

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:

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

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

$$F(v) = \int_{\Omega} f(x, y) v(x, y) dx dy \quad (3)$$

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.

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

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 tessellated 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 3rd 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 n th row of the stiffness matrix was cleared except for the diagonal value, $k_{n,n}$. The n th 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 3rd 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 $\frac{1}{20}$ 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 documentation¹, accessing any given element to write or read to scales on the order of $n \log(n)$, 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.

¹www.eigen.tuxfamily.org

A Source Code and Headers

A.1 fea_hw5.cpp

```
1 #include <iostream>
2 #include <cstdlib>
3 #include "driver.hpp"
4
5 int main( int argc , char** argv)
6 {
7     if ( argc != 3)
8     {
9         std::cout
10             << "Usage: _hw5_Case_Number_N+1_"
11             << std::endl;
12
13         std::abort();
14     }
15
16     int cn  = std::atoi( argv[1]);
17     int np1 = std::atoi( argv[2]);
18
19     std::cout << "cn=_ " << cn << std::endl;
20     std::cout << "np1=_ " << np1 << std::endl;
21
22     if( cn <= 0 || cn > 4)
23     {
24         std::cout
25             << "Unrecognized _Case_Number=_ "
26             << cn
27             << std::endl;
28
29         std::abort();
30     }
31
32     seed_random();
33
34     drive_problem( cn , np1);
35
36     return 0;
37 }
```

A.2 driver.cpp

```
1 #include "driver.hpp"
2
3 #include <iostream>
4
5 #include "solution.hpp"
6 #include "mesh.hpp"
7
8 void seed_random()
9 {
10     srand( time( NULL) );
11     return;
12 }
13
14 void drive_problem( int CaseNumber, int Np1)
15 {
16     bool isL = false;
17
18     mesh* m = new mesh( Np1, isL );
19
20     std::cout
21         << std::endl
22         << "Regular_mesh"
23         << std::endl;
24
25     solution* s = new solution( m, CaseNumber );
26     s->assemble_problem();
27     s->apply_boundary_conditions();
28     s->solve_system();
29     s->compute_errors();
30     delete s;
31
32     mesh* pm = m->get_perturbed();
33
34     std::cout
35         << std::endl
36         << "Peturbed_mesh"
```

```

37     << std::endl;
38
39     solution* sp = new solution( pm, CaseNumber);
40     sp->assemble_problem();
41     sp->apply_boundary_conditions();
42     sp->solve_system();
43     sp->compute_errors();
44     delete sp;
45
46     delete pm;
47     delete m;
48
49     return;
50 }

```

A.3 driver.hpp

```

1 #ifndef DRIVER_HPP
2 #define DRIVER_HPP
3
4 #include "eig_wrap.hpp"
5
6 void seed_random();
7
8 void drive_problem( int CaseNumber, int Np1);
9
10 #endif

```

A.4 mesh.cpp

```

1 #include "mesh.hpp"
2
3 #include "eig_wrap.hpp"
4
5 #include <iostream>
6 #include <algorithm>
7 #include <time.h>
8 #include <cstdlib>
9 #include <cmath>

```

```

10
11
12 double get_random( double min, double max)
13 {
14     double tmp = (double) rand() / RAND_MAX;
15     double ans = min + tmp * ( max - min);
16
17     return ans;
18 }
19
20 mesh::mesh( int Np1_, bool isL_)
21 {
22     isL = isL_;
23
24     // TODO: bonus part 1
25     if ( isL)
26     {
27         std::cout
28             << "Lshaped_domains_not_supported."
29             << std::endl;
30         std::abort();
31     }
32
33     N    = Np1_ - 1;
34
35     num_nodes = (N+2) * (N+2);
36     node_matrix = MatrixXd::Zero( num_nodes, 2);
37
38     create_nodes();
39
40     num_elems = 2 * (N+1) * (N+1);
41
42     elem_matrix = MatrixXd::Zero( num_elems, 3);
43
44     create_elems();
45     calc_areas();
46     calc_side_lengths();
47 }
48
49 mesh::~~mesh()

