Part II: Implementation

0.1 Introduction

The purpose of this exercise was to use the Finite Element Method (FEM) to solve the Boundary Value Problem (BVP) shown in Equations 1 to 3:

$$a(u,v) = F(v) \quad \forall v \in H_0^1(\Omega) \tag{1}$$

$$a(u,v) = \int_\Omega p(x,y)\nabla u(x,y) \cdot \nabla v(x,y) + q(x,y)u(x,y)v(x,y)dxdy \tag{2}$$

$$F(v) = \int_\Omega f(x,y)v(x,y)dxdy \tag{3}$$

where $p(x,y)$, $f(x,y)$, and $q(x,y)$ were given functions on Ω. The boundary of Ω is $\partial\Omega$ and the boundary information, $\alpha(x,y)$ on $\partial\Omega$ was also given. The solution is $u \in V = \{v \in H^1(\Omega) : v|_{\partial\Omega} = \alpha(x,y)\}$ which satisfies Eq. 1. The domain was tessellated with triangular elements; a typical element is denoted by K and the estimated domain and boundary as Ω_h and $\partial\Omega_h$, respectively. The FEM was then used to recover u_h from Eq. 4.

$$a(u_h, v) = F(v) \quad \forall v \in V_{h,0}^1 \tag{4}$$

The test space was defined as $V_{h,0}^1 = \{v \in H_0^1(\Omega) : v|_K \in P^1(K), \forall K \in \Omega_h\}$. The domain considered was $\Omega = [0,1] \times [0,1]$ with parameters $p(x,y) = 3$ and $q(x,y) = 2$. The tesselated domain Ω_h had $N+2$ nodes on each domain boundary for a total of $(N+2)^2$ nodes throughout the mesh.

0.2 Code Overview

The code followed the following major steps to construct and solve the FEM problem and solution:

1. Construct a mesh. This first involved calculating node location and then assembling the nodes into elements. Each element had counter-clockwise node ordering. Additional data, such as element areas, was constructed at this steps.

2. Assemble the linear algebra system. This was done by populating a sparse matrix (the stiffness matrix) and dense vector (the forcing vector). To populate these objects, the elemental contributions were summed into the global system. The elemental contributions of the forcing functions, $f(x,y)$, were approximated using 3$^{\text{rd}}$ order numerical quadrature.

3. Apply boundary conditions. This was done by removing all non-diagonal entries in the stiffness matrix for each row corresponding to each node on the boundary. The given boundary information, α, was then multiplied by the diagonal value which replaced that row's forcing vector term. For example, if node n at (X_n, Y_n) was on the boundary, then the nth row of the stiffness matrix was cleared except for the diagonal value, $k_{n,n}$. The nth term in the forcing vector, F_n, was then assigned $k_{n,n}\alpha(X_n, Y_n)$.

4. Solve the system. Sparse LU factorization was then used to solve the linear system.

5. Compute errors. The L^2 and H^1 errors were estimated by summing the elemental error contributions. The elemental error contributions were approximated using 3$^{\text{rd}}$ order numerical quadrature.

6. Perturb the mesh and repeat. The mesh was perturbed and the above steps from 2 to 5 was repeated. The mesh was perturbed by adjusting each node a random amount in both the x and y directions so long as the boundary was left unperturbed. The random amount had a maximum amount of $\frac{1}{20}^{\text{th}}$ of the original node spacing along each axis.

0.3 Testing and Results

Four test cases, outlined in Table 1, were considered.

Table 1: Case numbers and exact solutions.

Case Number	$u(x, y)$
1	1
2	x
3	y
4	$y^3 + sin(5(x + y)) + 2e^x$

For each case, the L^2 and H^1 norms of the error were measured. Additionally, the mesh was refined to determine the order of error convergence. For cases 1, 2, and 3 meshes of size $N + 1 = 20, 40$ were considered. For case 4, meshes of size $N + 1 = 10, 20, 40, 80, 160$ were used. The results for case 1 are shown in Tables 2 and 3. The results for case 2 are shown in Tables 4 and 5. The results for case 3 are shown in Tables 6 and 7.

As shown in Tables 2 through 7, the method used is able to approximate the solution up to machine accuracy for cases 1, 2, and 3. This is because the exact

Table 2: Errors for regular mesh for Case 1.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	4.063E-15	2.217E-14
40	2.684E-14	1.313E-13

Table 3: Errors for perturbed mesh for Case 1.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	7.780E-16	1.057E-14
40	3.401E-15	2.695E-14

Table 4: Errors for regular mesh for Case 2.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	2.196E-15	1.240E-14
40	1.197E-14	5.959E-14

Table 5: Errors for perturbed mesh for Case 2.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	9.345E-16	6.721E-14
40	9.639E-16	1.346E-14

Table 6: Errors for regular mesh for Case 3.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	2.404E-15	1.345E-14
40	1.272E-14	6.425E-14

Table 7: Errors for perturbed mesh for Case 3.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	$\|\|e_h(\cdot)\|\|_{H^1}$
20	3.927E-16	6.095E-15
40	1.415E-16	1.487E-14

solution is a member of the same space as the approximate solution. In essence, $u, u_h \in P^1(K)$ so $u_h = u$. The results for case 4 are shown in Tables 8 and 9.

As shown in Tables 8 and 9, the method is converging towards the exact so-

Table 8: Errors for regular mesh for Case 4. Assembly and Solve times in seconds.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	Order	$\|\|e_h(\cdot)\|\|_{H^1}$	Order	Assembly	Solve
10	7.136E-02	-	1.619E+00	-	2.512E-03	5.481E-05
20	1.831E-02	1.962	8.211E-01	0.979	1.658E-02	5.187E-05
40	4.607E-03	1.991	4.120E-01	0.995	9.366E-02	2.842E-04
80	1.154E-03	1.997	2.062E-01	1.000	1.957E+00	1.890E-03
160	2.885E-04	2.000	1.031E-01	1.000	3.907E+01	1.211E-02

Table 9: Errors for perturbed mesh for Case 4. Assembly and Solve times in seconds.

