoPi*r(I,axis2))*sin(Pi*r(I,axis1))+x0;

```

```

        x(I,axis2)=radius(r(I,axis3))*sin(twoPi*r(I,axis2))*sin(Pi*r(I,axis1))+y0;
        x(I,axis3)=radius(r(I,axis3))*cos(Pi*r(I,axis1))+z0;
    }
    if( computeMapDerivative )
    {
        xr(I,axis1,axis1)=radius(r(I,axis3))*cos(twoPi*r(I,axis2))*Pi*cos(Pi*r(I,axis1));
        ...
    }
    break;
default:
    cerr << "Sphere::map: ERROR not implemented for coordinateType = "
        << params.coordinateType << endl;
    exit(1);
}
}

```

## 2.10 Class MappingParams

Additional parameters are passed to the `map` and `inverseMap` functions by an object of the class `MappingParams`.

### 2.10.1 Data Members

<code>intisNull</code>	True if parameters have not been set =0,1,2,3
<code>int periodicityOfSpace</code>	vector(s) for periodicity
<code>realArray periodicityVector</code>	work space
<code>MappingWorkSpace workSpace</code>	TRUE by default
<code>int computeGlobalInverse</code>	evaluate mapping in this coordinate system
<code>coordinateSystem coordinateType</code>	pointer
<code>ApproximateGlobalInverse *approximateGlobalInverse</code>	pointer
<code>ExactLocalInverse *exactLocalInverse</code>	pointer

If space is periodic, then the parameters `periodicityOfSpace` and `periodicityVector` must be set in calls to `inverseMap`. Here is an example:

```

#include "Mapping.h"
#include "Square.h"

void main()
{
    realArray r1(10,2), x1(10,2), xr1(10,2,2);
    realArray r2(10,2), x2(10,2), rx2(10,2,2);

    SquareMapping square();

    MappingParameters periodicParams;
    // here is where we set the periodicity of Space, this should be consistent
    // with the periodicity of ALL mappings
    periodicParams.periodicityOfSpace=1;
    periodicParams.periodicityVector(axis1,axis1)=2.; // set vector to (2,0)
    periodicParams.periodicityVector(axis2,axis1)=0.;

    cout << "=====Periodic in Space=====" << endl;
    cout << " ---Call square map with an array of values:" << endl;
    for( i=0; i<10; i++ )

```

```

{
    r1(i,axis1)=i/9.;
    r1(i,axis2)=i/9.;
}
square.map( r1,x1,xr1 ); // get x1 and xr1 at an array of points
for( i=0; i<10; i++ )
    printf(" Square: r= (%6.3f,%6.3f) x = (%7.4f,%7.4f)\n",
           r1(i,axis1),r1(i,axis2),x1(i,axis1),x1(i,axis2));

cout << " ---Call square inverseMap with an array of values:" << endl;
for( i=0; i<10; i++ )
{
    x2(i,axis1)=1.5*x1(i,axis1);
    x2(i,axis2)=x1(i,axis2);
}
r2=1.; // initial guess
square.inverseMap( x2,r2,rx2,periodicParams );
for( i=0; i<10; i++ )
    printf(" Square: x= (%6.3f,%6.3f) r = (%7.4f,%7.4f)\n",
           x2(i,axis1),x2(i,axis2),r2(i,axis1),r2(i,axis2));
}

}

```

The `inverseMap` member function of the `ComposeMapping` class will use the `computeGlobalInverse` parameter.

## 2.11 Class ApproximateGlobalInverse

This class is used to define an inverse/approximate global inverse of a mapping. The approximate global inverse computes an approximate inverse to the mapping. This approximate inverse should be good enough so that a Newton iteration will converge.

Each mapping contains a pointer to an `ApproximateGlobalInverse`, called `approximateGlobalInverse`. This `ApproximateGlobalInverse` is used by the `inverseMap` member function.

We now describe the default implementation for the `ApproximateGlobalInverse`. The default approximate global inverse has a discrete grid that contains values of the mapping. The inverse finds the closest point on this grid. The number of points on the grid can be set by the `Mapping` member function `setGridDimensions` or the actual grid to be used can be specified with the `ApproximateGlobalInverse` member function `setGrid`.

### 2.11.1 constructor

`ApproximateGlobalInverse( Mapping & map0 )`

**Description:** Build an approximate inverse to go with a given mapping.

### 2.11.2 setGrid

```

// void
//=====================================================================
// /Description: // Give a grid that can be used for global search routines // The grid is assumed to
// have been assigned with values of the // mapping. The grid is assumed to be always declared as a
// // four-dimensional A++ array, grid(axis1,axis2,axis3,rangeDimension). //
// /grid0 (input) : use this grid.
// /gridIndexRange : index bounds for the sides of the grids //

```

### **2.11.3 getGrid**

```
const RealArray &
getGrid() const
```

**Description:** return the grid used for the inverse

### **2.11.4 getBoundingBox**

```
const RealArray &
getBoundingBox() const
```

**Description:** Return the bounding box for the entire mapping.

### **2.11.5 getBoundingBoxTree**

```
const BoundingBox &
getBoundingBoxTree(int side, int axis) const
```

**Description:** Return the bounding box tree for a given boundary of the mapping.

### **2.11.6 getParameter**

```
real
getParameter( const realParameter & param ) const
```

**Description:** Return the value of a parameter.

**param (input)** : One of `MappingParameters::THEboundingBoxExtensionFactor` or  
`MappingParameters::THEstencilWalkBoundingBoxExtensionFactor`.

### **2.11.7 getParameter**

```
int
getParameter( const intParameter & param ) const
```

**Description:** Return the value of a parameter.

**param (input)** : One of `MappingParameters::THEfindBestGuess`

### **2.11.8 setParameter**

```
void
setParameter( const realParameter & param, const real & value )
```

**Description:** Set the value of a parameter.

**param (input)** : One of `MappingParameters::THEboundingBoxExtensionFactor` or  
`MappingParameters::THEstencilWalkBoundingBoxExtensionFactor`.

**value (input)** : value for the parameter.

### **2.11.9 setParameter(int)**

```
void  
setParameter( const intParameter & param, const int & value )
```

**Description:** Set the value of a parameter.

**param (input) :** One of `MappingParameters::THEboundingBoxExtensionFactor` or `MappingParameters::THEstencilWalkBoundingBoxExtensionFactor`.

**value (input) :** value for the parameter.

### **2.11.10 useRobustInverse**

```
void  
useRobustInverse(const bool trueOrFalse =TRUE)
```

**Description:** If TRUE use the more robust approximate inverse that will work with highly stretched grids where the closest grid point x, to a given point may be many cells away from the cell containing the point x.

### **2.11.11 usingRobustInverse**

```
bool  
usingRobustInverse() const
```

**Description:** Return TRUE if using the more robust approximate inverse that will work with highly stretched grids where the closest grid point x, to a given point may be many cells away from the cell containing the point x.

### **2.11.12 sizeOf**

```
real  
sizeOf(FILE *file = NULL) const
```

**Description:** Return size of this object

### **2.11.13 get**

```
int  
get( const GenericDataBase & dir, const aString & name)
```

**Description:** Get this object from a sub-directory called "name"

### **2.11.14 put**

```
int  
put( GenericDataBase & dir, const aString & name) const
```

**Description:** save this object to a sub-directory called "name"

### 2.11.15 inverse

```
void  
inverse(const RealArray & x,  
        RealArray & r,  
        RealArray & rx,  
        MappingWorkSpace & workSpace,  
        MappingParameters & params )
```

**Purpose:** Find an approximate inverse of the mapping; this approximate inverse should be good enough so that Newton will converge

**Method:** 1. If space is periodic (e.g. if the grids all live on a background square which has one or more periodic edges) then we need to worry about values of x that are outside the basic periodic region. These points may have periodic images that lie inside the periodic region. We thus add new points to the list that are the periodic images that lie inside the basic square. \*\*\*NOTE\*\*\* space periodic rarely occurs and probably hasn't been tested enough.

```
-----  
| |  
| x | X  
| periodic | initial point to invert  
| image |  
| |  
| |  
| |  
-----
```

2. For all points to invert, find the closest point on the reference grid that goes with the mapping. This grid is usually just the grid that is used when plotting the mapping. This step is performed by the function `findNearestGridPoint`

**Notes:** The results produced by this routine are saved in the object `workSpace`.

**workSpace.x0 (output)** : list of points to invert with possible extra points if space is periodic.

**workSpace.r0 (output)** : unit square coordinates of the closest point.

**workSpace.I0 (output)** : Index object that marks the active points in `x0` and `r0`.

**workSpace.index0 (output)** : indirect addressing array that points back to the original `r` array; used when there are extra points added for periodicity in space.

**workSpace.index0IsSequential (output)** : if TRUE then space is periodic and the `index0` indirect addressing array should be used when storing results back in the user arrays `r` and `rx`.

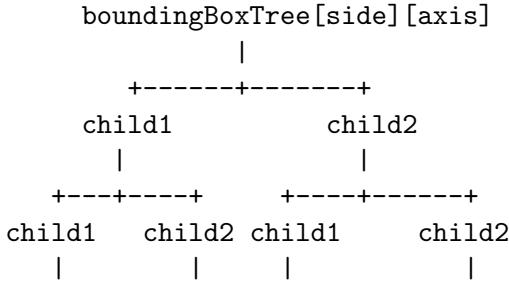
### 2.11.16 initializeBoundingBoxTrees

```
void  
initializeBoundingBoxTrees()
```

**Description:** Assign the binary tree's of Bounding Boxes

For `domainDimension==1` bounding boxes are made for the whole mapping (curve in 1,2 or 3d) For `domainDimension>rangeDimension` boxes are made for the whole mapping (3D surface)

For domainDimension<sub>j</sub>1 a binary tree is created for each side Any box that has too many grid points in it is subdivided into two



Each box contains:

```

domainBound(2, domainDimension) : index bounds for box
rangeBound(2, rangeDimension)   : bounds of box in physical space

```

Note that these Bounding Box trees will be automatically (recursively) deleted when the destructor is called for boundingBoxTree[2][3]

### 2.11.17 findNearestGridPoint

```

void
findNearestGridPoint( const int base1, const int bound1, RealArray & x, RealArray & r )

```

**Description:** Find the nearest grid point by a ‘stencil walk’ and possibly a global search over the boundary  
For each point x(i,.), i=base1,...,bound1, find the index of the closest point on the boundary r(i,.).

1. For a 1D grid start at the initial guess and look to the left or to the right depending on whether the distance decreases to the left or right.
2. For a 2D grid first do a local search, use the index arrays to indicate which points in the square to check (not all points need be searched as they would have been done on the previous checks). If the local search ends on a boundary then do a global search of all boundary points, followed by another local search.
3. For a 3D grid proceed as in 2 but use different index arrays

### 2.11.18 binarySearchOverBoundary

```

void
binarySearchOverBoundary( real x[3],
                           real & minimumDistance,
                           int iv[3],
                           int side = -1,
                           int axis = -1)

```

**Description:** Binary Search over the boundary.

For point x find the index of the closest point on the boundary iv iv should be given a value on input, current closest point

For curves or surfaces we search the entire surface for the closest point

**x (input)** : point to search for.

**minimumDistance (input/output)** : NOTE that this "distance" is the SQUARE of the L2 norm. On input : find a point with minimum distance less than this value. On output, if minimumDistance is less than the input value then this will be the new minimum distance and iv will hold the point on the boundary that is closest.

**iv (output)** : Closest boundary ONLY IF a point is found that is closer than the input value of minimumDistance.

**side,axis (input)** : optionally specify to only search this face.

### 2.11.19 findNearestCell

**int**

```
findNearestCell(real x[3],  
                int iv[3],  
                real & minimumDistance )
```

**Description:** Find the nearest grid cell by a 'stencil walk' . This search technique may be needed for highly stretched grids since the closest grid point to x may be many cells away.

**iv (input/output)** : on input, the initial guess for the closest cell. On output the nearest cell (locally, may end on a boundary).

**minimumDistance (output):** return 0 if x is inside the cell iv. Otherwise the minimum distance is NOT computed by this routine (for efficiency) since the algorithm does not really require it.

**Return values:** 0 point x is inside the cell.

1 stencil walk has reached a boundary and the point is apparently not inside the cell.

### 2.11.20 countCrossingsWithPolygon

**void**

```
countCrossingsWithPolygon(const RealArray & x,  
                           IntegerArray & crossings,  
                           const int & side_ =Start,  
                           const int & axis_ =axis1,  
                           RealArray & xCross = Overture::nullRealDistributedArray(),  
                           const IntegerArray & mask = Overture::nullIntArray(),  
                           const unsigned int & maskBit = UINT_MAX,  
                           const int & maskRatio1 =1,  
                           const int & maskRatio2 =1,  
                           const int & maskRatio3 =1)
```

**Description:** Count the number of times that the ray starting from position xv=(x,y) and extending to y=+ infinity, crosses the polygon approximation to the curve (domainDimension=1) or the triangulated approximation to the face of the mapping (domainDimension==3).

**x(I,0: r-1) (input)**: set of points to check

**crossings(I) (input/output)**: number of crossings for each point. \*\*NOTE\*\* this function will add on to the current values in this array, thus you should set this to zero on the first call, or subsequent calls, depending on your application.

**side,axis\_ (input):** For domainDimension $\neq 1$  these will indicate the side (domainDimension==2) or the face (domainDimension==3) to check. For domainDimension==1 these values are ignored.

**xCross (input/output) :** If this argument is supplied then we store each crossing point in the array as xCross(i,0:2r,cross) where cross=0,1,...,crossings(i)-1, r=rangeDimension. We save [x,y,i1,i2] if rangeDimension==2 and [x,y,z,i1,i2,i3] if rangeDimension==3. (i1,i2,i3) denotes the lower left corner of the cell that holds the intersection.

**mask (input):** optional arg that is used to mask out certain parts of the boundary. If this arg is given then ALL corners of a cell must have "mask(i1,i2,i3) & maskBit" in order that a ray crossing that cell to count as an actual crossing. In other words the valid points on the boundary are marked with "mask(i1,i2,i3) & maskBit".

**maskBit (input) :** by default the mask bit is `UINT_MAX == 2m - 1` (all bits on) so that invalid points would have `mask(i1,i2,i3)==0`

**maskRatio1, maskratio2,maskRatio3 (input) :** parameters from multigrid. These are the ratios of the current grid spacing to the finest grid spacing. (assuming that the grid associated with this mapping is the finest grid!).

**Return value :** number of times the ray crosses the polygon. For a closed curve there will be an odd number of crossings if the point is inside the polygon and an even number of crossings if the point is outside the polygon.

**NOTE:** If a point lies exactly on a vertical line segment then this routine will give zero crossings for this segment (it may cross other segments in which case the crossing count may be non-zero)

## 2.12 Class ExactLocalInverse

This class defines an exact inverse for a mapping, given a good initial guess.

Each mapping contains a pointer to an `ExactLocalInverse`, called `exactLocalInverse`. This `ExactLocalInverse` is used by the `inverseMap` member function.

The default `ExactLocalInverse` uses the Newton algorithm to invert the mapping (if the mapping is invertible) or uses Newton to find the closest point ( $L_2$ -norm) between a point and a surface or curve.

## 3 Inverting the Mapping by Newton's Method

### 3.1 The case of a square Jacobian

When the `domainDimension` equals the `rangeDimension` we use a fairly standard Newton's method, with some damping if the corrections are too large. Special considerations are required if the Jacobian (The Newton matrix) is singular; this could occur at a polar singularity, for example.

### 3.2 The case of a non-square Jacobian

When the `domainDimension` is not equal to the `rangeDimension`, such as a curve or surface, then we must define what is meant by inverting the Mapping. This amounts to finding some 'closest' point of the Mapping.

Denote the transformation defining the Mapping by

$$\mathbf{x} = \mathbf{S}(r_1, r_2)$$

where, to be specific, we consider the case of a surface in 3D.

### 3.2.1 Method 1 : Least Squares

Given a point  $\mathbf{x}$  not on the surface, the equation  $\mathbf{x} = \mathbf{S}(\mathbf{r})$  will have no solution. We need to define a best guess for the solution. By Taylor series

$$\mathbf{x} = \mathbf{S}(\mathbf{r}^{n-1}) + \nabla_{\mathbf{r}} \mathbf{S}(\mathbf{r}^n - \mathbf{r}^{n-1}) + \dots$$

Linearizing the equation (Netwon's method) gives the over-determined system

$$\begin{aligned}\nabla_{\mathbf{r}} \mathbf{S}(\mathbf{r}^n - \mathbf{r}^{n-1}) &= \mathbf{x} - \mathbf{S}(\mathbf{r}^{n-1}) \\ \text{or } A\Delta\mathbf{r} &= \Delta\mathbf{x}\end{aligned}$$

of 3 equations for the two unknowns in  $\Delta\mathbf{r}$ . We can ‘solve’ this over-determined system by least squares

$$A^T A \Delta\mathbf{r} = A^T \Delta\mathbf{x}$$

or equivalently using the QR algorithm

$$R\mathbf{r} = Q^T \Delta\mathbf{x}$$

to obtain the new guess  $\mathbf{r}^n$ . On convergence the residual  $\Delta\mathbf{x}$  will be orthogonal to the tangent vectors on the surface,  $A^T \Delta\mathbf{x} = 0$ , and thus the residual will be in the direction of the surface normal.

**Aside:** In the hyperbolic grid generation context there is another way to define the inverse. The problem is to find a point  $\mathbf{x}$  that is a given distance,  $d$ , from a point  $\mathbf{x}^0$  and lying on some plane  $\mathbf{n} \cdot (\mathbf{x} - \mathbf{x}^0) = 0$ . In this case we have a system of three equations for three unknowns,

$$\begin{aligned}\mathbf{x} &= \mathbf{S}(\mathbf{r}) \\ \mathbf{x} &= \mathbf{x}^0 + d(\mathbf{t}_1 \cos(\theta) + \mathbf{t}_2 \sin(\theta))\end{aligned}$$

Here  $\mathbf{t}_m$  are unit orthogonal tangent vectors on the plane and  $\theta$  is the extra unknown. This system may be faster to solve than the least squares approach (?)

### 3.2.2 Old way: minimize $l_2$ distance

Minimize the  $l_2$  distance (squared) between the point and the surface,

$$\min_{\mathbf{r}} g(\mathbf{r}) \quad \text{where } g = \|\mathbf{x} - \mathbf{S}(\mathbf{r})\|^2 = (\mathbf{x} - \mathbf{S})^T (\mathbf{x} - \mathbf{S})$$

To do this we solve  $\nabla_{\mathbf{r}} g = 0$  (which could also find the maximum distance),

$$\mathbf{h}(\mathbf{r}) = \nabla_{\mathbf{r}} g = -2\nabla_{\mathbf{r}} \mathbf{S}^T (\mathbf{x} - \mathbf{S}) = 0$$

i.e.

$$\sum_k \partial_{r_i} S_k (x_k - S_k) = 0 \quad \text{for } i = 0, 1$$

This equation  $\mathbf{h}(\mathbf{r}) = 0$  is solved by Newton's method,

$$\begin{aligned}\nabla_{\mathbf{r}} \mathbf{h}(\mathbf{r}^n - \mathbf{r}^{n-1}) &= -\mathbf{h}(\mathbf{r}^{n-1}) \\ \nabla_{\mathbf{r}} \mathbf{h} &= H(\mathbf{x} - \mathbf{S}) - \|\nabla_{\mathbf{r}} \mathbf{S}\|^2\end{aligned}$$

$$H_{ij} = \sum_k \partial_{r_i} \partial_{r_j} S_k (x_k - S_k) - (\partial_i S_k)^2$$

One disadvantage of this approach is that it requires the second derivative of the Mapping.

### 3.2.3 constructor

```
ExactLocalInverse( Mapping & map0 )
```

**Description:** Build an ExactLocalInverse from a Mapping.

### 3.2.4 getParameter

```
real  
getParameter( const realParameter & param ) const
```

**Description:** Return the value of a parameter.

**param (input)** : one of THEnonConvergenceValue, THEnewtonToleranceFactor, or  
THEnewtonDivergenceValue or newtonL2Factor from the enum MappingParameters.

### 3.2.5 setParameter

```
void  
setParameter( const realParameter & param, const real & value )
```

**Description:** Set the vaule of a parameter.

**param (input)** : one of THEnonConvergenceValue, THEnewtonToleranceFactor, or  
THEnewtonDivergenceValue or THEnewtonL2Factor from the enum MappingParameters.

**value (input)** : value to assign.

### 3.2.6 sizeOf

```
real  
sizeOf(FILE *file = NULL) const
```

**Description:** Return size of this object

### 3.2.7 reinitialize

```
void  
reinitialize()
```

**Description:** This will mark ExactLocalInverse as being in need of initialization. The actual call to initialize will occur when the inverse is actually used.

### 3.2.8 get

```
int  
get( const GenericDataBase & dir, const aString & name)
```

**Description:** Get this object from a sub-directory called "name"

### 3.2.9 put

```
int  
put( GenericDataBase & dir, const aString & name) const
```

**Description:** save this object to a sub-directory called "name"

### 3.2.10 initialize

```
void  
initialize()
```

**Description:** Initialize.

### 3.2.11 compressConvergedPoints

```
int  
compressConvergedPoints(Index & I,  
                         RealArray & x,  
                         RealArray & r,  
                         RealArray & ry,  
                         RealArray & det,  
                         IntegerArray & status,  
                         const RealArray & x1,  
                         RealArray & r1,  
                         RealArray & rx1,  
                         MappingWorkSpace & workSpace,  
                         const int computeGlobalInverse )
```

**Description:** Remove points that have converged or diverged so that we will only iterate on the smaller number of points that haven't converged,

### 3.2.12 inverse

```
void  
inverse(const RealArray & x1,  
        RealArray & r1,  
        RealArray & rx1,  
        MappingWorkSpace & workSpace,  
        const int computeGlobalInverse )
```

**Description:** Compute the inverse of the mapping using Newton's method. The initial guess must be good enough for Newton to converge

x1,r1,rx1 (input/output) :

workSpace (input) :

**computeGlobalInverse (input):** TRUE means that the approximateGlobal inverse routine was called previous to this call. In this case we look for information in the workSpace. FALSE means that the approximateGlobalInverse was not called before this call.

## 3.3 Registering Mappings and Reading Generic Mappings from the DataBase

In this section we describe how a mapping can be read from a database file and contructed even when the function constructing the mapping does not know the (derived) class to which the mapping belongs. For example, this situtaion occurs when a container class holds a pointer to a Mapping. The pointer is of type **Mapping\*** but the pointer may point to a derived class such as **SquareMapping**. Suppose the container class is saved to a database file with the function **Container::put**. When it is read back in again with **Container::get** the get function will not know how to "get" the mapping.

To solve this problem each mapping class has a member function **make** (a virtual member function of the base class) that look likes

```

Mapping *SquareMapping::make( const String & mappingClassName )
{ // Make a new mapping if the mappingClassName is the name of this Class
    Mapping *retval=0;
    if( mappingClassName==className )
        retval = new SquareMapping();
    return retval;
}

```

The function `make` creates a new mapping of it's own class provided that the String passed to `make` is the name of it's class.

The Mapping Class contains a static member that is a list of pointers to Mappings, `mappingList`. Each member of the list points to an instance of a different derived mapping Class. All possible Mapping Class's that may be read from the database should have a member in `mappingList`. Here, for example, is how to add members to the `mappingList`:

```

CircleMapping circle;
StretchMapping stretch;
...
Mapping::mappingList.add( &circle );
Mapping::mappingList.add( &stretch );

```

The `makeMapping` member function of the Mapping Class can be used to make a Mapping corresponding to a given class name. The `makeMapping` function takes as input the name of a class that it should try to make. For example, the argument to `makeMapping` may be the String `className=="SquareMapping"`. The `makeMapping` function goes through it's list of mappings, calling the `make` member function of each mapping, until it finds the mapping class that is able to make a "SquareMapping".

```

//=====
// Get a mapping from the database
// This routine looks through the list of mapping Class's
// that have been placed on the mappingList and tries to
// find one that knows how to make a mapping whose name
// is equal to the input argument className
//
// Input:
//   const String & className
//     : name of the mapping class to get from the database file
//   Dir dir
//     : directory on the database where the mapping is stored
//   const String & name
//     : the database name for the mapping
//
//=====
Mapping* Mapping::makeMapping( const String & className )
{ // Try to construct a mapping
    Mapping *retval = 0;
    for( Item *ptr=mappingList.start; ptr; ptr=ptr->next )
        if( retval = ptr->val->make( className ) ) break;
    return retval;
}

```

Here is the definition of a container class that calls `makeMapping` in order to construct a mapping. The container class has a pointer to a generic Mapping. There is no trouble saving the mapping to a database with the `put` member function. However when it reads the mapping back from the database it must be able to construct an instance of the appropriate derived class; this is done by `makeMapping`. Note that we assume that each Mapping class has a data member `String className` that holds the name of the class.

```

class Container
{

```

```

public:
    Mapping *mapPointer;

    ...

void get( const Dir & dir, const String & name )
{
    // Make a new directory unless name="."
    Dir subDir = name==".." ? dir : dir.findDir(name);

    // Look for the className of the Mapping:
    Dir mappingDir = subDir.findDir("containedMapping");
    String mappingClassName;
    mappingDir.get( mappingClassName,"className" );

    // Make an instance of the appropriate derived Mapping class
    mapPointer = Mapping::makeMapping( mappingClassName );
    getMap=TRUE;
    mapPointer->get( subDir,"containedMapping" );    // get the mapping

}

void put( const Dir & dir, const String & name )
{
    // destroy the directory if it exists
    if( !dir.locateDir(name).isNull() )
        dir.destroy(name, " R");
    Dir subDir = name==".." ? dir : dir.createDir(name);

    mapPointer->put( subDir,"containedMapping" );    // save the mapping

}

};


```

## 4 AnnulusMapping

This Mapping defines a circular or elliptical annulus in two or three space dimensions

$$\begin{aligned}\theta &= 2\pi(\theta_0 + r_1(\theta_1 - \theta_0)) \\ \mathbf{x}(r_1, r_2) &= (R_0 + r_2(R_1 - R_0))(\alpha \cos(\theta) + x_0, \sin(\theta) + y_0, z_0)\end{aligned}$$

By default the annulus is parameterized with a left-handed coordinate system. You can make the system right handed by choosing the `outerRadius` to be less than the `innerRadius`.

By default the annulus is two dimensional. To make a three dimensional annulus use the `setRangeDimension()` function or use the `setOrigin(x0,y0,z0)` function with a non-zero value of `z0`.

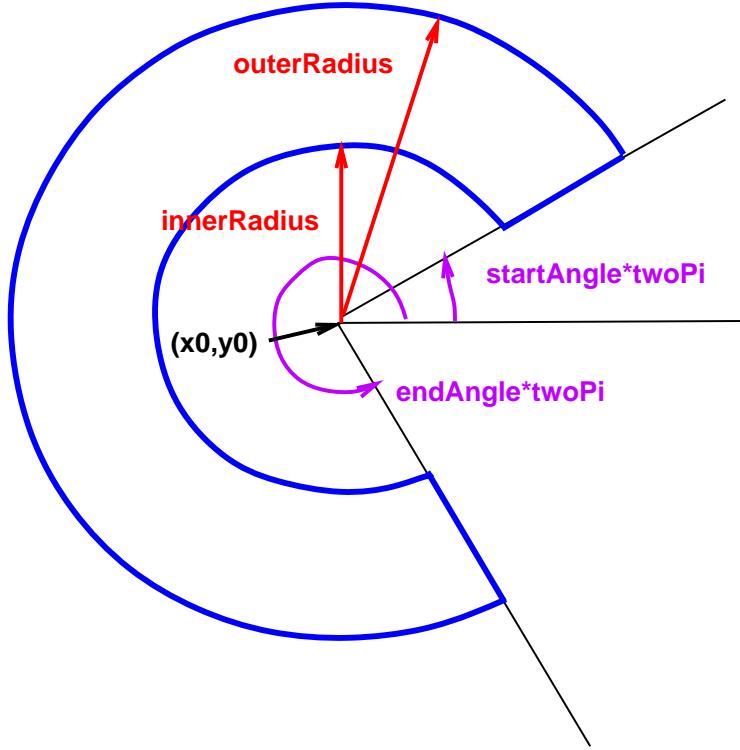


Figure 2: The AnnulusMapping defines an annulus

### 4.1 Constructor

```
AnnulusMapping(const real innerRadius_ =.5,
               const real outerRadius_ =1.,
               const real x0_ =0.,
               const real y0_ =0.,
               const real startAngle_ =0.,
               const real endAngle_ =1.,
               const real aOverB_ =1.)
```

**Purpose:** Create an annulus with a circular or elliptical boundary.

The annulus is defined by

$$\begin{aligned}x(r, s) &= x_0 + \text{aOverB } R(s) \cos(\theta(r)) \\y(r, s) &= y_0 + R(s) \sin(\theta(r)) \\R(s) &= \text{innerRadius} + s(\text{outerRadius} - \text{innerRadius}) \\\theta(r) &= 2\pi[\text{startAngle} + r(\text{endAngle} - \text{startAngle})]\end{aligned}$$

**innerRadius,outerRadius (input):** inner and outer radii.

**x0,y0 (input):** centre for the annulus.

**startAngle, endAngle (input):** The initial and final "angle" (in the range [0,1]).

**aOverB (input):** The ratio of the length of the horizontal-radius ("a") to the vertical-radius ("b") for an elliptical boundary. A value of aOverB=1 defines a circular boundary.

## 4.2 setRadii

**int**

```
setRadii(const real & innerRadius_ =.5,
         const real & outerRadius_ =1.,
         const real aOverB_ =1.)
```

**Purpose:** Define the radii of the annulus.

**innerRadius,outerRadius (input):** inner and outer radii of the annulus. There is NO restriction that `innerRadius < outerRadius`.

**aOverB (input):** The ratio of the length of the horizontal-radius ("a") to the vertical-radius ("b") for an elliptical boundary. A value of aOverB=1 defines a circular boundary.

## 4.3 setOrigin

**int**

```
setOrigin(const real & x0_ =0.,
          const real & y0_ =0.,
          const real & z0_ =0.)
```

**Purpose:** Set the centre of the annulus. Choosing a non-zero value for `z0` will cause the `rangeDimension` of the Mapping to become 3.

**x0,y0,z0 (input):** centre of the annulus.

## 4.4 setAngleBounds

**int**

```
setAngleBounds(const real & startAngle_ =0.,
               const real & endAngle_ =1.)
```

**Purpose:** Set the angular bounds on the annulus.

**startAngle, endAngle (input):** The initial and final "angle" (in the range [0,1]).

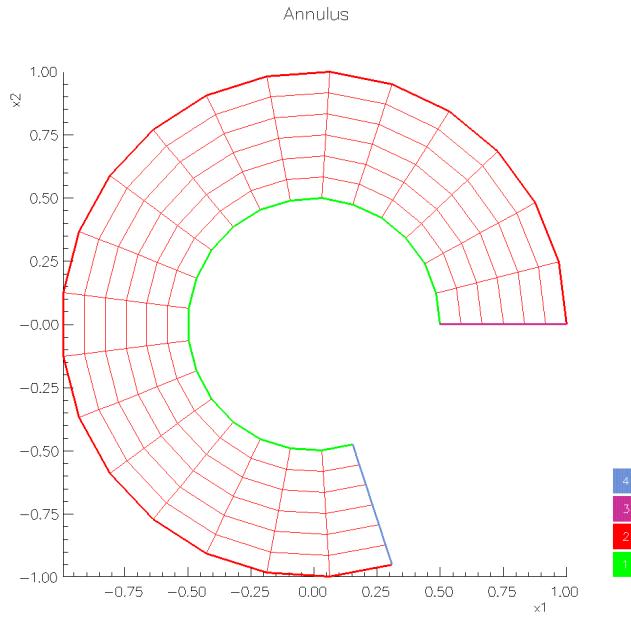


Figure 3: A mapping for a partial annulus.

## 5 AirfoilMapping: create some airfoil related grids or curves

### 5.1 NACA airfoils

The NACA 4 digit series airfoils (such as the NACA0012) are defined by

$$\begin{aligned} x_u(r) &= (r - y_t(r) \sin(\theta))c && \text{upper surface} \\ y_u(r) &= (y_c(r) + y_t(r) \cos(\theta))c && \text{upper surface} \\ x_l(r) &= (r + y_t(r) \sin(\theta))c && \text{lower surface} \\ y_l(r) &= (y_c(r) - y_t(r) \cos(\theta))c && \text{lower surface} \end{aligned}$$

where  $c$  is the chord length. The camber line,  $y_c$ , is defined by

$$\begin{aligned} y_c(r) &= c_{\max} \frac{1}{x_1^2} (2x_1 r - r^2) && \text{for } 0 \leq r \leq x_1 \\ y_c(r) &= c_{\max} \frac{1}{(1-x_1)^2} ((1-2x_1) + 2x_1 r - r^2) && \text{for } x_1 \leq r \leq 1 \\ x_1 &= \text{position of the maximum camber} \end{aligned}$$

and the thickness is defined by

$$y_t(r) = 5\delta(0.29690\sqrt{r} - 0.12600r - 0.35160r^2 + .28430r^3 - 0.10150r^4)$$

where

$$\delta = \text{thickness/chord}$$

The NACA[c][p][tc] airfoil is defined by:

- c** maximum camber/chord  $\times 100$  ( $c_{\max} \times 100$ ).

**p** position of maximum camber/chord  $\times 10$  ( $x_1 \times 10$ ).

**tc** thickness/Chord  $\times 100$  ( $\delta \times 100$ ).

Thus the NACA0012 has  $c_{\max} = 0$ ,  $x_1 = 0$  and  $\delta = .12$ .

## 5.2 Joukowsky Airfoil

The Joukowsky airfoil is defined by

$$\begin{aligned} z &= x + iy = w + \frac{1}{w} && z \text{ and } w \text{ are complex numbers} \\ w &= ae^{i\theta} + ide^{i\delta} \\ \theta &= 2\pi r_0 \end{aligned}$$

The parameters  $a, d, \delta$  in the definition have the following approximate properties,

**a** :  $<= 1$ , closer to 1 implies sharper trailing edge.

**d** : bigger d implies larger camber.

**$\delta$**  : bigger  $\delta$  implies greater asymmetry between leading and trailing edges.

## 5.3 Member function descriptions

### 5.3.1 Constructor

```
AirfoilMapping(const AirfoilTypes & airfoilType_,  
              const real xa = -1.5,  
              const real xb = 1.5,  
              const real ya = 0.,  
              const real yb = 2.)
```

**Description:** Create a mapping for an airfoil.

**Notes:** An airfoil mapping can be made from one of the following (enum AirfoilTypes)

**arc** : grid with a bump on the bottom that is an arc of a circle.

**sinusoid** : grid with a bump on the bottom that is a sinusoid.

**diamond** : grid with a bump on the bottom that is a diamond.

**naca** : a curve that is one of the NACA 4 digit airfoils.

**joukowsky** : a curve defining a Joukowsky airfoil.

**airfoilType\_ (input):** an airfoil type from the above choices.

**xa,xb,ya,yb (input) :** boundaries of the bounding box (not used for naca airfoils).

### 5.3.2 setBoxBounds

**int**

```
setBoxBounds(const real xa = -1.5,  
             const real xb = 1.5,  
             const real ya = 0.,  
             const real yb = 2.)
```

**Description:** set bounds on the rectangle that the airfoil sits in

**xa,xb,ya,yb (input) :** boundaries of the bounding box (not used for naca airfoils).

### 5.3.3 setParameters

```
int  
setParameters(const AirfoilTypes & airfoilType_,  
              const real & chord_ =1.,  
              const real & thicknessToChordRatio_ =.1,  
              const real & maximumCamber_ =0.,  
              const real & positionOfMaximumCamber_ =0.,  
              const real & trailingEdgeEpsilon_ =.02,  
              const real & sinusoidPower_ = 1.)
```

**Description:** Create a mapping for an airfoil.

**Notes:** An airfoil mapping can be made from one of the following (enum AirfoilTypes)

**arc** : grid with a bump on the bottom that is an arc of a circle.

**sinusoid** : grid with a bump on the bottom that is a sinusoid (or power of a sinusoid).

**diamond** : grid with a bump on the bottom that is a diamond.

**naca** : a curve that is one of the NACA 4 digit airfoils.

**joukowsky** : Joukowski airfoil. The other parameters in the argument list do not apply in this case. Use the `setJoukowskyParameters` function instead.

**airfoilType\_ (input):** an airfoil type from the above choices.

**chord\_ (input):** length of the chord.

**thicknessToChordRatio\_ (input):** thickness to chord ratio.

**maximumCamber\_ (input):** maximum camber

**positionOfMaximumCamber\_ (input):** position of maximum camber

**trailingEdgeEpsilon\_ (input) :** parameter for rounding the trailing edge.

### 5.3.4 setJoukowskyParameters

**int**

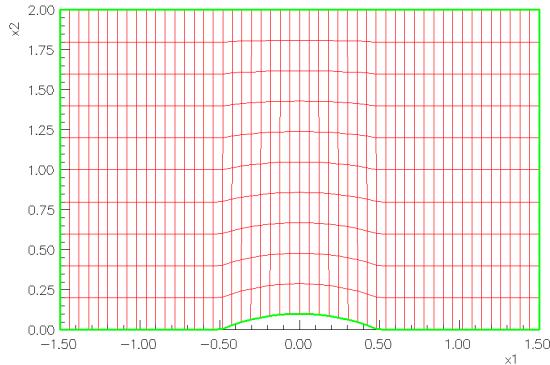
```
setJoukowskyParameters( const real & a, const real & d, const real & delta )
```

**Description:** Set parameters for the Joukowski airfoil.

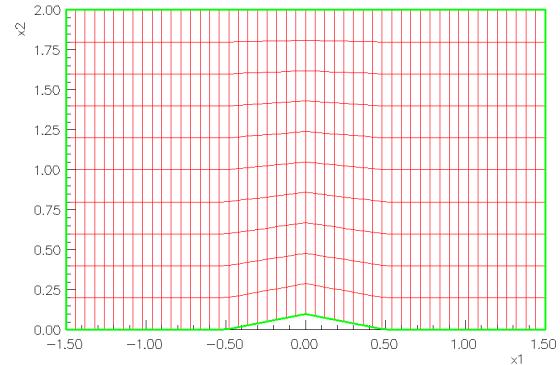
**a,d,delta** : see the documentation for a description of these.

## 5.4 Examples

airfoil

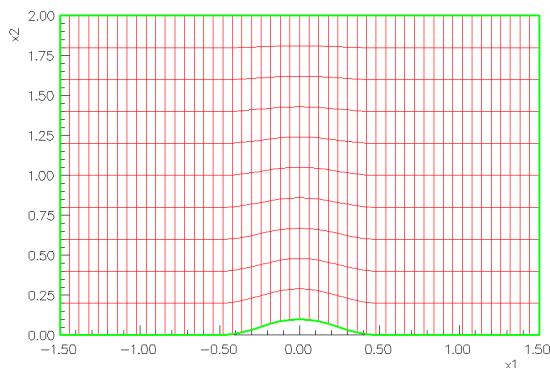


airfoil



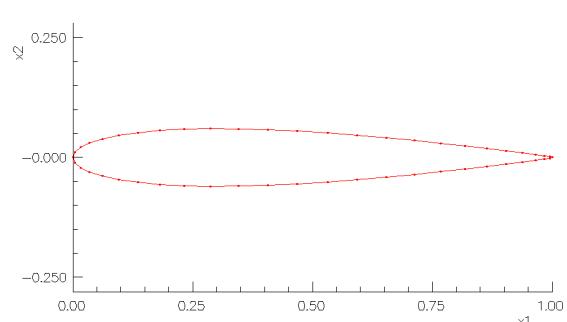
Airfoil grid created with `airfoilType=arc`

airfoil



airfoil NACA0012

Airfoil grid created with `airfoilType=diamond`



Airfoil grid created with `airfoilType=sinusoid`

NACA0012 airfoil created with  
`airfoilType=naca`

## 6 BoxMapping

This Mapping defines a box in three-dimensions:

$$\mathbf{x}(r_1, r_2, r_3) = (x_a + r_1(x_b - x_a), y_a + r_2(y_b - y_a), z_a + r_3(z_b - z_a))$$

The box can also be rotated around any one of the coordinate directions.

### 6.1 Member functions

#### 6.2 constructor

**BoxMapping(**

```
    const real xMin, const real xMax,  
    const real yMin, const real yMax,  
    const real zMin, const real zMax )
```

**Description:** Build a rectangular box in 3D. The box can also be rotated around one the coordinate directions.

**xMin,xMax** : minimum and maximum values for x(xAxis)

**yMin,yMax** : minimum and maximum values for y(yAxis)

**zMin,zMax** : minimum and maximum values for z(zAxis)

#### 6.3 rotate

**int**

```
rotate(const real angle,  
       const int axisOfRotation = 2,  
       const real x0 = 0.,  
       const real y0 = 0.,  
       const real z0 = 0.)
```

**Description:** Rotate the box around a coordinate direction.

**angle (input)** : angle in degrees.

**xisOfRotation (input)** : 0,1 or 2.

**x0,y0,z0 (input)** : rotate about this point.

#### 6.4 getVertices

**int**

```
getVertices(real & xMin, real & xMax, real & yMin, real & yMax, real & zMin, real &  
zMax ) const
```

**Purpose:** Return the bounds on the box.

**xMin, xMax, yMin, yMax, zMin, zMax (output)** : bounds on the box.

## 6.5 setVertices

```
int  
setVertices(const real & xMin =0.,  
           const real & xMax =1.,  
           const real & yMin =0.,  
           const real & yMax =1.,  
           const real & zMin =0.,  
           const real & zMax =1.)
```

**Purpose:** Set the bounds on the box.

**xMin, xMax, yMin, yMax, zMin, zMax (input)** : bounds on the box.

## 7 CircleMapping (ellipse too)

This mapping defines a circle or ellipse in two or three dimensions:

$$\begin{aligned}\mathbf{x}(r) &= (a \cos(2\pi r) + x_0, b \sin(2\pi r) + y_0) \\ \mathbf{x}(r) &= (a \cos(2\pi r) + x_0, b \sin(2\pi r) + y_0, z_0)\end{aligned}$$

on a constant  $z$ -plane. A partial arc can also be defined (see the figure with AnnulusMapping).

### 7.1 Constructor(2D)

```
CircleMapping(const real & x_=0.,
             const real & y_=0.,
             const real & a_=1.,
             const real & b_=a_,
             const real & startTheta_=0.,
             const real & endTheta_=1.)
```

**Description:** Define a circle or ellipse (or an arc there-of) in 2D, semi-axes a, and b, angle from startTheta\*twoPi to endTheta\*twoPi

```
x(I,axis1)=a*cos(thetaFactor*(r(I,axis1)-startTheta))+xa;
x(I,axis2)=b*sin(thetaFactor*(r(I,axis1)-startTheta))+ya;
```

**x\_ (input)** : x coordinate of center

**y\_ (input)** : y coordinate of center

**a\_ (input)** : length of semi axis along x (radius for a circle)

**b\_ (input)** : length of semi axis along y (radius for a circle)

**startTheta\_ (input)**: starting angle (in units of radians/(2 pi))

**endTheta\_ (input)**: ending angle (in units of radians/(2 pi))

### 7.2 Constructor(3D)

```
CircleMapping(const real & x_,
              const real & y_,
              const real & z_,
              const real & a_,
              const real & b_,
              const real & startTheta_,
              const real & endTheta_ )
```

**Description:** Define a circle or ellipse (or an arc there-of) in 3D (constant z), semi-axes a, and b, angle from startTheta\*twoPi to endTheta\*twoPi

```
x(I,axis1)=a*cos(thetaFactor*(r(I,axis1)-startTheta))+xa;  
x(I,axis2)=b*sin(thetaFactor*(r(I,axis1)-startTheta))+ya;  
x(I,axis3)=za;
```

**x\_ (input)** : x coordinate of center

**y\_ (input)** : y coordinate of center

**z\_ (input)** : z coordinate of center

**a\_ (input)** : length of semi axis along x (radius for a circle)

**b\_ (input)** : length of semi axis along y (radius for a circle)

**startTheta\_ (input)**: starting angle (in units of radians/(2 pi))

**endTheta\_ (input)**: ending angle (in units of radians/(2 pi))

## 8 ComposeMapping: compose two mappings

This mapping can be used to create a new mapping by composing two existing mappings.

### 8.1 Constructors

Mapping()	Default constructor
Mapping( Mapping & mapa, Mapping & mapb)	create a mapping, mapb o mapa

### 8.2 Member Functions

void map( realArray & r, realArray & x, realArray & xr )	evaluate the mapping and derivative
void inverseMap( realArray & x, realArray & r, realArray & xr )	evaluate the inverse mapping and derivative
void get( const Dir & dir, const String & name )	get from a database file
void put( const Dir & dir, const String & name )	put to a database file

Here is an example of the use of the ComposeMapping class. The composed mapping consists of a mapping for a cube followed by a rotation mapping.

```
#include "maputil.h"

void main()
{
    BoxMapping box(0.,.5,0.,.5,0.,.5);           // Define grid to be a cube

    MatrixMapping rotation;                      // Define a matrix mapping
    rotation.rotate( zAxis, Pi/2. );             // rotate about z axis

    ComposeMapping rotatedBox( box,rotation );    // define a mapping by composition

    r(axis1)=.5; r(axis2)=.5; r(axis3)=.5;
    rotatedBox.map( r,x,xr );                   // evaluate the mapping
}
```

## 9 CompositeSurface: define a surface formed from many sub-surfaces

\*\*\*\*\* This class is still under development \*\*\*\*\*

The `CompositeSurface` Mapping is used to represent a surface that is formed from a collection of sub-surfaces. This Mapping is not a normal Mapping since it does not represent a transformation from the unit square.

As an example, a `CompositeSurface` is used to represent the collection of NURBS and trimmed-NURBS surfaces that can be created by CAD packages. A single `CompositeSurface` can hold any number of these surfaces. Each sub-surface in a `CompositeSurface` is just any Mapping. Usually every sub-surface will actually be a surface in 3D but this is not necessary.

The most common use for a `CompositeSurface` is in combination with the Hyperbolic surface grid generator. This surface grid generator can grow a grid over a portion of a `CompositeSurface`, starting from some initial curve on the surface.

### 9.1 Projection onto the composite surface

The `CompositeSurface` has a function `project` that can be used to take one or more points in space,  $\mathbf{x}_i$ , and project these points onto the `CompositeSurface`, giving new points  $\mathbf{x}_i^p$ .

The hyperbolic surface grid generator, for example, will march a line of points over the `CompositeSurface`. At each step in its marching algorithm, new positions will be predicted for the next position for the line of points. These predicted values are then projected exactly onto the `CompositeSurface`.

The projection algorithm make use of the following variables:

$\mathbf{x}$  : point near the surface that needs to be projected.

$s_0$  : initial guess for the sub-surface patch on which to look (may be omitted).

$\mathbf{x}_0$  : a previous point on the `CompositeSurface` that is near to  $\mathbf{x}$ . This may be the previous location of  $\mathbf{x}$  from a surface grid generator (may be omitted).

$\mathbf{n}_0$  : normal to the `CompositeSurface` at the point  $\mathbf{x}_0$  (may be omitted).

$\mathbf{x}_p$  : projected point on the surface.

$s_p$  : subsurface index where the point was projected.

$\mathbf{n}_p$  : normal to the `CompositeSurface` at the point  $\mathbf{x}_p$ .

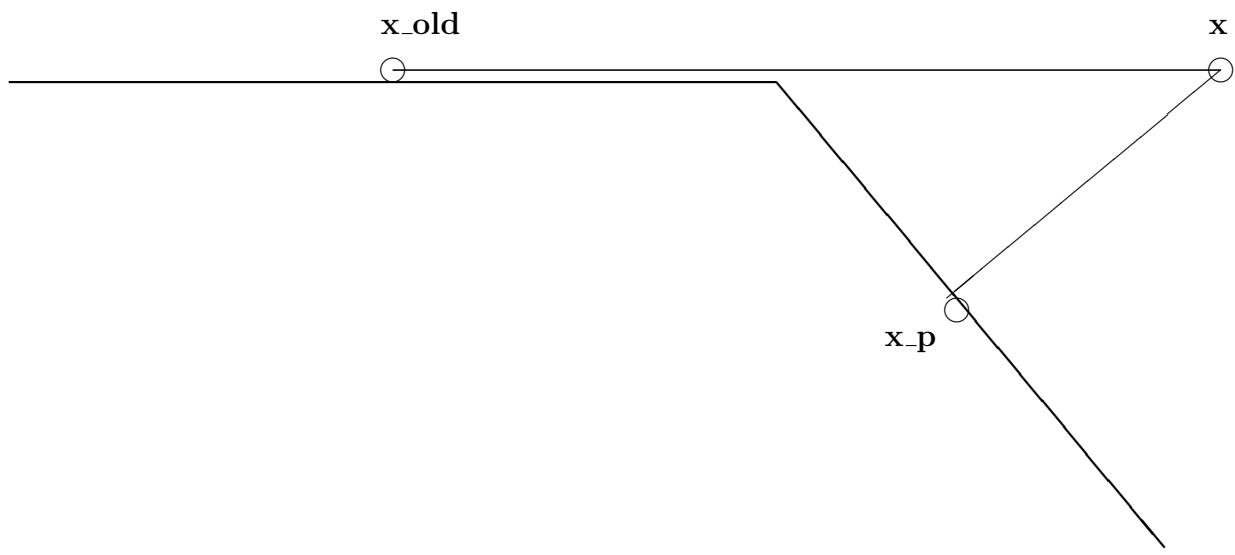
Here is the basic projection algorithm

1. Project  $\mathbf{x}$  on the sub-surface patch  $s_0$ , giving the point  $\mathbf{y} = P_{s_0}(\mathbf{x})$ . If  $\mathbf{y}$  is in the interior of the sub-surface then we are done.
2. If  $\mathbf{y}$  is on the boundary of the sub-surface  $s_0$  then compute the distance  $\mathbf{d}_0 = \|\mathbf{x} - \mathbf{y}\|$ . We will try to find a sub-surface that is closer than this distance.
3. Choose a new sub-surface to check,  $s_1$ , and project  $\mathbf{x}$  onto this surface,  $\mathbf{y}_1 = P_{s_1}(\mathbf{x})$ .

#### 9.1.1 Moving around sharp corners

If the point  $\mathbf{x}$  to be projected is near a sharp corner in the surface then there is some ambiguity as to the desired projection point  $\mathbf{x}_p$ .

If we are marching over the surface then we usually want the projected point  $\mathbf{x}_p$  to be some specified distance from the old point  $\mathbf{x}_0$ . In this case we may have to adjust the projected point and move it away from the corner.

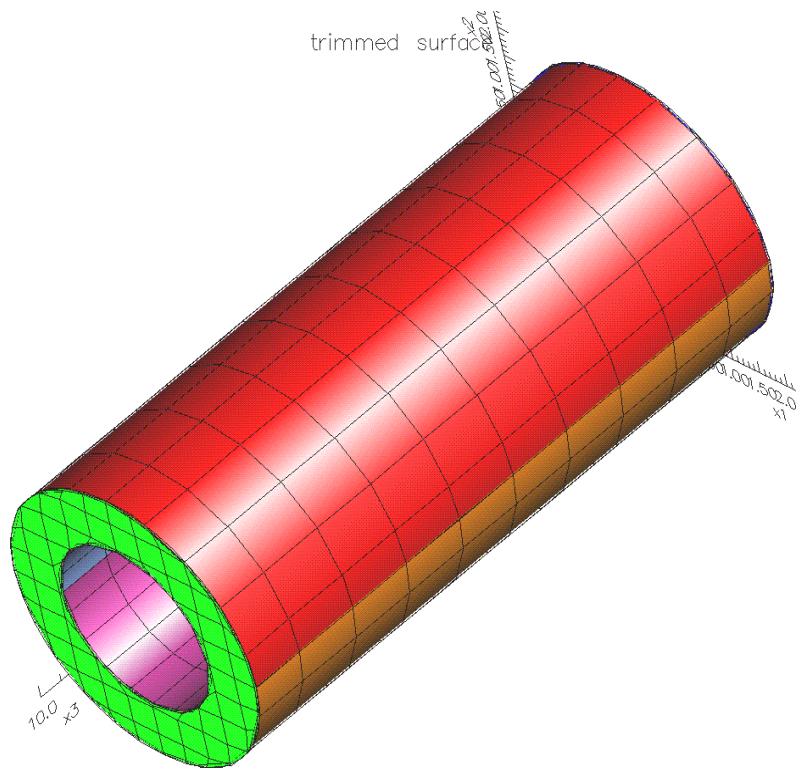


## 9.2 updateTopology

```
void  
updateTopology()
```

**Description:** Update the topology interactively.

### 9.3 Examples



A CompositeSurface for a cylindrical surface read from an IGES file created by pro/ENGINEER

## 10 CrossSectionMapping: define various surfaces by cross-sections

### 10.1 Description

The CrossSectionMapping can be used to define a Mapping from a collection of cross-sectional curves or surfaces. The available options for the cross-section type are

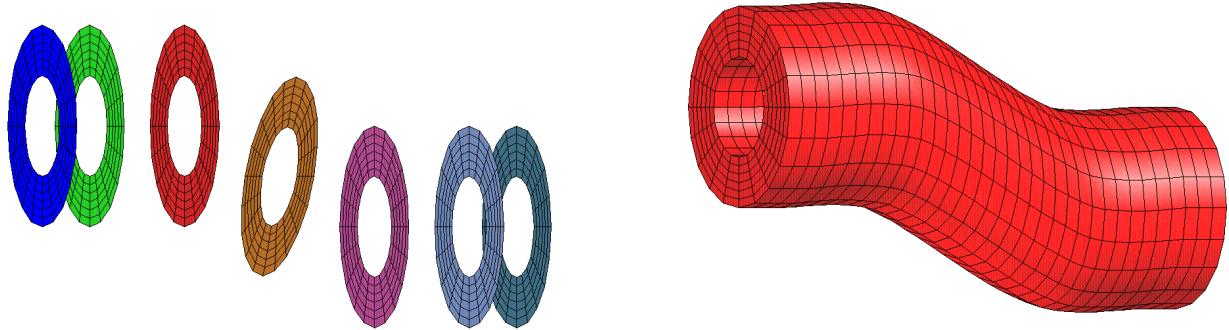
**general:** Build a mapping from a sequence of cross sections. The cross sections may be curves (such as circles or splines etc.) or they may be surfaces such as an Annulus or SmoothPolygonMapping.

**ellipse:** Define an ellipsoid, either a surface or a shell.

**joukowsky:** Define a “wing” surface with cross sections defined as Joukowsky airfoils.

Thanks go to Thomas Rutaganira for help with this Mapping.

pipe



Six AnnulusMapping's to be used as cross-sections.

A volume grid created from the six AnnulusMapping's (cubic interpolation).

### 10.2 General cross-section type

When the cross-section type is **general** the user specifies a sequence of curves (or surfaces) that will be used as the cross-sections.

Given a sequence of  $n$  cross-sectional curves

$$\mathbf{c}_i(r_0), \quad i = 0, 1, \dots, n - 1 \quad (\text{cross-section curves})$$

the CrossSectionMapping defines a surface by blending the curves in the regions between them.

$$\mathbf{x}(\mathbf{t}, r_a) = \mathbf{C}(\mathbf{c}_i(\mathbf{t}), r_a) .$$

The parameter direction(s)  $\mathbf{t}$  will be called the *tangential* direction(s). If the cross-sections are curves then  $\mathbf{t} = r_0$ ; if they are surfaces  $\mathbf{t} = (r_0, r_1)$ . The direction  $r_a$  will be called the *axial* direction. As  $r_a$  varies for fixed  $\mathbf{t}$  we trace a curve that follows the axis of the surface.

With linear interpolation (default) the curve is a linearly interpolated between successive cross-sections:

$$\mathbf{x}(\mathbf{t}, r_a) = (1 - s_a)\mathbf{c}_i(\mathbf{t}) + s_a\mathbf{c}_{i+1}(\mathbf{t}) \quad \text{for } \frac{i}{n-1} \leq S(r_a) \leq \frac{i+1}{n-1}$$

$$s_a = S(r_a)(n-1) - \lfloor S(r_a)(n-1) \rfloor$$

where the axial parameterization function  $S(r_a)$  is defined below. The variable  $s_a$  varies between 0 and 1 as we move from cross-section  $i$  to cross-section  $i+1$ . Here  $\lfloor x \rfloor$  is the biggest integer less or equal to  $x$ .

With **index** parameterization  $S(r_a) = r_a$  in which case the cross-sections are parameterized as if they were equally spaced. Thus there will be approximately an equal number of axial grid lines between any two cross-sections. Normally this is not a good parameterization unless the cross-sections are nearly equally spaced.

With **arcLength** parameterization (the default) the axial direction is parameterized using the average distance between the cross-sectional curves. The average distance between the curves is computed by evaluating each curve at  $m$  equally spaced points  $\{\mathbf{c}_i(t_j)\}_{j=0}^{m-1}$ ,  $t_j = j/(m-1)$ , and then taking the average of the distances between these points.

$$s_{i+1} = s_i + \|\mathbf{c}_{i+1} - \mathbf{c}_i\|/L \quad s_0 = 0, \quad L \text{ chosen so } s_{n-1} = 1.$$

$$\|\mathbf{x}_{i+1} - \mathbf{c}_i\| = \frac{1}{m} \sum_{j=0}^{m-1} \|\mathbf{x}_{i+1}(t_j) - \mathbf{c}_i(t_j)\|$$

The *inverse* of the function  $S(r_a)$  is defined by fitting a spline to the data points  $\{s_i\}_{i=0}^{n-1}$ . That is a spline fitted to the points  $\{s_i\}_{i=0}^{n-1}$  will define the function  $S^{-1}$ . The exact properties of the spline can be adjusted by choosing the “**change arclength spline parameters**” option. For example, one may want to use a spline with tension or a spline that is shape preserving. See the SplineMapping documentation, section (35), for further details.

With a **userDefined** parameterization the user defines the parameter values  $s_i$  for each of the cross-sections. The values  $s_i$  should satisfy  $s_0 = 0$ ,  $s_i < s_{i+1}$  and  $s_{n-1} = 1$ . Normally the value of  $s_i - s_{i-1}$  would be based on the distance between the cross-section curves  $i-1$  and  $i$ . The inverse of the function  $S(r_a)$  is defined by fitting a spline to the data points  $\{s_i\}_{i=0}^{n-1}$ .

With piecewise cubic interpolation the mapping is defined as a cubic polynomial on each interval (except the first and last where quadratic polynomials are used)

$$\mathbf{x}(\mathbf{t}, r_a) = q_{03}(s_a)\mathbf{c}_{i-1}(\mathbf{t}) + q_{13}(s_a)\mathbf{x}_i(\mathbf{t}) + q_{23}(s_a)\mathbf{x}_{i+1}(\mathbf{t}) + q_{33}(s_a)\mathbf{x}_{i+2}(\mathbf{t}) \quad \text{for } \frac{i}{n-1} \leq S(r_a) \leq \frac{i+1}{n-1}$$

$$s_a = S(r_a)(n-1) - \lfloor S(r_a)(n-1) \rfloor$$

where  $q_{i3}$  are cubic Lagrange polynomials. On the left edge a quadratic polynomial is used which passes through the cross sections 0, 1, 2. Similarly for the right edge.

### 10.2.1 Notes for generating general cross section mappings

1. For best results the cross sections should be **nearly equally spaced**.
2. With the cubic interpolation option: quadratic polynomials are used on the first and last segments. If you wish an end segment to be “straight” then you should place three cross sections in a straight line at the end.
3. It is up to you to make sure that the cross sections are all parameterized in a compatible fashion; if they are not then the axial grid lines may twist and the grid may not be invertible.
4. With the cubic interpolation option: if the cross sections vary rapidly from one to the next or the cross sections are very unevenly spaced then the cubic interpolant (or quadratic interpolants on the ends) may wiggle a lot. Adding more cross-sections should fix this problem.

### 10.3 Ellipse cross-section type

When the cross-section type is `ellipse` the Mapping defines an ellipsoid in cylindrical coordinates with semi-axes `a, b, c`:

$$\begin{aligned}\zeta &= (\text{endS} - \text{startS})r_0 - (1. - 2. * \text{startS}) \\ \rho &= \sqrt{1 - \zeta^2} \\ R &= \text{innerRadius} + r_2(\text{outerRadius} - \text{innerRadius}) \\ x_0 &= aR\rho \cos(2\pi r_1) + \text{x0} \\ x_1 &= bR\rho \sin(2\pi r_1) + \text{y0} \\ x_2 &= cR\zeta + \text{z0}\end{aligned}$$

The default values for the parameters are `startS=0`, `endS=1`, `innerRadius=1`, `outerRadius=1.5`, `x0=0`, `y0=0`, `z0=0`.

After building an ellipsoid one would normally remove the singularities at the poles by building patches to cover the ends using the `reparameterize` option. See the example in the overlapping grid documentation.

### 10.4 Joukowsky cross-section type

This section needs to be written.

### 10.5 Cross section Mappings with polar singularities

It is often the case that one desires the cross-sections to converge to a point at one or both ends. In this case one should indicate that the Mapping has a polar singularity at one or both ends. One should also choose the last cross section to be a small ellipse. The `CrossSectionMapping` will then slightly deform the Mapping to cause the last cross-section to converge to a point. The resulting deformed Mapping can then have an orthographic patch built to cover the singularity using the `ReparameterizationTransform`.

In order for the `OrthographicTransform` to nicely remove a polar singularity, the Mapping with the singularity must locally near the pole be parameterized like

$$\begin{aligned}\mathbf{x} &\sim A\rho(r_1)(a \cos(\theta(r_2)), b \sin(\theta(r_2))) \\ \rho &= \sqrt{1 - \zeta^2} \\ \zeta &= 2r_1 - 1\end{aligned}$$

Thus locally the surface must look like an ellipsoid (it can be oriented in any direction, the above equation assumes a particular orientation). The “radius” of the cross section, defined, say, by the average distance of the cross-section from its centroid, should be decaying like  $\rho \sim \sqrt{r_1}$  as  $r_1 \rightarrow 0$ . If the radius decays at a different rate then the coordinates lines on the orthographic patch will not be rectangular near the pole.

### 10.6 Constructor

`CrossSectionMapping(`

```
    const real startS_,  
    const real endS_,  
    const real startAngle_,  
    const real endAngle_,  
    const real innerRadius_,  
    const real outerRadius_,  
    const real x0_,
```

```
const real y0_ ,  
const real z0_ ,  
const real length_ ,  
const int domainDimension_ )
```

**Description:** Default Constructor, define a mapping from cross-sections.

Build a mapping defined by cross sections. In the **general** case the cross-sections are defined by other Mapping's. One can also build an ellipsoid when the cross-section type is **ellipse** or a Joukowsky wing. enum CrossSectionType:

- general
- ellipse
- joukowsky

enum Parameterization:

**arcLength** : parameterize by the arc length distance between the centroids of the cross sectional curves.

**index** : parameterize by the index of the cross section.

**userDefined** : supply a parameterization.

## 10.7 setCrossSectionType

