FEP: Assignment 2

Justin Clough, RIN:661682899

May 19, 2017

Algorithm Description

The reordering algorithm has two main steps, not including loading or writing the mesh. The first step was to find a starting vertex. The second step was to then use the reverse Cuthill-McKee algorithm to reorder the mesh nodes.

The starting mesh vertex was found by first using reverse classification to create a list of mesh vertices classified on model vertices. This list of mesh vertices was then iterated over twice. The first iteration determined the average spatial coordinates of the mesh vertices. The second iteration determined which mesh vertex was furthest away from this average point. The mesh vertex with the greatest distance from the center was then used as the starting vertex. A better alternative would have been to use graph traversal. This would instead find the maximum length of the minimum paths between mesh verteces. Either of the vertices at the ends of this path could then be used as a starting vertex for the reverse Cuthill-McKee algorithm.

The reverse Cuthill-McKee started by entering a given staring node into a queue. The queue had three functions to be used when passed a mesh entity:

1. *put_back()* which placed items in the queue if a copy of it was not already in the queue and it was not already numbered
2. *drop_front()* which returned and removed the front-most item in the queue
3. *visited()* which returned *true* if the item was already in the queue or has been numbered

While the queue was not empty, the first item was removed and numbered. The numbering started with the the total number of nodes on the mesh and was decremented each time a node was numbered. If the node corresponded to a mesh vertex, then the adjacent mesh edges were collected. These edges were then iterated over. For each edge, the adjacent mesh faces were collected. The mesh faces were then iterated over; they were numbered if they were not numbered already. While iterating over the adjacent edges, the vertex on the opposite side of the edge was collected and temporarily stored in another queue if it was not already numbered. Additionally, if mesh nodes corresponded to edges and the edge was not already numbered, then it was added to the temporary queue. After all adjacent edges were iterated over, the temporary queue was added to the main queue.

Figures of the mesh both before and after reordering are shown in Figures 1 through 6. The code begins on page 5. *Before* and *After* reordering images are shown on the same page for clarity.

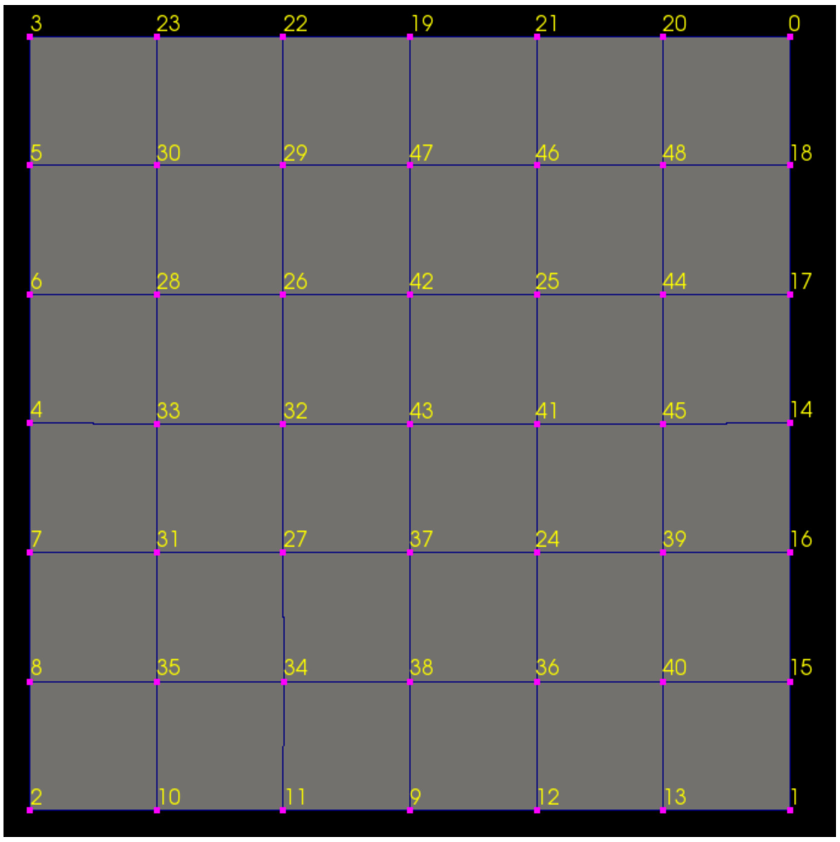

Figure 1: Original Numbering for Mesh A.

