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 with in 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 dislocation 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 a 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 then 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	$N+1$	$\dots$	$N(M-1)+1$
$u \downarrow$	2	$N+2$	$\dots$	$N(M-1)+2$
	$\vdots$	$\vdots$	$\dots$	$\vdots$
	$N-1$	$2N-1$	$\dots$	$MN-1$

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. Theses 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.

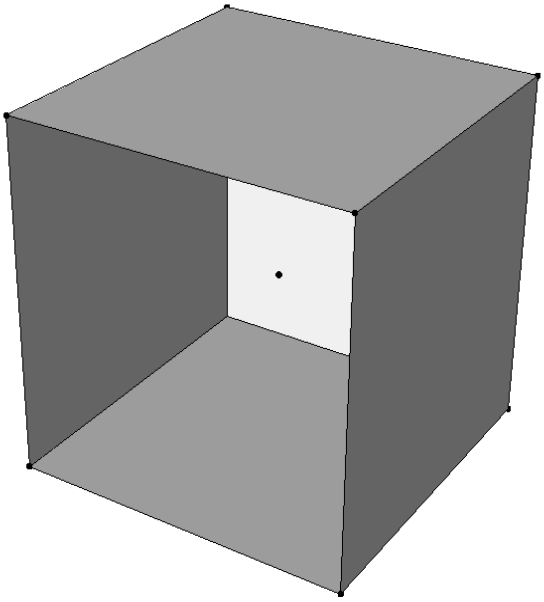
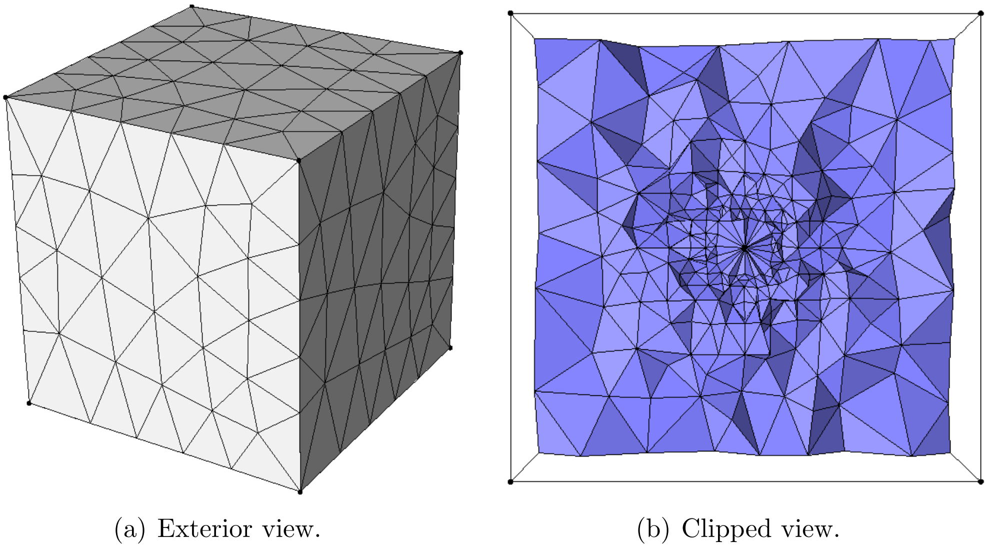


Figure 1: Cube with vertex placed at center. Front face hidden to to show interior.



(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.

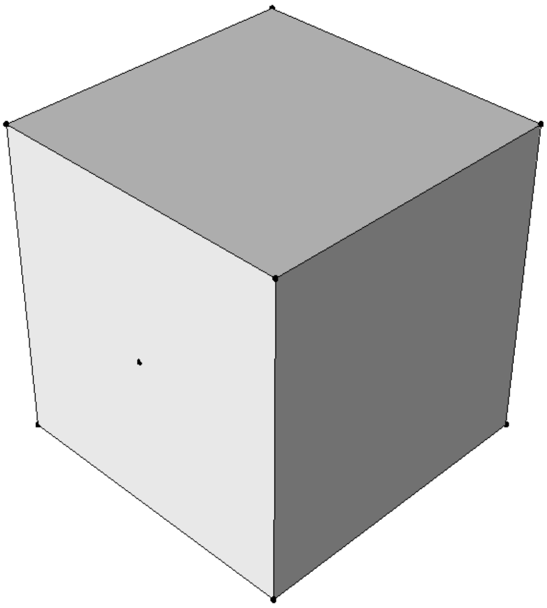


Figure 3: Cube with vertex placed at center of front face.

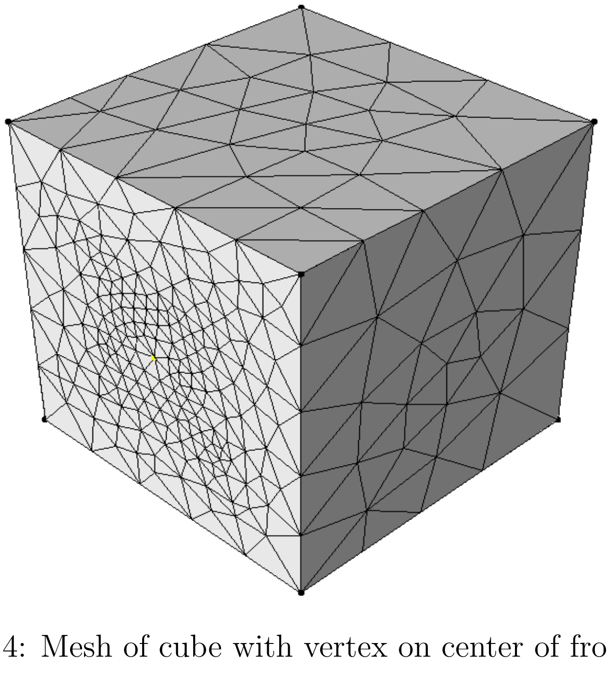


Figure 4: Mesh of cube with vertex on center of front face.

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.

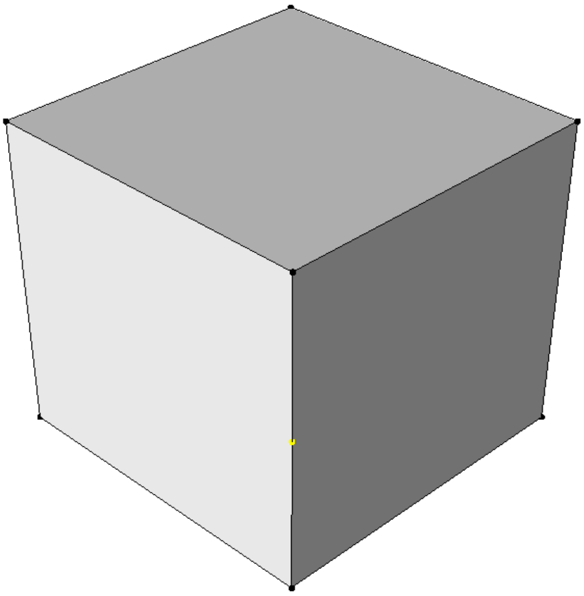


Figure 5: Cube with vertex placed at center of edge.

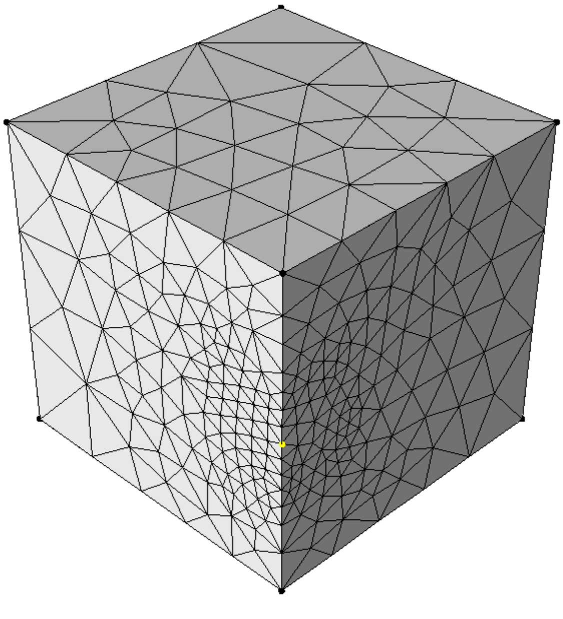


Figure 6: Mesh of cube with vertex on center of edge.

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.

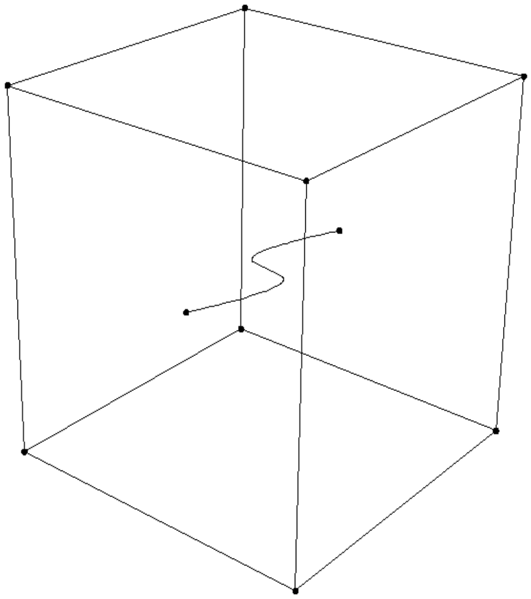


Figure 7: Cube with edge placed near center of region.

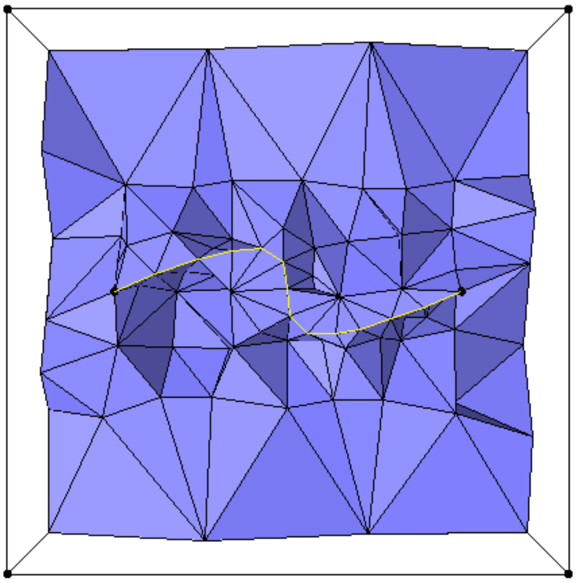


Figure 8: Clipped view of mesh for cube with edge near center of region. The edge is highlighted in yellow.

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.

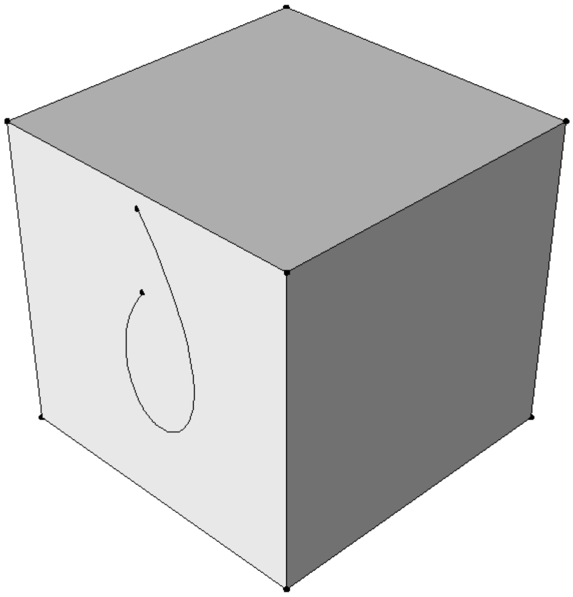


Figure 9: Cube with edge placed on model front face.