```
int  
setCrossSectionType(CrossSectionTypes type)
```

**Description:** Define the cross-section type. \*this is not finished yet\*

**type (input):**

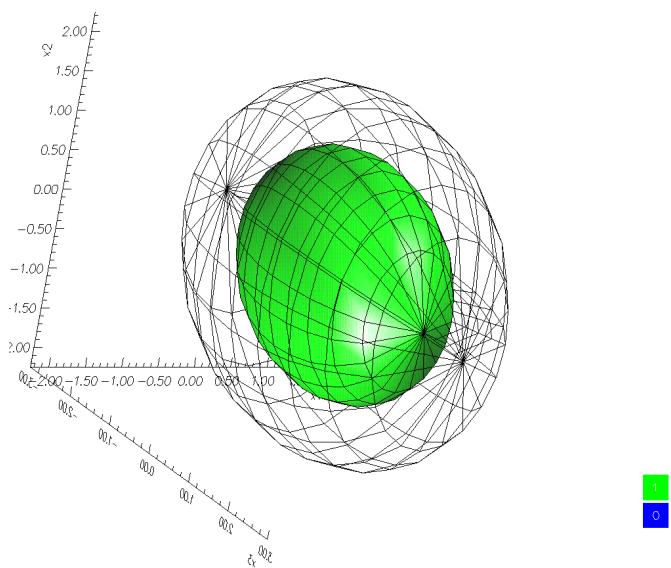
## 10.8 Constructor

```
int  
initialize()
```

**Description:** private routine. Initialize the parameterization for the cross sections.

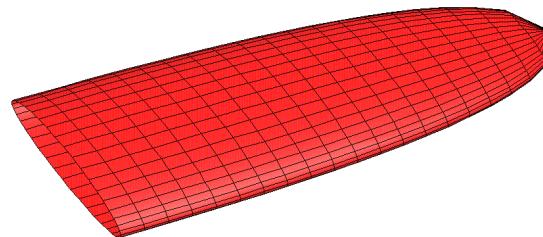
## 10.9 Examples

Joukowsky wing

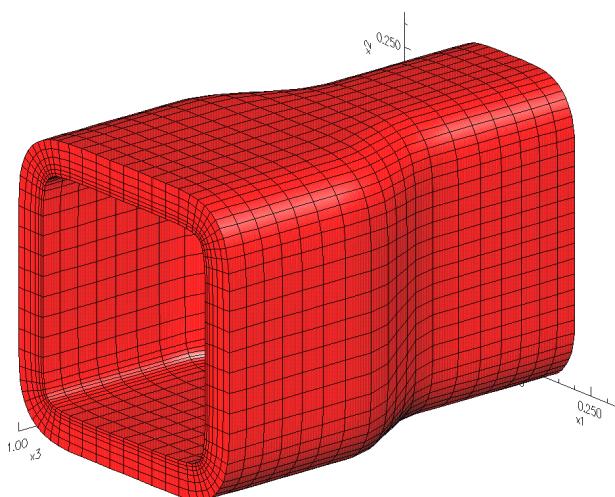


An ellipsoid created with the `ellipse` cross-section type.

Cross section from `SmoothedPolygons`



A Joukowsky airfoil created with the `joukowsky` cross-section type.

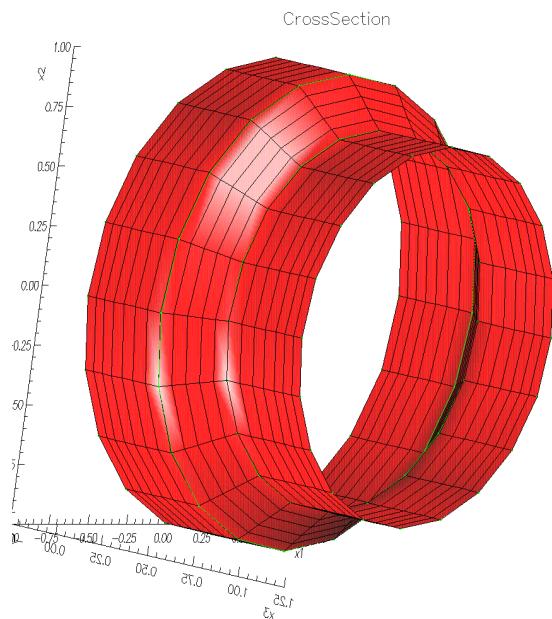


A volume grid created from 4 smoothed polygon cross-section surfaces (linear interpolation).

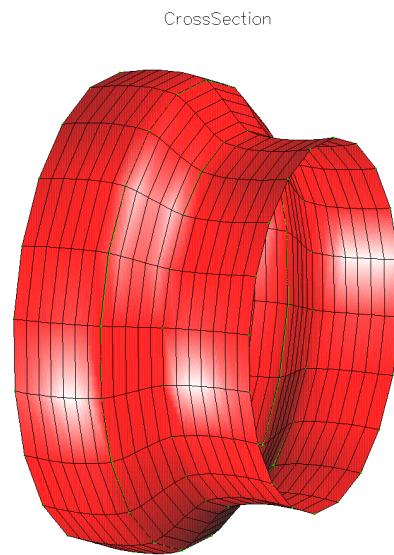
```

1  *
2 * Define a cross section mapping
3 *
4 Circle or ellipse (3D)
5   specify radius of the circle
6     1.
7   specify centre
8     0. 0. 0.
9   lines
10    21 25
11   mappingName
12     circle0
13 exit
14 Circle or ellipse (3D)
15   specify radius of the circle
16     1.
17   specify centre
18     0. 0. .4
19   lines
20    21 31
21   mappingName
22     circle1
23 exit
24 Circle or ellipse (3D)
25   specify radius of the circle
26     .8
27   specify centre
28     0. 0. .6
29   mappingName
30     circle2
31 exit
32 Circle or ellipse (3D)
33   specify radius of the circle
34     .8
35   specify centre
36     0. 0. 1.
37   mappingName
38     circle3
39 exit
40 *
41 * make a cross section mapping
42 CrossSection
43   general
44   4
45   circle0
46   circle1
47   circle2
48   circle3
49   x+r 30
50   y+r

```



A surface grid created from 4 circular cross-sections (linear interpolation). The cross-sections are shown in green.



As above with cubic interpolation.

## 11 CylinderMapping

This mapping defines a cylindrical volume or surface in three-dimensions.

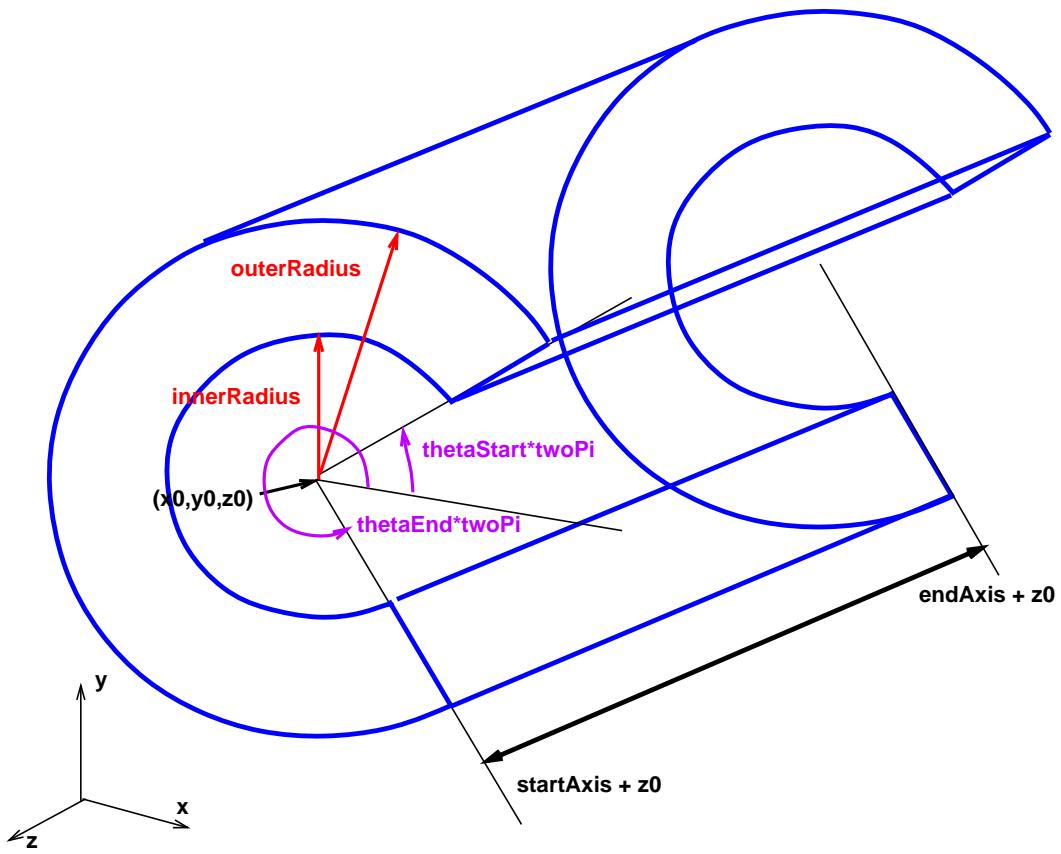


Figure 4: The CylinderMapping defines a cylinder in three-dimensions.

### 11.1 Constructor

CylinderMapping(

```
    const real & startAngle_ = 0.,
    const real & endAngle_ = 1.,
    const real & startAxis_ = -1.,
    const real & endAxis_ = +1.,
    const real & innerRadius_ = 1.,
    const real & outerRadius_ = 1.5,
    const real & x0_ = 0.,
    const real & y0_ = 0.,
    const real & z0_ = 0.,
    const int & domainDimension_ = 3,
    const int & cylAxis1_ = axis1,
    const int & cylAxis2_ = axis2,
    const int & cylAxis3_ = axis3
)
```

**Purpose:** Create a 3D cylindrical volume or surface.

**Notes:** This mapping defines a cylinder in three-dimensions:

$$\begin{aligned}\theta &= 2\pi(\theta_0 + r_0(\theta_1 - \theta_0)) \\ R(r_1) &= (R_0 + r_2(R_1 - R_0)) \\ \mathbf{x}(r_0, r_1, r_2) &= (R \cos(\theta) + x_0, R \sin(\theta) + y_0, s_0 + r_1(s_1 - s_0) + z_0)\end{aligned}$$

The above cylinder has the z-axis as the axial direction. It is also possible to have the axial direction to point in any of the coordinate directions using the (`cylAxis1`, `cylAxis2`, `cylAxis3`) variables (which should be a permutation of (0,1,2)): Changing these variables will permute the definition of  $(x_0, x_1, x_2)$ :

$$(x_{cylAxis1}, x_{cylAxis2}, x_{cylAxis3}) = (R \cos(\theta) + x_0, R \sin(\theta) + y_0, s_0 + r_2(s_1 - s_0) + z_0)$$

NOTE that the parameter space coordinates are always  $(\theta, \text{axial}, \text{radial})$ .

**startAngle (input)** : starting angle ( $\theta_0$ ) NOTE: angles are 1-periodic!

**endAngle (input)** : ending angle ( $\theta_1$ ) NOTE: angles are 1-periodic!

**startAxis (input)** : axial coordinate of the start of the cylinder ( $s_0$ ).

**endAxis (input)** : axial coordinate of the end of the cylinder ( $s_1$ ).

**innerRadius (input)** : inner radius ( $R_0$ ).

**outerRadius (input)** : outer radius ( $R_0$ ).

**x0,y0,z0 (input)** : center of the cylinder ( $x_0, y_0, z_0$ ).

**domainDimension (input)** : 3 means the cylinder is a volume, 2 means the cylinder is a surface.

**cylAxis1,cylAxis2,cylAxis3 (input)** : change these to be a permutation of (axis1,axis2,axis3) to change the orientation of the cylinder. NOTE: axis1==0, axis2==1, axis3==2.

## 11.2 setAngle

**int**

**setAngle(const real & startAngle\_ =0., const real & endAngle\_ =1.)**

**Description:** Set the initial and final angles.

**startAngle (input)** :

**endAngle (input)** :

## 11.3 setAxis

**int**

**setAxis(const real & startAxis\_ =-1., const real & endAxis\_ =+1.)**

**Description:** Set the starting and ending axial positions.

**startAxis (input)** : axial coordinate of the start of the cylinder ( $s_0$ ).

**endAxis (input)** : axial coordinate of the end of the cylinder ( $s_1$ ).

## 11.4 setOrientation

```
int  
setOrientation( const int & cylAxis1_ =0,  
                const int & cylAxis2_ =1,  
                const int & cylAxis3_ =2)
```

**Description:** Set the orientation of the cylinder.

**cylAxis1,cylAxis2,cylAxis3 (input) :** change these to be a permutation of (axis1, axis2, axis3) to change the orientation of the cylinder. NOTE: axis1==0, axis2==1, axis3==2.

## 11.5 setOrigin

```
int  
setOrigin(const real & x0_ =0.,  
          const real & y0_ =0.,  
          const real & z0_ =0.)
```

**Description:** Set the centre of the cylinder.

**x0,y0,z0 (input) :** center of the cylinder ( $x_0, y_0, z_0$ ).

## 11.6 setRadius

```
int  
setRadius(const real & innerRadius_ =1.,  
          const real & outerRadius_ =1.5)
```

**Description:** Set the inner and outer radii.

**innerRadius (input) :** inner radius ( $R_0$ ).

**outerRadius (input) :** outer radius ( $R_0$ ).

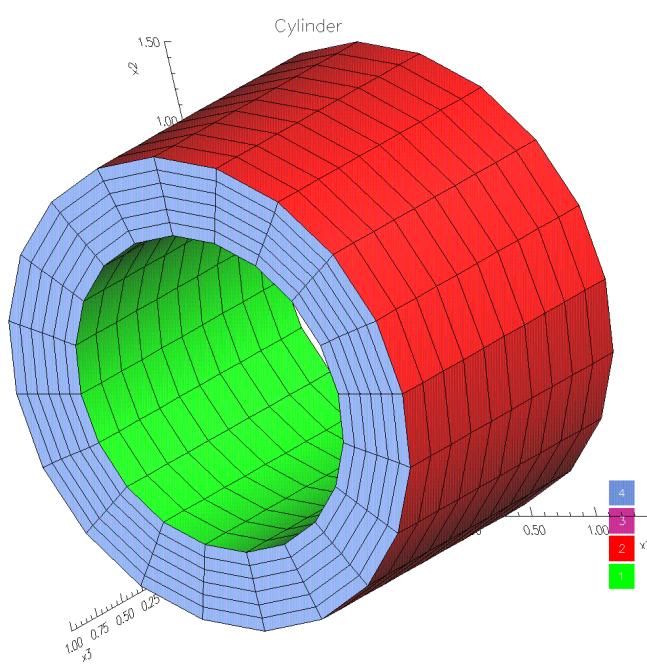


Figure 5: CylinderMapping. This is the volume representation. The cylinder may also represent a surface.

## 12 DataPointMapping: create a mapping from an array of grid points

### 12.1 Description

The `DataPointMapping` can be used to create a mapping from a set of grid points. The grid points may be in a file (such as Plot3D format) or can be in an A++ array.

The `DataPointMapping` defines a Mapping transformation by interpolating the grid points.

For `orderOfInterpolation=2` the transformation is defined as a linear interpolation (i.e. 2 points in each direction). In 2D this would be

$$\mathbf{x}(\mathbf{r}) = (1 - \hat{r}_2)[(1 - \hat{r}_1)\mathbf{x}_{00} + \hat{r}_1\mathbf{x}_{10}] + \hat{r}_2[(1 - \hat{r}_1)\mathbf{x}_{01} + \hat{r}_1\mathbf{x}_{11}]$$

where  $\mathbf{x}_{mn}$  are the grid points that define the cell and  $\hat{r}_m$  are the relative distances of the point  $\mathbf{r}$  from the r-coordinates of the corner point  $\mathbf{x}_{00}$

$$\hat{r}_1 = \frac{(r_1 - r_{00})}{\Delta r_1}, \quad \hat{r}_2 = \frac{(r_2 - r_{00})}{\Delta r_2}$$

The derivatives returned by the Mapping are just the derivatives of the above expression.

For `orderOfInterpolation=4` the transformation is defined as a cubic interpolation (i.e. 4 points in each direction). In 2D this would be defined as

$$\mathbf{x}(\mathbf{r}) = \sum_{m=0}^3 q_m(\hat{r}_2) \sum_{n=0}^3 q_n(\hat{r}_1) \mathbf{x}_{nm} \quad (1)$$

$$q_m(s) = \prod_{n \neq m} (s - n)/(m - n) \quad \text{Lagrange polynomials, } q_m(n) = \delta_{mn} \quad (2)$$

The derivatives returned by the Mapping are just the derivatives of the above expression.

Figure 12.1 shows a grid for part of the coast of the USA, created by Lotta Olsson. The grid was created with the help of the HYPGEN hyperbolic grid generator and saved in Plot3D format.

### 12.2 Fast Approximate Inverse

Since the `DataPointMapping` is extensively used, a specialized fast approximate inverse has been defined for both the second order (bi-linear/tri-linear) and fourth order representations..

The inverse consists of the steps:

1. Find the closest vertex on the grid to the point,  $\mathbf{x}$ , to be inverted.
2. Find the hexahedral that  $\mathbf{x}$  is in.
3. Invert the local interpolant with Newton's method.

The linear interpolant within a given cell is

$$\mathbf{x}(\mathbf{r}) = (1 - \hat{r}_0)[(1 - \hat{r}_1)\mathbf{x}_{00} + \hat{r}_1\mathbf{x}_{01}] + \hat{r}_0[(1 - \hat{r}_1)\mathbf{x}_{10} + \hat{r}_1\mathbf{x}_{11}]$$

in 2D or

$$\begin{aligned} \mathbf{x}(\mathbf{r}) = & (1 - \hat{r}_0)[(1 - \hat{r}_1)((1 - \hat{r}_2)\mathbf{x}_{000} + \hat{r}_2\mathbf{x}_{001}) + \hat{r}_1((1 - \hat{r}_2)\mathbf{x}_{010} + \hat{r}_2\mathbf{x}_{011})] \\ & + \hat{r}_0[(1 - \hat{r}_1)((1 - \hat{r}_2)\mathbf{x}_{100} + \hat{r}_2\mathbf{x}_{101}) + \hat{r}_1((1 - \hat{r}_2)\mathbf{x}_{110} + \hat{r}_2\mathbf{x}_{111})] \end{aligned}$$

in 3D where  $\mathbf{x}_{lmn}$  are the grid points that define the cell and  $\hat{r}_m$  are the scaled unit square coordinates,  $\hat{r}_m \in [0, 1]$  for points within the cell.

A Newton iteration to invert the mapping would look like

$$\mathbf{x}(\mathbf{r}_0 + \delta\mathbf{r}) = \mathbf{x}(\mathbf{r}_0) + \frac{\partial \mathbf{x}}{\partial \mathbf{r}}(\mathbf{r}_0)\delta\mathbf{r}.$$

where

$$\left[ \frac{\partial \mathbf{x}}{\partial \mathbf{r}} \right] = [\mathbf{a}_0 \quad \mathbf{a}_1 \quad \mathbf{a}_2]$$

where

$$\begin{aligned}\mathbf{a}_0 &= (1 - \hat{r}_1)((1 - \hat{r}_2)(\mathbf{x}_{100} - \mathbf{x}_{000}) + \hat{r}_2(\mathbf{x}_{101} - \mathbf{x}_{001})) + \hat{r}_1((1 - \hat{r}_2)(\mathbf{x}_{110} - \mathbf{x}_{010}) + \hat{r}_2(\mathbf{x}_{111} - \mathbf{x}_{011})) \\ \mathbf{a}_1 &= (1 - \hat{r}_0)((1 - \hat{r}_2)(\mathbf{x}_{010} - \mathbf{x}_{000}) + \hat{r}_2(\mathbf{x}_{011} - \mathbf{x}_{001})) + \hat{r}_1((1 - \hat{r}_2)(\mathbf{x}_{110} - \mathbf{x}_{100}) + \hat{r}_2(\mathbf{x}_{111} - \mathbf{x}_{101})) \\ \mathbf{a}_2 &= (1 - \hat{r}_0)((1 - \hat{r}_1)(\mathbf{x}_{001} - \mathbf{x}_{000}) + \hat{r}_2(\mathbf{x}_{101} - \mathbf{x}_{100})) + \hat{r}_1((1 - \hat{r}_2)(\mathbf{x}_{011} - \mathbf{x}_{010}) + \hat{r}_2(\mathbf{x}_{111} - \mathbf{x}_{110}))\end{aligned}$$

In the special case when the cell is a regular 'diamond' shape the linear interpolant simplifies to

$$\mathbf{x}(\mathbf{r}) - \mathbf{x}_{000} = \hat{r}_0(\mathbf{x}_{100} - \mathbf{x}_{000}) + \hat{r}_1(\mathbf{x}_{010} - \mathbf{x}_{000}) + \hat{r}_2(\mathbf{x}_{001} - \mathbf{x}_{000})$$

or

$$\mathbf{x}(\mathbf{r}) = \hat{r}_0\mathbf{x}_{100} + \hat{r}_1\mathbf{x}_{010} + \hat{r}_2\mathbf{x}_{001}$$

where the 'inverse' is computed by solving a  $d \times d$  matrix

$$[\mathbf{x}_{100} \quad \mathbf{x}_{010} \quad \mathbf{x}_{001}] \begin{bmatrix} \hat{r}_0 \\ \hat{r}_1 \\ \hat{r}_2 \end{bmatrix} = [\mathbf{x}]$$

\*\* The newton step will do this case exactly\*\*

### 12.3 Examples

Grid points saved in plot3d format can be read in to form a DataPointMapping. To illustrate this we first build a plot3d file by creating a mapping and saving it in plot3d format. This is accomplished by the following command file (`Overture/sampleMappings/createCylinderPlot3d.cmd`)

```
1  *
2  * ogen command file to create a plot3d file for a Mapping
3  *
4  create mappings
5  *
6  cylinder
7    bounds on theta
8      0. .75
9    mappingName
10   partialCylinder
11 exit
12 *
13 save plot3d file
14 partialCylinder
15 cylinder.plot3d
16   save file
17   exit
18 exit this menu
```

There are a variety of options for defining the contents of the plot3d file. For example it can be a binary file or ascii file, it can contain the `iblank` array (i.e. mask array) or not. These options are available from the `save plot3d file` menu.

A plot3d file can be either read from the DataPointMapping menu or the `create mappings` menu. Here is a command file to read the plot3d file created in the previous step, (`Overture/sampleMappings/readCylinderPlot3d.cmd`)

```

1  *
2  * ogen cmd file to read the plot3d file created from createCylinderPlot3d.cmd
3  *
4  create mappings
5  *
6  * These next two commands will read a plot3dfile and build a DataPointMapping
7  * You can use "change a mapping" to rename this mapping or change the lines etc.
8  read plot3d file
9  cylinder.plot3d
10 *
11 view mappings
12 cylinder.plot3d-grid0
13 pause
14 exit
15 exit this menu

```

## 12.4 Constructor

**DataPointMapping()**

**Purpose:** Default Constructor.

## 12.5 getDataPoints

**getDataPoints()**

**Description:** Return the array of data points. It will not be the same array as was given to setDataPoints since ghostlines will have been added. Use getGridIndexRange to determine the index positions for the grid boundaries.

**Return value:** array of data points,  $xy(I1, I2, I3, 0:r-1)$ ,  $r=rangeDimension$

## 12.6 getGridIndexRange

**const IntegerArray &**  
**getGridIndexRange()**

**Description:** Return the gridIndexRange array for the data points. These values indicate the index positions for the grid boundaries.

**Return value:** The gridIndexRange(0:1,0:2).

## 12.7 getDimension

**const IntegerArray &**  
**getDimension()**

**Description:** Return the dimension array for the data points. These values indicate the index positions for the array dimensions.

**Return value:** The dimension(0:1,0:2).

## 12.8 setDataPoints

```
int
setDataPoints(const realArray & xd,
              const int positionOfCoordinates = 3,
              const int domainDimension_ = -1,
              const int numberofGhostLinesInData = 0,
const IntegerArray & xGridIndexRange = Overture::nullIntArray())
```

**Purpose:** Supply data points as

1.  $xd(0:r-1,I,J,K)$  if  $positionOfCoordinates == 0 \rightarrow domainDimension = domainDimension_-$
2.  $xd(I,0:r-1)$  if  $positionOfCoordinates == 1 \rightarrow domainDimension = 1$
3.  $xd(I,J,0:r-1)$  if  $positionOfCoordinates == 2 \rightarrow domainDimension = 2$
4.  $xd(I,J,K,0:r-1)$  if  $positionOfCoordinates == 3 \rightarrow domainDimension = domainDimension_-$

where  $r = \text{number of dimensions}$  (range dimension)

**xd (input):** An array of values defining the coordinates of a grid of points. This routine make a COPY of this array.

**positionOfCoordinates (input):** indicates the "shape" of the input array xd.

**domainDimension\_ (input):** As indicated above this parameter defines the domainDimension when positionOfCoordinates is 0 or 3.

**numberOfGhostLinesInData (input) :** The data includes the coordinates of this many ghost lines (for all sides). These values are over-ridden by the index array argument.

**xGridIndexRange (input):** If this array is not null and size (2,0:r-1) then these values indicate the points in the array xd that represent the boundary points on the grid. Use this option to specify arbitrary number of ghost points on any side.

**Remarks:** Note that by default the DataPointMapping will have the properties

- domainSpace = parameterSpace
- rangeSpace = cartesianSpace
- not periodic
- boundary conditions all 1

You will have to change the above properties as appropriate. NOTE: you should set the periodicity of this mapping before supplying data points.

## 12.9 setDataPoints

```
int
setDataPoints(const realArray & xd,
              const int positionOfCoordinates,
              const int domainDimension_,
              const int numberofGhostLinesInData[2][3],
const IntegerArray & xGridIndexRange = Overture::nullIntArray())
```

**Description:** Supply data points: Same as above routine except that the numberofGhostLinesInData can be defined as separate values for each face.

**numberofGhostLinesInData[side][axis] :** specify the number of ghostlines in the input data for each face.

## 12.10 computeGhostPoints

```
int  
computeGhostPoints( IndexRangeType & numberOfGhostLinesOld,  
                    IndexRangeType & numberOfGhostLinesNew )
```

**Access Level:** protected

**Description:** Determine values at ghost points that have not been user set: extrapolate or use periodicity  
Ghost lines on sides with boundaryCondition $\neq 0$  are extrapolated with a stretchingFactor (see below)  
so that the grid lines get further apart. This is useful for highly stretched grids so that the ghost  
points move away from the boundary.

## 12.11 setNumberOfGhostLines

```
int  
setNumberOfGhostLines( IndexRangeType & numberOfGhostLinesNew )
```

**Description:** Specify the number of ghost lines.

**numberOfGhostLinesNew(side, axis)** : specify the number of ghostlines.

## 12.12 projectGhostPoints

```
int  
projectGhostPoints(MappingInformation & mapInfo )
```

**Description:** Project the ghost points on physical boundaries onto the closest mapping found in a list of  
Mapping's

**mapInfo (input):** Project onto the closest mapping found in mapInfo.mappingList.

## 12.13 setDataPoints( fileName )

```
int  
setDataPoints( const aString & fileName )
```

**Description:** Assign the data points from a file of data. By default this routine will attempt to automatically  
determine the format of the file.

**fileName (input)** : name of an existing file of data (such as a plot3d file)

## 12.14 setMapping

```
int  
setMapping( Mapping & map )
```

**Description:** Build a data point mapping from grids points obtained by evaluating a mapping.

**map (input)** : Mapping to get data points from.

## 12.15 setMapping

```
int  
setMapping( Mapping & map )
```

**Description:** Build a data point mapping from grids points obtained by evaluating a mapping.

**map (input) :** Mapping to get data points from.

## 12.16 setOrderOfInterpolation

```
void  
setOrderOfInterpolation( const int order )
```

**Purpose:** Set the order of interpolation, 2 or 4.

**order (input) :** A value of 2 or 4.

## 12.17 setOrderOfInterpolation

```
int  
getOrderOfInterpolation()
```

**Purpose:** Get the order of interpolation.

**Return value:** The order of interpolation.

## 12.18 useScalarArrayIndexing

```
void  
useScalarArrayIndexing(const bool & trueOrFalse =FALSE)
```

**Purpose:** Turn on or off the use of scalar indexing. Scalar indexing for array operations can be faster when the length of arrays are smaller.

**trueOrFalse (input) :** TRUE means turn on scalra indexing.

### 12.18.1 sizeOf

```
real  
sizeOf(FILE *file = NULL) const
```

**Description:** Return size of this object

## 12.19 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively change parameters describing the Mapping. The user may choose to read in data points from a file. The current supported file formats are

- plot3d

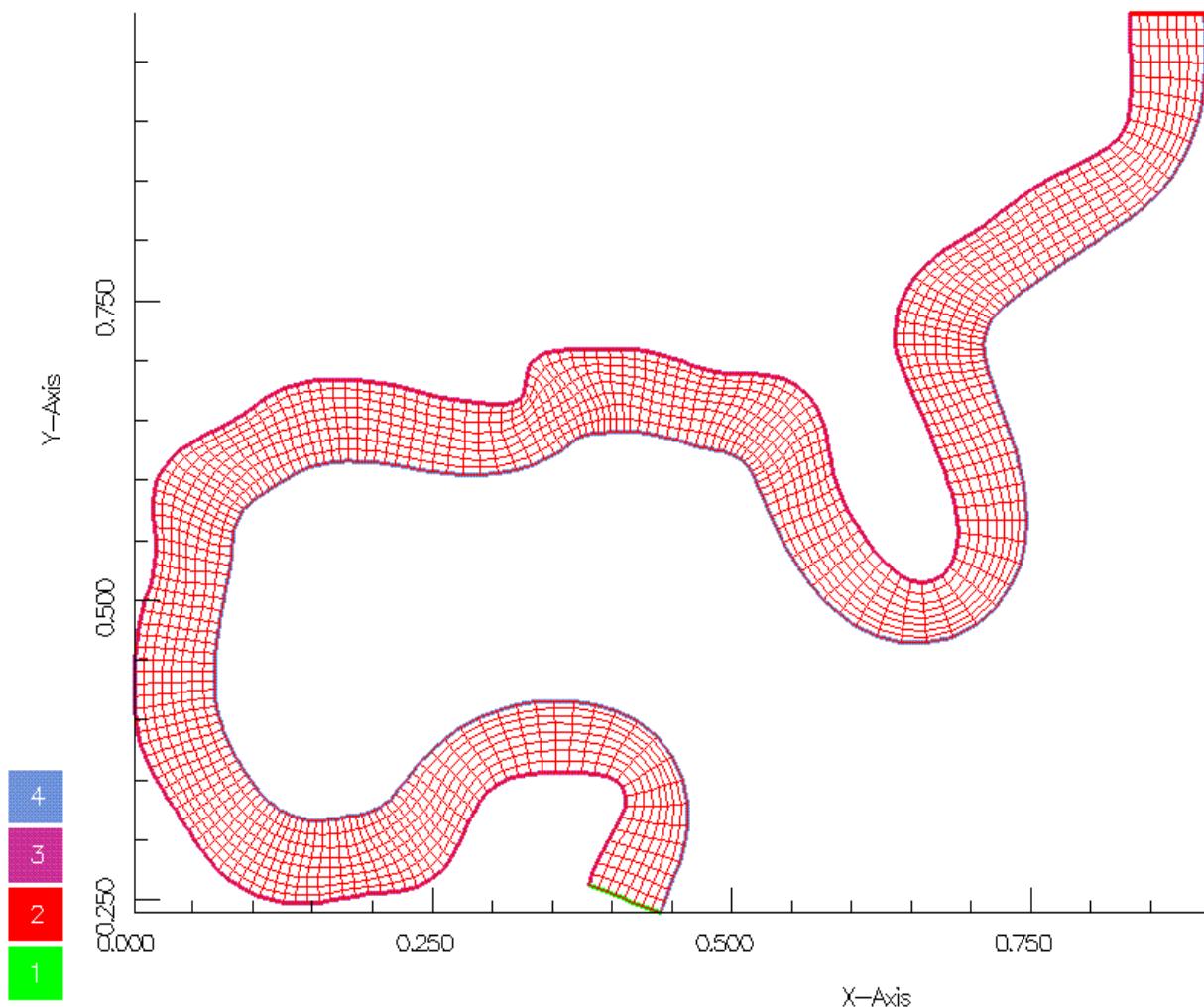


Figure 6: A DataPointMapping created from a Plot3D file.

## 13 DepthMapping: Add a depth to a 2D Mapping

### 13.1 Description

Define a 3D Mapping from 2D Mapping by extending in the z-direction by a variable amount

The depth mapping starts with some region defined in the  $(x, y) = (x_0, x_1)$  plane,  $(x_0, x_1) = \mathbf{x}_s(r_0, r_1)$ , such as an annulus or square etc. It then defines a 3D volume (or 3D surface) of the form

$$\mathbf{x}(r_0, r_1, r_2) = (x_0(r_0, r_1), x_1(r_0, r_1), x_2(r_0, r_1, r_2))$$

As the variable  $r_2$  varies, the initial 2D surface is deformed to define a generalized cylinder.

The depth coordinate,  $x_2$  is defined in a number of ways:

**Representation I:** In this case the depth is a function of  $(x_0, x_1)$  and uses some predefined functions available in the DepthMapping,

$$\mathbf{x}(\mathbf{r}) = (\mathbf{x}_s(r_0, r_1), z_0 + r_2 z(\mathbf{x}_s(r_0, r_1)))$$

#### Representation II: depthFunction

One would like to be able to provide a depth function  $z = d(x, y)$  which gives a depth as a function of  $(x, y)$ . This is not easily done with the existing Mapping's in Overture, since most Mapping's are parameterized as transformations from the unit square,  $\mathbf{x} = \mathbf{x}(\mathbf{r})$ . Thus instead of a function  $z = d(x, y)$ , we use a parameterized representation,

$$\mathbf{d}(r_0, r_1, r_2) = (d_0(r_0, r_1), d_1(r_0, r_1), d_2(r_0, r_1, r_2)).$$

where only  $d_2(r_0, r_1, r_2)$  is of interest. Now given a point  $(x, y)$  on the original 2D Mapping we want to determine the corresponding value for  $d_2$ . To do this we need a transformation from  $(x, y)$  to the arguments  $(r_0, r_1)$  of  $d_2$ , which we take to be the simple linear transformation  $(x, y) = \mathbf{g}(x, y) = (a_0 + b_0 r_0, a_1 + b_1 r_1)$ .

The volume depth mapping is then defined as

$$\mathbf{x}(r_0, r_1, r_2) = (x_0(r_0, r_1), x_1(r_0, r_1), d_2((x_0(r_0, r_1) - a_0)/b_0, (x_1(r_0, r_1) - a_1)/b_1, r_2))$$

A surface depth Mapping would simply omit the argument  $r_2$ . The scale parameters  $(a_0, b_0, a_1, b_1)$  must be supplied by the user to ensure that the scaled coordinates

$$\tilde{r}_i = (x_i - a_i)/b_i \quad i = 0, 1$$

satisfy  $0 \leq \tilde{r}_i \leq 1$  for all points  $\mathbf{x}_s(r_0, r_1)$ .

Normally the scale parameters just indicate how to scale from the unit square into  $(x, y)$  coordinates,

$$x_i = a_i + b_i r_i$$

So that if the physical domain covers the rectangle  $[-1, 1] \times [-2, 2]$  one should take  $a_0 = -1, b_0 = 2, a_1 = -2, b_1 = 4$  since  $x = -1 + 2 * r_0, y = -2 + 4 * r_1$ .

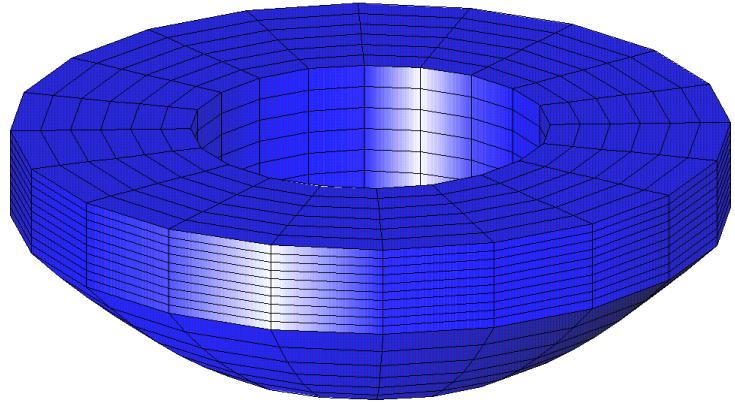
#### 13.1.1 Quadratic depth profile

The quadratic depth function is defined by a quadratic polynomial:

$$z(x_0, x_1) = a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2$$

## 13.2 Examples

```
1  Annulus
2  exit
3  depth mapping
4    quadratic depth
5      -.75 0 0 .5 0. .5
6  exit
```



A DepthMapping starting from an Annulus. The depth is defined by a parabola.

## 13.3 Constructor

**DepthMapping()**

**Description:** Define a 3D Mapping from 2D Mapping by extending in the z-direction by a variable amount

## 13.4 setDepthFunction

```
int
setDepthFunction( Mapping & depth_ )
```

**Description:** Supply a mapping that will define the depth.

**depth\_ (input)** : Use this mapping for the depth,  $z = \text{depth}(x_0, x_1)$ .

## 13.5 setDepthFunction

```
int
setDepthFunctionParameters( real a0, real b0, real a1, real b1 )
```

**Description:** Define the scaling parameters for the depth function.

**a0,b0,a1,b1\_ (input)** :

## 13.6 setSurface

```
int  
setSurface( Mapping & surface_ )
```

**Description:** Supply a 2D mapping that will define the surface of the 3D domain.

**surface\_ (input) :** 2D Mapping.

## 13.7 setQuadraticParameters

```
int  
setQuadraticParameters(const real & a00_,  
                      const real & a10_,  
                      const real & a01_,  
                      const real & a20_,  
                      const real & a11_,  
                      const real & a02_)
```

**Description:** Specify the parameters for a quadratic depth function:

$$z(x_0, x_1) = a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2$$

**a00\_, a10\_,... (input):** parameters in above formula.

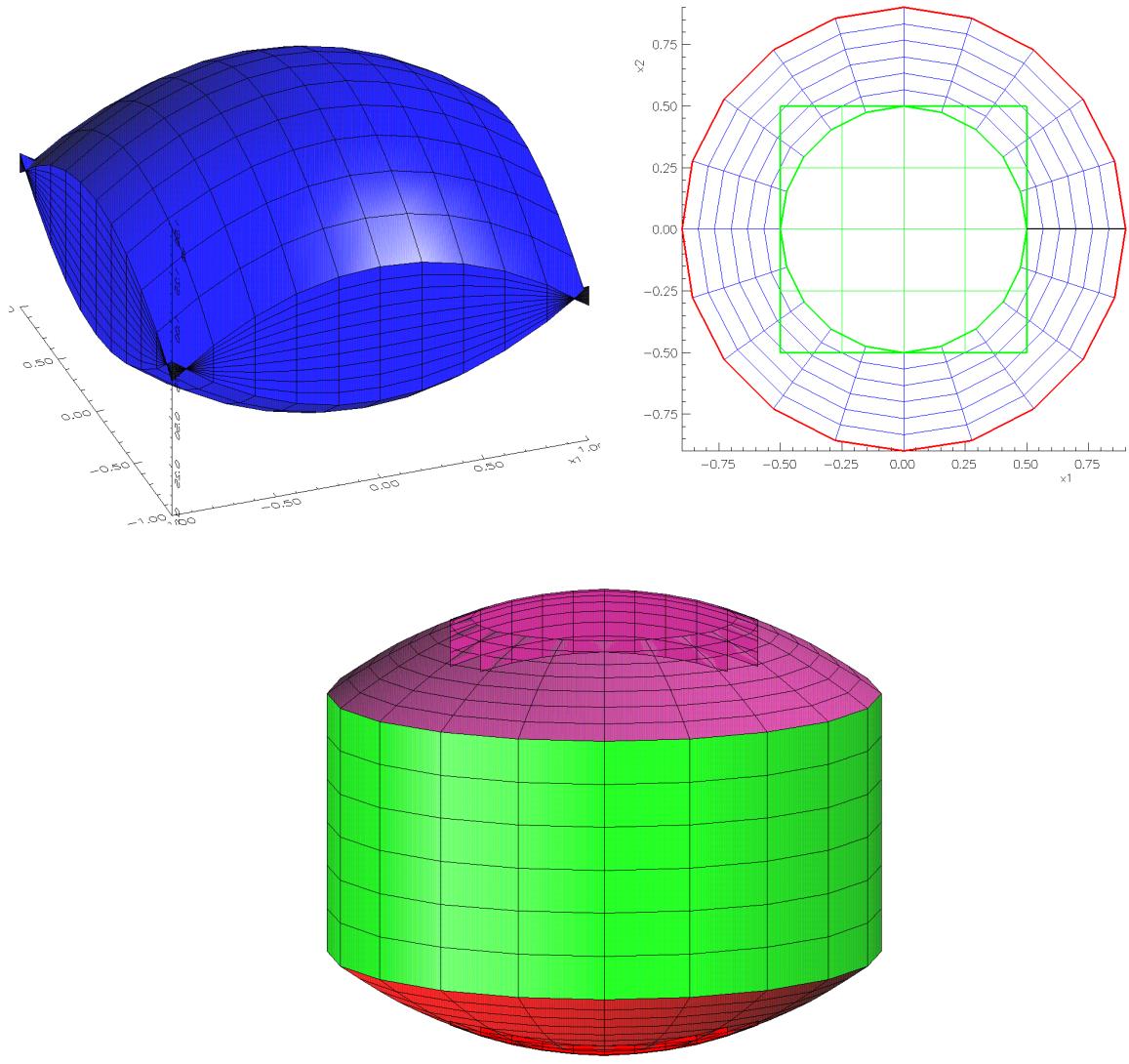


Figure 7: The DepthMapping (see bottom figure) is used to give a vertical dimension to mappings defined in the plane, `depth.cmd`. In this case a separate TFI mapping, top left, defines the vertical height function. Both an annulus and a square (top right) are given a depth.

## 14 Elliptic Transform

This Mapping was originally created by Eugene Sy. Changes made by Bill Henshaw.

\*\*\*\* This documentation is out of date as I have made lots of changes \*\*\*\*

### 14.1 Introduction

The program EllipticTransform.C performs smoothing on a desired mapping by solving a Poisson equation on the domain. A new elliptic mapping is created from the original mapping which is supplied as an initial condition.

The original mapping  $\mathbf{M}$  maps the unit square  $U$  into physical space  $\Omega_p$ . With the elliptic transform, a one to one function  $E'$  taking  $U$  into  $\Omega_p$  is first created. Then,  $\mathbf{M}^{-1}$  is applied to the solution in  $\Omega_p$  to map the points back into the unit square. In this way, a mapping  $\mathbf{D} = \mathbf{M}^{-1}(E')$  is obtained, where  $\mathbf{D} : U \mapsto U$ , and  $\mathbf{D}$  is a data point mapping. The elliptic mapping  $\mathbf{E}$  is then given by a composition, where

$$\mathbf{E}(U) = \mathbf{M}(\mathbf{D}(U))$$

### 14.2 The Governing Equations

The focus from here on will be on the creation of the mapping  $\mathbf{E}$ . Assume that  $U$  has coordinate directions  $\xi$  and  $\eta$ . The equations to be solved are:

$$\begin{aligned}\nabla^2 \xi &= P(\xi, \eta) \\ \nabla^2 \eta &= Q(\xi, \eta)\end{aligned}$$

These are transformed to computational space and take the form:

$$\alpha \left( \frac{\partial^2 x}{\partial \xi^2} + P \frac{\partial x}{\partial \xi} \right) + \beta \left( \frac{\partial^2 x}{\partial \eta^2} + Q \frac{\partial x}{\partial \eta} \right) - 2\gamma \frac{\partial^2 x}{\partial \xi \partial \eta} = 0$$

$$\alpha \left( \frac{\partial^2 y}{\partial \xi^2} + P \frac{\partial y}{\partial \xi} \right) + \beta \left( \frac{\partial^2 y}{\partial \eta^2} + Q \frac{\partial y}{\partial \eta} \right) - 2\gamma \frac{\partial^2 y}{\partial \xi \partial \eta} = 0$$

$$\begin{aligned}\alpha &= \left( \frac{\partial x}{\partial \eta} \right)^2 + \left( \frac{\partial y}{\partial \eta} \right)^2 \\ \beta &= \left( \frac{\partial x}{\partial \xi} \right)^2 + \left( \frac{\partial y}{\partial \xi} \right)^2 \\ \gamma &= \frac{\partial x}{\partial \xi} \frac{\partial x}{\partial \eta} + \frac{\partial y}{\partial \xi} \frac{\partial y}{\partial \eta}\end{aligned}$$

These equations are solved on the unit square  $U$  using finite difference methods.

### 14.3 Control of the Boundary

The existing code has the ability to evaluate Dirichlet, orthogonal, and periodic boundary conditions for solving the above equations.

### 14.3.1 Dirichlet Conditions

If the boundary conditions are Dirichlet the locations of the boundary points must be correctly specified in the initial condition. These points are considered by the elliptic grid generator to be fixed, and computations are done only on the interior of the grid.

### 14.3.2 Orthogonal Boundary Conditions

If the user desires that the gridlines meet the boundary orthogonally, then the points on the boundary are given a degree of freedom, and are allowed to move along the boundary. For example, consider the boundary  $\xi = 0$ . Assume that the  $(n+1)$ st iteration is being computed, and let  $j$  represent the  $\eta$  coordinate. The following two equations are solved to give the location of the boundary points.

$$\mathbf{x}_\eta \cdot \mathbf{x}_\xi = 0 \quad (3)$$

$$\mathbf{x}^{n+1} = \mathbf{x}^n + (\Delta \mathbf{x} \cdot \hat{\mathbf{x}}_\eta) \hat{\mathbf{x}}_\eta \quad (4)$$

Equation 1 is the orthogonality condition, and equation 2 prevents the boundary points from moving off the boundary.  $\Delta \mathbf{x}$  represents the change in position of the point  $\mathbf{x}$  dictated from equation 1. In other words, if  $\mathbf{x}^n$  is the location of the point at the  $n$ th iteration step, equation 1 will steer the point towards a position  $\mathbf{x}^*$  for the  $(n+1)$ st iterate. It follows that

$$\begin{aligned} \Delta \mathbf{x} &= \mathbf{x}^* - \mathbf{x}^n \\ \Delta \mathbf{x} \cdot \hat{\mathbf{x}}_\eta &= \mathbf{x}^* \cdot \hat{\mathbf{x}}_\eta - \mathbf{x}^n \cdot \hat{\mathbf{x}}_\eta \end{aligned}$$

To solve the equations,  $\mathbf{x}_\xi$  in equation 1 is first forward differenced to obtain the following result.

$$x_{0,j}^* x_\eta = y_\xi y_\eta + x_{1,j} x_\eta \quad (5)$$

$$y_{0,j}^* y_\eta = x_\xi x_\eta + y_{1,j} y_\eta \quad (6)$$

This gives an expression for  $\mathbf{x}^* \cdot \hat{\mathbf{x}}_\eta$ , where  $\mathbf{x}^*$  represents the position the boundary point wants to move to in order to satisfy the orthogonality condition.  $\mathbf{x}_\eta$  and all the terms on the right hand side are calculated from the solution at the  $n$ th iterate. Normalizing by  $\|\mathbf{x}_\eta\|$  yields the following:

$$x_{0,j}^* \hat{x}_\eta = y_\xi \hat{y}_\eta + x_{1,j} \hat{x}_\eta \quad (7)$$

$$y_{0,j}^* \hat{y}_\eta = x_\xi \hat{x}_\eta + y_{1,j} \hat{y}_\eta \quad (8)$$

This is  $\mathbf{x}^* \cdot \hat{\mathbf{x}}_\eta$ . Then, since  $\mathbf{x}^n \cdot \hat{\mathbf{x}}_\eta$  is also readily calculated from the  $n$ th iterate, the entire right hand side of equation 2 is known. This allows calculation of  $\mathbf{x}^{n+1}$ , and the continuation of the iteration.

Orthogonality at the boundary can also be obtained by a different means. As explained in Thompson, et al. [3], the boundary points can be fixed, and boundary orthogonality enforced by utilization of the proper forcing functions  $P$  and  $Q$ . In addition, the thickness of the boundary layer can be specified by the user.

Again, let  $\xi = 0$ , as above, and let  $j$  represent the  $\eta$  coordinate. Assume that the boundary layer thickness  $\|\mathbf{x}_\xi\|$  is chosen, and calculate  $\mathbf{x}_\eta$  and  $\mathbf{x}_{\eta\eta}$  from the fixed locations of the boundary points. All this, together with the orthogonality condition

$$\mathbf{x}_\eta \cdot \mathbf{x}_\xi = 0$$

allows for the determination of  $\mathbf{x}_\xi$ . As shown in Knupp and Steinberg[2].  $\mathbf{x}_\xi$  is given by:

$$\mathbf{x}_\xi = \frac{\|\mathbf{x}_\xi\|}{\|\mathbf{x}_\eta\|} \mathbf{x}_\eta^\perp \quad (9)$$

Here,  $\mathbf{x}_\eta^\perp$  is the vector perpendicular to the vector  $\mathbf{x}_\eta$ . In addition to this, the method requires  $\mathbf{x}_{\xi\xi}$ , and this is calculated by means of the Pade approximation

$$\mathbf{x}_{\xi\xi}|_0 = \frac{-7\mathbf{x}_{1,j} + 8\mathbf{x}_{1,j} - \mathbf{x}_{3,j}}{2\Delta\eta^2} - 3\frac{\mathbf{x}_\xi|_0}{\Delta\eta} \quad (10)$$

Using this information, the proper forcing functions  $P$  and  $Q$  can be determined.

$$\begin{aligned} P(\xi, \eta) &= -\frac{\mathbf{x}_\xi \cdot \mathbf{x}_{\xi\xi}}{\|\mathbf{x}_\xi\|^2} - \frac{\mathbf{x}_\xi \cdot \mathbf{x}_{\eta\eta}}{\|\mathbf{x}_\eta\|^2} \\ Q(\xi, \eta) &= -\frac{\mathbf{x}_\eta \cdot \mathbf{x}_{\eta\eta}}{\|\mathbf{x}_\eta\|^2} - \frac{\mathbf{x}_\eta \cdot \mathbf{x}_{\xi\xi}}{\|\mathbf{x}_\xi\|^2} \end{aligned}$$

Once the appropriate  $P$  and  $Q$  are determined for the boundary, the values are interpolated onto the interior points using a linear scaling, and the iteration is continued.

Note that because second order differences are being lagged, heavy underrelaxation is required for this scheme to converge (Knupp and Steinberg [2]).

#### 14.3.3 Periodic Boundaries

Two types of periodic boundaries exist. The first is derivative periodic and the second is function periodic. Derivative periodicity involves identical derivatives on the boundary, but not necessarily identical positions. Function periodicity involves matching both the derivatives and the positions at the boundary points (as in the case of an annulus).

In either case, the values beyond the boundary are evaluated by means of ghost points. A ghost array allows for calculation of values on the boundary in much the same way they are found on the interior.

#### 14.4 Sources

Should clustering of points or lines be necessary in the interior, certain points or lines may be designated as being lines of attraction. This involves manipulation of the source terms  $P$  and  $Q$  before iteration begins. If a coordinate line is to be made a line of attraction, the attraction power  $\pi$  must be specified along with the diffusivity  $\delta$ . With these two parameters, the following expressions for  $P$  and  $Q$  are evaluated at all points in the field.

$$\begin{aligned} P_{line}(\xi, \eta) &= -\sum_{i=1}^N \pi_i \text{Sign}(\xi - \xi_i) e^{-\delta_i |\xi - \xi_i|} \\ Q_{line}(\xi, \eta) &= -\sum_{j=1}^M \pi_j \text{Sign}(\eta - \eta_j) e^{-\delta_j |\eta - \eta_j|} \end{aligned}$$

Here,  $N$  and  $M$  represent the number of  $\xi$  and  $\eta$  lines of attraction respectively. Should points of attraction be desired, power and diffusivity are specified for each point, and two more sums are evaluated.

$$\begin{aligned} P_{point}(\xi, \eta) &= -\sum_{i=1}^L \pi_i \text{Sign}(\xi - \xi_i) e^{-\delta_i |\xi - \xi_i|} \\ Q_{point}(\xi, \eta) &= -\sum_{j=1}^L \pi_j \text{Sign}(\eta - \eta_j) e^{-\delta_j |\eta - \eta_j|} \end{aligned}$$

Here,  $L$  is the number of point sources. Note that a source can be made into a sink by merely changing the sign in front of the power  $\pi$ .

## 14.5 Using the Elliptic Grid Generator With Ogen

The user is assumed to be familiar with generation of mappings in Ogen. A mapping must first be made as an initial condition for the elliptic smoother.

### 14.5.1 Grid Dimensions

The first thing to specify after choosing a mapping is the amount of grid refinement desired. The number of grid lines in i and j (or  $\xi$  and  $\eta$  respectively) should be entered before all else, as these parameters are needed for correct implementation of the boundary conditions.

### 14.5.2 Boundary Conditions

The appropriate GRID boundary conditions should be entered next. These are not to be confused with the boundary conditions for the physical problem to be solved later, and the switches are completely independent. The following choices are available:

- -1: Refers to a **periodic** boundary condition. This is selected by default if a periodic boundary is declared when the original mapping is made. If a periodic boundary is not specified when the original mapping was created, this boundary condition should not be used.
- 1: Refers to a **Dirichlet** boundary condition. The positions of the boundary points in the original mapping are used as the boundary condition for the elliptic map.
- 2: Refers to a **orthogonal** boundary condition. This forces the gridlines to meet the boundary in an orthogonal fashion. The boundary points are free to move, but only along the boundary.
- 3: Refers to the **combined** boundary condition. The boundary points are fixed as in the Dirichlet case, but the sources and sinks are modified so as to guarantee orthogonality at the boundary. This boundary condition requires that the user specify the thickness of the boundary layer.

The boundary conditions are stored in a 2-dimensional array **gridBc(i,j)**, where i and j range from 0 to 1. On the unit square, the following are the locations of the boundaries:

- **gridBc(0,0)**: Refers to the boundary condition on  $0 \leq x \leq 1, y = 0$ .
- **gridBc(0,1)**: Refers to the boundary condition on  $x = 0, 0 \leq y \leq 1$ .
- **gridBc(1,0)**: Refers to the boundary condition on  $0 \leq x \leq 1, y = 1$ .
- **gridBc(1,1)**: Refers to the boundary condition on  $x = 1, 0 \leq y \leq 1$ .

### 14.5.3 Sources and Sinks

If the user decides that lines or points of attraction (repulsion) are desired for the elliptic grid, these can be specified. The selection *Poisson i-line sources* creates lines of constant  $\xi$  into lines of attraction. Similarly, *Poisson j-line sources* turns lines of constant  $\eta$  into lines of attraction, and *Poisson point sources* creates points of attraction in  $\xi - \eta$  space.

The power and diffusivity of each source must be selected carefully. Too much power or too little diffusivity can mar convergence of the grid.

#### 14.5.4 Other Functions

There are several other switches that can be used. These are:

- *change SOR parameter*: This changes the value of  $\omega$  used for either overrelaxation or underrelaxation. Setting  $\omega = 1$  indicates a point Gauss- Seidel method, and is a good conservative first choice. Note that if **combined** boundary conditions are used, then  $\omega$  needs to be quite small, usually on the order of 0.05 or 0.1. This is because of the lagged second order quantities that the scheme requires.
- *set maximum number of iterations*: This controls the maximum amount of iterations for the grid to converge.
- *set epsilon for convergence*: This value is the indicator for the convergence of the elliptic grid. If the differences between successive iterations drops below epsilon, then the grid is said to have converged. This is preset to  $10^{-5}$ .
- *set number of periods*: If the grid is periodic (either derivative or function), then any sources or sinks in the field need to be made periodic as well. The number entered here indicates the number of periods desired for these sources and sinks. 1 is the default value. Increasing this number is crucial if strong source terms exist, and 5 or 7 may be needed to properly resolve the periodicity.

### 14.6 In Conclusion

The elliptic transform is a fine way to smooth out a grid, and frees the user from the restriction to simple geometries. Unfortunately, convergence of the routine is not guaranteed for all geometries, and for all powers of sources and sinks.

The routine, however, does provide an ability to deal with boundaries well, and can greatly simplify many complex computations.

### 14.7 Member functions

#### 14.7.1 Constructor

**EllipticTransform()**

**Purpose:** Create a mapping that can be used to generate an elliptic grid from an existing grid. This can be useful to smooth out an existing Mapping.

#### 14.7.2 get

```
int  
get( const GenericDataBase & dir, const aString & name)
```

**Description:** Get a mapping from the database.

**dir (input):** get the Mapping from a sub-directory of this directory.

**name (input) :** name of the sub-directory to look for the Mapping in.

#### 14.7.3 put

```
int  
put( GenericDataBase & dir, const aString & name) const
```

**Description:** Save a mapping into a database.

**dir (input):** put the Mapping into a sub-directory of this directory.

**name (input) :** name of the sub-directory to save the Mapping in.

#### 14.7.4 generateGrid

**void**

```
generateGrid(GenericGraphicsInterface *gi = NULL,  
GraphicsParameters & parameters =Overture::nullMappingParameters())
```

**Description:** This function performs the iterations to solve the elliptic grid equations.

**gi (input) :** supply a graphics interface if you want to see the grid as it is being computed.

**parameters (input) :** optional parameters used by the graphics interface.

### 14.8 Examples

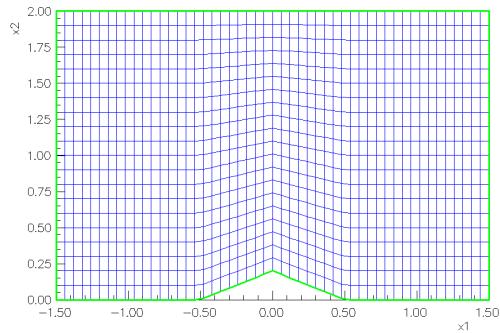
#### 14.8.1 Smoothed out diamond airfoil

In the left column is the command file that was used to generate the grid on the bottom right.

```

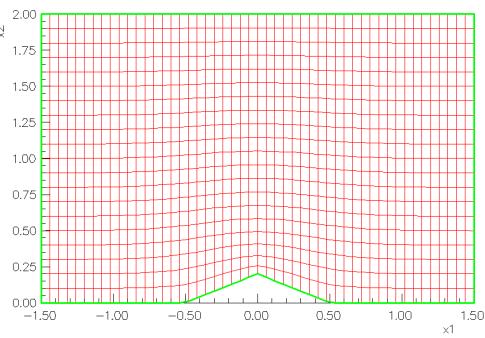
1  *
2  * Smooth a diamond airfoil with the
3  * elliptic transform mapping.
4  *
5  * first make a diamond airfoil
6  *
7  Airfoil
8    airfoil type
9      diamond
10   thickness-chord ratio
11     .2
12   lines
13     51 21
14 exit
15 *
16 * now smooth the diamond airfoil
17 *
18 elliptic
19 *
20  * do not project back onto the original mapping
21  * since it has a discontinuity
22 *
23 project onto original mapping (toggle)
24 *
25  * now generate the elliptic transform:
26 *
27 elliptic smoothing
28 exit

```



Diamond airfoil before elliptic transform.

elliptic-airfoil



Diamond airfoil after elliptic transform.

## 15 FilletMapping

This mapping can be used to create a fillet grid or a collar grid in order to join together two intersecting surfaces. A fillet grid smooths out the intersection while the collar grid does not.

This mapping will automatically compute the fillet given two intersecting surfaces. Various parameters control the resulting surface:

**orientation:** There are 4 possible quadrants in which to place the fillet.

**width:** This distance defines the width over which the fillet blends between the two surfaces and thus determines how rounded the fillet is. A width of zero will result in a collar grid which has a corner in it.

**overlapWidth:** Determines the distance to which the fillet extends onto each surface once it has touched the surface.

### 15.1 Description of Approach

Here are the basic steps that are used to create a fillet or collar grid:

**intersect surfaces:** Given two intersecting surfaces we first compute the curves(s) of intersection using the `IntersectionMapping`. (For a 2D fillet grid the intersection curves are just the points of intersection).

**generate surface grids:** The next step is to generate a hyperbolic surface grid on each of the two surfaces, using the curve of intersection as a starting curve (not necessary to do in 2D). The surface grid is grown in both directions from the starting curve. The `HyperbolicSurfaceMapping` is used to generate these grids.

**blend surface grids:** The fillet grid is defined as a blending of the two surface grids. The precise description of this blending is given below.

### 15.2 Fillet for two intersecting surfaces

To define a fillet to join two intersecting surfaces,  $\mathbf{S}_1$ ,  $\mathbf{S}_2$  we use

$$\begin{aligned}\mathbf{c}_I(r_1) &= \text{Curve of intersection} \\ \mathbf{c}_1(r_1, r_2) &= \text{Grid on surface 1, with } \mathbf{c}_1(r_1, .5) = \mathbf{c}_I(r_1) \\ \mathbf{c}_2(r_1, r_2) &= \text{Grid on surface 2, with } \mathbf{c}_2(r_1, .5) = \mathbf{c}_I(r_1)\end{aligned}$$

If the parameter  $r_1$  is tangential to the intersection and  $r_2$  varies in the direction normal to the intersection then the fillet is defined by blending the two surface grids:

$$\begin{aligned}\mathbf{x} &= b(s)\mathbf{c}_1(r_1, s_1(r_2)) + (1 - b(s))\mathbf{c}_2(r_1, s_2(r_2)) \\ b &= \frac{1}{2}(1 + \tanh(\beta(r_2 - .5)))\end{aligned}$$

where the parameter variables  $s_i(r_2)$  are chosen to be quadratic polynomials in  $r_2$ ,

$$s_i = a_{i0}(r_1) + r_2(a_{i1}(r_1) + r_2a_{i2}(r_1))$$

where

$$\begin{aligned} c_{i,0} &= .5 && \text{intersection point} \\ c_{i,1} &= c_{i,0} - pm[i] * .5 * filletWidth/crNorm && \text{distance from intersection point for c1} \\ c_{i,2} &= c_{i,0} - pm[i] * (.5 * filletWidth + filletOverlap)/crNorm && \text{distance from intersection point for c2} \\ c_{i,3} &= c_{i,0} + pm[i] * shift * .5 * filletWidth/crNorm \\ a_{i0} &= c_{i,2+i} \\ a_{i1} &= c_{i,3-i} - c_{i,2+i} + (16./3.) * (c_{i,1} - .75 * c_{i,2} - .25 * c_{i,3}) \\ a_{i2} &= -((16./3.) * (c_{i,1} - .75 * c_{i,2} - .25 * c_{i,3})) \end{aligned}$$

### 15.3 setCurves

```
int  
setCurves(Mapping & curve1,  
          Mapping & curve2)
```

**Description:** Supply the curves or surfaces from which the fillet will be defined.

**curve1, curve2 (input):**

### 15.4 map

```
void  
map( const realArray & r, realArray & x, realArray & xr, MappingParameters & params )
```

**Purpose:** Evaluate the mapping and/or derivatives.

### 15.5 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively create and/or change the Fillet mapping.

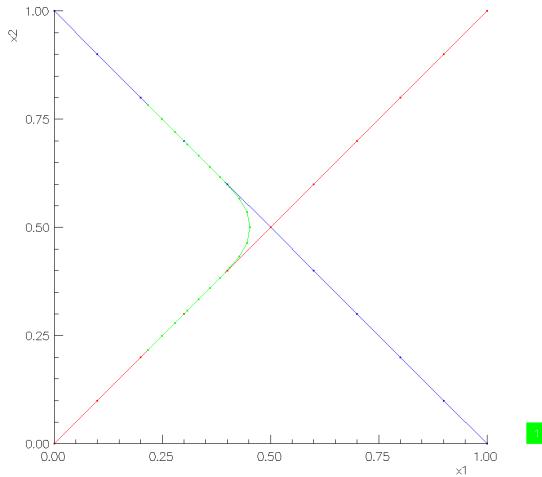
**mapInfo (input):** Holds a graphics interface to use.

### 15.6 examples

#### 15.6.1 2D Fillet joining two lines

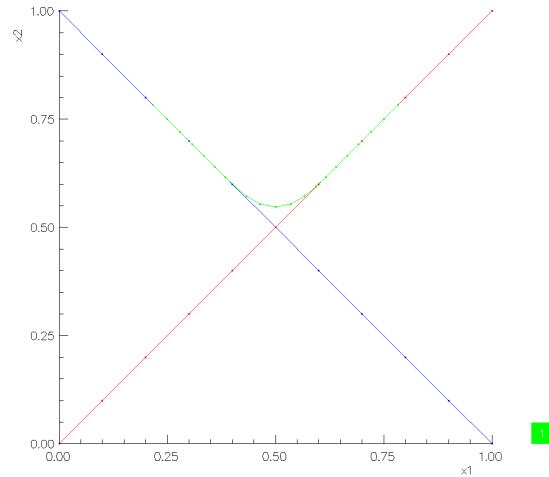
This is a 2D example showing a fillet that joins two line segments. These figures show the four possible fillets that can be generated between intersecting curves (or surfaces).

FilletMapping



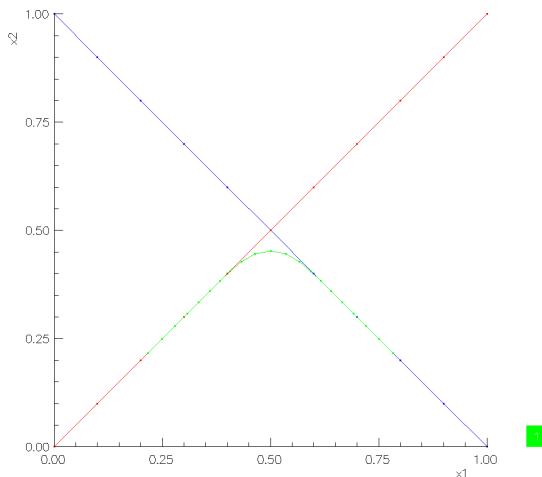
A fillet grid joining two lines. The orientation  
is **curve 1- to curve 2-**.

FilletMapping



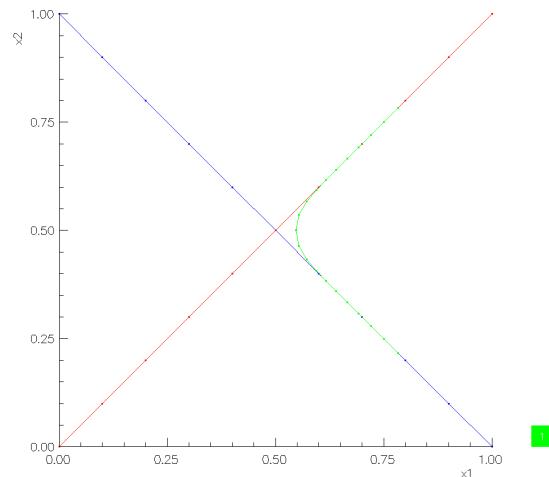
A fillet grid joining two lines. The orientation  
is **curve 1+ to curve 2-**.

FilletMapping



A fillet grid joining two lines. The orientation  
is **curve 1- to curve 2+**.

FilletMapping



A fillet grid joining two lines. The orientation  
is **curve 1+ to curve 2+**.

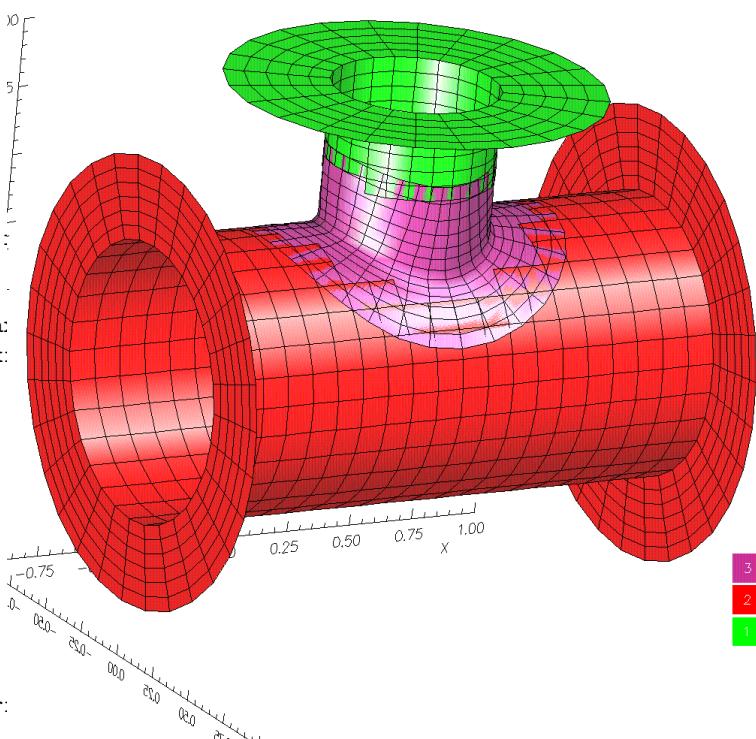
### 15.6.2 Fillet to join two cylinders

In the left column is the command file that was used to generate the grid on the right.

