

FEP Course Project: GeoMod

Justin Clough, RIN:661682899

June 29, 2017

Abstract

The use of non-manifold models can allow for better approximations of physical problems. This has been shown in previous research of matrix-fiber composites as well as on going work in coupled finite element-dislocation dynamics. The code created in this project allows users to make modifications to a preexisting model and define meshing attributes. The modifications include placement of model vertices, edges, and surfaces. When the user places these entities in the model, they also specify the local mesh refinement. Model entities are always created such that the topology of the model remains valid. Mesh refinement information is only updated if the new refinement level is finer than the current. Thirteen tests were created to confirm that expected behavior of the code was achieved. Of these tests, three focus on vertex placement, four on edge placement, and three on surface placement. The results of each test are presented as evidence of passing the test. Possible future work include parallelization, handling of assembly models, and inclusion of features that can delete or modify existing model entities.

1 Introduction

Non-manifold models can allow for better approximations of physical scenarios. In the scope of this report, a non-manifold model is taken to mean a model in which every point on a surface need not be two dimensional [1]. This allows for the creation of models where, for example, two or more three dimensional regions need only share a vertex or an edge. This also supports geometric entities containing lower entities. E.g., a vertex can be placed in a region, surface, or edge; an edge may be placed in a region or surface; a surface can be placed within a region.

An example of how non-manifold models are used is in the modeling of the micro structure of cross-linked fiber networks embedded in a continuous matrix [2]. In this dissertation, a three dimensional region was used to represent a continuous matrix; one dimensional edges were used to represent collagen network fibers. The fibers were able to begin and terminate on either a model face or in the model region. The resulting model was then discretized such that volumetric elements represented the matrix and beam elements represented the collagen fibers.

Another example of how a non-manifold model can be used is in the modeling of coupled finite element analysis with dislocation dynamics. In this scenario, a crystalline solid is represented as a three dimensional region. Within this region, edge and screw dislocations are represented as two dimensional edges. These dislocations both create and are acted on by the stress field around them [3]. Similar to the collagen fibers, the edges that represent the dislocations may begin and end in any model surface or region. The discretized model is comprised of volumetric finite elements. The edges of the elements near the dislocation are collinear with the dislocation; the vertices of the element are also coincident with the nodes of the dislocation. Additionally, the mesh near the dislocation needs to be finer to better approximate stress at the dislocation.

The goal of this project is to create an interface to the Simmetrix modeling and meshing Application Program Interfaces (APIs). This interface will allow the user to add non-manifold geometric features to a base geometric model. Additionally, the user needs to be able to define local mesh characteristics for the created geometry. The created geometry can include model vertices, edges, and faces. The edges and faces cannot be limited to straight lines and flat planes, respectively. During model modification, the validity of the topology must be maintained. This will allow for correct discretization of the model and interrogation of both the model and mesh. During mesh case specification, updated refinement criteria must be merged with the existing information. For these cases, finer level of refinement must always be used.

The remainder of this report is organized in the following manner. The general design of the code is presented in section 2; the specifics of the implementation for vertex, edge, and face placement is discussed in the subsections 2.1, 2.2, and 2.3,

respectively. Testing is presented in 3 with test cases and their results for vertex, edge, and face placement discussed in subsections 3.1, 3.2, and 3.3, respectively. Conclusions are drawn and possible future work is presented in section 4. Finally, the source code for the project is appended in Appendix A.

2 Code Design

The user interacts with the code by constructing a *gmd* object based on an existing in-memory Simmetrix geometric model that they wish to modify. The header and source files for the *gmd* class are shown in Appendices A.4 and A.5, respectively. The user can then modify the model by creating model vertices, edges, and faces by calling *gmd* member functions. Mesh case information for these modifications is also specified by the user at this time. The *gmd* class is primarily a wrapper around two other classes. It also has member functions that check the validity of user input for creating edges and faces. The two classes that it wraps in turn wrap around Simmetrix APIs.

The first class is the *model_helper* class which handles all model interactions. This includes making modifications to the model, checking its validity, and writing it to disk. Its member variables include a pointer to the Simmetrix model. The header and source file of the *model_helper* class are shown in Appendices A.6 and A.7, respectively. The APIs used are from Simmetrix’s GeoSim Core and Advanced libraries [4].

The second class wrapped within the *gmd* class is the *mesh_helper* class which handles all mesh interactions. This includes defining global and local mesh parameters, checking for validity, and writing the mesh. Its member variables include a pointer to a Simmetrix mesh object and mesh case object. The header and source file for the *mesh_helper* class are shown in Appendices A.8 and A.9, respectively. The APIs used are from Simmetrix’s MeshSim Core and Advanced libraries [4].

Model entities are always created on the geometric entity with the lowest possible dimension when applicable. This ensures correct classification when model or mesh entities are later interrogated. For example, say the user wants to create a vertex whose coordinates lie upon an edge which bounds a face of a region. The vertex will be created and classified as an entity on the edge as opposed to only the face or region. This is detailed in subsection 2.1 and demonstrated in subsection 3.1.

Mesh case information, such as local refinement level, are specified during model modification by the user; the finer specification is always used. For example, if the user creates an edge with a relative local refinement of 0.1 and creates a vertex on that edge with a refinement of 0.5, then only a refinement level of 0.1 will be used throughout the whole edge. Alternatively, if a refinement level

of 0.5 is specified for the edge and 0.1 specified for a vertex on that edge, then a refinement level of 0.5 will be used throughout the edge except for space near the vertex. The rate at which the refinement level decreases from 0.5 to 0.1 for this case is controlled by the gradation rate. A rate of 2/3 is used by default but can be changed by the user.

Details regarding model vertex, edge, and face placement are discussed in subsections 2.1, 2.2, and 2.3, respectively. Testing and results of tests are presented in section 3.

2.1 Vertex Placement

The gmd member function *place_point()* is used to create model vertices as well as define the local mesh refinement level for the created vertex. The user specifies the coordinates of the vertex they want to create, the relative mesh refinement level, the radius of refinement, and a Simmetrix model vertex pointer that is overwritten to point to the created vertex.

The Simmetrix APIs *GE_closestPoint()*, *GF_closestPoint()*, and *GR_containsPoint()* are used to determine whether a 3D coordinate is in a model edge, face, or region, respectively. The function *GR_containsPoint()* returns an integer denoting if the point is inside or outside of the model region. The functions *GE_closestPoint()* and *GF_closestPoint()* both return pointers to the real and parametric coordinates closest to a given test coordinate. The real coordinates of the closest point are then compared to the test point. If the magnitude of the distance between the two points is less than the tolerance of the model, then a model vertex is created. The value returned from the API *GM_tolerance()* is used as the tolerance for all geometric comparisons.

If the user specifies coordinates which are outside any model region, the vertex is created but either a warning or error message is printed. By default, an error message is printed but this can be changed by the user. If the user specifies coordinates which are found to lie on a preexisting edge, then the point is created by splitting the edge into two. In this case, the model vertex pointer returned to the user is for the vertex at the split. Examples of vertex placement are shown in subsection 3.1.

2.2 Edge Placement

The gmd member function *place_edge()* is used to create model edges and define the local mesh refinement for the created edge. The underlying API used is the *SCurve_createBSpline()* function which creates rational and non-rational basis splines. The user specifies the order of the curve, control points, knots, weights,

mesh refinement level, and a Simmetrix model edge pointer which is overwritten with the created edge.

The *place_edge()* function first checks the validity of the user's input. The checks ensure that the user defined input meets the following conditions:

1. The order is not less than one or greater than the number of control points.
2. The sum of the number of control points and order is equal to the number of knots.
3. The first order number of knots are equal to zero and the last order number of knots are equal to one. E.g., if order is two, then the first two knots must equal zero and the last two knots must equal one.
4. The knots are specified in a monotonically increasing order.
5. The number of weights satisfies one of two conditions:
 - (a) The number of weights is equal to the number of control points.
 - (b) The number of weights is one and the weight is equal to zero. This condition dictates the construction of a non-rational curve instead of a rational one.

If the user defined inputs do not meet the above conditions, then an error message is printed and the program aborts.

First, the end points of edge are created as vertices using the method described in subsection 2.2. Next, a curve is created using the *SCurve_createBSpline()* API. The end points of the edge, the first and last control points, are created as model vertices using the method described subsection 2.1. This curve and two vertices are then used to create a geometric edge using the *GIP_insertEdgeInRegion()* API. If all of the control points are coincident with a preexisting face, then the edge is inserted onto the corresponding face using the *GM_insertEdgeOnFace()* API. Examples of edge placement are shown in subsection 3.2.

2.3 Face Placement

The *gmd* member function *place_face()* is used to create model faces and define the local mesh refinement for the created face. The underlying API is the *SSurface_createBSpline()* function which creates rational and non-rational basis surfaces based on a local *u* and *v* coordinate system. The user specifies the following to create a surface:

1. The order of the surface in both the *u* and *v* directions.

2. The number of control points in the u and v directions.
3. The periodicity of the surface. In essence, whether the surface is periodic in either, neither, or both the u and v directions.
4. The coordinates of the control points. This is specified as shown in Table 1.
5. The weights for each control point. The order of the weights must also follow the pattern shown in Table 1.
6. The local mesh refinement on the face.
7. The geometric face pointer to be overwritten with the newly created face.

	$v \longrightarrow$			
	0	N	\dots	$N(M-1)$
	1	$N+1$	\dots	$N(M-1)+1$
	2	$N+2$	\dots	$N(M-1)+2$
\approx	\vdots	\vdots	\dots	\vdots
\downarrow	$N-1$	$2N-1$	\dots	$MN-1$

Table 1: Ordering for control points and weights for surface creation. N is the number of control points in the u direction. M is the number of control points in the v direction.

The *place_face()* function first checks the user defined inputs for validity. The checks ensure that the user's input meets the following conditions:

1. The order in both the u and v directions is not less than one or greater than the number of control points in the u and v directions respectively.
2. The total number of control points equals to the product of declared control points in the u and v directions.
3. The weights meet one of the two conditions:
 - (a) The number of weights control points equals to the product of declared control points in the u and v directions.
 - (b) The number of weights in one and that one weight equals to zero. This condition dictates that a non-rational surface be created instead of a rational one.

4. The number of knots is equal to the sum of the number of control points and order in both the u and v directions.
5. The knots are specified in a monotonically increasing order.
6. The first order number of knots are equal to zero and the last order number of knots are equal to one for each direction. E.g., if order is three, then the first three knots must equal zero and the last three knots must equal one.

If the user's input does not meet all of the above conditions, then an error message is printed and the program aborts. Otherwise, the program next creates a surface using the *SSurface_createBSpline()* API. The bounding edges of the face are created next. Each edge is created using the method described in subsection 2.2. Next, the edges and surfaces are used to create a model face using the *GIP_insertFaceInRegion()* function. The normal direction of the face is taken as the crossproduct of the u and v directions (i.e., $n = u \times v$) for any given position.

3 Testing

Each of the features described in section 2 were tested. In total, 13 tests were created; the header and source files for these tests are shown in Appendix A.2 and A.3. The first three tests (tests 0, 1, and 2) check for basic functionality such as proper linking between classes and functions as well as creation and deletion of classes without memory leaks. These tests are described in Appendix A.3 and their results are shown in Appendix B.1 and B.2. The next three tests (tests 3, 4, and 5) exercise vertex placement. They and their results are discussed in detail in subsection 3.1. The next four tests (tests 6, 7, 8, and 9) exercise edge placement in different cases. These tests and their results are shown in 3.2. The last three tests (tests 10, 11, and 12) check face placement functionality. They and their results are described in subsection 3.3.

3.1 Vertex Placement

Vertex placement was tested in three cases. For each test case, both the modified model and mesh were created and written to disk. Both the model and mesh were then viewed in the Simmodeler GUI to visually confirm that the expected behavior was achieved. Classification was also confirmed through use of Simmetrix APIs and through the GUI part tree.

The first case placed a vertex in the center of a three dimensional model region. The global mesh refinement level was set to 0.9 and the refinement level for the

inserted vertex was 0.1. The resulting model is shown in Figure 1. The resulting mesh is shown in Figure 2.

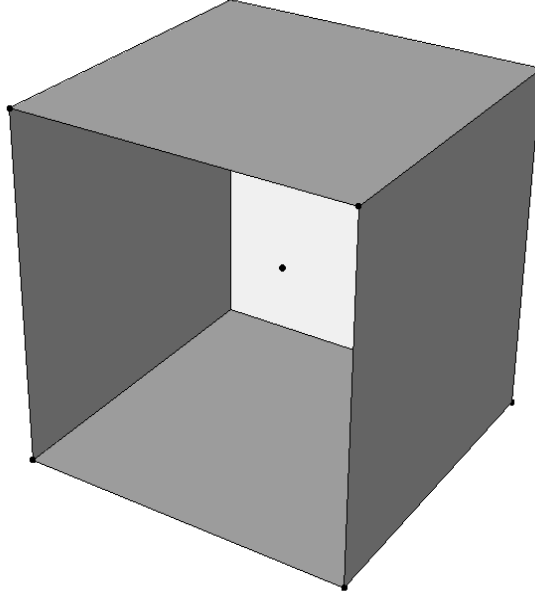
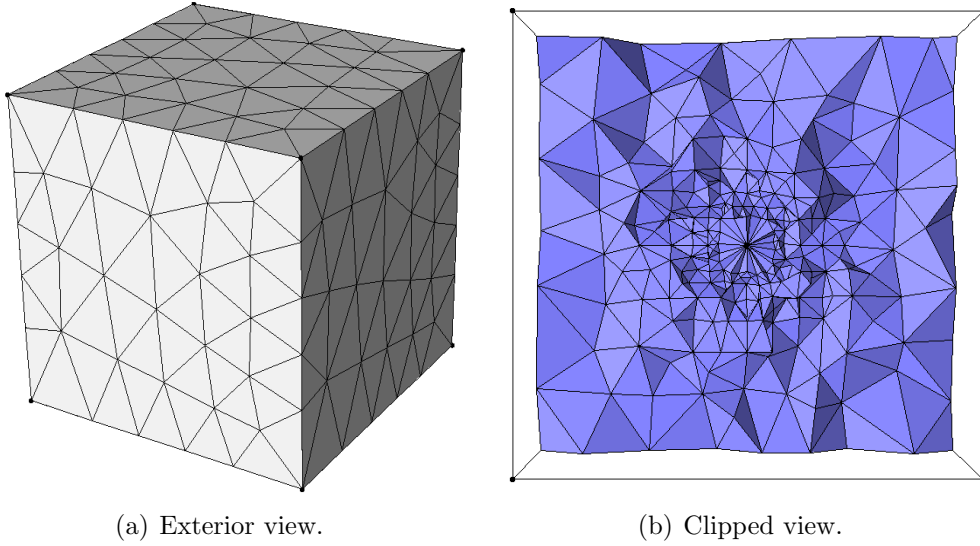


Figure 1: Cube with vertex placed at center. Front face hidden to to show interior.



(a) Exterior view.

(b) Clipped view.

Figure 2: Mesh of cube with vertex placed at center.

The second case placed a vertex in the center of a face of a three dimensional

model. The global mesh refinement level was set to 0.9 and the refinement level of the vertex was set to 0.1. The resulting model is shown in Figure 3. The resulting mesh is shown in Figure 4.

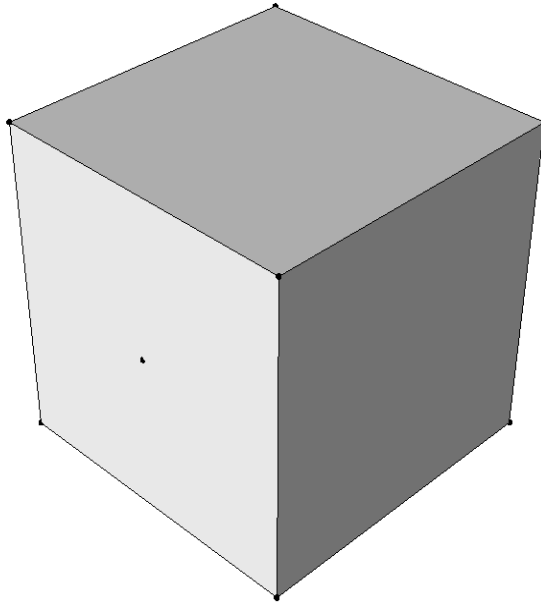


Figure 3: Cube with vertex placed at center of front face.