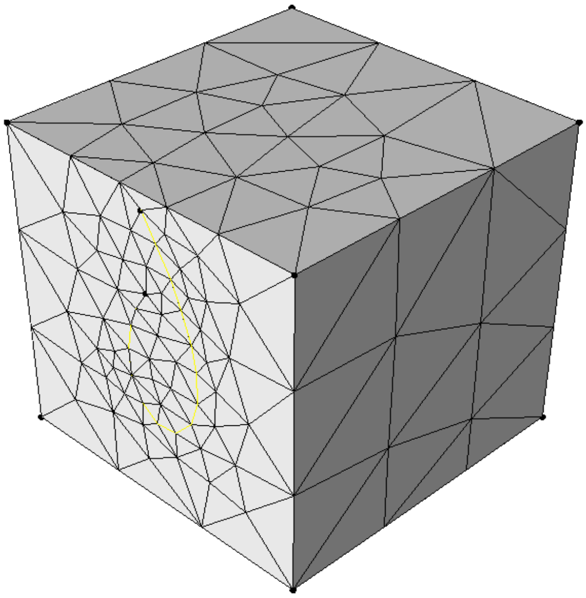
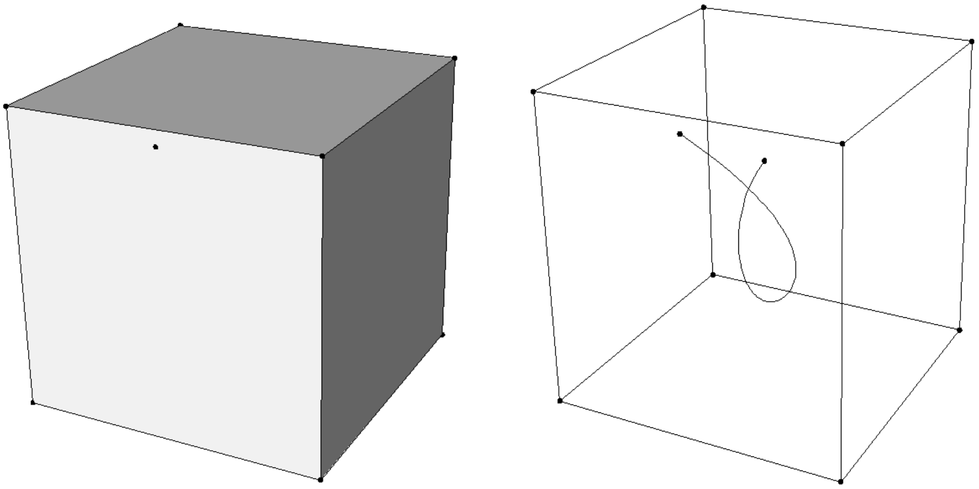


Figure 10: Clipped view of mesh for cube with edge on face. The edge is highlighted in yellow.

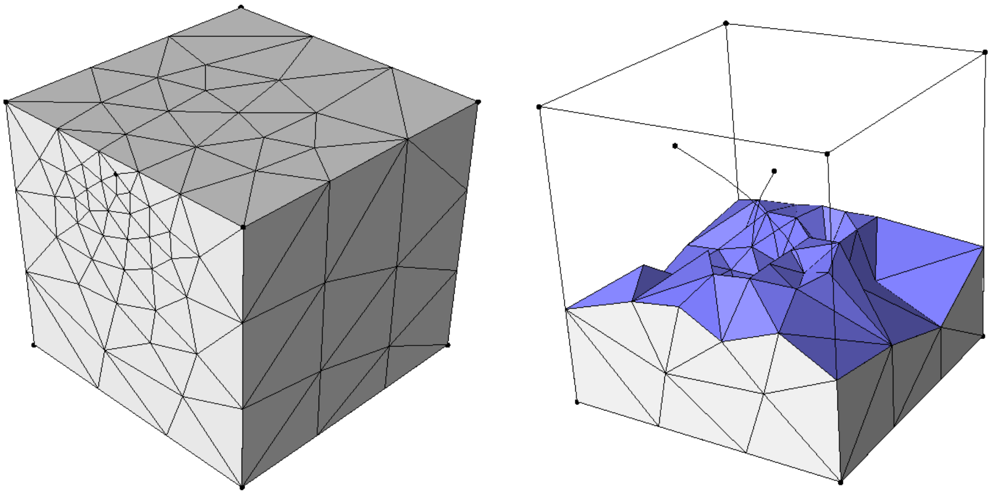
The third test placed an edge that started on a model face and terminated in the interior of the model region. The edge was created as a rational basis spline with five control points. The global mesh refinement was set to 0.9 and the local set to 0.1. Two images of the created model are shown in Figure 11. A external and clipped view of the mesh are shown in Figure 12.



(a) Solid view.

(b) Wire frame view.

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

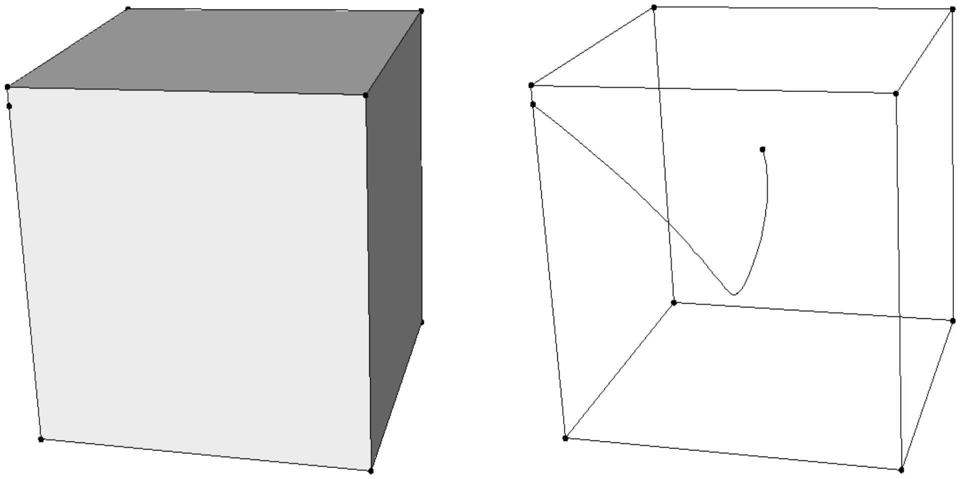


(a) External view.

(b) Clipped view.

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

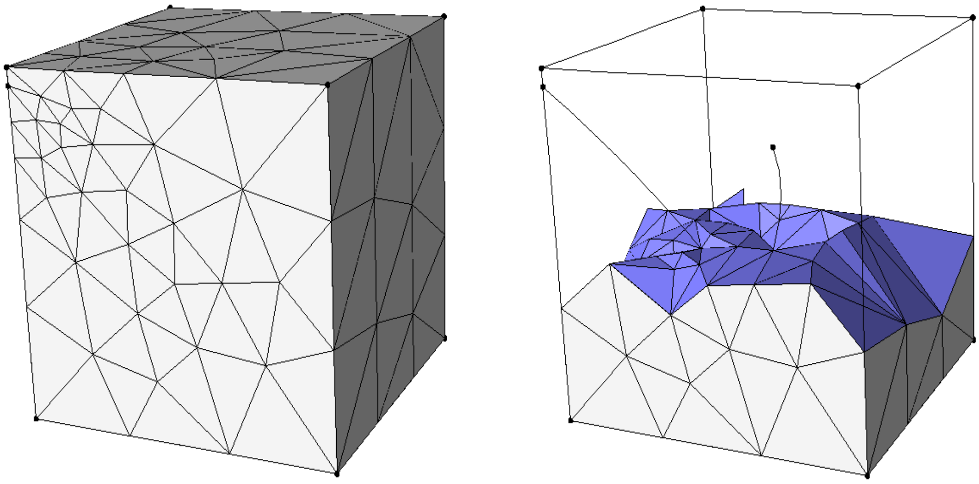
The fourth test placed an edge that started on a model edge and terminated with in the model region. The edge was created as a rational basis spline with five control points. The global mesh refinement was set to 0.9 and the local set to 0.1. Two images of the created model are shown in Figure 13. A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



(a) Solid view.

(b) Wire frame view.

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



(a) External view.

(b) Clipped view.

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 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.

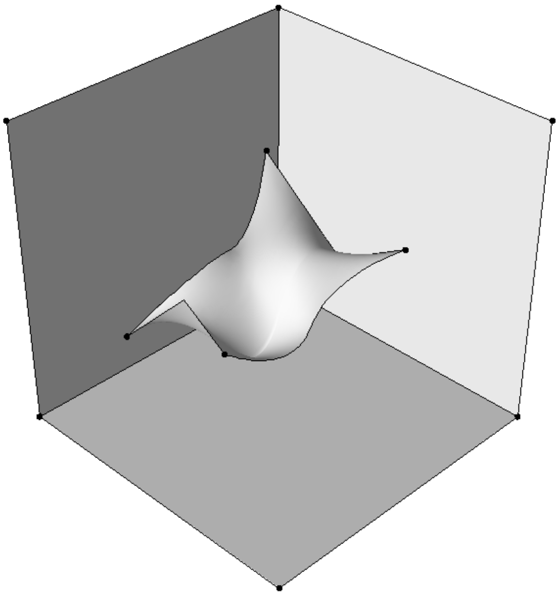
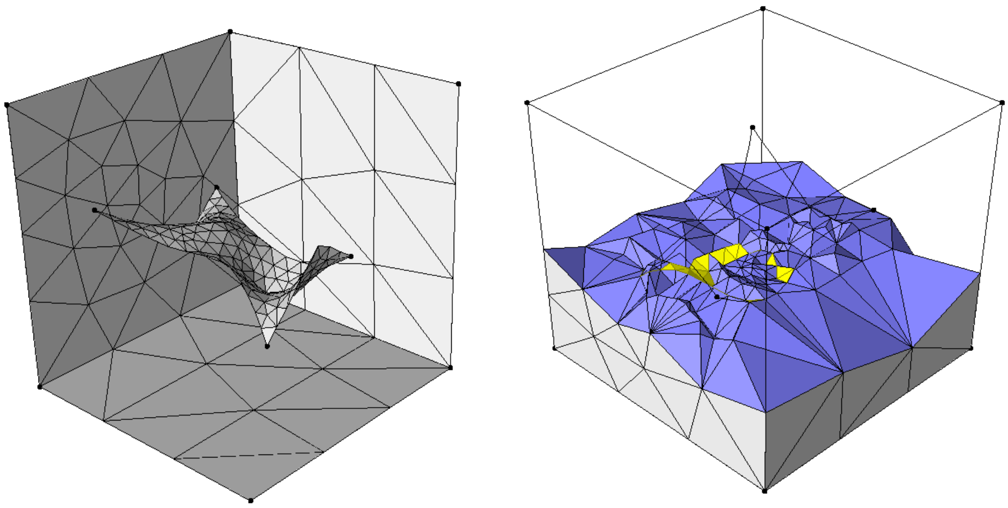


Figure 15: Cube with face created within the model region. Forward three faces were made transparent to show in the internal face.

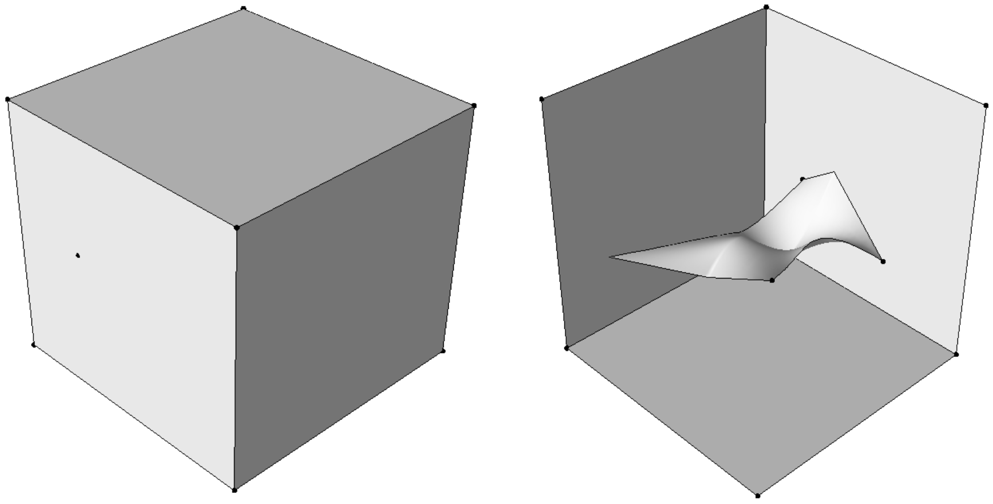


(a) Face view.