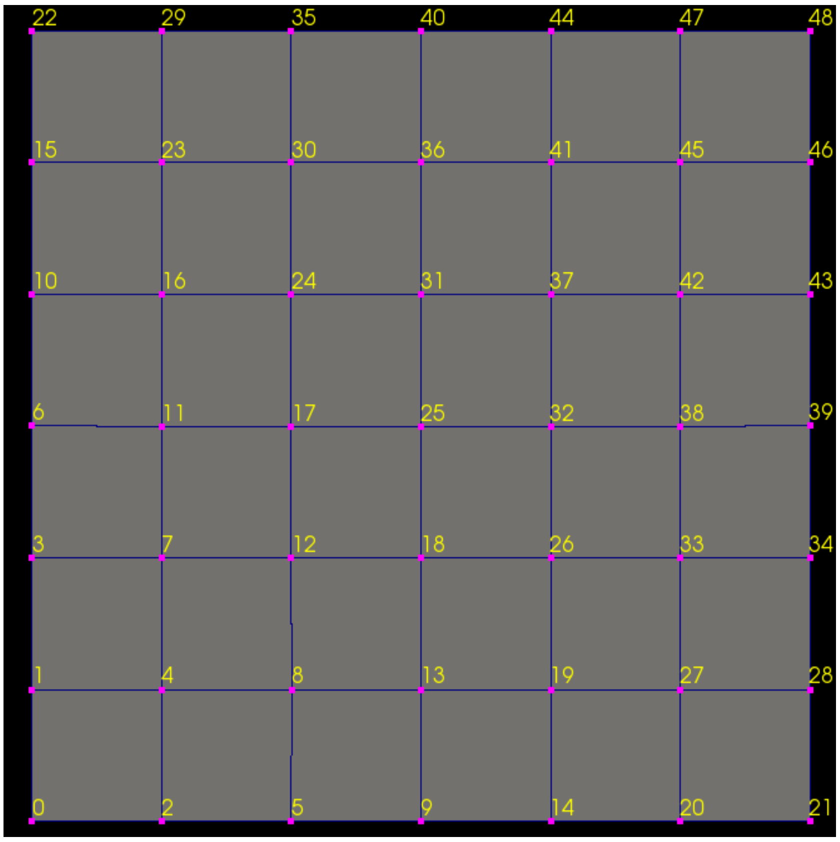

Figure 2: Reorded Numbering for Mesh A.