```



```

50 {
51     bottom_nodes.clear();
52     right_nodes.clear();
53     left_nodes.clear();
54     top_nodes.clear();
55
56     elem_areas.clear();
57 }
58
59 mesh* mesh::get_perturbed()
60 {
61     mesh* pm = new mesh( (N+1), isL);
62
63     pm->perturb();
64
65     return pm;
66 }
67
68 double mesh::get_elem_side_length( int elem, int side)
69 {
70     return side_lengths( elem, side);
71 }
72
73 double mesh::get_pos( int elem, int i, int xy)
74 {
75     int n = elem_matrix( elem, i);
76
77     return node_matrix( n, xy);
78 }
79
80 void mesh::calc_elem_sides( int i)
81 {
82     int n1 = elem_matrix( i, 0);
83     int n2 = elem_matrix( i, 1);
84     int n3 = elem_matrix( i, 2);
85
86     double x1 = node_matrix( n1, 0);
87     double y1 = node_matrix( n1, 1);
88
89     double x2 = node_matrix( n2, 0);

```

```

90  double y2 = node_matrix( n2, 1);
91
92  double x3 = node_matrix( n3, 0);
93  double y3 = node_matrix( n3, 1);
94
95  double x12 = x1 - x2;
96  double x23 = x2 - x3;
97  double x13 = x1 - x3;
98
99  double y12 = y1 - y2;
100 double y23 = y2 - y3;
101 double y13 = y1 - y3;
102
103 double h1 = std::sqrt( x23 * x23 + y23 * y23);
104 double h2 = std::sqrt( x13 * x13 + y13 * y13);
105 double h3 = std::sqrt( x12 * x12 + y12 * y12);
106
107 side_lengths(i, 0) = h1;
108 side_lengths(i, 1) = h2;
109 side_lengths(i, 2) = h3;
110 }
111
112 void mesh::calc_side_lengths()
113 {
114     side_lengths = MatrixXd::Zero( num_elems, 3);
115     for( int i = 0; i < num_elems; i++)
116     {
117         calc_elem_sides( i);
118     }
119     return;
120 }
121
122 int mesh::get_number_nodes()
123 {
124     return num_nodes;
125 }
126
127 double mesh::get_elem_area( int i)
128 {
129     return elem_areas[i];

```

```

130 }
131
132 int mesh::get_number_elements()
133 {
134     return num_elems;
135 }
136
137 void mesh::perturb()
138 {
139     for( int i = 0; i < num_nodes; i++)
140     {
141         bool bottom = false;
142         bool top    = false;
143         bool left   = false;
144         bool right  = false;
145
146         // Check if on bottom or top boundary
147         if ( std::find( bottom_nodes.begin(), bottom_nodes.↵
148             end(), i)
149             != bottom_nodes.end() )
150         {
151             bottom = true;
152         }
153         else if ( std::find( top_nodes.begin(), top_nodes.end(↵
154             ), i)
155             != top_nodes.end() )
156         {
157             top = true;
158         }
159         // Check if on left or right boundary
160         if ( std::find( left_nodes.begin(), left_nodes.end(), ↵
161             i)
162             != left_nodes.end() )
163         {
164             left = true;
165         }
166         else if ( std::find( right_nodes.begin(), right_nodes.↵
167             .end(), i)
168             != right_nodes.end() )

```

```

166     {
167         right = true;
168     }
169
170     perturb_node( i, bottom, top, left , right);
171 }
172
173     // Now correct area information
174     elem_areas.clear();
175     calc_areas();
176
177     return;
178 }
179
180 int mesh::which_boundary( int node)
181 {
182     if ( std::find( bottom_nodes.begin() , bottom_nodes.end()↵
183         ( ) , node)
184         != bottom_nodes.end() )
185     {
186         return 1;
187     }
188     else if ( std::find( top_nodes.begin() , top_nodes.end()↵
189         , node)
190         != top_nodes.end() )
191     {
192         return 3;
193     }
194
195     // Check if on left or right boundary
196     if ( std::find( left_nodes.begin() , left_nodes.end() , ↵
197         node)
198         != left_nodes.end() )
199     {
200         return 4;
201     }
202     else if ( std::find( right_nodes.begin() , right_nodes.↵
203         end() , node)
204         != right_nodes.end() )
205     {