(b) Clipped view.

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.

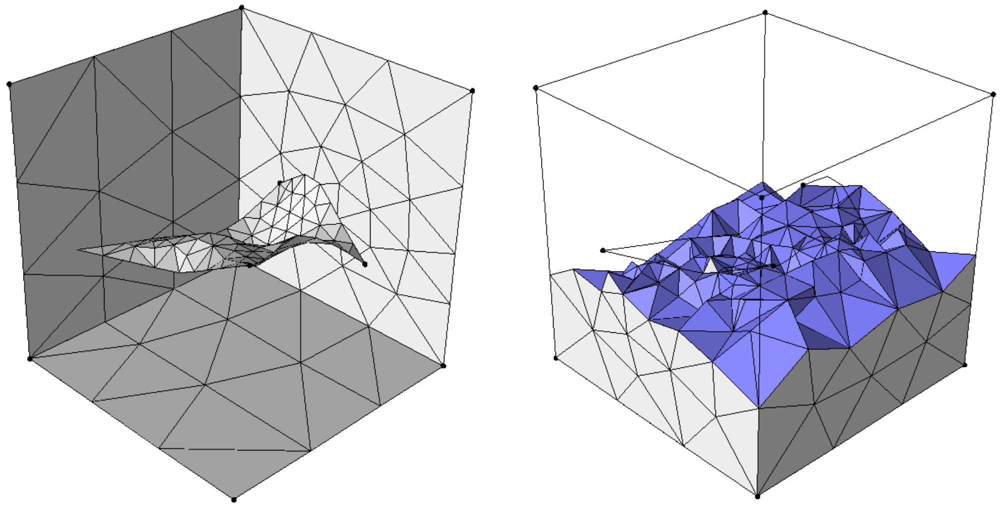
The second face placement test created a model face that was completely interior to a model region except for one corner vertex. The face, and its bounding edges, were created as as a rational basis surface and splines. In total, 12 control points were defined; all but one control point was inside the 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.



(a) Solid view.

(b) Face view.

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

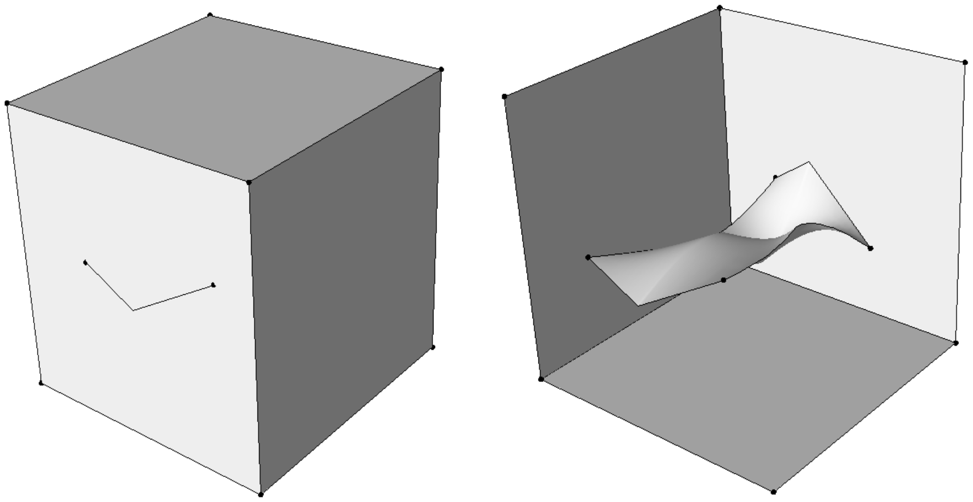


(a) Face view.

(b) Clipped view.

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

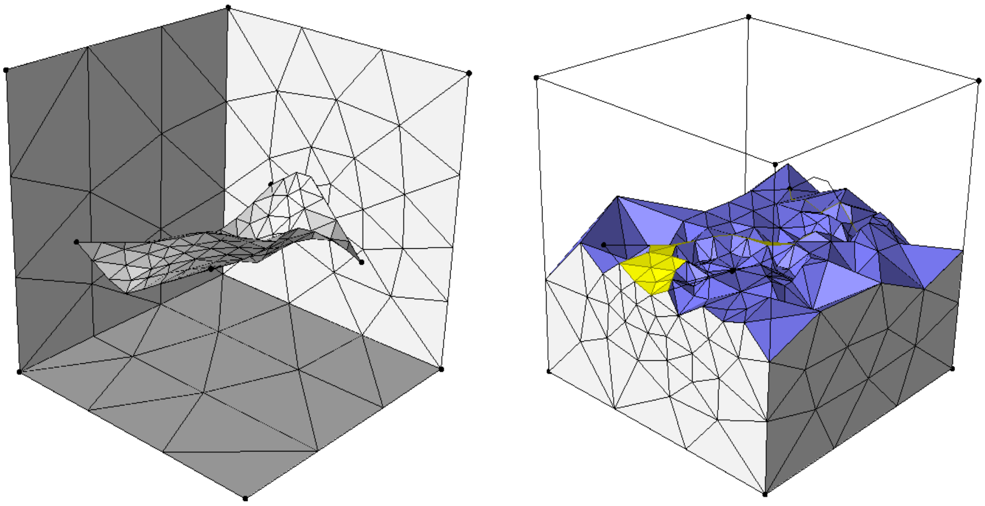
The third face placement test created a model face that was completely interior to a model region except for one edge. The face, and its bounding edges, were created as as a rational basis surface and splines. In total, 12 control points were defined; the control points created a 4 × 3 grid. The global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



(a) Solid view.

(b) Face view.

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



(a) Face view.

(b) Clipped view.

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 theses 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

```
// GeoMod Header
#include ”GeoMod.hpp”

#include ”GeoMod_Tests.hpp”

int main( int argc, char** argv)
{
  char sim_log[] = ”Sim_log.log”;
  std::cout << ”START” << std::endl;
  GMD::sim_start( sim_log, argc, argv);

  test0();
  test1();
  test2();
  test3();
  test4();
  test5();
  test6();
  test7();
  test8();
  test9();
  test10();
  test11();
  test12();

  GMD::sim_end();
  std::cout << ”END” << std::endl;
  return 0;
}
```

A.2 GeoMod_Tests.hpp

```
#ifndef GEOMOD_TESTS_HPP
#define GEOMOD_TESTS_HPP

#include ”GeoMod.hpp”
```

```


void test0();
void test1();
void test2();
void test3();
void test4();
void test5();
void test6();
void test7();
void test8();
void test9();
void test10();
void test11();
void test12();

#endif
```