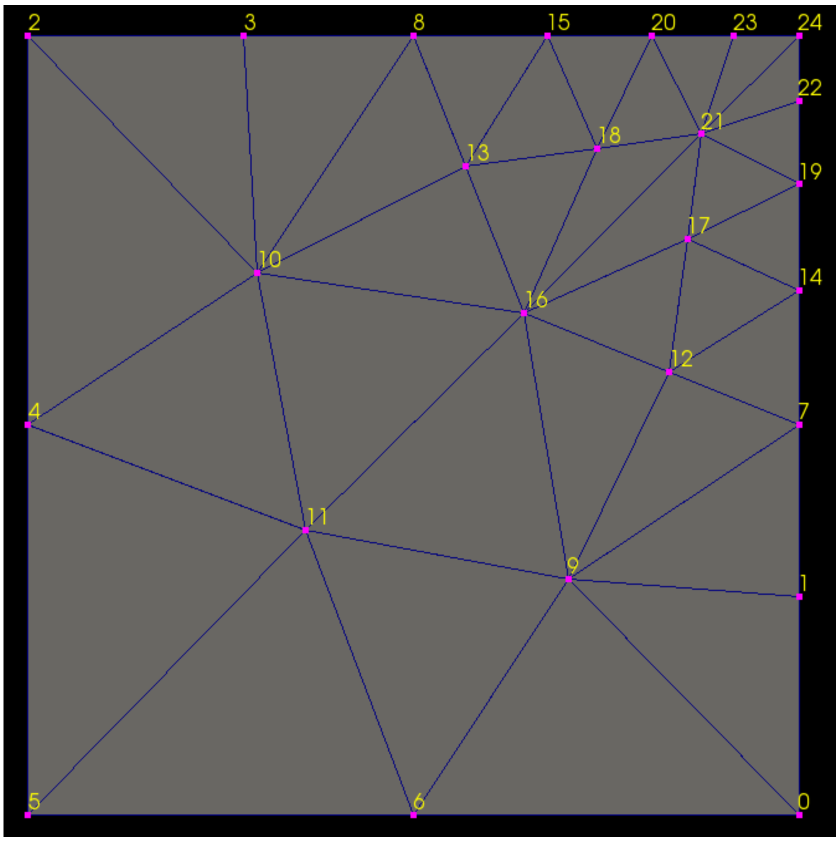

Figure 3: Original Numbering for Mesh B.

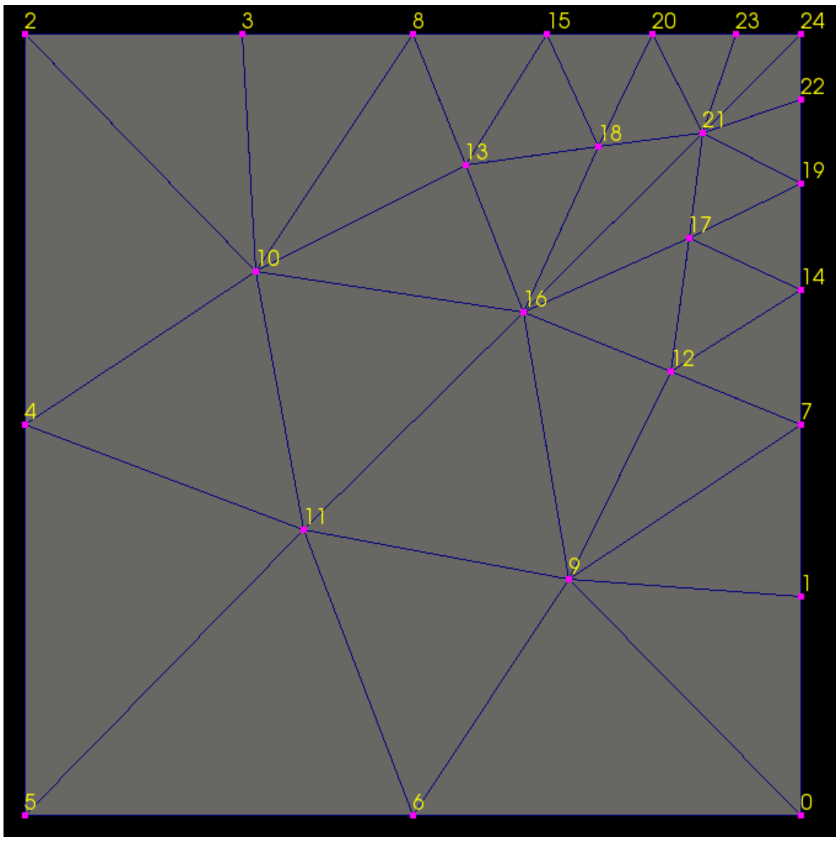

Figure 4: Reorded Numbering for Mesh B.