$N+1$	$\|\|e_h(\cdot)\|\|_{L^2(\Omega)}$	Order	$\|\|e_h(\cdot)\|\|_{H^1}$	Order	Assembly	Solve
10	7.153E-02	-	1.619E+00	-	2.561E-03	6.031E-05
20	1.841E-02	1.958	8.225E-01	0.977	4.797E-03	5.071E-05
40	4.638E-03	1.989	4.127E-01	0.995	6.699E-02	2.833E-04
80	1.162E-03	1.997	2.067E-01	0.998	1.842E+00	1.881E-03
160	2.908E-04	1.999	1.033E-01	1.000	2.904E+01	1.205E-02

lution. It converges at a rate of about 2.0 in the L^2 norm and 1.0 in the H^1 norm.

The `Eigen C++` library was used as the linear solver for this project. The native `Eigen::SparseMatrix<·>` class was used to store the stiffness vector. According to the documentation[1], accessing any given element to write or read to scales on the order of $nlog(n)$, where n is the number of non-zero entries at evaluation time. This is shown in the growth of assembly times. The `Eigen::Sparse_LU` solver was used to solve the linear system. It uses the main techniques from the sequential `SuperLU` package. The reduction of solve times from $N+1=10$ to $N+1=20$ is due to how sparse the stiffness matrix is as compared to how sparse `Eigen` assumes it will be. For both the original and perturbed meshes, overhead costs dominate the solve time when $N+1=10$. For $N+1=20$ and larger, the actual cost to solve dominates the solution time.

[1] www.eigen.tuxfamily.org

A Source Code and Headers

A.1 fea_hw5.cpp

```
#include <iostream>
#include <cstdlib>
#include "driver.hpp"

int main( int argc, char** argv)
{
  if ( argc != 3)
  {
    std::cout
      << "Usage: hw5 Case Number N+1 "
      << std::endl;

    std::abort();
  }

  int cn  = std::atoi( argv[1]);
  int np1 = std::atoi( argv[2]);

  std::cout << "cn = " << cn << std::endl;
  std::cout << "np1 = " << np1 << std::endl;

  if( cn <= 0 || cn > 4)
  {
    std::cout
      << "Unrecognized Case Number = "
      << cn
      << std::endl;

    std::abort();
  }

  seed_random();

  drive_problem( cn, np1);

  return 0;
}
```

A.2 driver.cpp

```
#include "driver.hpp"

#include <iostream>

#include "solution.hpp"
#include "mesh.hpp"

void seed_random()
{
  srand( time( NULL));
  return;
}

void drive_problem( int CaseNumber, int Np1)
{
  bool isL = false;

  mesh* m  = new mesh( Np1, isL);

  std::cout
    << std::endl
    << "Regular mesh"
    << std::endl;

  solution* s = new solution( m, CaseNumber);
  s->assemble_problem();
  s->apply_boundary_conditions();
  s->solve_system();
  s->compute_errors();
  delete s;

  mesh* pm = m->get_perturbed();

  std::cout
    << std::endl
    << "Peturbed mesh"
```

```
    << std::endl;

  solution* sp = new solution( pm, CaseNumber);
  sp->assemble_problem();
  sp->apply_boundary_conditions();
  sp->solve_system();
  sp->compute_errors();
  delete sp;

  delete pm;
  delete m;

  return;
}
```

A.3 driver.hpp

```
#ifndef DRIVER_HPP
#define DRIVER_HPP

#include "eig_wrap.hpp"

void seed_random();

void drive_problem( int CaseNumber, int Np1);

#endif
```