```

```

202     return 2;
203 }
204
205     return 0;
206 }
207
208 double mesh::get_pos( int node, int xy)
209 {
210     return node_matrix( node, xy);
211 }
212
213 void mesh::perturb_node( int i,
214                          bool bottom,
215                          bool top,
216                          bool left,
217                          bool right)
218 {
219     double max = (1.0 / 2.0) / 10.0;
220     double min = -max;
221     double r = get_random( min, max);
222
223     double h = 1.0 / ( (int)N + 1.0);
224
225     double p = r * h;
226     double x = node_matrix( i, 0);
227     double y = node_matrix( i, 1);
228
229     double xnew = x;
230     double ynew = y;
231
232     if( !bottom && !top)
233     {
234         ynew += p;
235     }
236
237     if( !left && !right)
238     {
239         xnew += p;
240     }
241

```

```

242     node_matrix( i, 0) = xnew;
243     node_matrix( i, 1) = ynew;
244
245     return;
246 }
247
248 void mesh::calc_areas()
249 {
250     for( int i = 0; i < num_elems; i++)
251     {
252         elem_areas.push_back( calc_elem_area( i));
253     }
254
255     return;
256 }
257
258 double mesh::calc_elem_area( int i)
259 {
260     int n1 = elem_matrix( i, 0);
261     int n2 = elem_matrix( i, 1);
262     int n3 = elem_matrix( i, 2);
263
264     double x1 = node_matrix( n1, 0);
265     double x2 = node_matrix( n2, 0);
266     double x3 = node_matrix( n3, 0);
267
268     double y1 = node_matrix( n1, 1);
269     double y2 = node_matrix( n2, 1);
270     double y3 = node_matrix( n3, 1);
271
272     double v1x = x2 - x1;
273     double v1y = y2 - y1;
274
275     double v2x = x3 - x1;
276     double v2y = y3 - y1;
277
278     double cross = (v1x * v2y) - (v1y * v2x);
279
280     double area = ( 1.0 / 2.0) * cross;
281

```

```

282 |     return std::abs(area);
283 | }
284 |
285 | void mesh::create_nodes()
286 | {
287 |     // x and y spacing between nodes
288 |     double dx = 1.0 / ( (int)N + 1.0);
289 |     double dy = 1.0 / ( (int)N + 1.0);
290 |
291 |     double x      = 0.0;
292 |     double y      = 0.0;
293 |     int      index = 0;
294 |
295 |     for( int i = 0; i < (N+2); i++)
296 |     {
297 |
298 |         for( int j = 0; j < (N+2); j++)
299 |         {
300 |             node_matrix( index , 0) = x;
301 |             node_matrix( index , 1) = y;
302 |
303 |             x += dx;
304 |
305 |             check_boundary( index);
306 |
307 |             index++;
308 |         }
309 |         x = 0.0;
310 |         y += dy;
311 |
312 |     }
313 |
314 |     return;
315 | }
316 |
317 | void mesh::check_boundary( int index)
318 | {
319 |     bool interior = true;
320 |
321 |

```

```

322 // Check top or bottom
323 if ( index <= (N+1) )
324 {
325     bottom_nodes.push_back( index);
326     interior = false;
327 }
328 else if ( index >= ((N+2)*(N+1) ) )
329 {
330     top_nodes.push_back( index);
331     interior = false;
332 }
333
334 // Check left or right
335 if ( index % (N+2) == 0)
336 {
337     left_nodes.push_back( index);
338     interior = false;
339 }
340 else if ( index % (N+2) == (N+1) )
341 {
342     right_nodes.push_back( index);
343     interior = false;
344 }
345
346 if( interior)
347 {
348     interior_nodes.push_back( index);
349 }
350
351 return;
352 }
353
354 void mesh::create_elems()
355 {
356     int elem = 0;
357     int n1    = 0;
358     int n2    = 0;
359     int n3    = 0;
360
361     for( int n = 0; n < ( (N+2) * (N+1)); n++)