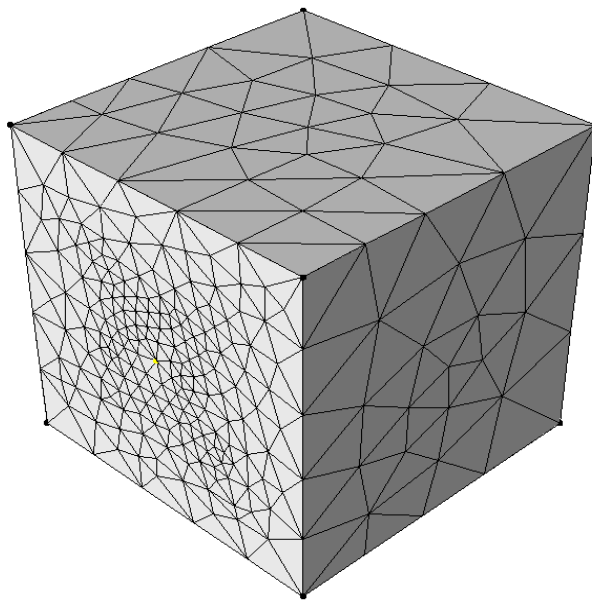


Figure 4: Mesh of cube with vertex on center of front face.

The third case placed a vertex on a preexisting model edge. The global mesh refinement level was 0.9 and the local refinement level for the vertex was 0.1. The resulting model is shown in Figure 5. The resulting mesh is shown in Figure 6.

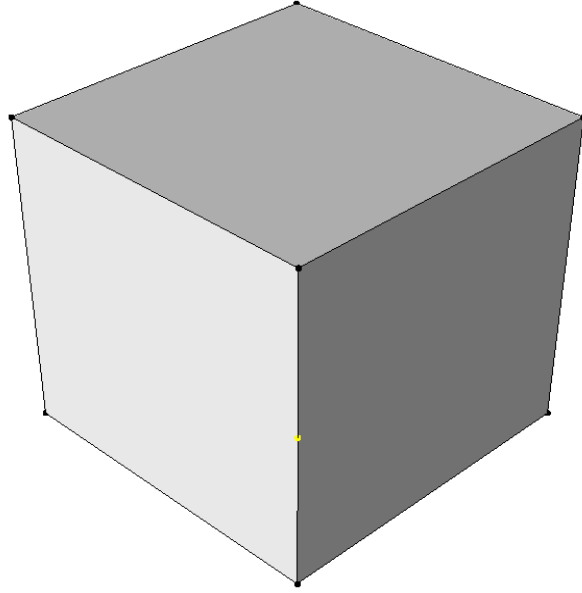


Figure 5: Cube with vertex placed at center of edge.

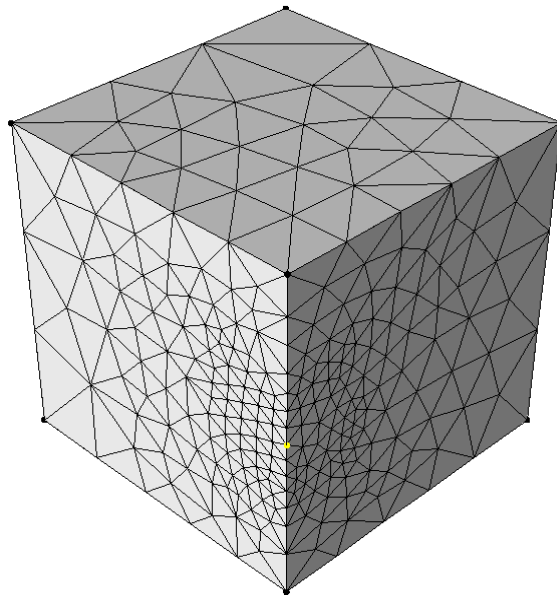


Figure 6: Mesh of cube with vertex on center of edge.

As shown in Figures 1 through 6, the gmd method *place_point()* is able to correctly determine the model classification given a user defined location. It is also able to change the local mesh refinement level to a user's specification.

3.2 Edge Placement

Edge placement was tested in four cases. Similar to the vertex tests presented in subsection 3.1, models and meshes were created and written to disk; expected behavior and classification was confirmed through both the Simmetrix APIs and the GUI.

The first test placed a fully interior edge inside a model region. The edge was created as a rational basis spline with five control points; each control point was inside the model region. The global mesh refinement was set to 0.9 and the local set to 0.1. A wire frame view of the created model is shown in Figure 7. A clipped view of the mesh is shown in Figure 8.

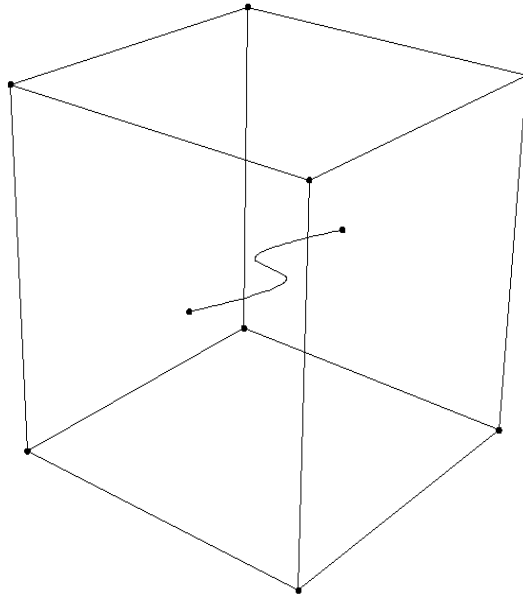


Figure 7: Cube with edge placed near center of region.

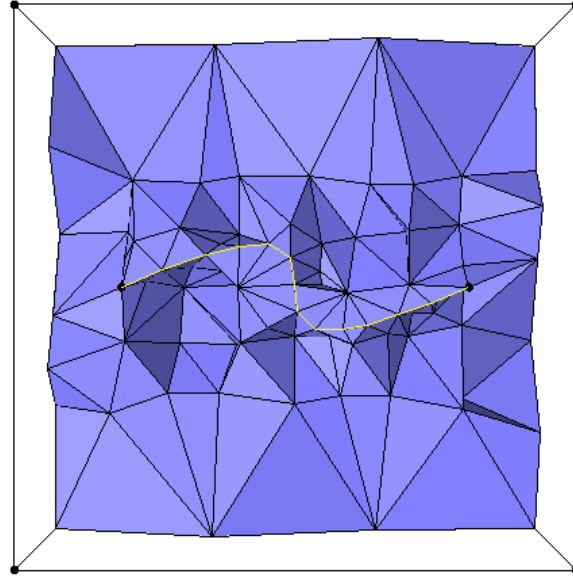


Figure 8: Clipped view of mesh for cube with edge near center of region. The edge is highlighted in yellow.

The second test placed an edge whose control points all laid on the same model face. The edge was created as a rational basis spline with five control points. The global mesh refinement was set to 0.9 and the local set to 0.1. The created model is shown in Figure 9. A view of the mesh is shown in Figure 10.

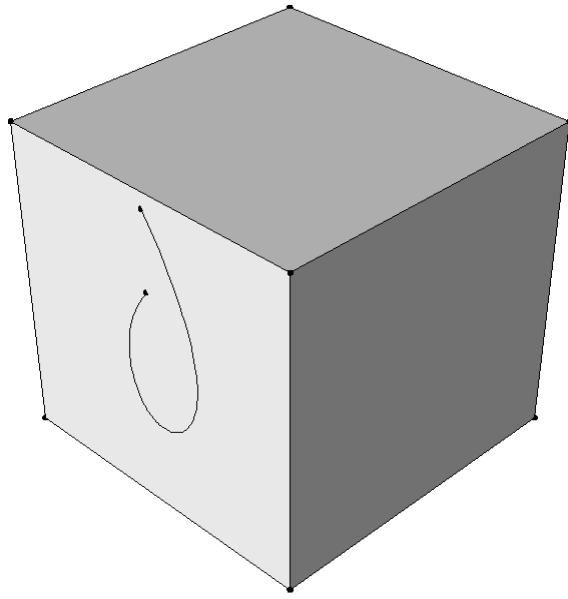


Figure 9: Cube with edge placed on model front face.

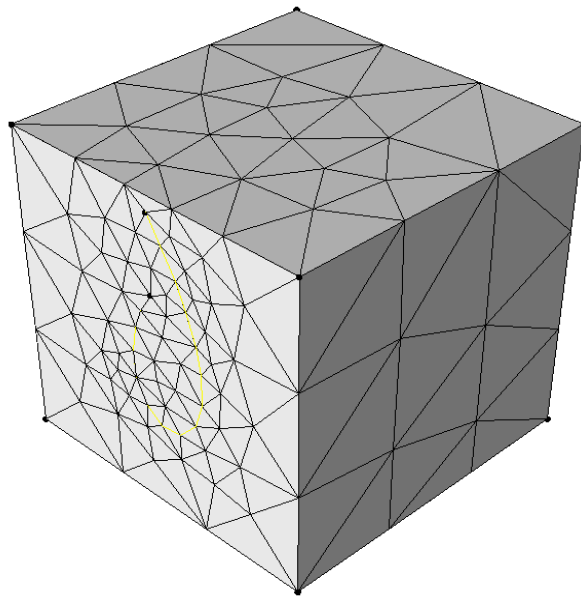


Figure 10: Clipped view of mesh for cube with edge on face. The edge is highlighted in yellow.

The third test placed an edge that started on a model face and terminated in the interior of the model region. The edge was created as a rational basis spline

with five control points. The global mesh refinement was set to 0.9 and the local set to 0.1. Two images of the created model are shown in Figure 11. A external and clipped view of the mesh are shown in Figure 12.

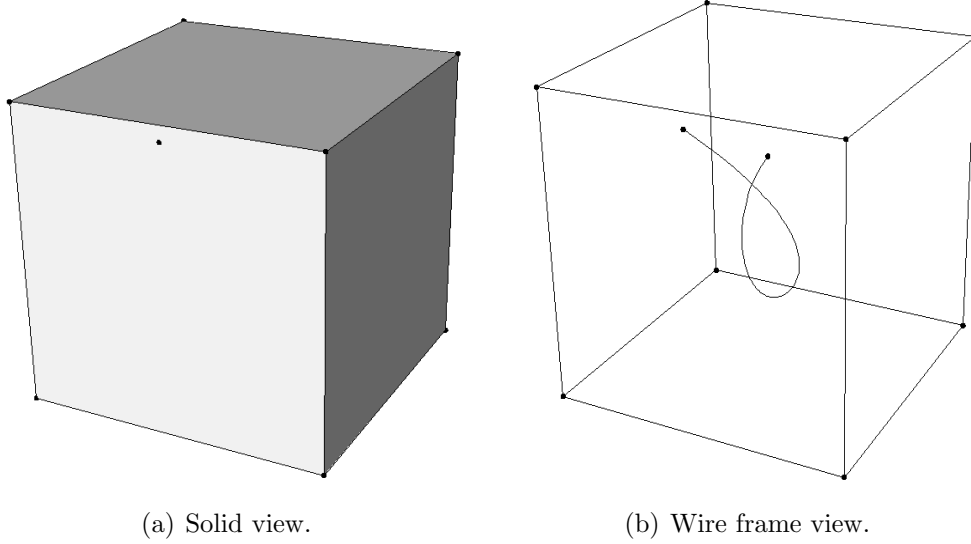


Figure 11: Model with edge inserted. Edge begins on front face and terminates with in the model region.

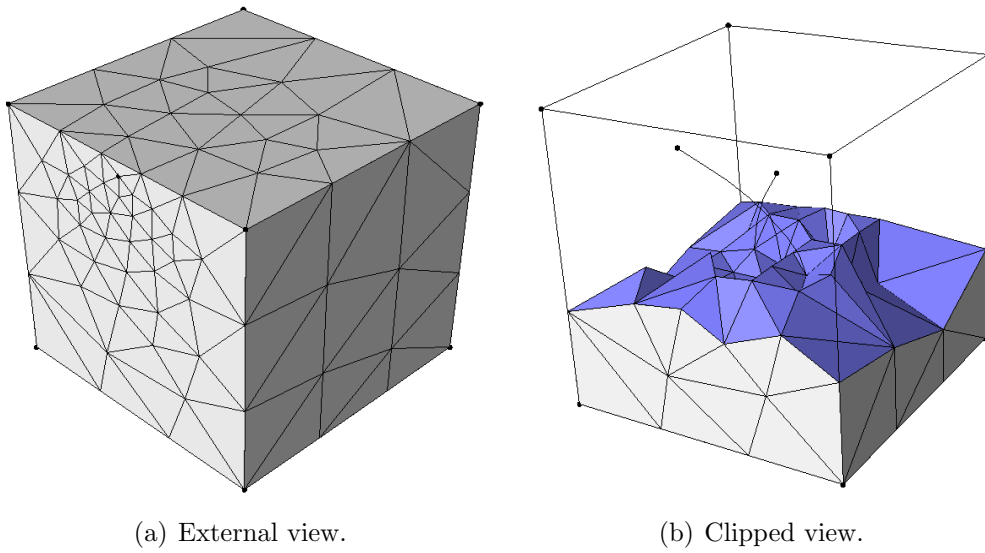


Figure 12: Mesh of model with edge from face to region inserted.

The fourth test placed an edge that started on a model edge and terminated

with in the model region. The edge was created as a rational basis spline with five control points. The global mesh refinement was set to 0.9 and the local set to 0.1. Two images of the created model are shown in Figure 13. A external and clipped view of the mesh are shown in Figure 14.

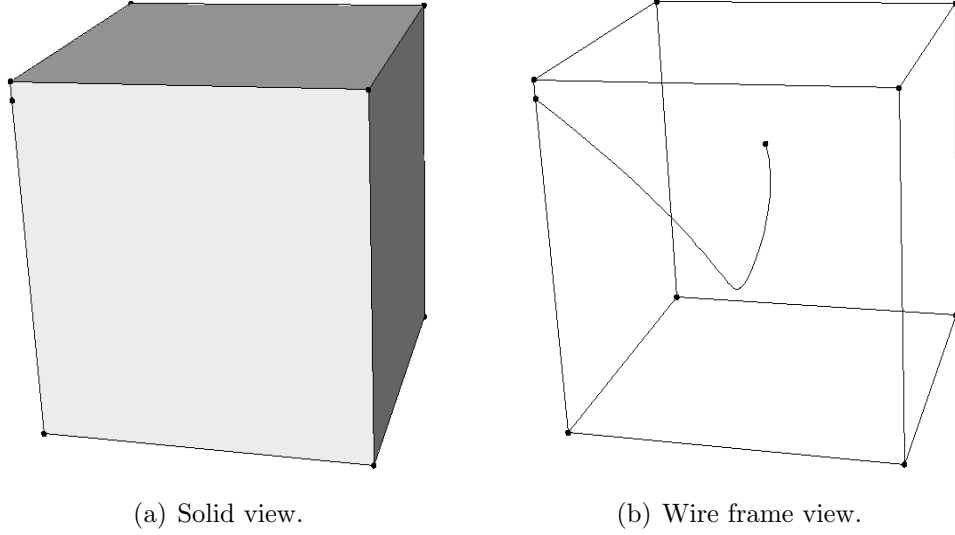


Figure 13: Model with edge inserted. Edge begins on left-most vertical model edge and terminates within the model region.