A.4 mesh.cpp

```
#include "mesh.hpp"

#include "eig_wrap.hpp"

#include <iostream>
#include <algorithm>
#include <time.h>
#include <cstdlib>
#include <cmath>
```

```cpp


double get_random( double min, double max)
{
  double tmp = (double) rand() / RAND_MAX;
  double ans = min + tmp * ( max - min);

  return ans;
}

mesh::mesh( int Np1_, bool isL_)
{
  isL = isL_;

  // TODO: bonus part 1
  if ( isL)
  {
    std::cout
      << "L shaped domains not supported."
      << std::endl;
    std::abort();
  }

  N    = Np1_ - 1;

  num_nodes = (N+2) * (N+2);
  node_matrix = MatrixXd::Zero( num_nodes, 2);

  create_nodes();

  num_elems = 2 * (N+1) * (N+1);

  elem_matrix = MatrixXd::Zero( num_elems, 3);

  create_elems();
  calc_areas();
  calc_side_lengths();
}

mesh::~mesh()
```

```
{
  bottom_nodes.clear();
  right_nodes.clear();
  left_nodes.clear();
  top_nodes.clear();

  elem_areas.clear();
}

mesh* mesh::get_perturbed()
{
  mesh* pm = new mesh( (N+1), isL);

  pm->perturb();

  return pm;
}

double mesh::get_elem_side_length( int elem, int side)
{
  return side_lengths( elem, side);
}

double mesh::get_pos( int elem, int i, int xy)
{
  int n = elem_matrix( elem, i);

  return node_matrix( n, xy);
}

void mesh::calc_elem_sides( int i)
{
  int n1 = elem_matrix( i, 0);
  int n2 = elem_matrix( i, 1);
  int n3 = elem_matrix( i, 2);

  double x1 = node_matrix( n1, 0);
  double y1 = node_matrix( n1, 1);

  double x2 = node_matrix( n2, 0);
```

```
  double y2 = node_matrix( n2, 1);

  double x3 = node_matrix( n3, 0);
  double y3 = node_matrix( n3, 1);

  double x12 = x1 - x2;
  double x23 = x2 - x3;
  double x13 = x1 - x3;

  double y12 = y1 - y2;
  double y23 = y2 - y3;
  double y13 = y1 - y3;

  double h1 = std::sqrt( x23 * x23 + y23 * y23);
  double h2 = std::sqrt( x13 * x13 + y13 * y13);
  double h3 = std::sqrt( x12 * x12 + y12 * x12);

  side_lengths(i, 0) = h1;
  side_lengths(i, 1) = h2;
  side_lengths(i, 2) = h3;
}

void mesh::calc_side_lengths()
{
  side_lengths = MatrixXd::Zero( num_elems, 3);
  for( int i = 0; i < num_elems; i++)
  {
    calc_elem_sides( i);
  }
  return;
}

int mesh::get_number_nodes()
{
  return num_nodes;
}

double mesh::get_elem_area( int i)
{
  return elem_areas[i];
```

```
}

int mesh::get_number_elements()
{
  return num_elems;
}

void mesh::perturb()
{
  for( int i = 0; i < num_nodes; i++)
  {
    bool bottom = false;
    bool top    = false;
    bool left   = false;
    bool right  = false;

    // Check if on bottom or top boundary
    if ( std::find( bottom_nodes.begin(), bottom_nodes.end(), i)
           != bottom_nodes.end() )
    {
      bottom = true;
    }
    else if ( std::find( top_nodes.begin(), top_nodes.end(), i)
                != top_nodes.end() )
    {
      top = true;
    }

    // Check if on left or right boundary
    if ( std::find( left_nodes.begin(), left_nodes.end(), i)
           != left_nodes.end() )
    {
      left = true;
    }
    else if ( std::find( right_nodes.begin(), right_nodes.end(), i)
                != right_nodes.end() )
```

```
    {
      right = true;
    }

    perturb_node( i, bottom, top, left, right);
  }

  // Now correct area information
  elem_areas.clear();
  calc_areas();

  return;
}

int mesh::which_boundary( int node)
{
  if ( std::find( bottom_nodes.begin(), bottom_nodes.end(), node)
      != bottom_nodes.end() )
  {
    return 1;
  }
  else if ( std::find( top_nodes.begin(), top_nodes.end(), node)
      != top_nodes.end() )
  {
    return 3;
  }

  // Check if on left or right boundary
  if ( std::find( left_nodes.begin(), left_nodes.end(), node)
      != left_nodes.end() )
  {
    return 4;
  }
  else if ( std::find( right_nodes.begin(), right_nodes.end(), node)
      != right_nodes.end() )
  {
```

```
      return 2;
   }

   return 0;
}

double mesh::get_pos( int node, int xy)
{
   return node_matrix( node, xy);
}

void mesh::perturb_node( int i,
                         bool bottom,
                         bool top,
                         bool left,
                         bool right)
{
   double max = (1.0 / 2.0) / 10.0;
   double min = -max;
   double r = get_random( min, max);

   double h = 1.0 / ( (int)N + 1.0);

   double p = r * h;
   double x = node_matrix( i, 0);
   double y = node_matrix( i, 1);

   double xnew = x;
   double ynew = y;

   if( !bottom && !top)
   {
      ynew += p;
   }

   if( !left && !right)
   {
      xnew += p;
   }

```

```
  node_matrix( i, 0) = xnew;
  node_matrix( i, 1) = ynew;

  return;
}

void mesh::calc_areas()
{
  for( int i = 0; i < num_elems; i++)
  {
    elem_areas.push_back( calc_elem_area( i));
  }

  return;
}

double mesh::calc_elem_area( int i)
{
  int n1 = elem_matrix( i, 0);
  int n2 = elem_matrix( i, 1);
  int n3 = elem_matrix( i, 2);

  double x1 = node_matrix( n1, 0);
  double x2 = node_matrix( n2, 0);
  double x3 = node_matrix( n3, 0);

  double y1 = node_matrix( n1, 1);
  double y2 = node_matrix( n2, 1);
  double y3 = node_matrix( n3, 1);

  double v1x = x2 - x1;
  double v1y = y2 - y1;

  double v2x = x3 - x1;
  double v2y = y3 - y1;

  double cross = (v1x * v2y) - (v1y * v2x);

  double area = ( 1.0 / 2.0) * cross;

```

```
  return std::abs(area);
}

void mesh::create_nodes()
{
  // x and y spacing between nodes
  double dx = 1.0 / ( (int)N + 1.0);
  double dy = 1.0 / ( (int)N + 1.0);

  double x     = 0.0;
  double y     = 0.0;
  int    index = 0;

  for( int i = 0; i < (N+2); i++)
  {

    for( int j = 0; j < (N+2); j++)
    {
      node_matrix( index, 0) = x;
      node_matrix( index, 1) = y;

      x += dx;

      check_boundary( index);

      index++;
    }
    x =  0.0;
    y += dy;

  }

  return;
}

void mesh::check_boundary( int index)
{
  bool interior = true;