```

1   Cylinder
2     orientation
3       1 2 0
4     bounds on the axial variable
5       -1. 1.
6     bounds on the radial variable
7       .5 .75
8     boundary conditions
9       -1 -1 1 2 3 0
10    mappingName
11      main-cylinder
12    lines
13      31 21 6
14    exit
15 Cylinder
16   mappingName
17     top-cylinder
18   orientation
19     2 0 1
20   bounds on the axial variable
21     .25 1.
22   bounds on the radial variable
23     .3 .6
24   boundary conditions
25     -1 -1 0 2 3 0
26   lines
27     25 15 5
28   exit
29 *
30 fillet
31   * define more lines for computing the :
32   lines
33     81 41 41 21
34   Start curve 1:main-cylinder (side=0,ax:
35   Start curve 2:top-cylinder (side=0,ax:
36   orient curve 1+ to curve 2-
37   compute fillet
38 *
39   pause
40 exit
41 * build a volume grid around the fillet
42 hyperbolic
43   grow grid in opposite direction
44   distance to march .2
45   points on initial curve 31 12
46   lines to march 7
47   uniform dissipation .1
48   outward splay .25 .25 .25 .25 (left,r:
49   * show parameters
50   BC: bottom outward splay
51   BC: top outward splay
52   generate
53
54   lines
55     31 12 6
56   mappingName
57     cylinderFillet
58   share
59     0 0 0 0 0 0
60
61   exit
62
63
64
65 choose curves
66   main-cylinder (side=0,axis=2)
67   top-cylinder (side=0,axis=2)
68   orient curve 1- to curve 2+
69   compute

```

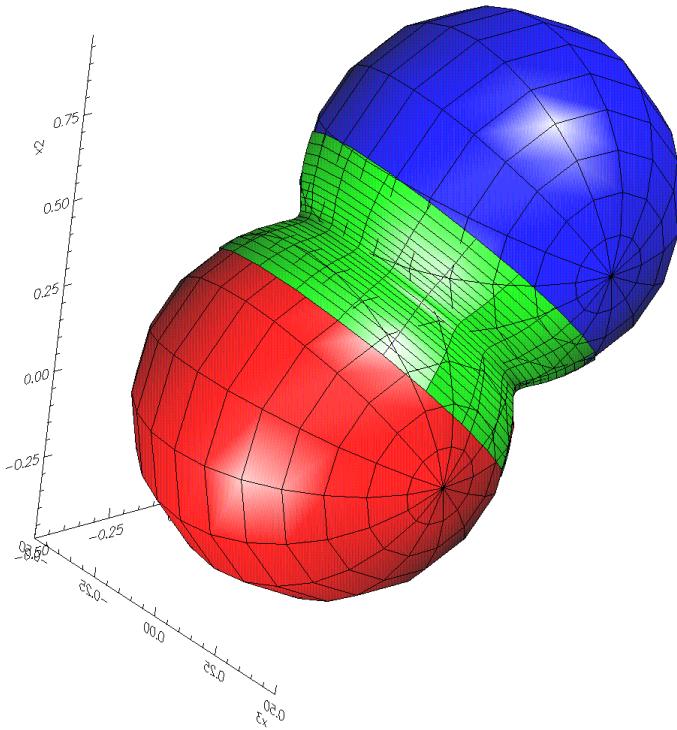


A fillet grid joining two cylinders. The fillet is created with the aid of hyperbolic grid generation.

### 15.6.3 Fillet to join two spheres

In the left column is the command file that was used to generate the grid on the right.

```
1 * build a sphere
2 Sphere
3   surface or volume (toggle)
4   mappingName
5     sphere1
6 exit
7 * build a second sphere
8 Sphere
9   surface or volume (toggle)
10  mappingName
11    sphere2
12  centre for sphere
13    .5 .5
14 exit
15 * build the fillet
16 fillet
17   Start curve 1:sphere1
18   Start curve 2:sphere2
19   orient curve 1+ to curve 2-
20   compute fillet
21
22
23 choose curves
24 sphere1
25 sphere2
26 orient curve 1+ to curve 2-
27 compute fillet
```



A fillet grid (green) joining two spheres.

## 15.7 HyperbolicMapping

See the separate document *The Overture Hyperbolic Grid Generator* [1].

# 16 IntersectionMapping

This mapping class can compute the intersection between two other mappings, such as the curve of intersection between two surfaces. See the comments with the `determineIntersection` function for a description of the fairly robust way in which we find the intersection.

## 16.1 Constructor

**IntersectionMapping()**

**Purpose:** Default Constructor

**Author:** WDH

## 16.2 Constructor

**IntersectionMapping(Mapping & map1\_-,  
                  Mapping & map2\_- )**

**Purpose:** Define a mapping for the intersection of map1\_- and map2\_-

**map1\_-, map2\_- :** two surfaces in 3D

## 16.3 intersect

```
int  
intersect(Mapping & map1_-, Mapping & map2_-,  
            GenericGraphicsInterface *gi =NULL,  
            GraphicsParameters & params =nullGraphicsParameters)
```

**Description:** Determine the intersection between two mappings, optionally supply graphic parameters so the intersection curves can be plotted, (for debugging purposes). NEW FEATURE: If the intersection curve has disjoint segments, these segments will be stored as sub curves in the NURBS for the physical and parameter curves on each surface.

**map1\_-, map2\_- (input) :** These two mappings will be intersected.

**gi, params (input) :** Optional parameters for graphics.

**Return value:** 0 for success

## 16.4 intersect

```
int  
intersectWithCompositeSurface(Mapping & map1_-, CompositeSurface & cs,  
                                  GenericGraphicsInterface *gi =NULL,  
                                  GraphicsParameters & params =nullGraphicsParameters)
```

**Description:** A Protected routine that computes the intersection between a Mapping and a CompositeSurface.

**map1\_**, **map2\_** (**input**) : These two mappings will be intersected.

**gi**, **paramas** (**input**) : Optional parameters for graphics.

**Return value:** 0 for success

**Output:** The output intersection curve is a NurbsMapping. The number of subcurves of this mapping defines the number of disconnected components of the intersection.

## 16.5 newtonIntersection

**int**

**newtonIntersection(realArray & x, realArray & r1, realArray & r2, const realArray & n )**

**Description:** This is a protected routine to determine the exact intersection point on two surfaces using Newton's method.

Solve for (x,r1,r2 ) such that

```
map1(r1) - x = 0  
map2(r2) - x = 0  
n.x = c
```

**x(.,3)** (**input/output**) : initial guess to the intersection point (in the Range space)

**r1(.,2)** (**input/output**): initial guess to the intersection point (in the domain space of map1)

**r2(.,2)** (**input/output**): initial guess to the intersection point (in the domain space of map2)

**n(.,3)** : a normal vector to a plane that crosses the intersection curve, often choosen to be  $n(i,.) = x(i+1,.) - x(i-1,.)$  if we are computing  $x(i,.)$

**Return values:** 0 for success. 1 if the newton iteration did not converge, 2 if there is a zero normal vector.

## 16.6 project

**int**

```
project( realArray & x,  
         int & iStart,  
         int & iEnd,  
         periodicType periodic)
```

**Description:** Project the points  $x(iStart:iEnd,0:6)$  onto the intersection NOTE: When the points are projected onto the curves it is possible that points fold back on themselves if they get out of order.

This routine will try and detect this situation and it may remove some points to fix the problem.

Return values: 0 for success, otherwise failure.

## 16.7 determineIntersection

**int**

```
determineIntersection(GenericGraphicsInterface *gi =NULL,  
                      GraphicsParameters & params =nullGraphicsParameters)
```

**Description:** This is a protected routine to determine the intersection curve(s) between two surfaces.

**Notes:** (1) First obtain an initial guess to the intersection: Using the bounding boxes that cover the surface to determine a list of pairs of (leaf) bounding boxes that intersect. Triangulate the surface quadrilaterals that are found in this "collision" list and find all line segments that are formed when two triangles intersect.

(2) Join the line segments found in step 1 into a continuous curve(s). There will be three different intersection curves – a curve in the Range space ( $x$ ) and a curve in each of the domain spaces ( $r$ ). Since the domain spaces may be periodic it may be necessary to shift parts of the domain-space curves by +1 or -1 so that the curves are continuous. Note that the domain curves will sometimes have to be outside the unit square. It is up to ?? to map these values back to [0,1] if they are used.

(3) Now fit a NURBS curve to all of the intersection curves, using chord-length of the space-curve to parameterize the three curves. (4) Re-evaluate the points on the curve using Newton's method to obtain the points that are exactly on the intersection of the surfaces. Refit the NURBS curves using these new points.

**Return values:** 0 for success, otherwise failure.

## 16.8 map

```
int  
reparameterize(const real & arcLengthWeight =1.,  
               const real & curvatureWeight =.2)
```

**Purpose:** Redistribute points on the intersection curve to place more points where the curvature is large.

**Description:** The default distribution of points in the intersection curve is equally spaced in arc length (really chord length). To cluster more points near sharp corners, call this routine with a non-zero value for `curvatureWeight`. In this case the points will be placed to equidistribute the weight function

```
w(r) = 1 + arcLength(r)*arcLengthWeight + curvature(r)*curvatureWeight  
where  
arcLength(r) = | x_r |  
curvature(r) = | x_rr |    (** this is not really the curvature, but close ***)
```

Note that the point distribution only depends on the ratio of `arcLengthWeight` to `curvatureWeight` and not on their absolute values. The weight function must be positive everywhere. Also note that for the unit circle,  $|x_r| = 2\pi$  and  $|x_{rr}| = (2\pi)^2$  so that the curvature is naturally  $2\pi$  times larger in the weight function.

**arcLengthWeight (input)** : weight for the arc length, should be positive.

**curvatureWeight (input)** : weight for the curvature, should normally be non-negative.

## 16.9 intersectCurves

```
int  
intersectCurves(Mapping & curve1,  
                  Mapping & curve2,  
                  int & number_of_intersection_points,
```

```
realArray & r1,  
realArray & r2,  
realArray & x )
```

**Description:** Determine the intersection between two 2D curves.

**curve1, curve2 (input) :** intersect these curves

**numberOfIntersectionPoints (output):** the number of intersection points found.

**r1,r2,x (output) :** r1(i),r2(i),x(0:1,i)      the      intersection      point(s)      for       $i = 0, \dots, \text{numberOfIntersectionPoints} - 1$  are  $\text{curve1}(r1(i)) = \text{curve2}(r2(i)) = x(i)$

## 16.10 map

**void**

```
map( const realArray & r, realArray & x, realArray & xr, MappingParameters & params )
```

**Purpose:** Evaluate the intersection curve.

## 16.11 get

**int**

```
get( const GenericDataBase & dir, const aString & name)
```

**Purpose:** get a mapping from the database.

## 16.12 put

**int**

```
put( GenericDataBase & dir, const aString & name) const
```

**Purpose:** put the mapping to the database.

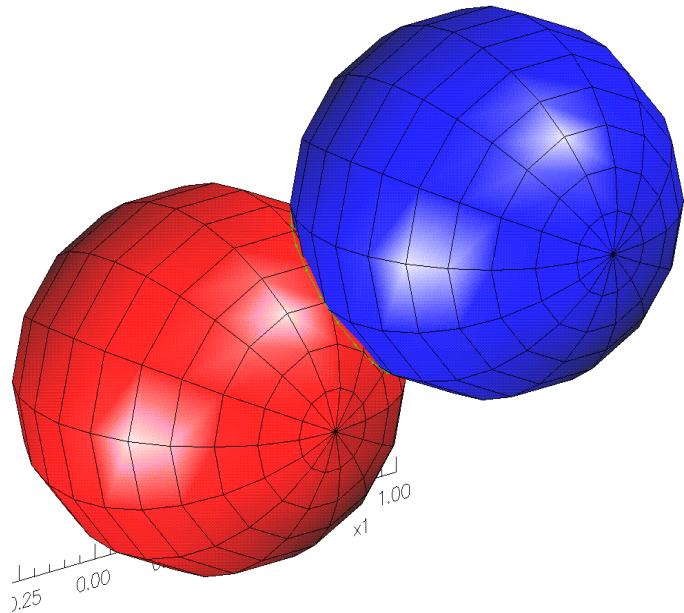
## 16.13 update

**int**

```
update( MappingInformation & mapInfo )
```

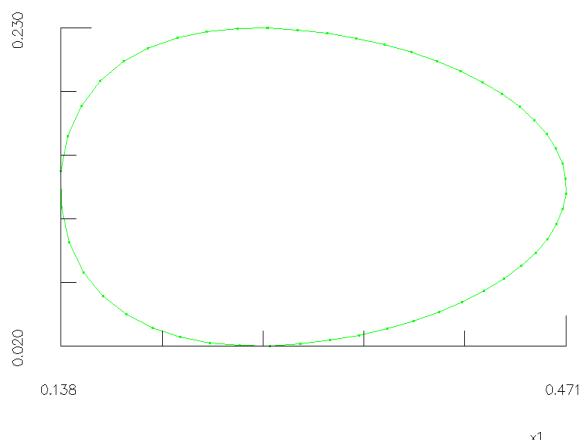
**Purpose:** Interactively create and/or change the mapping.

**mapInfo (input):** Holds a graphics interface to use.

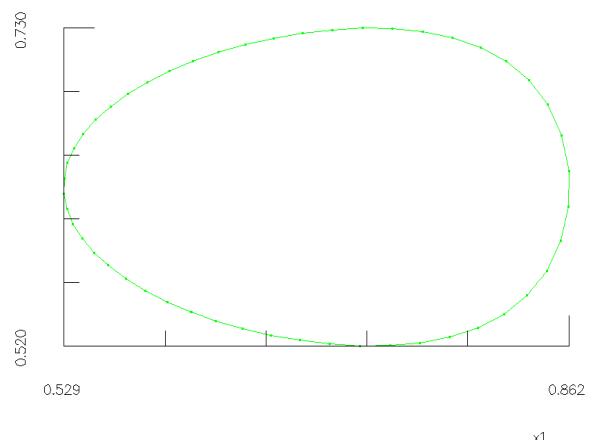


parametric curve rCurve1

parametric curve rCurve2

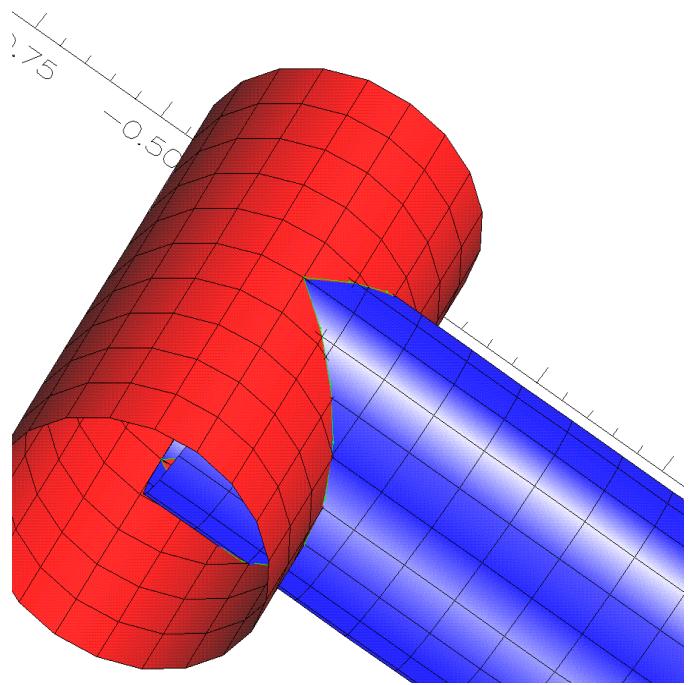


(a)



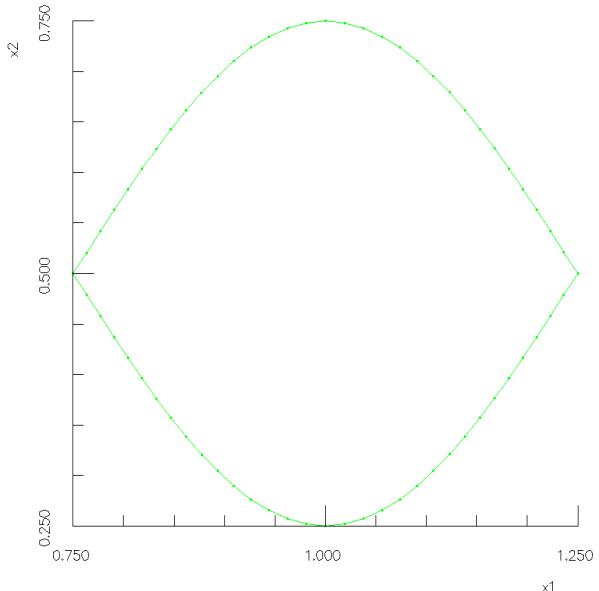
(b)

Figure 8: The sphere-sphere intersection curve in the range space and the domain spaces (unit squares).

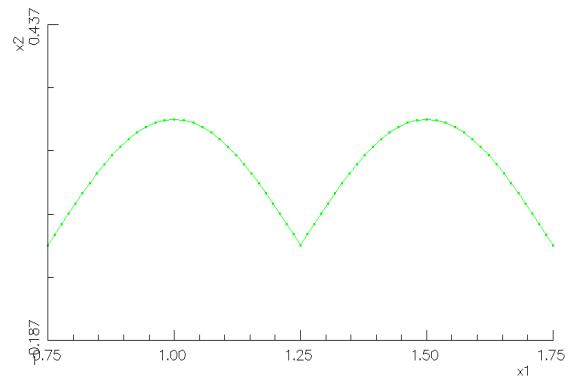


parametric curve rCurve1

parametric curve rCurve2

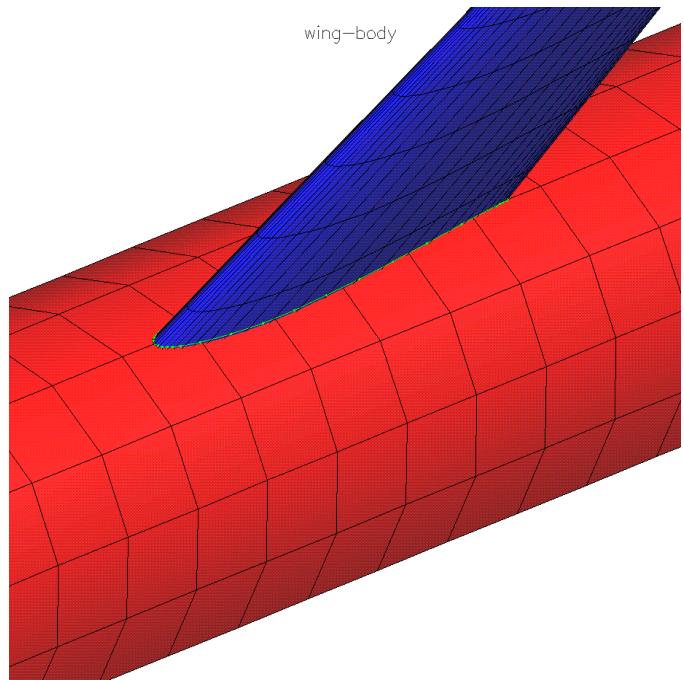


(a)



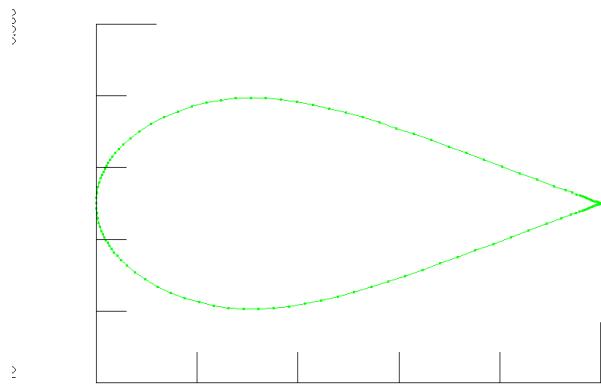
(b)

Figure 9: The cylinder-cylinder intersection curves in the range space and the domain spaces (unit squares). This is a reasonably hard case since the cylinders have the same radius and thus the surfaces are tangent at two points.



parametric curve rCurve1

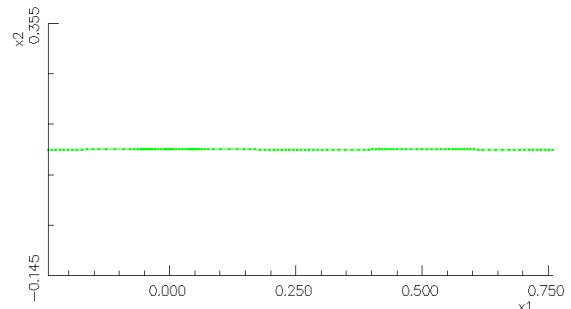
parametric curve rCurve2



0.416

0.584

(a)



(b)

Figure 10: Intersection curve for a wing-body configuration. The curve was reparameterized weighting arclength and curvature in order to redistribute more points to the high curvature regions.

## 17 JoinMapping

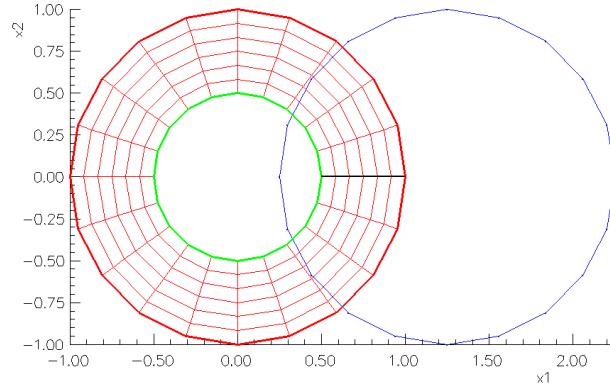
This mapping can be used to join together two Mappings that intersect. This is an alternative way to the **FilletMapping** to connect two intersecting surfaces.

The prototypical example of the use of a **JoinMapping** is the intersection of a wing (the source mapping, i.e. the mapping that will be changed) with a fuselage (the *clip-surface* mapping). If the end of the wing does not match exactly to the fuselage, there will be a part of the wing that extends inside the fuselage. The **JoinMapping** can be used to remove the part of the wing that is inside the fuselage and reparameterize the rest of the wing so that the new wing matches exactly to the fuselage.

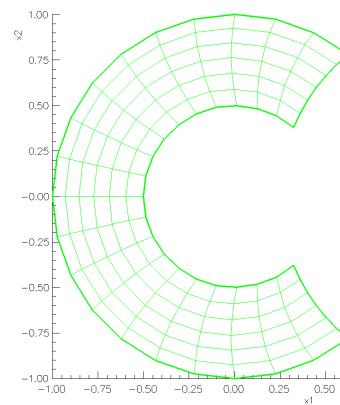
### 17.1 A 2D example

In this first example we consider an annulus (source mapping) that intersects a circle (the clip surface). We generate a new mapping that consists of the portion of the annulus that lies outside the circle. (We could also choose the part of the annulus that lies inside the circle).

```
1 * join an annulus to a circle
2 Annulus
3   exit
4 Circle or ellipse
5   specify centre
6     1.25 0
7   exit
8 join
9   choose curves
10 Annulus
11 circle
12 compute join
```



JoinMapping



A new mapping (bottom) is generated that replaces the annulus by a partial annulus that exactly matches to the circle.

## 17.2 Intersecting surfaces

Consider the case where the *source mapping* and the *clip-surface* Mappings are both surfaces in 3D. The **JoinMapping** will first compute the curve of intersection between the *source* and the *clip-surface* mappings. The curve of intersection, as generated by the **IntersectionMapping**, will have three representations:

- A space curve  $\mathbf{x}_i(s)$ ,  $s \in [0, 1]$ . matching the curve of intersection in physical space.
- A curve in parameter space of the source-mapping,  $\mathbf{r}_i(s)$ ,  $s \in [0, 1]$ , that is the pre-image of the space curve  $\mathbf{x}_i(s)$ . Thus if  $\mathbf{x} = \mathbf{C}_i(\mathbf{r})$  denotes the source mapping, then  $\mathbf{x}_i(s) = \mathbf{C}_i(\mathbf{r}_i(s))$ .
- There is also a parametric curve for the clip-surface mapping,  $\mathbf{r}_e(s)$ , with  $\mathbf{x}_i(s) = \mathbf{C}_e(\mathbf{r}_i(s))$ , where  $\mathbf{C}_e(\mathbf{r})$  is the clip-surface mapping.

To reparameterize the source mapping we first define a new mapping in the parameter space of the source-mapping that is bounded on one side by the parametric intersection curve  $\mathbf{r}_i$ . In the typical case this new mapping can be defined by trans-finite interpolation (**TFIMapping**), such as

$$\mathbf{P}(\mathbf{r}) = (1 - r_2)\mathbf{r}_i(r_1) + r_2(r_1, 1)$$

In this case the curve  $\mathbf{r}_i$  is assumed to be mainly in the  $r_1$  direction and we have chosen to extend the patch to  $r_2 = 1$  (we could also choose to extend to  $r_2 = 0$  or some other value of  $r_2$  (see the **setEndOfJoin** function)).

The **JoinMapping** is now defined by the composite Mapping,

$$\mathbf{J}(\mathbf{r}) = \mathbf{C}_i(\mathbf{P}(\mathbf{r}))$$

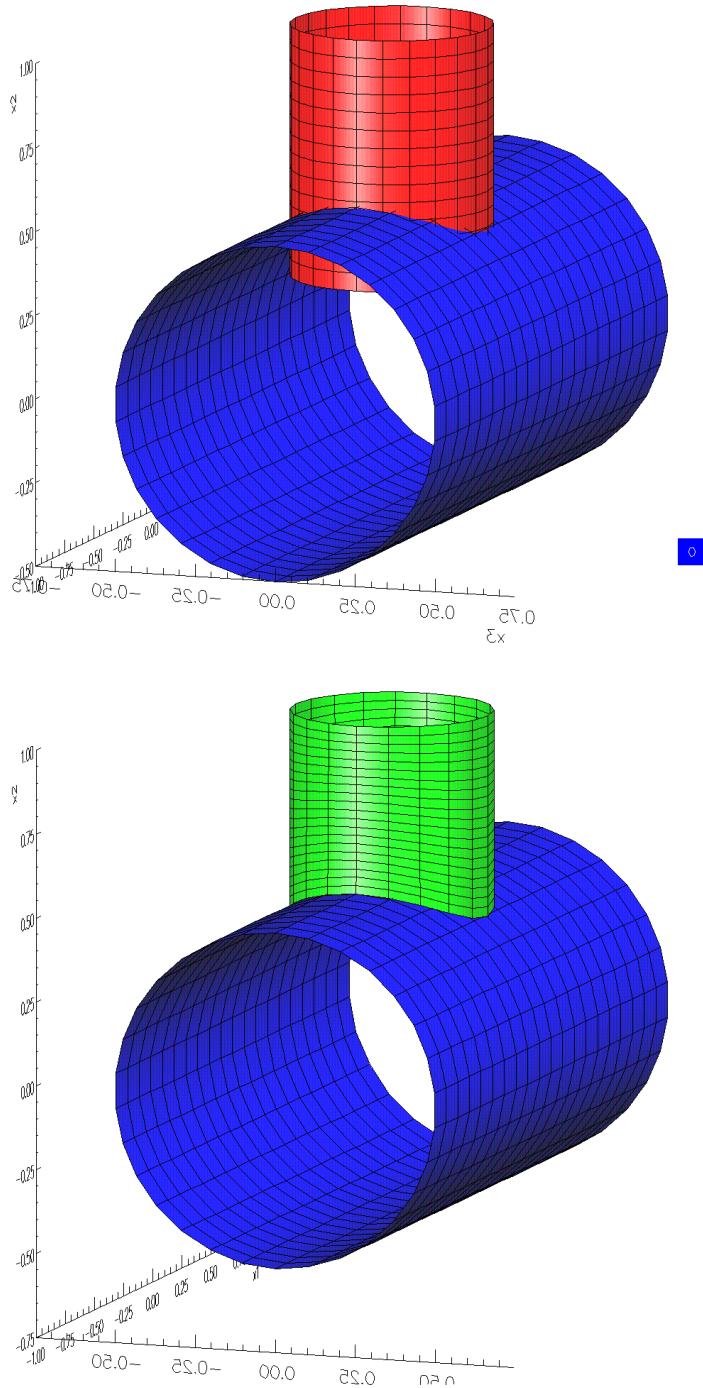
Through the definitions we see that  $\mathbf{J}(\mathbf{r})$  will exactly match the curve of intersection at  $r_2 = 0$ ,

$$\mathbf{J}(r_1, 0) = \mathbf{C}_i(\mathbf{P}(r_1, 0)) = \mathbf{C}_i(\mathbf{r}_i(r_1)) = \mathbf{x}_i(r_1)$$

```

1   Cylinder
2     orientation
3       1 2 0
4     bounds on the axial variable
5       -1. 1.
6     bounds on the radial variable
7       .5 .75
8     boundary conditions
9       -1 -1 0 0 3 0
10    mappingName
11      main-cylinder
12    lines
13      31 21 6
14    exit
15 Cylinder
16   mappingName
17     top-cylinder
18   orientation
19     2 0 1
20   bounds on the axial variable
21     .25 1.
22   bounds on the radial variable
23     .3 .4
24   boundary conditions
25     -1 -1 0 0 3 4
26   lines
27     25 15 5
28   exit
29 join
30 choose curves
31   top-cylinder
32   main-cylinder (side=0, axis=2)
33   compute join
34 lines
35   25 11 6  31 15 7
36 boundary conditions
37   -1 -1 1 1 1 0

```



A new surface mapping is generated (the upper ‘cylinder’ in the bottom figure) that lies on the vertical cylinder (source-mapping) and exactly matches to the horizontal cylinder (clip-surface).

### 17.3 Intersecting a volume intersector mapping with a surface intersectee mapping.

Suppose now that source-mapping defines a volume mapping,  $R^3 \rightarrow R^3$ . The JoinMapping can be used to build a new volume Mapping that will match exactly to the clip-surface.

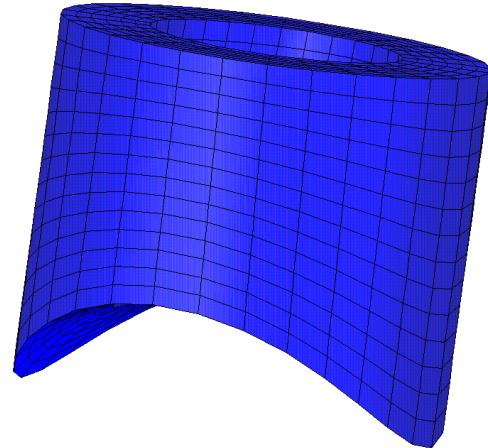
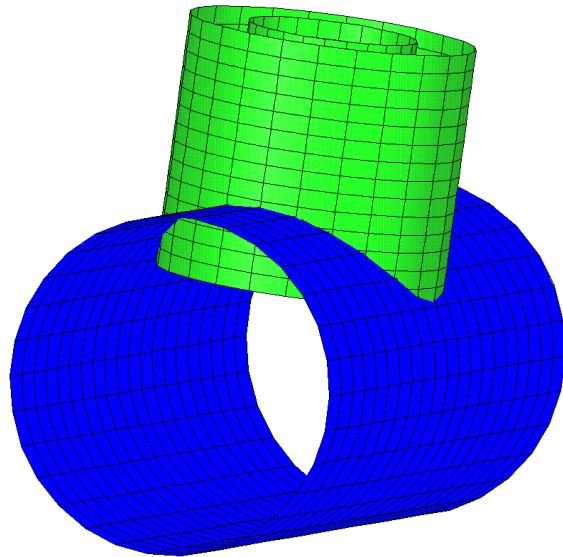
In this case we require that the two faces of the source-mapping intersect the clip-surface, say,

$\mathbf{C}_i(r_1, r_2, 0)$  and  $\mathbf{C}_i(r_1, r_2, 1)$ . We proceed as before to generate a `JoinMapping` for each of these intersecting surfaces,  $\mathbf{J}_m(\mathbf{r})$ ,  $m = 1, 2$ . We also generate a third...

```

1   Cylinder
2     orientation
3       1 2 0
4     bounds on the axial variable
5       -1. 1.
6     bounds on the radial variable
7       .5 .75
8     boundary conditions
9       -1 -1 0 0 3 0
10    mappingName
11      main-cylinder
12    lines
13      31 21 6
14    exit
15 Cylinder
16    mappingName
17      top-cylinder
18    orientation
19      2 0 1
20    bounds on the axial variable
21      .25 1.
22    bounds on the radial variable
23      .3 .4
24    boundary conditions
25      -1 -1 0 0 3 4
26    lines
27      25 15 5
28    exit
29 join
30 choose curves
31   top-cylinder
32   main-cylinder (side=0, axis=2)
33   compute join
34 lines
35   25 11 6  31 15 7
36 boundary conditions
37   -1 -1 1 1 1 0

```



A new volume mapping is generated that consists of a portion of the vertical cylinder (source-mapping) and exactly matches to the horizontal cylinder (clip-surface).

## 17.4 Constructor

`JoinMapping()`

**Description:** Define a mapping that transforms a "source-mapping" so that it intersects another "clip-

surface" "exactly". For example, a Mapping for a wing (source-mapping) can be joined to a fuselage (clip-surface).

## 17.5 Constructor

```
JoinMapping(Mapping & sourceMapping,  
           Mapping & clipSurface)
```

**Description:** Define a mapping that transforms a "source-mapping" so that it intersects another "clip-surface" "exactly". For example, a Mapping for a wing (source-mapping) can be joined to a fuselage (clip-surface).

**sourceMapping (input):** defines the source-mapping. This is the Mapping that will be changed.

**clipMapping :** defines the clip-surface. This Mapping will clip away a portion of the sourceMapping.  
Use the setEndOfJoin function to specify which portion of the sourceMapping to retain.

## 17.6 setCurves

```
int  
setCurves(Mapping & sourceMapping,  
           Mapping & clipSurface )
```

**Description:** Supply the source-mapping and clip-surface from which the join will be defined.

**sourceMapping (input):** defines the source-mapping. This is the Mapping that will be changed.

**clipMapping :** defines the clip-surface. This Mapping will clip away a portion of the sourceMapping.  
Use the setEndOfJoin function to specify which portion of the sourceMapping to retain.

## 17.7 setEndOfJoin

```
int  
setEndOfJoin( const real & endOfJoin_ )
```

**Description:** Specify the r value for the end of the join opposite the curve of intersection. Use this to specify which portion of the source-mapping to retain. For example, choosing a value of 0 or 1 will select the portion of the source-mapping that lies on one side of the clip-surface or the other side. Choosing a value of .8, for example, will shorten the join-mapping.

**endOfJoin\_ (input) :** a value in [0,1].

## 17.8 map

```
void  
map( const realArray & r, realArray & x, realArray & xr, MappingParameters & params )
```

**Purpose:** Evaluate the TFI and/or derivatives.

## 17.9 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively create and/or change the Join mapping.

**mapInfo (input):** Holds a graphics interface to use.

## 17.10 Class LineMapping

This mapping a line in one, two or three dimensions.

$$\begin{aligned}\mathbf{x}(r) &= x_a + r(x_b - x_a) \\ \mathbf{x}(r) &= (x_a, y_a) + r(x_b - x_a, y_b - y_a) \\ \mathbf{x}(r) &= (x_a, y_a, z_a) + r(x_b - x_a, y_b - y_a, z_b - z_a)\end{aligned}$$

## 17.11 Constructor

```
LineMapping(const real xa_,  
           const real xb_,  
           const int numberGridPoints )
```

**Description:** Build a mapping for a line in 1D.

**xa\_**, **xb\_** (**input**) : End points of the interval.

## 17.12 Constructor

```
LineMapping(const real xa_,const real ya_,  
           const real xb_,const real yb_,  
           const int numberGridPoints)
```

**Description:** Build a mapping for a line in 2D.

**xa\_**, **ya\_**, **xb\_**, **yb\_** (**input**) : End points of the line.

## 17.13 Constructor

```
LineMapping(const real xa_,const real ya_,const real za_,  
           const real xb_,const real yb_,const real zb_,  
           const int numberGridPoints)
```

**Description:** Build a mapping for a line in 3D.

**xa\_**, **ya\_**, **za\_**, **xb\_**, **yb\_**, **zb\_** (**input**) : End points of the line.

## 17.14 getPoints

```
int  
getPoints( real & xa_, real & xb_ ) const
```

**Description:** Get the end points of the line.

**xa\_**, **xb\_** (**output**) : End points of the line.

## 17.15 getPoints

```
int  
getPoints( real & xa_, real & ya_,  
           real & xb_, real & yb_ ) const
```

**Description:** Get the end points of the line.

**xa\_**, **ya\_**, **xb\_**, **yb\_** (**output**) : End points of the line.

## 17.16 getPoints

```
int  
getPoints( real & xa_, real & ya_, real & za_,  
           real & xb_, real & yb_, real & zb_ ) const
```

**Description:** Get the end points of the line.

**xa\_, ya\_,za\_, xb\_, yb\_,zb\_ (output) :** End points of the line.

## 17.17 setPoints

```
int  
setPoints( const real & xa_, const real & xb_ )
```

**Description:** Specify the end points for a line in 1D.

**xa\_, xb\_ (input) :** End points of the interval.

## 17.18 setPoints

```
int  
setPoints( const real & xa_, const real & ya_,  
           const real & xb_, const real & yb_ )
```

**Description:** Specify the end points for a line in 2D.

**xa\_, ya\_, xb\_, yb\_ (input) :** End points of the line.

## 17.19 setPoints

```
int  
setPoints( const real & xa_, const real & ya_, const real & za_,  
           const real & xb_, const real & yb_, const real & zb_ )
```

**Description:** Specify the end points for a line in 3D.

**xa\_, ya\_,za\_, xb\_, yb\_,zb\_ (input) :** End points of the line.

## 18 LoftedSurfaceMapping: build a lofted surface

The LoftedSurfaceMapping can be used to define a lofted surface such as the surface of a wing with a tip. A lofted surface is defined using *sections* (formulas that define the basic cross sections) and *profile* curves (that can be used to deform the sections, such as to form a wing tip). Figures 12, 13 and 14 show examples of some lofted surfaces.

NOTE: The current intention is that users should edit the `LoftedSurfaceMapping.C` file to define a new lofted surface by changing the formulas for the sections and profile curve.

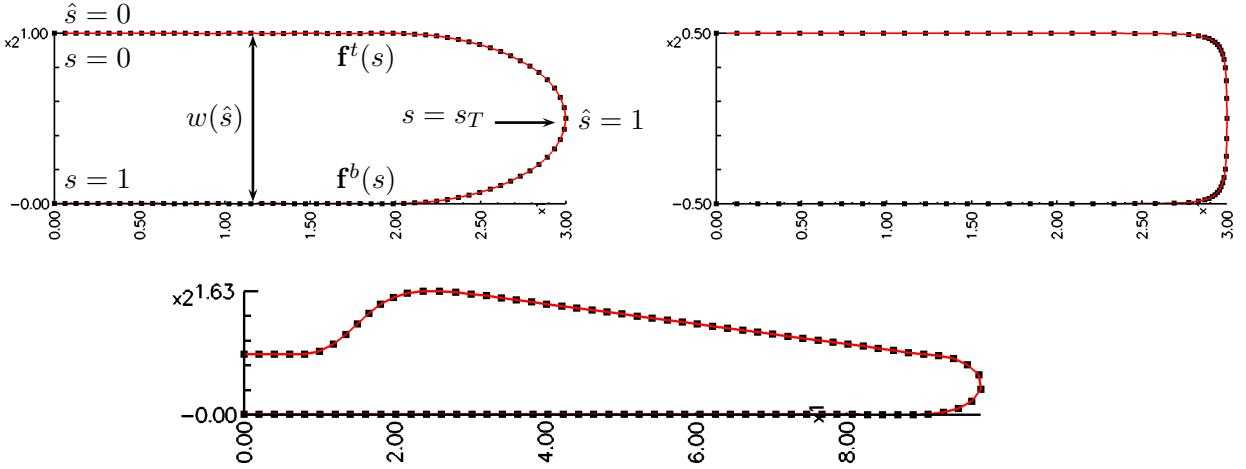


Figure 11: Some lofted surface profile curves. Top left: rounded-tip-profile. Top right: flat-tip-profile. Bottom: wind-turbine-profile.

The generic lofted surface is a three-dimensional surface,  $\mathbf{x} = \mathbf{L}(s, \theta)$ ,  $\mathbf{L} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ , that is a function of an axial variable  $s$  and a tangential variable  $\theta$ . In the examples given here, the surface is thought to extend primarily in the  $z$  direction, and be of the form

$$\mathbf{L}(s, \theta) = (x(s, \theta), y(s, \theta), z(s, \theta)) = (\mathbf{S}(s, \theta), Z_L(s)), \quad (11)$$

where the function  $\mathbf{S}(s, \theta)$  defines the  $(x, y)$  coordinates of the cross-sections and is usually of the form

$$\mathbf{S}(s, \theta) = R(s, \theta)(\mathbf{c}(s, \theta) w(s)) + \mathbf{g}(s), \quad (12)$$

where

- $\mathbf{c}(s, \theta)$  = basic unscaled cross-section curves,
- $w(s)$  = profile width function (that defines the shape of the tip),
- $R(s, \theta)$  = twist and scale matrix operator,
- $\mathbf{g}(s)$  = flex operator that bends the surface.

*Profile curves* are used to define  $w(s)$  and  $Z(s)$ . Examples of profile curves are given in Figure 11. In order to have a nicely defined tip, we first define a single profile function  $(x, y) = \mathbf{f}^p(s)$ , that is subsequently split into a top and bottom portion,

$$\mathbf{f}^p(s) = \begin{cases} \mathbf{f}^t(s) & \text{for } 0 < s < s_T, \\ \mathbf{f}^b(s) & \text{for } s_T < s < 1, \end{cases}$$

where  $s_T$  denotes the position of the “tip”. The profile curves have  $x$  and  $y$  components denoted by  $\mathbf{f}^p(s) = (f_x^p(s), f_y^p(s))$  and  $\mathbf{f}^b(s) = (f_x^b(s), f_y^b(s))$ . From the profile curves we define the *profile width* as the (vertical)  $y$ -distance between the top and bottom profile curves,

$$w(\hat{s}) = f_y^t(\hat{s} s_T) - f_y^b(1 - (1 - s_T)\hat{s}), \quad \text{for } 0 \leq \hat{s} \leq 1, (\hat{s} = s/s_T),$$

while the axial distance function  $z = Z_L(s)$  in (11) is defined as the  $x$ -component of the top profile,

$$Z_L(\hat{s}) = f_x^t(\hat{s} s_T), \quad \text{for } 0 \leq \hat{s} \leq 1.$$

Note that in the examples shown,  $w(1) = 0$ , and thus the surface will close to a point at the tip. This point on the surface is a coordinate singularity and later in this section we discuss how to deal with this singularity when constructing an overlapping grid.

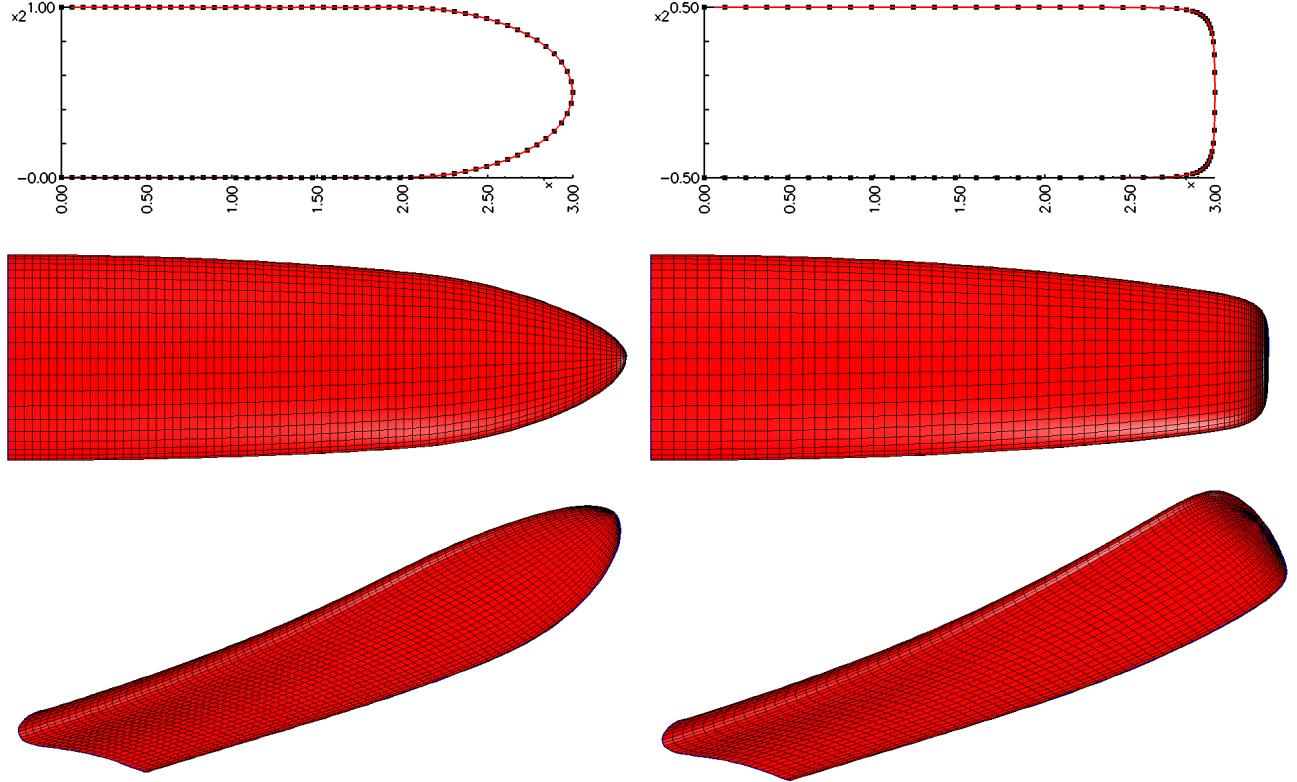


Figure 12: Lofted surface : generic wing with Joukowsky sections, variable chord, twist and flex. Left: rounded tip wing showing the profile function (top) and two views of the surface. Right: flat tip wing showing the profile function (top) and two views of the surface. The profile functions act primarily to form the tip. A scale function causes the chord to decrease through the main portion of the surface.

Two lofted surfaces for a generic wing with twist and flex are in shown in Figure 12. The basic section curve for this wing is a Joukowsky airfoil (defined in Section 5),

$$\mathbf{c}(s, \theta) = (x_J(\theta), y_J(\theta)) = \mathbf{J}(\theta) = \text{Joukowsky airfoil}.$$

The *twist and scale matrix*  $R$  in (12) is the product of a rotation matrix and a scaling factor,

$$R(\zeta, \theta) = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} S(\zeta)$$

where the rotation angle  $\phi = \phi(\zeta)$  and scale function  $S(\zeta)$  are some function of the scaled  $z$  coordinate,  $\zeta$ , defined by

$$\begin{aligned} \zeta &= Z(s)/L, \\ L &= Z(s_T). \quad (\text{length of the wing}) \end{aligned}$$

Normally one would use  $\zeta$  as a variable rather than  $s$  when defining properties of the surface in physical coordinates. For example, with the flat-tip-profile in Figure 11,  $s$  continues to vary along the flat face even

though  $\zeta$  does not. In the example shown in Figure 12 we choose  $\phi$  to vary between  $\phi_a = 0$  and  $\phi_b = -30^\circ$  by the formula

$$\phi(\zeta) = \phi_a(1 - \zeta^2) + \phi_b\zeta^2,$$

while the scale function varies between  $S_a = 1$  and  $S_b = .7$  according to

$$S(\zeta) = S_a(1 - \zeta^3) + S_b\zeta^3.$$

The scale function acts to shorten the chord and make the surface thinner. The flex function is chosen to bend the surface in the  $y$ -direction and is given by

$$\begin{aligned} \mathbf{g}(\zeta) &= (0, g_y(\zeta)), \\ g_y(\zeta) &= g_a(1 - \zeta^3) + g_b\zeta^3, \\ g_a &= 0, \quad g_b = 1/4. \end{aligned}$$

The powers of  $\zeta$  used in these formulae are arbitrary.

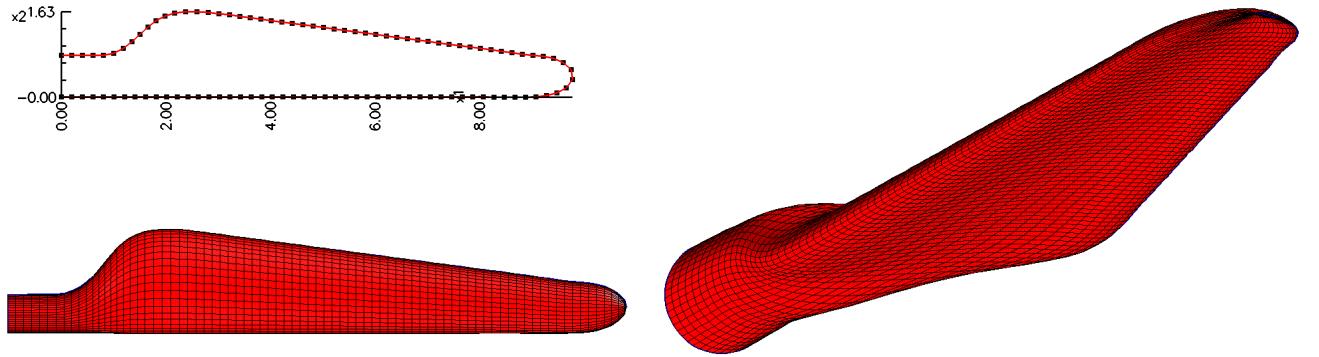


Figure 13: Lofted surface : wind-turbine blade showing the profile (top left) and two views of the wing. The profile function determines the overall shape and the tip.

The wind-turbine surface in Figure 13 uses the profile function to shape the entire surface. The unscaled cross-section function  $\mathbf{c}(s, \theta)$  starts as a circle and transitions to a Joukowsky airfoil.

#### Notes:

1. In the examples represented here the surface converges to a point at the tip. This is a result of the profile width function  $w(s)$  going to zero as  $s$  goes to 1. In order that the surface near the tip be single valued, it is important that the point  $(x, y) = (0, 0)$  be inside the the cross-section curves  $\mathbf{c}(s, \theta)$  as  $s$  approaches 1. To achieve this property we shift the basic cross-section (such as the Joukowsky profile) so that this is true. We then rely on the rotation, scaling and flex functions to define the actual shape (note that these functions as applied after we multiply by  $w(s)$ ).

Figure 14 shows a surface for a box with rounded corners generated with the lofted surface mapping. The profile is the *flat double tip profile*. The sections are defined from a smoothed polygon mapping that blends into a circle at the tips. This mapping can be used to create a grid around the exterior of a box.

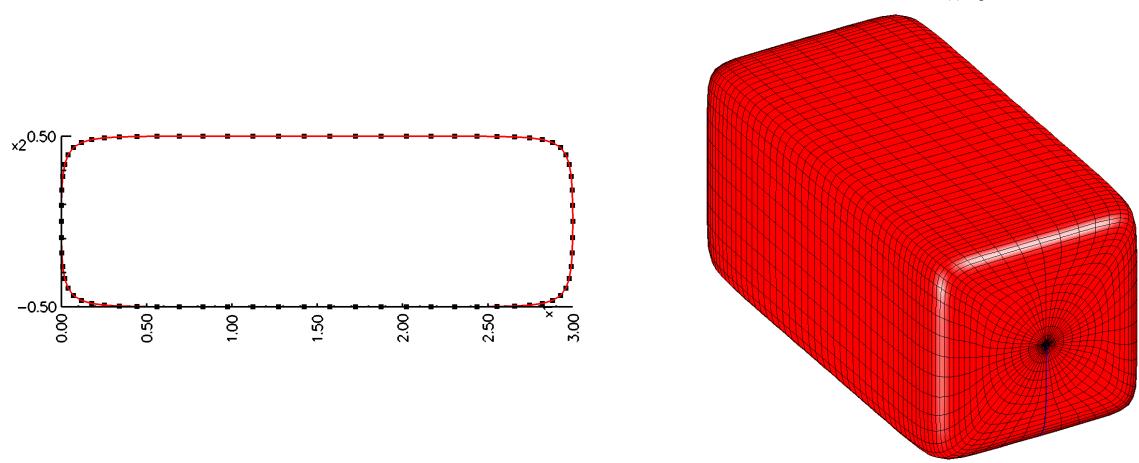


Figure 14: Lofted surface : box with rounded corners constructed from the *smooth polygon sections* and the *flat double tip profile*. Left: profile. Right: lofted surface for the box. Note that the cross-sections transition to a circle at the singular poles.

## 19 MatrixMapping: define a mapping from scalings, shifts and rotations

This mapping can be used for rotations, scalings and shifts or any transformation that can be represented as a matrix times a vector. The mapping is defined by a  $4 \times 4$  matrix that maps from  $r$  to  $x$  by the relation

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ 1 \end{bmatrix}$$

Each time the functions `rotate`, `scale` and `shift` are called the current `matrix` is updated and thus the transformations are cumulative.

Here is an example of the use of the `MatrixMapping` class.

```
#include "Overture.h"

void main()
{
    realArray r(1,3);
    realArray x(1,3);
    realArray xr(1,3,3);

    MatrixMapping rotScaleShift ;      // Define a matrix mapping

    rotScaleShift.rotate( axis3, Pi/2. ); // rotate about the x_3 axis
    rotScaleShift.scale( 2.,1.,1. );     // scale by 2 in x_1-direction
    rotScaleShift.shift( 0.,1.,0. );     // shift by 1 in x_2 direction

    r=.5;

    rotScaleShift.map( r,x,xr );
}
```

### 19.1 Constructor

```
MatrixMapping(int domainDimension_ = 3,
              int rangeDimension_ = 3)
```

**Purpose:** Build a matrix mapping. This is normally used with the `MatrixTransform` to rotate, scale, or translate an existing mapping.

**domainDimension\_, rangeDimension\_ (input)** : domain and range dimension.

### 19.2 rotate

```
void
rotate( const int axis, const real theta )
```

**Purpose:** Perform a rotation about a given axis. This rotation is applied after any existing transformations. Use the `reset` function first if you want to remove any existing transformations.

**axis (input)** : axis to rotate about (0,1,2)

**theta (input)** : angle in radians to rotate by.

### 19.3 rotate

```
void  
rotate( const RealArray & rotate, bool incremental =false)
```

**Purpose:** Perform a "rotation" using a  $3 \times 3$  matrix. This does not really have to be a rotation.

**rotate (input):** If incremental=false then the upper  $3 \times 3$  portion of the  $4 \times 4$  transformation matrix will be replaced by the matrix `rotate(0:2,0:2)`. Otherwise this rotation matrix will multiply the existing transformation.

**incremental (input) :** if true apply this rotation to the existing transformation, otherwise replace the existing rotation.

### 19.4 scale

```
void  
scale( const real scalex =1.,  
       const real scaley =1.,  
       const real scalez =1.,  
       bool incremental =true)
```

**Purpose:** Perform a scaling

**scalex, scaley, scalez (input):** Scale factors along each axis.

**incremental (input) :** if true then incrementally transform the existing mapping, other transform the original mapping.

### 19.5 shift

```
void  
shift( const real shiftx =0.,  
       const real shifty =0.,  
       const real shitz =0.,  
       bool incremental =true)
```

**Purpose:** Perform a shift.

**shiftx, shifty, shitz (input):** shifts along each axis.

**incremental (input) :** if true then incrementally transform the existing mapping, other transform the original mapping.

### 19.6 reset

```
void  
reset()
```

**Purpose:** reset the matrix to the identity.

## 19.7 matrixMatrixProduct

```
void  
matrixMatrixProduct( RealArray & m1, const RealArray & m2, const RealArray & m3 )
```

**Purpose:** Multiply two 4x4 matrices together. This is a utility routine (a static member function that can be called without a MatrixMapping object using MatrixMapping::matrixMatrixProduct(...)).

```
m1 <- m2*m3
```

## 19.8 matrixVectorProduct

```
void  
matrixVectorProduct( RealArray & v1, const RealArray & m2, const RealArray & v3 )
```

**Purpose:** Multiply a 4x4 matrix times a vector. This is a utility routine (a static member function).

