BetaMesh User Guide

W. Ross McLendon

July 13, 2011

Contents

1 BetaMesh Layout **3**
1.1 Nodes . . . 3
1.2 Elements . . . 3
1.3 Orientations . . . 3
1.4 Mesh . . . 3
1.5 Utility Functions . . . 4

2 BetaMesh Objects and Functions **4**
2.1 Nodes . . . 4
2.2 Elements . . . 4
2.3 Orientations . . . 4
2.4 Mesh . . . 4
2.4.1 Default Constructor . . . 4
2.4.2 Copy Constructor . . . 5
2.4.3 read_beta_mesh_file . . . 5
2.4.4 make_grid_mesh . . . 5
2.4.5 make_triangle_mesh . . . 6
2.4.6 make_step_down_mesh . . . 6
2.4.7 write_beta_mesh_file . . . 7
2.4.8 write_beta_elemat_file . . . 7
2.4.9 write_beta_mangles_file . . . 7
2.4.10 write_beta_orientation_file . . . 7
2.4.11 write_VTK_mesh_file . . . 7
2.4.12 get_min_coords and get_max_coords . . . 7
2.4.13 get_num_nodes and get_num_elements . . . 7
2.4.14 clone_mesh . . . 8
2.4.15 extrude_2D_mesh . . . 8
2.4.16 extrude_2D_mesh_along_path . . . 8
2.4.17 read_beta_elemat_file . . . 8
2.4.18 read_beta_mangles_file . . . 8
2.4.19 read_beta_orientation_file . . . 8
2.4.20 translate_mesh . . . 8

2.4.21 scale_mesh . . . 9
2.4.22 mirror_mesh . . . 9
2.4.23 rotate_mesh . . . 9
2.4.24 remove_duplicate_nodes . . . 9
2.4.25 remove_unattached_nodes . . . 9
2.4.26 map_to_circle_to_square . . . 10
2.4.27 standardize_orientations . . . 10
2.4.28 remove_material . . . 10
2.4.29 remove_element . . . 10
2.4.30 remove_elements_by_loop . . . 10
2.4.31 remove_elements_by_bounding_box . . . 11
2.4.32 renumber_nodes_and_elements . . . 11
2.4.33 make_mesh_quadratic . . . 11
2.4.34 make_mesh_linear . . . 11
2.4.35 assign_material_number_to_all . . . 11
2.4.36 assign_mat_number_using_loop . . . 11
2.4.37 remap_material_number . . . 12
2.4.38 = operator . . . 12
2.4.39 += operator . . . 12
2.4.40 + operator . . . 12
2.5 TextileMesh . . . 12
2.5.1 void make_half_tow_cross_section . . . 12
2.5.2 void make_half_tow_cross_section_rounded . . . 13
2.5.3 make_3D_half_tow . . . 14
2.5.4 make_matrix_pocket_cross_section . . . 14
2.5.5 make_matrix_pocket . . . 15
2.5.6 make_PW_unit_cell . . . 15
2.6 Q3DLaminateMesh . . . 15
2.7 Utility Functions . . . 16
2.7.1 linspace . . . 16
2.7.2 bias_vector . . . 16

3 Using BetaMesh Standalone Executable **16**

4 BetaMesh in Python **17**
4.1 Setup . . . 17
4.2 Building the BetaMesh Module for Python . . . 17
4.3 Using BetaMesh in Python . . . 17

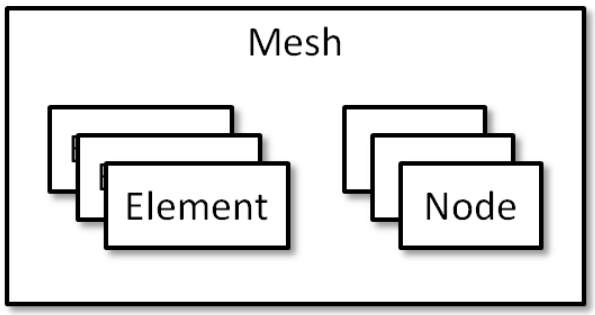

Figure 1: Layout of BetaMesh Mesh object

1 BetaMesh Layout

This section describes the structure of the BetaMesh mesh generation program.