A.3 GeoMod_Tests.cpp

```
#include "GeoMod_Tests.hpp"
#include <string>

using std::cout;

/* test0():
 *      - Create a 3D model
 *      - Create a gmd object
 *      - Test printing features
 *      - Implicitly destory the gmd object
 */
void test0()
{
  pGModel cube = GMD::create_cube( 10.0);
  GMD::gmd_t gmd( cube);
  gmd.test_printers();

  cout << "\n\nPassed test0\n\n" ;
  return;
}
```

```

/* test1():
 *      - Create 2D model
 *      - Create gmd_t instance with model
 *      - Write model
 *      - Create mesh from model
 *      - Write mesh
 */
void test1()
{
  pGModel rectangle = GMD::create_2D_rectangle( 5.0, 7.0);
  GMD::gmd_t gmd( rectangle);

  std::string name = "test1_rectangle";
  gmd.set_name( name);
  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test1\n\n";
  return;
}

/* test2():
 *      - Create a 3D model
 *      - Create gmd_t instance with model
 *      - Write model
 *      - Create mesh from model
 *      - Write mesh
 */
void test2()
{
  pGModel cube = GMD::create_cube( 10.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test2_cube";
  gmd.set_name( name);
  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed␣test2\n\n";
  return;
}

/* test3():
 *     - Create a 3D model
 *     - Place a point with defined mesh refinement in center
 *     - Write model
 *     - Create a mesh from model
 *     - Write mesh
 */
void test3()
{
  pGModel cube = GMD::create_cube( 5.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test3_cube";
  gmd.set_name( name);

  gmd.set_global_mesh_params( 1, 0.5, 0.3);
  double coords[3] = {0.0, 0.0, 0.0};
  double refine = 0.01;
  double radius = 0.01;
  pGVertex vert;
  gmd.place_point( coords, refine, radius, vert);

  gmd.write_model();
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed␣test3\n\n";
  return;
}

/* test4():
 *     - Create a 3D model
 *      - Place a point with defined mesh refinement on surface
 *      - Write model
 *      - Create mesh from model
 *      - Write mesh
 */
void test4()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test4_cube";
  gmd.set_name( name);

  double coords[3] = {1.0, 0.0, 0.0};
  double refine = 0.1;
  double radius = 0.5;
  pGVertex vert;
  gmd.place_point( coords, refine, radius, vert);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test4\n\n";
  return;
}

/* test5():
 *      - Create a 3D model
 *      - Place a point with defined mesh refinement on an edge
 *      - Write model
 *      - Create mesh from model
 *      - Write mesh
 */
void test5()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test5_cube";
  gmd.set_name( name);

  double coords[3] = {1.0, 1.0, 0.0};
  double refine = 0.1;
  double radius = 0.5;
  pGVertex vert;
  gmd.place_point( coords, refine, radius, vert);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test5\n\n";
  return;
}

/* test6():
 *      - Create a 3D model
 *      - Place a fully interior edge define by three points
 *      - Write model
 *      - Do not assign name so auto-nameing feature is checked
 *      - Create mesh
 *      - Write mesh
 */
void test6()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test6_cube";
  gmd.set_name( name);

  int order = 4;

  double p1[3] = {0.7, 0.0, 0.0};
  double p2[3] = {0.0, 0.3, 0.0};
  double p3[3] = {0.0, 0.0, 0.0};
  double p4[3] = {0.0, -0.3, 0.0};
  double p5[3] = {-0.7, 0.0, 0.0};
  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);

  std::vector<double> knots;
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.5);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGEdge edge;
  double refine = 0.1;
  gmd.place_edge( order, points, knots, weights, refine, edge);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test6\n\n";
  return;
}

/* test7():
 *      - Create a 3D model
 *      - Create one spline based on-face edge defined by five points
 *      - Write model
 *      - Create mesh
 *      - Write mesh
 */

void test7()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test7_cube";
  gmd.set_name( name);

  int order = 4;

  double p1[3] = {1.0, 0.0, 0.9};
  double p2[3] = {1.0, 0.6, 0.0};
  double p3[3] = {1.0, 0.0, -0.9};
  double p4[3] = {1.0, -0.3, 0.0};
  double p5[3] = {1.0, 0.0, 0.4};
  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);

  std::vector<double> knots;
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.5);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGEdge edge;
  double refine = 0.1;
  gmd.place_edge( order, points, knots, weights, refine, edge);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test7\n\n";
  return;
}

/*  test8():
 *      - Create a 3D model
 *      - Create an edge from a surface point to an interior point
 *      - Write model
 *      - Create mesh
 *      - Write mesh
 */
void test8()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test8_cube";
  gmd.set_name( name);

  int order = 4;

  double p1[3] = {1.0, 0.0, 0.9};
  double p2[3] = {0.1, 0.6, 0.0};
  double p3[3] = {0.3, 0.0, -0.9};
  double p4[3] = {0.0, -0.3, 0.0};
  double p5[3] = {0.0, 0.0, 0.4};
  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);

  std::vector<double> knots;
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.5);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGEdge edge;
  double refine = 0.1;
  gmd.place_edge( order, points, knots, weights, refine, edge);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test8\n\n";
  return;
}

/* test9();
 *     - Create a 3D model
 *     - Create an edge that starts at a pre existing edge and then terminates
 *         in the region
 *     - Write Model
 *     - Create Mesh
 *      - Write Mesh
 */
void test9()
{
  pGModel cube = GMD::create_cube( 2.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test9_cube";
  gmd.set_name( name);

  int order = 4;

  double p1[3] = {1.0, 1.0, 0.9};
  double p2[3] = {0.1, 0.6, 0.0};
  double p3[3] = {0.3, 0.0, -0.9};
  double p4[3] = {0.0, -0.3, 0.0};
  double p5[3] = {0.0, 0.0, 0.4};
  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);

  std::vector<double> knots;
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.0);
  knots.push_back(0.5);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);
  knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGEdge edge;
  double refine = 0.1;
  gmd.place_edge( order, points, knots, weights, refine, edge);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test9\n\n";
  return;
}

/* test10();
 *      - Create a 3D model
 *      - Create a face completely internal to the region
 *      - Write Model
 *      - Create Mesh
 *      - Write Mesh
 */
void test10()
{
  pGModel cube = GMD::create_cube( 20.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test10_cube";
  gmd.set_name( name);

  int u_order = 3;
  int v_order = 2;
  int u_num = 4;
  int v_num = 3;
  int periodicity = 0;

  // first 'row'      x      y     z
  double p1[3]  = {-6.0, -7.0, 3.0};
  double p2[3]  = {-2.0, -4.0, 0.0};
  double p3[3]  = { 3.0, -3.0, 1.0};
  double p4[3]  = { 7.0, -5.0, -3.0};
  // second 'row'
  double p5[3]  = {-5.0, 0.0, -1.0};
  double p6[3]  = {-2.0, 0.0, 3.0};
  double p7[3]  = { 3.0, 1.0, -7.0};
  double p8[3]  = { 7.0, 0.0, 2.0};
  // third 'row'
  double p9[3]  = {-6.0, 4.0, 1.0};
  double p10[3] = {-2.0, 2.0, 0.0};
  double p11[3] = { 3.0, 4.0, -3.0};
  double p12[3] = { 7.0, 3.0, 0.0};

  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);
  points.push_back(p6);
  points.push_back(p7);
  points.push_back(p8);
  points.push_back(p9);
  points.push_back(p10);
  points.push_back(p11);
  points.push_back(p12);

  std::vector<double> u_knots;
  u_knots.push_back(0.0);
  u_knots.push_back(0.0);
  u_knots.push_back(0.0);
  u_knots.push_back(0.5);
  u_knots.push_back(1.0);
  u_knots.push_back(1.0);
  u_knots.push_back(1.0);

  std::vector<double> v_knots;
  v_knots.push_back(0.0);
  v_knots.push_back(0.0);
  v_knots.push_back(0.5);
  v_knots.push_back(1.0);
  v_knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGFace face;
  double refine = 0.1;
  gmd.place_surface_by_spline(
      u_order, v_order, u_num, v_num, periodicity,
      points, u_knots, v_knots, weights,
      refine, face);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test10\n\n";
  return;
}

/* test11();
 *      - Create a 3D model
 *      - Create a face completely internal to the region expect for one point
 *          which is on a pre-existing surface
 *      - Write Model
 *      - Create Mesh
 *      - Write Mesh
 */
void test11()
{
  pGModel cube = GMD::create_cube( 20.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test11_cube";
  gmd.set_name( name);

  int u_order = 3;
  int v_order = 2;
  int u_num = 4;
  int v_num = 3;
  int periodicity = 0;

  // first 'row'      x       y      z
```

```
    double p1[3]  = {-6.0, -7.0, 3.0};
    double p2[3]  = {-2.0, -4.0, 0.0};
    double p3[3]  = { 3.0, -3.0, 1.0};
    double p4[3]  = { 7.0, -5.0, -3.0};
    // second 'row'
    double p5[3]  = {-5.0, 0.0, -1.0};
    double p6[3]  = {-2.0, 0.0, 3.0};
    double p7[3]  = { 3.0, 1.0, -7.0};
    double p8[3]  = { 7.0, 0.0, 2.0};
    // third 'row'
    double p9[3]  = {-10.0, 4.0, 1.0};
    double p10[3] = {-2.0, 2.0, 0.0};
    double p11[3] = { 3.0, 4.0, -3.0};
    double p12[3] = { 7.0, 3.0, 0.0};

    std::vector<double*> points;
    points.push_back(p1);
    points.push_back(p2);
    points.push_back(p3);
    points.push_back(p4);
    points.push_back(p5);
    points.push_back(p6);
    points.push_back(p7);
    points.push_back(p8);
    points.push_back(p9);
    points.push_back(p10);
    points.push_back(p11);
    points.push_back(p12);

    std::vector<double> u_knots;
    u_knots.push_back(0.0);
    u_knots.push_back(0.0);
    u_knots.push_back(0.0);
    u_knots.push_back(0.5);
    u_knots.push_back(1.0);
    u_knots.push_back(1.0);
    u_knots.push_back(1.0);

    std::vector<double> v_knots;
    v_knots.push_back(0.0);
```

```
  v_knots.push_back(0.0);
  v_knots.push_back(0.5);
  v_knots.push_back(1.0);
  v_knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGFace face;
  double refine = 0.1;
  gmd.place_surface_by_spline(
      u_order, v_order, u_num, v_num, periodicity,
      points, u_knots, v_knots, weights,
      refine, face);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << "\n\nPassed test11\n\n";
  return;
}

/* test12();
 *      - Create a 3D model
 *      - Create a face completely internal to the region expect for one edge
 *          which is on a pre-existing surface
 *      - Write Model
 *      - Create Mesh
 *      - Write Mesh
 */
void test12()
{
  pGModel cube = GMD::create_cube( 20.0);
  GMD::gmd_t gmd( cube);
  std::string name = "test12_cube";
  gmd.set_name( name);

  int u_order = 3;
  int v_order = 2;
  int u_num = 4;
  int v_num = 3;
  int periodicity = 0;

  // first 'row'      x      y     z
  double p1[3]  = {-10.0, -7.0, 3.0};
  double p2[3]  = {-2.0, -4.0, 0.0};
  double p3[3]  = { 3.0, -3.0, 1.0};
  double p4[3]  = { 7.0, -5.0, -3.0};
  // second 'row'
  double p5[3]  = {-10.0, 0.0, -1.0};
  double p6[3]  = {-2.0, 0.0, 3.0};
  double p7[3]  = { 3.0, 1.0, -7.0};
  double p8[3]  = { 7.0, 0.0, 2.0};
  // third 'row'
  double p9[3]  = {-10.0, 4.0, 1.0};
  double p10[3] = {-5.0, 2.0, 0.0};
  double p11[3] = { 3.0, 4.0, -3.0};
  double p12[3] = { 7.0, 3.0, 0.0};

  std::vector<double*> points;
  points.push_back(p1);
  points.push_back(p2);
  points.push_back(p3);
  points.push_back(p4);
  points.push_back(p5);
  points.push_back(p6);
  points.push_back(p7);
  points.push_back(p8);
  points.push_back(p9);
  points.push_back(p10);
  points.push_back(p11);
  points.push_back(p12);

  std::vector<double> u_knots;
  u_knots.push_back(0.0);
  u_knots.push_back(0.0);
  u_knots.push_back(0.0);
  u_knots.push_back(0.5);
  u_knots.push_back(1.0);
  u_knots.push_back(1.0);
  u_knots.push_back(1.0);

  std::vector<double> v_knots;
  v_knots.push_back(0.0);
  v_knots.push_back(0.0);
  v_knots.push_back(0.5);
  v_knots.push_back(1.0);
  v_knots.push_back(1.0);

  std::vector<double> weights;
  weights.push_back(0.0);

  pGFace face;
  double refine = 0.1;
  gmd.place_surface_by_spline(
      u_order, v_order, u_num, v_num, periodicity,
      points, u_knots, v_knots, weights,
      refine, face);

  gmd.write_model();
  gmd.set_global_mesh_params( 1, 0.9, 0.0);
  gmd.create_mesh();
  gmd.write_mesh();

  cout << ”\n\nPassed␣test12\n\n”;
  return;
}
```

A.4 GeoMod_gmd.hpp

```
#ifndef GEOMOD_GMD_T_HPP
#define GEOMOD_GMD_T_HPP

#include <string>
#include <vector>
#include ”GeoMod_printer.hpp”
#include ”GeoMod_model_helper.hpp”
```

```
#include ”GeoMod_mesh_helper.hpp”

namespace GMD
{

  class gmd_t
  {
    public:
      // Util methods
      gmd_t( pGModel in_model);
      ~gmd_t();
      void set_abort_on_fail( bool abort_on_fail);
      void test_printers();
      void verify_mesh();

      // Writing methods
      void set_name( std::string file_name);
      void write_model();
      void write_mesh();

      // Methods to modify the geometry
      void place_point(
          double coords[3],
          double refine,
          double radius,
          pGVertex& vert);
      void place_edge(
          int order,
          std::vector<double*> points,
          std::vector<double> knots,
          std::vector<double> weights,
          double refine,
          pGEdge& edge);
      void place_surface_by_spline(
          int u_order,
          int v_order,
          int u_num,
          int v_num,
          int periodicity,
            // 0=none, 1=u periodic, 2=v periodic, 3=u&&v periodic
          std::vector<double*> points,
          std::vector<double> u_knots,
          std::vector<double> v_knots,
          std::vector<double> weights,
          double refine,
          pGFace& face);

      // Methods for only meshing
      void set_global_mesh_params(
          int order_in,
          double refine_in,
          double grad_rate_in);
      void create_mesh();

    private:
      bool panicStatus;
      std::string name;
      model_helper_t* modeler;
      mesh_helper_t* mesher;
      void check_spline_params(
          int order,
          std::vector<double*> points,
          std::vector<double> knots,
          std::vector<double> weights);
      void check_surface_params(
          int u_order,
          int v_order,
          int u_num,
          int v_num,
          int periodicity,
          std::vector<double*> points,
          std::vector<double> u_knots,
          std::vector<double> v_knots,
          std::vector<double> weights);
  };

}

```

```
#endif
```

A.5 GeoMod_gmd.cpp

```
#include "GeoMod_gmd_t.hpp"

namespace GMD
{
  gmd_t::gmd_t( pGModel in_model)
  {
    panicStatus = true;
    modeler = new model_helper_t( in_model);
    mesher = new mesh_helper_t ( in_model);
    return;
  }

  gmd_t::~gmd_t()
  {
    delete modeler;
    delete mesher;
    return;
  }

  void gmd_t::set_abort_on_fail( bool abort_on_fail)
  {
    panicStatus = abort_on_fail;
    return;
  }

  void gmd_t::test_printers()
  {
    modeler->model_print();
    mesher->mesh_print();
    return;
  }

  void gmd_t::place_point( double coords[3], double refine, double radius, pGVertex& vert)
  {
```

```
  bool updateMesh = modeler->place_point( coords, vert, panicStatus);
  if( updateMesh)
  {
    mesher->place_point( coords, refine, radius, panicStatus);
  }

  return;
}

void gmd_t::set_name( std::string file_name)
{
  name = file_name;
  return;
}

void gmd_t::check_spline_params(
    int order,
    std::vector<double*> points,
    std::vector<double> knots,
    std::vector<double> weights)
{
  if( order <=1)
  { print_error("Spline order too low. Must be polnomial order + 1.");}
  if( (int)points.size() < order)
  { print_error("Spline order too high for number of given control points.");}
  if( ((int)points.size()+order) != (int)knots.size())
  { print_error("Mismatch between knot vector size and sum of order with control point size.");}

  for( int i=0; i<(int)knots.size(); i++)
  {
    double tmp1 = knots[i];
    if( i<((int)knots.size() -1) )
    {
      double tmp2 = knots[i+1];
      if (tmp2 < tmp1)
      { print_error("Knots must be in accending order");}
    }
    if( i<order && tmp1 != 0.0)
    { print_error("First order number of knots must be zero.");}
    if( i>((int)knots.size()-order) && tmp1 != 1.0)
    { print_error("Last order number of knots must be one.");}

    if( tmp1 > 1.0 || tmp1 < 0.0)
    {
      print_error("Knots must satisfy 0.0<=k[i]<=1.0 .");
    }
  }

  if( (int) weights.size() != 1 && (int) weights.size() != (int) points.size())
  { print_error("Mismatch between weight vector size and number of points.");}
  return;
}

void gmd_t::place_edge(
    int order,
    std::vector<double*> points,
    std::vector<double> knots,
    std::vector<double> weights,
    double refine,
    pGEdge& edge)
{
  check_spline_params( order, points, knots, weights);
  modeler->place_edge( order, points, knots, weights, edge);
  if( refine > 0.0)
  { mesher->refine_edge( refine, edge);}
  return;
}