```
v1 <- m2*v3
```

## 19.9 matrixInversion

```
int  
matrixInversion( RealArray & m1Inverse, const RealArray & m1 )
```

**Purpose:** Invert a 4x4 matrix. This is a utility routine (a static member function). This only works for matrices used in transforming 3D vectors which look like:

```
[ a00 a01 a02 a03 ]  
[ a10 a11 a12 a13 ]  
[ a20 a21 a22 a23 ]  
[ 0 0 0 1 ]
```

**Return value:** 0=success, 1=matrix is not invertible

## 20 MatrixTransform: rotate, scale or shift an existing mapping

### 20.1 Description

The `MatrixTransform` mapping can be used to rotate, scale and shift another mapping. It does this by composing the given mapping with a `MatrixMapping`, section (19).

### 20.2 Constructor

`MatrixTransform()`

**Purpose:** Build a mapping for matrix transform.

### 20.3 Constructor(`Mapping&`)

`MatrixTransform(Mapping & map)`

**Purpose:** Build a Mapping for matrix transformation of another Mapping.

### 20.4 `reset`

```
void  
reset()
```

**Purpose:** Reset the transformation to the identity.

### 20.5 `rotate`

```
void  
rotate( const int axis, const real theta )
```

**Purpose:** Perform a rotation about a given axis.

**axis (input)** : axis to rotate about (0,1,2)

**theta (input)** : angle in radians to rotate by.

### 20.6 `rotate`

```
void  
rotate( const RealArray & rotate, bool incremental =false)
```

**Purpose:** Perform a "rotation" using a  $3 \times 3$  matrix. This does not really have to be a rotation.

**rotate (input):** If `incremental=false` then the upper  $3 \times 3$  portion of the  $4 \times 4$  transformation matrix will be replaced by the matrix `rotate(0:2,0:2)`. Otherwise this rotation matrix will multiply the existing transformation.

**incremental (input) :** if true apply this rotation to the existing transformation, otherwise replace the existing rotation.

## 20.7 scale

```
void  
scale( const real scalex =1.,  
       const real scaley =1.,  
       const real scalez =1.,  
       bool incremental =true)
```

**Purpose:** Perform a scaling

**scalex, scaley, scalez (input):** Scale factors along each axis.

**incremental (input) :** if true then incrementally transform the existing mapping, other transform the original mapping.

## 20.8 shift

```
void  
shift( const real shiftx =0.,  
       const real shifty =0.,  
       const real shiftz =0.,  
       bool incremental =true)
```

**Purpose:** Perform a shift.

**shiftx, shifty, shiftz (input):** shifts along each axis.

**incremental (input) :** if true then incrementally transform the existing mapping, other transform the original mapping.

## 21 NormalMapping: define a new mapping by extending normals

### 21.1 Description

The NormalMapping extends normals from an existing curve or surface to generate a new mapping.

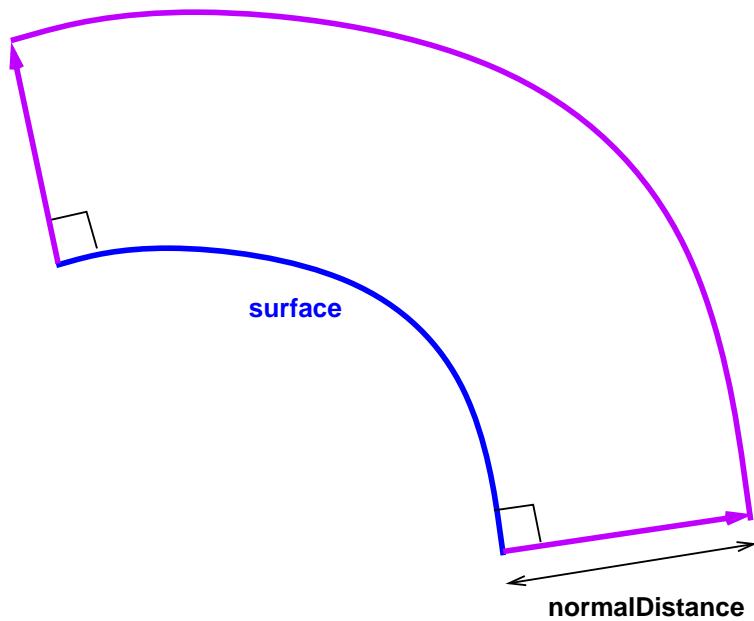


Figure 15: The NormalMapping defines a new mapping by extending normals from a given curve or surface

### 21.2 Member Functions

ormalMappingInclude.tex

## 22 NurbsMapping: define a new mapping as a NURBS.

The NurbsMapping class defines mapings in terms of a non-uniform rational b-spline, NURBS. The implementation here is based on the reference, *The NURBS Book* Les Piegl and Wayne Tiller, Springer, 1997.

The n-th degree Bernstein polynomial is

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

and the n-th degree Bezier curve

$$\mathbf{C}(u) = \sum_{i=0}^n B_{i,n}(u) \mathbf{P}_i, \quad 0 \leq u \leq 1$$

with control points  $\mathbf{P}_i$ .

The n-th degree rational Bezier curve is

$$\begin{aligned} \mathbf{C}(u) &= \frac{\sum_{i=0}^n B_{i,n}(u) w_i \mathbf{P}_i}{\sum_{i=0}^n B_{i,n}(u) w_i}, \quad 0 \leq u \leq 1 \\ &= \sum_{i=0}^n R_{i,n}(u) \mathbf{P}_i \end{aligned}$$

with weights  $w_i$ .

Written using homogeneous coordinates

$$\mathbf{C}^w(u) = \sum_{i=0}^n B_{i,n}(u) \mathbf{P}_i^w$$

where  $\mathbf{P}_i^w = (w_i \mathbf{P}_i, w_i)$ .

B-spline basis functions are defined as

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1 & u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \\ N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \end{aligned}$$

where  $\mathbf{U} = \{u_0, \dots, u_m\}$  are the knots,  $u_i \leq u_{i+1}$ .

We only use nonperiodic (clamped or open) knot vectors,

$$\mathbf{U} = \{a, \dots, a, u_{p+1}, \dots, u_{m-p-1}, b, \dots, b\}$$

with the end knots repeated  $p+1$  times.

NonUniform Rational B-Spline (NURBS). p-th degree NURBS curve

$$\begin{aligned} \mathbf{C}(u) &= \frac{\sum_{i=0}^n N_{i,p}(u) w_i \mathbf{P}_i}{\sum_{i=0}^n N_{i,p}(u) w_i}, \quad a \leq u \leq b \\ &= \sum_{i=0}^n R_{i,p}(u) \mathbf{P}_i \end{aligned}$$

Written using homogeneous coordinates

$$\mathbf{C}^w(u) = \sum_{i=0}^p N_{i,p}(u) \mathbf{P}_i^w$$

## 22.1 Constructor

**NurbsMapping()**

**Purpose:** Default Constructor, make a null NURBS.

**Remarks:** The implementation here is based on the reference, *The NURBS Book* by Les Piegl and Wayne Tiller, Springer, 1997. The notation here is:

- degree = p (variables p1,p2 for one and 2D)
- number of control points is n+1 (variables n1,n2)
- number of knots is m+1 ( $m=n+p+1$ ) (variables m1,m2)
- cPoint(0:n,0:r) : holds the control points and weights. r=rangeDimension.
- uKnot(0:m) : holds knots along axis1. These are normally scaled to [0,1] (see notes below).
- vKnot(0:m) : holds knots along axis2 (if domainDimension==2)
- note : Knots are scaled to [0,1]

**NOTES:** for those wanting to make changes to this class

**uMin,uMax,vMin,vMax** : A typical NURBS will have knots that span an arbitrary interval. For example the knots may go from [.5, 1.25]. This mapping however, is parameterized on [0,1]. To fix this we first save the actual min and max values for uKnot in [uMin,uMax] and similarly for [vMin,vMax]. We then rescale uKnot and vKnot to lie on the interval [0,1]. Note that the **reparameterize** function may subsequently rescale the knots to a larger interval in which case the NURBS will only represent a part of the initial surface. If we do this then we also rescale uMin,uMax,vMin,vMax. The **parametricCurve** function is used to indicate that this NURBS is actually a parametric curve on another NURBS, nurbs2. By default the values of uMin,uMax,vMin,vMax from nurbs2 are used to scale this NURBS in order to make it compatible with the rescaled nurbs2.

## 22.2 Constructor

**NurbsMapping(const int & domainDimension\_ , const int & rangeDimension\_ )**

**Purpose:** Constructor, make a default NURBS of the give domain dimension (1,2)

## 22.3 intersect3DLines

```
int
intersect3DLines( const RealArray & pt0, const RealArray & t0,
                  const RealArray & pt1, const RealArray & t1,
                  real & alpha0, real & alpha1,
                  RealArray & pt2) const
```

**Description:** Intersect two lines in 3D:  $x_0(s) = pt0 + s * t0$   $x_1(t) = pt1 + t * t1$

**alpha0,alpha1** : values of s and t at the intersection.

**pt2** : point of intrsection,  $x_0(alpha0)=pt2=x_1(alpha1)$

**Return values:** 1 if the line are parallel, 0 otherwise.

## 22.4 buildCurveOnSurface

```
int  
buildCurveOnSurface(NurbsMapping & curve,  
                    real r0,  
                    real r1 =-1.)
```

**Description:** Build a new Nurbs curve that matches a coordinate line on the surface.

**curve (output) :** on output a curve that matches a coordinate line on the surface.

**r0,r1 (input) :** if  $r1 == -1$  make a curve  $\mathbf{c}(r) = \mathbf{s}(r_0, r)$  where  $\mathbf{s}(r_0, r_1)$  is the NURBS surface defined by this mapping. If  $r0 == -1$  make the curve  $\mathbf{c}(r) = \mathbf{s}(r, r_1)$  the arc, measured starting from x.

## 22.5 buildComponentCurve

```
int  
buildComponentCurve(NurbsMapping & curve,  
                     int component = 0)
```

**Description:** Given a curve,  $(x(s), y(s)[, z(s)])$ , in 2D or 3D, build a new curve in 1D that represents the a given component such as  $x(s)$  or  $y(s)$  or  $z(s)$ .

**curve (output) :** on output a component curve.

**component (input) :** Choose component=0,1, or 2 to form the curve for x,y, or z

## 22.6 circle

```
int  
circle(RealArray & o,  
       RealArray & x,  
       RealArray & y,  
       real r,  
       real startAngle =0.,  
       real endAngle =1.)
```

**Description:** Build a circular arc. Reference the NURBS book Algorithm A7.1

**o (input):** center of the circle.

**x,y (input):** orthogonal unit vectors in the plane of the circle.

**startAngle,endAngle :** normalized angles [0,1] for the start and end of the arc, measured starting from x.

## 22.7 conic(a,b,c,...)

```
int  
conic( const real a, const real b, const real c, const real d, const real e, const real f,  
       const real z, const real x1, const real y1, const real x2, const real y2 )
```

**Description:** Build a NURBS for a conic defined by an implicit formula and two end points

$$a*x^2 + b*x*y + c*y^2 + d*x + e*y + f = 0$$

**a,b,c,d,e,f** : implicit formula for a conic

**z,x1,y1,x2,y2**: the end points of the conic are (x1,y1,z) and (x2,y2,z)

**return values:** 0=success, 1=error

## 22.8 conic

**int**

**conic( const RealArray &pt0, const RealArray &t0, const RealArray &pt2, const RealArray &t2, const RealArray &p )**

**Description:** Build a NURBS for a conic defined by end points, two tangents, and an additional point

**pt0(0,0: 2),pt2(0,0:2)** : end points

**t0(0,0: 2),t2(0,0:2)** : tangent directions at end points

**p(0,0: 2)** : another point on the conic

**NOTES:** Construct open conic arc in 3D (c.f. makeOpenConic, NURBS book p. 317)

**NOTE:** \*\*\*\*TODO: case of full ellipse, see page 319

**return values:** 0=success, 1=error

## 22.9 generalCylinder(curve,shift)

**int**

**generalCylinder( const Mapping & curve, real d[3] )**

**Description:** Build a general cylinder (tabulated cylinder) by extruding a curve along a direction vector d. The general cylinder is defined as

$$C(u, v) = \text{curve}(u) + v*d$$

**curve (input)** : curve to extrude. If this curve is not a NurbsMapping then it is converted into a NurbsMapping by interpolating the grid points from curve. Increase the number of grid points on curve if you want a more accurate result.

**d[3] input** : extrude the curve along this direction vector

## 22.10 generalCylinder(curve1,curve2)

**int**

**generalCylinder( const Mapping & curve1, const Mapping & curve2 )**

**Description:** Build a general cylinder (tabulated cylinder) by interpolating between two curves. The general cylinder is defined as

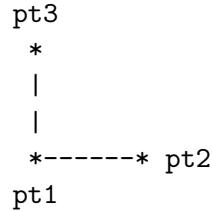
$$C(u, v) = (1-v)*\text{curve1}(u) + v*\text{curve2}$$

**curve1, curve2 (input)** : curves to interpolate. If either curve is not a NurbsMapping then it is converted into a NurbsMapping by interpolating the grid points from curve. Increase the number of grid points on curve if you want a more accurate result.

## 22.11 plane

```
int  
plane( real pt1[3], real pt2[3], real pt3[3] )
```

**Description:** Build a plane that passes through three points



The plane is defined as

$$C(u, v) = pt1 + u*(pt2-pt1) + v*(pt3-pt1)$$

**pt1[3], pt2[3], pt3[3] (input)** : 3 points on the plane

## 22.12 getKnots

```
const RealArray &  
getKnots( int direction =0) const
```

**Purpose:** get uKnot or vKnot, the knots in the first or second direction.

**direction:** 0=return uKnot, 1= return vKnot.

## 22.13 getControlPoints

```
const RealArray &  
getControlPoints() const
```

**Purpose:** Return the control points, scaled by the weight.

## 22.14 insertKnot

```
int  
insertKnot(const real & uBar,  
           const int & numberoftimesToInsert_ =1)
```

**Purpose:** Insert a knot

**uBar (input):** Insert this knot value.

**numberoftimesToInsert\_ (input):** insert the knot this many times. The multiplicity of the knot will not be allowed to exceed p1.

## 22.15 insertKnot

```
int  
normalizeKnots()
```

**Access:** Protected routine.

**Purpose:** Normalize the knots, uKnot (and vKnot if domainDimension==2) to lie from 0 to 1. This routine will NOT change the values of uMin,uMax, vMin,vMax since these values indicate the original bounds on uKnot and vKnot.

## 22.16 readFromIgesFile

```
int  
readFromIgesFile( IgesReader & iges, const int & item, bool normKnots /*=true*/ )
```

**Purpose:** Read a NURBS from an IGES file. An IGES file is a data file containing geometrical objects, usually generated by a CAD program.

**iges (input) :** Use this object to read the IGES file.

**item (input) :** read this item from the IGES file.

## 22.17 parametricCurve

```
int  
parametricCurve(const NurbsMapping & nurbs,  
                 const bool & scaleParameterSpace = true)
```

**Purpose:** Indicate that this nurb is actually a parametric curve on another nurb surface.

**nurbs (input) :** Here is the NURBS surface for which this NURBS is a parametric surface.

**scaleParameterSpace (input) :** if true, scale the range space of this nurb to be on the unit interval. This is usually required since the NurbsMapping scales the knots to lie on [0,1] (normally) and so we then need to scale this Mapping to be consistent.

## 22.18 shift

```
int  
shift(const real & shiftx =0.,  
      const real & shifty =0.,  
      const real & shiftz /* =0.*/ )
```

**Purpose:** Shift the NURBS in space.

## 22.19 scale

```
int  
scale(const real & scalex =0.,  
      const real & scaley =0.,  
      const real & scalez /* =0.*/ )
```

**Purpose:** Scale the NURBS in space.

## 22.20 rotate

```
int  
rotate( const int & axis, const real & theta )
```

**Purpose:** Perform a rotation about a given axis. This rotation is applied after any existing transformations.

**axis (input)** : axis to rotate about (0,1,2)

**theta (input)** : angle in radians to rotate by.

## 22.21 rotate

```
int  
matrixTransform( const RealArray & m )
```

**Purpose:** Perform a general matrix transform using a 2x2 or 3x3 matrix. Convert the NURBS to 2D or 3D if the transformation so specifies – i.e. if you transform a NURBS with rangeDimension==2 with a 3x3 matrix then the result will be a NURBS with rangeDimension==3.

**m (input)** : m(0:2,0:2) matrix to transform with

## 22.22 specify knots and control points

```
int  
specify(const int & m,  
        const int & n,  
        const int & p,  
        const RealArray & knot,  
        const RealArray & controlPoint,  
        const int & rangeDimension_ =3,  
        bool normalizeTheKnots /* =true */ )
```

**Purpose:** Specify a **curve** in 2D or 3D using knots and control points

**m (input)** : The number of knots is m+1

**n (input)** : the number of control points is n+1

**p (input)** : order of the B-spline

**controlPoint(0: n,0:rangeDimension) (input)** : control points and weights

**normalizeTheKnots (input)** : by default, normalize the knots to [0,1]. Set to false if you do not want the knots normalized.

## 22.23 specify knots and control points

```
int  
specify(const int & n1_,  
        const int & n2_,  
        const int & p1_,  
        const int & p2_,  
        const RealArray & uKnot_ ,
```

```

const RealArray & vKnot_ ,
const RealArray & controlPoint ,
const int & rangeDimension_ =3,
bool normalizeTheKnots /* =true*/ )

```

**Purpose:** Specify a NURBS with domainDimension==2 using knots and control points

**n1\_,n2\_ (input)** : the number of control points is n1+1 by n2+1

**p1\_,p2\_ (input)** : order of the B-spline in each direction.

**uKnot\_,vKnot\_ (input)** : knots.

**controlPoint(0: n1,0:n2,0:rangeDimenion) (input)** : control points and weights

**normalizeTheKnots (input)** : by default, normalize the knots to [0,1]. Set to false if you do not want the knots normalized.

### 22.23.1 setDomainInterval

**int**

```

setDomainInterval(const real & r1Start =0.,
                  const real & r1End =1.,
                  const real & r2Start =0.,
                  const real & r2End =1.,
                  const real & r3Start =0.,
                  const real & r3End =1.)

```

**Description:** Restrict the domain of the nurbs. By default the nurbs is parameterized on the interval [0,1] (1D) or [0,1]x[0,1] in 2D etc. You may choose a sub-section of the nurbs by choosing a new interval [rStart,rEnd]. For periodic nurbss the interval may lie in [-1,2] so the sub-section can cross the branch cut. You may even choose rEnd|rStart to reverse the order of the parameterization.

**rStart1,rEnd1,rStart2,rEnd2,rStart3,rEnd3 (input)** : define the new interval.

### 22.24 initialize()

**void**

```

initialize( bool setMappingHasChanged =true)

```

**Purpose:** Initialize the NURBS. This is a protected routine. Determine if the weights are constant so that we can use more efficient routines. Set bounds for the Mapping.

**setMappingHasChanged (input)** : if true (by default) we call mappingHasChanged so that the grid etc. will be marked out of date.

**NOTES:** Normally we multiply the control points by the weights. BUT, if the weights are constant we divide everything by this constant value so we can avoid dividing by the weight term when we evaluate. When the weights are constant **nonUniformWeights==false;**

### 22.25 setBounds

**void**

```

setBounds()

```

**Purpose:** protected routine. Set the approximate bounds on the mapping, used by plotting routines etc.

Use the control points as an approximation \*\*\* note only apply this to the normalized control-points  
\*\*\*

## 22.26 removeKnot

```
int  
removeKnot(const int & index,  
           const int & numberOfTimesToRemove why is this int passed by reference?,  
           int & numberRemoved, const real & tol ) why is tol passed by reference?
```

**Purpose:** Remove a knot (if possible) so that the Nurbs remains unchanged

**index (input)** : try to remove the knot at this index. AP: NOTE: uKnot(index) != uKnot(index+1)

**numberOfTimesToRemove (input)** : the number of times to try and remove the knot.

**numberRemoved (output):** the actual number of times the knot was removed

## 22.27 getParameterBounds

```
int  
getParameterBounds( int axis, real & rStart_, real & rEnd_ ) const
```

**Purpose:** Return current values for the parameter bounds.

**axis (input)** : return bounds for this axis.

**rStart\_, rEnd\_:** bounds.

## 22.28 reparameterize

```
int  
reparameterize(const real & uMin_,  
               const real & uMax_,  
               const real & vMin_ =0.,  
               const real & vMax_ =1.)
```

**Purpose:** Reparameterize the nurb to only use a sub-rectangle of the parameter space. This function can also be used to reverse the direction of the parameterization by choosing uMin > uMax and/or vMin > vMax.

**uMin,uMax (input):** subrange of u values to use, normally  $0 \leq u\text{Min} \neq u\text{Max} \leq 1$

**vMin,vMax (input):** subrange of v values to use, normally  $0 \leq v\text{Min} \neq v\text{Max} \leq 1$  (for domainDimension==2)

**Notes:** this routine just scales the knots to be on a larger interval than [0,1]. Thus when the Mapping is evaluated on [0,1] the result will only be a portion of the original surface.

**Return values:** 0 : success, 1 : failure

## 22.29 transformKnots

```
int  
transformKnots(const real & uScale,  
               const real & uShift,  
               const real & vScale =1.,  
               const real & vShift =0.)
```

**Purpose:** Apply a scaling and shift to the to the knots:  $uScale*uKnots+uShift$ . The scale factors should be positive.

**uScale,uShift (input):** scaling and shift for the knots in the u direction.

**vScale,vShift (input):** scaling and shift for the knots in the v direction. (for domainDimension==2)

### 22.30 elevateDegree

**int**

**elevateDegree(const int increment)**

**Purpose:** Elevate the degree of the nurbs.

**increment (input):** increase the degree of the nurb by this amount  $\_i=0$

**Return values:** 0 : success, 1 : failure

### 22.31 merge

**int**

**merge(NurbsMapping & nurbs, bool keepFailed = true, real eps /\*=-1\*/, bool attemptPeriodic /\*=true\*/ )**

**Purpose:** Try to merge "this" nurbs with the input nurbs. This routine will merge the two NURBS's into one if the endpoint of one matches the end point of the second.

**nurbs (input):** nurbs to merge with

**Return values:** 0 : success, 1 : failure

### 22.32 forcedMerge

**int**

**forcedMerge(NurbsMapping & nurbs )**

**Purpose:** Force a merge of "this" nurbs with the input nurbs. This routine will merge the two NURBS's into one if the endpoint of one matches the end point of the second. If the endpoints do not match, a straight line section is added between the closest end points.

**nurbs (input):** nurbs to merge with

**Return values:** 0 : success, 1 : failure

### 22.33 forcedPeriodic

**int**

**forcePeriodic()**

**Purpose:** force this mapping to be periodic by making the last control points the same as the first ( if the knots are "clamped", eg the knots are 0 0 0 0 ... 1 1 1 1 )

**Return values:** 0 : success, 1 : failure

## 22.34 split

```
int  
split(real uSplit, NurbsMapping &c1, NurbsMapping&c2, bool normalizePieces =true)
```

**Description:** Split a nurb curve into two pieces.

**uSplit (input) :** parameter value to split the curve at

**c1 (output) :** curve on the "left", parameter bounds [0,uSplit]

**c2 (output) :** curve on the "right", parameter bounds [uSplit,1]

**normalizePieces (input) :** in true then normalize each piece

**Returns :** 0 on success, 1 on failure ( uSplit<0 or uSplit>1 )

## 22.35 moveEndpoint

```
int  
moveEndpoint( int end, const RealArray &endPoint, real tol /*=-1*/ )
```

**Description:** Move either the beginning or the end of the curve to endPoint.

## 22.36 numberOfSubCurves

```
int  
numberOfSubCurves() const
```

**Description:** If the Nurb is formed by merging a sequence of Nurbs then function will return that number.  
By default the numberOfSubCurves would be 1 if no Nurbs were merged.

## 22.37 numberOfSubCurvesInList

```
int  
numberOfSubCurvesInList() const
```

**Description:** Return the number of subcurves used to build the Nurb plus the number of hidden curves  
By default the numberOfSubCurvesInList would be 1 if no Nurbs were merged.

## 22.38 subCurve

```
NurbsMapping&  
subCurve(int subCurveNumber)
```

**Description:** If the Nurb is formed by merging a sequence of Nurbs then function will return that Nurbs.  
If the numberOfSubCurves is 1 then the current (full) Nurbs is returned.

## 22.39 subCurveFromList

```
NurbsMapping &  
subCurveFromList(int subCurveNumber)
```

**Description:** Return a nurb curve directly from the list of subcurves. This can be a curve used to generate  
the nurb itself or one of the "hidden" curves. If the numberOfSubCurves is 1 then the current (full)  
Nurbs is returned.

## 22.40 interpolate

```
void  
interpolate(const RealArray & x,  
           const int & option = 0,  
RealArray & parameterization =Overture::nullRealArray(),  
           int degree = 3,  
           ParameterizationTypeEnum parameterizationType  
=parameterizeByChordLength,  
           int numberOfGhostPoints =0)
```

**Purpose:** Define a new NURBS curve that interpolates the points  $x(0:n1,0:r-1)$  OR define a new NURBS surface that interpolates the points  $x(0:n1,0:n2,0:r-1)$  (NEW feature). An even more recent (051031) addition allows the interpolation of a volume defined by the points  $x(0:n1,0:n2,0:n3,0:r-1)$ ; By default the NURBS curve will be parameterized by a the chord length.

**option (input) :** if option==0 then use the array parameterization. if option==1 then return the parameterization used in the array parameterization.

**parameterization\_(0: n1) (input) :** optionally specify the parameterization. These values should start from 0, end at 1 and be increasing. If this argument is not given then the parameterization will be based on chord length. If option==1 then the actual parameterization used will be returned in this array.

**degree (input) :** degree of approximation. Normally a value such as 1,2,3.

**parameterizationType (input) :** the default parameterization (if not user defined) is by chord-length. One can also specify to parameterizeByIndex which means use a uniform parameterization.

**numberOfGhostPoints (input) :** The array x includes this many ghost points on all sides.

## 22.41 interpolate

```
void  
interpolate(Mapping & map, int degree =3,  
           ParameterizationTypeEnum parameterizationType  
=parameterizeByChordLength,  
           int numberOfGhostPoints =0,  
           int *numPointsToInterpolate =NULL)
```

**Description:** Construct a NURBS by interpolating another mapping.

**map (input) :** interpolate this Mapping.

**degree (input) :** degree of NURBS

**parameterization (input) :**

**numberOfGhostPoints (input) :** include this many ghost points

**numPointsToInterpolate (input) :** numPointsToInterpolate[dir], dir=0,1,...,domainDimension-1, optionally specify how many points to interpolate in each direction. By default use the number of grid points from the Mapping "map".

## 22.42 put(fileName)

```
int  
put( const aString & fileName, const FileFormat & fileFormat = xxww)
```

**Description:** put NURBS data into an ascii readable file.

**fileName (input) :** name of the file.

**fileFormat (input) :** specify the file format. (see the comments with the get(const aString&,...) function).

## 22.43 put(FILE\*)

```
int  
put( FILE *file, const FileFormat & fileFormat = xxww)
```

**Description:** Save the NURBS data to an ascii readable file.

**fileFormat (input) :** specify the file format. (see the comments with the get(const aString&,...) function).

## 22.44 get(fileName)

```
int  
get( const aString & fileName, const FileFormat & fileFormat = xxww)
```

**Description:** read NURBS data from an ascii readable file.

**fileName (input) :** get from this file.

**fileFormat (input) :** specify the file format.

Here is the file format for fileFormat=xxww for a surface in 3D

```
domainDimension rangeDimension p1 n1 p2 n2  
uKnot(0) uKnot(1) ... uKnot(m1) --- on possibly multiple lines, at most 10 values per line  
vKnot(0) vKnot(1) ... vKnot(m2)  
x0 x1 x2 ... --- x coords of control pts. on multiple lines, at most 10 per line  
y0 y1 y2 ... --- y coords of control pts.  
z0 z1 z2 ... --- z coords of control pts.  
w0 w1 w2 ... --- weights of control pts.
```

If the domainDimension==1 then leave off p2 and n2. If the rangeDimension is 2 then leave off the z values. Here m1=n1+p1+1 and m2=n2+p2+1.

Here is the file format for fileFormat=xwxw for a surface in 3D

```
domainDimension rangeDimension p1 n1 p2 n2  
uKnot(0) uKnot(1) ... uKnot(m1) --- on possibly multiple lines, at most 10 values per line  
vKnot(0) vKnot(1) ... vKnot(m2)  
x0 y0 z0 w0 --- control point 0  
x1 y1 z1 w1 --- control point 1  
x1 y1 z1 w1 --- control point 2  
...
```

If the domainDimension==1 then leave off p2 and n2. If the rangeDimension is 2 then leave off the z values.

#### 22.45 put(FILE \*)

```
int  
get( FILE *file, const FileFormat & fileFormat = xxww)
```

**Description:** read NURBS data from an ascii readable file.

**file (input) :** get from this file.

**fileFormat (input) :** specify the file format. (see the comments with the get(const aString&,...) function).

#### 22.46 getOrder

```
int  
getOrder( int axis =0) const
```

**Purpose:** Return the order, p.

#### 22.47 getNumberOfKnots

```
int  
getNumberOfKnots( int axis =0) const
```

**Purpose:** Return the number of knots, m+1.

#### 22.48 getNumberOfControlPoints

```
int  
getNumberOfControlPoints( int axis =0) const
```

**Purpose:** Return the number of control points, n+1.

#### 22.49 buildSubCurves

```
int  
buildSubCurves( real angle =60.)
```

**Purpose:** Split a NURBS curve at corners into sub-curves. Currently this only applies if the order of the NURBS is 1 (piece-wise linear).

**angle (input) :** divide the curve at points where the tangent changes by more than this angle (degrees)

#### 22.50 truncateToDomainBounds

```
int  
truncateToDomainBounds()
```

**Purpose:** clip the knots and control polygon to the bounds set in rstart and rend

## 22.51 toggleSubCurveVisibility

**int**  
**toggleSubCurveVisibility( int sc )**

**Description:** Toggle a subcurve's "visibility", a visible subcurve is accessible through NurbsMapping::subCurve(..) method an invisible subcurve is only accessible through NurbsMapping::subCurveFromList()

**sc (input)** : the subcurve to toggle Returns : the new subcurve number NOTES : this will reorder the subcurves in the subCurves array

## 22.52 isSubCurveHidden

**bool**  
**isSubCurveHidden( int sc )**

**Description:** find out if a subcurve is hidden or not, returns true if hidden, false if visible

**sc (input)** : the subcurve to querry

## 22.53 isSubCurveOriginal

**bool**  
**isSubCurveOriginal( int sc )**

**Description:** find out if a subcurve is marked as "original"

**sc (input)** : the subcurve to querry

## 22.54 toggleSubCurveOriginal

**void**  
**toggleSubCurveOriginal( int sc )**

**Description:** toggle the "original" status on a subcurve, "original" is just a marker used to distinguish the original subcurves used to build this nurb from subsequent modifications.

**sc (input)** : the subcurve to alter

## 22.55 addSubCurve

**int**  
**addSubCurve(NurbsMapping &nurbs)**

**Description:** Add a subcurve to this mapping. Note that the nurb is copied and is set to visible. The "original" marker is set to false;

**Returns :** the index of the new curve in the list of visible curves

## 22.56 deleteSubCurve

```
int  
deleteSubCurve(int sc)
```

**Description:** Delete a subcurve from the list of curves. Note this shifts the subcurve list making previous indices invalid

**sc (input):** the curve to delete

**Returns :** 0 on success

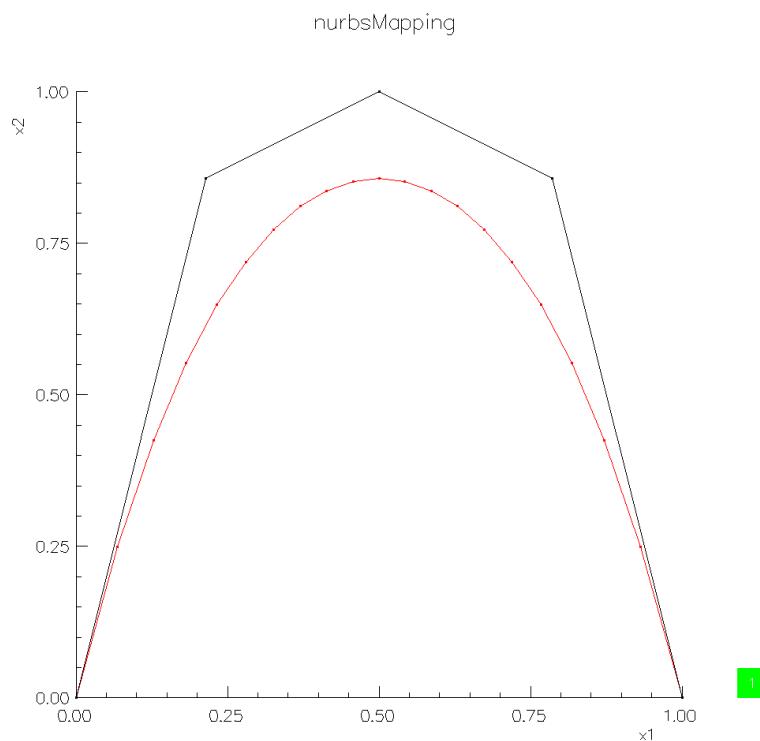
## 22.57 update

```
int  
update( MappingInformation & mapInfo )
```

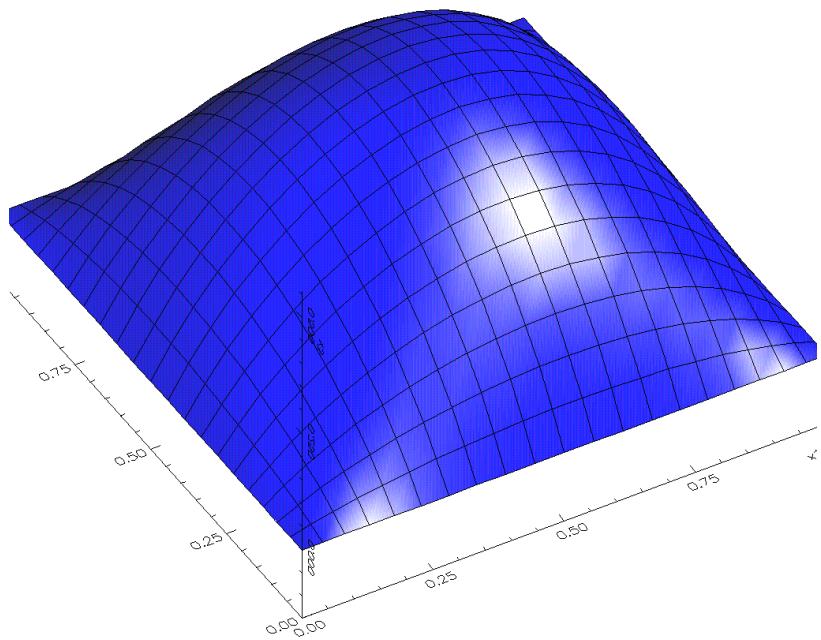
**Purpose:** Interactively create and/or change the nurbs mapping.

**mapInfo (input):** Holds a graphics interface to use.

## 22.58 Examples



A 2D NURBS curve defined by specifying control points.



A 3D NURBS surface defined by specifying control points.

## 23 OffsetShell: Define mappings to build a grid around a shell or plate.

The OffsetShell class starts with a 3D surface defining a thin shell or plate (this is called the reference surface). An offset surface will be built by translating the reference surface a small amount in a user specified direction. An edge surface will then be constructed that joins the reference and offset surfaces with a rounded edge that overlaps both surfaces.

Volume grids can be built for the reference, offset and edge surfaces.

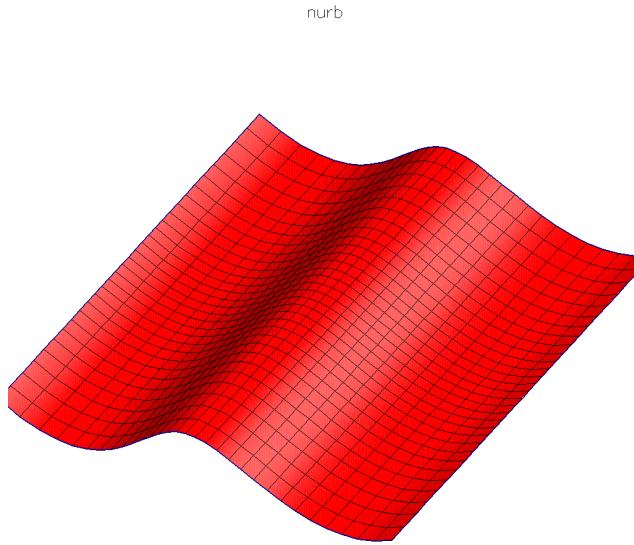


Figure 16: The original reference surface for a flying carpet given to OffsetShell.

### 23.1 Defining the edge surface : an overlapping round

The first step in defining the edge surface is to define a smooth curve on the reference surface that smoothly follows the boundary of the reference surface but is offset a small amount inside the boundary. To define this curve we first construct a smooth curve,  $\mathbf{c}_0(t) = (r_0, r_1)(t)$ , near the boundary of the unit square:

$$\mathbf{c}_0(t) = \begin{cases} (1 - \Delta_0, .5 + \xi) & 0 \leq \xi \leq t_0 \\ (1 - \Delta_1 - \xi, 1 - \Delta_0) & t_1 \leq \xi \leq t_2 \\ (\Delta_0, 1 - \Delta_1 - \xi) & t_3 \leq \xi \leq t_4 \\ (\Delta_1 + \xi, \Delta_0) & t_5 \leq \xi \leq t_6 \\ (1 - \Delta_0, \Delta_1 + \xi) & t_7 \leq \xi \leq t_8 \end{cases}$$

$$t_0 = .5 - \Delta_1$$

The curve on the reference surface is defined as  $\mathbf{c}(t) = \mathbf{R}(\mathbf{c}_0(t))$  where  $\mathbf{x}(\mathbf{r}) = \mathbf{R}(\mathbf{r})$  defines the reference surface.

Given the edge curve  $\mathbf{c}(t)$  we can define the tangent vector  $\mathbf{t}(t)$  to the curve as well as the vector normal

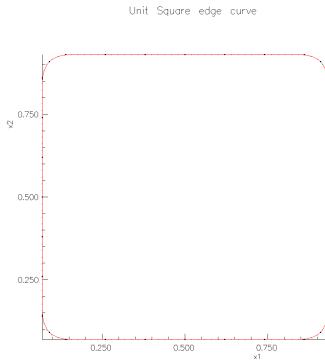


Figure 17: An edge curve  $\mathbf{c}_0(t)$  is defined on the unit square.

to the reference surface,  $\mathbf{n}(t)$ .

$$\begin{aligned}\mathbf{t}(t) &= \dot{\mathbf{c}} / \|\dot{\mathbf{c}}\| \\ \mathbf{n}(t) &= \frac{\partial \mathbf{R}}{\partial r_0}(\mathbf{c}_0(t)) \times \frac{\partial \mathbf{R}}{\partial r_1}(\mathbf{c}_0(t))\end{aligned}$$

Given  $\mathbf{t}(t)$  and  $\mathbf{n}(t)$  we define the direction vector,  $\mathbf{d}(t)$ , at each point on the edge curve to be orthogonal to these two vectors and point towards the boundary of the reference surface,

$$\mathbf{d}(t) = \mathbf{t} \times \mathbf{n} / \|\mathbf{t} \times \mathbf{n}\|$$

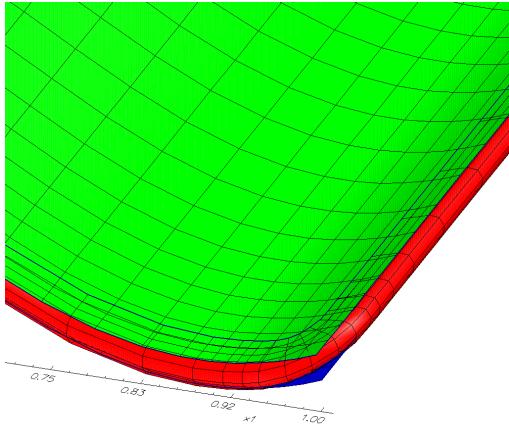


Figure 18: The reference surface, offset surface, and edge surface near a corner.

The edge surface defined as 3 sections, an initial and final portion that lie on the reference surface connected by half a circle:

$$\begin{aligned}\mathbf{e}(t, s) &= \begin{cases} \mathbf{c}(t) + a_0 s \mathbf{d}(t) & 0 \leq s \leq s_0 \\ \mathbf{e}(t, s_0) + .5(1 - \cos(\theta))\mathbf{s} + \sin(\theta)a_1\mathbf{d}(t) & s_0 < s \leq s_1 \\ \mathbf{e}(t, s_1) - a_0(1 - s)\mathbf{d}(t) & s_1 < s \leq 1 \end{cases} \\ \theta &= \pi(s - s_0)/(s_1 - s_0)\end{aligned}$$

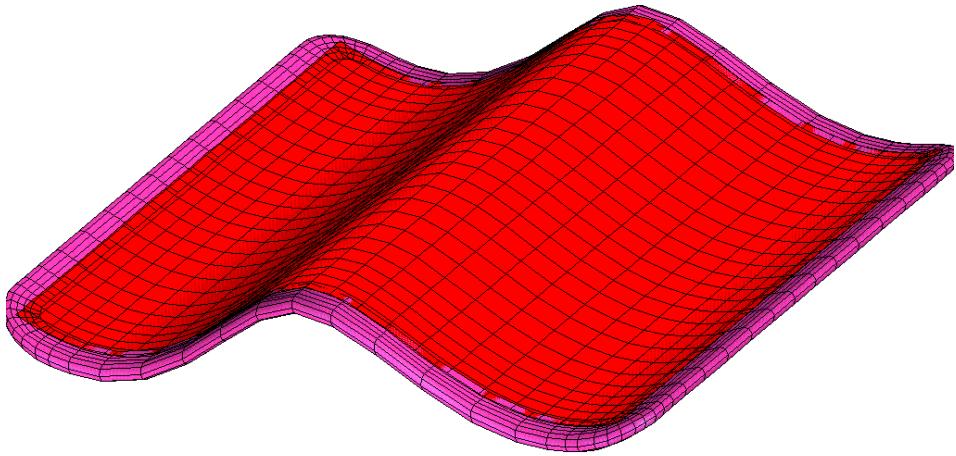


Figure 19: The overlapping grid for the flying carpet in a box.

### 23.2 Member function descriptions

#### 23.3 Constructor

**OffsetShell()**

**Description:** Starting from a 3D reference surface build an offset surface and joining edge surface.

#### 23.4 buildOffsetMappings

**int**

```
buildOffsetMappings( GenericGraphicsInterface & gi,  
                      GraphicsParameters & parameters, MappingInformation & mapInfo )
```

**Description:** Given a reference surface, build an offset surface, and then an edge surface to join two.  
Build volume grids for the reference surface, offset surface and edge surface.

**referenceSurface (input) :**

**offsetSurface (output) :**

**edgeSurface, edgeVolume (output):**

**referenceVolume, offsetVolume (output) :**

### 23.5 generateVolumeGrids

```
int  
generateVolumeGrids( GenericGraphicsInterface & gi,  
                      GraphicsParameters & parameters, MappingInformation & mapInfo  
)
```

**Description:** Build the volume grids.

### 23.6 createOffsetMappings

```
int  
createOffsetMappings( MappingInformation & mapInfo )
```

**Description:** Interactively build grids for a thin shell.

## 24 OrthographicTransform : define an orthographic transform

This mapping is used to create an orthographic patch to remove a spherical-polar singularity or a cylindrical polar singularity (i.e. a singular may occur at the end of a mapping defined in cylindrical coordinates when the cross-sections converge to a point). Normally one would use the ReparamertizationTransform mapping to construct the Orthographic patch.

### 24.1 Description

The orthographic transformation is a mapping from parameter space to parameter space. There are two forms to this mapping, the first can be used to reparameterize a mapping with a spherical polar singularity and the second can be used for a cylindrical mapping with a polar singularity.

#### 24.1.1 Orthographic transform to reparameterize a spherical-polar singularity

This form of the orthographic mapping transforms into spherical polar coordinates

$$(r_1, r_2) \rightarrow (t_1, t_2) = \left(\frac{\phi}{\pi}, \frac{\theta}{2\pi}\right)$$

and is defined by

$$\begin{aligned} s_1 &= (r_1 - \frac{1}{2})s_a, & s_2 &= (r_2 - \frac{1}{2})s_b, & \sigma^2 &= s_1^2 + s_2^2 \\ \cos \phi &= \pm \frac{1 - \sigma^2}{1 + \sigma^2}, & \sin \phi &= \frac{2\sigma}{1 + \sigma^2}, & \cos \theta &= \frac{s_1}{\sigma}, & \sin \theta &= \pm \frac{s_2}{\sigma}. \\ (t_1, t_2) &= \left(\frac{\phi}{\pi}, \frac{\theta}{2\pi}\right) \end{aligned}$$

The upper sign (+) is used for a reparametrization covering the north pole and the lower sign (-) for the south pole.

The derivatives are returned as

$$\frac{\partial t_1}{\partial r_1} = \frac{s_1}{(1 + \sigma^2)\sigma} \frac{\pm 2s_a}{\pi} \quad (13)$$

$$\frac{\partial t_1}{\partial r_2} = \frac{s_2}{(1 + \sigma^2)\sigma} \frac{\pm 2s_b}{\pi} \quad (14)$$

$$\sin(\phi) \frac{\partial t_2}{\partial r_1} = -\frac{s_2}{(1 + \sigma^2)\sigma} \frac{\pm 2s_a}{\pi} \quad (15)$$

$$\sin(\phi) \frac{\partial t_2}{\partial r_2} = +\frac{s_1}{(1 + \sigma^2)\sigma} \frac{\pm 2s_b}{\pi} \quad (16)$$

so that when this mapping is composed with a mapping in spherical-polar form the  $\sin(\phi)$  terms will cancel nicely to remove the removable singularity.

The inverse of the mapping  $(t_1, t_2) \rightarrow (r_1, r_2)$  is defined by

$$\phi = \pi t_1, \quad \theta = 2\pi t_2, \quad s_1 = \frac{\sin \phi}{1 \pm \cos \phi} \cos \theta, \quad s_2 = \pm \frac{\sin \phi}{1 \pm \cos \phi} \sin \theta.$$

$$r_1 = \frac{s_1}{s_a} + \frac{1}{2}, \quad r_2 = \frac{s_2}{s_b} + \frac{1}{2}$$

The derivatives are returned as

$$\begin{aligned} \frac{\partial r_1}{\partial t_1} &= \frac{\cos(\theta)}{(1 \pm \cos(\phi))} \frac{\pm \pi}{s_a} \\ \frac{\partial r_2}{\partial t_1} &= \frac{\sin(\theta)}{(1 \pm \cos(\phi))} \frac{\pi}{s_b} \\ \frac{1}{\sin(\phi)} \frac{\partial r_1}{\partial t_2} &= \frac{\sin(\theta)}{(1 \pm \cos(\phi))} \frac{-2\pi}{s_a} \\ \frac{1}{\sin(\phi)} \frac{\partial r_2}{\partial t_2} &= \frac{\cos(\theta)}{(1 \pm \cos(\phi))} \frac{\pm 2\pi}{s_b} \end{aligned}$$

#### 24.1.2 Orthographic transform to reparameterize a cylindrical polar singularity

This form of the orthographic mapping transforms into cylindrical coordinates

$$(r_1, r_2) \rightarrow (t_1, t_2) = (s, \frac{\theta}{2\pi})$$

and is defined by

$$\begin{aligned} s_1 &= (r_1 - \frac{1}{2})s_a, \quad s_2 = (r_2 - \frac{1}{2})s_b \quad \sigma^2 = s_1^2 + s_2^2 \\ t_1 &= \pm \frac{1}{2} \frac{1 - \sigma^2}{1 + \sigma^2} + \frac{1}{2}, \quad (2\pi t_2) = \pm s_2/s_1, \quad r = \frac{2\sigma}{1 + \sigma^2} \end{aligned}$$

The derivatives are returned as

$$\begin{aligned} -\frac{1}{r} \frac{\partial t_1}{\partial r_1} &= \frac{s_1}{(1 + \sigma^2)\sigma} \pm s_a \\ -\frac{1}{r} \frac{\partial t_1}{\partial r_2} &= \frac{s_2}{(1 + \sigma^2)\sigma} \pm s_b \\ r \frac{\partial t_2}{\partial r_1} &= -\frac{s_2}{(1 + \sigma^2)\sigma} \frac{\pm s_a}{\pi} \\ r \frac{\partial t_2}{\partial r_2} &= +\frac{s_1}{(1 + \sigma^2)\sigma} \frac{\pm s_b}{\pi} \end{aligned}$$

so that when this mapping is composed with a mapping in cylindrical coordinates form the terms will cancel nicely to remove the removable singularity.

The inverse of the mapping  $(t_1, t_2) \rightarrow (r_1, r_2)$  is defined by

$$\zeta = 2t_1 - 1, \quad r = \sqrt{1 - \zeta^2} \quad (\theta) = \frac{\pm s_2}{s_1}, \quad s_1 = \frac{r}{1 \pm \zeta} \cos \theta, \quad s_2 = \pm \frac{r}{1 \pm \zeta} \sin \theta.$$

$$r_1 = \frac{s_1}{s_a} + \frac{1}{2}, \quad r_2 = \frac{s_2}{s_b} + \frac{1}{2}$$

The derivatives are returned as

$$\begin{aligned} -r \frac{\partial r_1}{\partial t_1} &= \frac{\cos(\theta)}{(1 \pm \zeta)} \frac{\pm 2}{s_a} \\ -r \frac{\partial r_2}{\partial t_1} &= \frac{\sin(\theta)}{(1 \pm \zeta)} \frac{2}{s_b} \\ \frac{1}{r} \frac{\partial r_1}{\partial t_2} &= \frac{\sin(\theta)}{(1 \pm \zeta)} \frac{-2\pi}{s_a} \\ \frac{1}{r} \frac{\partial r_2}{\partial t_2} &= \frac{\cos(\theta)}{(1 \pm \zeta)} \frac{\pm 2\pi}{s_b} \end{aligned}$$

## 25 Member functions

### 25.1 Default Constructor

```
OrthographicTransform( const real sa_ = 1.,
                      const real sb_ = 1.,
                      const Pole pole_ = northPole)
```

**Purpose:** The `OrthographicTransform` is used by the `ReparameterizationTransform` to remove a polar singularity.

**sa\_, sb\_ (input)** : parameters that specify the dimensions of the plane that is projected onto the sphere in the orthographic transform.

**pole (input)** : reparameterize the `northPole` or the `southPole`.

### 25.2 setAngularAxis

```
int
setAngularAxis( const int & tAxis_ )
```

**Purpose:** Specify which axis (axis1 or axis2) corresponds to the angular ( $\theta$ ) direction of the mapping that will have an orthographic patch on it. The  $\phi$  direction will be axis1 if `tAxis=axis2` or axis2 if `tAxis=axis1`.

**tAxis\_ (input)** : axis1 (0) or axis2 (1).

### 25.3 setPole

```
int
setPole( const Pole & pole_ )
```

**Purpose:** Specify which pole to reparameterize.

**pole (input)** : reparameterize the `northPole` or the `southPole`.

## 25.4 setSize

```
int  
setSize( const real & sa_,  
         const real & sb_ )
```

**Purpose:** Specify the size of the orthographic patch.

**sa\_, sb\_ (input) :** parameters that specify the dimensions of the plane that is projected onto the sphere in the orthographic transform.

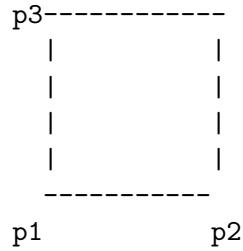
## 25.5 Class PlaneMapping

This mapping defines a plane or rhombus in three-dimensions.

## 25.6 Constructor

```
PlaneMapping(const real & x1 =0. /*/, const real & y1 /*=0. /*/, const real & z1 /*=0.,
               const real & x2 =1. /*/, const real & y2 /*=0. /*/, const real & z2 /*=0.,
               const real & x3 =0. /*/, const real & y3 /*=1. /*/, const real & z3 /*=0.)
```

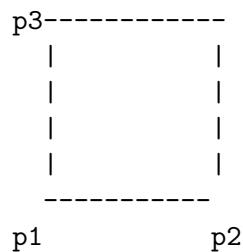
**Purpose:** Default Constructor, define a plane (or rhomboid) by three (non-collinear) points,  $p1=(x1,y1,z1)$ ,  $p2=(x2,y2,z2)$ ,  $p3=(x3,y3,z3)$  arranged as:



## 25.7 setPoints

```
int
setPoints(const real & x1 =0. /*/, const real & y1 /*=0. /*/, const real & z1 /*=0.,
           const real & x2 =1. /*/, const real & y2 /*=0. /*/, const real & z2 /*=0.,
           const real & x3 =0. /*/, const real & y3 /*=1. /*/, const real & z3 /*=0.)
```

**Purpose:** Set the corners of the plane or rhomboid. The plane (or rhomboid) is defined by three (non-collinear) points,  $p1=(x1,y1,z1)$ ,  $p2=(x2,y2,z2)$ ,  $p3=(x3,y3,z3)$  arranged as:



## 26 QuadraticMapping: define a quadratic curve or surface.

Use this mapping to define a quadratic curve or surface.

A parabola (curve in 2D) is defined by

$$x_0 = c_{0x} + c_{1x}r_0$$
$$x_1 = a_{00} + a_{10}x_0 + a_{20}x_0^2$$

A 3d paraboloid (surface) is defined by

$$x_0 = c_{0x} + c_{1x}r_0$$
$$x_1 = c_{0y} + c_{1y}r_1$$
$$x_2 = a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2$$

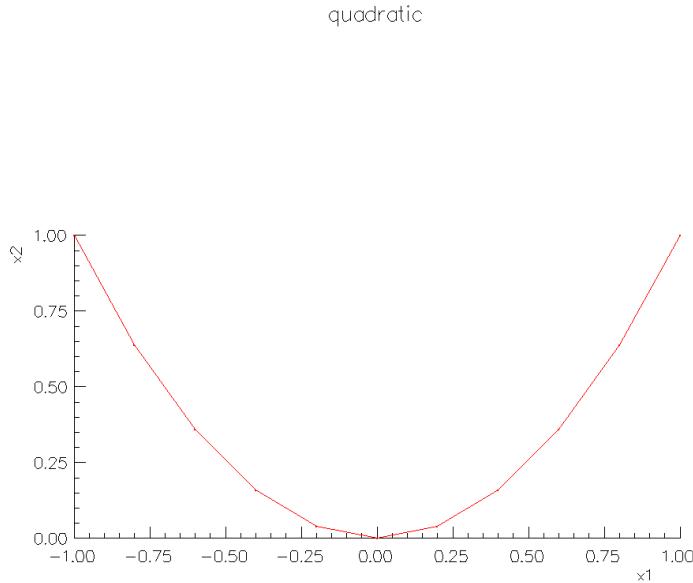
A hyperbola (2d curve) is defined by

$$x_0 = c_{0x} + c_{1x}r_0$$
$$x_1 = \pm(a_{00} + a_{10}x_0 + a_{20}x_0^2)^{1/2}$$

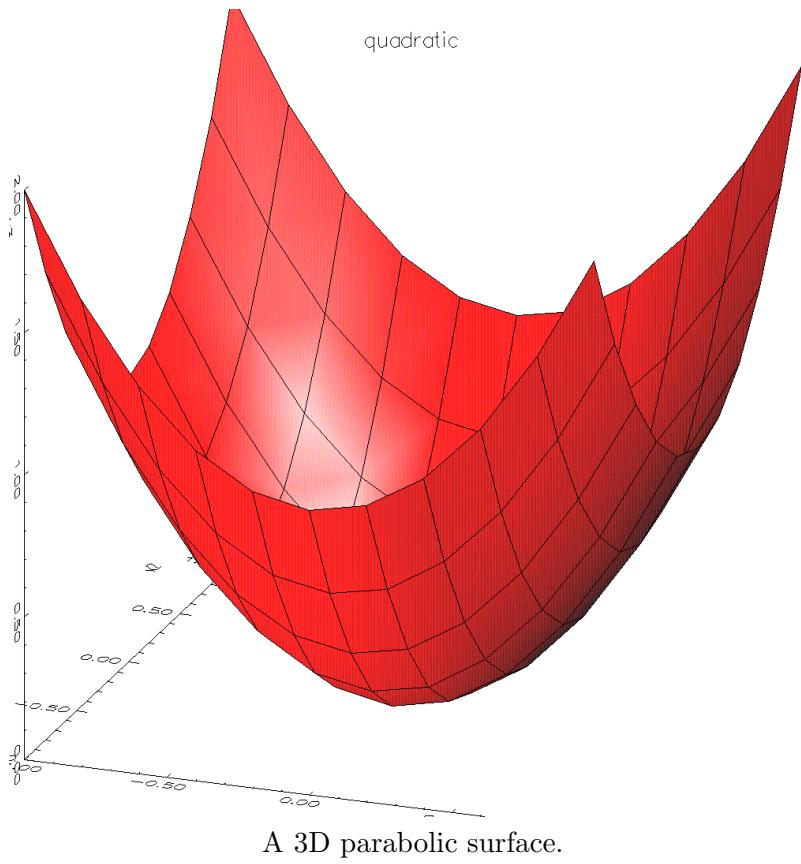
A 3d hyperboloid (surface) is defined by

$$x_0 = c_{0x} + c_{1x}r_0$$
$$x_1 = c_{0y} + c_{1y}r_1$$
$$x_2 = \pm(a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2)^{1/2}$$

### 26.1 Examples



A 2D parabola.



A 3D parabolic surface.

## 26.2 Constructor

**QuadraticMapping()**

**Description:** Define a quadratic curve or surface (parabola or hyperbola)

## 26.3 setQuadraticParameters

**int**

**chooseQuadratic( QuadraticOption option,**  
**int rangeDimension\_ =2)**

**Description:** Specify the parameters for a quadratic function:

**option (input):** An option from the enum QuadraticOption: **parabola** or **hyperbola**.

**rangeDimension\_ (input):** 2 or 3

## 26.4 setParameters

**int**

**setParameters(real c0x,**  
**real c1x,**  
**real c0y,**  
**real c1y,**  
**real a00,**  
**real a10,**  
**real a01,**

```

real a20,
real a11,
real a02,
real signForHyperbola_ = 1.)

```

**Description:** Specify the parameters for a quadratic function:

A parabola (curve in 2D) is defined by

$$x_0 = c_{0x} + c_{1x} * r_0$$

$$x_1 = a_{00} + a_{10}x_0 + a_{20}x_0^2$$

A 3d paraboloid (surface) is defined by

$$x_0 = c_{0x} + c_{1x} * r_0$$

$$x_1 = c_{0y} + c_{1y} * r_1$$

$$x_2 = a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2$$

A hyperbola (2d curve) is defined by

$$x_0 = c_{0x} + c_{1x} * r_0$$

$$x_1 = \pm(a_{00} + a_{10}x_0 + a_{20}x_0^2)^{1/2}$$

A 3d hyperboloid (surface) is defined by

$$x_0 = c_{0x} + c_{1x} * r_0$$

$$x_1 = c_{0y} + c_{1y} * r_1$$

$$x_2 = \pm(a_{00} + a_{10}x_0 + a_{01}x_1 + a_{20}x_0^2 + a_{11}x_0x_1 + a_{02}x_1^2)^{1/2}$$

**a00\_-, a10\_-,..., (input):** parameters in above formula.

## 27 ReductionMapping: create a Mapping from the face or edge of an existing Mapping

### 27.1 Description

The ReductionMapping can be used to make a new Mapping from the face or edge of another Mapping, thus reducing the domain dimension of the original mapping.

In general the new Mapping is defined by fixing one or more of the r-coordinates of the original mapping.

For example if we have a mapping from  $\mathbf{R}^3 \rightarrow \mathbf{R}^3$ ,  $\mathbf{x}(r_0, r_1, r_2)$ , we can define a new Mapping from  $\mathbf{R}^2 \rightarrow \mathbf{R}^3$  by the surface  $\mathbf{x}_r(r_0, r_1) = \mathbf{x}(r_0, r_1, r_{2a})$  for some fixed value  $r_{2a}$ . We could also define the curve in 3-space by  $\mathbf{x}_r(r_0) = \mathbf{x}(r_{0a}, r_0, r_{2a})$  for some fixed values  $r_{0a}$  and  $r_{2a}$ .

### 27.2 Constructor

**ReductionMapping()**

**Purpose:** Default Constructor

### 27.3 Constructor

```
ReductionMapping(Mapping & mapToReduce,  
                  const real & inactiveAxis1Value =0.,  
                  const real & inactiveAxis2Value =-1.,  
                  const real & inactiveAxis3Value =-1.)
```

**Purpose:** Create a reduction mapping.

**mapToReduce (input):** reduce the domain dimension of this mapping.

**inactiveAxis1Value (input):** if this value is between [0,1] then the r value for axis1 will be fixed to this value and axis1 will become an in-active axis; otherwise axis1 will remain active.

**inactiveAxis2Value (input):** fix an r value for axis2. See comments for inactiveAxis1Value.

**inactiveAxis3Value (input):** fix an r value for axis3. See comments for inactiveAxis1Value.

### 27.4 Constructor

```
ReductionMapping(Mapping & mapToReduce,  
                  const int & inactiveAxis,  
                  const real & inactiveAxisValue )
```

**Purpose:** Create a reduction mapping.

**mapToReduce (input):** reduce the domain dimension of this mapping.

**inactiveAxis (input):** This is the inactive axis.

**inactiveAxisValue (input):** This is the value of the inactive axis in [0,1].

## 27.5 set

```
int  
set(Mapping & mapToReduce,  
    const real & inactiveAxis1Value =0. ,  
    const real & inactiveAxis2Value =-1.,  
    const real & inactiveAxis3Value =-1.)
```

**Purpose:** Set parameters for a reduction mapping.

**mapToReduce (input):** reduce the domain dimension of this mapping.

**inactiveAxis1Value (input):** if this value is between [0,1] then the r value for axis1 will be fixed to this value and axis1 will become an in-active axis; otherwise axis1 will remain active.

**inactiveAxis2Value (input):** fix an r value for axis2. See comments for inactiveAxis1Value.

**inactiveAxis3Value (input):** fix an r value for axis3. See comments for inactiveAxis1Value.

## 27.6 set

```
int  
set(Mapping & mapToReduce,  
    const int & inactiveAxis,  
    const real & inactiveAxisValue )
```

**Purpose:** Set parameters for a reduction mapping.

**mapToReduce (input):** reduce the domain dimension of this mapping.

**inactiveAxis (input):** This is the inactive axis.

**inactiveAxisValue (input):** This is the value of the inactive axis in [0,1].

## 27.7 setInActiveAxes

```
int  
setInActiveAxes( const real & inactiveAxis1Value =0.,  
                  const real & inactiveAxis2Value =-1.,  
                  const real & inactiveAxis3Value =-1.)
```

**Purpose:** Specify the in-active axes.

**inactiveAxis1Value (input):** if this value is between [0,1] then the r value for axis1 will be fixed to this value and axis1 will become an in-active axis; otherwise axis1 will remain active.

**inactiveAxis2Value (input):** fix an r value for axis2. See comments for inactiveAxis1Value.

**inactiveAxis3Value (input):** fix an r value for axis3. See comments for inactiveAxis1Value.

## 27.8 setInActiveAxes

```
int  
setInActiveAxes(const int & inactiveAxis,  
                 const real & inactiveAxisValue )
```

**Purpose:** Set parameters for a reduction mapping.

**inactiveAxis (input):** This is the inactive axis.

**inactiveAxisValue (input):** This is the value of the inactive axis in [0,1].

## 28 ReorientMapping: reorder the domain coordinates of an existing mapping

### 28.1 Description

The ReorientMapping is used by the ReparameterizationTransform to reorder the domain coordinates. For example, given a Mapping that maps the parameter space (domain) variables  $(r_1, r_2, r_3)$  into  $\mathbf{x} = \mathbf{g}(r_1, r_2, r_3)$  one can make a new mapping  $\mathbf{x} = \tilde{\mathbf{g}}(r_1, r_2, r_3) = \mathbf{g}(r_3, r_1, r_2)$ . Thus the parameterization of the Mapping will be changed but the result of the Mapping will “look the same”.

This Mapping is defined very simply by

$$x_n = r_{\sigma(n)} \quad n = 1, 2, 3$$

where  $(\sigma(1), \sigma(2), \sigma(3))$  is a permutation of  $(1, 2, 3)$ .

### 28.2 Default Constructor

```
ReorientMapping(const int dir1_=0 /*=0*/, const int dir2_ /*=1*/, const int dir3_ /*=2,
const int dimension =2)
```

**Purpose:** Default Constructor The Reorient is a Mapping from parameter space to parameter space that can be used to reorder the domain space variables  $(r_1, r_2, r_3)$  and is defined by

$$\begin{aligned}x(I, axis1) &= r(I, dir1) \\x(I, axis2) &= r(I, dir2) \\x(I, axis3) &= r(I, dir3)\end{aligned}$$

**dir1\_,dir2\_,dir3\_ (input):** 3 integers that are a permutation of  $(0,1,2)$

**dimension (input):** define the domain and range dimension (which are equal).

### 28.3 setOrientation