```



```

362 {
363     if ( n % (N+2) != (N+1) || n == 0)
364     {
365         n1 = n;
366         n2 = n + 1;
367         n3 = n + N + 3;
368         create_elem_from_triple( elem, n1, n2, n3);
369         elem++;
370
371         n1 = n;
372         n2 = n + N + 3;
373         n3 = n + N + 2;
374         create_elem_from_triple( elem, n1, n2, n3);
375         elem++;
376     }
377 }
378
379 return;
380 }
381
382 int mesh::get_global_id( int elem, int node)
383 {
384     return elem_matrix( elem, node);
385 }
386
387 void mesh::create_elem_from_triple( int i, int n1, int n2←
    , int n3)
388 {
389     elem_matrix( i, 0) = n1;
390     elem_matrix( i, 1) = n2;
391     elem_matrix( i, 2) = n3;
392 }
393
394 int mesh::get_number_interior_nodes()
395 {
396     return interior_nodes.size();
397 }
398
399 void mesh::print_mesh_stats()
400 {

```

```

401 std::cout
402     << "The number of nodes = " << num_nodes
403     << std::endl;
404
405 std::cout
406     << "The number of elements = " << num_elems
407     << std::endl;
408
409 std::cout
410     << "node_matrix = "
411     << std::endl
412     << node_matrix
413     << std::endl;
414
415 std::cout
416     << "elem_matrix = "
417     << std::endl
418     << elem_matrix
419     << std::endl;
420
421 std::cout
422     << "Nodes on bottom are: "
423     << std::endl;
424 for( size_t i = 0; i < bottom_nodes.size(); i++)
425 {
426     std::cout << bottom_nodes[i] << std::endl;
427 }
428
429 std::cout
430     << "Nodes on top are: "
431     << std::endl;
432 for( size_t i = 0; i < top_nodes.size(); i++)
433 {
434     std::cout << top_nodes[i] << std::endl;
435 }
436
437 std::cout
438     << "Nodes on left are: "
439     << std::endl;
440 for( size_t i = 0; i < left_nodes.size(); i++)

```

```

441     {
442         std::cout << left_nodes[i] << std::endl;
443     }
444
445     std::cout
446         << "Nodes on right are: "
447         << std::endl;
448     for( size_t i = 0; i < right_nodes.size(); i++)
449     {
450         std::cout << right_nodes[i] << std::endl;
451     }
452
453     std::cout
454         << "Area of elements are: "
455         << std::endl;
456     double checksum = 0.0;
457     for( size_t i = 0; i < elem_areas.size(); i++)
458     {
459         double area = elem_areas[i];
460         std::cout << area << std::endl;
461         checksum += area;
462     }
463
464     std::cout
465         << "Total area covered by the elements: "
466         << checksum
467         << std::endl;
468
469     std::cout
470         << "Side Lengths Matrix: "
471         << std::endl
472         << side_lengths
473         << std::endl;
474
475     return;
476 }

```

A.5 mesh.hpp

```

1 #ifndef MESH_HPP

```

```

2 #define MESH_HPP
3
4 #include "eig_wrap.hpp"
5
6 class mesh
7 {
8     public:
9         // Constructor. Creates nodes and elements
10        mesh( int Np1_, bool isL_);
11
12        // Print mesh information:
13        // - number of nodes
14        // - number of elements
15        // - location of nodes (node_matrix)
16        // - node collections of elements (elem_matrix)
17        // - nodes on each geometric edge
18        void print_mesh_stats();
19
20        // Returns a pointer to a perturbed
21        // version of this mesh.
22        mesh* get_perturbed();
23
24        // Deconstructor
25        ~mesh();
26
27        // Gets the number of interior nodes
28        int get_number_interior_nodes();
29
30        // Gets the number of total nodes
31        int get_number_nodes();
32
33        // Gets the number of elements
34        int get_number_elements();
35
36        // Gets the i'th element's area
37        double get_elem_area( int i);
38
39        double get_elem_side_length( int elem, int side);
40
41        // Gets the x or y position of the elem's i'th node.