1.1 Nodes

Nodes are a fundamental objects in BetaMesh. A node contains a vector of coordinates which describe its location. This vector can be 1D, 2D, or 3D depending on the dimensionality of the model. A node also contains a set of Element pointers which point to elements that the node is attached to. Finally, a node contains an integer which designates its number. This typically is not of concern until the mesh is written to a file.

1.2 Elements

Elements are objects in BetaMesh. An element contains a vector of node pointers which define its connectivity, as well as integers which desgnate the number of nodes in the element and the element's number. Additionally, the element contains an integer which defines its material type, which by default is initially set to -1. Finally, an element can optionally contain a vector of orientation objects which define material orientation at each node in the element.

1.3 Orientations

These are fundamental objects which track the orientation of material at a point. They contain three unit vectors which describe the local coordinate axes for the material system expressed in the global coordinate system. The contain a variety of methods for manipulating the orientation at a point.

1.4 Mesh

The Mesh (Figure 1) is the primary class used in BetaMesh. It contains a list of nodes and elements, and contains a variety of methods for creating, modifying, copying, combining, and outputting meshes and their associated material and orienation data. Mesh methods are the interface that a user has for modifying a mesh (as opposed to directly modifying nodes and elements). The Mesh object exists in an

object-oriented hierarchy. The base class is simply the "Mesh" class, and it contains basic functionality for generating and manipulating meshes. For the generation of meshes which are tailored to specific types of problems, additional classes have been inherited from the Mesh class. It is important to keep specialized mesh generation functionality out of the base Mesh class to maintain its ease of use. There are currently several classes which are inherited from the mesh class:

- TextileMesh
- Q3DLaminateMesh

1.5 Utility Functions

There are a number of functions included in the code which are helpful for quickly performing mathematical operations on vectors and also for generating specific types of vectors quickly (such as a vector of vectors forming an identity matrix, or a vector of n evenly-spaced values). These functions are called utility functions.

2 BetaMesh Objects and Functions

This section provides a detailed description of the various objects and functions in BetaMesh.

2.1 Nodes

Nodes are not intended to be directly accessed by the user and thus are not described in the documentation.

2.2 Elements

Elements are not intended to be directly accessed by the user and thus are not described in the documentation.

2.3 Orientations

Orientations are not intended to be directly accessed by the user and thus are not described in the documentation.

2.4 Mesh

The Mesh class is the base class for generating, modifying, and outputting meshes. Its methods are described in the following sub-sections.

2.4.1 Default Constructor

```
Mesh()
```

This constructor creates a mesh containing no Node or Element objects.

2.4.2 Copy Constructor

```
Mesh(const Mesh &Arg)
```

This constructor creates a mesh from an existing mesh. New node and element objects are created and all pointers in those objects are updated to point to the Node and Element objects in the new mesh.

2.4.3 read_beta_mesh_file

```
void read_beta_mesh_file(const std::string &FileName)
```

This method reads a plotter/beta style mesh file. It is intended to be used on an initially empty mesh.

2.4.4 make_grid_mesh

```
void make_grid_mesh(const std::vector<double> &Coord1Vals,
                    const std::vector<double> &Coord2Vals,
                    const bool &Quadratic = true)
```

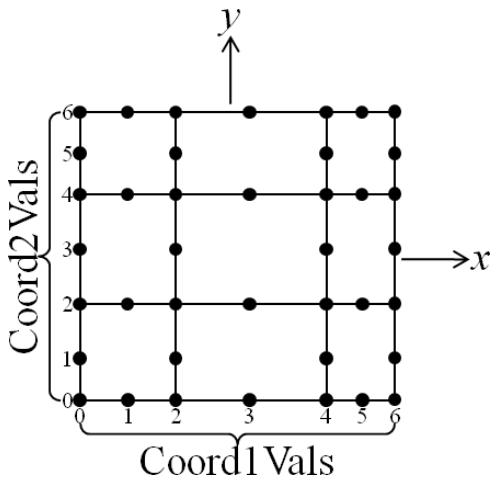

Figure 2: Mesh generated using make_grid_mesh

This method creates a grid of quadrilaterals with nodes located at the coordinates defined by the two vector arguments (Figure 2). The mesh can be either linear or quadratic (default is quadratic). Locations of midside nodes should be included in the vectors if a quadratic mesh is desired. Coord1Vals and Coord2Vals need not be the same length, but they must both contain at least two entries for linear grid meshes or an odd number of at least three entries for quadratic grid meshes. Note that the utility functions linspace (2.7.1) and bias_vector (2.7.2) can be quite helpful when used in conjunction with make_grid_mesh.