```

```
  // Check top or bottom
  if ( index <= (N+1) )
  {
    bottom_nodes.push_back( index);
    interior = false;
  }
  else if ( index >= ((N+2)*(N+1) ) )
  {
    top_nodes.push_back( index);
    interior = false;
  }

  // Check left or right
  if ( index % (N+2) == 0)
  {
    left_nodes.push_back( index);
    interior = false;
  }
  else if ( index % (N+2) == (N+1) )
  {
    right_nodes.push_back( index);
    interior = false;
  }

  if( interior)
  {
    interior_nodes.push_back( index);
  }

  return;
}

void mesh::create_elems()
{
  int elem = 0;
  int n1   = 0;
  int n2   = 0;
  int n3   = 0;

  for( int n = 0; n < ( (N+2) * (N+1)); n++)
```

```
  {
    if ( n % (N+2) != (N+1) || n == 0)
    {
      n1 = n;
      n2 = n + 1;
      n3 = n + N + 3;
      create_elem_from_triple( elem, n1, n2, n3);
      elem++;

      n1 = n;
      n2 = n + N + 3;
      n3 = n + N + 2;
      create_elem_from_triple( elem, n1, n2, n3);
      elem++;
    }
  }

  return;
}

int mesh::get_global_id( int elem, int node)
{
  return elem_matrix( elem, node);
}

void mesh::create_elem_from_triple( int i, int n1, int n2↩
    , int n3)
{
  elem_matrix( i, 0) = n1;
  elem_matrix( i, 1) = n2;
  elem_matrix( i, 2) = n3;
}

int mesh::get_number_interior_nodes()
{
  return interior_nodes.size();
}

void mesh::print_mesh_stats()
{
```

```
  std::cout
    << "The number of nodes = " << num_nodes
    << std::endl;

  std::cout
    << "The number of elements = " << num_elems
    << std::endl;

  std::cout
    << "node_matrix = "
    << std::endl
    << node_matrix
    << std::endl;

  std::cout
    << "elem_matrix = "
    << std::endl
    << elem_matrix
    << std::endl;

  std::cout
    << "Nodes on bottom are: "
    << std::endl;
  for( size_t i = 0; i < bottom_nodes.size(); i++)
  {
    std::cout << bottom_nodes[i] << std::endl;
  }

  std::cout
    << "Nodes on top are: "
    << std::endl;
  for( size_t i = 0; i < top_nodes.size(); i++)
  {
    std::cout << top_nodes[i] << std::endl;
  }

  std::cout
    << "Nodes on left are: "
    << std::endl;
  for( size_t i = 0; i < left_nodes.size(); i++)
```

```
  {
    std::cout << left_nodes[i] << std::endl;
  }

  std::cout
    << "Nodes on right are: "
    << std::endl;
  for( size_t i = 0; i < right_nodes.size(); i++)
  {
    std::cout << right_nodes[i] << std::endl;
  }

  std::cout
    << "Area of elements are: "
    << std::endl;
  double checksum = 0.0;
  for( size_t i = 0; i < elem_areas.size(); i++)
  {
    double area = elem_areas[i];
    std::cout << area << std::endl;
    checksum += area;
  }

  std::cout
    << "Total area covered by the elements: "
    << checksum
    << std::endl;

  std::cout
    << "Side Lengths Matrix: "
    << std::endl
    << side_lengths
    << std::endl;

  return;
}
```

A.5 mesh.hpp

```
#ifndef MESH_HPP
```

```
#define MESH_HPP

#include "eig_wrap.hpp"

class mesh
{
  public:
    // Constructor. Creates nodes and elements
    mesh( int Np1_, bool isL_);

    // Print mesh information:
    //  - number of nodes
    //  - number of elements
    //  - location of nodes (node_matrix)
    //  - node collections of elements (elem_matrix)
    //  - nodes on each geometric edge
    void print_mesh_stats();

    // Returns a pointer to a perturbed
    //  version of this mesh.
    mesh* get_perturbed();

    // Deconstructor
    ~mesh();

    // Gets the number of interior nodes
    int get_number_interior_nodes();

    // Gets the number of total nodes
    int get_number_nodes();

    // Gets the number of elements
    int get_number_elements();

    // Gets the i'th element's area
    double get_elem_area( int i);

    double get_elem_side_length( int elem, int side);

    // Gets the x or y position of the elem's i'th node.
```

```
    //  xy = 0 for x
    //  xy = 1 for y
    double get_pos( int elem, int i, int xy);

    // Gets the x or y position of the
    //  node'th node.
    //  xy = 0 for x
    //  xy = 1 for y
    double get_pos( int node, int xy);

    // Gets the global ID for the node'th node
    //  on element number elem.
    int get_global_id( int elem, int node);

    // Returns the boundary number the node is on.
    //  0 = Interior
    //  1 = Bottom
    //  2 = Right
    //  3 = Top
    //  4 = Left
    int which_boundary( int node);

  private:
    // True if the mesh is an L shape
    bool isL;

    // The value of N (from N+1).
    int N;

    // Number of elements
    int num_elems;

    // Number of total mesh nodes
    int num_nodes;

    // The num_nodes by 2 matrix of node locations
    MatrixXd node_matrix;

    // The num_elems by 3 matrix of node locations
    MatrixXd elem_matrix;
```

```
    // The num_elems by 3 matrix of side lengths
    MatrixXd side_lengths;

    // The num_elems vector of element areas
    std::vector<double> elem_areas;

    // Calcuates the element areas,
    //  populates elem_areas
    void calc_areas();

    // Calcualtes the side lengths of each element
    //  Populates the side_lengths matrix.
    void calc_side_lengths();

    // Calculates the side lengths for a
    //  single element
    void calc_elem_sides( int i);

    // Calcuates the area of a single element.
    //  Returns the area of element i.
    double calc_elem_area( int i);

    // Creates the nodes of the mesh,
    //  popuates the node_matrix
    void create_nodes();

    // Creates the elements of the mesh,
    //  populates the elem_matrix;
    void create_elems();

    // Pointer to nodes on bottom of mesh
    std::vector<int> bottom_nodes;

    // Pointer to nodes on right of mesh
    std::vector<int> right_nodes;

    // Pointer to nodes on left of mesh
    std::vector<int> left_nodes;