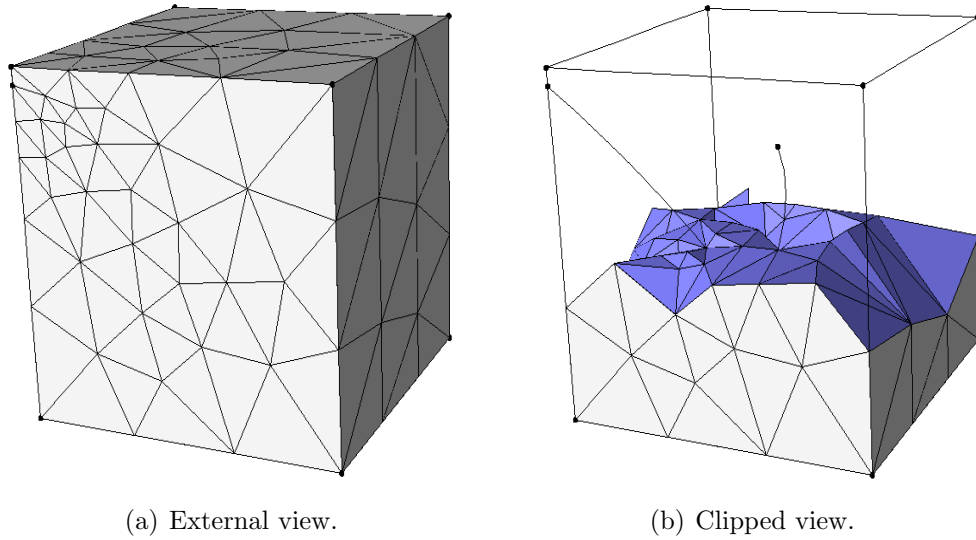


Figure 14: Mesh of model with edge from edge to region inserted.

As shown in Figures 7 through 14, the gmd method *place_edge()* is able to correctly determine the model classification given user defined spline parameters. It is also able to change the local mesh refinement level to a user's specification.

3.3 Face Placement

Face placement was tested similarly to both the vertex and edge tests described in subsections 3.1 and 3.2. In total, three tests were performed. For each test, the created models and meshes were written to disk and viewed in the Simmetrix GUI. Classifications of model entities were confirmed through both the GUI and APIs where applicable.

The first face placement test created a model face that was completely interior to a model region. The face, and its bounding edges, were created as a rational basis surface and splines. In total, 12 control points were defined; each control point was inside the model region and created a 4×3 grid. The global mesh refinement was set to 0.9 and the local set to 0.1. A face view of the created model is shown in Figure 15. The face and clipped view of the mesh are shown in Figure 16.

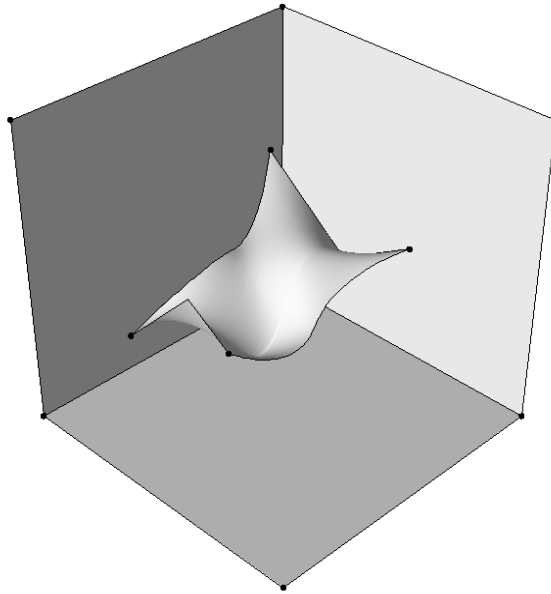


Figure 15: Cube with face created within the model region. Forward three faces were made transparent to show in the internal face.

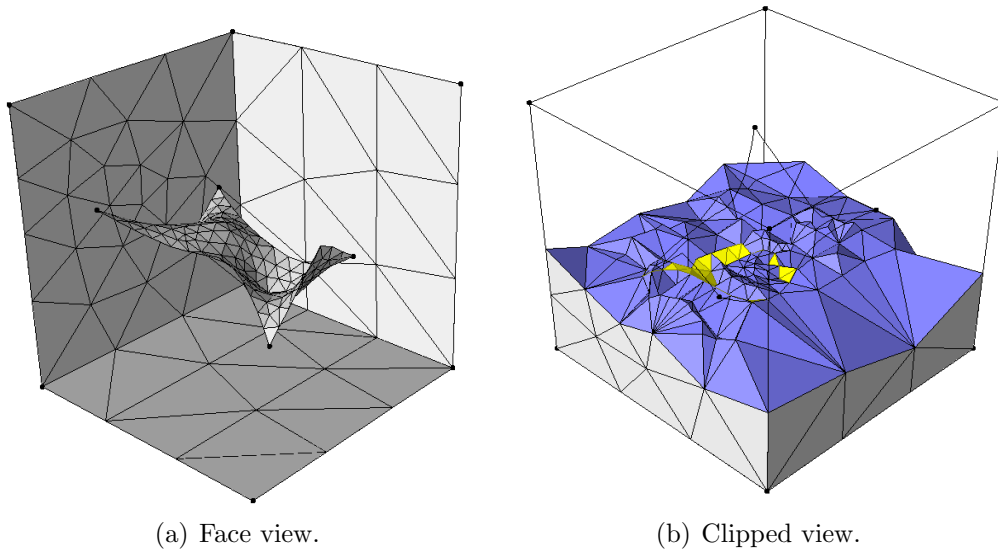


Figure 16: Mesh of model with interior face. Yellow mesh element faces indicate the mesh face is coplanar with the model face. Note the view is rotated right by 90° with respect to Figure 15.

The second face placement test created a model face that was completely interior to a model region except for one corner vertex. The face, and its bounding edges, were created as a rational basis surface and splines. In total, 12 control points were defined; all but one control point was inside the model region and created a 4×3 grid. The global mesh refinement was set to 0.9 and the local set to 0.1. A front and face view of the created model are shown in Figure 17. The face and clipped view of the mesh are shown in Figure 18. An external view of the mesh is shown in Appendix B.3.

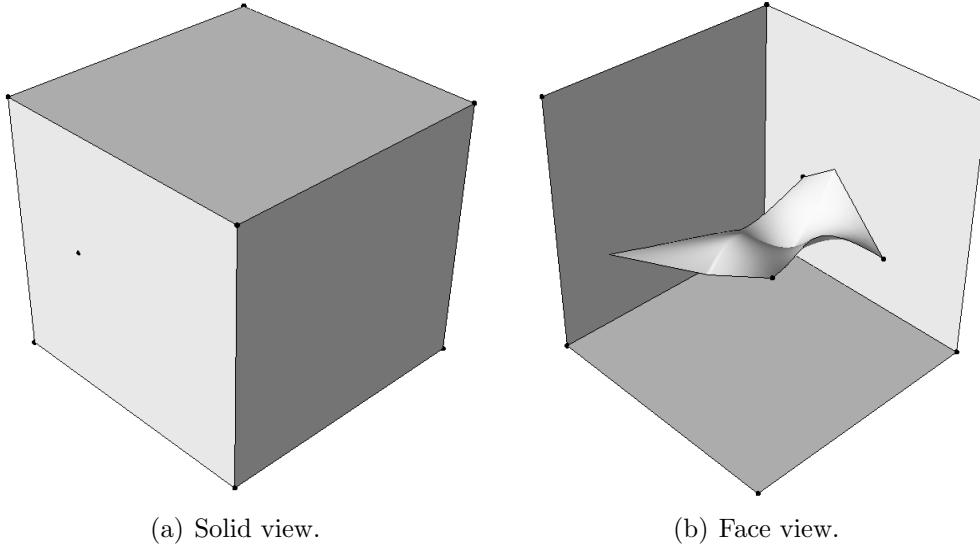


Figure 17: Model with face inserted. Face is fully interior except for one point. The point is shown on the leftmost face in (a).

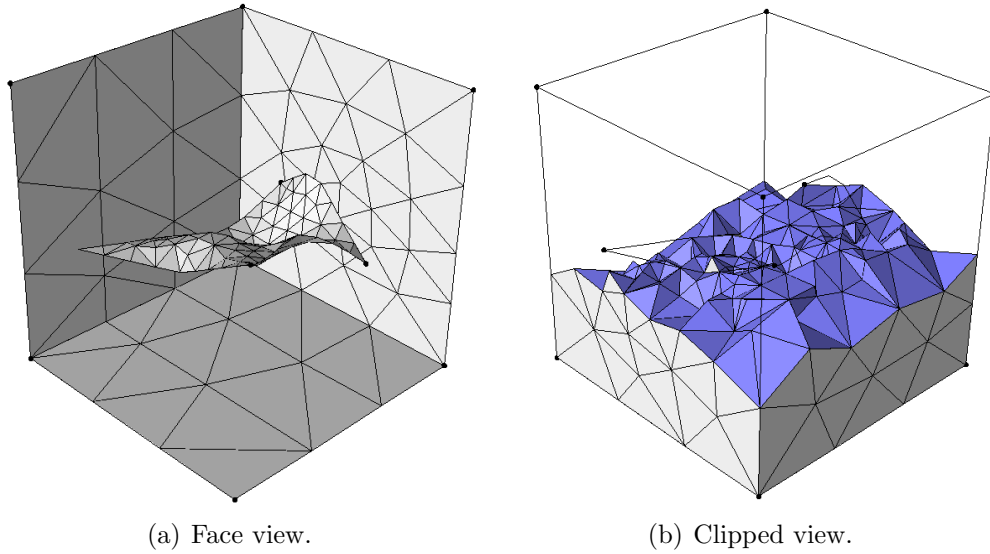


Figure 18: Mesh of model with interior face except for one point.

The third face placement test created a model face that was completely interior to a model region except for one edge. The face, and its bounding edges, were created as as a rational basis surface and splines. In total, 12 control points were defined; the control points created a 4×3 grid. The global mesh refinement was

set to 0.9 and the local set to 0.1. A front and face view of the created model are shown in Figure 19. The face and clipped view of the mesh are shown in Figure 20. An external view of the mesh is shown in Appendix B.4.

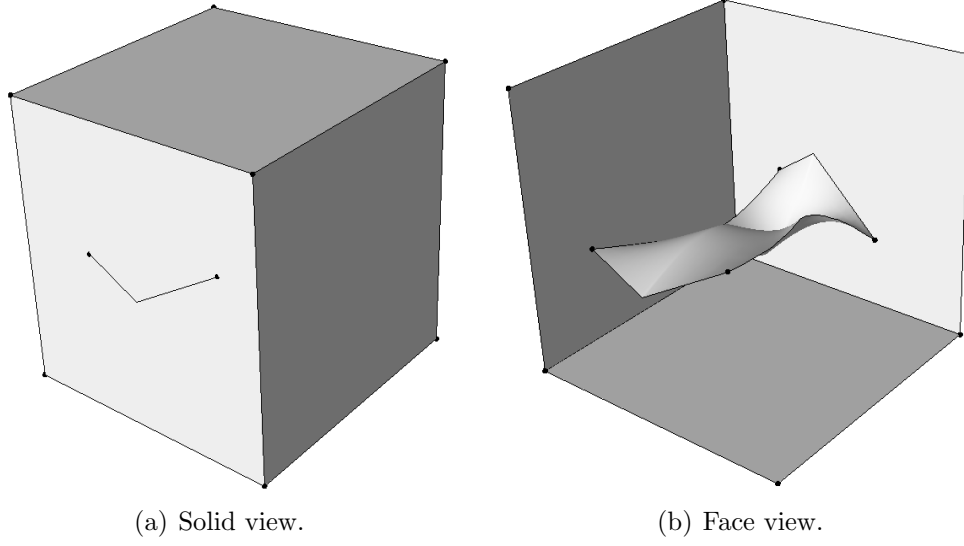


Figure 19: Model with face inserted. Face is fully interior except for one edge. The edge is shown on the leftmost face in (a).

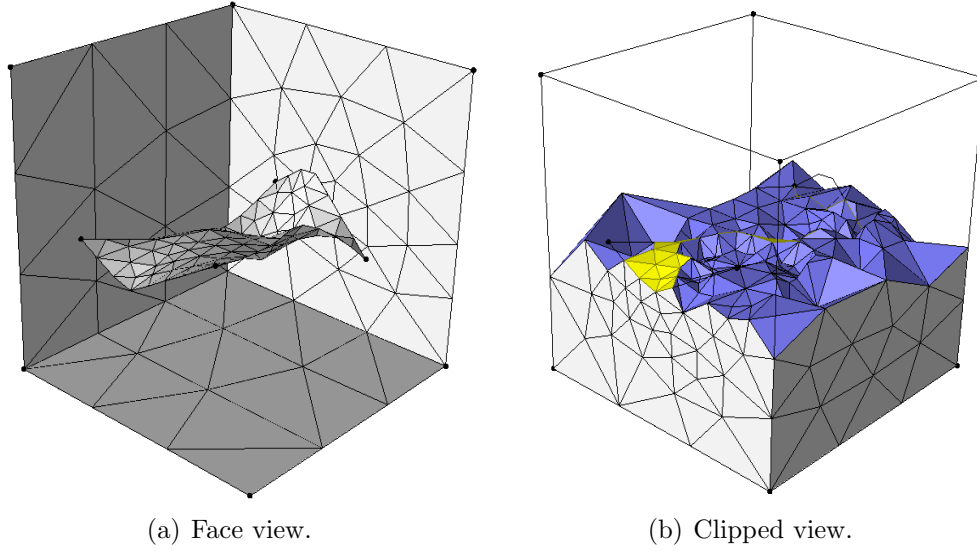


Figure 20: Mesh of model with interior face except for one edge. Yellow mesh element faces indicate the mesh face is coplanar with the model face.

As shown in Figures 15 through 20, the gmd method *place_face()* is able to correctly determine the model classification given user defined surface parameters. It is also able to change the local mesh refinement level to a user's specification.

4 Conclusion

The purpose of this project was to create an interface to the Simmetrix APIs that allows for model and mesh modification. This was to include placement of vertices, edges, and surfaces. The interface was to be general enough to allow for the edges not be limited to straight lines and the surfaces not to be limited to flat planes. Additionally, the modified model was to remain topologically valid to allow for proper discretization and interrogation.

These features were completed in the project. Vertex placement is described in subsection 2.1 and is demonstrated in three examples in subsection 3.1. Edge placement is described in subsection 2.2 and is demonstrated in four examples in subsection 3.2. Face placement is described in subsection 2.3 and is demonstrated in three examples in subsection 3.3. The topology of the lower order entities is also valid since higher order geometric features make use of lower order ones. For example, if the method which placed vertices did not reliably create topologically valid vertices, then neither the *place_edge()* or *place_face()* methods would work properly.

4.1 Future Work

A possible improvement to the created code is to allow for parallel model modification or model discretization. Currently, only one data structure that represents the model exists at any given time. This then leads to one data structure that represents the mesh. A parallel representation could use one common base model with different modification happening on different processors. Another possibility is to have a parallel mesh generated instead of a serial one.

The code currently can only handle native and non-manifold models, not assembly models. Additional work could be done to allow for modification of these types of models. This would mostly include changes to region and part based functions. These changes include accounting for multiple parts and regions along with their interactions.

Work can be done that would allow for geometric entity deletion that preserves topological validity. For example, a user can currently delete any edge. This will yield a topologically invalid model if the edge was one of the bounding edges of a model face. This addition of features could also include methods to trim or extend model entities as well.

References

- [1] Weiler, Kevin J. (1986). Topological Structures for Geometric Modeling. (Doctoral dissertation).
- [2] Zhang, Lijuan. (2013). Microstructural modeling of cross-linked fiber network embedded in continuous matrix. (Doctoral dissertation).
- [3] Askeland, Donald R., Fulay, Pradeep P., Wright, Wendelin J. (2011). The Science and Engineering of Materials. Sixth ed. Stamford, CT: Cengage Learning. Print.
- [4] Simmetrix Inc., "The simulation modeling suite." *[Online]*. Available: <http://www.simmetrix.com/>.