2.4.5 make_triangle_mesh

```
std::vector< std::vector<Node*> >
    make_triangle_mesh(const std::vector< std::vector< std::vector<double> > > &BoundaryCoords,
                       double minTriangleAngle = 28.6,
                       double maxElementArea = -1.0);
```

This method generates a trianglular mesh of input boundary coordinates and outputs pointers to the nodes that are on each input boundary. It uses the 3rd party code "Triangle" (the source is included with BetaMesh). Each coordinate is a double vector of length 2. The data is input as a collection of boundaries, where each boundary is defined by a sequence of coordinates. Coordinates should be input counterclockwise around the region to be meshed. Figure 3 shows an example of a mesh generated using make_triangle_mesh by inputting the coodinates around the circumference of a circle as the boundary coordinates.

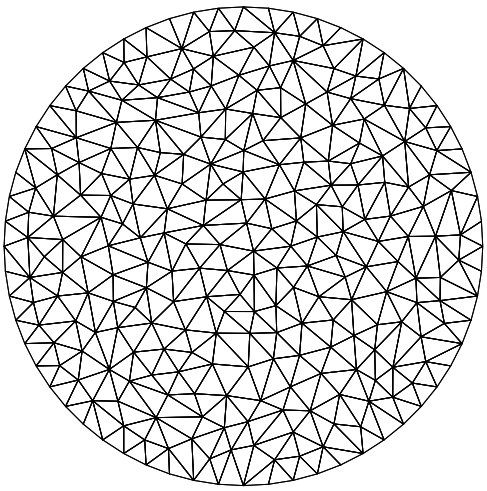

Figure 3: Mesh generated using make_triangle_mesh

2.4.6 make_step_down_mesh

```
std::vector<Element*> make_step_down_mesh(const std::vector<Node*> &BoundaryNodes1,
                                          const std::vector<Node*> &BoundaryNodes2,
                                          const int MatNumber = 0);
```

This method is used by the tow cross-section generators. It creates a symmetric (if the nodes are symmetric) mesh connecting the nodes of BoundaryNodes1 and BoundaryNodes2 using square and triangle elements. BoundaryNodes 1 and 2 should generally be nodes along a straight line, although this guideline doesn't necessarily have to be followed. Returns a vector of pointers to the elements which are added to the mesh by the method.

2.4.7 write_beta_mesh_file

```
void write_beta_mesh_file(const std::string &FileName)
```

This method writes the mesh to a Beta format mesh file with the input file name.

2.4.8 write_beta_elemat_file

```
void write_beta_elemat_file(const std::string &FileName) const
```

This method writes an elemat file for the mesh, provided that material data has been assigned.

2.4.9 write_beta_mangles_file

```
void write_beta_mangles_file(const std::string &FileName) const
```

This method writes a mangles file for the mesh. It will fail for any element that contains orientations which are not aligned to at least one of the (x, y, z) planes.

2.4.10 write_beta_orientation_file

```
void write_beta_orientation_file(const std::string &FileName) const
```

This method writes an orientation file for the mesh. Currently, it uses a 3-2-1 rotation sequence only and will fail if the 2 rotation is 90°. (This is a singularity for the 3-2-1 rotation sequence in which the 3 and 1 rotation cannot be distinguished from one another). If this creates a difficulty, it would be possible to add logic to the code to catch such a case and use a different sequence (i.e. 3-1-3) when it arises.

2.4.11 write_VTK_mesh_file

```
void write_VTK_mesh_file(const std::string &FileName)
```

This method currently provides limited functionality for writing a mesh in VTK format. Not all element types are supported.

2.4.12 get_min_coords and get_max_coords

```
std::vector<double> get_min_coords() const
std::vector<double> get_max_coords() const
```

These methods return a vector that contains the min or max of each coordinate present in the mesh.

2.4.13 get_num_nodes and get_num_elements

```
std::vector<double> get_num_nodes() const
std::vector<double> get_num_elements() const
```

These methods return the number of nodes and the number of elements in the mesh.

2.4.14 clone_mesh

```
Mesh & clone_mesh() const
```

This method returns a new mesh object which is a deep copy of the current mesh object.