```
int
setOrientation(const int dir1_, const int dir2_, const int dir3_ = -1)
```

**Purpose:** Define the reordering of the domain space variables.

**dir1\_,dir2\_,dir3\_ (input):** 3 integers that are a permutation of  $(0,1,2)$

### 28.4 getOrientation

```
int
getOrientation(int & dir1_, int & dir2_, int & dir3_) const
```

**Purpose:** Return the current ordering.

**dir1\_,dir2\_,dir3\_ (output):** 3 integers that are a permutation of  $(0,1,2)$

## 29 ReparameterizationTransform: reparameterize an existing mapping (e.g. remove a polar singularity)

### 29.1 Description

The `ReparameterizationTransform` can reparameterize a given Mapping in one of the following ways:

**Orthographic:** remove a polar singularity by using a orthographic projection to define a new patch over the singularity.

**Restriction:** restrict the parameter space to a sub-rectangle of the original parameter space. Use this, for example, to define a refined patch in an adaptive grid.

**Reorder domain coordinates:** uses the `ReorientMapping` to reorder the domain (i.e. parameter space) coordinates of a mapping. This can be useful for debugging purposes in order to see that the results of some application are independent of the order of  $(r_1, r_2, r_3)$ .

### 29.2 Reparameterizing a spherical-polar or cylindrical-polar singularity

The `orthographic` reparameterization can be used to remove a spherical polar singularity or cylindrical polar singularity by defining a new patch over the singularity.

In order for the `orthographic` reparameterization to be applicable the Mapping to be reparameterized must have the following properties:

**a polar singularity :** The mapping must have a polar singularity at  $r_1 = 0$  or  $r_1 = 1$  and the coordinate direction  $r_2$  must be the angular ( $\theta$ ) variable. ( $r_3$  would be the radial direction). If the mapping has such a singularity then one should indicate this property with the call of the form

```
setTypeOfCoordinateSingularity( side,axis,polarSingularity );
```

**can be evaluated in spherical (or cylindrical) coordinates :** this property should be set with a call

```
setCoordinateEvaluationType( spherical,TRUE );
```

or

```
setCoordinateEvaluationType( cylindrical,TRUE );
```

A Mapping that can be evaluated in spherical or cylindrical coordinates must define the `map` and `basicInverse` functions to optionally return the derivatives in a special form. For spherical coordinates the derivatives of the mapping are computed as

$$\left( \frac{\partial x_i}{\partial r_1}, \frac{1}{\sin(\phi)} \frac{\partial x_i}{\partial r_2}, \frac{\partial x_i}{\partial r_3} \right)$$

and the derivatives of the inverse mapping as

$$\left( \frac{\partial r_1}{\partial x_i}, \sin(\phi) \frac{\partial r_2}{\partial x_i}, \frac{\partial r_3}{\partial x_i} \right).$$

Here  $\phi = \pi r_0$  is the parameter (latitude) for which the spherical singularities occur at  $\phi = 0, \pi$ . See the implementation of the `SphereMapping` or the `RevolutionMapping` for two examples.

For cylindrical coordinates the derivatives of the mapping are computed as

$$\left( -\rho \frac{\partial x_i}{\partial r_1}, \frac{1}{\rho} \frac{\partial x_i}{\partial r_2}, \frac{\partial x_i}{\partial r_3} \right)$$

and the derivatives of the inverse mapping as

$$\left( \frac{-1}{\rho} \frac{\partial r_1}{\partial x_i}, \rho \frac{\partial r_2}{\partial x_i}, \frac{\partial r_3}{\partial x_i} \right).$$

Here the variable  $\rho$ , defined by

$$\zeta = 2r_0 - 1.$$

$$\rho = \sqrt{1 - r_0^2}$$

goes to zero at the singularity. See the implementation of the ellipse in `CrossSectionMapping.C` for an example of cylindrical coordinates.

### 29.3 Default Constructor

#### `ReparameterizationTransform()`

**Purpose:** Default Constructor The `ReparameterizationTransform` can reparameterize a given Mapping in one of the following ways:

**orthographic:** Remove a polar singularity by using a orthographic projection to define a new patch over the singularity.

**restriction:** restrict the parameter space to a sub-rectangle of the original parameter space. Use this, for example, to define a refined patch in an adaptive grid.

**equidistribution:** reparameterize a curve in 2D or 3D so as to equi-distribute a weighted sum of arclength and curvature.

### 29.4 Constructor(`Mapping`,`ReparameterizationTypes`)

```
ReparameterizationTransform(Mapping & map,
                           const ReparameterizationTypes type =
                           defaultReparameterization)
```

**Description:** Constructor for a Reparameterization.

**map (input) :** mapping to reparameterize.

**type (input) :**

### 29.5 Constructor(`MappingRC`,`ReparameterizationTypes`)

```
ReparameterizationTransform(MappingRC & mapRC,
                           const ReparameterizationTypes type =
                           defaultReparameterization)
```

**Description:** Constructor for a Reparameterization. See the comments in the constructor member function

## 29.6 constructor(MappingRC,ReparameterizationTypes)

```
void  
constructor(Mapping & map, const ReparameterizationTypes type)
```

**Description:** This is a protected routine, used internally. Constructor for a Reparameterization. This constructor will check to see if you are trying to reparameterize a Mapping that is already the same type of reparameterization of another mapping. For example you may be making a sub-mapping (restriction) of a sub-mapping. In this case this constructor will eliminate the multiple restriction operations and replace it by a single restriction. You should then use the scaleBounds member function to define a new restriction. This function will scale the bounds found in map.

## 29.7 constructorForMultipleReparams

```
void  
constructorForMultipleReparams(ReparameterizationTransform & rtMap )
```

**Description:** \*\*This is a protected routine\*\* If you want to reparameterize a mapping that is already Reparameterized then use this constructor. It will replace multiple reparams of the same type with just one reparam

**Notes:**

## 29.8 scaleBound

```
int  
scaleBounds(const real ra =0.,  
            const real rb =1.,  
            const real sa =0.,  
            const real sb =1.,  
            const real ta =0.,  
            const real tb =1.)
```

**Description:** Scale the current bounds for a restriction Mapping. See the documentation for the `RestrictionMapping` for further details.

`ra,rb,sa,sb,ta,tb` (input):

## 29.9 getBounds

```
int  
getBounds(real & ra, real & rb, real & sa, real & sb, real & ta, real & tb ) const
```

**Description:** Get the bounds for a restriction mapping. `RestrictionMapping` for further details.

`ra,rb,sa,sb,ta,tb` (output):

## 29.10 setBounds

```
int  
setBounds(const real ra =0.,  
          const real rb =1.,  
          const real sa =0.,  
          const real sb =1.,
```

```

const real ta =0.,
const real tb =1.)

```

**Description:** Set absolute bounds. See the documentation for the `RestrictionMapping` for further details.

**ra,rb,sa,sb,ta,tb (input):**

#### 29.10.1 `getBoundsForMultipleReparameterizations`

```

int
getBoundsForMultipleReparameterizations( real mrBounds[6] ) const

```

**Description:** Get the bounds for multiple reparameterizations. This routine will usually only be called by the Grid class.

**mrBounds (output):**

#### 29.10.2 `setBoundsForMultipleReparameterizations`

```

int
setBoundsForMultipleReparameterizations( real mrBounds[6] )

```

**Description:** Set the bounds for multiple reparameterizations. This routine will usually only be called by the Grid class.

**mrBounds (input):**

#### 29.10.3 `setEquidistributionParameters`

```

int
setEquidistributionParameters(const real & arcLengthWeight_ /* =1.*/,
                           const real & curvatureWeight_ /* =0.*/,
                           const int & numberOfSmooths = 3)

```

**Description:** Set the ‘arclength’ parameterization parameters. The parameterization is chosen to redistribute the points to resolve the arclength and/or the curvature of the curve. By default the curve is parameterized by arclength only. To resolve regions of high curvature choose the recommended values of `arcLengthWeight_=1.` and `curvatureWeight_=1..`

To determine the parameterization we equidistribute the weight function

$$w(r) = \text{arcLengthWeight} \frac{s(r)}{|s|_\infty} + \text{curvatureWeight} \frac{c(r)}{|c|_\infty}$$

where  $s(r)$  is the local arclength and  $c(r)$  is the curvature. Note that we normalize  $s$  and  $c$  by their maximum values.

$$c = |x_{ss}| = \frac{|x_{rr}|}{|x_r|^2}$$

**arcLengthWeight\_ (input):** A weight for arclength. A negative value may give undefined results.

**curvatureWeight\_ (input):** A weight for curvature. A negative value may give undefined results.

**numberOfSmooths (input):** Number of times to smooth the equidistribution weight function.

## 30 RestrictionMapping: define a restriction to a sub-rectangle of the unit cube

### 30.1 Description

The `RestrictionMapping` is a simple mapping that defines a restriction to a sub-rectangle of the unit square or unit cube. This Mapping is used by the `ReparameterizationTransform` where it can be used to define a mesh refinement patch on an adaptive grid.

The restriction is a Mapping from parameter space `r` to parameter space `x` defined by

$$\begin{aligned}x(I, \text{axis1}) &= (rb - ra)r(I, \text{axis1}) + ra \\x(I, \text{axis2}) &= (sb - sa)r(I, \text{axis2}) + sa \\x(I, \text{axis3}) &= (tb - ta)r(I, \text{axis3}) + ta\end{aligned}$$

### 30.2 Default Constructor

```
RestrictionMapping(const real ra_=0.,
                   const real rb_=1.,
                   const real sa_=0.,
                   const real sb_=1.,
                   const real ta_=0.,
                   const real tb_=1.,
                   const int dimension =2 ,
                   Mapping *restrictedMapping =NULL)
```

**Purpose:** Default Constructor The restriction is a Mapping from parameter space to parameter space defined by

$$\begin{aligned}x(I, \text{axis1}) &= (rb - ra)r(I, \text{axis1}) + ra \\x(I, \text{axis2}) &= (sb - sa)r(I, \text{axis2}) + sa \\x(I, \text{axis3}) &= (tb - ta)r(I, \text{axis3}) + ta\end{aligned}$$

**ra\_,rb\_,sa\_,sb\_,ta\_,tb\_ (input):** Parameters in the definition of the `RestrictionMapping`.

**dimension (input):** define the domain and range dimension (which are equal).

**restrictedMapping (input) :** optionally pass the Mapping being restricted. This Mapping is used to set `spaceIsPeriodic`.

### 30.3 scaleBounds

**int**

```
scaleBounds(const real ra_=0.,
            const real rb_=1.,
            const real sa_=0.,
            const real sb_=1.,
            const real ta_=0.,
            const real tb_=1.)
```

**Purpose:** Scale the current bounds. Define a sub-rectangle of the current restriction. These parameters apply to the current restriction as if it were the entire unit square or unit cube. For example for the

"r" variable the transformation from old values of (ra,rb) to new values of (ra,rb) is defined by:

$$\begin{aligned} rba &= rb - ra \\ rb &= ra + rb\_rba \\ ra &= ra + ra\_rba \end{aligned}$$

**ra\_,rb\_,sa\_,sb\_,ta\_,tb\_ (input):** These parameters define a sub-rectangle of the current restriction.

### 30.4 getBounds

```
int  
getBounds(real & ra_, real & rb_, real & sa_, real & sb_, real & ta_, real & tb_) const
```

**Description:** Get the bounds for a restriction mapping. `RestrictionMapping` for further details.

**ra\_,rb\_,sa\_,sb\_,ta\_,tb\_ (output):**

### 30.5 setBounds

```
int  
setBounds(const real ra_=0.,  
          const real rb_=1.,  
          const real sa_=0.,  
          const real sb_=1.,  
          const real ta_=0.,  
          const real tb_=1.)
```

**Purpose:** Set absolute bounds for the restriction.

**ra\_,rb\_,sa\_,sb\_,ta\_,tb\_ (input):** Parameters in the definition of the `RestrictionMapping`.

### 30.6 setSpaceIsPeriodic

```
int  
setSpaceIsPeriodic( int axis, bool trueOrFalse = true)
```

**Description:** Indicate whether the space being restricted is periodic. For example if you restrict an `AnnulusMapping` then you should set `periodic1=true` since the `Annulus` is periodic along axis1

## 31 RevolutionMapping: create a surface or volume of revolution

### 31.1 Description

The `RevolutionMapping` revolves a two-dimensional mapping (ie. a 2D curve or 2D region) in the plane around a given line to create a three-dimension mapping in three-space. The `RevolutionMapping` can also be used to revolve a curve in 3D about a line to create a surface in 3D.

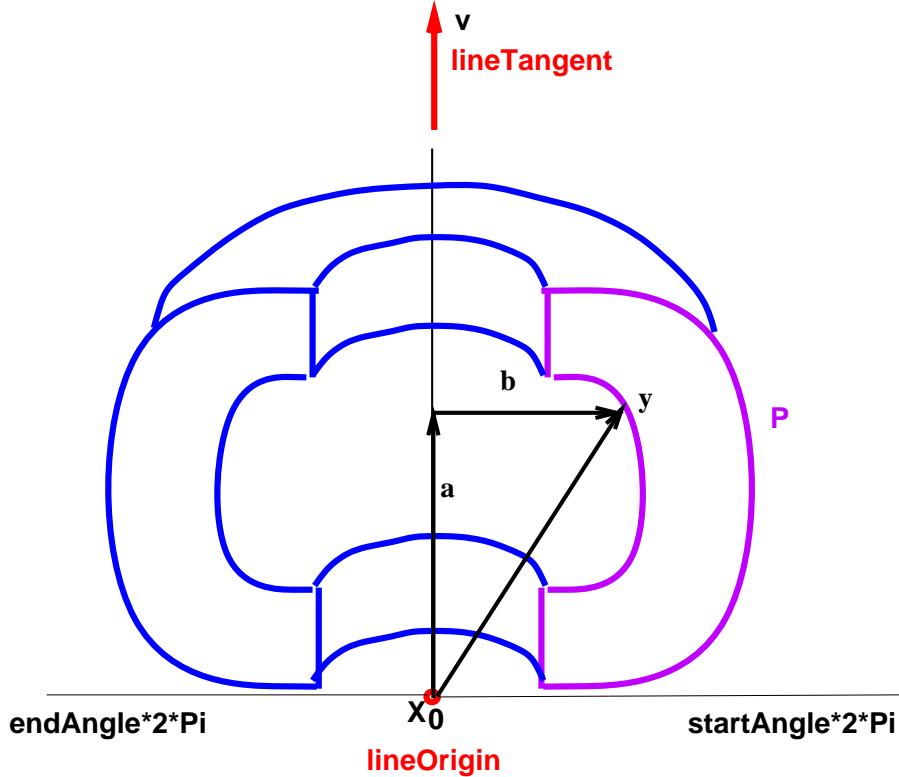


Figure 20: The `RevolutionMapping` can be used to revolve a 2D mapping about a line through a given angle. It can also be used to revolve a 3D curve.

The revolution mapping is defined in the following manner. (This description applies to the case when the mapping to be revolved is a 2D region. A similar definition applies in the other cases). Let

- $\mathbf{x}_0 = \text{lineOrigin}(0 : 2)$  = a point on the line of revolution
- $\mathbf{v} = \text{lineTangent}(0 : 2)$  = unit tangent to the line of revolution
- $\mathbf{P}$  : The two dimensional mapping in the plane that we will rotate

We first evaluate the two-dimensional mapping and save in a three dimension vector,  $\mathbf{y}$ ,

$$\mathbf{r}(0 : 1) \rightarrow (\mathbf{P}(\mathbf{r}), 0) = (y(0 : 1), 0) \equiv \mathbf{y}$$

Now decompose  $\mathbf{y} - \mathbf{x}_0$  into a component parallel and a component orthogonal to the line of revolution:

$$\begin{aligned}\mathbf{y} - \mathbf{x}_0 &= \mathbf{a} + \mathbf{b} \\ \mathbf{a} &= (\mathbf{a} \cdot \mathbf{v})\mathbf{v} \quad \text{component parallel to } \mathbf{v}\end{aligned}$$

Then rotate the part orthogonal to the line:

$$\mathbf{x} - \mathbf{x}_0 = \mathbf{a} + R\mathbf{b} \quad \text{where } R \text{ is the rotation matrix}$$

To compute  $R$  we determine a vector  $\mathbf{c}$  orthogonal to  $\mathbf{b}$  and  $\mathbf{v}$ ,

$$\mathbf{c} = \mathbf{v} \times \mathbf{b}$$

Then

$$R\mathbf{b} = \cos(\theta)\mathbf{b} + \sin(\theta)\mathbf{c}$$

where

$$\theta = r(2, I)\delta + \text{startAngle } 2\pi, \quad \delta = (\text{endAngle} - \text{startAngle})2\pi$$

In summary the revolution mapping is defined by

$$\begin{aligned}\mathbf{x} &= \mathbf{a} + \cos(\theta)\mathbf{b} + \sin(\theta)\mathbf{c} + \mathbf{x}_0 \\ \mathbf{a} &= ((\mathbf{y} - \mathbf{x}_0) \cdot \mathbf{v})\mathbf{v} \\ \mathbf{b} &= \mathbf{y} - \mathbf{x}_0 - \mathbf{a} \\ \mathbf{c} &= \mathbf{v} \times \mathbf{b}\end{aligned}$$

The derivatives of the mapping are defined as

$$\begin{aligned}\frac{\partial \mathbf{x}}{\partial r_i} &= \frac{\partial \mathbf{a}}{\partial r_i} + \cos(\theta)\frac{\partial \mathbf{b}}{\partial r_i} + \sin(\theta)\frac{\partial \mathbf{c}}{\partial r_i} \quad \text{for } i = 0, 1 \\ \frac{\partial \mathbf{a}}{\partial r_i} &= \left(\frac{\partial \mathbf{y}}{\partial r_i} \cdot \mathbf{v}\right)\mathbf{v} \\ \frac{\partial \mathbf{b}}{\partial r_i} &= \frac{\partial \mathbf{y}}{\partial r_i} - \frac{\partial \mathbf{a}}{\partial r_i} \\ \frac{\partial \mathbf{c}}{\partial r_i} &= \mathbf{v} \times \frac{\partial \mathbf{b}}{\partial r_i} \\ \frac{\partial \mathbf{x}}{\partial r_2} &= \delta * (-\sin(\theta)\mathbf{b} + \cos(\theta)\mathbf{c})\end{aligned}$$

### 31.2 Inverse of the mapping

When the mapping to be revolved is in the  $x$ - $y$  plane, the `RevolutionMapping` can be inverted analytically in terms of the inverse of the mapping that is being revolved. (When revolving a 3D curve we do not define a special inverse). Here is how we do the inversion.

Given a value  $\mathbf{x}$  we need to determine the corresponding value of  $\mathbf{r}$ . To do this we can first decide how to rotate the point  $\mathbf{x}$  (about the line through  $\mathbf{x}_0$  with tangent  $\mathbf{v}$ ) into the  $x$ - $y$  plane. This will determine  $\theta$  and  $\mathbf{y}$ . Given  $\mathbf{y}$  we can invert the two-dimensional mapping to determine  $(r_0, r_1)$ .

To perform the rotation back to the  $x - y$  plane we decompose  $\mathbf{x} - \mathbf{x}_0$  into

$$\mathbf{x} - \mathbf{x}_0 = \mathbf{a} + \hat{\mathbf{b}}$$

where  $\mathbf{a}$  is the component parallel to  $\mathbf{v}$ , (the same  $\mathbf{a}$  as above),

$$\mathbf{a} = ((\mathbf{x} - \mathbf{x}_0) \cdot \mathbf{v})\mathbf{v}$$

and

$$\hat{\mathbf{b}} = \mathbf{x} - \mathbf{x}_0 - \mathbf{a}.$$

Note that  $\hat{\mathbf{b}}$  is not the same as  $\mathbf{b}$ . Letting

$$\hat{\mathbf{c}} = \pm \mathbf{v} \times \hat{\mathbf{b}}$$

where the correct sign must be chosen, then we can rotate back to the  $x - y$  plane with the transformation

$$\mathbf{y} = \mathbf{a} + \cos(-\theta)\hat{\mathbf{b}} + \sin(-\theta)\hat{\mathbf{c}} \tag{17}$$

Since  $y_3 = 0$  ( $\mathbf{y} = (y_1, y_2, y_3)$ ) it follows from the third component of this last equation that

$$0 = a_3 + \cos(\theta)\hat{b}_3 - \sin(\theta)\hat{c}_3$$

Now assuming that  $a_3 = 0$  (which assumes that the line of rotation is in the  $x - y$  plane) then

$$(\theta) = \frac{\hat{b}_3}{\hat{c}_3}$$

Given  $\theta$  we then know the first two components of  $\mathbf{y} = (y_1, y_2, 0)$  from (17). We now determine the inverse of  $(y_1, y_2)$  using the `inverseMap` of the two-dimensional mapping,

$$\mathbf{r}(0 : 1) = \mathbf{P}^{-1}(\mathbf{y}(0 : 1))$$

### 31.3 Reparameterized to spherical-like coordinates

If the body of revolution created by this mapping has a spherical polar singularity at one or both ends we may wish to create a new mapping near the pole that does not have a singularity by using an orthographic mapping (created from the `reparameterize` menu in `ogen`). The orthographic transform expects the mapping to be parameterized like a sphere with parameters  $(\phi, \theta, r)$ . Thus we will want to change the order of the parameters in the above definition of the body of revolution:

$$\tilde{\mathbf{x}}(r_1, r_2, r_3) = \mathbf{x}(r_1, r_3, r_2) \quad \text{or} \quad \tilde{\mathbf{x}}(r_1, r_2, r_3) = \mathbf{x}(r_3, r_1, r_2)$$

so that the new variable  $\tilde{\mathbf{x}}$  will be parameterized like a sphere.

This re-ordering is done automatically if the body of revolution is detected to have a spherical polar type singularity.

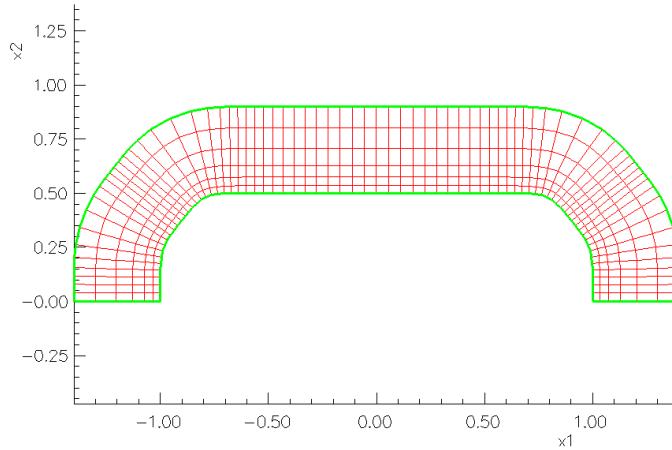
### 31.4 Examples

smoothedPolygon

```

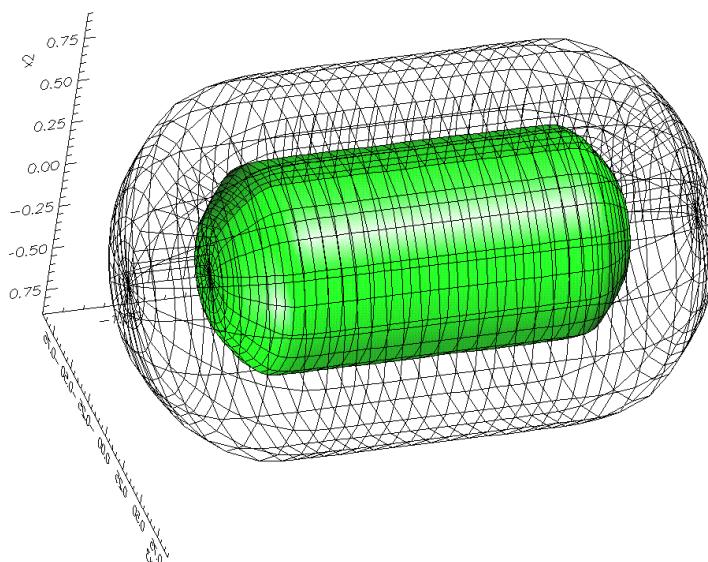
1  *
2  * Create a cylindrical body of revolution
3  * from a Smoothed Polygon
4  *
5  SmoothedPolygon
6    vertices
7    7
8    -1. 0.
9    -1. .25
10   -.8 .5
11   0. .5
12   .8 .5
13   1. .25
14   1. 0.
15   n-dist
16   fixed normal distance
17   .1
18   n-dist
19   fixed normal distance
20   .4
21   corners
22   specify positions of corners
23   -1. 0.
24   1. 0
25   -1.4 0.
26   1.4 0
27   t-stretch
28   0 5
29   .15 10
30   .15 10
31   0 10
32   .15 10
33   .15 10
34   0 10
35   exit
36   body of revolution
37   tangent of line to revolve about
38   1. 0 0
39   boundary conditions
40   0 0 -1 -1 1 0
41   mappingName
42   cylinder
43   lines
44   45 21 7
45   * exit

```



1

A two-dimensional smoothed polygon.



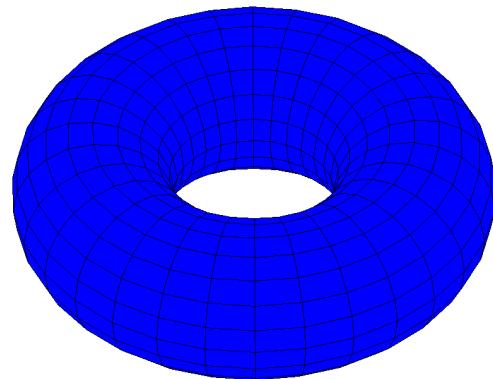
0

A body of revolution created by revolving a two-dimensional smoothed polygon.

```

1   Circle or ellipse
2     exit
3   body of revolution
4     choose a point on the line to revolve about
5       -2. 0 0
6
7

```



A body of revolution for a torus is created by revolving a circle.

### 31.5 Constructor

**RevolutionMapping()**

**Purpose:** Default Constructor

### 31.6 Constructor

```

RevolutionMapping(Mapping & revolutionary_,
                  const real startAngle_ =0.,
                  const real endAngle_ =1.,
const RealArray & lineOrigin_ =Overture::nullRealDistributedArray(),
const RealArray & lineTangent_ =Overture::nullRealDistributedArray()
)

```

**Purpose:** This constructor takes a mapping to revolve plus option parameters

**revolutionary\_ (input)** : mapping to revolve.

**startAngle\_ (input)** : starting "angle" (in [0,1]) for the revolution.

**endAngle\_ (input)** : ending "angle" (in [0,1]) for the revolution.

**lineOrigin\_ (input)** : the point of origin for the line of revolution.

**lineTangent\_ (input)** : the tangent to the line of revolution.

### 31.7 setRevolutionAngle

```

int
setRevolutionAngle(const real startAngle_ =0.,
                  const real endAngle_ =1.)

```

**Purpose:** Define the angle through which the revolution progresses.

**startAngle\_** (**input**) : starting "angle" (in [0,1]) for the revolution.

**endAngle\_** (**input**) : ending "angle" (in [0,1]) for the revolution.

### 31.8 getRevolutionAngle

**int**

**getRevolutionAngle( real & startAngle\_,  
real & endAngle\_ )**

**Purpose:** Get the bounding angles in the revolution progresses.

**startAngle\_** (**input**) : starting "angle" for the revolution.

**endAngle\_** (**input**) : ending "angle" for the revolution.

### 31.9 setParameterAxes

**int**

**setParameterAxes( const int & revAxis1\_, const int & revAxis2\_, const int & revAxis3\_ )**

**Purpose:** Define the parameter axes the mapping. The 2D mapping will be evaluated with  $(r(I, \text{revAxis1}), r(I, \text{revAxis2}))$  while  $r(I, \text{revAxis3})$  will correspond to the angle of revolution  $\theta$ . The choice of these variables is normally only important if the body of revolution has a spherical polar singularity at one or both ends and the user wants to remove the singularity using the orthographic projection.(reparameterization option). The orthographic project expects the mapping to parameterized like a sphere with the parameters in the order  $(\phi, \theta, r)$ .

**revAxis1** The axis corresponding to  $\phi$  in a spherical coordinate systems or the axial variable  $s$  in cylindrical coordinates. **revAxis1** will normally be 0 (or 1) and correspond to the axial like variable in the 2D mapping that is being revolved.

**revAxis2** The axis corresponding to  $r$  in a spherical coordinate system. Normally **revAxis2**=2 so the axial variable appears last.

**revAxis3** The axis corresponding to  $\theta$  in a spherical coordinate system. Normally **revAxis3**=1.

**revAxis1\_,revAxis2\_,revAxis3\_** (**input**) : A permutation of (0,1,2).

### 31.10 setRevolutionary

**int**

**setRevolutionary(Mapping & revolutionary\_)**

**Purpose:** Define the mapping that will be revolved.

**revolutionary\_input**) : mapping to revolve.

### 31.11 setLineOfRevolution

**int**

**setLineOfRevolution(const RealArray & lineOrigin\_,  
const RealArray & lineTangent\_ )**

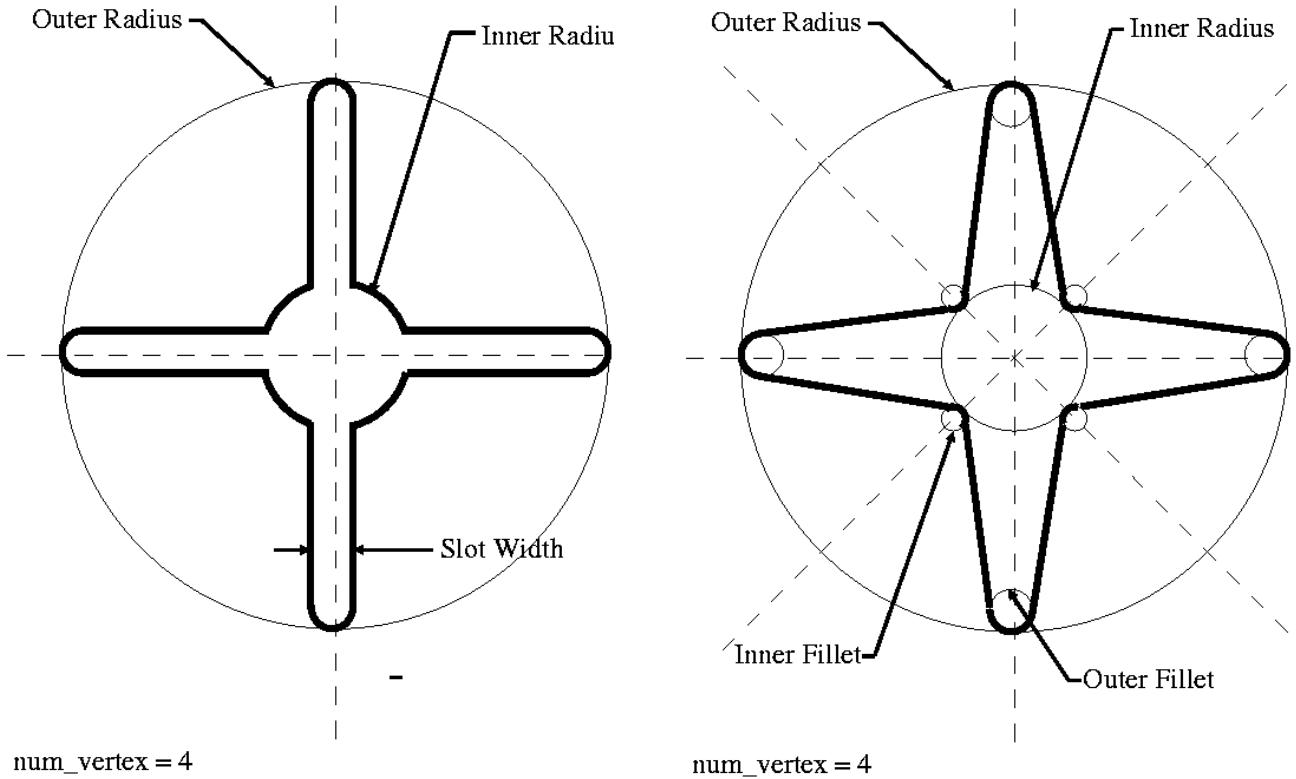
**Purpose:** Define the point of origin and the tangent of the line of revolution. \*For now this point and line must lie in the x-y plane (lineOrigin\_(2)==0, lineTangent\_(2)==0)

**lineOrigin\_ (input)** : the point of origin for the line of revolution. For now we require  $\text{lineOrigin}_\text{(2)} == 0$ .

**lineTangent\_ (input)** : the tangent to the line of revolution with  $\text{lineTangent}_\text{(2)} == 0$

## 32 RocketMapping: create rocket geometry curves

The `RocketMapping` defines a variety of curves related to rockets. The curves defined in this class were originally written by Nathan Crane (as three separate Mapping's), and then subsequently reorganized into a single class by WDH. There are currently 3 cross-section shapes supported, the **slot**, **star** and **circular** shapes. The slot and star shapes are illustrated in the next figure.



The slot shape.

The star cross-section.

### 32.1 Slot

**Overview :** The slot option creates by default a slotted grain shaped spline in the  $z=0$  plane. The mapping should be usable in every way as a standard spline. The slot spline mapping will always be periodic, the number and location of the spline knot points are generated automatically according to the slotted grain input parameters. A graphical description of the various parameters can found in the above figure.

Options:

**set range dimension** : Toggle between a 2D and 3D spline (spline will always lie in a plane, but in 3D that plane can be rotated or shifted to a arbitrary positon.)

**shape preserving (toggle)** : toggle between shape preserving and tension spline (see standard spline mapping documentation for more info)

**set bounding radii** : Set the inner and outer bounding radii for the slot grain. slots will just touch a circle of radius outer bounding radius. The slots will intersect a circle of radius inner bounding radius.

**set slot width** : Set the width of each slot. The slots wil be rounded on the ends by a circle of diameter slot width.

**set z value** : by default the spline lies in the z=0 plane. Changing the z value moves the spline to some other constant z value plane. This command is a shortcut for the shift operator. This command cannot be used with a 2D slotted grain mapping.

**set element size** : Set the size of the elements along the spline. The total length of the spline is computed, and the number of lines of grid points is taken as total.length/el.size. the number of spline knot points is taken as the same as the number of lines. Element size and number of lines are mutually exclusive commands.

**set number of vertex** : Set the number of vertices (number of slots) of the mapping valid values are 2 vertices and up.

## 32.2 Star

**Overview:** The star option creates by default a star shaped spline in the z=0 plane. The mapping should be usable in every way as a standard spline. The star spline mapping will always be periodic, the the number and location of the spline knot points are generated automatically according to the star input parameters. A graphical description of the various parameters can found in the above figure.

Options:

**set range dimension** : Toggle between a 2D and 3D spline (spline will always lie in a plane, but in 3D that plane can be rotated or shifted to a arbitrary positon.)

**shape preserving (toggle)** : toggle between shape preserving and tension spline (see standard spline mapping documentation for more info)

**set bounding radii** : set the inner and outer bounding radii for the star. The inside points the star will be circumscribed between the inner and outer radii. The outer points of the star will just touch a circle of radius outer bounding radius. The inner points of the star will just touch a circle of radius inner bounding radius.

**set fillet radii** : set the inner and outer fillet radii for the star. The fillet radi determine the sharpness of the points of the star. Large fillet radii create a more blun star, while smaller fillet radi create a sharper star. Note that when createing in 3D volume, a sharper pointed star will require a thiner boundry mesh, and thus more elements to mesh than a blunted star.

**set z value** : by default the star lies in the z=0 plane. Changing the z value moves the star to some other constant z value plane. This command is a shortcut for the shift operator. This command cannot be used with a 2D star.

**set element size** : Set the size of the elements along the star. The total length of the star spline is computed, and the number of lines of grid points is taken as total.length/el.size. the number of spline knot points is taken as the same as the number of lines. Element size and number of lines are mutually exclusive commands.

**set number of vertex** : Set the number of vertices (number of arms) of the star valid values are 2 vertices and up. For instance a space shuttle fuel grain is described by a 11 vertex star.

**set number of points** : explicitly sets the number of knot points for the spline. This command overrides the set element size command.

### 32.3 circle

**Overview :** The CircSplineMapping creates by default a circular spline in the z=0 plane. The mapping should be usable in every way as a standard spline. The circular spline mapping will always be periodic, the number and location of the spline knot points are generated automatically according to the circle parameters. The circualr spline mapping is need when creating rocket cross sections to correctly parameterize the star, anular, and sloted grain portions in a compatible way.

### 32.4 Member functions

#### 32.4.1 Constructor

**RocketMapping(const int & rangeDimension\_ =2)**

**Purpose:** Define various cross-sections related to rocket geometries

**rangeDimension\_ :** 2, 3

**Author:** Nathan Crane, cleaned up by Bill Henshaw.

#### 32.4.2 computePoints

```
int  
computePoints()
```

**Purpose:** Compute the spline points.

#### 32.4.3 computeSlotPoints

```
int  
computeSlotPoints()
```

**Purpose:** Compute the spline points for a slotted cross-section.

#### 32.4.4 computeStarPoints

```
int  
computeStarPoints()
```

**Purpose:** Supply spline points for a star cross-section.

#### 32.4.5 computeCirclePoints

```
int  
computeCirclePoints()
```

**Purpose:** Supply spline points for a 2D Circ.

#### 32.4.6 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively create and/or change the spline mapping.

**mapInfo (input):** Holds a graphics interface to use.

### 33 SmoothedPolygon

This mapping can be used to create a grid whose one edge is a polygon with smoothed corners. The grid is created by extending normals from the smoothed polygon.

The smoothed polygon is defined by a sequence of vertices

$$\mathbf{x}_v(i) = (x_v(i), y_v(i))), \quad i = 0, 1, \dots, n_v - 1$$

The curve is parameterized by a pseudo-arclength  $s$ ,  $0 \leq s \leq 1$  with the value of  $s$  at vertex  $i$  defined by the relative distance along the (un-smoothed) polygon:

$$s(i) = \frac{\sum_{j=0}^{i-2} \|\mathbf{x}_v(j+1) - \mathbf{x}_v(j)\|}{\sum_{j=0}^{n_v-2} \|\mathbf{x}_v(j+1) - \mathbf{x}_v(j)\|}$$

The smoothed polygon is defined using the *interval* functions  $V_j(s)$  of the StretchMapping class. The interval functions can be used to smoothly transition from one slope to a second slope. For example, the x component of the smoothed polygon is defined as

$$x(s) = \left[ s + \sum_{j=0}^{n_v} (V_j(s) - V_j(0)) \right] c_1 + c_0$$

Recall that the interval function  $V_j$  is dependent on the three parameters  $d_j$ ,  $e_j$  and  $f_j$ . The parameter  $d_j$  for  $V_j$  is given by

$$d_j = \frac{x_v(j+1) - x_v(j)}{s(j+1) - s(j)} \quad j = 0, 1, \dots, n_v - 2$$

while

$$f_j = s(j)$$

The value of  $e_j$  is specified by the user (default = 40) and determines the sharpness of the curve at the vertex.

A grid is defined from this smooth polygon by extending normals. The length of the normal can be constant or can be made to vary. If  $r_1$  parameterizes the curve in the tangential direction and  $r_2$  in the normal direction then the parameterization of the grid is given by

$$\mathbf{x}(r_1, r_2) = \mathbf{x}(r_1) + r_2 N(r_1) \mathbf{n}(r_1)$$

The function  $N(r_1)$  is itself defined in terms of stretching functions.

The user has the option to stretch the grid lines in the tangential direction in order to concentrate grid lines near the vertices. The user may also stretch the grid lines in the normal direction. Of course the grid lines may also be stretched by composing this mapping with a StretchMapping.

**Note:** Unfortunately the smoothed polygon only matches the corners exponentially close with respect to the **sharpness** parameter. Moreover the higher numbered vertices will have larger errors (cf. the formula above). If you choose small values for the sharpness then the SmoothedPolygon will not match the vertices very well, nor will it be symmetric.

### 33.1 update(MappingInformation &)

The SmoothPolygon Mapping is defined interactively through a graphics interface:

```
GL_GraphicsInterface graphicsInterface;           // create a GL_GraphicsInterface object
MappingInformation mappingInfo;
mappingInfo.graphXInterface=&graphicsInterface;

...
SmoothPolygon poly;
poly.interactiveConstructor( mappingInfo );        // interactively create the smoothed polygon
```

The user must specify the vertices of the polygon. The user may then optionally change various parameters from their default values.

- **sharpness** : Specify how sharp the corners are (exponent). Choose the value for  $e_j$  in  $V_j$ . Note that if you choose small values for the sharpness then the SmoothedPolygon will not match the vertices very well, nor will it necessarily be symmetric.
- **t-stretch** : Specify stretching in tangent-direction. Specify the values for  $a_j$  and  $b_j$  for the exponential layer stretching at corner  $j$
- **n-stretch** : Specify stretching in normal-direction. Specify the values for  $a_i$ ,  $b_i$  and  $c_i$  for the exponential layer function for stretching in the normal direction and specify the number of layer functions. By default there is one layer function and the grid lines are concentrated near the smoothed polygon with values  $a_0 = 1.$ ,  $b_0 = 4.$  and  $c_0 = 0.$
- **corners** : Fix the grid corners to specific positions. Use this option to fix the positions of the four corners of the grid. The corners of the grid that lie at a normal distance from the smoothed polygon may not be exactly where you want them because the normal may be slightly different from the line which is perpendicular to the straight line which joins the vertices. This option applies a bi-linear transformation to the entire grid in order to deform the corners to the specified positions.
- **n-dist** : Specify normal distance at vertex(+-epsilon) Choose the normal distance for the grid to extend from the polygon. Optionally the normal distance can be made to vary; a separate normal distance can be given at the position just before vertex  $i$  and just after vertex  $i$ .
- **curve or area (toggle)** : Change the mapping from defining an area to define a curve (or vice versa). In other words toggle the domain dimension between 1 and 2.
- **isPeriodic**: Specify periodicity array. Indicate whether the grid periodic in the tangential direction. Set this value to 2 if the grid is closed and periodic or to 1 if the grid is not closed but the derivative of the curve is periodic.
- **help** : Print this list
- **exit** : Finished with parameters, construct grid

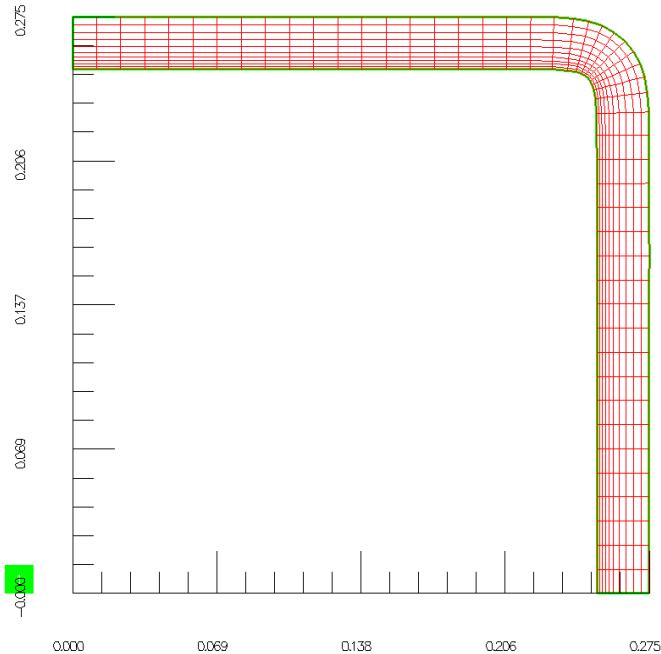
### 33.2 Examples

Here are some sample command files that create some SmoothedPolygon mappings. These command files can be read, for example, by the overlapping grid generator `ogen` from within the `create mappings` menu.

```

1  *
2  * SmoothedPolygon: example 1
3  *
4  SmoothedPolygon
5    vertices
6      * number of vertices:
7      3
8      * vertices:
9      .25 0.
10     .25 .25
11     0. .25
12   lines
13     51 11
14

```

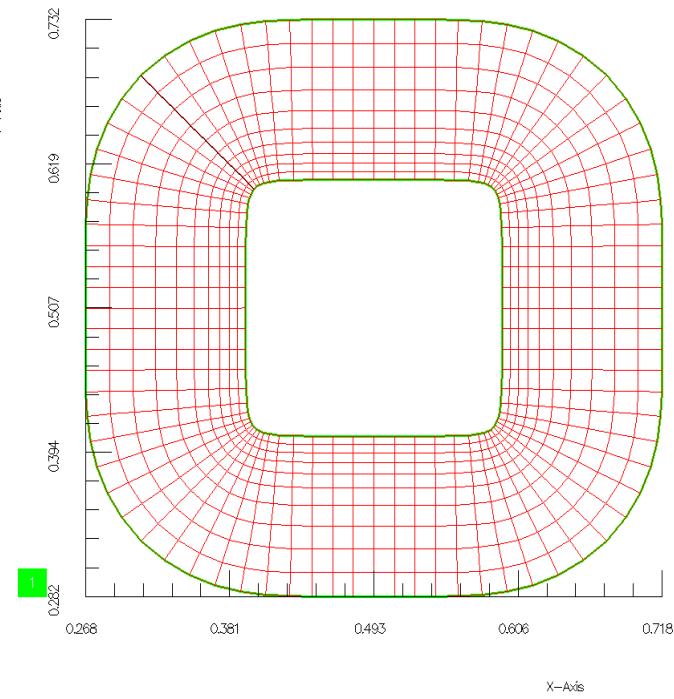


SmoothedPolygon example 1

```

1  *
2  * SmoothedPolygon: example 2
3  * Note: If first vertex is the same as the last,
4  * then the mapping is assumed to be periodic
5  *
6  *
7  SmoothedPolygon
8    vertices
9      * number of vertices:
10     5
11      * vertices:
12     .4 .6
13     .6 .6
14     .6 .4
15     .4 .4
16     .4 .6
17      * specify normal distance
18    n-dist
19      fixed normal distance
20      .125
21    lines
22      81 11
23

```

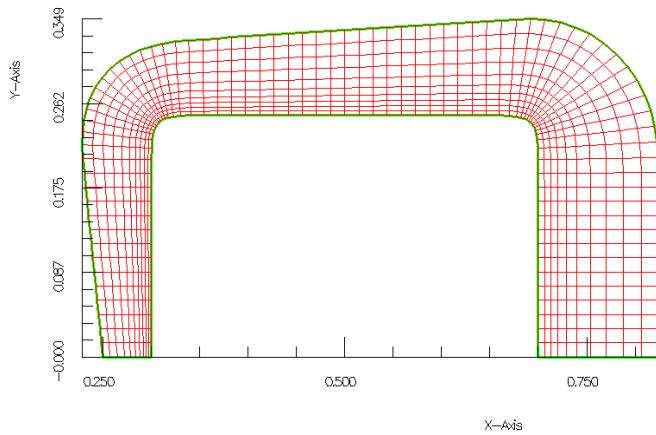


SmoothedPolygon example 2

```

1  *
2  * SmoothedPolygon: example 3
3  *
4  SmoothedPolygon
5    vertices
6      * number of vertices:
7      4
8      * vertices:
9      .3  .0
10     .3  .25
11     .7  .25
12     .7  0.
13   * specify normal distance
14   n-dist
15     variable normal distance
16     .05  .075  40.
17     .075  .1   40.
18     .125 .125  40.
19   lines
20     81  11

```



SmoothedPolygon example 3

## 34 SphereMapping

This mapping defines a spherical shell or spherical surface in three-dimensions,

$$\begin{aligned}\phi &= \pi(\phi_0 + r_1(\phi_1 - \phi_0)) \\ \theta &= 2\pi(\theta_0 + r_2(\theta_1 - \theta_0)) \\ R(r_3) &= (R_0 + r_3(R_1 - R_0)) \\ \mathbf{x}(r_1, r_2, r_3) &= (R \cos(\theta) \sin(\phi) + x_0, R \sin(\theta) \sin(\phi) + y_0, R \cos(\phi) + z_0)\end{aligned}$$

This mapping can be inverted analytically with the inverse defined by

$$\begin{aligned}r &:= \sqrt{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2} \\ r_1 &= [\cos^{-1}((z - z_0)/r) - \pi\phi_0]/(\pi(\phi_1 - \phi_0)) \\ r_2 &= [\text{atan2}(y_0 - y, x_0 - x) + \pi - 2\pi\theta_0]/(2\pi(\theta_1 - \theta_0)) \\ r_3 &= (r - R_0)/(R_1 - R_0)\end{aligned}$$

This mapping can have a spherical polar singularity at one or both ends. Either singularity can be removed by creating an orthographic patch over the pole using the **Reparameterization** transform. In order to do this we must be able to evaluate the derivatives of the **SphereMapping** and its inverse in spherical coordinates. This means we compute the derivatives of the mapping as

$$\left( \frac{\partial x_i}{\partial r_1}, \frac{1}{\sin(\phi)} \frac{\partial x_i}{\partial r_2}, \frac{\partial x_i}{\partial r_3} \right)$$

and the derivatives of the inverse mapping as

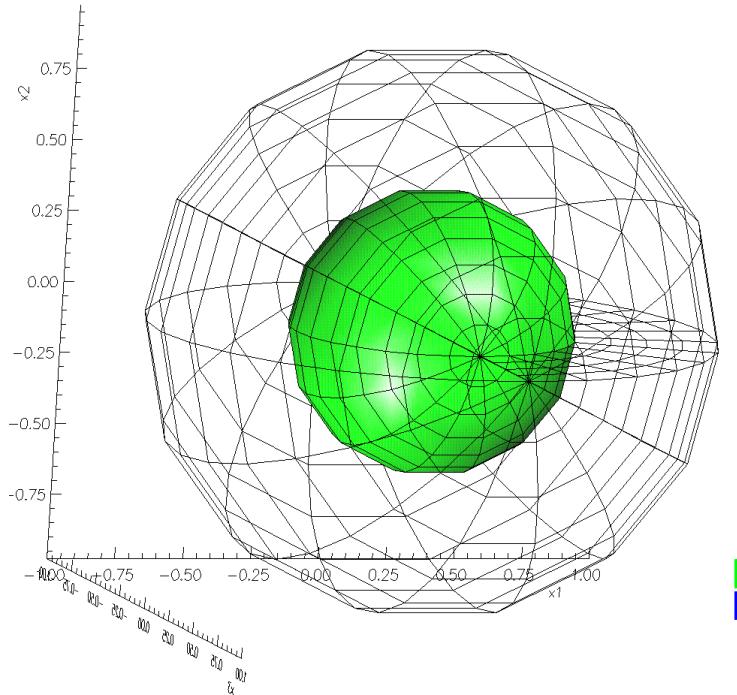
$$\left( \frac{\partial r_1}{\partial x_i}, \sin(\phi) \frac{\partial r_2}{\partial x_i}, \frac{\partial r_3}{\partial x_i} \right).$$

### 34.1 Examples

```

1  *
2  * Make a sphere
3  *
4  Sphere
5   mappingName
6   sphere
7   inner and outer radii
8   1. 1.5
9   bounds on phi (latitude)
10 .0001 .9999
11 surface or volume (toggle)
12 lines
13   101 101
14 pause
15 exit
16 unstructured
17   build from a mapping
18   sphere
19   exit
20 get geometric properties
21 unstructuredMapping

```

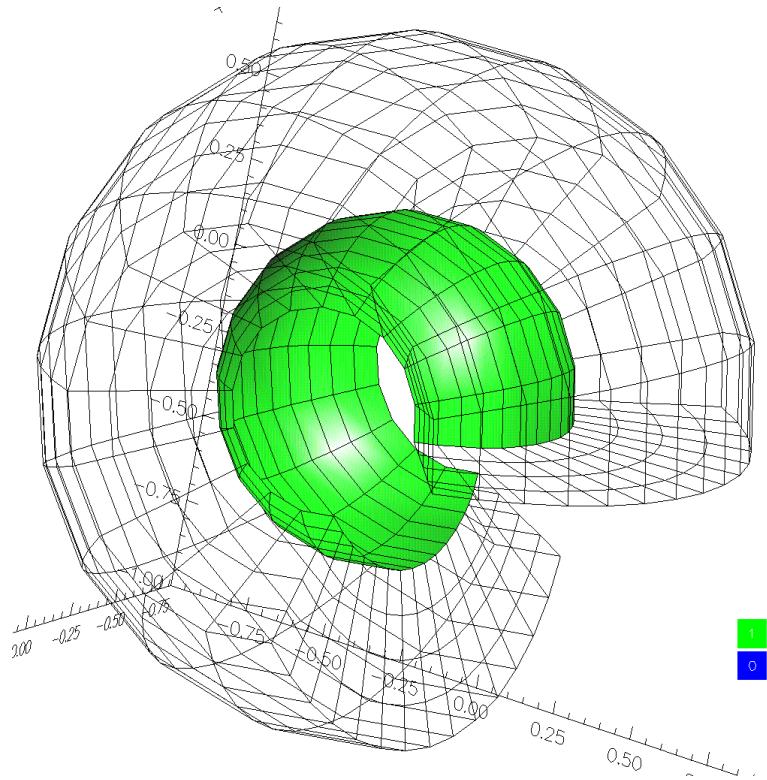


A spherical shell built with the **SphereMapping**.

```

1 Sphere
2   bounds on phi (latitude)
3     .2 .8
4   bounds on theta (longitude)
5     0. .8
6   exit
7

```



A partial spherical shell built with the `SphereMapping`.

### 34.2 Constructor

```

SphereMapping(const real & innerRadius_ .5,
              const real & outerRadius_ 1.,
              const real & x0_ .0,
              const real & y0_ .0,
              const real & z0_ .0,
              const real & startTheta_ .0,
              const real & endTheta_ 1.,
              const real & startPhi_ .0,
              const real & endPhi_ 1.)

```

**Description:** Define a spherical shell or spherical surface.

**innerRadius\_,outerRadius\_ (input):** bounds on the radius.

**x0\_,y0\_,z0\_ (input) :** center.

**startTheta\_,endTheta\_ (input) :** bounds on normalized  $\theta$ , in the range [0, 1].

**startPhi\_,endPhi\_ (input):** bounds on the normalized  $\phi$ , in the range [0, 1].

### 34.3 setOrigin

```

int
setOrigin(const real & x0_ =.0,
          const real & y0_ =.0,
          const real & z0_ =.0)

```

**Description:** Specify parameters for the sphere.

**x0\_-,y0\_-,z0\_- (input)** : center.

#### 34.4 setPhi

```
int  
setPhi(const real & startPhi_ =.0,  
       const real & endPhi_ =1.)
```

**Description:** Specify parameters for the sphere.

**startPhi\_,endPhi\_- (input):** bounds on the normalized  $\phi$ , in the range [0, 1].

#### 34.5 setRadii

```
int  
setRadii(const real & innerRadius_ =.5,  
         const real & outerRadius_ =1.)
```

**Description:** Specify parameters for the sphere.

**innerRadius\_,outerRadius\_- (input):** bounds on the radius.

#### 34.6 setTheta

```
int  
setTheta( const real & startTheta_ =.0,  
          const real & endTheta_ =1.)
```

**Description:** Specify parameters for the sphere.

**startTheta\_,endTheta\_- (input) :** bounds on normalized  $\theta$ , in the range [0, 1].

## 35 SplineMapping: create a spline curve

Define a cubic spline curve in 1, 2, or 3 space dimensions. The spline curve is chosen to pass through a set of user defined points. Options include

**tension** : create a spline under tension to remove wiggles, specify a constant tension.

**shape preservation** : automatic determination of tension factors that vary along the spline so as to create a shape preserving (“monotone”) spline.

**end conditions** : A variety of end conditions for the spline are available:

**periodic** : The spline can be periodic (choose the periodicity option ‘function periodic’).

**derivative periodic** : The derivative of the spline can be periodic (choose the periodicity option ‘derivative periodic’).

**monotone parabolic fit** : default BC for the shape preserving spline.

**first derivative** : user specified first derivatives.

**second derivative** : user specified second derivatives.

**parameterize** : by arclength or by weighting the arclength and curvature in order to concentrate grid points near regions with large curvature.

A 2D or 3D spline is parameterized by arclength. A 1D spline is parameterized by the index value of the point. For a spline which is periodic in space, the Mapping will automatically add an extra point if the first point is not equal to the last point.

The SplineMapping uses “**TSPACK**: Tension Spline Curve Fitting Package” by Robert J. Renka; available from Netlib. See the TSPACK documentation and the reference

**RENKA, R.J.** Interpolatory tension splines with automatic selection of tension factors. SIAM J. Sci. Stat. Comput. **8**, (1987), pp. 393-415.

### 35.1 Member functions

#### 35.1.1 Constructor

**SplineMapping(const int & rangeDimension\_ =2)**

**Purpose:** Default Constructor: create a spline curve with the given range dimension. Use this Mapping to create a cubic spline curve in two dimensions. This spline is defined by a set of points (knots),  $x(i), y(i)$ . The spline is normally parameterized by arclength. The pline can also be parameterized by a weighting of arclength and curvature so that more points are placed in regions with high curvature. For a spline which is periodic in space, the Mapping will automatically add an extra point if the first point is not equal to the last point.

**rangeDimension\_ :** 1,2, 3

The SplineMapping uses ‘**TSPACK**: Tension Spline Curve Fitting Package’ by Robert J. Renka; available from Netlib. See the TSPACK documentation and the reference

**RENKA, R.J.** Interpolatory tension splines with automatic selection of tension factors. SIAM J. Sci. Stat. Comput. **8**, (1987), pp. 393-415.

## 35.2 shift

```
int  
shift(const real & shiftx =0.,  
      const real & shifty =0.,  
      const real & shiftz /* =0.*/ )
```

**Purpose:** Shift the SPLINE in space.

## 35.3 scale

```
int  
scale(const real & scalex =0.,  
      const real & scaley =0.,  
      const real & scalez /* =0.*/ )
```

**Purpose:** Scale the SPLINE in space.

## 35.4 rotate

```
int  
rotate( const int & axis, const real & theta )
```

**Purpose:** Perform a rotation about a given axis. This rotation is applied after any existing transformations. Use the reset function first if you want to remove any existing transformations.

**axis (input) :** axis to rotate about (0,1,2)

**theta (input) :** angle in radians to rotate by.

### 35.4.1 setParameterizationType

```
int  
setParameterizationType(const ParameterizationType & type)
```

**Description:** Specify the parameterization for the Spline. With **index** parameterization the knots on the spline are parameterized as being equally spaced. With **arcLength** parameterization the knots are parameterized by arcLength or a weighted combination of arcLength and curvature. With **userDefined** parameterization the user must supply the parameterization through the **setParameterization** function.

**type (input) :** One of **index** or **arcLength** or **userDefined**.

### 35.4.2 getParameterization

```
const RealArray &  
getParameterizationS() const
```

**Description:** Return the current parameterization.

### 35.4.3 getNumberOfKnots

```
int  
getNumberOfKnots() const
```

**Purpose:** Return the number of knots on the spline.

#### 35.4.4 setParameterization

```
int  
setParameterization(const RealArray & s_ )
```

**Description:** Supply a user defined parameterization. This routine will set the parameterization type to be `userDefined`.

**s\_ (input) :** An increasing sequence of values that are to be used to parameterize the spline points. These values must cover the interval [0,1] which will be the interval defining the mapping. You could add values outside [0,1] to define the behaviour of the spline at "ghost points". The number of points in the array must be equal to the number of points supplied when the `setPoints` function is called.

#### 35.4.5 parameterize

```
int  
parameterize(const real & arcLengthWeight_ /* =1.*/,  
             const real & curvatureWeight_ /* =0.*/ )
```

**Description:** Set the 'arclength' parameterization parameters. The parameterization is chosen to redistribute the points to resolve the arclength and/or the curvature of the curve. By default the spline is parameterized by arclength only. To resolve regions of high curvature choose the recommended values of `arcLengthWeight_=1.` and `curvatureWeight_=0.5`.

To determine the parameterization we equidistribute the weight function

$$w(r) = 1. + \text{arcLengthWeight} \frac{s(r)}{|s|_\infty} + \text{curvatureWeight} \frac{c(r)}{|c|_\infty}$$

where  $s(r)$  is the local arclength and  $c(r)$  is the curvature. Note that we normalize  $s$  and  $c$  by their maximum values.

**arcLengthWeight\_ (input):** A weight for arclength. A negative value may give undefined results.

**curvatureWeight\_ (input):** A weight for curvature. A negative value may give undefined results.

#### 35.4.6 setEndConditions

```
int  
setEndConditions(const EndCondition & condition,  
                  const RealArray & endValues =Overture::nullRealDistributedArray())
```

**Description:** Specify end conditions for the spline

**condition (input) :** Specify an end condition.

**monotone parabolic fit** : default BC for the shape preserving spline.

**first derivative** : user specified first derivatives.

**second derivative** : user specified second derivatives.

**endValues (input) :** if `condition==firstDerivative` (or `condition==secondDerivative`) then `endValues(0:1,0:r-1)` should hold the values for the first (or second) derivatives of the spline at the start and end. Here  $r=\text{rangeDimension}$ .

### 35.4.7 setPoints

```
int  
setPoints( const RealArray & x )
```

**Purpose:** Supply spline points for a 1D curve.

**x (input) :** array of spline knots. The spline is parameterized by a NORMALIZED index, i/(number of points -1), i=0,1,...

### 35.4.8 setPoints

```
int  
setPoints( const RealArray & x, const RealArray & y )
```

**Purpose:** Supply spline points for a 2D curve. Use the points (x(i),y(i)) i=x.getBase(0),..,x.getBound(0)

**x,y (input) :** array of spline knots.

### 35.4.9 setPoints

```
int  
setPoints( const RealArray & x, const RealArray & y, const RealArray & z )
```

**Purpose:** Supply spline points for a 3D curve. Use the points (x(i),y(i),z(i))  
i=x.getBase(0),..,x.getBound(0)

**x,y,z (input) :** array of spline knots.

### 35.4.10 setShapePreserving

```
int  
setShapePreserving( const bool trueOrFalse = TRUE)
```

**Description:** Create a shape preserving (monotone) spline or not

**trueOrFalse (input) :** if TRUE, create a spline that preserves the shape. For a one dimensional curve the shape preserving spline will attempt to remain monotone where the knots are monotone. See the comments with TSPACK for further details.

### 35.4.11 setTension

```
int  
setTension( const real & tensionFactor )
```

**Description:** Specify a constant tension factor. Specifying this value will turn off the shape preserving feature.

**tensionFactor (input):** A value from 0. to 85. A value of 0. corresponds to no tension.

### 35.4.12 setDomainInterval

```
int  
setDomainInterval(const real & rStart_ =0.,  
                  const real & rEnd_ =1.)
```

**Description:** Restrict the domain of the spline. By default the spline is parameterized on the interval [0,1]. You may choose a sub-section of the spline by choosing a new interval [rStart,rEnd]. For periodic splines the interval may lie in [-1,2] so the sub-section can cross the branch cut. You may even choose rEnd>rStart to reverse the order of the parameterization.

**rStart\_,rEnd\_ (input)** : define the new interval.

### 35.4.13 getDomainInterval

```
int  
getDomainInterval(real & rStart_, real & rEnd_) const
```

**Description:** Get the current domain interval.

**rStart\_,rEnd\_ (output)** : the current domain interval.

### 35.4.14 setIsPeriodic