```

```
  void gmd_t::write_model()
  {
    modeler->write( name);
    return;
  }

  void gmd_t::create_mesh()
  {
    if(modeler->isValid())
    {
      mesher->create();
    }
    else if( (modeler->isValid()) && !panicStatus)
    {
      print_warning("Attempting␣to␣mesh␣invalid␣model.");
    }
    else
    {
      print_error("Invalid␣model.␣No␣mesh␣created");
    }
    return;
  }

  void gmd_t::verify_mesh()
  {
    if(!mesher->isValid())
    {
      if(panicStatus)
      { print_error("Mesh␣Not␣Valid.");}
      else if( !panicStatus)
      { print_warning("Mesh␣Not␣Valid.");}
    }
    return;
  }

  void gmd_t::write_mesh()
  {
    if(modeler->isWritten())
    { mesher->write( name); }
    else
    { print_error("Model must be written before mesh."); }
    return;
  }

  void gmd_t::set_global_mesh_params( int order_in, double refine_in, double grad_rate_in)
  {
    mesher->set_global( order_in, refine_in, grad_rate_in);
    return;
  }

  void gmd_t::place_surface_by_spline(
      int u_order,
      int v_order,
      int u_num,
      int v_num,
      int periodicity,
      std::vector<double*> points,
      std::vector<double> u_knots,
      std::vector<double> v_knots,
      std::vector<double> weights,
      double refine,
      pGFace& face)
  {
    check_surface_params(
        u_order, v_order, u_num, v_num, periodicity, points, u_knots, v_knots, weights);

    modeler->place_surface_by_spline(
        u_order, v_order, u_num, v_num, periodicity, points, u_knots, v_knots, weights, face);

    mesher->refine_face( refine, face);

    return;
  }

  void gmd_t::check_surface_params(
      int u_order,
      int v_order,
      int u_num,
      int v_num,
      int periodicity,
      std::vector<double*> points,
      std::vector<double> u_knots,
      std::vector<double> v_knots,
      std::vector<double> weights)
  {
    if( periodicity < 0 || periodicity > 4)
    { print_error("Bad periodicity.");}

    if( u_order<= 1 )
    { print_error("u_order too low.");}
    if( v_order<= 1 )
    { print_error("v_order too low.");}

    if( (int)points.size() != (u_num*v_num))
    { print_error("Mismatched number of declared and given control points.");}

    if( (int)weights.size() != 1 &&  (int)weights.size() != (u_num*v_num))
    { print_error("Mismatched number of weights and declared control points.");}
    if( (int)weights.size() == 1 && weights[0] != 0.0)
    { print_error("Bad weights vector.");}

    if( (int)u_knots.size() != (u_num+u_order))
    { print_error("Mismatch between number of u_knots and u_order+u_num.");}

    for( int i=0; i<(int)u_knots.size(); i++)
    {
      double tmp1 = u_knots[i];
      if( i<((int)u_knots.size() -1) )
      {
        double tmp2 = u_knots[i+1];
        if (tmp2 < tmp1)
        { print_error("Knots must be in accending u_order");}
      }
      if( i<u_order && tmp1 != 0.0)
      { print_error("First u_order_number of u_knots must be zero.");}
      if( i>((int)u_knots.size()-u_order) && tmp1 != 1.0)
      { print_error("Last u_order_number of u_knots must be one.");}

      if( tmp1 > 1.0 || tmp1 < 0.0)
      {
        print_error("Knots must satisfy 0.0<=k[i]<=1.0 .");
      }
    }

    if( (int)v_knots.size() != (v_num+v_order))
    { print_error("Mismatch between number of v_knots and v_order+v_num.");}

    for( int i=0; i<(int)v_knots.size(); i++)
    {
      double tmp1 = v_knots[i];
      if( i<((int)v_knots.size() -1) )
      {
        double tmp2 = v_knots[i+1];
        if (tmp2 < tmp1)
        { print_error("Knots must be in accending v_order");}
      }
      if( i<v_order && tmp1 != 0.0)
      { print_error("First v_order_number of v_knots must be zero.");}
      if( i>((int)v_knots.size()-v_order) && tmp1 != 1.0)
      { print_error("Last v_order_number of v_knots must be one.");}

      if( tmp1 > 1.0 || tmp1 < 0.0)
      {
        print_error("Knots␣must␣satisfy␣0.0<=k[i]<=1.0␣."↩
          );
      }
    }

    return;
  }
}
```

A.6 GeoMod_model_helper.hpp

```
#ifndef GEOMOD_MODEL_HELPER_HPP
#define GEOMOD_MODEL_HELPER_HPP

#include <string>
#include <vector>
#include "GeoMod_SIM.hpp"
#include "GeoMod_printer.hpp"
#include "GeoMod_coords.hpp"

pGVertex GIP_insertVertexInFace(pGIPart part, double* xyz↩
    , pGFace face);

namespace GMD
{
  class model_helper_t
  {
    friend class gmd_t;
    private:
      // Util methods
      model_helper_t (pGModel& in_model);
      ~model_helper_t();
      void model_print();
      void write( std::string name);
      bool isValid();
      bool isWritten();
      void unpack_vector_spline_points(
          std::vector<double*> vec,
          double* x);
```

```
        void unpack_vector( std::vector<double> vec, double* x);
        void unpack_surface_vector(
            std::vector<double*> points,
            double* all_points);
        void create_bounding_edges(
            int u_order,
            int v_order,
            int u_num,
            int v_num,
            std::vector<double*> points,
            std::vector<double> u_knots,
            std::vector<double> v_knots,
            std::vector<double> weights,
            std::vector<pGEdge>& edges);
        void unpack_bounding_edges(
            std::vector<pGEdge>& edges,
            pGEdge* bounding_edges);

        // Members
        pGModel model;
        pGIPart part;
        bool Written;

        // Methods to place a point
        bool place_point(
            double coords[3],
            pGVertex& vert,
            bool abort_on_fail);
        bool point_on_dim( int dim, double coords[3]);
        int point_location( double coords[3]);
        bool PointOnFace( double coords[3], pGFace face);
        bool PointOnEdge( double coords[3], pGEdge edge);
        void put_point_outside( double coords[3], pGVertex& vert);
        void put_point_in_line( double coords[3], pGVertex& vert);
        void put_point_in_face( double coords[3], pGVertex& vert);
        void put_point_in_region( double coords[3], pGVertex& vert);

        // Methods to place an edge
        void place_edge(
            int order,
            std::vector<double*> points,
            std::vector<double> knots,
            std::vector<double> weights,
            pGEdge& edge);
        void create_curve(
            int order,
            std::vector<double*> points,
            std::vector<double> knots,
            std::vector<double> weights,
            pCurve& curve);
        void create_edge(
            int order,
            std::vector<double*> points,
            pCurve& curve,
            pGEdge& edge);
        bool PointsOnSameFace( std::vector<double*> points);

        // Methods to place a surface
        void place_surface_by_spline(
            int u_order,
            int v_order,
            int u_num,
            int v_num,
            int periodicity,
            std::vector<double*> points,
            std::vector<double> u_knots,
            std::vector<double> v_knots,
            std::vector<double> weights,
            pGFace& face);
        void create_surface(
            int u_order,
            int v_order,
            int u_num,
            int v_num,
            int periodicity,
            std::vector<double*> points,
            std::vector<double> u_knots,
            std::vector<double> v_knots,
            std::vector<double> weights,
            pSurface& surface,
            std::vector<pGEdge>& edges);
        void create_face(
            pSurface& surface,
            std::vector<pGEdge>& edges,
            pGFace& face);

    };

  }

  #endif
```