```

```

42 // xy = 0 for x
43 // xy = 1 for y
44 double get_pos( int elem, int i, int xy);
45
46 // Gets the x or y position of the
47 // node'th node.
48 // xy = 0 for x
49 // xy = 1 for y
50 double get_pos( int node, int xy);
51
52 // Gets the global ID for the node'th node
53 // on element number elem.
54 int get_global_id( int elem, int node);
55
56 // Returns the boundary number the node is on.
57 // 0 = Interior
58 // 1 = Bottom
59 // 2 = Right
60 // 3 = Top
61 // 4 = Left
62 int which_boundary( int node);
63
64 private:
65 // True if the mesh is an L shape
66 bool isL;
67
68 // The value of N (from N+1).
69 int N;
70
71 // Number of elements
72 int num_elems;
73
74 // Number of total mesh nodes
75 int num_nodes;
76
77 // The num_nodes by 2 matrix of node locations
78 MatrixXd node_matrix;
79
80 // The num_elems by 3 matrix of node locations
81 MatrixXd elem_matrix;

```

```

82
83 // The num_elems by 3 matrix of side lengths
84 MatrixXd side_lengths;
85
86 // The num_elems vector of element areas
87 std::vector<double> elem_areas;
88
89 // Calcuates the element areas,
90 // populates elem_areas
91 void calc_areas();
92
93 // Calcualtes the side lengths of each element
94 // Populates the side_lengths matrix.
95 void calc_side_lengths();
96
97 // Calculates the side lengths for a
98 // single element
99 void calc_elem_sides( int i);
100
101 // Calcuates the area of a single element.
102 // Returns the area of element i.
103 double calc_elem_area( int i);
104
105 // Creates the nodes of the mesh,
106 // popuates the node_matrix
107 void create_nodes();
108
109 // Creates the elements of the mesh,
110 // populates the elem_matrix;
111 void create_elems();
112
113 // Pointer to nodes on bottom of mesh
114 std::vector<int> bottom_nodes;
115
116 // Pointer to nodes on right of mesh
117 std::vector<int> right_nodes;
118
119 // Pointer to nodes on left of mesh
120 std::vector<int> left_nodes;
121

```

```

122 // Pointer to nodes on top of mesh
123 std::vector<int> top_nodes;
124
125 // Pointer to nodes on the interior of mesh
126 std::vector<int> interior_nodes;
127
128 // Checks where a node is and places it into the
129 // correct array
130 void check_boundary( int index);
131
132 // Creates a single element number i from
133 // a triple of nodes n1, n2, n3
134 void create_elem_from_triple( int i, int n1, int n2, ←
    int n3);
135
136 // Perturb the mesh by adjusting nodal locations
137 void perturb();
138
139 // Perturbs node i of the mesh
140 void perturb_node( int i, bool bottom, bool top, bool ←
    left, bool right);
141 };
142
143 #endif

```

A.6 solution.cpp

```

1 #include "solution.hpp"
2
3 #include <iostream>
4 #include <ctime>
5 #include <time.h>
6
7 solution::solution( mesh* m_, int CaseNumber_)
8 {
9     m = m_;
10     CaseNumber = CaseNumber_;
11
12     p = 3.0;
13     q = 2.0;