A Code

A.1 main.cpp

```
1 // GeoMod Header
2 #include "GeoMod.hpp"
3
4 #include "GeoMod_Tests.hpp"
5
6 int main( int argc , char** argv)
7 {
8     char sim_log [] = "Sim_log.log";
9     std::cout << "START" << std::endl;
10    GMD::sim_start( sim_log , argc , argv);
11
12    test0();
13    test1();
14    test2();
15    test3();
16    test4();
17    test5();
18    test6();
19    test7();
20    test8();
21    test9();
22    test10();
23    test11();
24    test12();
25
26    GMD::sim_end();
27    std::cout << "END" << std::endl;
28    return 0;
29 }
```

A.2 GeoMod_Tests.hpp

```
1 #ifndef GEOMOD_TESTS_HPP
2 #define GEOMOD_TESTS_HPP
3
4 #include "GeoMod.hpp"
```

```

5
6
7 void test0 ();
8 void test1 ();
9 void test2 ();
10 void test3 ();
11 void test4 ();
12 void test5 ();
13 void test6 ();
14 void test7 ();
15 void test8 ();
16 void test9 ();
17 void test10 ();
18 void test11 ();
19 void test12 ();
20
21 #endif

```

A.3 GeoMod_Tests.cpp

```

1 #include "GeoMod_Tests.hpp"
2 #include <string>
3
4 using std::cout;
5
6 /* test0():
7  *      - Create a 3D model
8  *      - Create a gmd object
9  *      - Test printing features
10 *      - Implicitly destroy the gmd object
11 */
12 void test0 ()
13 {
14     pGModel cube = GMD::create_cube( 10.0);
15     GMD::gmd_t gmd( cube);
16     gmd.test_printers();
17
18     cout << "\n\nPassed test0\n\n" ;
19     return;
20 }

```

```

21
22 /* test1():
23 *      - Create 2D model
24 *      - Create gmd_t instance with model
25 *      - Write model
26 *      - Create mesh from model
27 *      - Write mesh
28 */
29 void test1()
30 {
31     pGModel rectangle = GMD::create_2D_rectangle( 5.0, 7.0)←
32     ;
33     GMD::gmd_t gmd( rectangle);
34     std::string name = "test1_rectangle";
35     gmd.set_name( name);
36     gmd.write_model();
37     gmd.set_global_mesh_params( 1, 0.9, 0.0);
38     gmd.create_mesh();
39     gmd.write_mesh();
40
41     cout << "\n\nPassed test1\n\n";
42     return;
43 }
44
45 /* test2():
46 *      - Create a 3D model
47 *      - Create gmd_t instance with model
48 *      - Write model
49 *      - Create mesh from model
50 *      - Write mesh
51 */
52 void test2()
53 {
54     pGModel cube = GMD::create_cube( 10.0);
55     GMD::gmd_t gmd( cube);
56     std::string name = "test2_cube";
57     gmd.set_name( name);
58     gmd.write_model();
59     gmd.set_global_mesh_params( 1, 0.9, 0.0);

```



```

60     gmd.create_mesh();
61     gmd.write_mesh();
62
63     cout << "\n\nPassed_test2\n\n";
64     return;
65 }
66
67 /* test3():
68  *      - Create a 3D model
69  *      - Place a point with defined mesh refinement in ↵
68         center
70  *      - Write model
71  *      - Create a mesh from model
72  *      - Write mesh
73  */
74 void test3()
75 {
76     pGModel cube = GMD::create_cube( 5.0);
77     GMD::gmd_t gmd( cube);
78     std::string name = "test3_cube";
79     gmd.set_name( name);
80
81     gmd.set_global_mesh_params( 1, 0.5, 0.3);
82     double coords[3] = {0.0, 0.0, 0.0};
83     double refine = 0.01;
84     double radius = 0.01;
85     pGVertex vert;
86     gmd.place_point( coords, refine, radius, vert);
87
88     gmd.write_model();
89     gmd.create_mesh();
90     gmd.write_mesh();
91
92     cout << "\n\nPassed_test3\n\n";
93     return;
94 }
95
96 /* test4():
97  *      - Create a 3D model

```

```

198  *      - Place a point with defined mesh refinement on an ↵
          surface
199  *      - Write model
200  *      - Create mesh from model
201  *      - Write mesh
202  */
203 void test4()
204 {
205     pGModel cube = GMD::create_cube( 2.0);
206     GMD::gmd_t gmd( cube);
207     std::string name = "test4_cube";
208     gmd.set_name( name);
209
210     double coords[3] = {1.0, 0.0, 0.0};
211     double refine = 0.1;
212     double radius = 0.5;
213     pGVertex vert;
214     gmd.place_point( coords, refine, radius, vert);
215
216     gmd.write_model();
217     gmd.set_global_mesh_params( 1, 0.9, 0.0);
218     gmd.create_mesh();
219     gmd.write_mesh();
220
221     cout << "\n\nPassed_test4\n\n";
222     return;
223 }
224
225 /* test5():
226  *      - Create a 3D model
227  *      - Place a point with defined mesh refinement on an ↵
          edge
228  *      - Write model
229  *      - Create mesh from model
230  *      - Write mesh
231  */
232 void test5()
233 {
234     pGModel cube = GMD::create_cube( 2.0);
235     GMD::gmd_t gmd( cube);

```

```

136     std::string name = "test5_cube";
137     gmd.set_name( name);
138
139     double coords[3] = {1.0, 1.0, 0.0};
140     double refine = 0.1;
141     double radius = 0.5;
142     pGVertex vert;
143     gmd.place_point( coords, refine, radius, vert);
144
145     gmd.write_model();
146     gmd.set_global_mesh_params( 1, 0.9, 0.0);
147     gmd.create_mesh();
148     gmd.write_mesh();
149
150     cout << "\n\nPassed_test5\n\n";
151     return;
152 }
153
154 /* test6():
155  *      - Create a 3D model
156  *      - Place a fully interior edge define by three  $\leftarrow$ 
157           points
158  *      - Write model
159  *      - Do not assign name so auto-nameing feature is  $\leftarrow$ 
160           checked
161  *      - Create mesh
162  *      - Write mesh
163  */
164 void test6()
165 {
166     pGModel cube = GMD::create_cube( 2.0);
167     GMD::gmd_t gmd( cube);
168     std::string name = "test6_cube";
169     gmd.set_name( name);
170
171     int order = 4;
172
173     double p1[3] = {0.7, 0.0, 0.0};
174     double p2[3] = {0.0, 0.3, 0.0};
175     double p3[3] = {0.0, 0.0, 0.0};

```

```

174  double p4[3] = {0.0, -0.3, 0.0};
175  double p5[3] = {-0.7, 0.0, 0.0};
176  std::vector<double*> points;
177  points.push_back(p1);
178  points.push_back(p2);
179  points.push_back(p3);
180  points.push_back(p4);
181  points.push_back(p5);
182
183  std::vector<double> knots;
184  knots.push_back(0.0);
185  knots.push_back(0.0);
186  knots.push_back(0.0);
187  knots.push_back(0.0);
188  knots.push_back(0.5);
189  knots.push_back(1.0);
190  knots.push_back(1.0);
191  knots.push_back(1.0);
192  knots.push_back(1.0);
193
194  std::vector<double> weights;
195  weights.push_back(0.0);
196
197  pGEdge edge;
198  double refine = 0.1;
199  gmd.place_edge( order, points, knots, weights, refine, ←
    edge);
200
201  gmd.write_model();
202  gmd.set_global_mesh_params( 1, 0.9, 0.0);
203  gmd.create_mesh();
204  gmd.write_mesh();
205
206  cout << "\n\nPassed test6\n\n";
207  return;
208 }
209
210 /* test7():
211  *      - Create a 3D model

```

```

212 *      - Create one spline based on-face edge defined by  $\leftarrow$ 
      five points
213 *      - Write model
214 *      - Create mesh
215 *      - Write mesh
216 */
217
218 void test7()
219 {
220     pGModel cube = GMD::create_cube( 2.0);
221     GMD::gmdl_t gmd( cube);
222     std::string name = "test7_cube";
223     gmd.set_name( name);
224
225     int order = 4;
226
227     double p1[3] = {1.0, 0.0, 0.9};
228     double p2[3] = {1.0, 0.6, 0.0};
229     double p3[3] = {1.0, 0.0, -0.9};
230     double p4[3] = {1.0, -0.3, 0.0};
231     double p5[3] = {1.0, 0.0, 0.4};
232     std::vector<double*> points;
233     points.push_back(p1);
234     points.push_back(p2);
235     points.push_back(p3);
236     points.push_back(p4);
237     points.push_back(p5);
238
239     std::vector<double> knots;
240     knots.push_back(0.0);
241     knots.push_back(0.0);
242     knots.push_back(0.0);
243     knots.push_back(0.0);
244     knots.push_back(0.5);
245     knots.push_back(1.0);
246     knots.push_back(1.0);
247     knots.push_back(1.0);
248     knots.push_back(1.0);
249
250     std::vector<double> weights;

```

```

251     weights.push_back(0.0);
252
253     pGEdge edge;
254     double refine = 0.1;
255     gmd.place_edge( order , points , knots , weights , refine ,  $\leftarrow$ 
        edge);
256
257     gmd.write_model();
258     gmd.set_global_mesh_params( 1, 0.9, 0.0);
259     gmd.create_mesh();
260     gmd.write_mesh();
261
262     cout << "\n\nPassed_test7\n\n";
263     return;
264 }
265
266 /* test8():
267  *      - Create a 3D model
268  *      - Create an edge from a surface point to an  $\leftarrow$ 
        interior point
269  *      - Write model
270  *      - Create mesh
271  *      - Write mesh
272  */
273 void test8()
274 {
275     pGModel cube = GMD::create_cube( 2.0);
276     GMD::gmd_t gmd( cube);
277     std::string name = "test8_cube";
278     gmd.set_name( name);
279
280     int order = 4;
281
282     double p1[3] = {1.0, 0.0, 0.9};
283     double p2[3] = {0.1, 0.6, 0.0};
284     double p3[3] = {0.3, 0.0, -0.9};
285     double p4[3] = {0.0, -0.3, 0.0};
286     double p5[3] = {0.0, 0.0, 0.4};
287     std::vector<double*> points;
288     points.push_back(p1);

```

```

289     points.push_back(p2);
290     points.push_back(p3);
291     points.push_back(p4);
292     points.push_back(p5);
293
294     std::vector<double> knots;
295     knots.push_back(0.0);
296     knots.push_back(0.0);
297     knots.push_back(0.0);
298     knots.push_back(0.0);
299     knots.push_back(0.5);
300     knots.push_back(1.0);
301     knots.push_back(1.0);
302     knots.push_back(1.0);
303     knots.push_back(1.0);
304
305     std::vector<double> weights;
306     weights.push_back(0.0);
307
308     pGEdge edge;
309     double refine = 0.1;
310     gmd.place_edge( order , points , knots , weights , refine , ←
        edge);
311
312     gmd.write_model();
313     gmd.set_global_mesh_params( 1, 0.9, 0.0);
314     gmd.create_mesh();
315     gmd.write_mesh();
316
317     cout << "\n\nPassed_test8\n\n";
318     return;
319 }
320
321 /* test9();
322  *      - Create a 3D model
323  *      - Create an edge that starts at a pre existing ←
        edge and then terminates
324  *          in the region
325  *      - Write Model
326  *      - Create Mesh

```

```

327 *      - Write Mesh
328 */
329 void test9()
330 {
331     pGModel cube = GMD::create_cube( 2.0);
332     GMD::gmd_t gmd( cube);
333     std::string name = "test9_cube";
334     gmd.set_name( name);
335
336     int order = 4;
337
338     double p1[3] = {1.0, 1.0, 0.9};
339     double p2[3] = {0.1, 0.6, 0.0};
340     double p3[3] = {0.3, 0.0, -0.9};
341     double p4[3] = {0.0, -0.3, 0.0};
342     double p5[3] = {0.0, 0.0, 0.4};
343     std::vector<double*> points;
344     points.push_back(p1);
345     points.push_back(p2);
346     points.push_back(p3);
347     points.push_back(p4);
348     points.push_back(p5);
349
350     std::vector<double> knots;
351     knots.push_back(0.0);
352     knots.push_back(0.0);
353     knots.push_back(0.0);
354     knots.push_back(0.0);
355     knots.push_back(0.5);
356     knots.push_back(1.0);
357     knots.push_back(1.0);
358     knots.push_back(1.0);
359     knots.push_back(1.0);
360
361     std::vector<double> weights;
362     weights.push_back(0.0);
363
364     pGEdge edge;
365     double refine = 0.1;

```



```

366     gmd.place_edge( order , points , knots , weights , refine , ←
           edge);
367
368     gmd.write_model();
369     gmd.set_global_mesh_params( 1, 0.9, 0.0);
370     gmd.create_mesh();
371     gmd.write_mesh();
372
373     cout << "\n\nPassed test9\n\n";
374     return;
375 }
376
377 /* test10();
378  *      - Create a 3D model
379  *      - Create a face completely internal to the region
380  *      - Write Model
381  *      - Create Mesh
382  *      - Write Mesh
383  */
384 void test10()
385 {
386     pGModel cube = GMD::create_cube( 20.0);
387     GMD::gmd_t gmd( cube);
388     std::string name = "test10_cube";
389     gmd.set_name( name);
390
391     int u_order = 3;
392     int v_order = 2;
393     int u_num = 4;
394     int v_num = 3;
395     int periodicity = 0;
396
397     // first 'row'      x      y      z
398     double p1[3]  = { -6.0, -7.0, 3.0};
399     double p2[3]  = { -2.0, -4.0, 0.0};
400     double p3[3]  = { 3.0, -3.0, 1.0};
401     double p4[3]  = { 7.0, -5.0, -3.0};
402     // second 'row'
403     double p5[3]  = { -5.0, 0.0, -1.0};
404     double p6[3]  = { -2.0, 0.0, 3.0};

```

```

405 double p7[3] = { 3.0, 1.0, -7.0};
406 double p8[3] = { 7.0, 0.0, 2.0};
407 // third 'row'
408 double p9[3] = {-6.0, 4.0, 1.0};
409 double p10[3] = {-2.0, 2.0, 0.0};
410 double p11[3] = { 3.0, 4.0, -3.0};
411 double p12[3] = { 7.0, 3.0, 0.0};
412
413 std::vector<double*> points;
414 points.push_back(p1);
415 points.push_back(p2);
416 points.push_back(p3);
417 points.push_back(p4);
418 points.push_back(p5);
419 points.push_back(p6);
420 points.push_back(p7);
421 points.push_back(p8);
422 points.push_back(p9);
423 points.push_back(p10);
424 points.push_back(p11);
425 points.push_back(p12);
426
427 std::vector<double> u_knots;
428 u_knots.push_back(0.0);
429 u_knots.push_back(0.0);
430 u_knots.push_back(0.0);
431 u_knots.push_back(0.5);
432 u_knots.push_back(1.0);
433 u_knots.push_back(1.0);
434 u_knots.push_back(1.0);
435
436 std::vector<double> v_knots;
437 v_knots.push_back(0.0);
438 v_knots.push_back(0.0);
439 v_knots.push_back(0.5);
440 v_knots.push_back(1.0);
441 v_knots.push_back(1.0);
442
443 std::vector<double> weights;
444 weights.push_back(0.0);

```

```

445
446     pGFace face;
447     double refine = 0.1;
448     gmd.place_surface_by_spline(
449         u_order, v_order, u_num, v_num, periodicity,
450         points, u_knots, v_knots, weights,
451         refine, face);
452
453     gmd.write_model();
454     gmd.set_global_mesh_params( 1, 0.9, 0.0);
455     gmd.create_mesh();
456     gmd.write_mesh();
457
458     cout << "\n\nPassed_test10\n\n";
459     return;
460 }
461
462 /* test11();
463 *      - Create a 3D model
464 *      - Create a face completely internal to the region ←
465 *          expect for one point
466 *          which is on a pre-existing surface
467 *      - Write Model
468 *      - Create Mesh
469 *      - Write Mesh
470 */
471 void test11()
472 {
473     pGModel cube = GMD::create_cube( 20.0);
474     GMD::gmd_t gmd( cube);
475     std::string name = "test11_cube";
476     gmd.set_name( name);
477
478     int u_order = 3;
479     int v_order = 2;
480     int u_num = 4;
481     int v_num = 3;
482     int periodicity = 0;
483
484     // first 'row'      x      y      z