A.7 GeoMod_model_helper.cpp

```
#include "GeoMod_model_helper.hpp"

namespace GMD
{
  model_helper_t::model_helper_t( pGModel& in_model)
  {
    model = in_model;
    Written = false;
    if(!isValid())
    {
      print_warning("Created gmd object on invalid model.");
    }
    part = GM_part( model);
    if( part == NULL)
    {
      print_error("GeoMod does not support assembly models.");
    }
    return;
  }

  model_helper_t::~model_helper_t()
  {
    GM_release( model);
    return;
  }

  void model_helper_t::model_print()
  {
    std::cout << "Modeler says hello!" << std::endl;
    return;
  }

  bool model_helper_t::isValid()
  {
    if(GM_isValid( model, 1, NULL) == 1)
    { return true; }
    else
    { return false; }
  }

  void model_helper_t::write( std::string name)
  {
    name = name + ".smd";
    const char* name_c = name.c_str();
    if( !isValid())
    { print_warning("Attempting to write invalid model.");}

    std::cout << "MODEL INFORMATION: "
      << "\nVertices: "<< GM_numVertices(model)
      << "\nEdges: "<< GM_numEdges(model)
      << "\nFaces: "<< GM_numFaces(model)
      << "\nRegions: "<< GM_numRegions(model) << std::endl;

    int writestat = GM_write(model, name_c, 0,0);
    if(writestat == 0)
    {
      std::cout << "Model " << name << " written." << std::endl;
      Written = true;
    }
    else
    {
      std::cout << "Model " << name
        << " failed to be written." << std::endl;
    }

    return;
  }

  bool model_helper_t::point_on_dim( int dim, double coords[3])
  {
    double tol = GM_tolerance( model);
    bool answer = false;
    bool areSame = false;
    double closest[] = {0.0, 0.0, 0.0};
    if( dim == 1)
    {
      GEIter e_it = GM_edgeIter( model);
      pGEdge e;
      while (( e = GEIter_next(e_it)))
      {
        GE_closestPoint(e, coords, closest, NULL);
        compare_coords(coords, closest, areSame, tol);
        if(areSame)
        {
          answer = true;
        }
      }
      GEIter_delete( e_it);
    }
    else if (dim == 2)
    {
      GFIter f_it = GM_faceIter( model);
      pGFace f;
      while (( f = GFIter_next(f_it)))
      {
        GF_closestPoint(f, coords, closest, NULL);
        compare_coords(coords, closest, areSame, tol);
        if(areSame)
        {
          answer = true;
        }
      }
      GFIter_delete( f_it);
    }
    else if (dim == 3)
    {
      print_warning("point on dim does not support this dimension");
    }
    return answer;
  }

  int model_helper_t::point_location(double coords[3])
  {
    int answer = 0;
    // Want to find classification on lowest gEnt dim
    // start from top work down, overwrite old answer
    for(int i=3; i>0; i--)
    {
      if (i<3)
      {
        if(point_on_dim( i, coords))
        {
          answer = i;
        }
      }
      else
      {
        GRIter r_it = GM_regionIter( model);
        pGRegion region;
        while(( region= GRIter_next(r_it)))
        {
          if(GR_containsPoint( region, coords) == 0)
          { // Point is in the void region (spooky!)
            return 0;
          }
          else if (GR_containsPoint( region, coords) ==1)
          {
            answer = 3;
          }
          else
          { print_error("Point placement not possible.");}
        }
        GRIter_delete( r_it);
      }
    }
    return answer;
  }

  bool model_helper_t::PointOnFace( double coords[3], pGFace face)
  {
    double tol = GM_tolerance( model);
    bool ans = false;
    double cp[] = {0.0, 0.0, 0.0};
    GF_closestPoint( face, coords, cp, NULL);
    compare_coords(coords, cp, ans, tol);
    return ans;
  }

  bool model_helper_t::PointOnEdge( double coords[3], pGEdge edge)
  {
    double tol = GM_tolerance( model);
    bool ans = false;
    double cp[] = {0.0, 0.0, 0.0};
    double para[] = {0.0, 0.0, 0.0};
    GE_closestPoint( edge, coords, cp, para);
    compare_coords(coords, cp, ans, tol);

    return ans;
  }
```

```

void model_helper_t::put_point_outside( double coords[3], pGVertex& vert)
{
  pGRegion out_region = GIP_outerRegion( part);
  vert = GIP_insertVertexInRegion( part, coords, out_region);
  return;
}

void model_helper_t::put_point_in_line( double coords[3], pGVertex& vert)
{
  bool placed = false;
  GEIter e_it = GM_edgeIter( model);
  pGEdge edge;
  while ( !placed && (edge = GEIter_next( e_it)))
  {
    if (PointOnEdge(coords,edge))
    {
      double param = 0.0;
      GE_closestPoint( edge, coords, NULL, &param);
      vert = GM_splitEdge( edge, param);
      if(vert == NULL)
      {
        pPList vert_list = PList_new();
        vert_list = GE_vertices( edge);
        pGVertex tmp_vert;
        void* iter = 0;
        bool found = false;
        while((!found) && ( tmp_vert = (pGVertex) PList_next( vert_list, &iter)))
        {
          bool areSame = false;
          double tol = GM_tolerance( model);
          double conf_coords[3] = {0.0};
          GV_point( tmp_vert, conf_coords);
          compare_coords( coords, conf_coords, areSame,tol);
          if( areSame)
          {
            found = true;
            vert = tmp_vert;
          }
        }
        PList_delete( vert_list);
      }
      placed = true;
    }
  }
  GEIter_delete( e_it);

  if( !placed)
  {
    print_warning("Failed to place point on edge at");
    print_coords( coords);
  }
  return;
}

void model_helper_t::put_point_in_face( double coords[3], pGVertex& vert)
{
  bool placed = false;
  GFIter f_it = GM_faceIter( model);
  pGFace face;
  while ( !placed && (face = GFIter_next( f_it)))
  {
    if (PointOnFace(coords,face))
    {
      vert = GIP_insertVertexInFace( part, coords, face);
      placed = true;
    }
  }
  GFIter_delete( f_it);

  if( !placed)
  {
    print_warning("Failed to place point in face at");
    print_coords( coords);
  }

  return;
}

void model_helper_t::put_point_in_region(
    double coords[3],
    pGVertex& vert)
{
  bool placed = false;
  GRIter r_it = GM_regionIter( model);
  pGRegion region;
  while( !placed && ( region= GRIter_next(r_it)))
  {
    if (GR_containsPoint( region, coords) ==1)
    {
      vert = GIP_insertVertexInRegion( part, coords, region);
      placed = true;
    }
  }
  GRIter_delete( r_it);
  return;
}

bool model_helper_t::place_point(
    double coords[3],
    pGVertex& vert,
    bool abort_on_fail)
{
  bool updateMesh = true;
  int location = point_location(coords);
  if( location == 0)
  {
    print_warning("Point outside of known regions.");
    updateMesh = false;
    put_point_outside( coords, vert);
  }
  else if ( location == 1)
  {
    put_point_in_line( coords, vert);
  }
  else if ( location == 2)
  {
    put_point_in_face( coords, vert);
  }
  else if ( location == 3)
  {
    put_point_in_region( coords, vert);
  }
  (void) abort_on_fail;
  return updateMesh;
}

void model_helper_t::unpack_vector_spline_points(
    std::vector<double*> vec,
    double* x)
{
  int pos = 0;
  double* tmp = vec[0];
  for( int i=0; i<(int)vec.size(); i++)
  {
    tmp = vec[i];
    for( int j=0; j<3; j++)
    {
      pos = j+3*i;
      x[pos] = tmp[j];
    }
  }
  return;
}

void model_helper_t::unpack_vector( std::vector<double> vec, double* x)
{
  for (int i=0; i<(int)vec.size(); i++)
  {
    x[i] = vec[i];
  }
  return;
}

void model_helper_t::create_curve(
    int order,
    std::vector<double*> points,
    std::vector<double> knots,
    std::vector<double> weights,
    pCurve& curve)
{
  int num_points = (int)points.size();
  bool weightLess = false;
  if((int)weights.size() == 1 && weights[0] == 0.0)
  { weightLess = true;}

  double u_points[num_points*3] = {0.0};
  double u_knots[(int)knots.size()] = {0.0};
  double un_weights[(int)weights.size()] = {0.0};
  unpack_vector_spline_points( points, u_points);
  unpack_vector( knots, u_knots);

  if(weightLess)
  {
    curve = SCurve_createBSpline(
      order, num_points, u_points, u_knots, NULL);
  }
  else
  {
    unpack_vector( weights, un_weights);
    curve = SCurve_createBSpline(
      order, num_points, u_points, u_knots, un_weights);
  }
  return;
}

bool model_helper_t::PointsOnSameFace( std::vector<double*> points)
{
  bool onFace = true;
  for( int i=0; i<(int)points.size(); i++)
  { // Check if all points are on any face
    if( !point_on_dim( 2, points[i]))
    {
      onFace = false;
    }
  }

  if( onFace)
  {
    bool answer = false;
    pGFace face;
    pGFace conFace;
    GFIter f_it = GM_faceIter( model);
    while(( face = GFIter_next(f_it)))
    {
      for(int i=0; i<(int)points.size(); i++)
      {
        if( (i==0) && PointOnFace( points[i], face))
        {
          conFace = face;
        }
        else if( PointOnFace( points[i], face))
        {
          if( face == conFace)
          { answer = true; }
          else
          { answer = false; }
        }
      }
    }
    GFIter_delete(f_it);
    return answer;
  }
  else
  {
    return false;
  }
}

```

```
void model_helper_t::create_edge(
    int order,
    std::vector<double*> points,
    pCurve& curve,
    pGEdge& edge)
{
  double* start_point = points[0];
  pGVertex start_vert = NULL;
  if( start_point == NULL)
  { print_error(" start_point is NULL ");}
  place_point( start_point, start_vert, false);

  double* end_point = points[(int)points.size()-1];
  pGVertex end_vert = NULL;
  place_point( end_point, end_vert, false);

  // Assumes a one region, one part model
  pGIPart part = GM_part( model);
  GRIter r_it = GM_regionIter( model);
  pGRegion region = GRIter_next( r_it);

  if( part == NULL)
  { print_error("Part is NULL");}
  if( start_vert == NULL)
  { print_error("start_vert is NULL");}
  if( end_vert == NULL)
  { print_error("end_vert is NULL");}
  if( curve == NULL)
  { print_error("curve is NULL");}
  if( region == NULL)
  { print_error("region is NULL");}

  edge = GIP_insertEdgeInRegion(
      part, start_vert, end_vert, curve, 1, region);
  GRIter_delete( r_it);

  bool onSameFace = PointsOnSameFace( points);
  if( onSameFace)
  {
    pGFace face;
    GFIter f_it = GM_faceIter( model);
    bool found = false;
    while( !found && (face = GFIter_next(f_it)))
    {
      if(PointOnFace(points[0], face))
      { found = true; }
    }
    GFIter_delete( f_it);
    pGFace new_faces[2] = {NULL, NULL};
    GM_insertEdgeOnFace( face, edge, new_faces);
  }

  (void) order;
  return;
}

void model_helper_t::place_edge(
    int order,
    std::vector<double*> points,
    std::vector<double> knots,
    std::vector<double> weights,
    pGEdge& edge)
{
  pCurve curve;
  create_curve( order, points, knots, weights, curve);
  create_edge( order, points, curve, edge);
  return;
}

void model_helper_t::place_surface_by_spline(
    int u_order,
    int v_order,
    int u_num,
    int v_num,
    int periodicity,
    std::vector<double*> points,
    std::vector<double> u_knots,
    std::vector<double> v_knots,
    std::vector<double> weights,
    pGFace& face)
```

```
{
  pSurface surf;
  std::vector<pGEdge> edges;

  create_surface(
      u_order, v_order, u_num, v_num, periodicity,
      points, u_knots, v_knots, weights, surf, edges);
  create_face( surf, edges, face);
  return;
}

void model_helper_t::unpack_bounding_edges(
    std::vector<pGEdge>& edges,
    pGEdge* bounding_edges)
{
  for( int i=0; i<(int)edges.size(); i++)
  {
    bounding_edges[i] = edges[i];
  }
  return;
}

void model_helper_t::create_face(
    pSurface& surface,
    std::vector<pGEdge>& edges,
    pGFace& face)
{

  pGIPart part = GM_part( model);
  int numEdges = (int)edges.size();
  pGEdge bounding_edges[numEdges] = {NULL};
  unpack_bounding_edges( edges, bounding_edges);
  int dirs[numEdges] = {0};
  for( int i = 0; i< numEdges; i++)
  {
    dirs[i] = 1;
  }
  int numLoops = 1;
  int indLoop[numLoops] = {0};
  for( int i = 0; i< numLoops; i++)
  {
    indLoop[i] = 0;
  }
  int normal = 1;

  GRIter r_it = GM_regionIter( model);
  pGRegion region = GRIter_next( r_it);

  face = GIP_insertFaceInRegion(
      part, numEdges, bounding_edges, dirs,
      numLoops, indLoop, surface, normal, region);

  GRIter_delete( r_it);
  return;
}

void model_helper_t::create_bounding_edges(
    int u_order,
    int v_order,
    int u_num,
    int v_num,
    std::vector<double*> points,
    std::vector<double> u_knots,
    std::vector<double> v_knots,
    std::vector<double> weights,
    std::vector<pGEdge>& edges)
{
  // Since GeomSim needs splines surface to be four sided
  // and have a regular control point spacing (X by Y),
  // the edges need can be infered from the surface points

  int N = u_num;
  int M = v_num;
  pGEdge tmp_edge0;
  bool weightLess = false;
  if((int)weights.size() == 1 && weights[0] == 0.0)
  { weightLess = true;}