```

```

14
15     elemental_dofs = 3;
16
17     numNodes = m->get_number_nodes();
18     numElems = m->get_number_elements();
19
20     K = SparseMatrix<double, Eigen::RowMajor>(numNodes, ←
        numNodes);
21     F = VectorXd::Zero( numNodes);
22     U = VectorXd::Zero( numNodes);
23 }
24
25 solution::~~solution()
26 {
27
28 }
29
30 void solution::assemble_problem()
31 {
32
33     timespec ts;
34     clock_gettime( CLOCK_REALTIME, &ts);
35
36     assemble_stiffness();
37     assemble_forcing();
38
39     timespec tf;
40     clock_gettime( CLOCK_REALTIME, &tf);
41
42     double b = 1.0e9;
43     double assembly_time = b * (tf.tv_sec - ts.tv_sec) + tf←
        .tv_nsec - ts.tv_nsec;
44     assembly_time /= b;
45
46     std::cout << std::scientific
47         << "Assembly_time:_"
48         << assembly_time
49         << "_seconds."
50         << std::endl;
51

```



```

52     return;
53 }
54
55 MatrixXd solution::get_elemental_M( int i)
56 {
57     MatrixXd mass = MatrixXd::Zero( elemental_dofs , ←
        elemental_dofs);
58     double area = m->get_elem_area( i);
59
60     // Diagonal of the 'mass' matrix
61     mass( 0, 0) = 2.0;
62     mass( 1, 1) = 2.0;
63     mass( 2, 2) = 2.0;
64
65     // Off-diagonal entries of 'mass' matrix
66     // upper tri
67     mass( 0, 1) = 1.0;
68     mass( 0, 2) = 1.0;
69     mass( 1, 2) = 1.0;
70     // lower tri
71     mass( 1, 0) = 1.0;
72     mass( 2, 0) = 1.0;
73     mass( 2, 1) = 1.0;
74
75     mass *= ( area / 12.0);
76
77     return mass;
78 }
79
80 MatrixXd solution::get_elemental_S( int i)
81 {
82     MatrixXd spring = MatrixXd::Zero( elemental_dofs , ←
        elemental_dofs);
83     double area = m->get_elem_area( i);
84
85     double x1 = m->get_pos( i, 0, 0);
86     double y1 = m->get_pos( i, 0, 1);
87
88     double x2 = m->get_pos( i, 1, 0);
89     double y2 = m->get_pos( i, 1, 1);

```

```

90
91     double x3 = m->get_pos( i , 2, 0);
92     double y3 = m->get_pos( i , 2, 1);
93
94     double x31 = x3 - x1;
95     double x12 = x1 - x2;
96     double x23 = x2 - x3;
97
98     double y31 = y3 - y1;
99     double y12 = y1 - y2;
100    double y23 = y2 - y3;
101
102    // Diagonal Values of spring matrix
103    spring( 0, 0) = y23 * y23 + x23 * x23;
104    spring( 1, 1) = y31 * y31 + x31 * x31;
105    spring( 2, 2) = y12 * y12 + x12 * x12;
106
107    // Upper
108    spring( 0, 1) = y23 * y31 + x23 * x31;
109    spring( 0, 2) = y23 * y12 + x23 * x12;
110    spring( 1, 2) = y31 * y12 + x31 * x12;
111
112    // Lower
113    spring( 1, 0) = y23 * y31 + x23 * x31;
114    spring( 2, 0) = y23 * y12 + x23 * x12;
115    spring( 2, 1) = y31 * y12 + x31 * x12;
116
117    spring /= (4.0 * area);
118
119    return spring;
120 }
121
122
123 MatrixXd solution::get_elemental_stiffness( int i)
124 {
125     MatrixXd mass = get_elemental_M( i);
126     MatrixXd spring = get_elemental_S( i);
127
128     return p * spring + q * mass;
129 }

```

```

130
131 void solution::assign_elemental_stiffness( MatrixXd &
      k_elem, int i)
132 {
133     for( int j = 0; j < elemental_dofs; j++)
134     {
135         for( int n = 0; n < elemental_dofs; n++)
136         {
137             int Erow = j;
138             int Ecol = n;
139
140             int row = m->get_global_id( i, j);
141             int col = m->get_global_id( i, n);
142
143             K.coeffRef( row, col) += k_elem( Erow, Ecol);
144         }
145     }
146     return;
147 }
148
149 void solution::assemble_stiffness()
150 {
151     for( int i = 0; i < numElems; i++)
152     {
153         MatrixXd k_elem = get_elemental_stiffness( i);
154         assign_elemental_stiffness( k_elem, i);
155     }
156
157     return;
158 }
159
160 void solution::apply_boundary_conditions()
161 {
162
163     for( int i = 0; i < numNodes; i++)
164     {
165         if( m->which_boundary( i) != 0)
166         {
167             double bv = get_boundary_value( i);
168             fix_global_system( bv, i);