```

```

484 double p1[3] = {-6.0, -7.0, 3.0};
485 double p2[3] = {-2.0, -4.0, 0.0};
486 double p3[3] = { 3.0, -3.0, 1.0};
487 double p4[3] = { 7.0, -5.0, -3.0};
488 // second 'row'
489 double p5[3] = {-5.0, 0.0, -1.0};
490 double p6[3] = {-2.0, 0.0, 3.0};
491 double p7[3] = { 3.0, 1.0, -7.0};
492 double p8[3] = { 7.0, 0.0, 2.0};
493 // third 'row'
494 double p9[3] = {-10.0, 4.0, 1.0};
495 double p10[3] = {-2.0, 2.0, 0.0};
496 double p11[3] = { 3.0, 4.0, -3.0};
497 double p12[3] = { 7.0, 3.0, 0.0};
498
499 std::vector<double*> points;
500 points.push_back(p1);
501 points.push_back(p2);
502 points.push_back(p3);
503 points.push_back(p4);
504 points.push_back(p5);
505 points.push_back(p6);
506 points.push_back(p7);
507 points.push_back(p8);
508 points.push_back(p9);
509 points.push_back(p10);
510 points.push_back(p11);
511 points.push_back(p12);
512
513 std::vector<double> u_knots;
514 u_knots.push_back(0.0);
515 u_knots.push_back(0.0);
516 u_knots.push_back(0.0);
517 u_knots.push_back(0.5);
518 u_knots.push_back(1.0);
519 u_knots.push_back(1.0);
520 u_knots.push_back(1.0);
521
522 std::vector<double> v_knots;
523 v_knots.push_back(0.0);

```

```

524     v_knots.push_back(0.0);
525     v_knots.push_back(0.5);
526     v_knots.push_back(1.0);
527     v_knots.push_back(1.0);
528
529     std::vector<double> weights;
530     weights.push_back(0.0);
531
532     pGFace face;
533     double refine = 0.1;
534     gmd.place_surface_by_spline(
535         u_order, v_order, u_num, v_num, periodicity,
536         points, u_knots, v_knots, weights,
537         refine, face);
538
539     gmd.write_model();
540     gmd.set_global_mesh_params( 1, 0.9, 0.0);
541     gmd.create_mesh();
542     gmd.write_mesh();
543
544     cout << "\n\nPassed test11\n\n";
545     return;
546 }
547
548 /* test12();
549 *      - Create a 3D model
550 *      - Create a face completely internal to the region ←
551 *          expect for one edge
552 *          which is on a pre-existing surface
553 *      - Write Model
554 *      - Create Mesh
555 *      - Write Mesh
556 */
557 void test12()
558 {
559     pGModel cube = GMD::create_cube( 20.0);
560     GMD::gmd_t gmd( cube);
561     std::string name = "test12_cube";
562     gmd.set_name( name);

```

```

563  int u_order = 3;
564  int v_order = 2;
565  int u_num = 4;
566  int v_num = 3;
567  int periodicity = 0;
568
569  // first 'row'      x      y      z
570  double p1[3]  = { -10.0, -7.0, 3.0 };
571  double p2[3]  = { -2.0, -4.0, 0.0 };
572  double p3[3]  = { 3.0, -3.0, 1.0 };
573  double p4[3]  = { 7.0, -5.0, -3.0 };
574  // second 'row'
575  double p5[3]  = { -10.0, 0.0, -1.0 };
576  double p6[3]  = { -2.0, 0.0, 3.0 };
577  double p7[3]  = { 3.0, 1.0, -7.0 };
578  double p8[3]  = { 7.0, 0.0, 2.0 };
579  // third 'row'
580  double p9[3]  = { -10.0, 4.0, 1.0 };
581  double p10[3] = { -5.0, 2.0, 0.0 };
582  double p11[3] = { 3.0, 4.0, -3.0 };
583  double p12[3] = { 7.0, 3.0, 0.0 };
584
585  std::vector<double*> points;
586  points.push_back(p1);
587  points.push_back(p2);
588  points.push_back(p3);
589  points.push_back(p4);
590  points.push_back(p5);
591  points.push_back(p6);
592  points.push_back(p7);
593  points.push_back(p8);
594  points.push_back(p9);
595  points.push_back(p10);
596  points.push_back(p11);
597  points.push_back(p12);
598
599  std::vector<double> u_knots;
600  u_knots.push_back(0.0);
601  u_knots.push_back(0.0);
602  u_knots.push_back(0.0);

```

```

603     u_knots.push_back(0.5);
604     u_knots.push_back(1.0);
605     u_knots.push_back(1.0);
606     u_knots.push_back(1.0);
607
608     std::vector<double> v_knots;
609     v_knots.push_back(0.0);
610     v_knots.push_back(0.0);
611     v_knots.push_back(0.5);
612     v_knots.push_back(1.0);
613     v_knots.push_back(1.0);
614
615     std::vector<double> weights;
616     weights.push_back(0.0);
617
618     pGFace face;
619     double refine = 0.1;
620     gmd.place_surface_by_spline(
621         u_order, v_order, u_num, v_num, periodicity,
622         points, u_knots, v_knots, weights,
623         refine, face);
624
625     gmd.write_model();
626     gmd.set_global_mesh_params( 1, 0.9, 0.0);
627     gmd.create_mesh();
628     gmd.write_mesh();
629
630     cout << "\n\nPassed_test12\n\n";
631     return;
632 }

```

A.4 GeoMod_gmd.hpp

```

1 #ifndef GEOMOD_GMD_T_HPP
2 #define GEOMOD_GMD_T_HPP
3
4 #include <string>
5 #include <vector>
6 #include "GeoMod_printer.hpp"
7 #include "GeoMod_model_helper.hpp"

```

```

8 #include "GeoMod_mesh_helper.hpp"
9
10 namespace GMD
11 {
12
13     class gmd_t
14     {
15     public:
16         // Util methods
17         gmd_t( pGModel in_model);
18         ~gmd_t();
19         void set_abort_on_fail( bool abort_on_fail);
20         void test_printers();
21         void verify_mesh();
22
23         // Writing methods
24         void set_name( std::string file_name);
25         void write_model();
26         void write_mesh();
27
28         // Methods to modify the geometry
29         void place_point(
30             double coords[3],
31             double refine,
32             double radius,
33             pGVertex& vert);
34         void place_edge(
35             int order,
36             std::vector<double*> points,
37             std::vector<double> knots,
38             std::vector<double> weights,
39             double refine,
40             pGEdge& edge);
41         void place_surface_by_spline(
42             int u_order,
43             int v_order,
44             int u_num,
45             int v_num,
46             int periodicity,

```



```

47         // 0=none, 1=u periodic, 2=v periodic, 3=u&v ←
           periodic
48         std::vector<double*> points,
49         std::vector<double> u_knots,
50         std::vector<double> v_knots,
51         std::vector<double> weights,
52         double refine,
53         pGFace& face);
54
55     // Methods for only meshing
56     void set_global_mesh_params(
57         int order_in,
58         double refine_in,
59         double grad_rate_in);
60     void create_mesh();
61
62     private:
63         bool panicStatus;
64         std::string name;
65         model_helper_t* modeler;
66         mesh_helper_t* mesher;
67         void check_spline_params(
68             int order,
69             std::vector<double*> points,
70             std::vector<double> knots,
71             std::vector<double> weights);
72         void check_surface_params(
73             int u_order,
74             int v_order,
75             int u_num,
76             int v_num,
77             int periodicity,
78             std::vector<double*> points,
79             std::vector<double> u_knots,
80             std::vector<double> v_knots,
81             std::vector<double> weights);
82     };
83
84 }
85

```

86 **#endif**

A.5 GeoMod_gmd.cpp

```
1 #include "GeoMod_gmd_t.hpp"
2
3 namespace GMD
4 {
5     gmd_t::gmd_t( pGModel in_model)
6     {
7         panicStatus = true;
8         modeler = new model_helper_t( in_model);
9         mesher = new mesh_helper_t ( in_model);
10        return;
11    }
12
13    gmd_t::~~gmd_t()
14    {
15        delete modeler;
16        delete mesher;
17        return;
18    }
19
20    void gmd_t::set_abort_on_fail( bool abort_on_fail)
21    {
22        panicStatus = abort_on_fail;
23        return;
24    }
25
26    void gmd_t::test_printers()
27    {
28        modeler->model_print();
29        mesher->mesh_print();
30        return;
31    }
32
33    void gmd_t::place_point( double coords[3], double ←
34        refine, double radius, pGVertex& vert)
35    {
```

```

35     bool updateMesh = modeler->place_point( coords , vert , ←
        panicStatus);
36     if( updateMesh)
37     {
38         mesher->place_point( coords , refine , radius , ←
            panicStatus);
39     }
40
41     return;
42 }
43
44 void gmd_t::set_name( std::string file_name)
45 {
46     name = file_name;
47     return;
48 }
49
50 void gmd_t::check_spline_params(
51     int order ,
52     std::vector<double*> points ,
53     std::vector<double> knots ,
54     std::vector<double> weights)
55 {
56     if( order <=1)
57     { print_error("Spline_order_too_low..Must_be_←
        polynomial_order_+1.");}
58     if( (int)points.size() < order)
59     { print_error("Spline_order_too_high_for_number_of_←
        given_control_points.");}
60     if( ((int)points.size()+order) != (int)knots.size())
61     { print_error("Mismatch_between_knot_vector_size_and_←
        sum_of_order_with_control_point_size.");}
62
63     for( int i=0; i<(int)knots.size(); i++)
64     {
65         double tmp1 = knots[i];
66         if( i<((int)knots.size() -1) )
67         {
68             double tmp2 = knots[i+1];
69             if (tmp2 < tmp1)

```

```

70         { print_error("Knots must be in ascending order")↵
71             ;}
72     }
73     if( i<order && tmp1 != 0.0)
74     { print_error("First order number of knots must be ↵
75         zero.");}
76     if( i>((int) knots.size()-order) && tmp1 != 1.0)
77     { print_error("Last order number of knots must be ↵
78         one.");}
79
80     if( tmp1 > 1.0 || tmp1 < 0.0)
81     {
82         print_error("Knots must satisfy 0.0<=k[i]<=1.0. ↵
83             ");
84     }
85     if( (int) weights.size() != 1 && (int) weights.size() ↵
86         != (int) points.size())
87     { print_error("Mismatch between weight vector size ↵
88         and number of points.");}
89     return;
90 }
91
92 void gmd_t::place_edge(
93     int order ,
94     std::vector<double*> points ,
95     std::vector<double> knots ,
96     std::vector<double> weights ,
97     double refine ,
98     pGEdge& edge)
99 {
100     check_spline_params( order , points , knots , weights);
101     modeler->place_edge( order , points , knots , weights , ↵
102         edge);
103     if( refine > 0.0)
104     { mesher->refine_edge( refine , edge);}
105     return;
106 }

```

```

103 void gmd_t::write_model()
104 {
105     modeler->write( name);
106     return;
107 }
108
109 void gmd_t::create_mesh()
110 {
111     if(modeler->isValid())
112     {
113         mesher->create();
114     }
115     else if( (modeler->isValid()) && !panicStatus)
116     {
117         print_warning(" Attempting to mesh invalid model.");
118     }
119     else
120     {
121         print_error(" Invalid model. No mesh created");
122     }
123     return;
124 }
125
126 void gmd_t::verify_mesh()
127 {
128     if(!mesher->isValid())
129     {
130         if(panicStatus)
131             { print_error(" Mesh Not Valid.");}
132         else if( !panicStatus)
133             { print_warning(" Mesh Not Valid.");}
134     }
135     return;
136 }
137
138 void gmd_t::write_mesh()
139 {
140     if(modeler->isWritten())
141     { mesher->write( name); }
142     else

```

```

143     { print_error("Model must be written before mesh."); ↵
144     }
145     return;
146 }
147 void gmd_t::set_global_mesh_params( int order_in, ↵
148     double refine_in, double grad_rate_in)
149 {
150     mesher->set_global( order_in, refine_in, grad_rate_in↵
151     );
152     return;
153 }
154 void gmd_t::place_surface_by_spline(
155     int u_order,
156     int v_order,
157     int u_num,
158     int v_num,
159     int periodicity,
160     std::vector<double*> points,
161     std::vector<double> u_knots,
162     std::vector<double> v_knots,
163     std::vector<double> weights,
164     double refine,
165     pGFace& face)
166 {
167     check_surface_params(
168         u_order, v_order, u_num, v_num, periodicity, ↵
169         points, u_knots, v_knots, weights);
170
171     modeler->place_surface_by_spline(
172         u_order, v_order, u_num, v_num, periodicity, ↵
173         points, u_knots, v_knots, weights, face);
174
175     mesher->refine_face( refine, face);
176
177     return;
178 }
179 void gmd_t::check_surface_params(

```

```

178     int u_order ,
179     int v_order ,
180     int u_num ,
181     int v_num ,
182     int periodicity ,
183     std::vector<double*> points ,
184     std::vector<double> u_knots ,
185     std::vector<double> v_knots ,
186     std::vector<double> weights)
187 {
188     if( periodicity < 0 || periodicity > 4)
189     { print_error("Bad_periodicity.");}
190
191     if( u_order<= 1 )
192     { print_error("u_order_too_low.");}
193     if( v_order<= 1 )
194     { print_error("v_order_too_low.");}
195
196     if( (int)points.size() != (u_num*v_num))
197     { print_error("Mismatched_number_of_declared_and↵
        given_control_points.");}
198
199     if( (int)weights.size() != 1 && (int)weights.size() ↵
        != (u_num*v_num))
200     { print_error("Mismatched_number_of_weights_and↵
        declared_control_points.");}
201     if( (int)weights.size() == 1 && weights[0] != 0.0)
202     { print_error("Bad_weights_vector.");}
203
204     if( (int)u_knots.size() != (u_num+u_order))
205     { print_error("Mismatch_between_number_of_u_knots_and↵
        u_order+u_num.");}
206
207     for( int i=0; i<(int)u_knots.size(); i++)
208     {
209         double tmp1 = u_knots[i];
210         if( i<((int)u_knots.size() -1) )
211         {
212             double tmp2 = u_knots[i+1];
213             if (tmp2 < tmp1)

```

```

214         { print_error("Knots must be in ascending u_order↵
215             ");}
216     }
217     if( i<u_order && tmp1 != 0.0)
218     { print_error("First u_order number of u_knots must↵
219         be zero.");}
220     if( i>((int)u_knots.size()-u_order) && tmp1 != 1.0)
221     { print_error("Last u_order number of u_knots must↵
222         be one.");}
223
224     if( tmp1 > 1.0 || tmp1 < 0.0)
225     {
226         print_error("Knots must satisfy 0.0<=k[i]<=1.0.↵
227             ");
228     }
229
230     if( (int)v_knots.size() != (v_num+v_order))
231     { print_error("Mismatch between number of v_knots and↵
232         v_order+v_num.");}
233
234     for( int i=0; i<(int)v_knots.size(); i++)
235     {
236         double tmp1 = v_knots[i];
237         if( i<((int)v_knots.size() -1) )
238         {
239             double tmp2 = v_knots[i+1];
240             if (tmp2 < tmp1)
241             { print_error("Knots must be in ascending v_order↵
242                 ");}
243         }
244         if( i<v_order && tmp1 != 0.0)
245         { print_error("First v_order number of v_knots must↵
246             be zero.");}
247         if( i>((int)v_knots.size()-v_order) && tmp1 != 1.0)
248         { print_error("Last v_order number of v_knots must↵
249             be one.");}
250
251         if( tmp1 > 1.0 || tmp1 < 0.0)
252         {