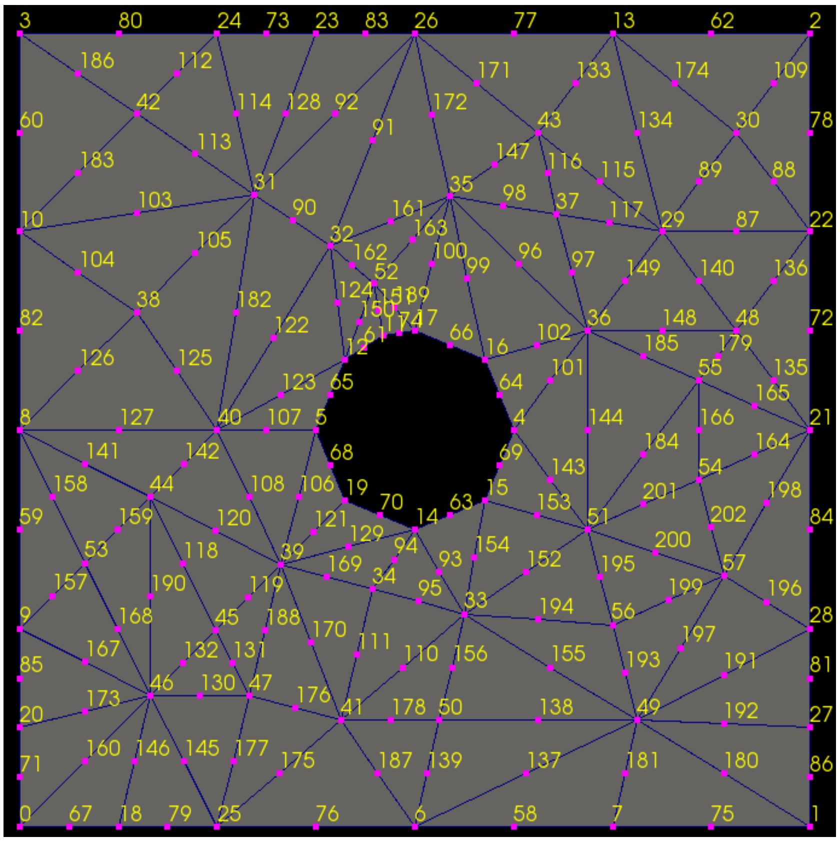
Figure 5: Original Numbering for Mesh C.

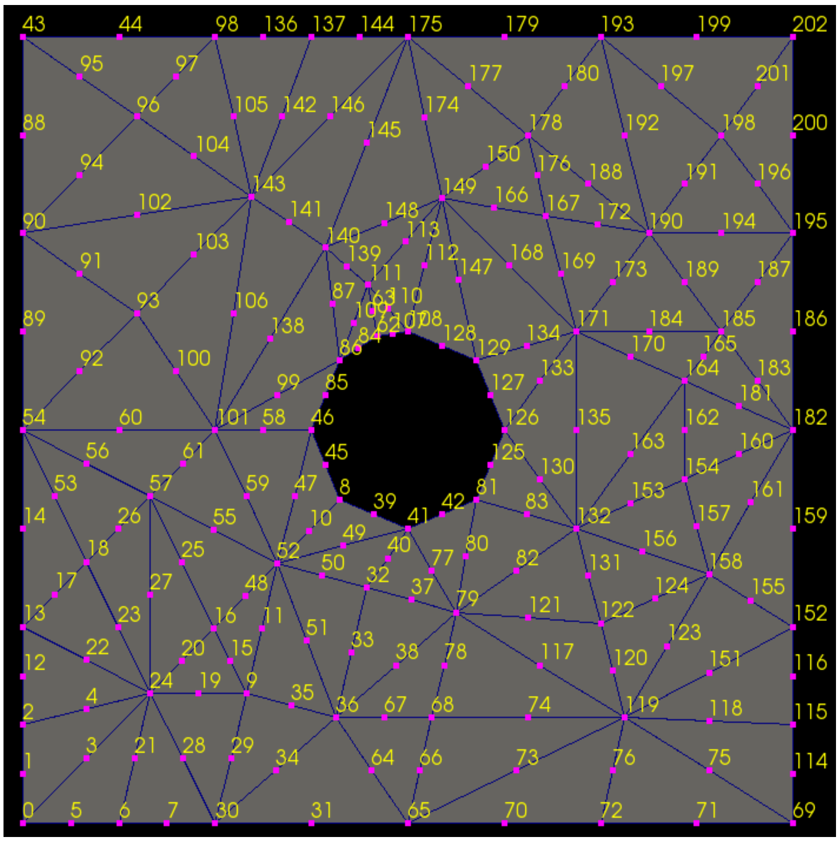
Figure 6: Reorded Numbering for Mesh C.