2.4.15 extrude_2D_mesh

```
Mesh extrude_2D_mesh(const std::vector<double> &ExtrusionCoordVals) const
```

This method returns a new mesh which is the result of extruding the current 2D mesh in the z direction. Currently the mesh must be quadratic.

2.4.16 extrude_2D_mesh_along_path

```
Mesh extrude_2D_mesh_along_path(const std::vector<double> &ExtrusionZCoordVals,
                                const std::vector<double> &ExtrusionXCoordOffsets,
                                const std::vector<double> &ExtrusionYCoordOffsets) const
```

This method allows you to sweep a 2D mesh in the z direction with some offsets in the x and y directions. Currently the mesh must be quadratic.

2.4.17 read_beta_elemat_file

```
void read_beta_elemat_file(const std::string &FileName)
```

This method reads a Beta format .elemat file to populate the elements' material numbers. It will likely not work right if you add or remove elements after reading in the mesh but before calling this method.

2.4.18 read_beta_mangles_file

```
void read_beta_mangles_file(const std::string &FileName)
```

This method reads a Beta format .mangles file to populate the elements' orientation data. It will likely not work right if you add or remove elements after reading in the mesh but before calling this method.

2.4.19 read_beta_orientation_file

```
void read_beta_orientation_file(const std::string &FileName)
```

This method reads a Beta format .orient file to populate the elements' orientation data. It will likely not work right if you add or remove elements after reading in the mesh but before calling this method.

2.4.20 translate_mesh

```
void translate_mesh(const std::vector<double> &TranslationVector)
```

This method translates the mesh according to the input argument.

2.4.21 scale_mesh

```
void scale_mesh(const double &Factor,
                const char &Dir,
                const double &FixedCoord)
```

This method stretches the mesh. Orientations are scaled as well so that the x direction is correctly rotated due to the scaling and the y direction stays on the global XY plane. The logic for modifying the orientaiton is based on orientations for weaves, and is possibly not appropriate for all potential cases. The arguements are as follows:

- **Factor -** The amount to stretch the mesh. This value should be positive. 1.0 would leave the mesh unchanged
- **Dir -** A single character, either X, Y, or Z, that specifies the dimension to be stretched
- **FixedCoord -** A double which specifies which coordinate in the stretch direction should remain unchanged.

2.4.22 mirror_mesh

```
void mirror_mesh(const char &Dir,
                 const double &Value)
```

This method flips the mesh about a plane normal to the axis specified by the argument Dir that passes through the argument Value. The y axes of mirrored orientations are flipped to keep the local coordinate system right-handed.

2.4.23 rotate_mesh

```
void rotate_mesh(const std::vector<double> &Axis,
                 const double &Angle)
```

This method rotates the mesh about an axis passing through the origin specified by argument Axis by input argument Angle. Angle is specified in degrees. All orientation are rotated as well.

2.4.24 remove_duplicate_nodes

```
int remove_duplicate_nodes(const double &Tolerance = 0.0)
```

This method removes any nodes which are within a bounding box of size Tolerance compared to other nodes. It returns the number of nodes removed.

2.4.25 remove_unattached_nodes

```
int remove_unattached_nodes(const int &NodesToLeave = 0)
```

This method removes nodes in the mesh which are not attached to any element, leaving NodesToLeave unattached nodes in place. It returns the number of nodes removed.

2.4.26 map_to_circle_to_square

```
void map_to_circle_to_square(const double &SideLength,
                             const double &SquareMappingStartRadius,
                             const double &StopAngle,
                             const double &StartAngle = 0.0)
```

This method, when implemented, will map a mesh from (r, θ) to (x, y), additionally mapping the resulting circular region to a square region at its borders. This method needs better documentation and some pictures to describe it appropriately.

2.4.27 standardize_orientations

```
void standardize_orientations()
```

This method runs though all the orientations and applies 180 degree rotations as needed so that the local x axis is pointing in the same general direction as the positive global x axis, or in the positive global y/z direction if it is perpendicular to the global x axis. Also, the local z axis is made to point in the same general directioin as the global z axis, or in the negative global x/y axis if it is perpendicular to the global z axis. This way, orientations for mirrored/rotated meshes will be easier to compare to the orientations in the meshes they originally came from.

2.4.28 remove_material

```
void remove_material(const int &MatNumber)
```

This method removes all elements that have material number MatNumber from the mesh. Automatically calls remove_unattached_nodes(0) to remove the nodes associated with these elements.