```



```

246         print_error("Knots must satisfy  $0.0 \leq k[i] \leq 1.0$ ." ←
247             );
248     }
249 }
250     return;
251 }
252 }

```

A.6 GeoMod_model_helper.hpp

```

1  #ifndef GEOMOD_MODEL_HELPER_HPP
2  #define GEOMOD_MODEL_HELPER_HPP
3
4  #include <string>
5  #include <vector>
6  #include "GeoMod_SIM.hpp"
7  #include "GeoMod_printer.hpp"
8  #include "GeoMod_coords.hpp"
9
10 pGVertex GIP_insertVertexInFace(pGIPart part, double* xyz ←
    , pGFace face);
11
12 namespace GMD
13 {
14     class model_helper_t
15     {
16     friend class gmd_t;
17     private:
18         // Util methods
19         model_helper_t (pGModel& in_model);
20         ~model_helper_t ();
21         void model_print();
22         void write( std::string name);
23         bool isValid();
24         bool isWritten();
25         void unpack_vector_spline_points(
26             std::vector<double*> vec,
27             double* x);

```

```

28 void unpack_vector( std::vector<double> vec, double←
    * x);
29 void unpack_surface_vector(
30     std::vector<double*> points,
31     double* all_points);
32 void create_bounding_edges(
33     int u_order,
34     int v_order,
35     int u_num,
36     int v_num,
37     std::vector<double*> points,
38     std::vector<double> u_knots,
39     std::vector<double> v_knots,
40     std::vector<double> weights,
41     std::vector<pGEdge>& edges);
42 void unpack_bounding_edges(
43     std::vector<pGEdge>& edges,
44     pGEdge* bounding_edges);
45
46 // Members
47 pGModel model;
48 pGIPart part;
49 bool Written;
50
51 // Methods to place a point
52 bool place_point(
53     double coords[3],
54     pGVertex& vert,
55     bool abort_on_fail);
56 bool point_on_dim( int dim, double coords[3]);
57 int point_location( double coords[3]);
58 bool PointOnFace( double coords[3], pGFace face);
59 bool PointOnEdge( double coords[3], pGEdge edge);
60 void put_point_outside( double coords[3], pGVertex&←
    vert);
61 void put_point_in_line( double coords[3], pGVertex&←
    vert);
62 void put_point_in_face( double coords[3], pGVertex&←
    vert);

```

```

63 void put_point_in_region( double coords[3], ←
    pGVertex& vert);
64
65 // Methods to place an edge
66 void place_edge(
67     int order,
68     std::vector<double*> points,
69     std::vector<double> knots,
70     std::vector<double> weights,
71     pGEdge& edge);
72 void create_curve(
73     int order,
74     std::vector<double*> points,
75     std::vector<double> knots,
76     std::vector<double> weights,
77     pCurve& curve);
78 void create_edge(
79     int order,
80     std::vector<double*> points,
81     pCurve& curve,
82     pGEdge& edge);
83 bool PointsOnSameFace( std::vector<double*> points)←
    ;
84
85 // Methods to place a surface
86 void place_surface_by_spline(
87     int u_order,
88     int v_order,
89     int u_num,
90     int v_num,
91     int periodicity,
92     std::vector<double*> points,
93     std::vector<double> u_knots,
94     std::vector<double> v_knots,
95     std::vector<double> weights,
96     pGFace& face);
97 void create_surface(
98     int u_order,
99     int v_order,
100    int u_num,

```

```

101         int v_num,
102         int periodicity ,
103         std::vector<double*> points ,
104         std::vector<double> u_knots ,
105         std::vector<double> v_knots ,
106         std::vector<double> weights ,
107         pSurface& surface ,
108         std::vector<pGEdge>& edges);
109     void create_face(
110         pSurface& surface ,
111         std::vector<pGEdge>& edges ,
112         pGFace& face);
113
114     };
115
116 }
117
118 #endif

```

A.7 GeoMod_model_helper.cpp

```

1 #include "GeoMod_model_helper.hpp"
2
3 namespace GMD
4 {
5     model_helper_t::model_helper_t( pGModel& in_model)
6     {
7         model = in_model;
8         Written = false;
9         if (!isValid())
10         {
11             print_warning("Created_gmd_object_on_invalid_model.↵
12                             ");
13         }
14         part = GM_part( model);
15         if( part == NULL)
16         {
17             print_error("GeoMod_does_not_support_assembly_↵
18                             models.");
19         }
20     }

```

```

18     return;
19 }
20
21 model_helper_t::~~model_helper_t()
22 {
23     GM_release( model);
24     return;
25 }
26
27 void model_helper_t::model_print()
28 {
29     std::cout << "Modeler_says_hello!" << std::endl;
30     return;
31 }
32
33 bool model_helper_t::isValid()
34 {
35     if(GM_isValid( model, 1, NULL) == 1)
36     { return true; }
37     else
38     { return false; }
39 }
40
41 void model_helper_t::write( std::string name)
42 {
43     name = name + ".smd";
44     const char* name_c = name.c_str();
45     if( !isValid())
46     { print_warning("Attempting_to_write_invalid_model.")↵
47         ;}
48
49     std::cout << "MODEL_INFORMATION:↵"
50         << "\nVertices:↵"<< GM_numVertices(model)
51         << "\nEdges:↵"<< GM_numEdges(model)
52         << "\nFaces:↵"<< GM_numFaces(model)
53         << "\nRegions:↵"<< GM_numRegions(model) << std::↵
54         endl;
55
56     int writestat = GM_write(model, name_c, 0,0);
57     if(writestat == 0)

```

```

56     {
57         std::cout << "Model_" << name << "_written." << std::endl;
58         Written = true;
59     }
60     else
61     {
62         std::cout << "Model_" << name
63             << "_failed_to_be_written." << std::endl;
64     }
65
66     return;
67 }
68
69 bool model_helper_t::point_on_dim( int dim, double coords[3])
70 {
71     double tol = GM_tolerance( model);
72     bool answer = false;
73     bool areSame = false;
74     double closest [] = {0.0, 0.0, 0.0};
75     if( dim == 1)
76     {
77         GEIter e_it = GM_edgeIter( model);
78         pGEdge e;
79         while (( e = GEIter_next(e_it)))
80         {
81             GE_closestPoint(e, coords, closest, NULL);
82             compare_coords(coords, closest, areSame, tol);
83             if(areSame)
84             {
85                 answer = true;
86             }
87         }
88         GEIter_delete( e_it);
89     }
90     else if (dim == 2)
91     {
92         GFIter f_it = GM_faceIter( model);
93         pGFace f;

```

```

94     while (( f = GFilter_next(f_it)))
95     {
96         GF_closestPoint(f, coords, closest, NULL);
97         compare_coords(coords, closest, areSame, tol);
98         if(areSame)
99         {
100             answer = true;
101         }
102     }
103     GFilter_delete( f_it);
104 }
105 else if (dim == 3)
106 {
107     print_warning("point_on_dim_does_not_support_this_↵
108                 dimension");
109 }
110 return answer;
111 }
112
113 int model_helper_t::point_location(double coords[3])
114 {
115     int answer = 0;
116     // Want to find classification on lowest gEnt dim
117     // start from top work down, overwrite old answer
118     for(int i=3; i>0; i--)
119     {
120         if (i<3)
121         {
122             if(point_on_dim( i, coords))
123             {
124                 answer = i;
125             }
126         }
127         else
128         {
129             GRIter r_it = GM_regionIter( model);
130             pGRegion region;
131             while(( region= GRIter_next(r_it)))
132             {
133                 if(GR_containsPoint( region, coords) == 0)

```

```

133         { // Point is in the void region (spooky!)
134             return 0;
135         }
136         else if (GR_containsPoint( region , coords) ==1)
137         {
138             answer = 3;
139         }
140         else
141         { print_error("Point_placement_not_possible.")↵
            ;}
142     }
143     GRIter_delete( r_it);
144 }
145 }
146 return answer;
147 }
148
149 bool model_helper_t::PointOnFace( double coords[3] , ↵
    pGFace face)
150 {
151     double tol = GM_tolerance( model);
152     bool ans = false;
153     double cp[] = {0.0, 0.0, 0.0};
154     GF_closestPoint( face , coords , cp, NULL);
155     compare_coords(coords , cp , ans , tol);
156     return ans;
157 }
158
159 bool model_helper_t::PointOnEdge( double coords[3] , ↵
    pGEdge edge)
160 {
161     double tol = GM_tolerance( model);
162     bool ans = false;
163     double cp[] = {0.0, 0.0, 0.0};
164     double para[] = {0.0, 0.0, 0.0};
165     GE_closestPoint( edge , coords , cp , para);
166     compare_coords(coords , cp , ans , tol);
167
168     return ans;
169 }

```



```

170
171 void model_helper_t::put_point_outside( double coords↵
    [3], pGVertex& vert)
172 {
173     pGRegion out_region = GIP_outerRegion( part);
174     vert = GIP_insertVertexInRegion( part, coords, ↵
        out_region);
175     return;
176 }
177
178 void model_helper_t::put_point_in_line( double coords↵
    [3], pGVertex& vert)
179 {
180     bool placed = false;
181     GEIter e_it = GM_edgeIter( model);
182     pGEdge edge;
183     while ( !placed && (edge = GEIter_next( e_it)))
184     {
185         if (PointOnEdge(coords,edge))
186         {
187             double param = 0.0;
188             GE_closestPoint( edge, coords, NULL, &param);
189             vert = GM_splitEdge( edge, param);
190             if(vert == NULL)
191             {
192                 pPList vert_list = PList_new();
193                 vert_list = GE_vertices( edge);
194                 pGVertex tmp_vert;
195                 void* iter = 0;
196                 bool found = false;
197                 while((!found) && ( tmp_vert = (pGVertex) ↵
                    PList_next( vert_list, &iter)))
198                 {
199                     bool areSame = false;
200                     double tol = GM_tolerance( model);
201                     double conf_coords[3] = {0.0};
202                     GV_point( tmp_vert, conf_coords);
203                     compare_coords( coords, conf_coords, areSame,↵
                        tol);
204                     if( areSame)

```

```

205         {
206             found = true;
207             vert = tmp_vert;
208         }
209     }
210     PList_delete( vert_list);
211 }
212 placed = true;
213 }
214 }
215 GEIter_delete( e_it);
216
217 if( !placed)
218 {
219     print_warning("Failed_to_place_point_on_edge_at");
220     print_coords( coords);
221 }
222 return;
223 }
224
225 void model_helper_t::put_point_in_face( double coords←
226     [3], pGVertex& vert)
227 {
228     bool placed = false;
229     GFIter f_it = GM_faceIter( model);
230     pGFace face;
231     while ( !placed && (face = GFIter_next( f_it)))
232     {
233         if (PointOnFace(coords, face))
234         {
235             vert = GIP_insertVertexInFace( part, coords, face←
236                 );
237             placed = true;
238         }
239     }
240     GFIter_delete( f_it);
241
242     if( !placed)
243     {
244         print_warning("Failed_to_place_point_in_face_at");

```

```

243     print_coords( coords);
244 }
245
246     return;
247 }
248
249 void model_helper_t::put_point_in_region(
250     double coords[3],
251     pGVertex& vert)
252 {
253     bool placed = false;
254     GRIter r_it = GM_regionIter( model);
255     pGRegion region;
256     while( !placed && ( region= GRIter_next(r_it)))
257     {
258         if (GR_containsPoint( region , coords) ==1)
259         {
260             vert = GIP_insertVertexInRegion( part , coords , ←
                region);
261             placed = true;
262         }
263     }
264     GRIter_delete( r_it);
265     return;
266 }
267
268 bool model_helper_t::place_point(
269     double coords[3],
270     pGVertex& vert ,
271     bool abort_on_fail)
272 {
273     bool updateMesh = true;
274     int location = point_location(coords);
275     if( location == 0)
276     {
277         print_warning("Point_outside_of_known_regions.");
278         updateMesh = false;
279         put_point_outside( coords , vert);
280     }
281     else if ( location == 1)

```

```

282     {
283         put_point_in_line( coords, vert);
284     }
285     else if ( location == 2)
286     {
287         put_point_in_face( coords, vert);
288     }
289     else if ( location == 3)
290     {
291         put_point_in_region( coords, vert);
292     }
293     (void) abort_on_fail;
294     return updateMesh;
295 }
296
297 void model_helper_t::unpack_vector_spline_points(
298     std::vector<double*> vec,
299     double* x)
300 {
301     int pos = 0;
302     double* tmp = vec[0];
303     for( int i=0; i<(int)vec.size(); i++)
304     {
305         tmp = vec[i];
306         for( int j=0; j<3; j++)
307         {
308             pos = j+3*i;
309             x[pos] = tmp[j];
310         }
311     }
312     return;
313 }
314
315 void model_helper_t::unpack_vector( std::vector<double>←
316     vec, double* x)
317 {
318     for (int i=0; i<(int)vec.size(); i++)
319     {
320         x[i] = vec[i];
321     }

```

```

321     return;
322 }
323
324 void model_helper_t::create_curve(
325     int order ,
326     std::vector<double*> points ,
327     std::vector<double> knots ,
328     std::vector<double> weights ,
329     pCurve& curve)
330 {
331     int num_points = (int)points.size();
332     bool weightLess = false;
333     if((int)weights.size() == 1 && weights[0] == 0.0)
334     { weightLess = true; }
335
336     double u_points[num_points*3] = {0.0};
337     double u_knots[(int)knots.size()] = {0.0};
338     double un_weights[(int)weights.size()] = {0.0};
339     unpack_vector_spline_points( points , u_points);
340     unpack_vector( knots , u_knots);
341
342     if(weightLess)
343     {
344         curve = SCurve_createBSpline(
345             order , num_points , u_points , u_knots , NULL);
346     }
347     else
348     {
349         unpack_vector( weights , un_weights);
350         curve = SCurve_createBSpline(
351             order , num_points , u_points , u_knots , un_weights)↵
352             ;
353     }
354     return;
355 }
356
357 bool model_helper_t::PointsOnSameFace( std::vector<↵
358     double*> points)
359 {
360     bool onFace = true;

```

```

359     for( int i=0; i<(int)points.size(); i++)
360     { // Check if all points are on any face
361         if( !point_on_dim( 2, points[i]))
362         {
363             onFace = false;
364         }
365     }
366
367     if( onFace)
368     {
369         bool answer = false;
370         pGFace face;
371         pGFace conFace;
372         GFIter f_it = GM_faceIter( model);
373         while(( face = GFIter_next(f_it)))
374         {
375             for(int i=0; i<(int)points.size(); i++)
376             {
377                 if( (i==0) && PointOnFace( points[i], face))
378                 {
379                     conFace = face;
380                 }
381                 else if( PointOnFace( points[i], face))
382                 {
383                     if( face == conFace)
384                     { answer = true; }
385                     else
386                     { answer = false; }
387                 }
388             }
389         }
390         GFIter_delete(f_it);
391         return answer;
392     }
393     else
394     {
395         return false;
396     }
397 }
398

```

```

399 void model_helper_t::create_edge(
400     int order,
401     std::vector<double*> points,
402     pCurve& curve,
403     pGEdge& edge)
404 {
405     double* start_point = points[0];
406     pGVertex start_vert = NULL;
407     if( start_point == NULL)
408     { print_error("_start_point_is_NULL_");}
409     place_point( start_point, start_vert, false);
410
411     double* end_point = points[(int)points.size()-1];
412     pGVertex end_vert = NULL;
413     place_point( end_point, end_vert, false);
414
415     // Assumes a one region, one part model
416     pGIPart part = GM_part( model);
417     GRIter r_it = GM_regionIter( model);
418     pGRegion region = GRIter_next( r_it);
419
420     if( part == NULL)
421     { print_error("Part_is_NULL");}
422     if( start_vert == NULL)
423     { print_error("start_vert_is_NULL");}
424     if( end_vert == NULL)
425     { print_error("end_vert_is_NULL");}
426     if( curve == NULL)
427     { print_error("curve_is_NULL");}
428     if( region == NULL)
429     { print_error("region_is_NULL");}
430
431     edge = GIP_insertEdgeInRegion(
432         part, start_vert, end_vert, curve, 1, region);
433     GRIter_delete( r_it);
434
435     bool onSameFace = PointsOnSameFace( points);
436     if( onSameFace)
437     {
438         pGFace face;