```

```
    // Pointer to nodes on top of mesh
    std::vector<int> top_nodes;

    // Pointer to nodes on the interior of mesh
    std::vector<int> interior_nodes;

    // Checks where a node is and places it into the
    //  correct array
    void check_boundary( int index);

    // Creates a single element number i from
    //  a triple of nodes n1, n2, n3
    void create_elem_from_triple( int i, int n1, int n2, int n3);

    // Perturb the mesh by adjusting nodal locations
    void perturb();

    // Perturbs node i of the mesh
    void perturb_node( int i, bool bottom, bool top, bool left, bool right);
};

#endif
```

A.6 solution.cpp

```
#include "solution.hpp"

#include <iostream>
#include <ctime>
#include <time.h>

solution::solution( mesh* m_, int CaseNumber_)
{
  m  = m_;
  CaseNumber = CaseNumber_;

  p = 3.0;
  q = 2.0;
```

```

  elemental_dofs = 3;

  numNodes = m->get_number_nodes();
  numElems = m->get_number_elements();

  K = SparseMatrix<double, Eigen::RowMajor>(numNodes, numNodes);
  F = VectorXd::Zero( numNodes);
  U = VectorXd::Zero( numNodes);
}

solution::~solution()
{

}

void solution::assemble_problem()
{

  timespec ts;
  clock_gettime( CLOCK_REALTIME, &ts);

  assemble_stiffness();
  assemble_forcing();

  timespec tf;
  clock_gettime( CLOCK_REALTIME, &tf);

  double b = 1.0e9;
  double assembly_time = b * (tf.tv_sec - ts.tv_sec) + tf.tv_nsec - ts.tv_nsec;
  assembly_time /= b;

  std::cout << std::scientific
    << "Assembly time: "
    << assembly_time
    << " seconds."
    << std::endl;

```

```
  return;
}

MatrixXd solution::get_elemental_M( int i)
{
  MatrixXd mass = MatrixXd::Zero( elemental_dofs, elemental_dofs);
  double area = m->get_elem_area( i);

  // Diagonal of the 'mass' matrix
  mass( 0, 0) = 2.0;
  mass( 1, 1) = 2.0;
  mass( 2, 2) = 2.0;

  // Off-diagonal entries of 'mass' matrix
  //  upper tri
  mass( 0, 1) = 1.0;
  mass( 0, 2) = 1.0;
  mass( 1, 2) = 1.0;
  //  lower tri
  mass( 1, 0) = 1.0;
  mass( 2, 0) = 1.0;
  mass( 2, 1) = 1.0;

  mass *= ( area / 12.0);

  return mass;
}

MatrixXd solution::get_elemental_S( int i)
{
  MatrixXd spring = MatrixXd::Zero( elemental_dofs, elemental_dofs);
  double area = m->get_elem_area( i);

  double x1 = m->get_pos( i, 0, 0);
  double y1 = m->get_pos( i, 0, 1);

  double x2 = m->get_pos( i, 1, 0);
  double y2 = m->get_pos( i, 1, 1);
```

```
  double x3 = m->get_pos( i, 2, 0);
  double y3 = m->get_pos( i, 2, 1);

  double x31 = x3 - x1;
  double x12 = x1 - x2;
  double x23 = x2 - x3;

  double y31 = y3 - y1;
  double y12 = y1 - y2;
  double y23 = y2 - y3;

  // Diagonal Values of spring matrix
  spring( 0, 0) = y23 * y23 + x23 * x23;
  spring( 1, 1) = y31 * y31 + x31 * x31;
  spring( 2, 2) = y12 * y12 + x12 * x12;

  // Upper
  spring( 0, 1) = y23 * y31 + x23 * x31;
  spring( 0, 2) = y23 * y12 + x23 * x12;
  spring( 1, 2) = y31 * y12 + x31 * x12;

  // Lower
  spring( 1, 0) = y23 * y31 + x23 * x31;
  spring( 2, 0) = y23 * y12 + x23 * x12;
  spring( 2, 1) = y31 * y12 + x31 * x12;

  spring /= (4.0 * area);

  return spring;
}


MatrixXd solution::get_elemental_stiffness( int i)
{
  MatrixXd mass = get_elemental_M( i);
  MatrixXd spring = get_elemental_S( i);

  return p * spring + q * mass;
}
```

```
void solution::assign_elemental_stiffness( MatrixXd ↩
    k_elem, int i)
{
  for( int j = 0; j < elemental_dofs; j++)
  {
    for( int n = 0; n < elemental_dofs; n++)
    {
      int Erow = j;
      int Ecol = n;

      int row = m->get_global_id( i, j);
      int col = m->get_global_id( i, n);

      K.coeffRef( row, col) += k_elem( Erow, Ecol);
    }
  }
  return;
}

void solution::assemble_stiffness()
{
  for( int i = 0; i < numElems; i++)
  {
    MatrixXd k_elem = get_elemental_stiffness( i);
    assign_elemental_stiffness( k_elem, i);
  }

  return;
}

void solution::apply_boundary_conditions()
{

  for( int i = 0; i < numNodes; i++)
  {
    if( m->which_boundary( i) != 0)
    {
      double bv  = get_boundary_value( i);
      fix_global_system( bv, i);
```

```
    }
  }
  // two bugs make a feature
  K = K.transpose();
  return;
}

double solution::get_boundary_value( int i)
{
  double x = m->get_pos( i, 0);
  double y = m->get_pos( i, 1);

  double ans = 0.0;

  if( CaseNumber == 1)
  {
    ans = 1.0;
  }
  else if( CaseNumber == 2)
  {
    ans = x;
  }
  else if( CaseNumber == 3)
  {
    ans = y;
  }
  else if( CaseNumber == 4)
  {
    double a = y * y * y;
    double b = std::sin( 5.0 * (x + y));
    double c = 2.0 * std::exp( x);
    ans = a + b + c;
  }
  else
  {
    std::cout
    << "Unrecognized CaseNumber. CaseNumber = "
    << CaseNumber
    << std::endl;