```
void  
setIsPeriodic( const int axis, const periodicType isPeriodic0 )
```

**Description:**

**axis (input):** axis = (0,1,2) (or axis = (axis1,axis2,axis3)) with *axis* < *domainDimension*.

**Notes:** This routine has some side effects. It will change the boundaryConditions to be consistent with the periodicity (if necessary).

### 35.4.15 useOldSpline

```
int  
useOldSpline( const bool & trueOrFalse =TRUE)
```

**Description:** Use the old spline routines from FMM, Forsythe Malcolm and Moler. This is for backward compatibility.

**trueOrFalse (input)** : If TRUE Use the old spline from FMM, otherwise use the tension splines.

### 35.4.16 map

```
void  
mapS( const RealArray & r, RealArray & x, RealArray & xr, MappingParameters &  
      params )
```

**Purpose:** Evaluate the spline and/or derivatives.

### 35.4.17 update

int

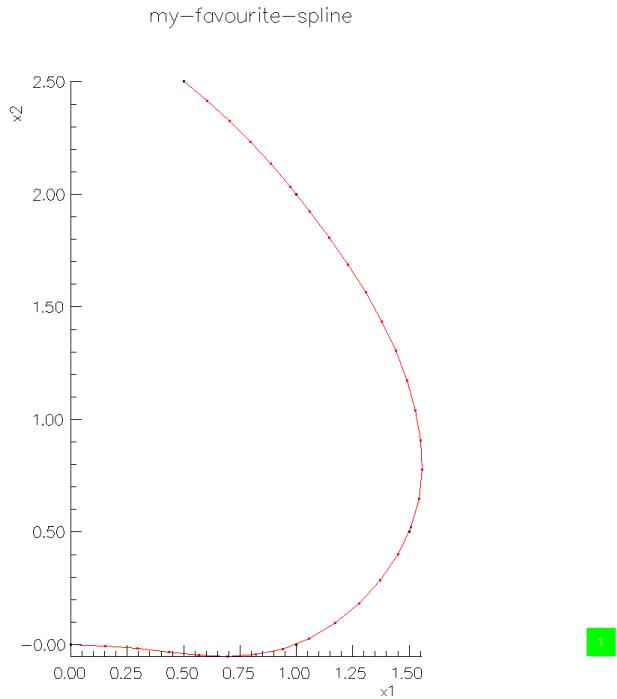
update( MappingInformation & mapInfo )

**Purpose:** Interactively create and/or change the spline mapping.

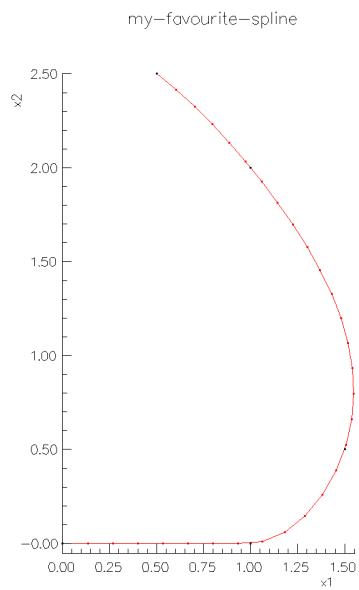
**mapInfo (input):** Holds a graphics interface to use.

## 35.5 Examples

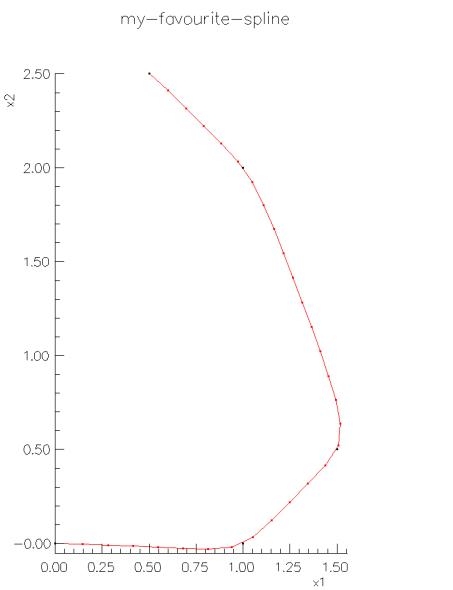
```
1  *
2  * Make a 2D spline curve
3  *
4  spline
5    enter spline points
6      * first enter the number of points
7      5
8      * here are the points (x,y)
9      0.  0.
10     1.  0.
11     1.5 .5
12     1.  2.
13     .5  2.5
14   mappingName
15     my-favourite-spline
```



A spline curve in 2D. No Tension.

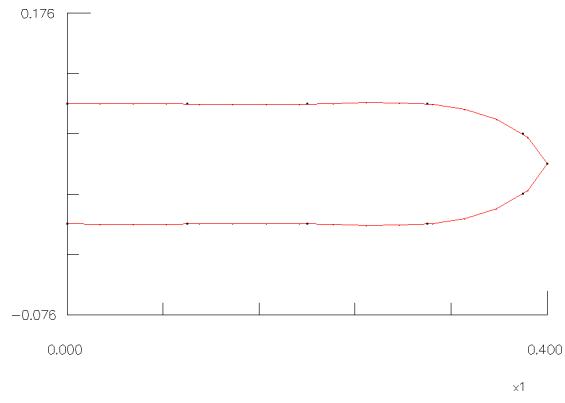


Spline curve with shape preserving option.



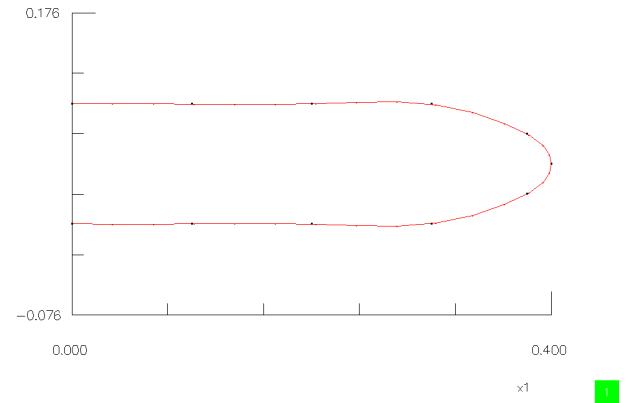
Spline curve with tension=20.

splineMapping



Spline curve with default arclength parameterization.

splineMapping



Spline curve with `curvatureWeight=1` so that more points are put where the curvature is large.

## 36 SquareMapping (rectangles too)

This mapping defines a square or rectangle in two-dimensions

$$\mathbf{x}(r_1, r_2) = (x_a + r_1(x_b - x_a), y_a + r_2(y_b - y_a))$$

### 36.1 Constructor

**SquareMapping( const real xa\_, const real xb\_, const real ya\_, const real yb\_ )**

**Purpose:** Build a mapping for a square with given bounds.

**xa\_, xb\_, ya\_, yb\_ (input) :** The square is  $[x_a, x_b] \times [y_a, y_b]$ .

### 36.2 getVertices

**real**

**getVertices(real & xa\_ , real & xb\_ , real & ya\_ , real & yb\_ ) const**

**Description:** return the vertices of the square.

**xa\_, xb\_, ya\_, yb\_ (output) :** The square is  $[x_a, x_b] \times [y_a, y_b]$ .

**Return value:** is the z-level

### 36.3 setVertices

**void**

**setVertices(const real xa\_ =0.,  
          const real xb\_ =1.,  
          const real ya\_ =0.,  
          const real yb\_ =1.,  
          const real z\_ =0.)**

**Purpose:** Build a mapping for a square with given corners.

**xa\_, xb\_, ya\_, yb\_ (input) :** The square is  $[x_a, x_b] \times [y_a, y_b]$ .

**z\_ :** z level if the rangeDimension is 3.

## 37 StretchMapping: create 1D stretching functions

The StretchMapping class, derived from the Mapping Class can be used to define one-dimensional “stretching” functions. These functions are often used to stretch grid lines on existing Mappings. These functions can also be used as a blending function for the TFI Mapping.

There are several types of stretching functions:

### 37.1 Inverse hyperbolic tangent stretching function

This stretching function is a one-dimensional map from  $r$  into  $x$  defined (in an inverse fashion) by

$$r = R(x) = \left[ x + \sum_{i=1}^{nu} (U_i(x) - U_i(0)) + \sum_{j=1}^{nv} (V_j(x) - V_j(0)) \right] \times \text{scale} + \text{origin}$$

where  $U_i(x)$  is a “layer” function

$$U_i(x) = \frac{a_i}{2} \tanh b_i(x - c_i)$$

and  $V_i(x)$  is an “interval” function

$$V_j(x) = \frac{d_j - 1}{2} \log \left( \frac{\cosh e_j(x - f_j)}{\cosh e_j(x - f_{j+1})} \right) \frac{1}{2e_j}$$

The stretching mapping is often used to stretch grid points in parameter space. The functions  $U_i$  are used to concentrate grid points in at a point while the functions  $V_j$  are used to transition from one grid spacing to another. When the mapping is invertible a spline can be fitted to the inverse to be used as an initial guess for Netwon. Usually only 1-3 Netwon iterations are needed.

Here the terms scale and origin are normalization factors determined so that  $R(0) = 0$  and  $R(1) = 1$ . The remaining parameters are input by the user and have the following constraints:

$$\begin{aligned} b_j &> 0, \quad j = 1, \dots, n_u, \\ 0 \leq c_j &\leq 1, \quad j = 1, \dots, n_u, \\ e_j &> 0, \quad j = 1, \dots, n_v, \\ f_1 \leq 1, \quad f_{n_v} &\geq 0, \quad f_j \leq 1, \quad j = 2, \dots, n_v - 1, \quad \text{and} \\ f_1 < f_2 < f_3 &< \dots < f_{n_v}. \end{aligned}$$

The function  $U_i(x)$  is a hyperbolic tangent that is centered at  $x = c_i$  and asymptotes to  $-a_i/2$  or  $a_i/2$  (see Figure 22). As  $b_i$  tends to infinity, the function  $U$  tends toward a step function.

The function  $V_j(x)$  (which is the integral of the difference between two layer functions) is a smoothed-out ramp function with transitions at  $f_j$  and  $f_{j+1}$  (see Figure 22). The slope of the ramp is  $d_{j-1}$ . Thus  $d_j$  indicates the relative slope of the ramp compared to the linear term “ $x$ ,” which appears in  $R(x)$ . That is, if  $d_j = 2$ , then the slope of  $R(x)$  between  $f_j$  and  $f_{j+1}$  will be approximately twice the slope of the region where the linear term is dominant. A sloped region can be made to extend past  $x = 0$  or  $x = 1$  (so that  $x = 0$  or  $x = 1$  is in the middle of the sloped region) by choosing  $f_1 < 0$  or  $f_{n_v+1} > 1$ . A reasonable value might be  $f_1 = -.5$  or  $f_{n_v+1} = 1.5$ . Note that when a grid is periodic in the  $r$ -direction, the functions  $U_i(x)$  and  $V_j(x)$  are replaced by functions  $U_i^p(x)$  and  $V_j^p(x)$ , respectively, which are given by

$$U_i^p(x) = \sum_{k=-\infty}^{+\infty} U_i(t+k), \quad V_j^p(x) = \sum_{k=-\infty}^{+\infty} V_j(t+k).$$

These functions are not really periodic, but their derivatives with respect to  $x$  are periodic with period 1.

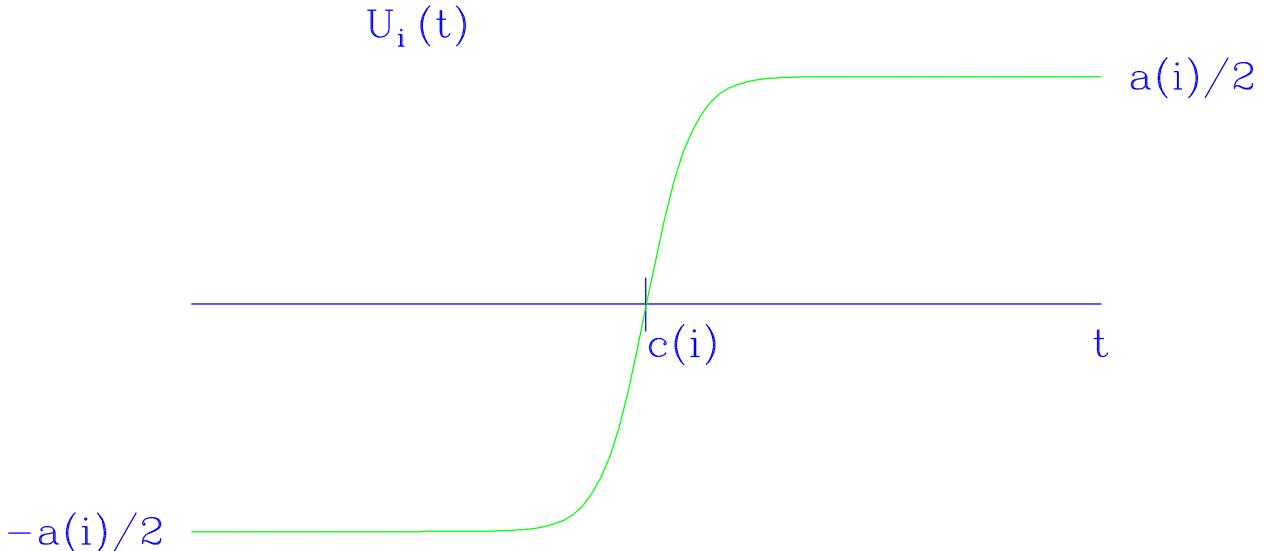


Figure 21: The ‘layer’ function  $U(t)$  for concentrating grid lines at a point. The grid spacing is smaller where the slope is larger.

The following remarks may prove useful in making choices for the parameters  $a_i, \dots, f_i$ . Below, the variable  $r$  typically refers to a uniform grid, while  $x$  refers to a grid that has been stretched so that points are clustered in certain locations on the  $x$  axis. The clustering of points can be done in two ways. Using the  $U_i(x)$  functions (tanh’s), the point spacing can be made to decrease exponentially to a minimum spacing at  $c_i$ . The value of  $b_i$  determines how small the spacing can get. Roughly speaking, a value of  $b_i = 10.0$  means the spacing will be about 10 times smaller at the center of the layer than in the unstretched parts of the grid. The relative number of points in this stretched region is proportional to  $a_i$ . The linear term  $x$  appearing in the definition of  $R(x)$  has a weight of one (1), so if there is only one term  $U_i(x)$ , the relative number of points in the layer is essentially  $a_1/(1 + a_1)$ . Thus, if  $n_u = 1$ , ( $n_v = 0$ ), and  $a_1 = 1$ , then half the points will be in the stretched layer. For two layers, the relative number of points in layer  $i$  ( $i = 1$  or  $i = 2$ ) is  $a_i/(1 + a_1 + a_2)$ . The functions  $V_j(x)$  allow you to have intervals where the grid point spacing is relatively smaller or larger than the grid spacing in the region where the linear term  $x$  is dominant. In each interval, the grid spacing is nearly constant, except near the transition points  $f_i$  and  $f_{i+1}$ . The parameter  $d_i$  denotes the relative grid spacing in each interval. For example, to make the grid spacing twice as fine for  $t$  between 0.25 and 0.5, you would specify  $f_1 = 0.25$ ,  $f_2 = 0.5$ , and  $d_1 = 2$ . As another example, to make the spacing 5 times smaller for  $x$  between 0 and 0.5, you could say  $f_1 = -0.5$ ,  $f_2 = 0.5$ , and  $d_1 = 5$ . Assigning the first transition point a value less than zero,  $f_1 = -0.5$ , means that  $x = 0$  will be in the middle of the interval where the spacing will be 5 times smaller. (If instead  $f_1 = 0$ , then near  $t = 0$  the spacing would be in transition to the default relative grid spacing of 1). The parameters  $e_i$  denote how rapid the transition is from one spacing to another. A reasonable value for  $e_i$  might be 10.0 or 20.0.

Figure 24 shows an example of using the inverse hyperbolic tangent function to stretch grid lines on a square.

### 37.2 Hyperbolic tangent stretching function

This function is defined as

$$x(r) = \{a_0 + a_r r + a_1 \tanh(b_1(r - c_1)) + \text{origin}\} \text{ scale}$$

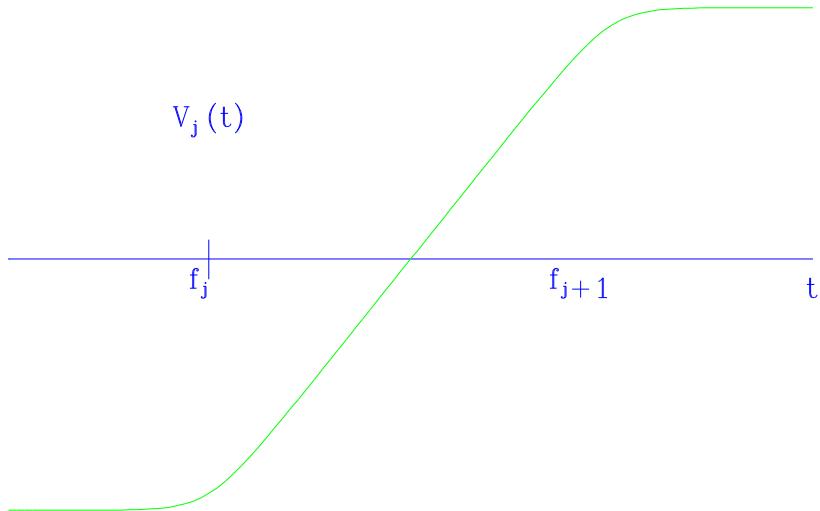


Figure 22: The ‘ramp’ function  $V(t)$  for changing from one grid spacing to another. The grid spacing is smaller where the slope is larger.

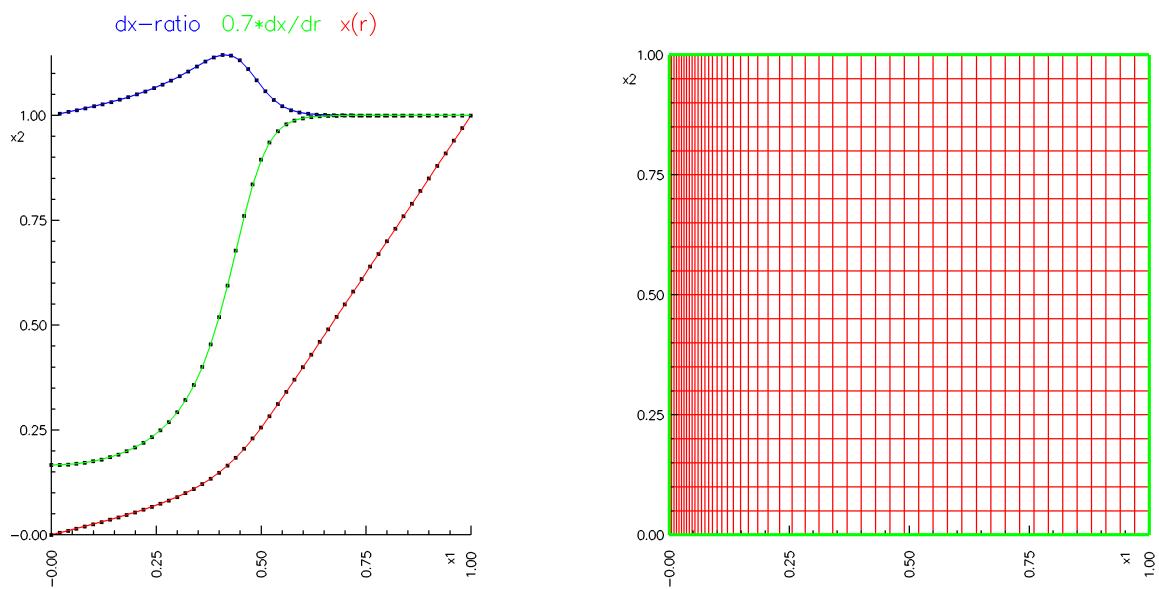


Figure 23: The inverse hyperbolic tangent stretching function (red curve on left) used to cluster grid points on a square.

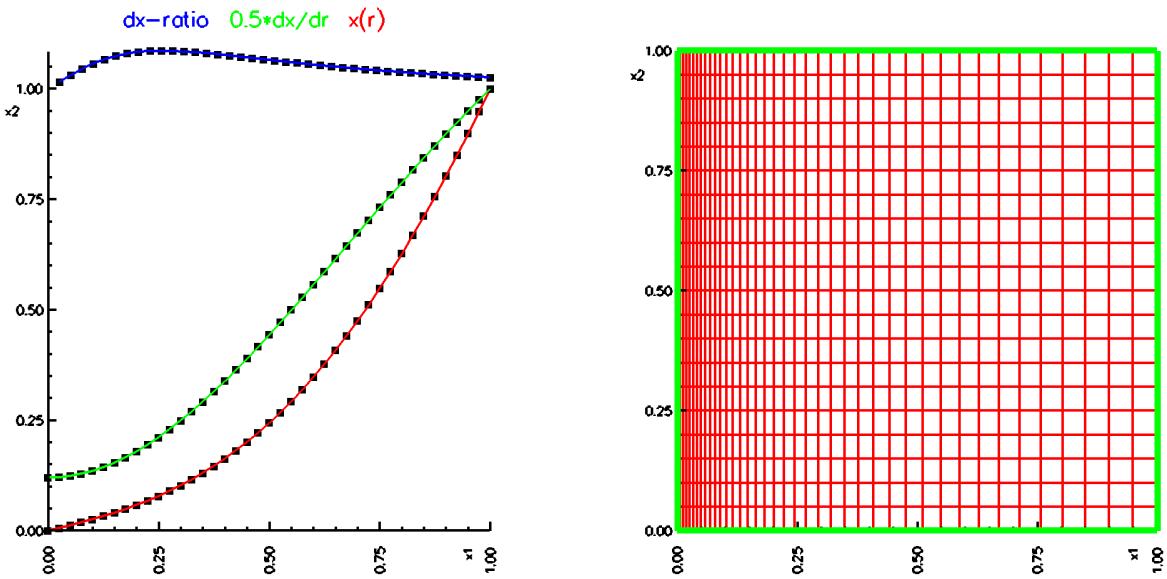


Figure 24: The hyperbolic tangent stretching function (red curve on left) used to cluster grid points on a square.

If the function is normalized (optional) then origin and scale are chosen to that  $x(0) = 0$  and  $x(1) = 1$ . Note that  $a_1$  will normally be negative in order to concentrate lines near  $r = c_1$ . To be invertible one should choose  $a_r > -a_1 b_1$  (a sufficient but not necessary condition).

Figure 24 shows an example of using the hyperbolic tangent function to stretch grid lines on a square.

### 37.3 Exponential stretching function

This function is defined as

$$x(r) = \{a_0 + a_r r + a_1 \exp(b_1(r - c_1)) + \text{origin}\} \text{scale}$$

If the function is normalized (optional) then origin and scale are chosen to that  $x(0) = 0$  and  $x(1) = 1$ .

### 37.4 Exponential blending function

This function is defined as

$$x(r) = \begin{cases} 1 & \frac{3}{4} \leq s \leq 1 \\ \left[ 1 + \exp\left(-\frac{\sqrt{3}}{4} \frac{2s-1}{(s-\frac{1}{4})(\frac{3}{4}-s)}\right) \right]^{-1} & \frac{1}{4} < s < \frac{3}{4} \\ 0 & 0 \leq s \leq \frac{1}{4} \end{cases}$$

This function is used by the `FilletMapping` in order to make a smooth curve in the region where two curves intersect.

### 37.5 Exponential to Linear function

The exponential to linear stretching function can be used to transition between an exponentially stretched grid at one boundary and a uniform distribution of points at the opposite boundary as shown in Figure 25.

(The version of this function for stretching at an interior point is described below). This stretching function is defined as

$$x(r) = \left\{ \log(1 + ae^{sb(r-c)}) - x_0 \right\} d \quad (18)$$

where  $x_0$  and  $d$  are chosen so that  $x(0) = 0$  and  $x(1) = 1$ . The parameter  $c$  takes the values  $c = 0$  and  $c = 1$  to stretch at  $r = 0$  or  $r = 1$ . The parameter  $s$  is  $+1$  or  $-1$  depending on  $c$ ,  $s = 1 - 2c$ . The parameter  $a$  is normally taken to be small and is approximately the ratio of the finest grid spacing to the largest grid spacing. The derivative of this stretching function is

$$\begin{aligned} x_r(r) &= \left\{ \frac{asbe^{sb(r-c)}}{1 + ae^{sb(r-c)}} \right\} d, \\ &= (sbd) \phi(ae^{sb(r-c)}), \\ \phi(y) &= \frac{y}{1 + y}. \end{aligned}$$

The grid spacing is  $x_r(r)\Delta r$  (where  $\Delta r$  is the grid spacing in  $r$ ) and for  $a \ll 1$ ,

$$\begin{aligned} x_r(c) &\approx (sbd)a, \\ x_r(1 - c) &\approx (sbd) \end{aligned}$$

and thus the grid spacing at  $r = c$  is a factor of  $a$  times that at  $r = 1 - c$ . The function  $\phi(y)$  is used to transition between an exponentially increasing grid spacing when  $y = ae^{sb(r-c)}$  is small (since  $\phi(y) \sim y$  for  $y \ll 1$ ), and a constant grid spacing for  $y \gg 1$  (since  $\phi(y) \sim 1$  for  $y \gg 1$ ). The parameter  $b$  determines how fast the grid spacing increases near  $r = c$ . The grid spacings will increase by a factor of about  $e^{b\Delta r}$ . Normally for PDE applications  $b$  and  $\Delta r$  should be chosen so that the growth factor  $\kappa = e^{b\Delta r}$  is not be too large or else the grid metrics change too rapidly resulting in lower accuracy. Typically  $\kappa < 1.15$  for example. The parameter  $b$  might usefully be chosen so that  $ae^b = B$  where  $B$  is fairly large ( $B \approx 10$ ) since then  $\phi(B) \approx 1 - 1/B$  and thus the function will be nearly linear at  $r = 1 - c$ .

The exponential to linear stretching function was derived by first choosing  $\phi(y)$  and then integrating to get  $x(r)$ . Besides the choice  $\phi(y) = y/(1 + y)$ ,  $\phi$  could also be chosen as  $\phi = \arctan(y)$  or  $\phi = \tanh(y)$ , for example.

The above exponential to linear stretching function can be extended to provide exponential stretching at an interior point  $c$  in the domain as shown in Figure 26. In this case the function is defined by

$$x(r) = \left\{ \log \left( \frac{1 + ae^{b(r-c)}}{1 + ae^{-b(r-c)}} \right) - x_0 \right\} d \quad (19)$$

## 37.6 Member function descriptions

### 37.6.1 Constructor

**StretchMapping( const StretchingType & stretchingType\_ = noStretching )**

**Purpose:** Construct a function with the given stretching type, one of

**inverseHyperbolicTangent** : the most commonly used stretching function defined in an inverse way as a combination of hyperbolic tangents and logarithms of hyperbolic cosines.

**hyperbolicTangent** : hyperbolic tangent stretching.

**exponential** : exponential stretching.

**exponentialBlend** : a  $C^\infty$  blending function that is exactly 0 for  $r < \frac{1}{4}$  and exactly 1 for  $r > \frac{3}{4}$ .

**exponentialToLinear** exponential stretching that transitions to linear stretching

**stretchingType\_ (input):**

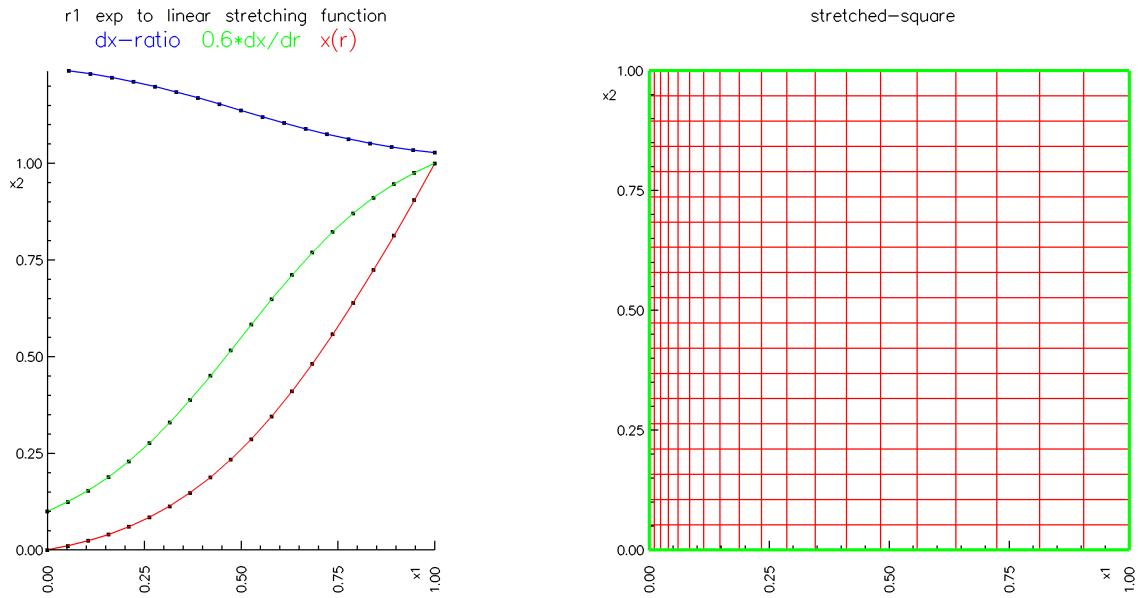


Figure 25: The exponential to linear stretching function (red curve on left) transitions from an exponential function to a linear function. The grid spacing increases exponentially near the left and transitions to a constant grid spacing on the right.

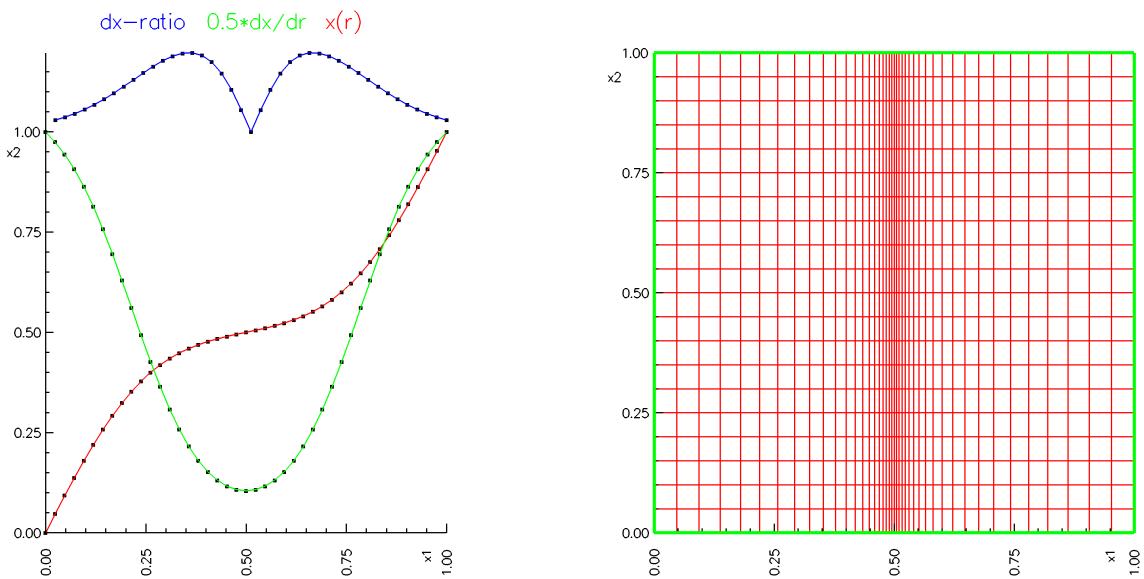


Figure 26: The exponentialToLinear stretching function for clustering points at an interior location.

### 37.6.2 Constructor

```
StretchMapping( const int numberOfLayers_,  
                const int numberOfIntervals_ = 0)
```

**Purpose:** Construct an `inverseHyperbolicTangent` stretching function.

**numberOfLayers\_ (input):** number of layers.

**numberOfIntervals\_ (input):** number of intervals.

### 37.6.3 setStretchingType

**int**

```
setStretchingType( const StretchingType & stretchingType_ )
```

**Description:** Set the stretching type, one of

**inverseHyperbolicTangent** : the most commonly used stretching function defined in an inverse way as a combination of hyperbolic tangents and logarithms of hyperbolic cosines.

**hyperbolicTangent** : hyperbolic tangent stretching.

**exponential** : exponential stretching.

**exponentialBlend** : a  $C^\infty$  blending function that is exactly 0 for  $r < \frac{1}{4}$  and exactly 1 for  $r > \frac{3}{4}$ .

**exponentialToLinear** exponential stretching that transitions to linear stretching

**stretchingType\_ (input):**

### 37.6.4 setNumberOfLayers

**int**

```
setNumberOfLayers( const int numberOfLayers_ )
```

**Description:** Set the number of layer (`tanh`) functions in the `inverseHyperbolicTangent` stretching function.

**numberOfLayers\_ (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.5 setNumberOfIntervals

**int**

```
setNumberOfIntervals( const int numberofIntervals_ )
```

**Description:** Set the number of interval (`log(cosh)`) functions in the `inverseHyperbolicTangent` stretching function.

**numberOfIntervals\_ (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.6 setNumberOfSplinePoints

int

**setNumberOfSplinePoints( const int numberOfSplinePoints0 )**

**Description:** Set the number of interval ( $\log(\cosh)$ ) functions in the `inverseHyperbolicTangent` stretching function.

**a,b,c (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.7 setLayerParameters

int

**setLayerParameters( const int index, const real a, const real b, const real c )**

**Description:** Set parameters for the interval ( $\log(\cosh)$ ) function numbered `index`.

**a,b,c (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.8 setIntervalParameters

int

**setIntervalParameters( const int index, const real d, const real e, const real f )**

**Description:** Set parameters for the interval ( $\log(\cosh)$ ) function numbered `index`.

**d,e,f (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.9 setEndPoints

int

**setEndPoints( const real rmin, const real rmax )**

**Description:** Set the end points for the `inverseHyperbolicTangent` stretching function.

**rmin,rmax (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.10 setIsNormalized

int

**setIsNormalized( const bool & trueOrFalse =TRUE )**

**Description:** Indicate whether the stretching function should be normalized to go from 0 to 1.

**trueOrFalse (input):** if TRUE the function is normalized.

### 37.6.11 setScaleParameters

```
int  
setScaleParameters( const real origin_, const real scale_ )
```

**Description:** Set the origin and scale parameters for the `inverseHyperbolicTangent` stretching function.

**origin\_, scale\_ (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.12 setIsPeriodic

```
int  
setIsPeriodic( const int trueOrFalse )
```

**Description:** Define the periodicity of the function, only applies to the `inverseHyperbolicTangent` stretching function.

**trueOrFalse (input):** TRUE or FALSE.

**Return value:** 0 on success, 1 if the stretching type has not been set to `inverseHyperbolicTangent` in which case no changes are made.

### 37.6.13 setHyperbolicTangentParameters

```
int  
setHyperbolicTangentParameters(const real & a0_,  
                                const real & ar_,  
                                const real & a1_,  
                                const real & b1_,  
                                const real & c1_)
```

**Description:** Set the parameters for the `hyperbolicTangent` stretching function.

**a0\_,ar\_,a1\_,b1\_,c1\_, (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `hyperbolicTangent` in which case no changes are made.

### 37.6.14 setExponentialParameters

```
int  
setExponentialParameters(const real & a0_,  
                        const real & ar_,  
                        const real & a1_,  
                        const real & b1_,  
                        const real & c1_)
```

**Description:** Set the parameters for the `exponential` stretching function.

**a0\_,a1\_,b1\_,c1\_, (input):**

**Return value:** 0 on success, 1 if the stretching type has not been set to `exponential` in which case no changes are made.

### 37.6.15 setLinearSpacingParameters

```
int  
setLinearSpacingParameters(const real & a0_, const real & a1_)
```

**Description:** Set the parameters for the `linear spacing` stretching function. The grid spacing for this stretching will exponentially increase from `a0` to `a1`. (wdh)

**a0, a1 (input):** specify the grid spacings at  $r=0$  and  $r=1$

**Return value:** 0 on success, 1 if the stretching type has not been set to `linearSpacing` in which case no changes are made.

### 37.6.16 setExponentialToLinearParameters

```
int  
setExponentialToLinearParameters(const real & a,  
                                 const real & b,  
                                 const real & c )
```

**Description:** Set the parameters for the `exponentialToLinear` stretching function.

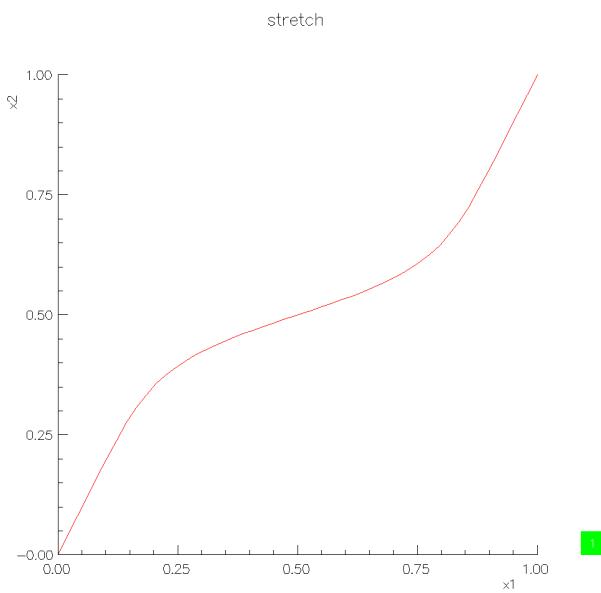
**a,b,c (input):** see documentation for the formula.

**Return value:** 0 on success, 1 if the stretching type has not been set to `exponential` in which case no changes are made.

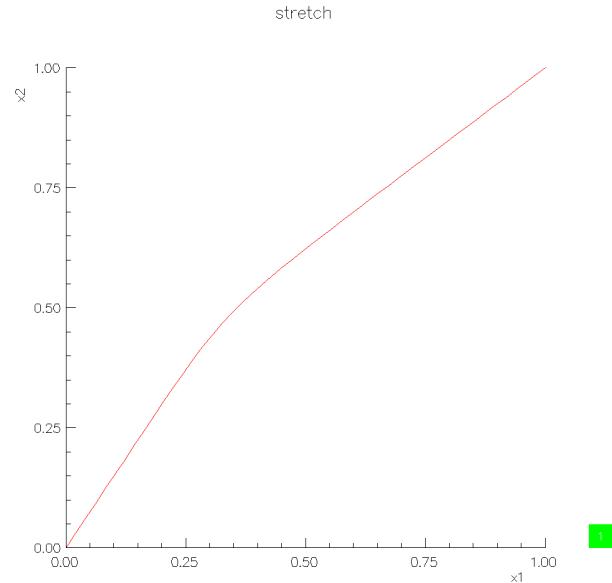
## 37.7 Examples

Here is an example of the use of the `StretchMapping` class.

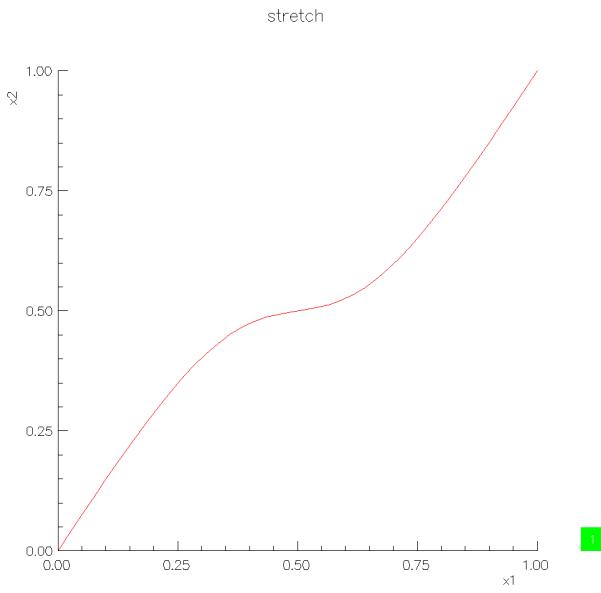
```
#include "Stretch.h"  
  
void main()  
{  
    const int axis1 = 0;  
    const int axis2 = 1;  
    const int axis3 = 2;  
    realArray r(1,3);  
    realArray t(1,3);  
    realArray tr(1,3,3);  
  
    StretchMapping stretch1( 2, 0 );           // two layers, zero intervals  
    stretch1.setLayerParameters( 0, 1., 10., .25 ); // set layer 0, a,b,c  
    stretch1.setLayerParameters( 1, 1., 10., .75 ); // set layer 1, a,b,c  
    stretch1.setIsPeriodic(FALSE);             // default is FALSE  
  
    r(0,axis1)=.5;                           // evaluate  
    stretch1.map( r,t,tr );  
  
    StretchMapping stretch2( 0, 1 );           // zero layers, one interval  
    stretch2.setIntervalParameters( 0, 5., 20., .25 ); // spacing is smaller  
    stretch2.setIntervalParameters( 1, 0., 0., .75 ); // between .25 and .75  
    stretch2.setIsPeriodic(FALSE);             // default is FALSE  
  
    r(0,axis1)=.25;                          // evaluate  
    stretch2.map( r,t,tr );  
}
```



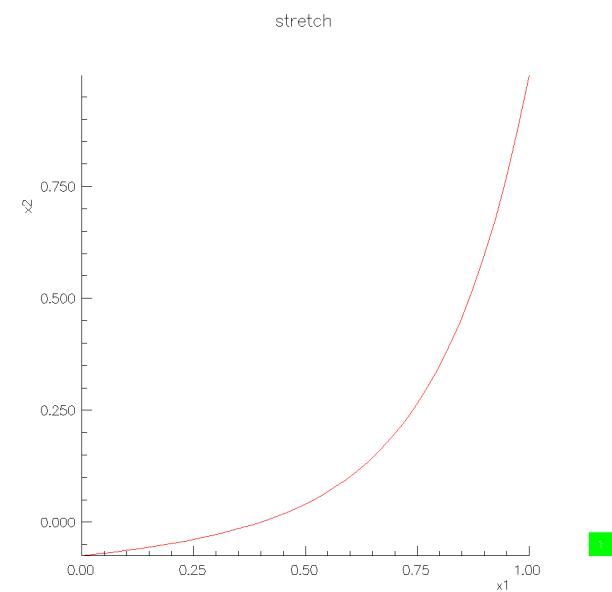
Stretching function: inverseHyperbolicTangent, 1 layer,  $a_0 = 1.$ ,  $b_0 = 10.$ ,  $c_0 = .5$ . This function will concentrate grid points near  $r = .5$



Stretching function: inverseHyperbolicTangent, 1 interval,  $d_0 = 2.$ ,  $e_0 = 10.$ ,  $f_0 = .5$ ,  $f_1 = 1.5$ . This function will have grid spacing that is twice as small for  $r > .5$

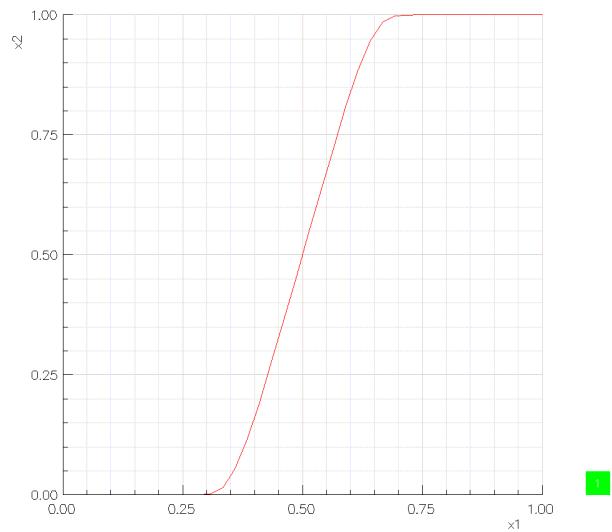


Stretching function: hyperbolicTangent,  $a_0 = 0.$ ,  $a_r = 1.$ ,  $a_1 = -.9a_0/b_1$ ,  $b_1 = 5.$ ,  $c_1 = .5$ . This function will concentrate grid points near  $r = .5$



Stretching function: exponential,  $a_0 = 0.$ ,  $a_r = 1.$ ,  $a_1 = 1.$ ,  $b_1 = 5.$ ,  $c_1 = .5$ . This function will have grid spacing that concentrated near  $r = 0$ .

stretch



Stretching function: exponentialBlend. This function is exactly 0. for  $r < \frac{1}{4}$  and exactly 1 for  $r > \frac{3}{4}$ .

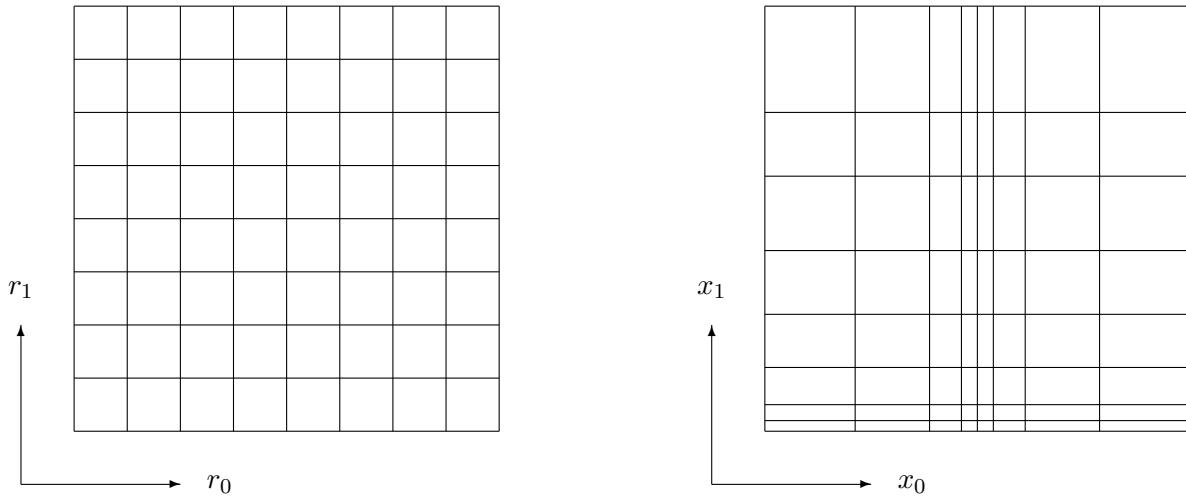


Figure 27: The `StretchedSquare` class can be used to stretch grids lines on the unit square or unit cube. The class defines a mapping from the parameter space  $(r_0, r_1)$  to the parameter space  $(x_0, x_1)$ . Grids lines can be stretched on other Mapping's (such as an Annulus) by using the `StretchTransform` class which composes a Mapping with a `StretchedSquare`.

## 38 StretchedSquare: stretch grid lines on the unit interval

### 38.1 Description

This mapping uses the stretching functions of the `StretchMapping` class to stretch grid lines on the unit square or unit cube. The `StretchedSquare` defines a mapping from parameter space to parameter space. It can be used to stretch grids lines on other mappings (such as an annulus) using the `StretchTransform` class.

## 39 StretchTransform: stretch grid lines of an existing mapping

### 39.1 Description

This mapping can be used to reparameterize another mapping by stretching the grid lines in the parameter directions. It does this by composing the `StretchedSquare` mapping with the given mapping. The `StretchedSquare` mapping in turn uses the `StretchMapping` to create stretching functions.

### 39.2 Constructors

`StretchTransform()` Default constructor

### 39.3 Data Members

### 39.4 Member Functions

<code>void map( ... )</code>	evaluate the mapping and derivative
<code>void inverseMap( ... )</code>	evaluate the inverse mapping and derivative
<code>void get( const Dir &amp; dir, const String &amp; name )</code>	get from a database file
<code>void put( const Dir &amp; dir, const String &amp; name )</code>	put to a database file

## 40 Sweep Mapping

The SweepMapping can be used to create these types of mapping's,

**sweep** : sweep a curve or surface in the  $x - y$  plane along a 3D curve.

**extrude** : extrude a curve or surface in the  $z$  direction.

**tabulated-cylinder** : generate a surface from a 3D curve by extruding along a specified line.

### 40.1 Sweep

The **sweep** option of the SweepMapping will take a planar reference surface  $\mathbf{S}(r_1, r_2)$  (or reference curve  $\mathbf{S}(r_1)$ ) and form a three-dimensional volume (or surface) by sweeping the reference surface along a 3D 'sweep-curve'  $\mathbf{C}(r_3)$ .

The formula defining the sweep mapping is

$$\mathbf{X}(r_1, r_2, r_3) = \left\{ \mathbf{M}(\mathbf{r}) [\mathbf{S}(r_1, r_2) - \mathbf{c}_0] \right\} \alpha(r_3) + \mathbf{C}(r_3)$$

where

**S** : the reference surface (or reference curve) to be swept.

**C** : the curve used for sweeping, the *sweep-curve*.

**M** : a rotation-translation matrix defined from **C** and **S**.

$\alpha(r_3)$  : a scaling function.

**c**<sub>0</sub> a vector used to centre the sweep mapping in different ways.

The vector **c**<sub>0</sub> determines the centering of the SweepMapping with respect to the reference surface. There are three options for specifying the centering of the SweepMapping,

**c**<sub>0</sub> = 0 : the centering is based on the sweep curve

**c**<sub>0</sub> =  $\bar{\mathbf{S}}(\cdot, \cdot)$  : the centering is based on the reference surface.

**c**<sub>0</sub> = user-specified : the centering is user specified.

Here  $\bar{\mathbf{S}}(\cdot, \cdot)$  is the centroid of the reference surface.

The initial rotation-translation matrix  $\mathbf{M}(r_1, r_2, 0)$  will be chosen to translate the reference surface so that it's centroid,  $\bar{\mathbf{S}}(\cdot, \cdot)$ , is located at the point **c**<sub>0</sub>. The centroid is defined as the average value of the grid point locations,

$$\bar{\mathbf{S}}(\cdot, \cdot) = \frac{\sum_{i_1=0}^{n_1} \sum_{i_2=0}^{n_2} \mathbf{S}_{i_1, i_2}}{(n_1 + 1) (n_2 + 1)}.$$

$\mathbf{M}(r_1, r_2, 0)$  will also rotate the reference surface to align with the tangent to the sweep-curve. After this rotation the the normal to the rotated reference surface will be parallel to the initial tangent of the the sweep-curve,  $\mathbf{C}'(0)$ . The **orientation** parameter (+1 or -1) will determine whether the normal to **S** is in the same or opposite direction to the tangent to the sweep curve. Thus if the sweep mapping appears 'inside-out' one should change the orientation. Instead of changing the orientation one could also reverse the parameterization of the sweep curve.

This needs to be finished

**Here is the old documentation.**

Purpose:

Given a planar surface (or curve)  $\mathbf{S}(r_1, r_2)$  (or  $\mathbf{S}(r_1)$ ), and a 3D curve  $\mathbf{C}(r_3)$ , we would like to generate a

3D volume or surface by sweeping  $\mathbf{S}$  perpendicularly to  $\mathbf{C}$  in such a way that the center of each  $\mathbf{S}_k$  ring lie on the curve  $\mathbf{C}$ . At  $r_3 = 0$ , it is assumed that  $\mathbf{S} = S_0$  is orthogonal to  $\mathbf{C}$  and the tangent to  $\mathbf{C}$  coincide with the normal  $\mathbf{n}$  to  $\mathbf{S}$ . To make sure that the center of  $\mathbf{S} = S_0$  lies at  $\mathbf{C}(0)$ , We first find the center  $(x_0, y_0, z_0)$  as the average of all the points that make up the sweep surface  $\mathbf{S}$ , namely

$$\mathbf{x}_0 = \frac{\sum_{i=0}^n \mathbf{x}_i}{n+1},$$

Then a translation that maps  $\mathbf{C}(0)$  to  $(x_0, y_0, z_0)$  is applied to  $\mathbf{C}$ .

#### Strategy:

With a sufficient number of grid points in each direction, we incrementally compute the matrix transformation to be used the following way. At  $k = 0$  corresponding to  $r_3 = 0$ , the identity matrix is used since  $\mathbf{S}$  and  $\mathbf{C}$  satisfy the required conditions and  $\mathbf{S}_0 = \mathbf{S}$ . For  $k > 0$ , the ring  $\mathbf{S}_k$  is gotten from the ring  $\mathbf{S}_{k-1}$  the following way:

A translation that maps the center of  $\mathbf{S}_{k-1}$  (which is the same point as  $\mathbf{C}_{k-1}$ ) to the point  $\mathbf{C}_k$  is applied to  $\mathbf{S}_{k-1}$ . A rotation is then applied to the resulting points is such a way that the unit normal to the surface  $\mathbf{S}_{k-1}$  coincides with the tangent to the curve  $\mathbf{C}$  at the point  $C_k$ . To implement this, the unit vector  $\mathbf{n}_0$  of the surface  $\mathbf{S}_{k-1}$  is chosen to be the first vector in a new orthonormal basis. The second basis vector  $\mathbf{n}_1$  is given by  $\mathbf{n}_1 = \frac{n_0 \times t}{\|n_0 \times t\|}$  where  $t = \frac{\partial C(r_3 + \Delta r_3)}{\partial r_3}$ . The third basis vector  $\mathbf{n}_2$  is given by  $\frac{n_0 \times n_1}{\|n_0 \times n_1\|}$ . In the new coordinate system, the rotation is about  $\mathbf{n}_1$  with center at  $C_k$ . Since  $n_0$  is rotated to coincide with  $t$ , the rotation angle is given by  $\cos \theta = n_0 \cdot t$  and  $\sin \theta = t \cdot n_2$ . The overall matrix transformation is therefore a product of three matrices; first the matrix transformation from the canonic basis of the 3D vector space to the basis  $(n_0, n_1, n_2)$ , the rotation of angle  $\theta$  with center  $(0, 0, 0)$  around  $\mathbf{n}_1$  and finally the matrix transformation from the basis  $(n_0, n_1, n_2)$  to the canonic basis.

For the simplification of the mapping calculations, the discrete values of the global transformation  $M(r_{1k}, r_{2k}, r_{3k})$  are considered as the points for three splines. With these splines we can calculate the image of any triplet  $(r_1, r_2, r_3)$ . If  $\alpha(r_3)$  is the value of the scalar we will multiply (also stored in a spline), the image  $\mathbf{X}(r_1, r_2, r_3)$  is given by

$$\mathbf{X}(r_1, r_2, r_3) = \{M(r_1, r_2, r_3) * [\mathbf{S}(r_1, r_2) - \mathbf{C}(0)]\} \alpha(r_3) + \mathbf{C}(r_3)$$

#### Remark

At the limit ( $\Delta r_3 \rightarrow 0$ ) corresponding to the continuous case, the basis  $(n_0, n_1, n_2)$  becomes proportional to  $\frac{\partial C(r_3)}{\partial r_3}, \frac{\partial^2 C(r_3)}{\partial r_3^2}, \frac{\partial C(r_3)}{\partial r_3} \times \frac{\partial^2 C(r_3)}{\partial r_3^2}$ . In fact when  $\Delta r_3$  is very small then

$$\begin{aligned} n_1 &\approx \frac{\partial C(r_3)}{\partial r_3} \times \frac{\partial C(r_3 + \Delta r_3)}{\partial r_3} \\ n_1 &\approx \frac{\partial C(r_3)}{\partial r_3} \times \left( \frac{\partial C(r_3)}{\partial r_3} + \Delta r_3 \frac{\partial^2 C(r_3)}{\partial r_3^2} + \dots \right) \\ &\approx \Delta r_3 \frac{\partial C(r_3)}{\partial r_3} \times \frac{\partial^2 C(r_3)}{\partial r_3^2} \end{aligned}$$

Acknowledgement: Thanks to Thomas Rutaganira for creating the first version of the SweepMapping.

Here are the description of some functions of the class

## 40.2 Constructor

SweepMapping(Mapping \*sweepmap = NULL,

```
Mapping *dirsweepmap = NULL,  
Mapping *scale = NULL,  
const int domainDimension0 =3)
```

**Description:** Define a sweep mapping or an extruded mapping.

Build a mapping defined by a sweep surface or curve (a mapping with domainDimension=2 rangeDimension=3 or domainDimension=1, rangeDimension=3) and a sweep curve or line (domainDimension=1, rangeDimension=3).

**sweepmap (input) :** is the mapping for the sweep surface or curve; default: an annulus with inner radius=0 and outer radius=1

**dirsweepmap (input) :** The mapping for the sweep curve; default: a half circle of radius=4.

**scale (input) :** to scale up ( $> 1$ ) or down ( $0 < s < 1$ ); default 1.

**Author:** Thomas Rutaganira.

**Changes:** WDH + AP

#### 40.3 SetSweepSurface

```
void  
setSweepSurface(Mapping *sweepmap)
```

**Description:** Specify the mapping to use as the sweepMap, a 3D surface or a 3D curve. If it is a 3D surface, the resulting SweepMapping will be a 3D volume and if it is a 3D curve, the SweepMapping will be a 3D surface.

#### 40.4 setCentering

```
int  
setCentering( CenteringOptionsEnum centering )
```

**Description:** Specify the centering.

**centering (input) :** Specify the manner in which the reference surface should be centered. One of **useCenterOfSweepSurface**, **useCenterOfSweepCurve** or **specifiedCenter**. See the documentation for further details.

#### 40.5 setOrientation

```
int  
setOrientation( real orientation_ =1.)
```

**Description:** Specify the orientation of the sweepmapping, +1 or -1. When the sweep surface is rotated to align with the sweep curve it may face in a forward or reverse direction depending on the orientation. Thus if a swept surface appears ‘inside-out’ one should change the orientation.

## 40.6 setExtrudeBounds

```
int  
setExtrudeBounds(real za_=0.,  
                  real zb_=1.)
```

**Description:** Specify the bounds on an extruded mapping.

za\_,zb\_ (input) :

## 40.7 setStraightLine

```
int  
setStraightLine(real lx =0. /*, real ly /* =0. /*, real lz /* =1.)
```

**Description:** Specify the straight line of a tabulated cylinder mapping

lx,ly,lz (input) :

## 40.8 SetSweepCurve

```
void  
setSweepCurve(Mapping *dirsweepmap)
```

**Description:** Specify the mapping to use as the curve to sweep along (a 3D curve).

## 40.9 SetScaleSpline

```
void  
setScale(Mapping *scale)
```

**Description:** Specify the mapping to use as the curve to sweep along (a 3D curve).

## 40.10 setMappingProperties

```
int  
setMappingProperties()
```

Access: protected.

**Description:** Initialize the parameters of the sweep mapping.

## 40.11 FindRowSplines

```
void SweepMapping  
findRowSplines(void)
```

**Description:** This function initializes the splines rowSpline0, 1, 2 that will give the matrix transformation as well as its derivatives for the mapping calculations. A point of the spline gives a row for the matrix transformation.

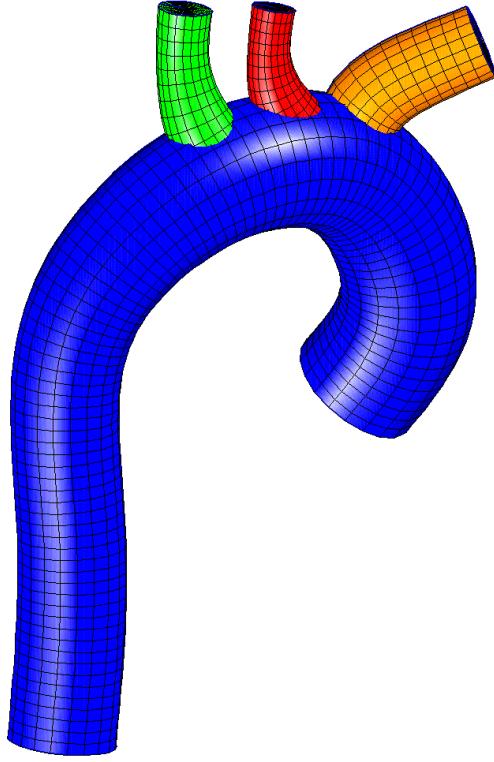


Figure 28: The SweepMapping is used to generate mappings for the aortic arch.

#### 40.12 map