```

```

439     GFIter f_it = GM_faceIter( model);
440     bool found = false;
441     while( !found && (face = GFIter_next(f_it)))
442     {
443         if(PointOnFace(points[0], face))
444         { found = true; }
445     }
446     GFIter_delete( f_it);
447     pGFace new_faces[2] = {NULL, NULL};
448     GM_insertEdgeOnFace( face, edge, new_faces);
449 }
450
451 (void) order;
452 return;
453 }
454
455 void model_helper_t::place_edge(
456     int order,
457     std::vector<double*> points,
458     std::vector<double> knots,
459     std::vector<double> weights,
460     pGEdge& edge)
461 {
462     pCurve curve;
463     create_curve( order, points, knots, weights, curve);
464     create_edge( order, points, curve, edge);
465     return;
466 }
467
468 void model_helper_t::place_surface_by_spline(
469     int u_order,
470     int v_order,
471     int u_num,
472     int v_num,
473     int periodicity,
474     std::vector<double*> points,
475     std::vector<double> u_knots,
476     std::vector<double> v_knots,
477     std::vector<double> weights,
478     pGFace& face)

```



```

479 {
480     pSurface surf;
481     std::vector<pGEdge> edges;
482
483     create_surface(
484         u_order, v_order, u_num, v_num, periodicity,
485         points, u_knots, v_knots, weights, surf, edges);
486     create_face( surf, edges, face);
487     return;
488 }
489
490 void model_helper_t::unpack_bounding_edges(
491     std::vector<pGEdge>& edges,
492     pGEdge* bounding_edges)
493 {
494     for( int i=0; i<(int)edges.size(); i++)
495     {
496         bounding_edges[i] = edges[i];
497     }
498     return;
499 }
500
501 void model_helper_t::create_face(
502     pSurface& surface,
503     std::vector<pGEdge>& edges,
504     pGFace& face)
505 {
506
507     pGIPart part = GM_part( model);
508     int numEdges = (int)edges.size();
509     pGEdge bounding_edges[numEdges] = {NULL};
510     unpack_bounding_edges( edges, bounding_edges);
511     int dirs[numEdges] = {0};
512     for( int i = 0; i< numEdges; i++)
513     {
514         dirs[i] = 1;
515     }
516     int numLoops = 1;
517     int indLoop[numLoops] = {0};
518     for( int i = 0; i< numLoops; i++)

```

```

519     {
520         indLoop[i] = 0;
521     }
522     int normal = 1;
523
524     GRIter r_it = GM_regionIter( model);
525     pGRegion region = GRIter_next( r_it);
526
527     face = GIP_insertFaceInRegion(
528         part, numEdges, bounding_edges, dirs,
529         numLoops, indLoop, surface, normal, region);
530
531     GRIter_delete( r_it);
532     return;
533 }
534
535 void model_helper_t::create_bounding_edges(
536     int u_order,
537     int v_order,
538     int u_num,
539     int v_num,
540     std::vector<double*> points,
541     std::vector<double> u_knots,
542     std::vector<double> v_knots,
543     std::vector<double> weights,
544     std::vector<pGEdge>& edges)
545 {
546     // Since GeomSim needs splines surface to be four ←
547     sided
548     // and have a regular control point spacing (X by Y),
549     // the edges need can be infered from the surface ←
550     points
551
552     int N = u_num;
553     int M = v_num;
554     pGEdge tmp_edge0;
555     bool weightLess = false;
556     if((int)weights.size() == 1 && weights[0] == 0.0)
557     { weightLess = true;}
```

```

557     std::vector<double*> edge_points;
558     std::vector<double> edge_weights;
559     for( int n = 0; n<N; n++)
560     {
561         edge_points.push_back( points[n] );
562         if(!weightLess)
563         {
564             edge_weights.push_back( weights[n]);
565         }
566     }
567
568     if(weightLess)
569     {
570         edge_weights.push_back( 0.0);
571     }
572     place_edge( u_order , edge_points , u_knots ,  $\leftrightarrow$ 
573               edge_weights , tmp_edge0);
574     edges.push_back(tmp_edge0);
575     edge_points.clear();
576
577     pGEdge tmp_edge1;
578     for( int m=0; m<M; m++)
579     {
580         int ind = (N-1)+m*N;
581         edge_points.push_back( points[ind] );
582         if(!weightLess)
583         {
584             edge_weights.push_back( weights[ind]);
585         }
586     }
587
588     place_edge( v_order , edge_points , v_knots ,  $\leftrightarrow$ 
589               edge_weights , tmp_edge1);
590     edges.push_back(tmp_edge1);
591
592     edge_points.clear();
593
594     pGEdge tmp_edge2;
595     for( int n=(N-1); n>(-1); n--)
596     {

```

```

595     int ind = n+(M-1)*N;
596     edge_points.push_back( points[ind] );
597     if(!weightLess)
598     {
599         edge_weights.push_back( weights[ind] );
600     }
601 }
602
603 place_edge( u_order , edge_points , u_knots ,  $\leftrightarrow$ 
604             edge_weights , tmp_edge2);
605 edges.push_back(tmp_edge2);
606 edge_points.clear();
607
608 pGEdge tmp_edge3;
609 for( int m=(M-1); m>(-1); m--)
610 {
611     int ind = N*m;
612     edge_points.push_back( points[ind] );
613     if(!weightLess)
614     {
615         edge_weights.push_back( weights[ind] );
616     }
617 }
618
619 place_edge( v_order , edge_points , v_knots ,  $\leftrightarrow$ 
620             edge_weights , tmp_edge3);
621 edges.push_back(tmp_edge3);
622
623 edge_points.clear();
624 edge_weights.clear();
625
626 return;
627 }
628
629 void model_helper_t::create_surface(
630     int u_order ,
631     int v_order ,
632     int u_num ,
633     int v_num ,
634     int periodicity ,

```

```

633     std::vector<double*> points ,
634     std::vector<double> u_knots ,
635     std::vector<double> v_knots ,
636     std::vector<double> weights ,
637     pSurface& surface ,
638     std::vector<pGEdge>& edges )
639 {
640     int u_per = 0;
641     int v_per = 0;
642     if (periodicity == 0)
643     { // No chages from default
644     }
645     else if (periodicity == 1)
646     {
647         u_per = 1;
648     }
649     else if (periodicity == 2)
650     {
651         v_per = 1;
652     }
653     else if (periodicity == 3)
654     {
655         u_per = 1;
656         v_per = 1;
657     }
658
659     bool weightLess = false;
660     if((int)weights.size() == 1 && weights[0] == 0.0)
661     { weightLess = true;}
662
663     int num_points = (int)points.size();
664
665     double unp_u_knots[(int)u_knots.size()] = {0.0};
666     unpack_vector( u_knots , unp_u_knots);
667
668     double unp_v_knots[(int)v_knots.size()] = {0.0};
669     unpack_vector( v_knots , unp_v_knots);
670
671     double all_points[3*num_points] = {0.0};
672     unpack_vector_spline_points( points , all_points);

```

```

673
674     create_bounding_edges(
675         u_order , v_order , u_num , v_num ,
676         points , u_knots , v_knots , weights , edges);
677
678     if(weightLess)
679     {
680         surface = SSurface_createBSpline(
681             u_order , v_order ,
682             u_num , v_num ,
683             u_per , v_per ,
684             all_points , NULL ,
685             unp_u_knots , unp_v_knots);
686     }
687     else
688     {
689         double unp_weights[num_points] = {0.0};
690         unpack_vector( weights , unp_weights);
691         surface = SSurface_createBSpline(
692             u_order , v_order ,
693             u_num , v_num ,
694             u_per , v_per ,
695             all_points , unp_weights ,
696             unp_u_knots , unp_v_knots);
697     }
698     return;
699 }
700
701 bool model_helper_t::isWritten()
702 { return Written;}
703 }

```

A.8 GeoMod_mesh_helper.hpp

```

1 #ifndef GEOMOD_MESH_HELPER_HPP
2 #define GEOMOD_MESH_HELPER_HPP
3
4 #include <string>
5 #include <vector>
6 #include "GeoMod_SIM.hpp"

```

```

7 #include "GeoMod_printer.hpp"
8
9 namespace GMD
10 {
11     class mesh_helper_t
12     {
13     friend class gmd_t;
14     private:
15         // Util methods
16         mesh_helper_t( pGModel in_model);
17         ~mesh_helper_t();
18         void mesh_print();
19         void write( std::string name);
20         bool isValid();
21         void create();
22
23         // Members
24         pMesh mesh;
25         pACase m_case;
26         bool globalSet;
27         double order;
28         double refine;
29         double grad_rate;
30
31         // Mesh preping methods
32         void place_point(
33             double coords[3],
34             double refine,
35             double radius,
36             bool abort_on_fail);
37         void set_global( int order_in, double refine_in, ←
38             double grad_rate_in);
39         void refine_vertex( double refine, pGVertex vert);
40         void refine_edge( double refine, pGEdge edge);
41         void refine_face( double refine, pGFace face);
42     };
43
44 }
45

```

46 **#endif**

A.9 GeoMod_mesh_helper.cpp

```
1 #include "GeoMod_mesh_helper.hpp"
2
3 namespace GMD
4 {
5     mesh_helper_t::mesh_helper_t( pGModel in_model)
6     {
7         mesh = M_new( 0, in_model);
8         m_case = MS_newMeshCase( in_model);
9         globalSet = false;
10
11         return;
12     }
13
14     mesh_helper_t::~~mesh_helper_t()
15     {
16         MS_deleteMeshCase(m_case);
17         M_release( mesh);
18         return;
19     }
20
21     void mesh_helper_t::mesh_print()
22     {
23         std::cout << "Mesher_says_hello!" << std::endl;
24         return;
25     }
26
27     bool mesh_helper_t::isValid()
28     { // Only validates serial meshes for now
29         bool ans = false;
30         pGModel model = M_model( mesh);
31         pPList mesh_list = PList_new();
32         PList_append( mesh_list, mesh);
33         pParMesh par_mesh = PM_createFromMesh(
34             model,
35             M_representation(mesh),
36             mesh_list, NULL, NULL, NULL);
```



```

37
38     int status = PM_verify(par_mesh , 0 , NULL);
39     if (status == 1)
40     { ans = true; }
41
42     M_release(par_mesh);
43     PList_delete( mesh_list);
44     return ans;
45 }
46
47 void mesh_helper_t::create()
48 {
49     if (!globalSet)
50     { print_error("Global_Mesh_Parameters_not_set."); }
51     pModelItem domain = GM_domain( M_model(mesh));
52     MS_setMeshSize(m_case , domain , 2 , refine , NULL);
53
54     if( grad_rate > 0.0)
55     { MS_setGlobalSizeGradationRate(m_case , grad_rate); }
56
57     if( order == 2)
58     { MS_setMeshOrder(m_case , order); }
59
60     pSurfaceMesher surf = SurfaceMesher_new(m_case , mesh)←
61     ;
62     SurfaceMesher_execute( surf , NULL);
63     SurfaceMesher_delete( surf);
64
65     pVolumeMesher vol = VolumeMesher_new( m_case , mesh);
66     VolumeMesher_execute( vol , NULL);
67     VolumeMesher_delete(vol);
68
69     return;
70 }
71
72 void mesh_helper_t::write( std::string name)
73 {
74     std::string tmp_name = name + ".sms";
75     const char* name_c = tmp_name.c_str();
76     if( !isValid())

```

```

76     { print_warning(" Attempting to write invalid mesh.")↵
77         ;}
78     std::cout << "MESH_INFORMATION:↵"
79         << "\nVertices:↵"<< M_numVertices(mesh)
80         << "\nEdges:↵"<< M_numEdges(mesh)
81         << "\nFaces:↵"<< M_numFaces(mesh)
82         << "\nRegions:↵"<< M_numRegions(mesh) << std::endl;
83
84     int writestat = M_write(mesh, name_c, 0,0);
85     if(writestat == 0)
86     {
87         std::cout << "Mesh↵" << name << "↵written." << std↵
88             ::endl;
89     }
90     else
91     { std::cout << "Mesh↵" << name << "↵failed to be↵
92         written." << std::endl; }
93     return;
94 }
95
96 void mesh_helper_t::refine_vertex( double refine , ↵
97     pGVertex vert)
98 {
99     MS_setMeshSize(m_case, vert, 2, refine, NULL);
100     return;
101 }
102
103 void mesh_helper_t::place_point(
104     double coords[3],
105     double refine,
106     double radius,
107     bool abort_on_fail)
108 {
109     if (refine >0.0)
110     {
111         if (radius == 0.0)
112         {
113             MS_addPointRefinement( m_case, refine, coords);
114         }
115     }
116 }

```

```

112     else if (radius > 0.0)
113     {
114         MS_addSphereRefinement( m_case, refine, radius, ←
            coords);
115     }
116     else if(abort_on_fail)
117     {
118         print_error( "Refinement_radius_must_be_zero_or_←
            greater");
119     }
120     else
121     {
122         print_warning( "Refinement_radius_must_be_zero_or_←
            _greater");
123     }
124 }
125 return;
126 }
127
128 void mesh_helper_t::set_global( int order_in, double ←
    refine_in, double grad_rate_in)
129 {
130     globalSet = true;
131     order = order_in;
132     refine = refine_in;
133     grad_rate = grad_rate_in;
134     return;
135 }
136
137 void mesh_helper_t::refine_edge( double refine, pGEdge ←
    edge)
138 {
139     MS_setMeshSize(m_case, edge, 2, refine, NULL);
140     return;
141 }
142
143 void mesh_helper_t::refine_face( double refine, pGFace ←
    face)
144 {
145     MS_setMeshSize(m_case, face, 2, refine, NULL);

```

```

146     return;
147 }
148 }

```

A.10 GeoMod_printer.hpp

```

1 #ifndef GEOMOD_PRINTER_HPP
2 #define GEOMOD_PRINTER_HPP
3
4 #include <cstdlib>
5 #include <iostream>
6 #include <string.h>
7
8 namespace GMD
9 {
10     void print_error( std::string message, bool ↵
        abort_on_fail=true);
11
12     void print_warning( std::string message);
13
14     void print_coords( double x[3]);
15 }
16 #endif

```

A.11 GeoMod_printer.cpp

```

1 #include "GeoMod_printer.hpp"
2
3 #include <cstdlib>
4 #include <iostream>
5 #include <string.h>
6
7 namespace GMD
8 {
9     void print_error( std::string message, bool ↵
        abort_on_fail)
10     {
11         std::cout << "Error:_" << message << std::endl;
12         if( abort_on_fail)

```

```

13     {
14         std::abort();
15     }
16     return;
17 }
18
19 void print_warning( std::string message)
20 {
21     std::cout << "Warning:_" << message << std::endl;
22     return;
23 }
24
25 void print_coords( double x[3])
26 {
27     std::cout << "(" ;
28     for (int i=0; i<3; i++)
29     {
30         std::cout << x[i] << ",_";
31     }
32     std::cout << "\b\b)\n";
33     return;
34 }
35 }

```

A.12 GeoMod_coords.hpp

```

1 #ifndef GEOMOD_COORDS_HPP
2 #define GEOMOD_COORDS_HPP
3
4 namespace GMD
5 {
6     void sum_coords(double x[3], double y[3], double ans←
7         [3]);
8
9     void subtract_coords( double fin[3], double intil[3], ←
10         double ans[3]);
11
12     void divide( double vec[3], double denom, double ans←
13         [3]);

```

```

12 void get_mag(double vec[3], double& mag);
13
14 void get_unit_vector( double vec[3], double unit[3]);
15
16 void cross_product( double x[3], double y[3], double <←
    ans[3]);
17
18 void dot_product( double x[3], double y[3], double& ans<←
    );
19
20 void compare_coords( double x[3], double y[3], bool& <←
    areSame, double tol=0.0);
21
22 }
23 #endif