```

```
    std::abort();
  }

  return ans;
}

void solution::fix_global_system( double bv, int i)
{
  // Reassign K(i,:) to all zeros except diag
  double diag = 0.0;
  for( SparseMatrix< double>::InnerIterator
        it( K, i); it; ++it)
  {
    if( it.row() == it.col() )
    {
      diag = it.value();
    }
    else
    {
      it.valueRef() = 0.0;
    }
  }

  F( i) = bv * diag;

  return;
}

void solution::assemble_forcing()
{
  for( int i = 0; i < numElems; i++)
  {
    VectorXd f_elem = get_elemental_forcing( i);
    assign_elemental_forcing( f_elem, i);
  }

  return;
}

double average( double a, double b)
```

```
{
  double ans = (a + b) / 2.0;
  return ans;
}

double average( double a, double b, double c)
{
  double ans = ( a + b + c) / 3.0;
  return ans;
}

VectorXd solution::get_elemental_forcing( int elem_num)
{
  VectorXd f_elem = VectorXd::Zero( elemental_dofs);

  double area = m->get_elem_area( elem_num);

  double x1 = m->get_pos( elem_num, 0, 0);
  double y1 = m->get_pos( elem_num, 0, 1);

  double x2 = m->get_pos( elem_num, 1, 0);
  double y2 = m->get_pos( elem_num, 1, 1);

  double x3 = m->get_pos( elem_num, 2, 0);
  double y3 = m->get_pos( elem_num, 2, 1);

  double x12 = average( x1, x2);
  double x13 = average( x1, x3);
  double x23 = average( x2, x3);

  double y12 = average( y1, y2);
  double y13 = average( y1, y3);
  double y23 = average( y2, y3);

  double x123 = average( x1, x2, x3);
  double y123 = average( y1, y2, y3);

  double f1 = force_at_point( x1, y1);
  double f2 = force_at_point( x2, y2);
  double f3 = force_at_point( x3, y3);
```

```
  double f12 = force_at_point( x12, y12);
  double f13 = force_at_point( x13, y13);
  double f23 = force_at_point( x23, y23);

  double f123 = force_at_point( x123, y123);

  double tmp1 = 0.0;
  double tmp2 = 0.0;
  double tmp3 = 0.0;

  tmp1 += 3.0 * f1;
  tmp2 += 3.0 * f2;
  tmp3 += 3.0 * f3;

  tmp1 += 4.0 * ( f12 + f13);
  tmp2 += 4.0 * ( f12 + f23);
  tmp3 += 4.0 * ( f13 + f23);

  tmp1 += 9.0 * f123;
  tmp2 += 9.0 * f123;
  tmp3 += 9.0 * f123;

  f_elem(0) = tmp1;
  f_elem(1) = tmp2;
  f_elem(2) = tmp3;

  f_elem *= area / 60.0;

  return f_elem;
}

void solution::assign_elemental_forcing( VectorXd f_elem, int elem)
{
  for( int i = 0; i < elemental_dofs; i++)
  {
    int row  = m->get_global_id( elem, i);
    F( row) += f_elem( i);
  }
```

```
  return;
}

double solution::force_at_point( double x, double y)
{
  double ans = 0.0;
  if( CaseNumber == 1)
  {
    ans = 2.0;
  }
  else if( CaseNumber == 2)
  {
    ans = 2.0 * x;
  }
  else if( CaseNumber == 3)
  {
    ans = 2.0 * y;
  }
  else if( CaseNumber == 4)
  {
    double a =  2.0 * y * y * y;
    double b = -18.0 * y;
    double c =  152.0 * std::sin( 5.0 * ( x + y));
    double d = -2.0 * std::exp( x);

    ans = a + b + c + d;
  }
  else
  {
    std::cout
    << "Unrecognized CaseNumber. CaseNumber = "
    << CaseNumber
    << std::endl;

    std::abort();
  }

  return ans;
}