Code

```
// PUMI Headers
#include <PCU.h>
#include <pumi.h>
#include <apfNumbering.h>
#include <apfShape.h>

// STL Headers
#include <stdlib.h>
#include <string.h>
#include <vector>
#include <iterator>
#include <iostream>
#include <fstream>
#include <deque>

using std::cout;
using std::endl;

class Line
{
  public:
    std::deque<pMeshEnt> q;
    std::set<pMeshEnt> set;
    void put_back(pMeshEnt e, pNumbering num)
    {
      if( (set.count(e) == 0) && (!pumi_ment_isNumbered( e, num)))
      {
        q.push_back(e);
        set.insert(e);
      }
    }
    pMeshEnt drop_front()
    {
      pMeshEnt e = q.front();
      q.pop_front();
      set.erase(e);
      return e;
    }
    bool visited( pMeshEnt e, pNumbering num)
    {
      bool isin;
      if(set.count(e) > 0)
      {
        isin = true;
      }
     return (pumi_ment_isNumbered( e, num) || isin);
    }
    void add_list( std::vector<pMeshEnt> list , pNumbering num)
    {
      for(int i=0; i< (int) list.size(); i++)
      {
        put_back(list[i], num);
      }
    }
  //End Public:
};

double get_sq_magnitude( double* Coords)
{
  double sq_mag = 0.0;
  // Calculate sum of squares of components
  for (int i=0; i<3; i++)
  {
    sq_mag = sq_mag + (Coords[i])*(Coords[i]);
  }
  return sq_mag;
}

pMeshEnt find_start( pGeom geom, pMesh mesh)
{
  std::cout << std::endl;
  std::cout << "Finding starting vertex." << std::endl;
  // Declare working variables
  std::vector<pMeshEnt> model_Verts;
  std::vector<pMeshEnt> Verts;
  pMeshEnt Start_Vert;
  double Coords[3] = {0,0,0};
  double Sum_Coords[3] = {0,0,0};
  double Ave_Coords[3] = {0,0,0};
  double Dif_Coords[3] = {0,0,0};
  double distance = 0.0;
  double g_distance = 0.0;
  double n = 0;

  // For every model vertex...
  for(pGeomIter it=geom->begin(0); it!=geom->end(0); ++it)
  {
    // Get mesh vertex classified on model vertex
    pumi_gent_getRevClas( *it, model_Verts);
    Verts.push_back(model_Verts[0]);
    // Update vertex set stats
    pumi_node_getCoord( model_Verts[0], 0, Coords);
    // Clear vector for reuse
    model_Verts.clear();
    for(int i=0; i<3; i++)
    {
      Sum_Coords[i] = Coords[i];
    }
    n = n + 1.0;
  }
  // Get average location of all model vertices
  for(int i=0; i<3; i++)
  {
    Ave_Coords[i] = Sum_Coords[i]/n;
  }
  // For every mesh vertex on model
  for( int i=0; i< (int)Verts.size(); i++)
  {
    // Get this Vertex's Coordinates
    pumi_node_getCoord( Verts[i], 0, Coords);
    // Compare this vertex's coordinates to the average; get the
       distance
    for(int j=0; j<3; j++)
    {
      Dif_Coords[j] = Ave_Coords[j] - Coords[j];
    }
    distance = get_sq_magnitude( Dif_Coords );
    // If this vertex is further from the center of all vertices than
        all others
    //  (ties go to the home team, not runner)
    if( distance  g_distance)
    {
      // Then, it is likely to be the most extreme vertex
      Start_Vert = Verts[i];
      // Update distance to beat
      g_distance = distance;
    } // ENDIF
  } // ENDFOR every vertex on the model vertices

  // Clean up
  Verts.clear();

  std::cout << "Found starting vertex." << std::endl;
  return Start_Vert;
} //END find_start( pGeom geom, pMesh mesh);

void write_before( pGeom geom, pMesh mesh)
{
  pNumbering un_node = pumi_numbering_createOwnedNode( mesh, "Node");
  pNumbering un_elem = pumi_numbering_createOwned( mesh, "Element",
     pumi_mesh_getDim(mesh));

  pumi_mesh_write(mesh,"Before_Reorder","vtk");

  pumi_numbering_delete( un_node);
  pumi_numbering_delete( un_elem);
  return;
}
```