```

```
    std::vector<double*> edge_points;
    std::vector<double> edge_weights;
    for( int n = 0; n<N; n++)
    {
      edge_points.push_back( points[n] );
      if(!weightLess)
      {
        edge_weights.push_back( weights[n]);
      }
    }

    if(weightLess)
    {
      edge_weights.push_back( 0.0);
    }
    place_edge( u_order, edge_points, u_knots, edge_weights, tmp_edge0);
    edges.push_back(tmp_edge0);
    edge_points.clear();

    pGEdge tmp_edge1;
    for( int m=0; m<M; m++)
    {
      int ind = (N-1)+m*N;
      edge_points.push_back( points[ind] );
      if(!weightLess)
      {
        edge_weights.push_back( weights[ind]);
      }
    }

    place_edge( v_order, edge_points, v_knots, edge_weights, tmp_edge1);
    edges.push_back(tmp_edge1);

    edge_points.clear();

    pGEdge tmp_edge2;
    for( int n=(N-1); n>(-1); n--)
    {
```

```
    int ind = n+(M-1)*N;
    edge_points.push_back( points[ind] );
    if(!weightLess)
    {
      edge_weights.push_back( weights[ind]);
    }
  }

  place_edge( u_order, edge_points, u_knots, edge_weights, tmp_edge2);
  edges.push_back(tmp_edge2);
  edge_points.clear();

  pGEdge tmp_edge3;
  for( int m=(M-1); m>(-1); m--)
  {
    int ind = N*m;
    edge_points.push_back( points[ind] );
    if(!weightLess)
    {
      edge_weights.push_back( weights[ind]);
    }
  }

  place_edge( v_order, edge_points, v_knots, edge_weights, tmp_edge3);
  edges.push_back(tmp_edge3);

  edge_points.clear();
  edge_weights.clear();

  return;
}

void model_helper_t::create_surface(
    int u_order,
    int v_order,
    int u_num,
    int v_num,
    int periodicity,
    std::vector<double*> points,
    std::vector<double> u_knots,
    std::vector<double> v_knots,
    std::vector<double> weights,
    pSurface& surface,
    std::vector<pGEdge>& edges)
{
  int u_per = 0;
  int v_per = 0;
  if (periodicity == 0)
  { // No chages from default
  }
  else if (periodicity == 1)
  {
    u_per = 1;
  }
  else if (periodicity == 2)
  {
    v_per = 1;
  }
  else if (periodicity == 3)
  {
    u_per = 1;
    v_per = 1;
  }

  bool weightLess = false;
  if((int)weights.size() == 1 && weights[0] == 0.0)
  { weightLess = true;}

  int num_points = (int)points.size();

  double unp_u_knots[(int)u_knots.size()] = {0.0};
  unpack_vector( u_knots, unp_u_knots);

  double unp_v_knots[(int)v_knots.size()] = {0.0};
  unpack_vector( v_knots, unp_v_knots);

  double all_points[3*num_points] = {0.0};
  unpack_vector_spline_points( points, all_points);

  create_bounding_edges(
      u_order, v_order, u_num, v_num,
      points, u_knots, v_knots, weights, edges);

  if(weightLess)
  {
    surface = SSurface_createBSpline(
        u_order, v_order,
        u_num, v_num,
        u_per, v_per,
        all_points, NULL,
        unp_u_knots, unp_v_knots);
  }
  else
  {
    double unp_weights[num_points] = {0.0};
    unpack_vector( weights, unp_weights);
    surface = SSurface_createBSpline(
        u_order, v_order,
        u_num,   v_num,
        u_per,   v_per,
        all_points, unp_weights,
        unp_u_knots, unp_v_knots);
  }
  return;
}

bool model_helper_t::isWritten()
{ return Written;}
}
```

A.8 GeoMod_mesh_helper.hpp

```
#ifndef GEOMOD_MESH_HELPER_HPP
#define GEOMOD_MESH_HELPER_HPP

#include <string>
#include <vector>
#include "GeoMod_SIM.hpp"
```

```
#include "GeoMod_printer.hpp"

namespace GMD
{
  class mesh_helper_t
  {
    friend class gmd_t;
    private:
      // Util methods
      mesh_helper_t( pGModel in_model);
      ~mesh_helper_t();
      void mesh_print();
      void write( std::string name);
      bool isValid();
      void create();

      // Members
      pMesh mesh;
      pACase m_case;
      bool globalSet;
      double order;
      double refine;
      double grad_rate;

      // Mesh preping methods
      void place_point(
          double coords[3],
          double refine,
          double radius,
          bool abort_on_fail);
      void set_global( int order_in, double refine_in, double grad_rate_in);
      void refine_vertex( double refine, pGVertex vert);
      void refine_edge( double refine, pGEdge edge);
      void refine_face( double refine, pGFace face);

  };

}

```

```
#endif
```

A.9 GeoMod_mesh_helper.cpp

```
#include "GeoMod_mesh_helper.hpp"

namespace GMD
{
  mesh_helper_t::mesh_helper_t( pGModel in_model)
  {
    mesh = M_new( 0, in_model);
    m_case = MS_newMeshCase( in_model);
    globalSet = false;

    return;
  }

  mesh_helper_t::~mesh_helper_t()
  {
    MS_deleteMeshCase(m_case);
    M_release( mesh);
    return;
  }

  void mesh_helper_t::mesh_print()
  {
    std::cout << "Mesher says hello!" << std::endl;
    return;
  }

  bool mesh_helper_t::isValid()
  { // Only validates serial meshes for now
    bool ans = false;
    pGModel model = M_model( mesh);
    pPList mesh_list = PList_new();
    PList_append( mesh_list, mesh);
    pParMesh par_mesh = PM_createFromMesh(
        model,
        M_representation(mesh),
        mesh_list, NULL, NULL, NULL);
```

```cpp

  int status = PM_verify(par_mesh, 0, NULL);
  if (status == 1)
  { ans = true;}

  M_release(par_mesh);
  PList_delete( mesh_list);
  return ans;
}

void mesh_helper_t::create()
{
  if(!globalSet)
  { print_error("Global␣Mesh␣Parameters␣not␣set.");}
  pModelItem domain = GM_domain( M_model(mesh));
  MS_setMeshSize(m_case, domain, 2, refine, NULL);

  if( grad_rate > 0.0)
  { MS_setGlobalSizeGradationRate(m_case, grad_rate); }

  if( order == 2)
  { MS_setMeshOrder(m_case, order);}

  pSurfaceMesher surf = SurfaceMesher_new(m_case, mesh);
  SurfaceMesher_execute( surf, NULL);
  SurfaceMesher_delete( surf);

  pVolumeMesher vol = VolumeMesher_new( m_case, mesh);
  VolumeMesher_execute( vol, NULL);
  VolumeMesher_delete(vol);

  return;
}

void mesh_helper_t::write( std::string name)
{
  std::string tmp_name = name + ".sms";
  const char* name_c = tmp_name.c_str();
  if( !isValid())
  { print_warning("Attempting to write invalid mesh.");}

  std::cout << "MESH_INFORMATION: "
    << "\nVertices: "<< M_numVertices(mesh)
    << "\nEdges: "<< M_numEdges(mesh)
    << "\nFaces: "<< M_numFaces(mesh)
    << "\nRegions: "<< M_numRegions(mesh) << std::endl;

  int writestat = M_write(mesh, name_c, 0,0);
  if(writestat == 0)
  {
    std::cout << "Mesh " << name << " written." << std::endl;
  }
  else
  { std::cout << "Mesh " << name << " failed to be written." << std::endl; }
  return;
}

void mesh_helper_t::refine_vertex( double refine, pGVertex vert)
{
  MS_setMeshSize(m_case, vert, 2, refine, NULL);
  return;
}

void mesh_helper_t::place_point(
    double coords[3],
    double refine,
    double radius,
    bool abort_on_fail)
{
  if (refine >0.0)
  {
    if (radius == 0.0)
    {
      MS_addPointRefinement( m_case, refine, coords);
    }
    else if (radius > 0.0)
    {
      MS_addSphereRefinement( m_case, refine, radius, coords);
    }
    else if(abort_on_fail)
    {
      print_error( "Refinement radius must be zero or greater");
    }
    else
    {
      print_warning( "Refinement radius must be zero or greater");
    }
  }
  return;
}

void mesh_helper_t::set_global( int order_in, double refine_in, double grad_rate_in)
{
  globalSet = true;
  order = order_in;
  refine = refine_in;
  grad_rate = grad_rate_in;
  return;
}

void mesh_helper_t::refine_edge( double refine, pGEdge edge)
{
  MS_setMeshSize(m_case, edge, 2, refine, NULL);
  return;
}

void mesh_helper_t::refine_face( double refine, pGFace face)
{
  MS_setMeshSize(m_case, face, 2, refine, NULL);
```

```
      return;
    }
}
```

A.10 GeoMod_printer.hpp

```
#ifndef GEOMOD_PRINTER_HPP
#define GEOMOD_PRINTER_HPP

#include <cstdlib>
#include <iostream>
#include <string.h>

namespace GMD
{
  void print_error( std::string message, bool ↩
      abort_on_fail=true);

  void print_warning( std::string message);

  void print_coords( double x[3]);
}
#endif
```

A.11 GeoMod_printer.cpp

```
#include "GeoMod_printer.hpp"

#include <cstdlib>
#include <iostream>
#include <string.h>

namespace GMD
{
  void print_error( std::string message, bool ↩
      abort_on_fail)
  {
    std::cout << "Error:␣" <<  message << std::endl;
    if( abort_on_fail)
    {
      std::abort();
    }
    return;
  }

  void print_warning( std::string message)
  {
    std::cout << "Warning:␣" << message << std::endl;
    return;
  }

  void print_coords( double x[3])
  {
    std::cout << "("  ;
    for (int i=0; i<3; i++)
    {
      std::cout << x[i] << ",␣";
    }
    std::cout << "\b\b)\n";
    return;
  }
}
```

A.12 GeoMod_coords.hpp

```
#ifndef GEOMOD_COORDS_HPP
#define GEOMOD_COORDS_HPP

namespace GMD
{
  void sum_coords(double x[3], double y[3], double ans[3]);

  void subtract_coords( double fin[3], double intil[3], double ans[3]);

  void divide( double vec[3], double denom, double ans[3]);

  void get_mag(double vec[3], double& mag);

  void get_unit_vector( double vec[3], double unit[3]);

  void cross_product( double x[3], double y[3], double ↩
     ans[3]);

  void dot_product( double x[3], double y[3], double& ans↩
     );

  void compare_coords( double x[3], double y[3], bool& ↩
     areSame, double tol=0.0);

}
#endif
```

A.13 GeoMod_coords.cpp

```
#include ”GeoMod_coords.hpp”
#include ”GeoMod_printer.hpp”
#include <cstdlib>
#include <math.h>

namespace GMD
{
  void sum_coords(double x[3], double y[3], double ans↩
     [3])
  {
    for(int i=0; i<3; i++)
    {
      ans[i] = x[i]+y[i];
    }
    return;
  }

  void subtract_coords( double fin[3], double intil[3], ↩
     double ans[3])
  {
    for(int i=0; i<3; i++)
    {
```

```
    ans[i] = fin[i] - intil[i];
  }
  return;
}

void get_mag(double vec[3], double& mag)
{
  double tmp = 0.0;
  for (int i=0; i<3; i++)
  {
    tmp += (vec[i]*vec[i]);
  }
  mag = sqrt( tmp);
}

void compare_coords( double x[3], double y[3], bool& areSame, double tol)
{
  areSame = false;
  double mag;
  double ans[] = {0.0, 0.0, 0.0};
  subtract_coords( x, y, ans);
  get_mag( ans, mag);
  if (mag<=tol)
  {
    areSame = true;
  }
  return;
}

void divide( double vec[3], double denom, double ans[3])
{
  if(denom == 0.0)
  {
    print_error("Denominator is zero. Division not defined.");
  }
  else
  {
    for(int i=0; i<3; i++)
    {
      ans[i] = vec[i]/denom;
    }
  }
  return;
}

void get_unit_vector( double vec[3], double unit[3])
{
  double mag = 0.0;
  get_mag( vec, mag);
  divide(vec, mag, unit);
  return;
}

void cross_product( double x[3], double y[3], double ans[3])
{
  ans[0] = x[1]*y[2]-x[2]*y[1];
  ans[1] = x[2]*y[0]-x[0]*y[2];
  ans[2] = x[0]*y[1]-x[1]*y[0];
  return;
}

void dot_product( double x[3], double y[3], double& ans)
{
  for(int i=0; i<3; i++)
  {
    ans = x[i]*y[i];
  }
  return;
}

}
```

A.14 GeoMod_util.hpp

```
#ifndef GEOMOD_UTIL_HPP
```

```
#define GEOMOD_UTIL_HPP

// Simmetrix Headers
#include "SimUtil.h"
#include "SimModel.h"
#include "SimAdvModel.h"
#include "MeshSim.h"
#include "SimPartitionedMesh.h"

// Standard C++ Headers
#include <cstdlib>
#include <iostream>

pGVertex GIP_insertVertexInFace(pGIPart part, double* xyz, pGFace face);

namespace GMD
{
  void sim_start( char* Sim_log_file_name, int argc, char** argv);

  void sim_end();
  pGModel create_cube(double length);
  pGModel create_2D_rectangle( double y_length, double x_width);
}
#endif
```