2.4.29 remove_element

```
void remove_element(const int &ElemNumber)
```

This method removes any element with the number ElemNumber. Note that unless renumber_nodes_and_elements (Section 2.4.32) has been called, there could be multiple elements with the same number.

2.4.30 remove_elements_by_loop

```
void remove_elements_by_loop(const int &FirstElemNumber,
                             const int &LastElemNumber,
                             const int &Increment)
```

This method removes any element with a number that would be touched in a loop of the following form:

```
for(int i=FirstElemNumber,i<=LastElemNumber;i+=Increment){
    ElementNumber = i;
}
```

2.4.31 remove_elements_by_bounding_box

```
void remove_elements_by_bounding_box(const std::vector<double> &coord1,
                                     const std::vector<double> &coord2)
```

This method removes any element with a central point that lies in the box aligned with the coordinate axes and defined by the input coordinates (coordinates define opposite corners of the box).

2.4.32 renumber_nodes_and_elements

```
void renumber_nodes_and_elements()
```

This method loops through the nodes and elements in the mesh and assigns them numbers in ascending order.

2.4.33 make_mesh_quadratic

```
void make_mesh_quadratic()
```

This method replaces all linear elements in the mesh with quadratic elements by adding quadratic nodes halfway between the corner nodes. It also results in remove_duplicate_nodes being run (Section 2.4.24). Currently it only works for 1D and 2D meshes.

2.4.34 make_mesh_linear

```
void make_mesh_linear()
```

This method replaces all quadratic elements in the mesh with linear elements by removing quadratic nodes. It also results in remove_unattached_nodes being run (Section 2.4.25). Works for 1D, 2D, or 3D meshes.

2.4.35 assign_material_number_to_all

```
void assign_material_number_to_all(const int &MatNumber)
```

This method assigns the material number MatNumber to all elements in the mesh.

2.4.36 assign_mat_number_using_loop

```
void assign_mat_number_using_loop(const int &First,
                                  const int &Last,
                                  const int &Increment,
                                  const int &MatNumber)
```

This method assigns material numbers to elements in the mesh by looping through them with a loop of the following form:

```
for(int i=First,i<=Last;i+=Increment){
    element[i].set_material_number(MatNumber);
}
```

2.4.37 remap_material_number

```
void remap_material_number(const int &From,
                           const int &To)
```

This method causes any element with the material number From to be assigned the material number To instead.

2.4.38 = operator

```
Mesh & operator = (const Mesh &Arg)
```

This operator causes this mesh object to be a deep copy of the argument mesh object (creates new node and element objects).

2.4.39 += operator

```
Mesh & operator += (const Mesh &Arg)
```

This operator adds the argument mesh object to the current mesh object by creating copies of all nodes and elements from the argument mesh in the current mesh object. The pointers in these elements and nodes are then mapped to the new elements and nodes that have been created in the current mesh.

2.4.40 + operator

```
inline Mesh & operator + (const Mesh &Arg) const
```

This operator returns a new mesh object that is the result of combining the current and arugment meshes.

2.5 TextileMesh

The TextileMesh class is inherited from the Mesh class. It contains special functionality to generate a textile mesh.

2.5.1 void make_half_tow_cross_section

```
std::vector<std::vector<double> >
    make_half_tow_cross_section(const int &NumElementsAlongHalfWidth,
                                const double &WavinessRatio,
                                const double &ThroughThicknessRefinementRatio = 1.0,
                                const double &TowMaxThickness = 0.5,
                                const double &TowVolumeFraction = 2.0/M_PI)
```

This method creates a 2D cross section of a half-tow with a linticuar shape (Figure 4). The cross section will contain quad and triangle elements, and the number of elements through the thickness will decrease towards the tip of the tow. Note that it may be possible to specify geometries that this method cannot handle. All elements in the cross section are assigned material number 1. The method returns a vector of vectors that are (x,y) pairs representing the coordinates of the nodes on the top of the cross-section. The input arguments are defined as follows:

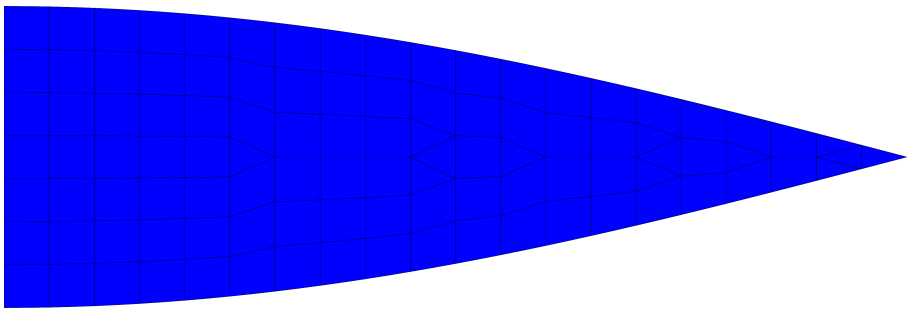

Figure 4: Output from make_half_tow_cross_section

- **NumElementsAlongHalfWidth -** The number of elements from the tow's center to its tip
- **WavinessRatio -** The amplitude of the tow's sine-wave path divided by its period
- **ThroughThicknessRefinementRatio -** A parameter which controls the refinement through the tow's thickness. The default value of 1.0 means that quad elements will be generated such that they have an aspect ratio as close to 1.0 as possible. Higher numbers give more elements through the thickness.
- **TowMaxThickness -** The maximum thickness of the tow. The default value of 0.5 will create tows with boundaries that match those output by MeshWeaver.
- **TowVolumeFraction -** The desired volume fraction of tow in the textile unit cell. Anything below $\frac{2}{\pi}$ will be set equal to $\frac{2}{\pi}$ (the tow volume fraction for lenticular tows). Anything above $\frac{2}{\pi}$ will result in a flattened region being inserted at the tow center.

2.5.2 void make_half_tow_cross_section_rounded

```
std::vector<std::vector<double> >
    make_half_tow_cross_section_rounded(const int &NumElementsAlongHalfWidth,
                                        const double &WavinessRatio,
                                        const int &ElementWidthsToRound,
                                        const double &ThroughThicknessRefinementRatio = 1.0,
                                        const double &TowMaxThickness = 0.5,
                                        const double &TowVolumeFraction = 2.0/M_PI)
```

This method creates a 2D cross section of a half-tow with a linticuar shape, except that it rounds the tow edge (Figure 5) by transitioning the tow cross section to a circular arc at specified node, maintaining C1 smoothness of the tow cross-section boundary at the transition node. A flat region is added to the tow to maintain the same ratio of tow to matrix in the resulting unit cell. The cross section will contain quad and triangle elements, and the number of elements through the thickness will decrease towards the tip of the tow. It is possible that some types of geometries and refinements are not possible. All elements which are part of the tow are assigned material number 1, and the matrix region beyond the tip of the tow is assigned material number 4 (this is to make it simple to differentiate between typical matrix

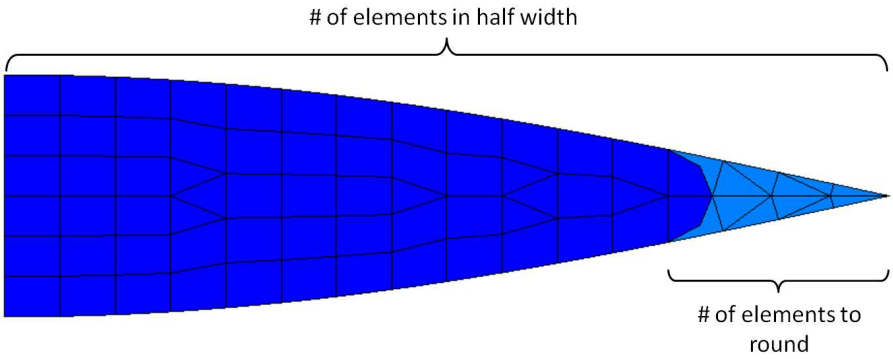

Figure 5: Output from make_half_tow_cross_section_rounded

pocket regions and matrix which is padding the edge of a rounded tow). The arguments are the same as those for make_half_tow_cross_section (section 2.5.1) with the exception of ElementWidthsToRound, which indicates the number of element widths from the tow tip where the element should be rounded (must be greater than 0).