```

A.13 GeoMod_coords.cpp

```

1 #include "GeoMod_coords.hpp"
2 #include "GeoMod_printer.hpp"
3 #include <cstdlib>
4 #include <math.h>
5
6 namespace GMD
7 {
8     void sum_coords(double x[3], double y[3], double ans<←
        [3])
9     {
10         for(int i=0; i<3; i++)
11         {
12             ans[i] = x[i]+y[i];
13         }
14         return;
15     }
16
17     void subtract_coords( double fin[3], double intil[3], <←
        double ans[3])
18     {
19         for(int i=0; i<3; i++)
20         {

```

```

21     ans[i] = fin[i] - intil[i];
22 }
23 return;
24 }
25
26 void get_mag(double vec[3], double& mag)
27 {
28     double tmp = 0.0;
29     for (int i=0; i<3; i++)
30     {
31         tmp += (vec[i]*vec[i]);
32     }
33     mag = sqrt( tmp);
34 }
35
36 void compare_coords( double x[3], double y[3], bool& ←
    areSame, double tol)
37 {
38     areSame = false;
39     double mag;
40     double ans[] = {0.0, 0.0, 0.0};
41     subtract_coords( x, y, ans);
42     get_mag( ans, mag);
43     if (mag<=tol)
44     {
45         areSame = true;
46     }
47     return;
48 }
49
50 void divide( double vec[3], double denom, double ans←
    [3])
51 {
52     if(denom == 0.0)
53     {
54         print_error("Denominator_is_zero..Division_not_←
            defined.");
55     }
56     else
57     {

```

```

58     for(int i=0; i<3; i++)
59     {
60         ans[i] = vec[i]/denom;
61     }
62 }
63 return;
64 }
65
66 void get_unit_vector( double vec[3], double unit[3])
67 {
68     double mag = 0.0;
69     get_mag( vec , mag);
70     divide(vec , mag, unit);
71     return;
72 }
73
74 void cross_product( double x[3], double y[3], double &ans[3])
75 {
76     ans[0] = x[1]*y[2]-x[2]*y[1];
77     ans[1] = x[2]*y[0]-x[0]*y[2];
78     ans[2] = x[0]*y[1]-x[1]*y[0];
79     return;
80 }
81
82 void dot_product( double x[3], double y[3], double& ans)
83 {
84     for(int i=0; i<3; i++)
85     {
86         ans = x[i]*y[i];
87     }
88     return;
89 }
90
91 }

```

A.14 GeoMod_util.hpp

```

1 #ifndef GEOMOD_UTIL_HPP

```



```

2 #define GEOMOD_UTIL_HPP
3
4 // Simmetrix Headers
5 #include "SimUtil.h"
6 #include "SimModel.h"
7 #include "SimAdvModel.h"
8 #include "MeshSim.h"
9 #include "SimPartitionedMesh.h"
10
11 // Standard C++ Headers
12 #include <cstdlib>
13 #include <iostream>
14
15 pGVertex GIP_insertVertexInFace(pGIPart part, double* xyz↵
    , pGFace face);
16
17 namespace GMD
18 {
19     void sim_start( char* Sim_log_file_name, int argc, char↵
        ** argv);
20
21     void sim_end();
22     pGModel create_cube(double length);
23     pGModel create_2D_rectangle( double y_length, double ↵
        x_width);
24 }
25 #endif

```

A.15 GeoMod_util.cpp

```

1 // This File's Header
2 #include "GeoMod_util.hpp"
3 // NOTE: All needed headers belong in GeoMod_util.hpp
4
5 namespace GMD
6 {
7     void sim_start( char* Sim_log_file_name, int argc, char↵
        ** argv)
8     {
9         std::cout << "Starting_Simmetrix" << std::endl;

```

```

10     SimPartitionedMesh_start(&argc , &argv);
11     Sim_logOn( Sim_log_file_name);
12     SimUtil_start();
13     Sim_readLicenseFile(0);
14     SimModel_start();
15     MS_init();
16 }
17
18 void sim_end()
19 {
20     std::cout << "Stopping_Simmetrix" << std::endl;
21     MS_exit();
22     SimModel_stop();
23     Sim_unregisterAllKeys();
24     SimUtil_stop();
25     Sim_logOff();
26     SimPartitionedMesh_stop();
27 }
28 pGModel create_cube(double length)
29 {
30     pGModel model;
31     pGIPart part;
32     pGRegion outerRegion;
33     pGVertex vertices[8]; // array to store the returned ↵
34         model vertices
35     pGEdge edges[12]; // array to store the returned ↵
36         model edges
37     double hl = length/2;
38     model = GM_new();
39     part = GM_part(model);
40     outerRegion = GIP_outerRegion(part);
41
42     double vert_xyz[8][3] =
43     { {-hl,-hl,-hl},
44       { hl,-hl,-hl},
45       { hl, hl,-hl},
46       {-hl, hl,-hl},
47       {-hl,-hl, hl},
48       { hl,-hl, hl},
49       { hl, hl, hl},
50       {-hl, hl, hl}
51     };

```

```

48     {-hl, hl, hl} };
49
50     int i;
51     for(i=0; i<8; i++)
52         vertices[i] = GIP_insertVertexInRegion(part, ←
            vert_xyz[i], outerRegion);
53
54     pGVertex startVert, endVert;
55     double point0[3], point1[3]; // xyz locations of the ←
        two vertices
56     pCurve linearCurve;
57
58     // First, the bottom edges
59     for(i=0; i<4; i++) {
60         startVert = vertices[i];
61         endVert = vertices[(i+1)%4];
62         GV_point(startVert, point0);
63         GV_point(endVert, point1);
64         linearCurve = SCurve_createLine(point0, point1);
65         edges[i] =
66             GIP_insertEdgeInRegion(part, startVert, endVert, ←
                linearCurve, 1, outerRegion);
67     }
68
69     // Now the side edges of the box
70     for(i=0; i<4; i++) {
71         startVert = vertices[i];
72         endVert = vertices[i+4];
73         GV_point(startVert, point0);
74         GV_point(endVert, point1);
75         linearCurve = SCurve_createLine(point0, point1);
76         edges[i+4] =
77             GIP_insertEdgeInRegion(part, startVert, endVert, ←
                linearCurve, 1, outerRegion);
78     }
79
80     // Finally the top edges
81     for(i=0; i<4; i++) {
82         startVert = vertices[i+4];
83         endVert = vertices[(i+1)%4+4];

```

```

84     GV_point(startVert , point0);
85     GV_point(endVert , point1);
86     linearCurve = SCurve_createLine(point0 , point1);
87     edges[i+8] =
88         GIP_insertEdgeInRegion(part , startVert , endVert , ←
            linearCurve , 1, outerRegion);
89 }
90
91 double corner[3] , xPt[3] , yPt[3]; // the points ←
    defining the surface of the face
92 pGEdge faceEdges[4]; // the array of ←
    edges connected to the face
93 int faceDirs[4]; // the direction ←
    of the edge with respect to the face
94 int loopDef[1] = {0};
95 pSurface planarSurface;
96
97 // First the bottom face
98 // Define the surface – we want the normal to point ←
    out of the box
99 for(i=0; i<3; i++)
100 {
101     corner[i] = vert_xyz[1][i];
102     xPt[i] = vert_xyz[0][i];
103     yPt[i] = vert_xyz[2][i];
104 }
105 planarSurface = SSurface_createPlane(corner , xPt , yPt);
106 // Define and insert the face into the outer "void" ←
    region
107 for(i=0; i<4; i++) {
108     faceDirs[i] = 0;
109     faceEdges[i] = edges[3-i]; // edge order 3→2→1→0
110 }
111 GIP_insertFaceInRegion(part , 4 , faceEdges , faceDirs , 1 , ←
    loopDef , planarSurface , 1 , outerRegion);
112
113 // Now the side faces of the box – each side face has ←
    the edges defined in the same way
114 // for the first side face, the edge order is ←
    0→5→8→4

```

```

115  for(i=0; i<4; i++) {
116      //Define surface such that normals all point out of
117      the box
118      for(int j=0; j<3; j++) {
119          corner[j] = vert_xyz[i][j];          // the corner is
120          the lower left vertex location
121          xPt[j] = vert_xyz[(i+1)%4][j];        // the xPt the
122          lower right vertex location
123          yPt[j] = vert_xyz[i+4][j];            // the yPt is
124          the upper left vertex location
125      }
126      planarSurface = SSurface_createPlane(corner ,xPt ,yPt
127      );
128
129      faceEdges[0] = edges[i];
130      faceDirs[0] = 1;
131      faceEdges[1] = edges[(i+1)%4+4];
132      faceDirs[1] = 1;
133      faceEdges[2] = edges[i+8];
134      faceDirs[2] = 0;
135      faceEdges[3] = edges[i+4];
136      faceDirs[3] = 0;
137
138      GIP_insertFaceInRegion(part ,4 ,faceEdges ,faceDirs ,1 ,
139      loopDef ,planarSurface ,1 ,outerRegion);
140  }
141
142  // Finally the top face of the box
143  // Define the surface – we want the normal to point
144  out of the box
145  for(i=0; i<3; i++) {
146      corner[i] = vert_xyz[4][i];
147      xPt[i] = vert_xyz[5][i];
148      yPt[i] = vert_xyz[7][i];
149  }
150  planarSurface = SSurface_createPlane(corner ,xPt ,yPt);
151  // Define and insert the face
152  for(i=0; i<4; i++) {
153      faceDirs[i] = 1;

```

```

147     faceEdges[i] = edges[i+8]; // edge order ←
148         8->9->10->11
149     }
150     // when this face is inserted, a new model region ←
151     will automatically be created
152     GIP_insertFaceInRegion(part,4,faceEdges,faceDirs,1,←
153     loopDef,planarSurface,1,outerRegion);
154
155     return model;
156 }
157
158 pGModel create_2D_rectangle( double y_length, double ←
159     x_width)
160 {
161     // Create an empty modeling space to work with
162     pGModel model = GM_new();
163     pGIPart part = GM_part(model);
164     pGRegion outRegion = GIP_outerRegion(part);
165
166     // Create model vertices
167     pGVertex verts[4];
168     double vert_xyz[4][3] ={{ 0.0, 0.0, 0.0},
169     { y_length, 0.0, 0.0},
170     { y_length, x_width, 0.0},
171     { 0.0, x_width, 0.0}};
172     for(int i=0; i<4; i++)
173     {
174         verts[i] = GIP_insertVertexInRegion( part, vert_xyz←
175         [i], outRegion);
176     }
177
178     // Create model edges
179     pCurve line;
180     pGEdge edges[4];
181     for(int i=0; i<4; i++)
182     {
183         pGVertex start = verts[i];
184         pGVertex end = verts[(i+1)%4];
185         line = SCurve_createLine(vert_xyz[i], vert_xyz[(i←
186         +1)%4]);

```

```

181     edges[i] = GIP_insertEdgeInRegion( part , start , end↵
182     , line , 1, outRegion);
183 }
184 // Create Face
185 pSurface plane;
186 int face_dirs[4] = {1,1,1,1};
187 int loopDef[1] = {0};
188 plane = SSurface_createPlane(vert_xyz[0] , vert_xyz↵
189     [1] , vert_xyz[2]);
190 GIP_insertFaceInRegion(part , 4, edges , face_dirs , 1, ↵
191     loopDef , plane , 1, outRegion);
192 return model;
193 }
194 }

```

A.16 GeoMod_SIM.hpp

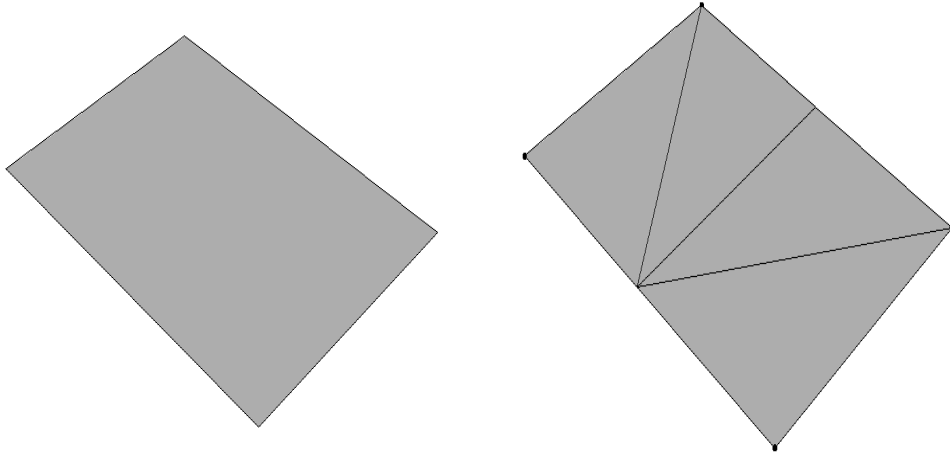
```

1 #ifndef GEOMOD_SIM_HPP
2 #define GEOMOD_SIM_HPP
3
4 // Simmetrix Headers
5 #include "SimUtil.h"
6 #include "SimModel.h"
7 #include "SimAdvModel.h"
8 #include "MeshSim.h"
9 #include "SimPartitionedMesh.h"
10
11 #endif

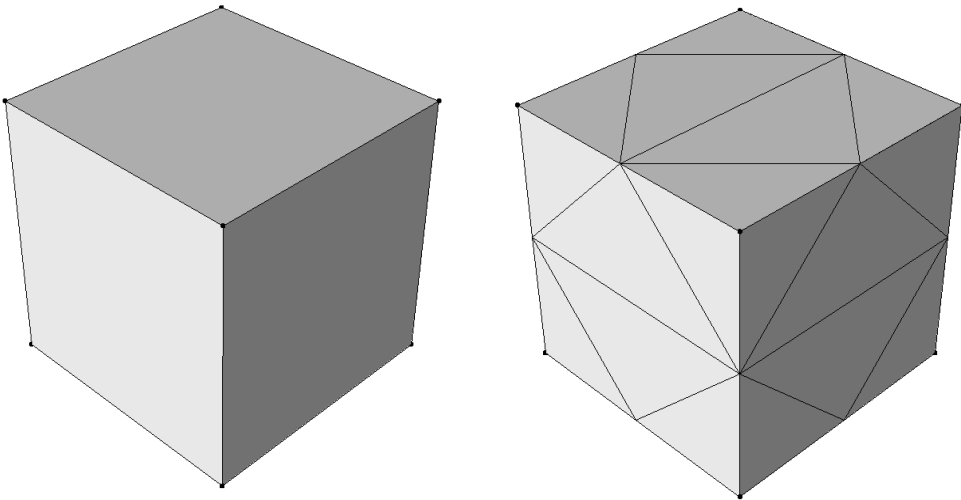
```

B Supplementary Images

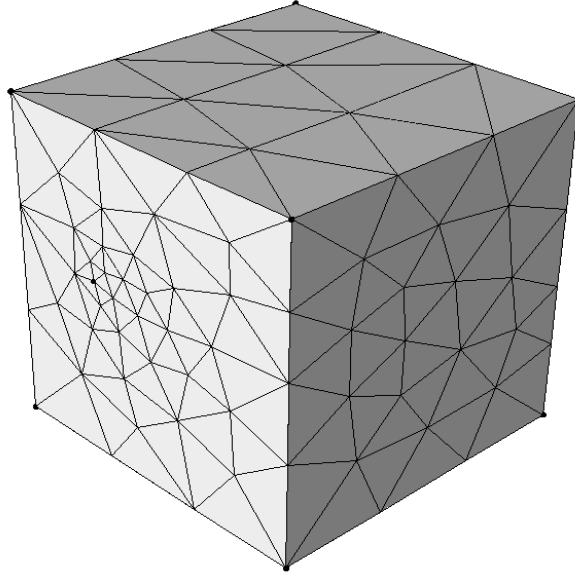
B.1 Model and Mesh of Test 1



B.2 Model and Mesh of Test 2



B.3 Mesh of Test 11



B.4 Mesh of Test 12