```

```
void solution::solve_system()
{
  K.makeCompressed();
  Eigen::SparseLU< SparseMatrix< double> > solver;

  solver.analyzePattern( K);

  solver.factorize( K);


  timespec ts;
  clock_gettime( CLOCK_REALTIME, &ts);

  U = solver.solve( F);

  timespec tf;
  clock_gettime( CLOCK_REALTIME, &tf);

  double b = 1.0e9;
  double solve_time = b * (tf.tv_sec - ts.tv_sec) + tf.tv_nsec - ts.tv_nsec;
  solve_time /= b;

  std::cout << std::scientific
    << "Solve time: "
    << solve_time
    << " seconds."
    << std::endl;

  return;

}

void solution::compute_errors()
{
  compute_L2_error();
  compute_H1_error();

  return;
}
```

```
void solution::compute_L2_error()
{
  double error = 0.0;
  for( int elem = 0; elem < numElems; elem++)
  {
    error += get_elemental_error_L2( elem);
  }

  L2_error = error;
  L2_error = std::sqrt( L2_error);

  error = std::sqrt( error);
  std::cout << std::scientific
    << "L2 Error = "
    << L2_error
    << std::endl;

  return;
}

double solution::get_elemental_error_L2( int elem)
{
  double area = m->get_elem_area( elem);

  double x1 = m->get_pos( elem, 0, 0);
  double y1 = m->get_pos( elem, 0, 1);

  double x2 = m->get_pos( elem, 1, 0);
  double y2 = m->get_pos( elem, 1, 1);

  double x3 = m->get_pos( elem, 2, 0);
  double y3 = m->get_pos( elem, 2, 1);

  double x12 = average( x1, x2);
  double x13 = average( x1, x3);
  double x23 = average( x2, x3);

  double y12 = average( y1, y2);
  double y13 = average( y1, y3);
```

```
  double y23 = average( y2, y3);

  double x123 = average( x1, x2, x3);
  double y123 = average( y1, y2, y3);

  double u1 = get_exact_solution( x1, y1);
  double u2 = get_exact_solution( x2, y2);
  double u3 = get_exact_solution( x3, y3);

  double u12 = get_exact_solution( x12, y12);
  double u13 = get_exact_solution( x13, y13);
  double u23 = get_exact_solution( x23, y23);

  double u123 = get_exact_solution( x123, y123);

  double uh1 = U( m->get_global_id( elem, 0));
  double uh2 = U( m->get_global_id( elem, 1));
  double uh3 = U( m->get_global_id( elem, 2));

  double uh12 = average( uh1, uh2);
  double uh13 = average( uh1, uh3);
  double uh23 = average( uh2, uh3);

  double uh123 = average( uh1, uh2, uh3);

  double e1 = (u1 - uh1) * (u1 - uh1);
  double e2 = (u2 - uh2) * (u2 - uh2);
  double e3 = (u3 - uh3) * (u3 - uh3);

  double e12 = (u12 - uh12) * (u12 - uh12);
  double e23 = (u23 - uh23) * (u23 - uh23);
  double e13 = (u13 - uh13) * (u13 - uh13);

  double e123 = (u123 - uh123) * (u123 - uh123);

  double tmp1 = e1  + e2  + e3;
  double tmp2 = e12 + e13 + e23;
  double tmp3 = e123;

  double error = 3.0 * tmp1 + 8.0 * tmp2 + 27 * tmp3;
```

```

  error *= area / 60.0;

  return error;
}

void solution::compute_H1_error()
{
  double error = 0.0;
  for( int elem = 0; elem < numElems; elem++)
  {
    error += get_elemental_error_H1( elem);
  }

  H1_error = error + L2_error * L2_error;
  H1_error = std::sqrt( H1_error);

  std::cout << std::scientific
    << "H1 Error = "
    << H1_error
    << std::endl;
  return;
}

double solution::get_elemental_error_H1( int elem)
{
  double area = m->get_elem_area( elem);

  double x1 = m->get_pos( elem, 0, 0);
  double y1 = m->get_pos( elem, 0, 1);

  double x2 = m->get_pos( elem, 1, 0);
  double y2 = m->get_pos( elem, 1, 1);

  double x3 = m->get_pos( elem, 2, 0);
  double y3 = m->get_pos( elem, 2, 1);

  double x12 = average( x1, x2);
  double x13 = average( x1, x3);
  double x23 = average( x2, x3);
```

```

    double y12 = average( y1, y2);
    double y13 = average( y1, y3);
    double y23 = average( y2, y3);

    double x123 = average( x1, x2, x3);
    double y123 = average( y1, y2, y3);

    double ux1 = get_exact_solution_grad( x1, y1, 0);
    double ux2 = get_exact_solution_grad( x2, y2, 0);
    double ux3 = get_exact_solution_grad( x3, y3, 0);
    double uy1 = get_exact_solution_grad( x1, y1, 1);
    double uy2 = get_exact_solution_grad( x2, y2, 1);
    double uy3 = get_exact_solution_grad( x3, y3, 1);

    double ux12 = get_exact_solution_grad( x12, y12, 0);
    double ux23 = get_exact_solution_grad( x23, y23, 0);
    double ux13 = get_exact_solution_grad( x13, y13, 0);
    double uy12 = get_exact_solution_grad( x12, y12, 1);
    double uy13 = get_exact_solution_grad( x13, y13, 1);
    double uy23 = get_exact_solution_grad( x23, y23, 1);

    double ux123 = get_exact_solution_grad( x123, y123, 0);
    double uy123 = get_exact_solution_grad( x123, y123, 1);

    double gx1 = (y2 - y3) / (2.0 * area);
    double gy1 = (x3 - x2) / (2.0 * area);

    double gx2 = (y3 - y1) / (2.0 * area);
    double gy2 = (x1 - x3) / (2.0 * area);

    double gx3 = (y1 - y2) / (2.0 * area);
    double gy3 = (x2 - x1) / (2.0 * area);

    double uh1 = U( m->get_global_id( elem, 0));
    double uh2 = U( m->get_global_id( elem, 1));
    double uh3 = U( m->get_global_id( elem, 2));

    double uhx1 = uh1 * gx1;
    double uhx2 = uh2 * gx2;
```

```
    double uhx3 = uh3 * gx3;
    double uhy1 = uh1 * gy1;
    double uhy2 = uh2 * gy2;
    double uhy3 = uh3 * gy3;

    double guhx = uhx1 + uhx2 + uhx3;
    double guhy = uhy1 + uhy2 + uhy3;

    double ex1  = (guhx - ux1) * (guhx - ux1);
    double ey1  = (guhy - uy1) * (guhy - uy1);
    double ex2  = (guhx - ux2) * (guhx - ux2);
    double ey2  = (guhy - uy2) * (guhy - uy2);
    double ex3  = (guhx - ux3) * (guhx - ux3);
    double ey3  = (guhy - uy3) * (guhy - uy3);

    double tmp1 = ex1 + ey1 + ex2 + ey2 + ex3 + ey3;

    double ex12  = (guhx - ux12) * (guhx - ux12);
    double ey12  = (guhy - uy12) * (guhy - uy12);
    double ex23  = (guhx - ux23) * (guhx - ux23);
    double ey23  = (guhy - uy23) * (guhy - uy23);
    double ex13  = (guhx - ux13) * (guhx - ux13);
    double ey13  = (guhy - uy13) * (guhy - uy13);

    double tmp2 = ex12 + ey12 + ex13 + ey13 + ex23 + ey23;

    double ex123  = (guhx - ux123) * (guhx - ux123);
    double ey123  = (guhy - uy123) * (guhy - uy123);

    double tmp3 = ex123 + ey123;

    double error = 3.0 * tmp1 + 8.0 * tmp2 + 27 * tmp3;

    error *= area / 60.0;

    return error;
}

double solution::get_exact_solution( double x, double y)
{
```

```
  double u = 0.0;

  if( CaseNumber == 1)
  {
    u = 1.0;
  }
  else if( CaseNumber == 2)
  {
    u = x;
  }
  else if( CaseNumber == 3)
  {
    u = y;
  }
  else if( CaseNumber == 4)
  {
    double a = y * y * y;
    double b = std::sin( 5.0 * (x + y));
    double c = 2.0 * std::exp( x);
    u = a + b + c;
  }
  return u;
}

double solution::get_exact_solution_grad( double x, double y, int xy)
{
  double du = 0.0;

  if( xy == 0)
  {
    if( CaseNumber == 1)
    {
      du = 0.0;
    }
    else if( CaseNumber == 2)
    {
      du = 1.0;
    }
    else if( CaseNumber == 3)
```

```
      {
        du = 0.0;
      }
      else if( CaseNumber == 4)
      {
        double a = 0.0;
        double b = 5.0 * std::cos( 5.0 * (x + y));
        double c = 2.0 * std::exp( x);
        du = a + b + c;
      }
    }
    else if ( xy == 1)
    {
      if( CaseNumber == 1)
      {
        du = 0.0;
      }
      else if( CaseNumber == 2)
      {
        du = 0.0;
      }
      else if( CaseNumber == 3)
      {
        du = 1.0;
      }
      else if( CaseNumber == 4)
      {
        double a = 3.0 * y * y;
        double b = 5.0 * std::cos( 5.0 * (x + y));
        double c = 0.0;
        du = a + b + c;
      }
    }
    else
    {
      std::cout
        << "Unrecognized xy option: "
        << xy
        << std::endl;