2.5.3 make_3D_half_tow

```
void make_3D_half_tow(const int &NumElementsAlongHalfWidth,
                      const double &WavinessRatio,
                      const int &ElementsToRound = 0,
                      const double &ThroughThicknessRefinementRatio = 1.0,
                      const double &TowMaxThickness = 0.5,
                      const double &TowVolumeFraction = 2.0/M_PI)
```

This method creates a 3D half tow which is suitible to replace the x direction tow of a 1/16 simple or 1/32 symmetric unit cell generated by MeshWeaver after proper translation. Can be rotated about ¡1.0,1.0,0.0¿ to obtain the y direction tow. The arguments are the same as those for make_half_tow_cross_section (section 2.5.1), with the addition of **ElementsToRound**, which is a integer determining whether how the edge of the tow is to be rounded (default is 0, meaning no rounded edge).

2.5.4 make_matrix_pocket_cross_section

```
void make_matrix_pocket_cross_section(const int &NumElementsAlongHalfWidth,
                                      const double &WavinessRatio,
                                      const double &ThroughThicknessRefinementRatio = 1.0,
                                      const double &TowMaxThickness = 0.5)
```

This method creates a cross section of matrix (Figure 6) that will conform to the tow cross section generated by either make_half_tow_cross_section (section 2.5.1) or make_half_tow_cross_section_rounded (section 2.5.2). The input parameters are the same as those described in (section 2.5.1), and with the exception of ThroughThicknessRefinementRatio, must be the same as those used to define the tow that

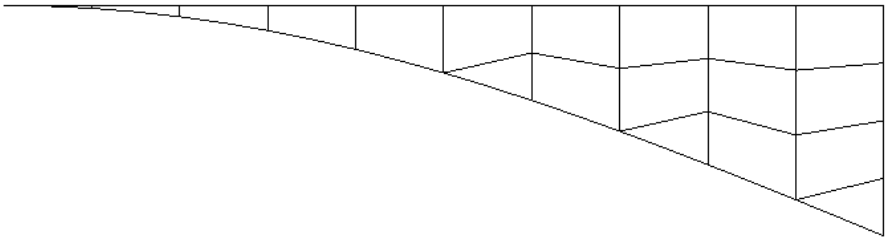

Figure 6: Output from make_matrix_pocket_cross_section

the matrix pocket is intended to conform to. Note that for low values of NumElementsAlongHalfWidth and high values of ThroughThicknessRefinementRatio, it may be possible to create a broken matrix pocket.

2.5.5 make_matrix_pocket

```
void make_matrix_pocket(const std::vector< std::vector <double> > &TopNodesForTowMesh,
                        const int &NumElementsAlongHalfWidth,
                        const double &WavinessRatio,
                        const double &ThroughThicknessRefinementRatio = 1.0,
                        const double &TowMaxThickness = 0.5)
```

This method uses the cross section from make_matrix_pocket_cross_section to generate the 3D matrix pocket regions for a 1/16 single mat plain-weave unit cell. The input parameters are the same as those described for make_half_tow_cross_section (section 2.5.1)

2.5.6 make_PW_unit_cell

```
void make_PW_unit_cell(const int &NumElementsAlongHalfWidth,
                       const double &WavinessRatio,
                       const int &ElementsToRound = 0,
                       const double &ThroughThicknessRefinementRatio = 1.0,
                       const double &TowMaxThickness = 0.5,
                       const double &TowVolumeFraction = 2.0/M_PI)
```

This method generates an entire 1/16 single mat unit cell for a plain-weave composite, including matrix pocket. All elements contain orientaiton data. The X direction tow is material 1, the Y direction tow is material 2, and the matrix is material 3. The parameters are the same as those for make_3D_half_tow (section 2.5.3).

2.6 Q3DLaminateMesh

This section of the documentation has not yet been completed.

2.7 Utility Functions

This section of the documentation has not yet been completed.

2.7.1 linspace

```
std::vector<double> linspace(const double &first,
                             const double &last,
                             const int &NumIntervals)
```

This function returns a vector of doubles with "NumIntervals"+1 values evenly spaced between "first" and "last". For instance, linspace(2.0, 9.0, 7) would return the vector seen in figure 7.