A.15 GeoMod_util.cpp

```
// This File's Header
#include "GeoMod_util.hpp"
// NOTE: All needed headers belong in GeoMod_util.hpp

namespace GMD
{
  void sim_start( char* Sim_log_file_name, int argc, char** argv)
  {
    std::cout << "Starting Simmetrix" << std::endl;
    SimPartitionedMesh_start(&argc, &argv);
    Sim_logOn( Sim_log_file_name);
    SimUtil_start();
    Sim_readLicenseFile(0);
    SimModel_start();
    MS_init();
  }

  void sim_end()
  {
    std::cout << "Stopping␣Simmetrix" << std::endl;
    MS_exit();
    SimModel_stop();
    Sim_unregisterAllKeys();
    SimUtil_stop();
    Sim_logOff();
    SimPartitionedMesh_stop();
  }
  pGModel create_cube(double length)
  {
    pGModel model;
    pGIPart part;
    pGRegion outerRegion;
    pGVertex vertices[8]; // array to store the returned model vertices
    pGEdge edges[12]; // array to store the returned model edges
    double hl = length/2;
    model = GM_new();
    part = GM_part(model);
    outerRegion = GIP_outerRegion(part);

    double vert_xyz[8][3] =
    { {-hl,-hl,-hl},
      { hl,-hl,-hl},
      { hl, hl,-hl},
      {-hl, hl,-hl},
      {-hl,-hl, hl},
      { hl,-hl, hl},
      { hl, hl, hl},
          {-hl, hl, hl} };

      int i;
      for(i=0; i<8; i++)
          vertices[i] = GIP_insertVertexInRegion(part,vert_xyz[i],outerRegion);

      pGVertex startVert, endVert;
      double point0[3],point1[3];  // xyz locations of the two vertices
      pCurve linearCurve;

      // First, the bottom edges
      for(i=0; i<4; i++) {
          startVert = vertices[i];
          endVert = vertices[(i+1)%4];
          GV_point(startVert, point0);
          GV_point(endVert, point1);
          linearCurve = SCurve_createLine(point0, point1);
          edges[i] =
              GIP_insertEdgeInRegion(part, startVert, endVert, linearCurve, 1, outerRegion);
      }

      // Now the side edges of the box
      for(i=0; i<4; i++) {
          startVert = vertices[i];
          endVert = vertices[i+4];
          GV_point(startVert,point0);
          GV_point(endVert,point1);
          linearCurve = SCurve_createLine(point0,point1);
          edges[i+4] =
              GIP_insertEdgeInRegion(part,startVert, endVert, linearCurve, 1, outerRegion);
      }

      // Finally the top edges
      for(i=0; i<4; i++) {
          startVert = vertices[i+4];
          endVert = vertices[(i+1)%4+4];
          GV_point(startVert, point0);
          GV_point(endVert, point1);
          linearCurve = SCurve_createLine(point0, point1);
          edges[i+8] =
              GIP_insertEdgeInRegion(part, startVert, endVert, linearCurve, 1, outerRegion);
          }

      double corner[3], xPt[3], yPt[3];  // the points defining the surface of the face
      pGEdge faceEdges[4];               // the array of edges connected to the face
      int faceDirs[4];                   // the direction of the edge with respect to the face
      int loopDef[1] = {0};
      pSurface planarSurface;

      // First the bottom face
      // Define the surface - we want the normal to point out of the box
      for(i=0; i<3; i++)
      {
          corner[i] = vert_xyz[1][i];
          xPt[i] = vert_xyz[0][i];
          yPt[i] = vert_xyz[2][i];
      }
      planarSurface = SSurface_createPlane(corner,xPt,yPt);
      // Define and insert the face into the outer "void" region
      for(i=0; i<4; i++) {
          faceDirs[i] = 0;
          faceEdges[i] = edges[3-i]; // edge order 3->2->1->0
      }
      GIP_insertFaceInRegion(part,4,faceEdges,faceDirs,1,loopDef,planarSurface,1,outerRegion);

      // Now the side faces of the box - each side face has the edges defined in the same way
      // for the first side face, the edge order is 0->5->8->4
      for(i=0; i<4; i++) {
          //Define surface such that normals all point out of the box
          for(int j=0; j<3; j++) {
              corner[j] = vert_xyz[i][j];        // the corner is the lower left vertex location
              xPt[j] = vert_xyz[(i+1)%4][j];     // the xPt the lower right vertex location
              yPt[j] = vert_xyz[i+4][j];         // the yPt is the upper left vertex location
          }
          planarSurface = SSurface_createPlane(corner,xPt,yPt);

          faceEdges[0] = edges[i];
          faceDirs[0] = 1;
          faceEdges[1] = edges[(i+1)%4+4];
          faceDirs[1] = 1;
          faceEdges[2] = edges[i+8];
          faceDirs[2] = 0;
          faceEdges[3] = edges[i+4];
          faceDirs[3] = 0;

          GIP_insertFaceInRegion(part,4,faceEdges,faceDirs,1,loopDef,planarSurface,1,outerRegion);
      }

      // Finally the top face of the box
      // Define the surface - we want the normal to point out of the box
      for(i=0; i<3; i++) {
          corner[i] = vert_xyz[4][i];
          xPt[i] = vert_xyz[5][i];
          yPt[i] = vert_xyz[7][i];
      }
      planarSurface = SSurface_createPlane(corner,xPt,yPt);
      // Define and insert the face
      for(i=0; i<4; i++) {
          faceDirs[i] = 1;
          faceEdges[i] = edges[i+8]; // edge order 8->9->10->11
      }
      // when this face is inserted, a new model region will automatically be created
      GIP_insertFaceInRegion(part,4,faceEdges,faceDirs,1,loopDef,planarSurface,1,outerRegion);

      return model;
    }

    pGModel create_2D_rectangle( double y_length, double x_width)
    {
      // Create an empty modeling space to work with
      pGModel model = GM_new();
      pGIPart part = GM_part(model);
      pGRegion outRegion = GIP_outerRegion(part);

      // Create model vertices
      pGVertex verts[4];
      double vert_xyz[4][3] ={{ 0.0, 0.0, 0.0},
          { y_length, 0.0, 0.0},
          { y_length, x_width, 0.0},
          { 0.0, x_width, 0.0}};
      for(int i=0; i<4; i++)
      {
          verts[i] = GIP_insertVertexInRegion( part, vert_xyz[i], outRegion);
      }

      // Create model edges
      pCurve line;
      pGEdge edges[4];
      for(int i=0; i<4; i++)
      {
          pGVertex start = verts[i];
          pGVertex end = verts[(i+1)%4];
          line = SCurve_createLine(vert_xyz[i], vert_xyz[(i+1)%4]);
```

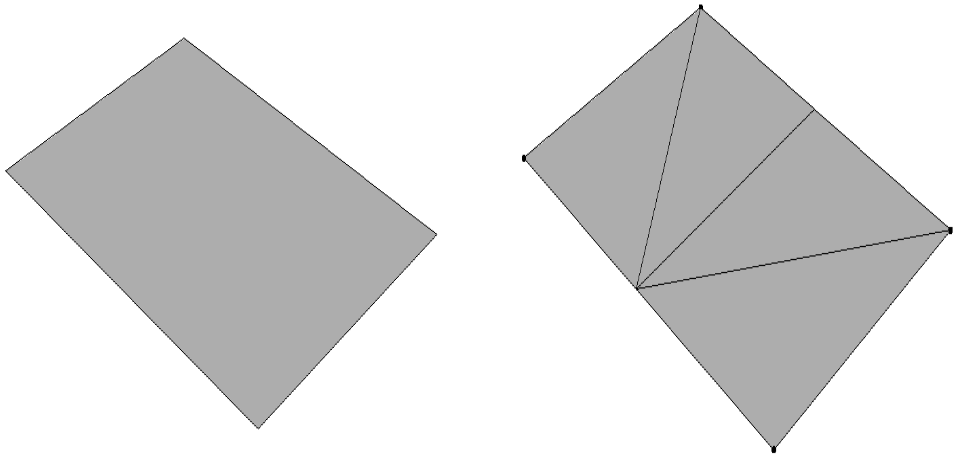
```
      edges[i] = GIP_insertEdgeInRegion( part, start, end, line, 1, outRegion);
    }

    // Create Face
    pSurface plane;
    int face_dirs[4] = {1,1,1,1};
    int loopDef[1] = {0};
    plane = SSurface_createPlane(vert_xyz[0], vert_xyz[1], vert_xyz[2]);
    GIP_insertFaceInRegion(part, 4, edges, face_dirs, 1, loopDef, plane, 1, outRegion);

    return model;
  }
}
```

A.16 GeoMod_SIM.hpp

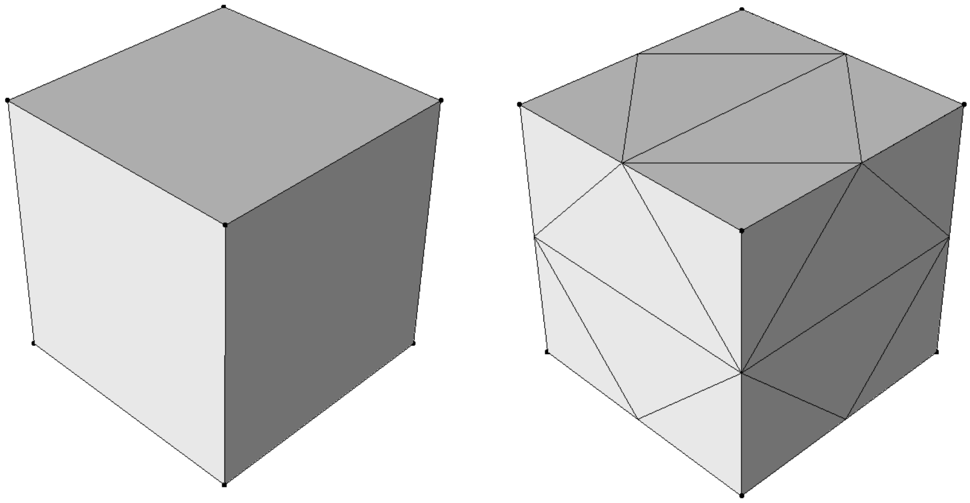
```
#ifndef GEOMOD_SIM_HPP
#define GEOMOD_SIM_HPP

// Simmetrix Headers
#include ”SimUtil.h”
#include ”SimModel.h”
#include ”SimAdvModel.h”
#include ”MeshSim.h”
#include ”SimPartitionedMesh.h”

#endif
```

B Supplementary Images

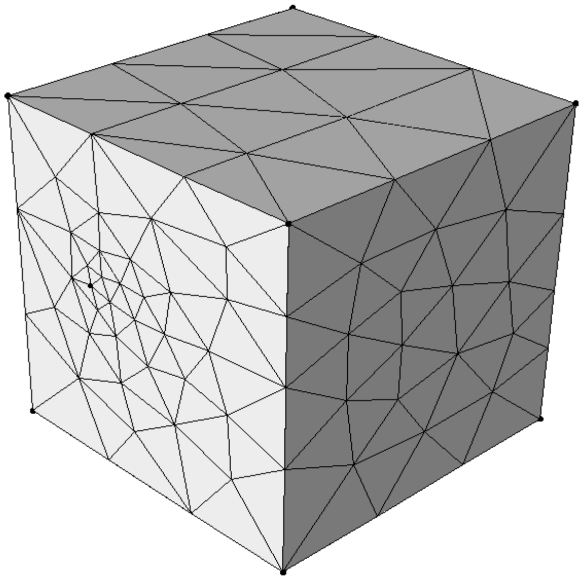
B.1 Model and Mesh of Test 1



B.2 Model and Mesh of Test 2



B.3 Mesh of Test 11



B.4 Mesh of Test 12