```

```
    std::abort();
  }

  return du;
}
```

A.7 solution.hpp

```
#ifndef SOLUTION_HPP
#define SOLUTION_HPP

#include "mesh.hpp"

class solution
{
  public:
    // Constructor
    solution( mesh* m_, int CaseNumber_);

    // Assembles the stiffness and forcing functions
    void assemble_problem();

    // Apply the Dirichlet boundary conditions
    //  for the specified case number.
    void apply_boundary_conditions();

    // Solves the system KU=F for U
    void solve_system();

    // Compute the errors
    void compute_errors();

    // Destructor
    ~solution();

  private:
    // Pointer to the mesh
    mesh* m;

    // PDE Parameter "p"
```

```
    double p;

    // PDE Parameter "q"
    double q;

    // Number of mesh nodes
    int numNodes;

    // Number of elements in the mesh
    int numElems;

    // Number of degrees of freedom per element
    int elemental_dofs;

    // CaseNumber
    int CaseNumber;

    // Global Reduced Stiffness Matrix
    SparseMatrix<double> K;

    // Global Reduced Forcing Vector
    VectorXd F;

    // Solution Vector
    VectorXd U;

    // Assembles only the stiffness matrix
    void assemble_stiffness();

    // Assembles only the forcing matrix
    void assemble_forcing();

    // Returns the stiffness matrix for element i
    MatrixXd get_elemental_stiffness( int i);

    // Assigns the i'th elemental stiffness k_elem
    //  to the global stiffness matrix
    void assign_elemental_stiffness( MatrixXd k_elem, int i);

```

```
    // Gets the elemental pure mass matrix
    MatrixXd get_elemental_M( int i);

    // Gets the elemental pure spring matrix
    MatrixXd get_elemental_S( int i);

    // Gets the elemental forcing function.
    //  Body forces only.
    VectorXd get_elemental_forcing( int elem_num);

    // Assigns the elemental forcing vector to the global one.
    void assign_elemental_forcing( VectorXd f_elem, int elem);

    // Gets the force at a point based on the
    //  case nuber defined at construction time.
    double force_at_point( double x, double y);

    // Adjusts the global system for boundary value bv
    //  at node i.
    void fix_global_system( double bv, int i);

    // Gets the DBC boundary value for node number i
    //  based on Case number defined at construction.
    double get_boundary_value( int i);

    // Computes the L2 error.
    void compute_L2_error();

    // Computes the H1 error.
    void compute_H1_error();

    // Gets the exact solution based on the Case
    //  number provided at construction time.
    double get_exact_solution( double x, double y);

    // Gets the L2 error for element elem.
    double get_elemental_error_L2( int elem);

```

```
    // Gets the H1 error for element elem.
    double get_elemental_error_H1(int elem);

    // Gets the exact gradient of the solution.
    double get_exact_solution_grad(double x, double y, int xy);

    // The value of the H1 norm
    double H1_error;

    // The value of the L2 norm
    double L2_error;
};

#endif
```

A.8 CMakeLists.txt

```
set( HW5_SOURCES
coord.cpp
coord.hpp
driver.cpp
driver.hpp
mesh.cpp
mesh.hpp
element.cpp
element.hpp
vertex.cpp
vertex.hpp
solution.cpp
solution.hpp
eig_wrap.hpp
fea_hw5.cpp)

add_executable(hw5 ${HW5_SOURCES})

target_include_directories(hw5 PUBLIC ${EIG_DIR})
```

A.9 eig_wrap.hpp

```
#ifndef EIG_WRAP
#define EIG_WRAP

// eig_wrap.hpp
// This wraps all needed Eigen package
// headers into a "convenient" header.
// It also defines using directives.

#include </lore/clougj/Learning_Codes/FEA/Eigen_Package/Eigen/Eigen>

using Eigen::MatrixXd;
using Eigen::VectorXd;
using Eigen::SparseMatrix;

#endif
```