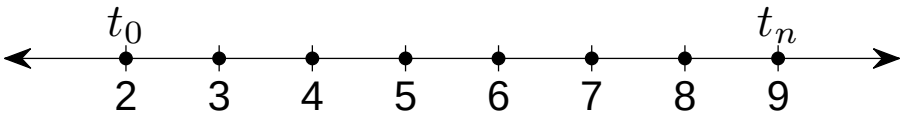

Figure 7: Output from linspace(2.0, 9.0, 7)

2.7.2 bias_vector

```
std::vector<double> bias_vector(const double & BiasFactor,
                                const std::vector<double> &VectorToBias)
```

This function performs a mapping on values in a vector which can shift nodes towards the first or last value in the vector. It is intended to work on monotonically increasing or decreasing vectors. The input parameter "BiasFactor" can be any value on the interval [-1,1]. The mapping that is performed is of the form

$$x(t) = t_0 + \int_{t_0}^{t} f(\xi)\, d\xi \tag{1}$$

where

$$f(t) = \lambda \frac{2}{t_n - t_0} \left(t - \frac{t_0 + t_n}{2} \right) + 1, \quad -1 \leq \lambda \leq 1 \tag{2}$$

and λ is the value input as "BiasFactor". Positive λ biases towards the last value in the vector, while negative λ biases towards the first value in the vector. $\lambda = 0$ will leave the vector unchanged. For instance, if the vector from figure 7 were to be biased with $\lambda = 1$, the result would be that seen in figure 8. Note that this function can be recursively applied to a vector to obtain additional biasing.

3 Using BetaMesh Standalone Executable

This section describes how to use the BetaMesh executable. It has not been completed. The standalone executable reads a script file which results in calls to the various mesh methods in BetaMesh.

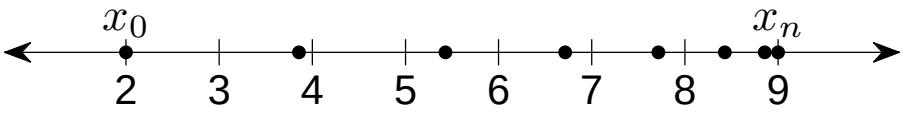

Figure 8: Vector biased with BiasFactor = 1

4 BetaMesh in Python

This section describes how to setup and use the BetaMesh extension for Python.

4.1 Setup

Currently the code is set up to be wrapped for Python 2.6 with SWIG. You need to have both of these installed, and the SWIG and Python executables need to be in your path. For best results, install the 64 bit version of Python. The Python module is set to install to C:/exe/PythonLibs. This folder needs to be added to an environment variable called PYTHONPATH. PYTHONPATH is a set of directories where python will look for modules. If PYTHONPATH doesn't exist on your machine, you will need to create it.

4.2 Building the BetaMesh Module for Python

Follow the following steps to create the BetaMesh module for Python...

1. Run the script BetaMesh/PythonExtension/SwigFiles/SwigBetaMesh.bat to create the wrapper file (it will be called BetaMesh_wrap.cxx)
2. Open and build the VS project BetaMeshPythonExt. Make sure the target platform matches the verion of Python that you have installed (32 or 64 bit)
3. Make sure that the directory C:/exe/PythonLibs is in your PYTHONPATH environment variable.

That's it. You now can use BetaMesh from Python.

4.3 Using BetaMesh in Python

Probably the best way to figure out how to use BetaMesh in Python is to check the example scripts in BetaMesh/PythonExamples. To use BetaMesh in Python, you need to import the BetaMesh module you created using the import command. For example, to create an empty mesh object, you would use the following set of commands:

```
import BetaMesh
MyMesh = BetaMesh.Mesh()
```

To import all of the BetaMesh methods directly into the namespace (essentially allowing you to type Mesh() instead of BetaMesh.Mesh()), you would do the following:

```
from BetaMesh import *
MyMesh = Mesh()
```

Note that python does not support templated types. Therefore, you cannot directly access the STL vector class from Python. However, the BetaMesh Python module contains special classes called DoubleVector and IntVector which are essentially STL vectors with the template resolved to a particular type. Here is an example of creating and using a STL vector of doubles in Python:

```
import BetaMesh
MyMesh = BetaMesh.Mesh()
# create a STL vector of doubles with 5 elements, inialized to 0.0
CoordVector = BetaMesh.DoubleVector(5,0.0)
# populate CoordVector so that it is {0.0, 0.2, 0.4, 0.6, 0.8}
for i in range(5):
    DoubleVector[i] = i*0.2
# Do something with the DoubleVector object you created
MyMesh.make_grid_mesh(CoordVector,CoordVector)
```

Note that for make_triangle_mesh, you need to input a vector of vectors of vectors of doubles. This datatype is defined by DoubleVVVector. Similarly there is a DoubleVVector.