```
void reorder_mesh( pGeom geom, pMesh mesh, pMeshEnt start_vert)
{
  // Get Mesh dimension and shapes
  int mesh_dim = pumi_mesh_getDim(mesh);
  pShape node_shape = pumi_mesh_getShape( mesh );
  pShape elem_shape = pumi_shape_getConstant( mesh_dim );

  // Create empty numbering schemes
  pNumbering node_numbering = pumi_numbering_create( mesh, "Node",
     node_shape);
  pNumbering elem_numbering = pumi_numbering_create( mesh, "Element",
      elem_shape);

  int label_dof = pumi_mesh_getNumEnt( mesh, 0);
      // Number of nodes on verts
  int num_node_line = pumi_shape_getNumNode( node_shape, 1);
  if (num_node_line > 0)
  {
    label_dof+=num_node_line*pumi_mesh_getNumEnt( mesh,1);
                         // Num nodes on edges
  }
  int label_elem = pumi_mesh_getNumEnt( mesh, mesh_dim);

  Line line;
  // Begin line with starting vertex
  line.put_back(start_vert, node_numbering);
  while( (int)line.q.size() > 0)
  {
    pMeshEnt e = line.drop_front();
    if(! pumi_ment_isNumbered( e, node_numbering))
    {
      pumi_ment_setNumber( e, node_numbering, 0, 0, --label_dof);
    }
    std::vector<pMeshEnt> list;
    if(pumi_ment_getDim( e) == 0)
    {
      // Iterate on edges connected to vertex
      pMeshEnt vert = e;
      std::vector<pMeshEnt> adj_edges;
      pumi_ment_getAdj( vert, 1, adj_edges);
      for (int i=0; i<(int) adj_edges.size(); i++)
      {
        pMeshEnt edge = adj_edges[i];
        std::vector<pMeshEnt> adj_faces;
        pumi_ment_getAdj( edge, 2, adj_faces);
        for (int j=0; j<(int) adj_faces.size(); j++)
        {
          // label faces not already labeled
          pMeshEnt face = adj_faces[j];
          if( !pumi_ment_isNumbered(face, elem_numbering))
          {
            pumi_ment_setNumber( face, elem_numbering, 0, 0, --
               label_elem);
          }
        }
        adj_faces.clear();
        adj_faces.resize(0);
        // Get vertex opposite this one on current edge
        pMeshEnt otherVert = pumi_medge_getOtherVtx( edge, vert);
        if(pumi_shape_hasNode( node_shape, 1))
        {
          if(line.visited( otherVert, node_numbering) &&
             (!pumi_ment_isNumbered(edge,node_numbering)))
          {
            pumi_ment_setNumber( edge, node_numbering, 0,0, --
               label_dof);
          }
          else
          {
            line.put_back(edge, node_numbering);
            list.push_back(otherVert);
          }
        }
        else
        {
          if(!pumi_ment_isNumbered( otherVert, node_numbering))
          {
            list.push_back(otherVert);
          }
        }
      }
      adj_edges.clear();
      adj_edges.resize(0);
      // Add contents of the list to the line
      line.add_list(list, node_numbering);
      list.clear();
      list.resize(0);
    }
  }
  // Write out reorded mesh
  pumi_mesh_write(mesh,"Reordered","vtk");

  // Clean up
  pumi_numbering_delete( elem_numbering);
  pumi_numbering_delete( node_numbering);

  return;
}
```

```
int main(int argc, char** argv)
{
  if (argc != 3) {
    printf("usage: %s reorder_?.dmg reorder_?.smb\n", argv[0]);
    return 0;
  }

  MPI_Init(&argc,&argv);
  pumi_start();
  pGeom geom = pumi_geom_load(argv[1],"mesh");
  pMesh mesh = pumi_mesh_load(geom,argv[2],1);
  if(!strcmp (argv[1], "reorder_c.dmg"))
    pumi_mesh_setShape(mesh,pumi_shape_getLagrange(2));

  pMeshEnt v_start = find_start(geom, mesh);
  write_before( geom, mesh);
  reorder_mesh(geom, mesh, v_start);

  // Finish
  pumi_mesh_delete(mesh);
  pumi_finalize();
  MPI_Finalize();
  return 0;
}
```