```
void  
mapS(const RealArray & r, RealArray & x, RealArray & xr, MappingParameters &  
params)
```

**Description:** Use the transformations defined by rowSpline0, rowSpline1, and rowSpline2 and the additional scaling mapping to compute the image(s) and/or the derivatives for the parameter point(s) defined by  $r$ .

#### 40.13 Examples

The command file **Overture/sampleMappings/aorticArch.cmd** generates the mappings for a model of the aortic arch shown in figure 28.

Figure 29 shows the grids generated by the SweepMapping for a model of a stadium.

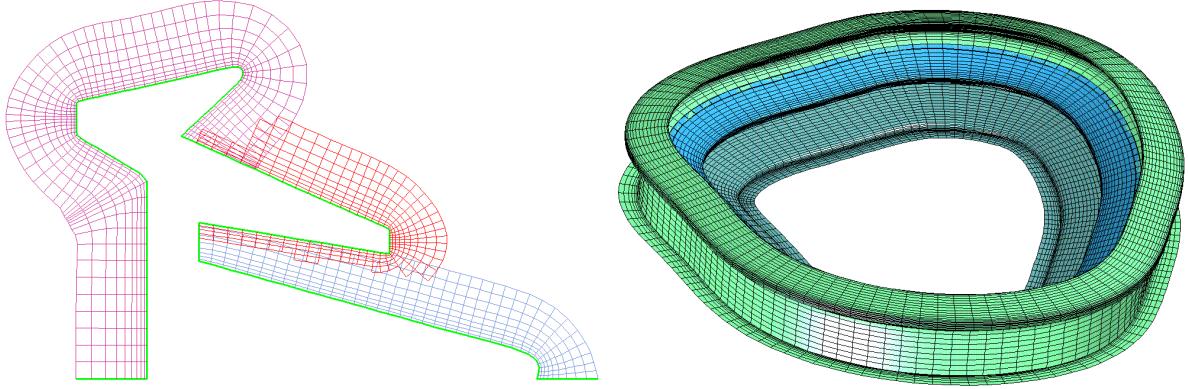


Figure 29: The SweepMapping is used to generate mappings for a stadium.

## 41 TFIMapping: Transfinite-Interpolation

Thanks to Brian Miller for help with this Mapping.

Use this Mapping to create a transfinite-interpolation mapping (also known as a Coon's patch). Transfinite-interpolation creates a patch (grid) in 2D (3D) using curves (surfaces) that define the boundaries. The quality of this algebraic grid is strongly dependent on the parameterizations of the boundary curves. // In its simplest form this mapping blends two given curves to create a grid in the region between the curves. Given the two curves  $\mathbf{c}_i(r_1)$  the linear interpolation formula is

$$\mathbf{x}(r_1, r_2) = \mathbf{c}_0(r_1)(1 - r_2) + \mathbf{c}_1(r_1)r_2$$

With an hermite interpolation it is possible to also specify the  $r_2$  derivative of the patch at the boundaries. The formula for hermite interpolation between two curves is

$$\mathbf{x}(r_1, r_2) = \mathbf{c}_0(r_1)\Phi_0(r_2) + \mathbf{c}_1(r_1)\Phi_1(r_2) + \dot{\mathbf{c}}_0(r_1)\Psi_0(r_2) + \dot{\mathbf{c}}_1(r_1)\Psi_1(r_2)$$

where the blending functions are given by

$$\begin{aligned} \Phi_0(r) &= (1 + 2r)(1 - r)^2 & \Phi_1(r) &= (3 - 2r)r^2 \\ \Psi_0(r) &= r(1 - r)^2 & \Psi_1(r) &= (r - 1)r^2 \end{aligned}$$

The patch will satisfy the conditions

$$\mathbf{x}(r_1, i) = \mathbf{c}_i(r_1) \quad \text{and} \quad \frac{\partial}{\partial r_2} \mathbf{x}(r_1, 0) = \dot{\mathbf{c}}_i(r_1) \quad i = 0, 1$$

The grid lines will be normal to the boundary provided that we choose the derivatives  $\dot{\mathbf{c}}_i$  to be proportional to the normal vectors. The scaling of this vector will determine the grid spacing near the boundary. We choose

$$\dot{\mathbf{c}}_i(r) = \|\mathbf{c}_1(r) - \mathbf{c}_0(r)\| \mathbf{n}_i(r)$$

where  $\mathbf{n}_i(r)$  is the unit normal vector and  $\|(a, b)\| = \sqrt{a^2 + b^2}$ .

The derivative of the hermite-interpolant involves the second derivative of the boundary curves and is given by

$$\frac{\partial}{\partial r_1} \mathbf{x}(r_1, r_2) = \frac{\partial}{\partial r_1} \mathbf{c}_0(r_1) \Phi_0(r_2) + \frac{\partial}{\partial r_1} \mathbf{c}(r_1) \Phi_1(r_2) + \frac{\partial}{\partial r_1} \dot{\mathbf{c}}_0(r_1) \Psi_0(r_2) + \frac{\partial}{\partial r_1} \dot{\mathbf{c}}_1(r_1) \Psi_1(r_2)$$

where

$$\frac{\partial}{\partial r_1} \dot{\mathbf{c}}_i = \frac{(\mathbf{c}_1 - \mathbf{c}_0) \cdot (\mathbf{c}_{1,r_1} - \mathbf{c}_{0,r_1}) \dot{\mathbf{c}}_i}{\|\mathbf{c}_1 - \mathbf{c}_0\|^2} - \frac{(\dot{\mathbf{c}}_i \cdot \mathbf{c}_{i,r_1}) \mathbf{c}_{i,r_1}}{\|\mathbf{c}_{i,r_1}\|^2}$$

More generally a transfinite-interpolation mapping can interpolate 4 curves in 2D/3D or up to 6 curves in 3D. Define

- $\mathbf{c}_{\text{left}}$  : side corresponding to  $r_1 = 0$ .
- $\mathbf{c}_{\text{right}}$  : side corresponding to  $r_1 = 1$ .
- $\mathbf{c}_{\text{bottom}}$  : side corresponding to  $r_2 = 0$ .
- $\mathbf{c}_{\text{top}}$  : side corresponding to  $r_2 = 1$ .
- $\mathbf{c}_{\text{back}}$  : side corresponding to  $r_3 = 0$ .
- $\mathbf{c}_{\text{Front}}$  : side corresponding to  $r_3 = 1$ .

The notation ‘left’, ‘right’, etc. comes from the fact that when a cube is plotted in the graphics window the left side is the face  $r_1 = 0$ , the right face is  $r_1 = 1$  etc.

In two dimensions, when 4 sides are specified, the formula for linear interpolation is

$$\begin{aligned} \mathbf{c}(r_1, r_2) &= \mathbf{c}_{\text{left}}(r_2)(1 - r_1) + \mathbf{c}_{\text{right}}(r_2)r_1 \\ &\quad + \mathbf{c}_{\text{bottom}}(r_1)(1 - r_2) + \mathbf{c}_{\text{top}}(r_1)r_2 \\ &\quad - \left\{ (1 - r_2)((1 - r_1)\mathbf{c}_{\text{left}}(0) + r_1\mathbf{c}_{\text{right}}(1)) + r_2((1 - r_1)\mathbf{c}_{\text{bottom}}(0) + r_1\mathbf{c}_{\text{top}}(1)) \right\}. \end{aligned}$$

The last line in this formula represents a correction, a bilinear function that passes through the four corners, that ensures the mapping,  $\mathbf{c}$ , matches the four boundary curves. In three-dimensions with 6 sides specified

$$\begin{aligned} \mathbf{c}(r_1, r_2, r_3) &= \mathbf{c}_{\text{left}}(r_2, r_3)(1 - r_1) + \mathbf{c}_{\text{right}}(r_2, r_3)r_1 \\ &\quad + \mathbf{c}_{\text{bottom}}(r_1, r_3)(1 - r_2) + \mathbf{c}_{\text{top}}(r_1, r_3)r_2 \\ &\quad + \mathbf{c}_{\text{back}}(r_1, r_2)(1 - r_3) + \mathbf{c}_{\text{Front}}(r_1, r_2)r_3 \\ &\quad - \left\{ (1 - r_2)((1 - r_1)\mathbf{c}_{\text{left}}(0, r_3) + r_1\mathbf{c}_{\text{right}}(0, r_3)) + r_2((1 - r_1)\mathbf{c}_{\text{bottom}}(0, r_3) + r_1\mathbf{c}_{\text{top}}(1, r_3)) \right. \\ &\quad + (1 - r_2)((1 - r_3)\mathbf{c}_{\text{bottom}}(r_1, 0) + r_3\mathbf{c}_{\text{top}}(r_1, 1)) + r_2((1 - r_3)\mathbf{c}_{\text{back}}(r_1, 1) + r_3\mathbf{c}_{\text{Front}}(r_1, 1)) \\ &\quad + (1 - r_3)((1 - r_1)\mathbf{c}_{\text{back}}(0, r_2) + r_1\mathbf{c}_{\text{Front}}(1, r_2)) + r_3((1 - r_1)\mathbf{c}_{\text{left}}(r_2, 1) + r_1\mathbf{c}_{\text{right}}(r_2, 1)) \Big\} \\ &\quad + \left[ (1 - r_3)[(1 - r_2)((1 - r_1)\mathbf{c}(0, 0, 0) + r_1\mathbf{c}(1, 0, 0)) + r_2((1 - r_1)\mathbf{c}(0, 1, 0) + r_1\mathbf{c}(1, 1, 0))] \right. \\ &\quad \left. + r_3[(1 - r_2)((1 - r_1)\mathbf{c}(0, 0, 1) + r_1\mathbf{c}(1, 0, 1)) + r_2((1 - r_1)\mathbf{c}(0, 1, 1) + r_1\mathbf{c}(1, 1, 1))] \right]. \end{aligned}$$

In the 3d case we must first subtract off corrections as in 2D (3 such corrections) and then add back a trilinear function that passes through the 8 vertices to ensure that all 6 sides are matched.

## 41.1 Compatibility conditions

The above TFI formulae will only give a reasonable grid if

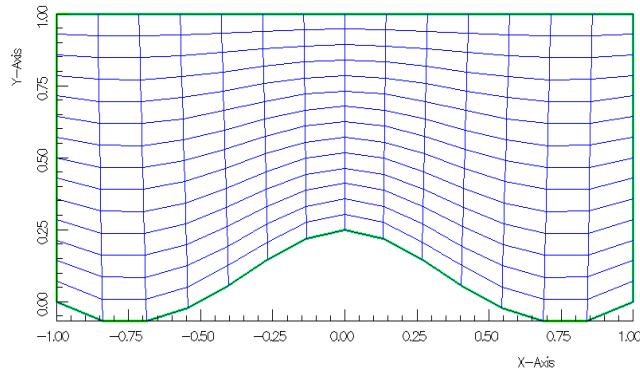
1. The curves that define the faces match at vertices (and edges in 3d).
2. The curves are parameterized in the ‘same direction’, otherwise the grids lines could cross.
3. Curves on opposite sides are parameterized in a similar way.

Even if all these conditions are met the grid lines may still cross if the boundary curves are strange enough.

## 41.2 Examples

### 41.2.1 2D linear TFI mapping with 2 sides specified

```
1  *
2  * Create a patch with transfinite interpolation
3  *
4  * create a line for the top boundary
5  line
6    number of dimensions
7      2
8    specify end points
9      -1. 1. 1. 1.
10 exit
11 * create a spline for the bottom boundary
12 spline
13   enter spline points
14     5
15     -1. 0.
16     -.5 0.
17     0. .25
18     .5 0
19     1. 0.
20 exit
21 * create a tfi patch
22 tfi
23   choose bottom curve
24     spline
25   choose top curve
26     line
27     mappingName
28     tfi
29   pause
30 exit
31 check
32 tfi
33
34
```



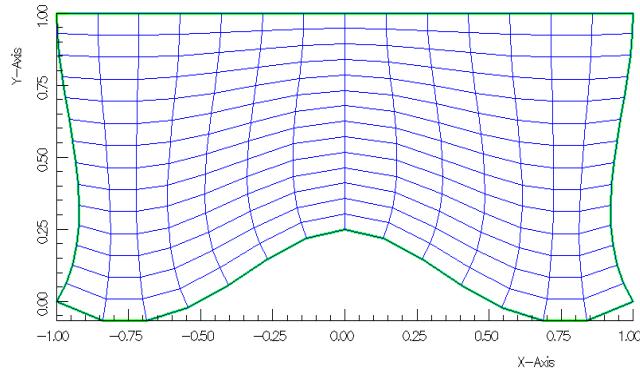
A grid created with linear trans-finite interpolation

#### 41.2.2 2D hermite TFI mapping with 2 sides specified

```

1  *
2  * Create a patch with transfinite interpolation
3  *
4  * create a line for the top boundary
5  line
6    number of dimensions
7    2
8    specify end points
9    -1. 1. 1. 1.
10 exit
11 * create a spline for the bottom boundary
12 spline
13   enter spline points
14   5
15   -1. 0.
16   -.5 0.
17   0. .25
18   .5 0
19   1. 0.
20 exit
21 * create a tfi patch
22 tfi
23   choose bottom
24     spline
25   choose top
26     line
27   hermite interpolation
28   mappingName
29     tfi
30   pause
31 exit
32 check
33   tfi
34

```



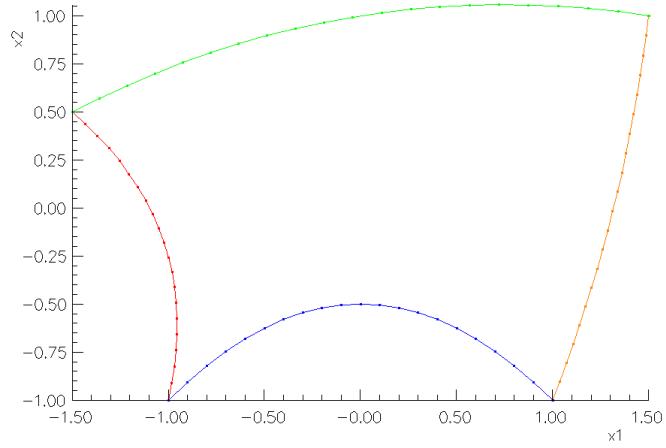
A grid created with hermite trans-finite interpolation; grid lines are normal to the bottom and top boundaries

### 41.2.3 2D linear TFI mapping with 4 sides specified

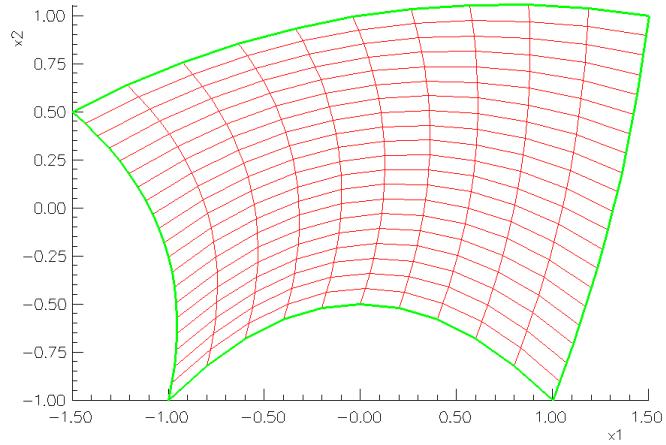
```

1  spline
2    enter spline points
3      3
4      -1.0 -1.0
5      0.0 -.5
6      1.0 -1.0
7  lines
8    21
9  mappingName
10   bottomSpline
11  exit
12 spline
13   enter spline points
14   3
15   -1.5 .5
16   0.0 1.0
17   1.5 1.0
18 lines
19   21
20  mappingName
21   topSpline
22  exit
23 spline
24   enter spline points
25   3
26   -1.0 -1.0
27   -1.0 -.25
28   -1.5 .5
29 lines
30   21
31  mappingName
32   leftSpline
33  exit
34 spline
35   enter spline points
36   3
37   1.0 -1.0
38   1.25 -.25
39   1.5 1.0
40 lines
41   21
42  mappingName
43   rightSpline
44  exit
45 tfi
46  mappingName
47   tfi2d4
48  choose bottom
49   bottomSpline
50  choose top
51   topSpline
52  choose left
53   leftSpline
54  choose right
55   rightSpline
56 pause
57  exit
58 check
59  tfi2d4

```



1



1

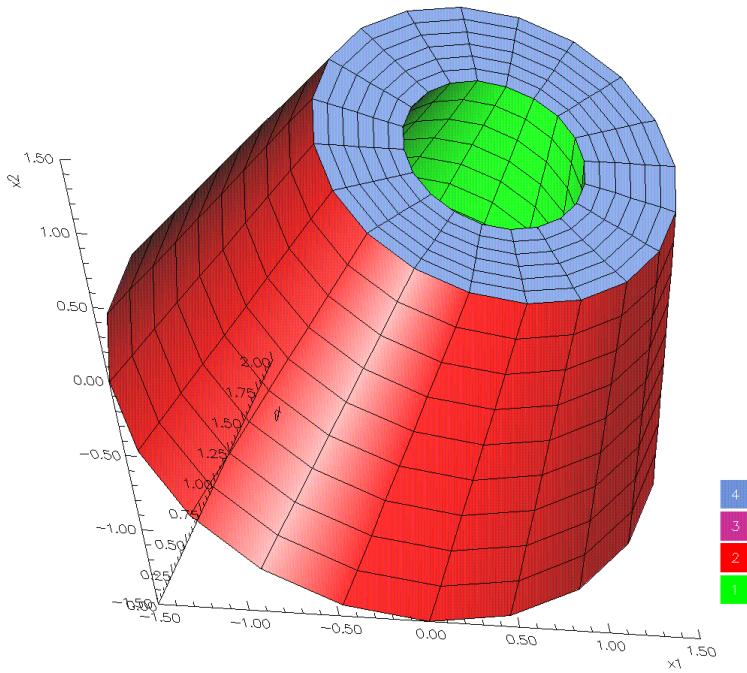
A grid created with linear trans-finite interpolation, all four sides are specified. The top figure shows the 4 boundary curves before interpolation.

#### 41.2.4 3D linear TFI mapping with 2 sides specified

```

1  *
2  * 3D TFI Mapping between two Annulus map]          TFIMapping
3  *
4  * make the annulus for the top
5  Annulus
6    make 3d (toggle)
7      2.
8    mappingName
9      top-annulus
10   exit
11 * make the annulus for the bottom
12 Annulus
13   outer radius
14     1.5
15   inner radius
16     1.
17   make 3d (toggle)
18     0.
19   mappingName
20     bottom-annulus
21 exit
22 tfi
23   choose back curve
24     bottom-annulus
25   choose front curve
26     top-annulus
27   boundary conditions
28     -1 -1 1 2 3 4
29
30

```



A grid created with linear trans-finite interpolation between two Annulus mappings.

#### 41.3 setSides

```

int
setSides(Mapping *left =NULL,
          Mapping *right =NULL,
          Mapping *bottom =NULL,
          Mapping *top =NULL,
          Mapping *front =NULL,
          Mapping *back =NULL)

```

**Purpose:** Build a TFIMapping and supply curves that define the boundaries. Specify 0, 2, 4 or 6 curves.

The Trans-Finite-Interpolation (TFI) Mapping (also known as a Coon's patch) will interpolate between the boundary curves to define a mapping in the space between. See the documentation for further details.

**left, right (input):** curves for  $r_1 = 0$  and  $r_1 = 1$ .

**bottom, top (input):** curves for  $r_2 = 0$  and  $r_2 = 1$ .

**front, back (input):** curves for  $r_3 = 0$  and  $r_3 = 1$  (3D only).

#### 41.4 flipper

```

int
flipper()

```

**Purpose:** Try to flip the curve parameterizations to make the mapping non-singular.

**Notes:** Fix up a TFIMapping that turns inside out because the bounding curves are not parameterized in compatible ways.

#### 41.5 map

```
void  
map( const realArray & r_, realArray & x_, realArray & xr_, MappingParameters &  
params )
```

**Purpose:** Evaluate the TFI and/or derivatives.

#### 41.6 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively create and/or change the TFI mapping.

**mapInfo (input):** Holds a graphics interface to use.

## 42 TrimmedMapping: define a trimmed surface in 3D

### 42.1 Description

A trimmed surface consists of a standard (i.e. logically rectangular) mapping ("surface") which has regions removed from it. The portions removed are defined by curves in the parameter space of the mapping. The IN-ACTIVE part of the trimmed surface is any point that is outside the outer boundary or inside any of the inner curves. Thus one inner curve cannot be inside another inner curve. None of the trimming curves are allowed to intersect each other.

Here is how you should evaluate a trimmed mapping (accessing the mask array to indicate whether the point is inside or outside):

```
TrimmedMapping trim;
... assign the TrimmedMapping somehow ...
RealArray r(10,2), x(10,3), xr(10,3,2);
... assign r ....
MappingParameters params; // we need to pass this option argument to "map"
trim.map(r,x,xr,params);
IntegerArray & mask = params.mask; // mask(i) = 0 if point is outside, =1 if inside
for( int i=0; i<9; i++; )
{
    if( mask(i)==0 )
        // point is outside, x(i,0:2) are the coordinates of the untrimmed surface at r(i,0:1)
    else
        // point is inside, x(i,0:2) are the coordinates of the trimmed surface at r(i,0:1)
}
```

trimmed mapping

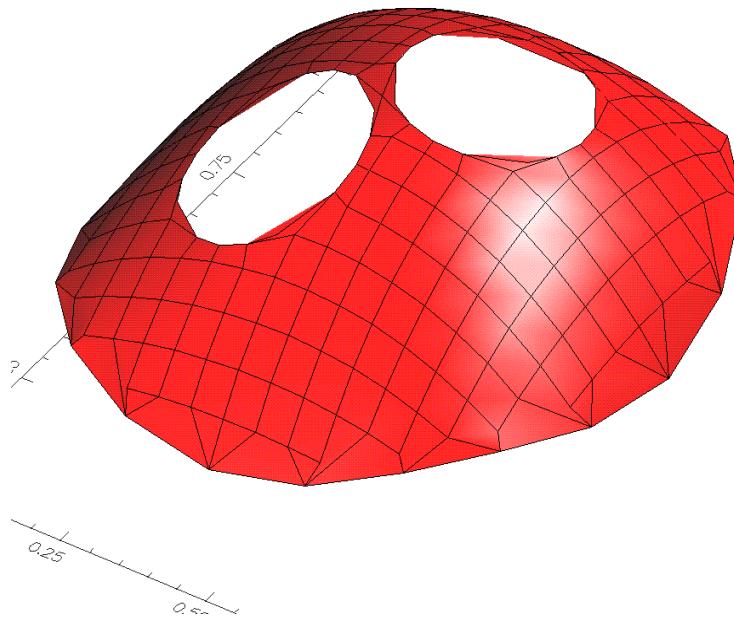


Figure 30: A trimmed mapping with an outer trimming curve and 2 inner trimming curves. To plot the mapping we project points which are just outside the trimmed region onto the boundary

TrimmedMapping from the data base

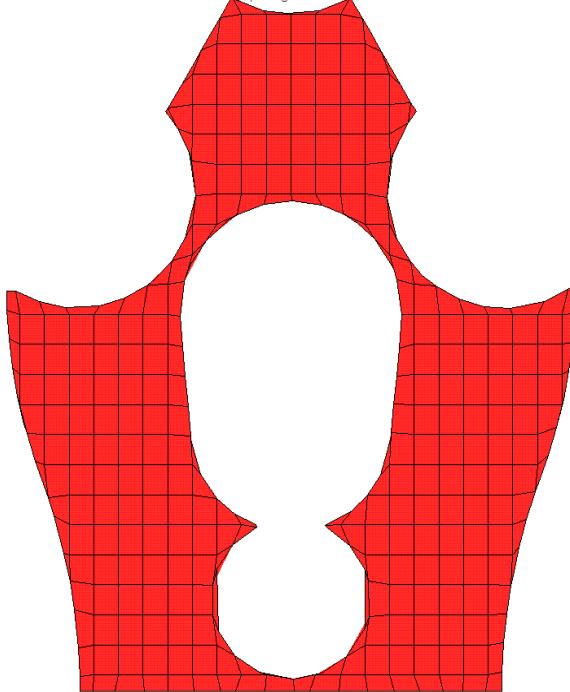


Figure 31: A trimmed mapping with an outer trimming curve and 1 inner trimming curve

## 42.2 Constructor

**TrimmedMapping()**

**Purpose:** Default Constructor

## 42.3 Constructor

```
TrimmedMapping(Mapping & surface_,  
               Mapping *outerCurve_ =NULL,  
               const int & numberOfInnerCurves_ =0,  
               Mapping **innerCurve_ =NULL)
```

**Purpose:** Create a trimmed surface

**surface (input)** : surface to be trimmed

**outerCurve (input)** : curve defining the outer boundary, if NULL then the outer boundary is the boundary of surface

**numberOfInnerCurves (input)** : number of closed curves in the interior that trim the surface

**General Notes:** In order to evaluate a trimmed mapping we need to decide whether we are inside or outside. To make this determination faster, we divide the domain space ( $r$ ) with a quadtree mesh: the domain is broken into 4 squares, each of which is subdivided into 4 again, recursively as needed. Each square is marked whether it lies inside the domain, outside, or partly in and partly out. It is quick to traverse the quadtree to find which square a given point is in. If the square is inside or outside we are done. If it is mixed, we usually have to check the point against only one curve to determine insideness.

## 42.4 Constructor

```
TrimmedMapping(Mapping & surface_,  
               const int & numberOfTrimCurves_ =0,  
               Mapping **trimCurves_ =NULL)
```

**Purpose:** Create a trimmed surface

**surface (input)** : surface to be trimmed

**numberOfTrimCurves\_ (input)** : number of closed curves that trim the surface

**trimCurves\_ (input)** : the trimming curves

**General Notes:** In order to evaluate a trimmed mapping we need to decide whether we are inside or outside. To make this determination faster, we divide the domain space ( $r$ ) with a quadtree mesh: the domain is broken into 4 squares, each of which is subdivided into 4 again, recursively as needed. Each square is marked whether it lies inside the domain, outside, or partly in and partly out. It is quick to traverse the quadtree to find which square a given point is in. If the square is inside or outside we are done. If it is mixed, we usually have to check the point against only one curve to determine insideness.

## 42.5 getTriangulationParameters

**int**

```
getTriangulationParameters( real & area, real & minAngle, real & elementDensity ) const
```

**Purpose:** Return the values of parameters that control the triangulation.

## 42.6 setMaxAreaForTriangulation

**int**

```
setMaxAreaForTriangulation( real area =.1)
```

**Purpose:** Set the maximum area (approximately) for triangles.

**area (input)** : max area for triangles (approx). 0=use default

## 42.7 setMinAngleForTriangulation

**int**

```
setMinAngleForTriangulation( real minAngle =20.)
```

**Purpose:** Set the minimum angle for the triangulation of the trimmed mapping.

**minAngle (input)** : choosing a smaller value will result in fewer triangles. A negative value means use the default. A value of zero will give the fewest triangles.

## 42.8 setElementDensityTolerance

**int**

```
setElementDensityToleranceForTriangulation( real elementDensity =.05)
```

**Purpose:** Specify the element density tolerance

**elementDensity (input)** : choosing a smaller value will result in more triangles. The number of grid points on an edge is based on the ratio of the curvature to the elementDensity. A negative value means use the default.

## 42.9 setCurves

```
int  
setCurves(Mapping & surface_,  
          const int & numberOfTrimCurves_ =0,  
          Mapping **trimCurves_ =NULL)
```

**Purpose:** Specify the surface and trimming curves.

**surface (input) :** surface to be trimmed

**numberOfInnerCurves (input) :** number of closed curves that trim the surface

**trimCurves\_ (input) :** the oriented trim curves that trim the surface

## 42.10 setUnInitialized

```
void  
setUnInitialized()
```

**Description:** Indicate that this Mapping is not up to date. This will destroy the triangulation used to plot it.

## 42.11 initializeTrimCurves

```
void  
initializeTrimCurves()
```

**Access:** protected

**Description:** Compute trimming curve arclengths, areas, orientation, dRmin, dSmin.

## 42.12 addCurve

```
bool  
addTrimCurve(Mapping *newCurve)
```

**Purpose:** Add a trimming curve to the surface

**newCurve (input) :** the new trim curve

**returns :** true if there were no problems with the trimming curve, false otherwise

## 42.13 deleteTrimCurve

```
bool  
deleteTrimCurve(int curveToDelete)
```

**Purpose:** delete a specific trimming curve from the surface. Note : if the curve to be delete is the last counterclockwise curve, then a default trimming curve is built consisting of the untrimmed surface's boundary.

**curveToDelete :** index of the trim curve to be removed

**returns :** returns false if the last counterclockwise trim curve was removed resulting in the creation of a default outer curve.

#### **42.14 deleteTrimCurve**

**bool**  
**deleteTrimCurve( int numberOfCurvesToDelete, int \*curvesToDelete)**

**Purpose:** delete multiple trimming curves from the surface. Note : if the curve to be delete is the last counterclockwise curve, then a default trimming curve is built consisting of the untrimmed surface's boundary.

**numberOfCurvesToDelete (input):** the length of the array curvesToDelete

**curvesToDelete (input):** an array containing a list of the curves to be deleted

**returns :** returns false if the last counterclockwise trim curve was removed resulting in the creation of a default outer curve.

#### **42.15 undoDelete**

**bool**  
**undoLastDelete()**

**Purpose:** undo the last call to deleteTrimCurve

**returns :** true if successfull, false otherwise

#### **42.16 initializeQuadTree (protected)**

**void**  
**initializeQuadTree(bool buildQuadTree =true)**

**Description:** Initialize things needed by the quad-tree search and optionally build the quad-tree.

- initialize the bounding boxes for each of the trimming curves
- Make the array rCurve[c] point to the "grid" for each trimming curve
- determine the rBound array which holds the bounds on the unit square in which conatins the trimmed surface.

#### **42.17 getOuterCurve**

**Mapping\***  
**getOuterCurve()**

**Description:** Return a pointer to the outer trimming curve.

#### **42.18 getInnerCurve**

**Mapping\***  
**getInnerCurve(const int & curveNumber)**

**Description:** Return a pointer to the inner trimming curve number curveNumber.

**curveNumber (input) :** number of the trimming curve, between 0 and `getNumberOfInnerCurves()`.  
Return 0 if the curveNumber is invalid.

## 42.19 curveGoesThrough

```
bool  
curveGoesThrough(const TMquad& square, const int& c, int& segstart, int& segstop )  
const
```

**Access:** public.

**Description:** Determine whether the polygonal curve c goes through the square quadtree node "square". If so, return true. One may specify starting and stopping segment numbers of the curve. These will be reset to indicate the curve segments which pass through the square. 0 and -1 mean to use all segments.

## 42.20 insideOrOutside

```
int  
insideOrOutside( const realArray & rr_, const int & c )
```

**Access:** protected.

**Purpose:** find if the point rr lies inside the curve c (actually inside the polygon defined by rCurve(0:n,0:1)) and return the distance to that curve.

**Method:** Use the routine from the mapping inverse to count how many times a vertical ray traced above the point crosses the polygon. NOTE: points exactly on the boundary are "outside" by this definition

**rr (input):** point the the parameter space of the untrimmed surface.

**c (input):** curve number.

**distance (output):**

**Return values:** +1 if the point is inside the outerCurve (c==1) or oustide the inner curve (c<1)  
-1 otherwise

## 42.21 insideOrOutside

```
int  
insideOrOutside( const realArray & rr, const int & c, realArray & distance )
```

**Access:** protected.

**Purpose:** find if the point rr lies inside the curve c (actually inside the polygon defined by rCurve(0:n,0:1)) and return the distance to that curve. This routine calls the `insideOrOutside( const realArray & rr, const int & c )` function.

**rr (input):** point the the parameter space of the untrimmed surface.

**c (input):** curve number.

**distance (output):**

**Return values:** +1 if the point is inside the outerCurve (c==1) or oustide the inner curve (c<1)  
-1 otherwise

## 42.22 findClosestCurve

```
int  
findClosestCurve(const realArray & x_,  
                 intArray & cMin_,  
                 realArray & rC_,  
                 realArray & xC_,  
                 realArray & dist_,  
                 const int & approximate =TRUE)
```

/ N.B. I HAVE CHANGED THIS: Some changes in the specification are to match the actual code, and some changes in the code are to match the pre-existing specification. But the code has changed in that  $cMin=-2$  has a new, special meaning. (jfp 0399)

**Access:** protected.

**Description:** Find the closest curve to a point and/or determine if the point is inside the curve.

**x(R,.) (input) :** points in the untrimmed surfaces parameter space

**cMin(R) (input/output) :** If  $cMin(base) \geq 0$ , then each  $cMin(i)$  is the number of the curve to be used for  $x(i,.)$ . If  $cMin(base) < 0$ , then all curves will be checked, and on output  $cMin(i)$  will be the number of the curve nearest  $x(i,.)$  (This has been implemented only for the case where  $x$  is one point.) When  $cMin(base) \geq 0$ ,  $cMin(i) == -2$  means to skip computing the projection of  $x(i,.)$ . When  $cMin(base) == -2$ , nothing is computed.

**xC(R,.) (output) :** closest point on closest curve

**dist (output) :**  $dist(R) =$  minimum distance

**approximate (input) :** if TRUE only determine an approximation to the closest closest point on the closest curve (based on the nearest grid point on the polygonal representation of the curve).

## 42.23 findDistanceToACurve

```
int  
findDistanceToACurve(const realArray & x,  
                      IntegerArray & cMin,  
                      realArray & dist,  
                      const real & delta )
```

**Access:** protected.

**Description:** Find the approximate distance to a curve. (approximate if the distance  $\leq \delta$  )

**x(R,.) (input) :** points

**cMin(R) (input/output) :** if  $\geq 0$  on input then use this curve, on output it is the number of closest curve

**dist (output) :**  $dist(R) =$  approximate distance

## 42.24 map

```
void  
map( const realArray & r, realArray & x, realArray & xr, MappingParameters & params )
```

**Purpose:** Evaluate the Trimmed and/or derivatives.

**NOTE:** In order to evaluate a trimmed surface you MUST provided a MappingParameters argument.  
Otherwise only the untrimmed mapping will be defined.

**Notes:** (1) The array params.mask(I) is returned with the values -1=outside, 0=inside

(2) if point i is outside the grid but near the trimmed boundary the array distanceToBoundary(i) is set to be the distance (in parameter space) of the point r(i,.) to the nearest trimming curve. The the point is far from the boundary, distance(i) is set to a large value.

## 42.25 map

```
void  
mapGrid(const realArray & r,  
        realArray & x,  
        realArray & xr,  
        MappingParameters & params =Overture::nullMappingParameters())
```

**Purpose:** Map grid points and project grid points that cross a trimming curve onto the trimming curve.  
This routine is called by the plotting routine so that trimmed curves are properly plotted.

## 42.26 hasTriangulation

```
bool  
hasTriangulation() const
```

**Description:** return true if there is a triangulation computed

## 42.27 getTriangulation

```
UnstructuredMapping &  
getTriangulation()
```

**Description:** Return the triangulation for the TrimmedMapping (compute it first if necessary).

## 42.28 triangulate

```
void  
triangulate( )
```

**Description:** Triangulate a TrimmedMapping.

## 42.29 update

```
int  
update( MappingInformation & mapInfo )
```

**Purpose:** Interactively create and/or change the Trimmed mapping.

**mapInfo (input):** Holds a graphics interface to use.

#### 42.30 reportTrimCurveInfo

aString  
reportTrimCurveInfo(Mapping \*c, bool & curveok)

**Purpose:** return a string describing the state of a trim curve

c (**input**) : the curve in question

#### 42.31 reportTrimmingInfo

aString  
reportTrimmingInfo()

**Purpose:** return a string describing the state of the trimming

#### 42.32 editTrimCurve

int  
editTrimCurve( Mapping &trimCurve, MappingInformation & mapInfo )

**Purpose:** Interactively edit a trim curve

**mapInfo (input):** Holds a graphics interface to use.

#### 42.33 editNurbsTrimCurve

int  
editNurbsTrimCurve( NurbsMapping &trimCurve, MappingInformation & mapInfo )

**Purpose:** Interactively edit a nurbs trim curve

**mapInfo (input):** Holds a graphics interface to use.

## 43 UnstructuredMapping

The `UnstructuredMapping` class encapsulates the connectivity for an unstructured mesh. Currently the class supports the “Finite Element Zoo” collection of element types. This zoo consists of quadrilaterals and triangles in surface meshes and hexahedra, triangle prisms, pyramids and tetrahedra in volume meshes.

### 43.1 Connectivity Iterator Interface

### 43.2 Introduction

Connectivity in the `UnstructuredMapping` is based on the set of “entities” specified in `UnstructuredMapping ::EntityTypeEnum`. “Vertex”, “Edge”, “Face”, and “Region” refer to entities in ascending topological dimension, with “Vertex” equivalent to 0 and “Region” equivalent to 3:

```
enum EntityTypeEnum
{
    Invalid=-1,
    Vertex=0,
    Edge,
    Face,
    Region,
    Mesh, // kkc put this here to enable "Mesh" tagging...
    NumberOfEntityTypes
};
```

This approach will be familiar to users of the TSTT mesh query interface. Entities themselves are defined by their topological dimension (Type) and the specific vertices that they contain. For example, no two edges have the same vertices. Each entity has an orientation defined by their vertices and “Element” type as specified in `UnstructuredMapping ::ElementTypeEnum`:

```
enum ElementType
{
    triangle,
    quadrilateral,
    tetrahedron,
    pyramid,
    triPrism,
    septahedron,
    hexahedron,
    other,
    boundary,
    NumberOfElementTypes
};
```

`ElementTypeEnum` is analogous to the `EntityTopology` enum from the TSTT interface and specifies the “shape” of a given entity (triangle, quad, etc...). Iterating through the entities and their adjacencies is managed most effectively (though not exclusively) through the `UnstructuredMappingIterator` and `UnstructuredMappingAdjacencyIterator` classes. The Iterator classes provide iterations through entities of a specific type (eg all the Regions) while the AdjacencyIterators provide access to the entities adjacent to a specific entities. A tagging mechanism has also been added, again following the TSTT approach, in order to allow a very general method for placing information on entities and grouping them together into sets. Several additions to the methods and data in the `UnstructuredMapping` class have been created to facilitate the construction and use of the new connectivity and iterators.

Prior to discussing the details of the interface and changes to the `UnstructuredMapping` class, we demonstrate the basic use of the interface with two examples. In the first example we read an `UnstructuredMapping` from a file and then iterate through all the “Faces” in the mesh:

```
UnstructuredMapping umap;
umap.get(fileName);

// loop through all the Faces in the mesh
UnstructuredMappingIterator face;
for ( face = umap.begin(UnstructuredMapping::Face);
      face!=umap.end(UnstructuredMapping::Face);
      face++ )
    cout<<"Here is a Face with index "<< *face <<endl;
```

The declaration and `get` method should be familiar to users of the `UnstructuredMapping`. Looping through all the faces in the `UnstructuredMapping`, however, is performed using the new `UnstructuredMappingIterator` iterator class. `UnstructuredMappingIterator`’s use mirrors that of STL containers and iterators. `UnstructuredMapping`’s `begin` and `end` methods require an argument specifying the type of entity requested (from `EntityTypeEnum`). The dereference operator of the iterator returns an integer index which may be used as part of an identification mechanism or access into another array. In order to obtain adjacency information (such as the vertices attached to the a specific face) we need to use an `UnstructuredMappingAdjacencyIterator`:

```
UnstructuredMappingIterator face;
for ( face = umap.begin(UnstructuredMapping::Face);
      face != umap.end(UnstructuredMapping::Face);
      face++ )
{
  cout<<"Face "<<*face<<" has vertices : ";

  UnstructuredMappingAdjacencyIterator faceVert;
  for ( faceVert = umap.adjacency_begin(face, UnstructuredMapping::Vertex);
        faceVert != umap.adjacency_end(face, UnstructuredMapping::Vertex);
        faceVert++ )
    cout<<*faceVert<< " ";

  cout<<endl;
}
```

The `UnstructuredMapping`’s `adjacency_begin` and `adjacency_end` methods are used in the same fashion as the `UnstructuredMappingIterator` versions. They require an `UnstructuredMappingIterator` as the first argument in order to specify the “from” entity and a second argument to specify the type of the “to”, or adjacent, entities. Variations on these two iterations will form the bulk of the examples and are expected to provide the necessary functionality to build unstructured mesh operators and manipulations.

### 43.3 Additions to the `UnstructuredMapping` class

#### 43.3.1 Generic size and data access methods

- `const intArray & getEntities(EntityTypeEnum);`  
return an array of the vertices in all the entities of the type requested by the argument

- `inline int size( EntityTypeEnum t ) const;`  
returns the number of entities of type `t` available in the mesh.
- `inline int capacity( EntityTypeEnum t ) const;`  
return the total storage available for entities of type `t`.
- `int reserve( EntityTypeEnum, int) const;`  
reserve a specified amount of space (increase/decrease the capacity) for entities of a specified type

### 43.3.2 Iterator related methods

- `inline UnstructuredMappingIterator begin(EntityTypeEnum entityType_, bool skipGhostEntities) const;`  
returns an iterator pointing to the beginning of the list of entities of type `entityType_`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the specified entity does not exist the mapping will attempt to build it!
- `inline UnstructuredMappingIterator end(EntityTypeEnum entityType_, bool skipGhostEntities) const;`  
returns an iterator pointing to the end of the list of entities of type `entityType_`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested entity does not exist the mapping will try to build it!
- `inline UnstructuredMappingAdjacencyIterator adjacency_begin(UnstructuredMappingIterator from, EntityTypeEnum to, bool skipGhostEntities) const;`  
returns an iterator pointing to the beginning of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested adjacency information does not exist then the mapping will attempt to build it.
- `inline UnstructuredMappingAdjacencyIterator adjacency_end(UnstructuredMappingIterator from, EntityTypeEnum to, bool skipGhostEntities) const;`  
returns an iterator pointing to the end of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested adjacency information does not exist then the mapping will attempt to build it.
- `inline UnstructuredMappingAdjacencyIterator adjacency_begin(UnstructuredMappingAdjacencyIterator from, EntityTypeEnum to, bool skipGhostEntities) const;`  
returns an iterator pointing to the beginning of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included.
- `inline UnstructuredMappingAdjacencyIterator adjacency_end(UnstructuredMappingAdjacencyIterator from, EntityTypeEnum to, bool skipGhostEntities) const;`

returns an iterator pointing to the end of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included.

#### 43.3.3 Entity and connectivity manipulations

- `inline int maxVerticesInEntity(EntityTypeEnum type);`  
returns the maximum number of vertices in a given entity type.
- `inline ElementType computeElementType(EntityTypeEnum type, int e);`  
computes the element type as given in the `ElementType` enum for a `type` entity with index `e`.
- `inline int numberOfVertices(EntityTypeEnum, int);`
- `bool entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices);`  
two entities are equivalent if their vertices are the same. This method returns true if the `type` entity with index `e` is equivalent to an entity specified by `vertices`.
- `void setAsGhost(EntityTypeEnum type, int entity);`  
toggles the `type` entity at index `entity` to be a ghost. This also adds a tag marking the entity as a ghost through the tagging mechanism. The entity is added to the list of entities with the same ghost tag.
- `bool specifyVertices(const realArray &verts);`  
provide a list of vertices to be copied into the mapping. This replaces the “node” array from the old interface. returns true if successful.
- `bool buildEntity(EntityTypeEnum type, bool rebuild, bool keepDownward, bool keepUpward);`  
Build the connectivity data for entities of type `type`. `rebuild` is an optional argument that tells the method to destroy existing connectivity if present; the default is false. `keepDownward` and `keepUpward` are optional arguments that instruct the mapping to keep downward or upward adjacencies built during the construction of the entities; both are true by default.
- `bool specifyEntity(const EntityTypeEnum type, const intArray &entity);`  
specify the entities of a type given by `type` by providing an array of the vertices in each entity. The `entity` array is dimensioned by the number of entities and the maximum number of vertices in that entity type. Each entity in the array is specified by a list of vertex indices terminated by a `-1` or by the use of all the available vertices for that entity type.
- `bool buildConnectivity(EntityTypeEnum from, EntityTypeEnum to, bool rebuild);`  
build the connectivity (adjacency) information from entities of type `from` to entities of type `to`. The optional argument `rebuild` forces the mapping to destroy any previous data if set to true; the default is false.
- `bool specifyConnectivity(const EntityTypeEnum from, const EntityTypeEnum to,  
const intArray &index, const char *orientation,  
const intArray &offset);`

allows the user to specify the adjacencies between `from` and `to` entities. The `index` array contains a list of `to` entities for each `from` entity in compressed form. The beginning of the list of `to` entities for each `from` is specified by the `offset` array.

- `inline bool connectivityExists(EntityTypeEnum from, EntityTypeEnum to) const;`  
returns true if the adjacency information between `from` and `to` has been built already.
- `void deleteConnectivity(EntityTypeEnum from, EntityTypeEnum to);`  
destroys the data associated with the connectivity between `from` and `to` entities.
- `void deleteConnectivity(EntityTypeEnum type);`  
destroys ALL the connectivity information relating to entities of type `type`.
- `void deleteConnectivity();`  
destroys ALL the connectivity information in the mesh.

## 43.4 Iterator Classes

There are two iterator classes available : `UnstructuredMappingIterator` , and `UnstructuredMappingAdjacencyIterator` . The former provides iteration through the entities of a particular type and the latter allows the iteration through the entities adjacent to a particular entity. Both skip holes in the entity data structures (unused locations in the arrays) and both optionally skip “ghost” entities by providing an argument to the iterator. Comparisons are also possible between the iterators.

### 43.4.1 Common methods and operators

- `void operator++(int);`  
increments the iterator to the next entity.
- `int operator *() const {return e;}`  
returns an unique (amongst the entity type of the iterator) index for the current entity.
- `bool operator==(const UnstructuredMappingIterator & iter) const;`
- `bool operator==(const UnstructuredMappingAdjacencyIterator & iter) const;`  
return true if two iterators point to the same entity.
- `bool operator!=(const UnstructuredMappingIterator & iter) const`
- `bool operator!=(const UnstructuredMappingAdjacencyIterator & iter) const`  
return true if two iterators point to different entities.

### 43.4.2 UnstructuredMappingIterator constructor

The `UnstructuredMappingIterator` is typically initialized by the use of a `UnstructuredMapping` ’s begin or end method. While not intended for common use by the user, the constructor for this class is:

- `UnstructuredMappingIterator(const UnstructuredMapping & uns,`  
`UnstructuredMapping::EntityTypeEnum entityType_ ,`  
`int position, bool skipGhostEntities );`

The first argument is the `UnstructuredMapping` containing the data to iterate through. `entityType_` designates the the type, or topological dimension, of the entities for iteration. The `position` argument is 0 if an iterator pointing to the beginning of the list is requested and 1 for the end. Finally, if `skipGhostEntities` is true, ghost entities will be skipped during the iteration.

#### 43.4.3 `UnstructuredMappingAdjacencyIterator` constructor

As with the `UnstructuredMappingIterator`, the `UnstructuredMappingAdjacencyIterator` constructor is intended to be used by methods in the `UnstructuredMapping` class for beginning, ending and querying iterators:

- `UnstructuredMappingAdjacencyIterator(const UnstructuredMapping & uns,  
                                  UnstructuredMapping::EntityTypeEnum from,  
                          int adjTo,  
                          UnstructuredMapping::EntityTypeEnum to,  
                          int position, bool skipGhostEntities_ = false );`

The first argument is the `UnstructuredMapping` containing the data to iterate through. `from` is an iterator (either a `UnstructuredMappingIterator` or `UnstructuredMappingAdjacencyIterator`) pointing to the entity around which the iteration will take place. The `EntityTypeEnum` variable `to` specifies the desired topological dimension of the adjacent entities. `position` argument is 0 if an iterator pointing to the beginning of the list is requested and 1 for the end. Finally, if `skipGhostEntities` is true, ghost entities will be skipped during the iteration.

#### 43.4.4 `UnstructuredMappingAdjacencyIterator` orientation

Each entity is given an orientation when it is constructed by the `UnstructuredMapping`. For example, the first vertex index in each edge is always the one with the lowest index value. Each entity maintains information about the orientation of adjacent (upward or downward adjacencies) entities relative to its own definition. For example, a “Face” adjacent to two “Regions” will have its vertices defined counterclockwise when viewed from outside one (the first in the upward adjacency) of the regions. This is by definition the adjacency with a positive orientation. When viewed relative to and from outside the second “Region”, the “Face” will have a clockwise, or negative, orientation. (MAYBE ADD A FIGURE TO DESCRIBE THIS?). In the downward adjacency from the first region to the face the adjacency is positive (the face is oriented correctly relative the the first region) and negative to the second (it is reversed relative to the second). In the upward adjacency, the first “Region” has a positive orientation while the second has negative. The orientation information is provided in the `UnstructuredMappingAdjacencyIterator` by the following method:

- `int orientation() const;`

This method returns 1 for positive orientation and -1 for negative. Use of the adjacency orientation will be demonstrated in the Examples section.

### 43.5 The tagging interface

Often it is convenient to tag an entity or group of entities with a piece of data for later use. Common examples would include tagging boundary condition information on boundary entities and adding material property information to groups of entities. A new interface has been created to support these features and consists of the typedefs `entity_tag_iterator`, and `tag_entity_iterator` and the following new methods:

- `EntityTag &`  
`addTag( const EntityTypeEnum entityType, const int entityIndex,`  
`const std::string tagName,`  
`const void *tagData, const bool copyTag, const int tagSize );`

adds a tag onto an entity of type `entityType` at index `entityIndex`. The “name” of the tag, used later for lookups of the tag, is specified by the string `tagName`. An optional argument `tagData` is a void pointer, possibly pointing to user defined data. User defined data can be managed by the tagging system by forcing deep copies of the `tagData` by setting `copyTag` to true. If `copyTag` is true and `tagData` is a pointer to user allocated data, `tagSize` must specify the size of the tag instance.

- `int`  
`deleteTag( const EntityTypeEnum entityType, const int entityIndex,`  
`const EntityTag &tagToDelete );`

deletes the tag associated with an entity of type `entityType` at index `entityIndex` and matching the name of `tagToDelete`.

- `int`  
`deleteTag( const EntityTypeEnum entityType, const int entityIndex,`  
`const std::string tagToDelete );`

deletes the tag associated with an entity of type `entityType` at index `entityIndex` and matching the string `tagToDelete`.

- `bool`  
`hasTag( const EntityTypeEnum entityType, const int entityIndex,`  
`const std::string tag );`

returns true if an entity has a tag with the name `tag`.

- `EntityTag &`  
`getTag( const EntityTypeEnum entityType,`  
`const int entityIndex, const std::string tagName);`

returns a reference to the instance of `EntityTag` associated with a particular entity and tag name.

- `void *`  
`getTagData( const EntityTypeEnum entityType, const int entityIndex,`  
`const std::string tag );`

returns a pointer to that data stored by a tag corresponding to a particular name on a given entity defined by `entityType` and `entityIndex`.

- `int`  
`setTag( const EntityTypeEnum entityType, const int entityIndex,`  
`const EntityTag & newTag );`

copies the data in `newTag` to the matching tag in the entity defined by `entityType` and `entityIndex`.

- `int setTagData( const EntityTypeEnum entityType, const int entityIndex,`  
`const std::string tagName,`  
`const void *data, const bool copyData=false, const int tagSize=0 );`

sets the data in a tag on the specified entity. User defined data can be managed by the tagging system by forcing deep copies of the `tagData` by setting `copyTag` to true. If `copyTag` is true and `tagData` is a pointer to user allocated data, `tagSize` must specify the size of the tag instance.

- `void maintainTagToEntityMap( bool v );`  
if `v` is true, the mapping creates and maintains the list of entities associated with each tag. If false, the mapping destroys this information and does not maintain it until reset to true.
- `bool maintainsTagToEntityMap() const;`  
returns true if the mapping maintains the list of entities with associated with each tag;
- `inline entity_tag_iterator entity_tag_begin(EntityTypeEnum et, int index);`  
returns an iterator to the beginning of the tags associated with an entity defined by `et` and `index`.
- `inline entity_tag_iterator entity_tag_end(EntityTypeEnum et, int index);`  
returns an iterator to the end of the tags associated with an entity defined by `et` and `index`.
- `inline tag_entity_iterator tag_entity_begin(std::string tagName);`  
returns an iterator to the beginning of the entities associated with a tag named `tagName`.
- `inline tag_entity_iterator tag_entity_end(std::string tagName);`  
returns an iterator to the end of the entities associated with a tag named `tagName`.

Note that any of these methods may throw an exception of type `UnstructuredMapping::TagError`. Dereferencing an `entity_tag_iterator` results in a reference to an `EntityTag`. Dereferencing a `tag_entity_iterator` results in a reference to an instance of `UnstructuredMapping::IDTuple` whose straightforward definition is :

```
struct IDTuple
{
    EntityTypeEnum et; int e;

    inline IDTuple(EntityTypeEnum et_=Invalid, int e_=-1) : et(et_), e(e_) { }
    inline IDTuple( const IDTuple &id ) : et(id.et), e(id.e) { }
    inline ~IDTuple() {}

    inline bool operator< ( const IDTuple & id ) const
    inline bool operator<= ( const IDTuple & id )
    inline bool operator==( const IDTuple & id ) const
    inline bool operator!=( const IDTuple & id ) const
    inline bool operator!= ( const IDTuple & id )
};

};
```

`IDTuple`'s can be used to provide a shorthand for comparing entities. This class may be merged into `UnstructuredMappingIterator` at some later date.

### 43.6 Other useful stuff

Two new public static class members have been added to `UnstructuredMapping` :

```
static aString EntityTypeStrings[];
static aString ElementTypeStrings[];
```

Each of these arrays store string “names” associated with each `EntityTypeEnum` (except `Invalid` whose value is -1) and `ElementType`. These are often useful for diagnostic purposes.

A function to provide integrity checking of meshes and the connectivity built by `UnstructuredMapping` has been created:

- `bool verifyUnstructuredConnectivity( UnstructuredMapping &umap, bool verbose );`

performs several tests on the connectivity and geometry in the `UnstructuredMapping` `umap`. If `verbose` is true the function will output any errors as well as some information collected during the integrity checks. The function returns true if there were no errors.

`verifyUnstructuredConnectivity` first checks to make sure the downward and upward adjacencies agree. In other words, it makes sure that each entity is contained in the downward adjacencies of higher dimensional entities and the upward adjacency of lower dimensional ones. These tests ensure that adjacency loops are consistent with one another. The second part of the test is more comprehensive and involves checking the orientations of the adjacencies as well as the geometric validity of the mesh. Geometric validity means that each “Face” in a 2D mesh has a positive area computed with the adjacency information and that each “Region” in 3D has a positive volume. Each “Edge” is also checked for duplicate nodes (zero length edges). During the tests, the volumes or areas are computed several times using the available adjacencies. Inconsistencies in the orientations are detected when these volumes are not the same sign or magnitude. The typical output of this function when the “`verbose`” option is true looks like:

```
==== VERIFY CONNECTIVITY REPORT =====
NUMBER OF ERRORS : 0
NUMBER OF WARNINGS : 0
--- Entity Information -----
Vertex Count : 8121
Edge Count : 34712
Face Count : 47248
Region Count : 20571
--- Element Information -----
triangle Count : 34268
quadrilateral Count : 12980
tetrahedron Count : 14419
pyramid Count : 2644
triPrism Count : 0
septahedron Count : 0
hexahedron Count : 3508
--- Geometric Information -----
Min. Edge : 0.0302402
Max. Edge : 0.245611
Avg. Edge : 0.0889839
Min. Vol : 1.65491e-09
Max. Vol : 0.00461466
Avg. Vol : 0.00017042
=====
```

## 43.7 Examples

### 43.7.1 Iterating through entities of a given type

Here is an iteration similar to the one in the introduction, except we choose a different entity to iterate through:

```
UnstructuredMappingIterator edge;
for ( edge = umap.begin(UnstructuredMapping::Edge);
      edge != umap.end(UnstructuredMapping::Edge);
      edge++ )
  cout<<"Here is an Edge with index "<<*edge<<endl;
```

### 43.7.2 Adjacency iterations

Upward and downward adjacencies are determined by the arguments to the `adjacency_begin/end` functions or the constructor to the `UnstructuredMappingAdjacencyIterator`. The `to` entity type is compared to the type of the `from` iterator for this purpose. Here is a sample downward iteration of the vertices in an edge:

```
UnstructuredMappingIterator edge = umap.begin(UnstructuredMapping::Edge);
UnstructuredMappingAdjacencyIterator edgeVert;
for ( edgeVert = umap.adjacency_begin(edge, UnstructuredMapping::Vertex);
      edgeVert != umap.adjacency_end(edge, UnstructuredMapping::Vertex);
      edgeVert++ )
    cout<<"Edge "<<*edge<<" has Vertex "<<*edgeVert<<endl;
```

We can define an upwards iteration of the regions, for example, surrounding the edge by altering the second argument of the adjacency begin and end methods:

```
UnstructuredMappingIterator edge = umap.begin(UnstructuredMapping::Edge);
UnstructuredMappingAdjacencyIterator edgeReg;
for ( edgeReg = umap.adjacency_begin(edge, UnstructuredMapping::Region);
      edgeReg != umap.adjacency_end(edge, UnstructuredMapping::Region);
      edgeReg++ )
    cout<<"Edge "<<*edge<<" has Region "<<*edgeReg<<endl;
```

### 43.7.3 Tag all vertices on the boundary of the mesh

This example shows how to use `EntityTypeEnum`'s in a general way to write generic algorithms. This example marks all the vertices that live on the boundary of the mesh with a tag called "boundary vertex".

```
// determine the highest dimensional entity that bounds the mesh
UnstructuredMapping::EntityTypeEnum cellBdyType = umap.getRangeDimension() == 2 ?
    UnstructuredMapping::Edge : UnstructuredMapping::Face;

// the next higher entity we will call the ‘‘cell’’
UnstructuredMapping::EntityTypeEnum cellType = ((int)cellBdyType) + 1;

UnstructuredMappingIterator e_iter;
UnstructuredMappingAdjacencyIterator cellIter, vertIter;

// iterate through the bounding entities and determine if they are on the boundary
for ( e_iter=umap.begin(cellBdyType); e_iter!=umap.end(cellBdyType); e_iter++ )
{
    // an e_iter is on the boundary if it only has one neighboring cell
    int nAdj=0;
    for ( cellIter=umap.adjacency_begin(e_iter, cellType);
          cellIter!=umap.adjacency_end(e_iter, cellType); cellIter++ )
        nAdj++;

    if ( nAdj==1 )
    {
        // we are on a boundary, tag the vertices as such
        for ( vertIter=umap.adjacency_begin(e_iter, UnstructuredMapping::Vertex);
              vertIter!=umap.adjacency_end(e_iter, UnstructuredMapping::Vertex);
```

```

        vertIter++ )
    if ( !umap.has_tag(UnstructuredMapping::Vertex,
                        *vertIter, "boundary vertex") )
        umap.addTag(UnstructuredMapping::Vertex, *vertIter,
                    "boundary vertex", ((void *)*vertIter));
}
}

```

The first step determines the highest dimensional bounding entity of the mesh, in 2D this is an Edge, in 3D this is a Face. We then compute the “cell” type, which is the first upward adjacency of the bounding entity (Face in 2D, Region in 3D). We loop through each of the entities of the type that bound the mesh and determine whether they sit on the boundary. If an entity does sit on the boundary, each of its vertices are tagged with a tag named “boundary vertex”. The data we store in the tag is just the index of the vertex associated with the tag.

#### 43.7.4 Computing 2D Face areas

Here is one way to compute the areas of faces in a 2D mesh. Note that this method will also work for arbitrary polygonal faces and could be optimized for tets and quads if necessary:

```

UnstructuredMapping umap; // get this from somewhere
UnstructuredMappingIterator edge;
UnstructuredMappingAdjacencyIterator edgeVert, edgeCell;

ArraySimple< ArraySimpleFixed<real,3,1,1,1> >
    cellCenters(umap.size(UnstructuredMapping::Face));

// compute the cell centers with another loop ...

realArray cellAreas(umap.size(UnstructuredMapping::Face));
cellAreas = 0;

// loop through all the edges in the mesh and compute the ‘‘side’’ area’s
// side areas are computed from the edge and the cell centeres. These areas
// are then added to the corresponding Face.

for ( edge = umap.begin(UnstructuredMapping::Edge);
      edge != umap.end(UnstructuredMapping::Edge);
      edge++ )
{
    ArraySimpleFixed<real,2,1,1,1> edgeVertices[2];

    // in 2D we connect the cell (face) center to
    // the center of the edge to form the area normals
    ArraySimpleFixed<real,2,1,1,1> edgeVertices[2], edgeCenter;
    int v=0;

    // get the vertices for the edge
    for ( edgeVert = umap.adjacency_begin(edge,UnstructuredMapping::Vertex);
          edgeVert != umap.adjacency_end(edge,UnstructuredMapping::Vertex);
          edgeVert++ )

```

```

{ // we could optimize this loop by getting the edge entities directly
    for ( int a=0; a<rDim; a++ )
        edgeVertices[v][a] = vertices(*edgeVert,a);
}

// loop through the adjacent Face's and compute each side area,
// adding each area to each Face as we go along
for ( edgeCell = umap.adjacency_begin(edge, UnstructuredMapping::Face );
      edgeCell != umap.adjacency_end(edge, UnstructuredMapping::Face );
      edgeCell++ )
{
    // compute the area of the "side"
    real area = edgeCell.orientation()*triangleArea2D(edgeVertices[0],
                                                       edgeVertices[1], cellCenters[*edgeCell]);
    // add the area to the cell;
    cellAreas(*edgeCell) += area;
}
}

```

### 43.7.5 A generic finite volume gradient operator

In this example we construct a generic discrete gradient operator using the finite volume method. We can approximate the gradient of a function  $u$  using finite volumes in the usual way with the identity:

$$\int_{\Omega} \nabla u d\Omega = \int_{\partial\Omega} u \mathbf{n} ds \quad (20)$$

Where  $\Omega$  is some domain bounded by  $\partial\Omega$  with an area normal given by  $\mathbf{n} ds$ . If we replace  $\nabla u$  by its average , denoted  $(\nabla u)_{\Omega}$  we get the following familiar approximation:

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \int_{\partial\Omega} u \mathbf{n} ds \quad (21)$$

There are, of course, a variety of ways to compute this approximation depending on the centering of  $u$  on the mesh (Face, Vertex, Edge, etc) and the desired centering of the resulting gradient. If we assume that  $\Omega$  is represented by polygons in 2D and polyhedra in 3D we can compute the approximation by summing the value of  $u \mathbf{n} ds$  on the boundary of a polygon or polyhedra using  $\mathbf{n} ds$  computed from the straight line segments (2D) or polygonal surfaces (3D) bounding  $\Omega$  :

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \sum_{b=1}^m u_b (\mathbf{n} ds)_b \quad (22)$$

where the subscript  $b$  denotes a facet of the boundary of the polygon or polyhedron. An approximation for  $u_b$  is still needed. In some instances, this value may exist at the same centering for  $u$  and can be computed by averaging the values on either side of each facet. In other cases, such as a staggered grid,  $u$  may exist at a centering with a different adjacency to the gradient ( $u$  could exist on the Vertices while  $\nabla u$  lives on Faces or Regions). In either case, we compute  $u_b$  by averaging values adjacent to the facets of  $\Omega$  :

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \sum_{b=1}^m \left( \frac{1}{n} \sum_{c=1}^n u_c^b \right) (\mathbf{n} ds)_b \quad (23)$$

where  $u_c^b$  are the values of  $u$  adjacent to facet  $b$ .

First we compute the volumes and surface normals required. By noting that each surface is adjacent to two regions,  $\Omega_1$  and  $\Omega_2$ , we can loop through the surfaces computing the value of  $u_b$  and adding the result of  $u_b(\mathbf{nds})_b$  to  $\Omega_1$  and subtracting it from  $\Omega_2$  (subtract since the area normal relative to  $\Omega_2$  points in the opposite direction to  $\Omega_1$ ). The resulting algorithm looks like:

1. Compute the volumes,  $\Omega$ , of the polygons or polyhedra at the desired centering for  $(\nabla u)_\Omega$
2. Compute the area normals,  $\mathbf{nds}$ , of the facets at the centering corresponding to the boundary of each  $\Omega$
3. For each surface **surf**:
  - (a) **numberAdjacent** = 0
  - (b)  $u_b = 0$
  - (c) For each  $u$ , (denoted by **u\_cell**), adjacent to **surf**
    - i.  $u_b+ = u_{\mathbf{u\_cell}}^{\mathbf{surf}}$
    - ii. **numberAdjacent** ++
  - (d)  $u_b/ = \mathbf{numberAdjacent}$
  - (e)  $u_{\Omega_1}+ = \frac{1}{\Omega_1} u_b(\mathbf{nds})_{\mathbf{surf}}$
  - (f)  $u_{\Omega_2}- = \frac{1}{\Omega_2} u_b(\mathbf{nds})_{\mathbf{surf}}$

The code for this algorithm using the new `UnstructuredMapping` iterator interface is given in the function `computeGradient`. `centering` refers to the centering of the function  $u$ ; `surfaceCentering` the centering of the surface normals; and `gradientCentering` refers to the centering of `gradu`.

```

void computeGradient(UnstructuredMapping &umap,
                     UnstructuredMapping::EntityTypeEnum centering,
                     UnstructuredMapping::EntityTypeEnum surfaceCentering,
                     UnstructuredMapping::EntityTypeEnum gradientCentering,
                     realArray &u, realArray &surfaceNormals, realArray &volumes,
                     realArray &gradu)
{
    UnstructuredMappingIterator surf;
    UnstructuredMappingAdjacencyIterator u_cell, g_cell;

    gradu.redim(umap.size(centering), umap.getRangeDimension());
    gradu = 0;

    for (surf = umap.begin(surfaceCentering);
         surf != umap.end(surfaceCentering);
         surf++)
    {
        int nC=0;
        real uOnSurf=0;
        for (u_cell = umap.adjacency_begin(surf, centering);
             u_cell != umap.adjacency_end(surf, centering);
             u_cell++)
        {
            uOnSurf += u(*u_cell);
            nC++;
        }

        uOnSurf /= real(nC);

        for (g_cell = umap.adjacency_begin(surf, gradientCentering);
             g_cell != umap.adjacency_end(surf, gradientCentering);
             g_cell++)
            for (int a=0; a<umap.getRangeDimension(); a++)
                gradu(*g_cell,a) +=
                    g_cell.orientation()*surfaceNormals(*surf,a)*uOnSurf/volumes(*g_cell);
    }
}

```

This function can be used to compute Vertex centered gradients using Vertex centered values for  $u$  by calling:

```
computeGradient(umap,
                UnstructuredMapping::Vertex, UnstructuredMapping::Edge,
                UnstructuredMapping::Vertex,
                vertexCenteredScalar, edgeAreaNormals, vertexVolumes,
                vertexCenteredGradient);
```

It can also be used to compute “cell” (Face or Region) centered gradients using:

```
computeGradient(umap,
                cellType, UnstructuredMapping::EntityTypeEnum( ((int)cellType)-1 ),
```

```

    cellType,
    cellCenteredScalar, cellSurfNormals, cellVolumes,
    cellCenteredGradient);

```

Yet another example places  $u$  at the vertices but computes  $(\nabla u)_\Omega$  at Face or Region (Cell) centers:

```

computeGradient(umap,
    UnstructuredMapping::Vertex, UnstructuredMapping::EntityTypeEnum( ((int)cellType)-
    cellType,
    vertexCenteredScalar, cellSurfNormals, cellVolumes,
    cellCenteredGradientFromNodeU);

```

## 43.8 Implementation Details

Since the supported element types are limited to the zoo, a canonical ordering of the vertices, faces, edges, sides, etc., in each element type can be constructed. Currently implemented three dimensional orderings are illustrated by Figure 32. In two dimensions the vertices and faces are simply ordered counter-clockwise starting from 0.

## 43.9 Iterations using the old connectivity interface

Iterating through the connectivity using the old interface consists of using a few inlinable functions which abstract away the underlying representation, including the canonical orderings of the elements. Normally, a user will never even know the orderings exist (at least they should beware of depending upon them!). Iterations are also independent of the dimension of the mesh since the connectivity for all dimensions share the same nomenclature ( eg. an edge in 2D is the same as an edge in 3D ). The following subsections provide examples of how to navigate and use the limited set of iterations available in an `UnstructuredMapping`.

### 43.9.1 Element iteration

```

// assuming an UnstructuredMapping named um exists...
const IntegerArray &elements = um.getElements();
for ( int e=0; e<um.getNumberOfElements(); e++ ) {
    // ... do stuff with the element index
    elementScalar(e) = whatever;
}

```

### 43.9.2 Vertex iteration

```

// assuming an UnstructuredMapping named um exists...
const IntegerArray &vertices = um.getVertices();
for ( int v=0; v<um.getNumberOfVertices(); v++ ) {
    // ... do stuff with the vertex index
    vertexScalar(v) = whatever;
}

```

### 43.9.3 Iteration through the vertices in an element

```

for ( int e=0; e<um.getNumberOfElements(); e++ ) {
    for ( int v=0; v<um.getNumberOfVerticesThisElement(e); v++ ) {
        vGlobalIndex = um.elementGlobalVertex(e,v);
        // ... do stuff with global vertex index
        vertexScalar(vGlobalIndex) = whatever;
    }
}

```

#### 43.9.4 Iteration through the faces

```
const IntegerArray &elements = um.getElements();
const IntegerArray &faceElements = um.getFaceElements();
for ( int f=0; f<um.getNumberOfFaces(); f++ ) {
    // get the elements on either side of the face
    int element0 = faceElements(f, 0);
    int element1 = faceElements(f, 1);
    // ... do stuff with the face and element indices
    faceScalar(f) = whatever;
    elementScalar(element0) += -face;
    elementScalar(element1) += face;
}
```

#### 43.9.5 Iteration through the vertices in a face

```
const IntegerArray &elements = um.getElements();
for ( int f=0; f<um.getNumberOfFaces(); f++ ) {
    const IntegerArray &faceVertices = um.getFaceVertices(f);
    for ( int v=0; v<um.getNumberOfVerticesThisFace(f); v++ )
        vGlobalIndex = um.faceGlobalVertex(f,v);
        // ... do stuff with global vertex index
        vertexScalar(vGlobalIndex) = whatever;
}
```

### 43.10 Enum Types

ElementType enumerates the supported unstructured entity types.

```
enum ElementType
{
    triangle,
    quadrilateral,
    tetrahedron,
    pyramid,
    triPrism,
    septahedron, // pray we never need...
    hexahedron,
    other,
    boundary
};
```

EntityTypeEnum is used to determine the topological dimension of a given entity.

```
enum EntityTypeEnum
{
    Invalid=-1,
    Vertex=0,
    Edge,
    Face,
    Region,
    Mesh, // kkc put this here to enable "Mesh" tagging...
    NumberOfEntityTypes
}
```

In 2D, only triangles and quadrilaterals are supported. More types are supported in 3D, but in general these consist of the “finite element zoo”. These elements are hexahedra or degenerate hexahedra. Currently the septahedron is not supported as this shape is rather unusual and rarely (?) encountered (7 nodes, 6 faces). **other** implies any shape not described by the previous enums and at this point could include arbitrary polyhedra (although the connectivity to support arbitrary polyhedra is not implemented). **boundary** elements are typically placeholders. **boundaries** will not have a specific geometry associated with them and may only consist of a limited set of connectivity.

### 43.11 File Formats

`UnstructuredMapping`'s can be written to two different kinds of files using the member functions `get` and `put`. Using `get` or `put` with an `Overture GenericDatabase` class as the first argument performs io directly to an Overture database file. During a `put`, the instance's arrays `node` and `element` are written to the database file. A `get` retrieves these arrays and reconstructs the connectivity. `UnstructuredMapping`'s can also be read/written to ASCII files using a simplified version of a format commonly called "ingrid", or "DYNA". This io method can be invoked by calling `get` and `put` with a string, the filename, as the first argument. The resulting file looks like :

```
A text header line (optional)
number of meshes, number of nodes, number of elements, domain dimension(optional), range dimension(optional)
node0 ID, x0, y0, z0
node1 ID, x1, y1, z1
.

.

nodeN ID, xN, yN,zN
element0 ID, tag0, n1,n2,n3,n4,n5,n6,n7,n8
element1 ID, tagN, node ID list
.

.

elementN ID, tagN, node ID list
```

Typically, Overture writes "OVERTUREUMapping" in the comment line and uses the optional spaces for the domain dimension and range dimension. These details, however, are not required and meshes from a variety of mesh generation tools have been read in. The node ID lists in the element lines are lists of global node ID's (listed in the node section), that are in each element. The ordering of the nodes in the list follows the canonical ordering described in Figure 32. Currently the code requires that the nodes be listed in ascending order of thier node ID's and that the node IDs be contiguous. By the way, the number of meshes in the file is ignored, only one mesh per file is supported at the moment.

### 43.12 Relationship to Normal Overture Mappings

While `UnstructuredMapping` inherits from class `Mapping`, there are a few caveats. By its very nature, the inverse does not exist for an `UnstructuredMapping`. Any use of an `UnstructuredMapping` in the context of mapping inverses should be prevented; all member functions dealing with inverses now throw exceptions. However, a `domainDimension` and `rangeDimension` are both still used in the mapping to help denote the difference between 2D meshes, 3D surface meshes and 3D volume meshes. With these exceptions, `UnstructuredMappings` should play nicely with conventional sturctured ones. In particular, a structured mapping can be converted into an unstructured one by using the member function `buildFromAMapping`.

### 43.13 New Connectivity Interface

### 43.14 Member Function Descriptions

### 43.15 Constructor

```
UnstructuredMapping(int domainDimension_ =3,
                    int rangeDimension_ =3,
                    mappingSpace domainSpace_ =parameterSpace,
                    mappingSpace rangeSpace_ =cartesianSpace)
```

**Description:** Default Constructor

### 43.16 Constructor

**UnstructuredMapping()**

**Description:** Default Constructor

### 43.17 addGhostElements

```
void  
addGhostElements( bool trueOrFalse )
```

**Description:** Specify whether to add ghost elements to the unstructured mapping.

**trueOrFalse (input):** If true add ghost elements to the unstructured mapping.

### 43.18 getBoundaryFace

```
const intArray &  
getBoundaryFace()
```

**Description:** Return a list of boundary faces,

**boundaryFace (return value) :** faces on the boundary.

### 43.19 getGhostElements

```
const intArray &  
getGhostElements() const
```

**Description:** Return a list of ghost elements.

**boundaryFace (return value) :** faces on the boundary.

### 43.20 getMask

```
const intArray &  
getMask(EntityTypeEnum entityType) const
```

**Description:** Return a the mask associated with the nodes, edges, faces or regions.

**boundaryFace (return value) :** faces on the boundary.

### 43.21 getBoundaryFaceTags

```
const intArray &  
getBoundaryFaceTags()
```

**Description:** Return a list of the tags on each boundary face, usefull for boundary conditions

**boundaryFaceTags (return value) :** tags for faces on the boundary.

### 43.22 getNumberOfNodes

```
int  
getNumberOfNodes() const
```

**Description:** Return the number of nodes.

### **43.23 getMaxNumberOfNodesPerElement**

```
int  
getMaxNumberOfNodesPerElement() const
```

**Description:** Return the maximum number of nodes per element (max over all elements).

### **43.24 getMaxNumberOfNodesPerElement**

```
int  
getMaxNumberOfFacesPerElement() const
```

**Description:** Return the maximum number of faces per element (max over all elements).

### **43.25 getMaxNumberOfNodesPerFace**

```
int  
getMaxNumberOfNodesPerFace() const
```

**Description:** Return the maximum number of nodes per face (max over all faces).

### **43.26 getNumberOfElements**

```
int  
getNumberOfElements() const
```

**Description:** Return the number of elements (such as the number of triangles on a 2d grid or 3d surface or the number of tetrahedra in a 3d grid).

### **43.27 getNumberOfFaces**

```
int  
getNumberOfFaces() const
```

**Description:** Return the number of faces.

### **43.28 getNumberOfBoundaryFaces**

```
int  
getNumberOfBoundaryFaces() const
```

**Description:** Return the number of faces.

### **43.29 getNumberOfEdges**

```
int  
getNumberOfEdges() const
```

**Description:** Return the number of edges.

### 43.30 getNodes

```
const realArray &
getNodes() const
```

**Description:** Return the list of nodes.

**node (return value)** : list of nodes, node(i,0:r-1) : (x,y) or (x,y,z) coordinates for each node, i=0,1,...  
r=rangeDimension

### 43.31 getElements

```
const intArray &
getElements() const
```

**Description:** Return the node information for each element.

**element (return value)** : defines the nodes that make up each element (e.g. triangle), element(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element i will have nodes (element(i,0),element(i,1),...)

### 43.32 getFaces

```
const intArray &
getFaces()
```

**Description:** Return the connectivity information for each face.

**face (return value)** : defines the nodes that make up each face (e.g. triangle), face(i,n) index into the nodes array for the node n of face i,

### 43.33 getFaces

```
const intArray &
getFaceElements()
```

**Description:** Return the connectivity information containing the elements adjacent to each face.

**faceElements (return value)** : defines the elements adjacent to each face, faceElements(i,e) index into the elements array for the element e of face i, for now e=0,1 since each face has two elements. For now, faces on a boundary return -1 for e=1.

### 43.34 getEdges

```
const intArray &
getEdges()
```

**Description:** Return the connectivity information for each edge.

**edge (return value)** : defines the 2 nodes that make up each edge. face(i,n) index into the nodes array for the node n of face i, for now n=0,1. Thus edge 0 will have end points with node numbers (edge(i,0),edge(i,1))

### **43.35 getElementFaces**

```
const intArray &
getElementFaces()
```

**Description:** Return the connectivety array describing the faces that belong to an element.

**elementFaces (return value) :** defines the faces that belong to an element. face=elementFaces(e,i) is the face for i=0,1,...

### **43.36 getTags**

```
const intArray &
getTags()
```

**Description:** Return the element tagging information.

**tags (return value) :** an integer tag for each element, defaults to 0 for every element

### **43.37 setElementDensityTolerance**

```
void
setElementDensityTolerance(real tol)
```

**Description:** Specify the tolerance for determining the triangle density when building from a mapping.  
The smaller the tolerance the more triangles. Choose a value of zero to use the default number of elements

**tol (input) :** new tolerance.

### **43.38 setTags**

```
void
setTags( const intArray &new_tags )
```

**Description:** Set the list of tags for each element;

**tags (input) :** an array the length of the number of elements containing an integer tag for each element  
(eg like material region identifier)

### **43.39 setNodesAndConnectivity**

```
int
setNodesAndConnectivity( const realArray & nodes,
                        const intArray & elements,
                        int domainDimension_ ==-1,
                        bool buildConnectivity =true)
```

**Description:** Supply a list of nodes and a list of connectivity information.

**nodes (input) :** nodes(i,0:r-1) (x,y) or (x,y,z) coordinates for each node, i=0,1,... r=rangeDimension

**elements (input) :** defines the nodes that make up each element (e.g. triangle), elements(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element 0 will have nodes (elements(i,0),elements(i,1),...) A value of elements(i,n)==-1 means no node is used. This option is used to specify elements with different numbers of nodes per elements. For example if one has quadrilaterals and triangles then set element(i,3)==-1 for triangles.

## 43.40 setNodesElementsAndNeighbours

int

```
setNodesElementsAndNeighbours(const realArray & nodes,
                               const intArray & elements,
                               const intArray & neighbours,
                               int numberOfFaces_ ==-1,
                               int numberOfBoundaryFaces_ ==-1,
                               int domainDimension_ ==-1)
```

**Description:** Supply a list of nodes, elements and element neighbours. The element neighbours are used in building the connectivity information. This should be faster than using setNodesAndConnectivity.

**nodes (input) :** nodes(i,0:r-1) (x,y) or (x,y,z) coordinates for each node, i=0,1,... r=rangeDimension

**elements (input) :** defines the nodes that make up each element (e.g. triangle), elements(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element 0 will have nodes (elements(i,0),elements(i,1),...) A value of elements(i,n)==-1 means no node is used. This option is used to specify elements with different numbers of nodes per elements. For example if one has quadrilaterals and triangles then set element(i,3)==-1 for triangles.

**neighbours (input) :** a list of neighbours for each element. / **numberOfFaces\_ (input) :** optionally supply the number of faces, if known. / **numberOfBoundaryFaces\_ (input) :** optionally supply the number of boundary faces, if known.

## 43.41 setNodesAndConnectivity

int

```
setNodesAndConnectivity( const realArray & nodes,
                        const intArray & elements,
                        const intArray & faces,
                        const intArray & faceElements_,
                        const intArray & elementFaces_,
                        int numberOfFaces_ ==-1,
                        int numberOfBoundaryFaces_ ==-1,
                        int domainDimension_ ==-1,
                        bool constantNumberOfNodesPerElement =false)
```

**Description:** Supply a list of nodes, elements and element neighbours. The element neighbours are used in building the connectivity information. This should be faster than using setNodesAndConnectivity.

**nodes (input) :** nodes(i,0:r-1) (x,y) or (x,y,z) coordinates for each node, i=0,1,... r=rangeDimension

**elements (input) :** defines the nodes that make up each element (e.g. triangle), elements(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element 0 will have nodes (elements(i,0),elements(i,1),...) A value of elements(i,n)==-1 means no node is used. This option is used to specify elements with different numbers of nodes per elements. For example if one has quadrilaterals and triangles then set element(i,3)==-1 for triangles.

**faces (input):**

**faceElements\_ (input):**

**elementFaces\_ (input):** / **numberOfFaces\_ (input) :** optionally supply the number of faces, if known. / **numberOfBoundaryFaces\_ (input) :** optionally supply the number of boundary faces, if known.

#### 43.42 buildFromAMapping

```
intArray  
buildFromAMapping( Mapping & map, intArray &maskin = nullIntArray())
```

**Description:** Builds an unstructured mapping from another mapping. There are no duplicate nodes.

Degenerate elements occurring from coordinate singularities and periodic boundaries are detected and the appropriate element ( hex, prism, pyramid, tet) is created in the UnstructuredMapping. For example, a spherical polar mesh will, in general, have all four element types with pyramids at the spherical singularity, tetrahedron connecting the pyramids to the polar axes, prisms along each polar axis and hexahedra everywhere else. A mask array can optionally be provided to exclude vertices/elements from the new UnstructuredMapping. However, building a new UnstructuredMapping from a masked UnstructuredMapping is NOT yet supported. The implementor is a bit lazy.

**map (input)** : Mapping to use.

**maskin (input)** : pointer to a vertex mask array to determine which nodes/elements to use

**Returns :** An IntegerArray mapping the vertices in the original Mapping to the vertices in the new UnstructuredMapping. If the value of the returned array is -1 at any vertex, then that vertex was masked out of the original mapping.

**Comments :** Currently the code implements a rather complex algorithm to assign vertex id's to the boundary nodes. The complexity of the coding is due to the possibility of polar singularities ( with the possible occurrence of a spherical singularity ) as well as periodic boundaries. These special cases can occur on any side of any coordinate axis in 2 and 3d. The approach became more complicated than originally intended, there may be a more straightforward way and any suggestions are welcome.

#### 43.43 printConnectivity

```
int  
printConnectivity( FILE *file stdout)
```

**Description:**

#### 43.44 printConnectivity

```
int  
checkConnectivity( bool printResults =true,  
                  IntegerArray *pBadElements =NULL)
```

**Description:** Perform consistency checks on the connectivity.

**printResults (input):** output the results if true.

**pBadElements (input/output) :** If not null, return a list of the bad Elements.

**return value:** number of errors found.

#### 43.45 printStatistics

```
int  
printStatistics(FILE *file =stdout)
```

**Description:** print some timing statistics.

#### 43.46 get from an ascii file

```
int  
get( const aString & fileName )
```

**Description:** Read the unstructured grid from an ascii file.

**fileName (input)** : name of the file to save the results in.

#### 43.47 put to an ascii file

```
int  
put( const aString & fileName ) const
```

**Description:** Save the unstructured grid to an ascii file.

**fileName (input)** : name of the file to save the results in.

#### 43.48 findBoundaryCurves

```
int  
findBoundaryCurves(int & numberOfBoundaryCurves, Mapping **& boundaryCurves )
```

**Description:** Locate boundary curves on a 3D surface – booth curve segments on the boundary.

**numberOfBoundaryCurves (output)** : number of boundary curves found.

**boundaryCurves (output)** : Boundary curves as spline mappings. **NOTE:** This routine will increment the reference count for you.

#### 43.49 Constructor

```
UnstructuredMapping(int domainDimension_ =3,  
                    int rangeDimension_ =3,  
                    mappingSpace domainSpace_ =parameterSpace,  
                    mappingSpace rangeSpace_ =cartesianSpace)
```

**Description:** Default Constructor

#### 43.50 Constructor

```
UnstructuredMapping()
```

**Description:** Default Constructor

#### 43.51 getNumberOfNodes

```
int  
getNumberOfNodes() const
```

**Description:** Return the number of nodes.

### **43.52 getMaxNumberOfNodesPerElement**

```
int  
getMaxNumberOfNodesPerElement() const
```

**Description:** Return the maximum number of nodes per element (max over all elements).

### **43.53 getMaxNumberOfNodesPerElement**

```
int  
getMaxNumberOfFacesPerElement() const
```

**Description:** Return the maximum number of faces per element (max over all elements).

### **43.54 getMaxNumberOfNodesPerFace**

```
int  
getMaxNumberOfNodesPerFace() const
```

**Description:** Return the maximum number of nodes per face (max over all faces).

### **43.55 getNumberOfElements**

```
int  
getNumberOfElements() const
```

**Description:** Return the number of elements (such as the number of triangles on a 2d grid or 3d surface or the number of tetrahedra in a 3d grid).

### **43.56 getNumberOfFaces**

```
int  
getNumberOfFaces() const
```

**Description:** Return the number of faces.

### **43.57 getNumberOfBoundaryFaces**

```
int  
getNumberOfBoundaryFaces() const
```

**Description:** Return the number of faces.

### **43.58 getNumberOfEdges**

```
int  
getNumberOfEdges() const
```

**Description:** Return the number of edges.

### 43.59 getNodes

```
const realArray &
getNodes() const
```

**Description:** Return the list of nodes.

**node (return value)** : list of nodes, node(i,0:r-1) : (x,y) or (x,y,z) coordinates for each node, i=0,1,...  
r=rangeDimension

### 43.60 getElements

```
const intArray &
getElements() const
```

**Description:** Return the node information for each element.

**element (return value)** : defines the nodes that make up each element (e.g. triangle), element(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element i will have nodes (element(i,0),element(i,1),...)

### 43.61 getFaces

```
const intArray &
getFaces() const
```

**Description:** Return the connectivity information for each face.

**face (return value)** : defines the nodes that make up each face (e.g. triangle), face(i,n) index into the nodes array for the node n of face i, for now n=0,1,2 for triangles. Thus face 0 will have nodes (face(i,0),face(i,1),...)

### 43.62 getFaceElements

```
const intArray &
getFaceElements() const
```

**Description:** Return the connectivity information containing the elements adjacent to each face.

**faceElements (return value)** : defines the elements adjacent to each face, faceElements(i,e) index into the elements array for the element e of face i, for now e=0,1 since each face has two elements. For now, faces on a boundary return -1 for e=1.

### 43.63 getEdges

```
const intArray &
getEdges() const
```

**Description:** Return the connectivity information for each edge.

**edge (return value)** : defines the 2 nodes that make up each edge. face(i,n) index into the nodes array for the node n of face i, for now n=0,1. Thus edge 0 will have end points with node numbers (edge(i,0),edge(i,1))

### 43.64 getTags

```
const IntegerArray &
getTags() const
```

**Description:** Return the element tagging information.

**tags (return value)** : an integer tag for each element, defaults to 0 for every element

### 43.65 setTags

```
void
setTags( const IntegerArray &new_tags )
```

**Description:** Set the list of tags for each element;

**tags (input)** : an array the length of the number of elements containing an integer tag for each element  
(eg like material region identifier)

### 43.66 setNodesAndConnectivity

```
int
setNodesAndConnectivity( const realArray & nodes,
                        const intArray & elements,
                        int domainDimension_ = -1)
```

**Description:** Supply a list of nodes and a list of connectivity information.

**nodes (input)** : nodes(i,0:r-1) (x,y) or (x,y,z) coordinates for each node, i=0,1,... r=rangeDimension

**elements (input)** : defines the nodes that make up each element (e.g. triangle), elements(i,n) index into the nodes array for the node n of element i, for now n=0,1,2 for triangles. Thus element 0 will have nodes (elements(i,0),elements(i,1),...)

### 43.67 project

```
int
project( realArray & x, MappingProjectionParameters & mpParameters )
```

**Description:** Project points onto the surface

**x (input)** : project these points.

**mpParameters** : holds auxillary data to aid in the projection.

### 43.68 buildFromAMapping

```
intArray
buildFromAMapping( Mapping & map, intArray &maskin = nullIntArray() )
```

**Description:** Builds an unstructured mapping from another mapping. There are no duplicate nodes. Degenerate elements occurring from coordinate singularities and periodic boundaries are detected and the appropriate element ( hex, prism, pyramid, tet) is created in the UnstructuredMapping. For example, a spherical polar mesh will, in general, have all four element types with pyramids at the spherical singularity, tetrahedron connecting the pyramids to the polar axes, prisms along each

polar axis and hexahedra everywhere else. A mask array can optionally be provided to exclude vertices/elements from the new UnstructuredMapping. However, building a new UnstructuredMapping from a masked UnstructuredMapping is NOT yet supported. The implementor is a bit lazy.

**map (input) :** Mapping to use.

**maskin (input) :** pointer to a vertex mask array to determine which nodes/elements to use

**Returns :** An IntegerArray mapping the vertices in the original Mapping to the vertices in the new UnstructuredMapping. If the value of the returned array is -1 at any vertex, then that vertex was masked out of the original mapping.

**Comments :** Currently the code implements a rather complex algorithm to assign vertex id's to the boundary nodes. The complexity of the coding is due to the possibility of polar singularities ( with the possible occurrence of a spherical singularity ) as well as periodic boundaries. These special cases can occur on any side of any coordinate axis in 2 and 3d. The approach became more complicated than originally intended, there may be a more straightforward way and any suggestions are welcome.

#### 43.69 get from an ascii file

```
int  
get( const String & fileName )
```

**Description:** Read the unstructured grid from an ascii file.

**fileName (input) :** name of the file to save the results in.

#### 43.70 put to an ascii file

```
int  
put( const String & fileName ) const
```

**Description:** Save the unstructured grid to an ascii file.

**fileName (input) :** name of the file to save the results in.

#### 43.71 Constructor

```
void  
buildFromACompositeGrid( CompositeGrid &cg )
```

**Description:** build an unstructured mapping from a composite grid

**cg (input) :** a composite grid that may or may not be a hybrid grid

**Comments :** The composite grid has no restrictions, it could be an overlapping grid or hybrid mesh. In the case of an overlapping grid, the UnstructuredMapping essentially consists of overlapping sections and holes that have no connectivity information. A hybrid mesh becomes one consistent UnstructuredMapping.

#### 43.72 getColour

```
aString  
getColour( ) const
```

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.73 setColour

```
int  
setColour( const aString & colour )
```

**Purpose:** Set the colour for the grid.

**colour (input) :** the name of the colour such as "red", "green",...

### 43.74 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &gi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.75 getColour

```
aString  
getColour( ) const
```

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.76 setColour

```
int  
setColour( const aString & colour )
```

**Purpose:** Set the colour for the grid.

**colour (input) :** the name of the colour such as "red", "green",...

### 43.77 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &gi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.78 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag  
**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input):** if copyTag==true, this is the size of the tagData  
**Returns :** a reference to the added EntityTag

#### 43.79 deleteTag

**int**  
**deleteTag( const EntityTypeEnum entityType, const int entityIndex,**  
**const EntityTag &tagToDelete )**

**Purpose:** delete an EntityTag from the mesh  
**entityType (input) :** the EntityTypeEnum of the entity  
**entityIndex (input):** the index of the entity  
**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

#### 43.80 deleteTag

**int**  
**deleteTag( const EntityTypeEnum entityType, const int entityIndex,**  
**const std::string tagToDelete )**

**Purpose:** delete an EntityTag from the mesh  
**entityType (input) :** the EntityTypeEnum of the entity  
**entityIndex (input):** the index of the entity  
**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

#### 43.81 hasTag

**bool**  
**hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )**

**Purpose:** check to see if an entity has a particular tag  
**entityType (input) :** the EntityTypeEnum of the entity  
**entityIndex (input):** the index of the entity  
**tag (input):** a string specifying the name of the tag in question  
**Returns :** true if the tag exists on the entity

### 43.82 getTag

```
EntityTag &
getTag( const EntityTypeEnum entityType,
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input):** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**Returns :** the tag requested

**Throws :** TagError if the tag is not found

### 43.83 getTagData

```
void *
getTagData( const EntityTypeEnum entityType, const int entityIndex,
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input):** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** NULL if the tag did not exist

### 43.84 setTagData

```
int
setTagData( const EntityTypeEnum entityType, const int entityIndex,
            const std::string tagName,
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input):** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**data (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** 0 if successfull

### 43.85 maintainTagToEntityMap

```
void  
maintainTagToEntityMap( bool v )
```

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input) :** if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

### 43.86 maintainsTagToEntityMap

```
bool  
maintainsTagToEntityMap( ) const
```

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

```
=====  
/ compare the vertices of an entity to a list of vertices, return true if the list specifies the entity //  
FALSE : no entities of this type created yet! // FALSE : invalid entity id given! // FALSE : the  
number of vertices do not match in each entity // FALSE : number of vertices do not match! // two  
entities are the same if thier vertices are the same, note that the ordering can // be reversed. //  
first find the starting point for each entity // the starting point is the lowest vertex id // FALSE :  
minimum vertex index does not match // now check the vertices in the current order // now check  
in the opposite direction (only the previous did not work!) / setAsGhost takes an entity and adjusts  
the data structures to make it a ghost // if the entity mask array is there (if not build it?) set the  
mask // now add the info as a tag // note this is a simple tag; the only data is the index "entity"  
// later we may allow construction using connectivity info // vertices do have an "orientation"  
relative to thier edges, the lowest vertex index is +ive // later we may allow construction using  
connectivity info / connectivityBuilder directs the construction of the connectivity arrays, it returns  
true if successfull if ( !entities[to] ) return false; // we don't have enough information // there is no  
downward from here! // XXX else add generic downward builder here! /// we always have this  
if there are Regions // XXX else add generic downward builder here! // XXX else add generic  
downward builder here! / specifyConnectivity tells the mapping to use the given connectivity  
information rather than building it / delete specific connectivity information / delete all connectivity  
information for a specific entity type / delete ALL the connectivity information
```

### 43.87 getColour

```
aString  
getColour( ) const
```

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.88 setColour

```
int  
setColour( const aString & colour )
```

**Purpose:** Set the colour for the grid.

**colour (input) :** the name of the colour such as "red", "green",...

### 43.89 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &ggi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.90 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.91 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

### 43.92 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

### 43.93 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.94 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.95 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.96 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.97 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.98 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

---

```
/ compare the vertices of an entity to a list of vertices, return true if the list specifies the entity //  

FALSE : no entities of this type created yet! // FALSE : invalid entity id given! // FALSE : the  

number of vertices do not match in each entity // FALSE : number of vertices do not match! // two  

entities are the same if thier vertices are the same, note that the ordering can // be reversed. //  

first find the starting point for each entity // the starting point is the lowest vertex id // FALSE :  

minimum vertex index does not match // now check the vertices in the current order // now check  

in the opposite direction (only the previous did not work!) / setAsGhost takes an entity and adjusts  

the data structures to make it a ghost // if the entity mask array is there (if not build it?) set the  

mask // now add the info as a tag // note this is a simple tag; the only data is the index "entity" //  

later we may allow construction using connectivity info // vertices do have an "orientation" relative  

to thier edges, the lowest vertex index is +ive // later we may allow construction using connectivity  

info type;j=domainDimension ) // *wdh* 030108 : there are no regions in 2D / connectivityBuilder  

directs the construction of the connectivity arrays, it returns true if successfull if ( !entities[to] )  

return false; // we don't have enough information // there is no downward from here! // XXX  

else add generic downward builder here! // we always have this if there are Regions // XXX  

else add generic downward builder here! // XXX else add generic downward builder here! /  

specifyConnectivity tells the mapping to use the given connectivity information rather than building  

it / delete specific connectivity information / delete all connectivity information for a specific entity  

type / delete ALL the connectivity information
```

### 43.99 getColour

```
aString  
getColour( ) const
```

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.100 setColour

```
int  
setColour( const aString & colour )
```

**Purpose:** Set the colour for the grid.

**colour (input) :** the name of the colour such as "red", "green",...

### 43.101 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &gi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.102 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.103 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

#### 43.104 deleteTag

**int**

```
deleteTag( const EntityTypeEnum entityType, const int entityIndex,
            const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

#### 43.105 hasTag

**bool**

```
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** true if the tag exists on the entity

#### 43.106 getTag

**EntityTag &**

```
getTag( const EntityTypeEnum entityType,
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**Returns :** the tag requested

**Throws :** TagError if the tag is not found

### 43.107 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** NULL if the tag did not exist

### 43.108 setTagData

**int**

```
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**data (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** 0 if successfull

### 43.109 maintainTagToEntityMap

```
void  
maintainTagToEntityMap( bool v )
```

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input) :** if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

### 43.110 maintainsTagToEntityMap

```
bool  
maintainsTagToEntityMap( ) const
```

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

```
=====  
/ compare the vertices of an entity to a list of vertices, return true if the list specifies the entity  
// FALSE : no entities of this type created yet! // FALSE : invalid entity id given! // FALSE :  
the number of vertices do not match in each entity // FALSE : number of vertices do not match!  
// two entities are the same if thier vertices are the same, note that the ordering can // be  
reversed. // first find the starting point for each entity // the starting point is the lowest vertex  
id // FALSE : minimum vertex index does not match // now check the vertices in the current  
order // now check in the opposite direction (only the previous did not work!) / setAsGhost  
takes an entity and adjusts the data structures to make it a ghost // if the entity mask array is  
there (if not build it?) set the mask // now add the info as a tag // note this is a simple tag;  
the only data is the index "entity" // later we may allow construction using connectivity info  
// vertices do have an "orientation" relative to thier edges, the lowest vertex index is +ive //  
link to the old connectivity // later we may allow construction using connectivity info // link to  
the old connectivity type;j=domainDimension ) // *wdh* 030108 : there are no regions in 2D /  
connectivityBuilder directs the construction of the connectivity arrays, it returns true if successfull  
if ( !entities[to] ) return false; // we don't have enough information // there is no downward from  
here! // XXX else add generic downward builder here! // we always have this if there are Regions  
// XXX else add generic downward builder here! // XXX else add generic downward builder here! /  
specifyConnectivity tells the mapping to use the given connectivity information rather than building  
it / delete specific connectivity information / delete all connectivity information for a specific entity  
type / delete ALL the connectivity information / expand the ghost boundary by a layer // (*en-  
tities[domainDimension]).display("entities before ghost"); //cout;j;"there are "jjnGhostLayersjjendl;  
// cout;j;bdyEntTagjjendl;j;bdyVertTagjjendl;j;entTagjjendl; // cout;j;"adding "jjnNewCellsjj;" new  
cells and "jjnNewVertsjj;" new verts"jjendl; std::string extrudedVertexTag = "extruded vertex";  
// we will use this to keep track of new vertex ids reserve(Vertex,size(Vertex)+nNewVerts); //  
reserve some space, keep some extra for later use. // first compute the normal and average edge  
length // get the sign for the normal from the adjacency information // make the entity and  
vertices // newEnt(n) = *vert; // cout;j;"new entity has id "jjnewEntIndexjjendl;j;newEnt;jjendl;  
// cout;j;"size of entity storage "jj(*entities[domainDimension]).getLength(0)jjendl; // (*en-  
tities[domainDimension]).display("entities"); // for now, force a rebuild of the connectivity when  
needed next (so, what happens to the tags...) // deleteConnectivity(EntityTypeEnum(e)); // make  
old and new connectivities consistent // this will retag the boundaries in the new connectivity //  
loop through the new ghost entities and add any new, lowerD entities. // cout;j;"before reserve,  
capacity is "jjcapacity(type)jjendl; // what about the connectivity arrays ... ??? // cout;j;"reserve to  
new size of "jjcapacity(type)jjendl; // ok, here we could use the data structure holes but lets make  
it simple for now... // what about the connectivity arrays ... ???
```

### 43.111 getColour

```
aString  
getColour( ) const
```

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.112 setColour

```
int  
setColour( const aString & colour )
```

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.113 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &gi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.114 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: name to give the tag instance

**tagData (input)**: data stored by the tag

**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input)**: if copyTag==true, this is the size of the tagData

**Returns** : a reference to the added EntityTag

### 43.115 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a reference to a tag specifying the deletion

**Returns** : 0 if successfull

### 43.116 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a string specifying the name of the tag to delete

**Returns** : 0 if successfull

### 43.117 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.118 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.119 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** NULL if the tag did not exist

#### 43.120 setTagData

**int**

```
setTagData( const EntityTypeEnum entityType, const int entityIndex,
            const std::string tagName,
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**data (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** 0 if successfull

#### 43.121 maintainTagToEntityMap

**void**

```
maintainTagToEntityMap( bool v )
```

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input) :** if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.122 maintainsTagToEntityMap

**bool**

```
maintainsTagToEntityMap( ) const
```

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

```
=====
/ compare the vertices of an entity to a list of vertices, return true if the list specifies the entity
// FALSE : no entities of this type created yet! // FALSE : invalid entity id given! // FALSE :
the number of vertices do not match in each entity // FALSE : number of vertices do not match!
// two entities are the same if thier vertices are the same, note that the ordering can // be
reversed. // first find the starting point for each entity // the starting point is the lowest vertex
id // FALSE : minimum vertex index does not match // now check the vertices in the current
order // now check in the opposite direction (only the previous did not work!) / setAsGhost
takes an entity and adjusts the data structures to make it a ghost // if the entity mask array is
there (if not build it?) set the mask // now add the info as a tag // note this is a simple tag;
```

```

the only data is the index "entity" // later we may allow construction using connectivity info
// vertices do have an "orientation" relative to thier edges, the lowest vertex index is +ive //
link to the old connectivity // later we may allow construction using connectivity info // link to
the old connectivity typej=domainDimension ) // *wdh* 030108 : there are no regions in 2D /
connectivityBuilder directs the construction of the connectivity arrays, it returns true if successfull
if ( !entities[to] ) return false; // we don't have enough information // there is no downward from
here! // XXX else add generic downward builder here! /// we always have this if there are Regions
// XXX else add generic downward builder here! // XXX else add generic downward builder here! /
specifyConnectivity tells the mapping to use the given connectivity information rather than building
it / delete specific connectivity information / delete all connectivity information for a specific entity
type / delete ALL the connectivity information / expand the ghost boundary by a layer // (*en-
tities[domainDimension]).display("entities before ghost"); //coutjj"there are "jjnGhostLayersjjendl;
// coutjjbdyEntTagjjendljjbdyVertTagjjendljjentTagjjendl; // coutjj"adding "jjnNewCellsjj" new
cells and "jjnNewVertsjj" new verts"jjendl; std::string extrudedVertexTag = "extruded vertex";
// we will use this to keep track of new vertex ids reserve(Vertex.size(Vertex)+nNewVerts); //
reserve some space, keep some extra for later use. // first compute the normal and average edge
length // get the sign for the normal from the adjacency information // make the entity and
vertices // newEnt(n) = *vert; // coutjj"new entity has id "jjnewEntIndexjjendljjnewEntjjendl;
// coutjj"size of entity storage "jj(*entities[domainDimension]).getLength(0)jjendl;; // (*en-
tities[domainDimension]).display("entities"); // for now, force a rebuild of the connectivity when
needed next (so, what happens to the tags...) // deleteConnectivity(EntityTypeEnum(e)); // make
old and new connectivities consistent // this will retag the boundaries in the new connectivity //
loop through the new ghost entities and add any new, lowerD entities. // coutjj"before reserve,
capacity is "jjcapacity(type)jjendl; // what about the connectivity arrays ... ??? // coutjj"reserve to
new size of "jjcapacity(type)jjendl; // ok, here we could use the data structure holes but lets make
it simple for now... // what about the connectivity arrays ... ???

```

### 43.123 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value :** the name of the colour.

### 43.124 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.125 eraseUnstructuredMapping

**void**  
**eraseUnstructuredMapping(GenericGraphicsInterface &gi)**

**Description:** purge all display lists for the unstructured mapping

### 43.126 addTag

```
EntityTag &
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string
tagName,
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: name to give the tag instance

**tagData (input)**: data stored by the tag

**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input)**: if copyTag==true, this is the size of the tagData

**Returns** : a reference to the added EntityTag

### 43.127 deleteTag

```
int
deleteTag( const EntityTypeEnum entityType, const int entityIndex,
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a reference to a tag specifying the deletion

**Returns** : 0 if successfull

### 43.128 deleteTag

```
int
deleteTag( const EntityTypeEnum entityType, const int entityIndex,
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a string specifying the name of the tag to delete

**Returns** : 0 if successfull

### 43.129 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.130 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.131 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.132 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.133 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.134 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.135 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value** : the name of the colour.

#### 43.136 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

#### 43.137 eraseUnstructuredMapping

**void**  
**eraseUnstructuredMapping(GenericGraphicsInterface &gi)**

**Description:** purge all display lists for the unstructured mapping

### 43.138 addTag

```
EntityTag &
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string
tagName,
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: name to give the tag instance

**tagData (input)**: data stored by the tag

**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input)**: if copyTag==true, this is the size of the tagData

**Returns** : a reference to the added EntityTag

### 43.139 deleteTag

```
int
deleteTag( const EntityTypeEnum entityType, const int entityIndex,
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a reference to a tag specifying the deletion

**Returns** : 0 if successfull

### 43.140 deleteTag

```
int
deleteTag( const EntityTypeEnum entityType, const int entityIndex,
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagToDelete (input)**: a string specifying the name of the tag to delete

**Returns** : 0 if successfull

#### 43.141 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** true if the tag exists on the entity

#### 43.142 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** a string specifying the name of the tag in question

**Returns :** the tag requested

**Throws :** TagError if the tag is not found

#### 43.143 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tag (input):** a string specifying the name of the tag in question

**Returns :** NULL if the tag did not exist

#### 43.144 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.145 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.146 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.147 entitiesAreEquivalent

**bool**  
**entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices)**

**Purpose:** compare the vertices of an entity to a list of vertices, return true if the list specifies the entity

#### 43.148 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value** : the name of the colour.

#### 43.149 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.150 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &ggi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.151 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.152 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

### 43.153 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

### 43.154 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.155 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.156 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.157 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.158 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.159 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.160 entitiesAreEquivalent

**bool**  
**entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices)**

**Purpose:** compare the vertices of an entity to a list of vertices, return true if the list specifies the entity

#### 43.161 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value** : the name of the colour.

#### 43.162 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.163 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &ggi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.164 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.165 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

### 43.166 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

### 43.167 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.168 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.169 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.170 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.171 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.172 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.173 entitiesAreEquivalent

**bool**  
**entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices)**

**Purpose:** compare the vertices of an entity to a list of vertices, return true if the list specifies the entity

#### 43.174 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value** : the name of the colour.

#### 43.175 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.176 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &ggi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.177 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.178 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

### 43.179 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

### 43.180 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.181 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.182 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.183 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.184 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.185 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.186 entitiesAreEquivalent

**bool**  
**entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices)**

**Purpose:** compare the vertices of an entity to a list of vertices, return true if the list specifies the entity

#### 43.187 getColour

**aString**  
**getColour( ) const**

**Purpose:** Get the colour of the grid.

**Return value** : the name of the colour.

#### 43.188 setColour

**int**  
**setColour( const aString & colour )**

**Purpose:** Set the colour for the grid.

**colour (input)** : the name of the colour such as "red", "green",...

### 43.189 eraseUnstructuredMapping

```
void  
eraseUnstructuredMapping(GenericGraphicsInterface &ggi)
```

**Description:** purge all display lists for the unstructured mapping

### 43.190 addTag

```
EntityTag &  
addTag( const EntityTypeEnum entityType, const int entityIndex, const std::string  
tagName,  
        const void *tagData, const bool copyTag, const int tagSize )
```

**Purpose:** add an EntityTag to a specific entity in the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagName (input):** name to give the tag instance

**tagData (input):** data stored by the tag

**copyTag (input):** deep copy tagData if copyTag==true, shallow copy if false

**tagSize (input):** if copyTag==true, this is the size of the tagData

**Returns :** a reference to the added EntityTag

### 43.191 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const EntityTag &tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a reference to a tag specifying the deletion

**Returns :** 0 if successfull

### 43.192 deleteTag

```
int  
deleteTag( const EntityTypeEnum entityType, const int entityIndex,  
           const std::string tagToDelete )
```

**Purpose:** delete an EntityTag from the mesh

**entityType (input) :** the EntityTypeEnum of the entity

**entityIndex (input):** the index of the entity

**tagToDelete (input):** a string specifying the name of the tag to delete

**Returns :** 0 if successfull

### 43.193 hasTag

```
bool  
hasTag( const EntityTypeEnum entityType, const int entityIndex, const std::string tag )
```

**Purpose:** check to see if an entity has a particular tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : true if the tag exists on the entity

### 43.194 getTag

```
EntityTag &  
getTag( const EntityTypeEnum entityType,  
        const int entityIndex, const std::string tagName)
```

**Purpose:** obtain a reference to a tag on a specific entity

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tagName (input)**: a string specifying the name of the tag in question

**Returns** : the tag requested

**Throws** : TagError if the tag is not found

### 43.195 getTagData

```
void *  
getTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tag )
```

**Purpose:** obtain the the data in a tag

**entityType (input)** : the EntityTypeEnum of the entity

**entityIndex (input)**: the index of the entity

**tag (input)**: a string specifying the name of the tag in question

**Returns** : NULL if the tag did not exist

### 43.196 setTagData

```
int  
setTagData( const EntityTypeEnum entityType, const int entityIndex,  
            const std::string tagName,  
            const void *data, const bool copyData, const int tagSize )
```

**Purpose:** set the data in an existing tag

**entityType (input)** : the EntityTypeEnum of the entity  
**entityIndex (input)**: the index of the entity  
**tagName (input)**: a string specifying the name of the tag in question  
**data (input)**: data stored by the tag  
**copyTag (input)**: deep copy tagData if copyTag==true, shallow copy if false  
**tagSize (input)**: if copyTag==true, this is the size of the tagData  
**Returns** : 0 if successfull

#### 43.197 maintainTagToEntityMap

**void**  
**maintainTagToEntityMap( bool v )**

**Purpose:** turn on/off maintainance of the mapping from tags to thier entities

**v (input)** : if true turn on the tag to entity mapping, if false turn it off

**Note:** If v==true, this method will build the mapping. If false, it will destroy the mapping

#### 43.198 maintainsTagToEntityMap

**bool**  
**maintainsTagToEntityMap( ) const**

**Purpose:** return true if the Mapping maintains the list of entities with a given tag

#### 43.199 entitiesAreEquivalent

**bool**  
**entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &vertices)**

**Purpose:** compare the vertices of an entity to a list of vertices, return true if the list specifies the entity

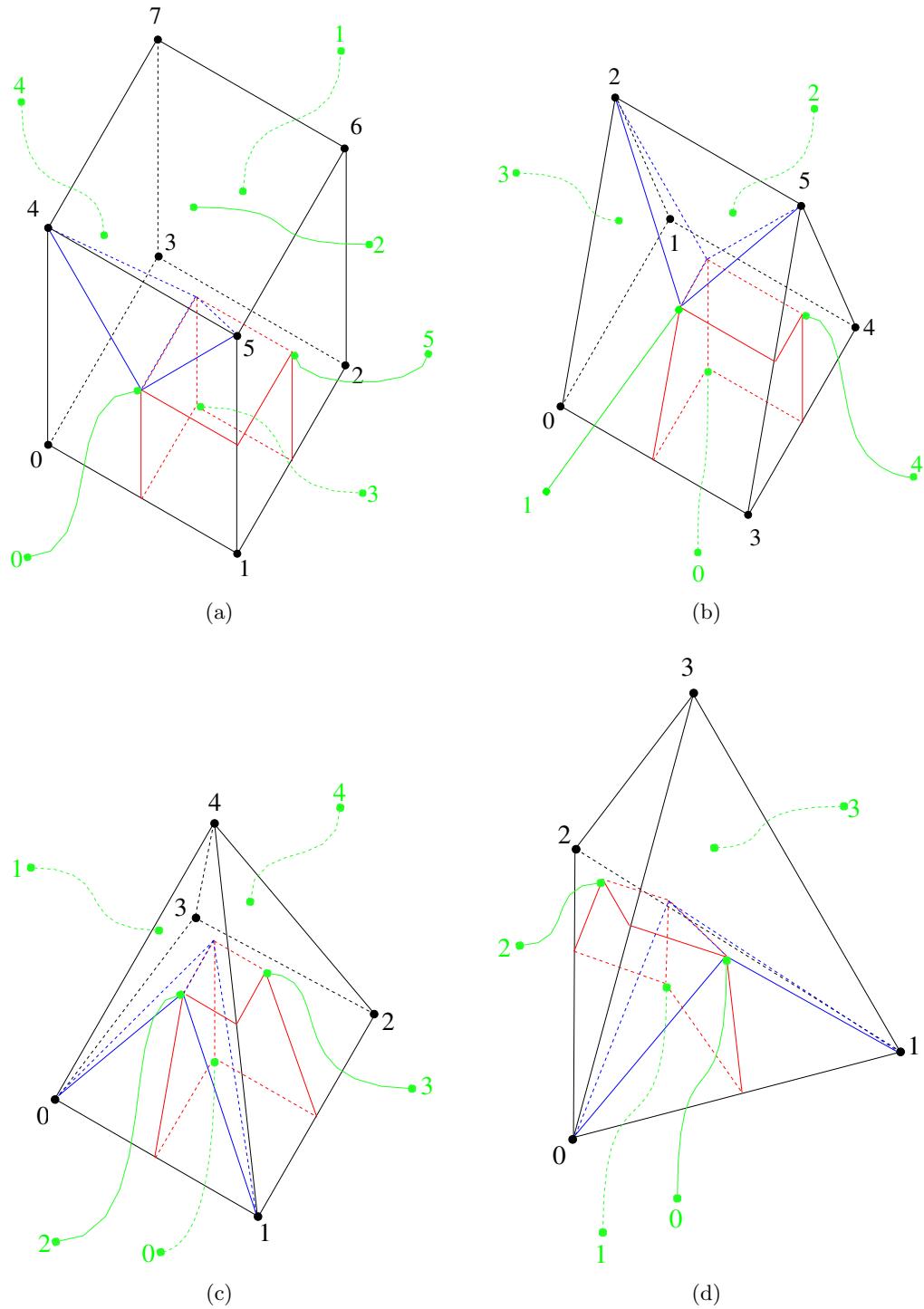


Figure 32: Canonical orderings for the 3D finite element zoo: (a) hexahedra; (b) triangle prisms; (c) pyramids; (d) tetrahedra. Black indicates vertex numbers, green indicates face indices. Simple corners are drawn in red and sides are in blue.

## 44 Class Fraction

This class is used to define “fractions”, the ratio of two integers. Fractions can represent infinity and -infinity with a zero numerator and nonzero denominator. Thus  $1/0$  is infinity and  $-1/0$  is -infinity. We define  $2/0$  be greater than  $1/0$ .

### 44.1 Constructors

```
Fraction( int n, int d=1 )      define a fraction, n = numerator, d = denominator
```

Note that we do not know how to construct a fraction from a real number.

### 44.2 Member Functions

The relational operators  $\leq$ ,  $<$ ,  $\geq$ ,  $>$  and  $==$  are defined for the comparison of two fractions or a fraction and a real number. In addition, the arithmetic operators  $+$ ,  $-$ ,  $*$  and  $/$  are defined for two objects of type Fraction (or a Fraction and a real or int).

NOTE: By definition the result of the operators  $+$ ,  $-$ ,  $*$ , or  $/$  between a Fraction and a real results in a real.

Here are the member functions that can be used to access the numerator and denominator

```
int setNumerator()      set the numerator  
int setDenominator()   set the denominator  
int getNumerator()     get the numerator  
int getDenominator()   get the denominator
```

## 45 Class Bound

A bound is defined as a real number, a fraction or null. The bound class implements the bound and supplies functions for comparing bounds. Bounds allow rational numbers to be specified precisely. Bounds can represent infinity and -infinity by fractions with a zero numerator and nonzero denominator. Thus  $1/0$  is infinity and  $-1/0$  is -infinity. We define  $2/0$  be greater than  $1/0$ .

### 45.1 enum types

```
enum boundType{ realNumber, fraction, null };
```

### 45.2 Constructors

```
Bound()                  default constructor, boundType=null  
Bound( real x0 )        define a bound from a real number  
Bound( int i )          define a bound from a int  
Bound( Fraction f0 )   define a bound from a fraction
```

### 45.3 Member Functions

The relational operators  $\leq$ ,  $<$ ,  $\geq$ ,  $>$  and  $==$  are defined for the comparison of two bounds or a bound and a real number, or a bound and a fraction. In addition the arithmetic operators  $+$ ,  $-$ ,  $*$  and  $/$  are defined for two objects of type Bound. There are also member functions to assign and retrieve values

void set( real value )	assign a real value to the bound
void set( int value )	assign an integer value to the bound
void set( int n, int d )	assign a numerator and denominator
void get( boundType bt, real x, Fraction f )	get boundType and value

## 46 Class Triangle

The Triangle class is used to represent a triangle in three dimensional space. It has a function for determining how two triangles intersect that is used by the IntersectionMapping class.

### 46.1 Constructor

**Triangle()**

**Purpose:** Default Constructor, make a default triangle with vertices (0,0,0), (1,0,0), (0,1,0)

### 46.2 Constructor( const real x1[],x2[],x3[] )

**Triangle( const real x1-[3], const real x2-[3], const real x3-[3] )**

**Purpose:** Create a triangle with vertices x1,x2,x3

**x1,x2,x3 (input) :** the three vertices of the triangle

### 46.3 Constructor( const RealArray & x1,x2,x3 )

**Triangle( const RealArray & x1-, const RealArray & x2-, const RealArray & x3- )**

**Purpose:** Create a triangle with vertices x1,x2,x3

**x1,x2,x3 (input) :** the three vertices of the triangle

### 46.4 Constructor(grid)

**Triangle(const realArray & grid,  
         const int & i1,  
         const int & i2,  
         const int & i3,  
         const int & choice =0,  
         const int & axis =axis1)**

**Purpose:** Build a triangle from a quadrilateral on the face of a grid grid, This constructor just calls the corresponding **setVertices** function. See the comments there.

**46.5 setVertices(const real x1,x2,x3)**

**void**

**setVertices( const real x1-[3], const real x2-[3], const real x3-[3] )**

**Purpose:** Assign the vertices to a triangle.

**x1,x2,x3 (input) :** the three vertices of the triangle

**46.6 setVertices( const RealArray & x1,x2,x3)**

**void**

**setVertices( const RealArray & x1-, const RealArray & x2-, const RealArray & x3- )**

**Purpose:** Assign the vertices to a triangle.

**x1,x2,x3 (input) :** the three vertices of the triangle

**46.7 setVertices**

**void**

**setVertices(const realArray & grid,**  
    **const int & i1,**  
    **const int & i2,**  
    **const int & i3,**  
    **const int & choice =0,**  
    **const int & axis =axis3)**

**Purpose:** Form a triangle from a quadrilateral on the face of a grid grid, there are six possible choices.

**grid (input) :** an array containing the four points  $\text{grid}(i1+m, i2+n, i3, 0:2)$ ,  $m=0,1$ ,  $n=0,1$ .

**i1,i2,i3 (input) :** indicates which quadrilateral to use

**choice, axis (input) :** These define which of 6 possible triangles to choose:

**choice=0, axis=axis3(==2) :** use points  $(i1, i2, i3)$ ,  $(i1+1, i2, i3)$ ,  $(i1, i2+1, i3)$ . Lower left triangle in the plane  $i3 == \text{constant}$ .

**choice=1, axis=axis3(==2) :** use points  $(i1+1, i2+1, i3)$ ,  $(i1, i2+1, i3)$ ,  $(i1+1, i2, i3)$ . Upper right triangle in the plane  $i3 == \text{constant}$ .

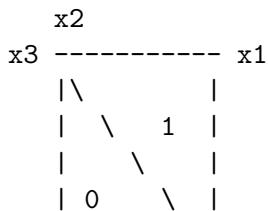
**choice=0, axis=axis2(==1) :** use points  $(i1, i2, i3)$ ,  $(i1, i2, i3+1)$ ,  $(i1+1, i2, i3)$ .

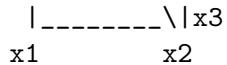
**choice=1, axis=axis2(==1) :** use points  $(i1+1, i2, i3+1)$ ,  $(i1+1, i2, i3)$ ,  $(i1, i2, i3+1)$ .

**choice=0, axis=axis1(==0) :** use points  $(i1, i2, i3)$ ,  $(i1, i2+1, i3)$ ,  $(i1, i2, i3+1)$ .

**choice=1, axis=axis1(==0) :** use points  $(i1, i2+1, i3+1)$ ,  $(i1, i2, i3+1)$ ,  $(i1, i2+1, i3)$ .

The figure below shows the two choices for  $\text{axis}=\text{axis}3$ :





## 46.8 area

**real**  
**area() const**

**Purpose:** return the area of the triangle

## 46.9 display

**void**  
**display(const aString & label =blankString) const**

**Purpose:** print out the vertices and the normal.

## 46.10 tetrahedralVolume

**double**  
**tetrahedralVolume(const real a[], const real b[], const real c[], const real d[]) const**

**Purpose:** Return the approximate volume (actually 6 times the volume) of the tretrahedra formed by the points (a,b,c,d)

## 46.11 intersects

**bool**  
**intersects(Triangle & tri, real xi1[3], real xi2[3] ) const**

**Purpose:** Determine if this triangle intersect another.

**tri (input) :** intersect with this triangle.

**xi1, xi2 (output) :** if the return value is true then these are the endpoints of the line of intersection between the two triangles.

**Return value :** TRUE if the triangles intersect, false otherwise.

## 46.12 intersects

**bool**  
**intersects(Triangle & triangle, RealArray & xi1, RealArray & xi2 ) const**

**Purpose:** Determine if this triangle intersect another.

**tri (input) :** intersect with this triangle.

**xi1, xi2 (output) :** if the return vaule is true then these are the endpoints of the line of intersection between the two triangles.

**Return value :** TRUE if the triangles intersect, false otherwise.

#### 46.13 intersects

```
bool  
intersects(real x[3], real xi[3] ) const
```

**Purpose:** Determine if this triangle intersects a ray starting at the point x[] and extending to y=+infinity.

**x (input)** : find the intersection with a vertical ray starting at this point.

**xi (output)** : if the return value is true then this is the intersection point.

**Return value** : TRUE if the ray intersects the triangle, false otherwise.

#### 46.14 intersects

```
bool  
intersects(RealArray & x, RealArray & xi ) const
```

**Purpose:** Determine if this triangle intersects a line starting at the point x and extending to y=+infinity.

**x (input)** : find the intersection with a vertical ray starting at this point.

**xi (output)** : if the return value is true then this is the intersection point.

**Return value** : TRUE if the ray intersects the triangle, false otherwise.

#### 46.15 intersects

```
bool  
intersects(real x[3], real xi[3], real b0[3], real b1[3], real b2[3] ) const
```

**Purpose:** Determine if this triangle intersects a ray starting at the point x[] and extending in the direction b1

**x (input)** : find the intersection with a vertical ray starting at this point.

**xi (output)** : if the return value is true then this is the intersection point.

**b0,b1,b2** : these vectors form an ortho-normal set

**Return value** : TRUE if the ray intersects the triangle, false otherwise.

#### 46.16 intersects

```
bool  
intersects(RealArray & x, RealArray & xi, real b0[3], real b1[3], real b2[3] ) const
```

**Purpose:** Determine if this triangle intersects a ray starting at the point x and extending in the direction b1

**x (input)** : find the intersection with a vertical ray starting at this point.

**xi (output)** : if the return value is true then this is the intersection point.

**b0,b1,b2** : these vectors form an ortho-normal set

**Return value** : TRUE if the ray intersects the triangle, false otherwise.

#### 46.17 getRelativeCoordinates

```
int  
getRelativeCoordinates( const real x[3],  
                        real & alpha1,  
                        real & alpha2,  
                        const bool & shouldBeInside =TRUE) const
```

**Description:** Determine the coordinates of the point x with respect to this triangle. I.e. solve for alpha1,alpha2 where  $x - x_1 = \alpha_1 * v_1 + \alpha_2 * v_2$

where  $v_1 = x_2 - x_1$  and  $v_2 = x_3 - x_1$  are two vectors from the sides of the triangle,  $(x_1,x_2,x_3)$  Solve

$$\begin{aligned} [v_1.v_1 \ v_1.v_2] [ \alpha_1 ] &= [v_1.x] \\ [v_1.v_2 \ v_2.v_2] [ \alpha_2 ] &= [v_2.x] \\ \alpha_1 &= (v_1.x * v_2.v_2 - v_2.x * v_1.v_2) / (v_1.v_1 * v_2.v_2 - v_1.v_2 * v_1.v_2) \\ \alpha_2 &= (v_1.x * v_2.v_2 - v_2.x * v_1.v_2) / (v_1.v_1 * v_2.v_2 - v_1.v_2 * v_1.v_2) \end{aligned}$$

**x (input)** : find coordinates of this point.

**alpha1, alpha2 (output)** : relative coordinates.

**shouldBeInside (input)** : if true, this routine will print out a message if alpha1 or alpha are not in the range [0,1] ( +/- epsilon), AND return a value of 1

**Return value** : 0 on sucess, 1 if shouldBeInside==TRUE and the point is not inside.

## References

- [1] W. HENSHAW, *The overture hyperbolic grid generator, user guide, version 1.0*, Research Report UCRL-MA-??, Lawrence Livermore National Laboratory, 1999.

# Index

- airfoil mapping, 63
- annulus mapping, 61
- axisymmetric, 171
- basicInverse, 45
- bathymetry, 92
- body of revolution, 171
- box mapping, 67
- cartesian space, 20
- circle mapping, 69
- compose mapping, 71
- composite surface mapping, 72
- Coon's patch, 215
- coordinate singularity, 47
- coordinate systems, 23, 47
- cross-section mapping, 75
- cylinder mapping, 81
- data-point mapping, 85
- depth mapping, 92
- domainDimension, 20
- ellipse, 69
- elliptic mapping, 96
- fillet mapping, 103
- getIndex, 45
- hyperbola, 158
- hyperboloid, 158
- interpolation
  - transfinite, 215
- intersecting surfaces, 115
- intersection mapping, 108
- inverse
  - approximate global inverse, 49
  - exact local inverse, 55
- inverseMap, 45
- join mapping, 115
- line mapping, 120
- lofted surface, 122
- map, 44
- Mapping
  - AirfoilMapping, 63
  - AnnulusMapping, 61
- BoxMapping, 67
- CircleMapping, 69
- ComposeMapping, 71
- CompositeSurface, 72
- CrossSectionMapping, 75
- CylinderMapping, 81
- DataPointMapping, 85
- DepthMapping, 92
- discrete mapping, 85
- EllipticTransform, 96
- external, 85
- FilletMapping, 103
- IntersectionMapping, 108
- JoinMapping, 115
- LineMapping, 120
- LoftedSurfaceMapping, 122
- make a 3D mapping by extruding a 2D mapping, 210
- MatrixMapping, 126
- MatrixTransformMapping, 129
- NormalMapping, 131
- NurbsMapping, 133
- OrthographicTransformMapping, 153
- PlaneMapping, 157
- plot3d, 85
- QuadraticMapping, 158
- ReductionMapping, 161
- ReparameterizationTransform, 165
- restrict an existing mapping to face or line, 161
- RestrictionMapping, 169
- RevolutionMapping, 171
- RocketMapping, 178
- rotate, 129
- scale, 129
- shift, 129
- SmoothedPolygonMapping, 181
- SphereMapping, 185
- SplineMapping, 188
- SquareMapping, 195
- StretchedSquareMapping, 208
- StretchMapping, 196
- StretchTransformMapping, 209
- SweepMapping, 210
- TFIMapping, 215
- translate, 129
- TrimmedMapping, 222
- mapping parameters, 48

matrix mapping, 126  
matrix transform mapping, 129  
  
NACA, 63  
normal mapping, 131  
Nurbs  
    trimmed, 222  
nurbs mapping, 133  
  
offset mapping, 150  
orthographic, 165  
orthographic mapping, 153  
overlapping round, 150  
  
parabola, 158  
paraboloid, 158  
parameter space, 20  
patched surface, 72  
periodic mappings, 44  
periodicity, 22  
plane mapping, 157  
polar singularity  
    remove, 165  
  
quadratic mapping, 158  
  
rangeDimension, 20  
rectangle, 195  
reduction mapping, 161  
registering a new mapping, 58  
reorder domain coordinates, 164, 165  
reparameterization mapping, 165  
restrict a mapping to a sub-rectangle, 165  
restriction mapping, 169  
revolution mapping, 171  
rhombus, 157  
Rocket mapping, 178  
  
smoothed-polygon mapping, 181  
sphere mapping, 185  
spline  
    curve, 133, 150, 188  
    shape preserving, 188  
    surface, 133, 150  
    tension, 188  
spline mapping, 188  
square mapping, 195  
stretch mapping, 196  
stretch-transform mapping, 209  
stretched-square mapping, 208  
stretching  
    exponential, 199  
exponential blend, 199  
exponential to linear, 199  
hyperbolic tangent, 197  
    inverse hyperbolic tangent, 196  
sweep mapping, 210  
tfi mapping, 215  
transfinite interpolation, 215  
triangle, 294  
trimmed mapping, 222  
TSPACK, 188