```

```

169     }
170 }
171 // two bugs make a feature
172 K = K.transpose();
173 return;
174 }
175
176 double solution::get_boundary_value( int i)
177 {
178     double x = m->get_pos( i, 0);
179     double y = m->get_pos( i, 1);
180
181     double ans = 0.0;
182
183     if( CaseNumber == 1)
184     {
185         ans = 1.0;
186     }
187     else if( CaseNumber == 2)
188     {
189         ans = x;
190     }
191     else if( CaseNumber == 3)
192     {
193         ans = y;
194     }
195     else if( CaseNumber == 4)
196     {
197         double a = y * y * y;
198         double b = std::sin( 5.0 * (x + y));
199         double c = 2.0 * std::exp( x);
200         ans = a + b + c;
201     }
202     else
203     {
204         std::cout
205         << "Unrecognized _CaseNumber. _CaseNumber _=_ "
206         << CaseNumber
207         << std::endl;
208

```

```

209     std::abort();
210 }
211
212     return ans;
213 }
214
215 void solution::fix_global_system( double bv, int i)
216 {
217     // Reassign K(i,:) to all zeros except diag
218     double diag = 0.0;
219     for( SparseMatrix< double>::InnerIterator
220         it( K, i); it; ++it)
221     {
222         if( it.row() == it.col() )
223         {
224             diag = it.value();
225         }
226         else
227         {
228             it.valueRef() = 0.0;
229         }
230     }
231
232     F( i) = bv * diag;
233
234     return;
235 }
236
237 void solution::assemble_forcing()
238 {
239     for( int i = 0; i < numElems; i++)
240     {
241         VectorXd f_elem = get_elemental_forcing( i);
242         assign_elemental_forcing( f_elem, i);
243     }
244
245     return;
246 }
247
248 double average( double a, double b)

```

```

249 {
250     double ans = (a + b) / 2.0;
251     return ans;
252 }
253
254 double average( double a, double b, double c)
255 {
256     double ans = ( a + b + c) / 3.0;
257     return ans;
258 }
259
260 VectorXd solution::get_elemental_forcing( int elem_num)
261 {
262     VectorXd f_elem = VectorXd::Zero( elemental_dofs);
263
264     double area = m->get_elem_area( elem_num);
265
266     double x1 = m->get_pos( elem_num, 0, 0);
267     double y1 = m->get_pos( elem_num, 0, 1);
268
269     double x2 = m->get_pos( elem_num, 1, 0);
270     double y2 = m->get_pos( elem_num, 1, 1);
271
272     double x3 = m->get_pos( elem_num, 2, 0);
273     double y3 = m->get_pos( elem_num, 2, 1);
274
275     double x12 = average( x1, x2);
276     double x13 = average( x1, x3);
277     double x23 = average( x2, x3);
278
279     double y12 = average( y1, y2);
280     double y13 = average( y1, y3);
281     double y23 = average( y2, y3);
282
283     double x123 = average( x1, x2, x3);
284     double y123 = average( y1, y2, y3);
285
286     double f1 = force_at_point( x1, y1);
287     double f2 = force_at_point( x2, y2);
288     double f3 = force_at_point( x3, y3);

```

```

289
290     double f12 = force_at_point( x12, y12);
291     double f13 = force_at_point( x13, y13);
292     double f23 = force_at_point( x23, y23);
293
294     double f123 = force_at_point( x123, y123);
295
296     double tmp1 = 0.0;
297     double tmp2 = 0.0;
298     double tmp3 = 0.0;
299
300     tmp1 += 3.0 * f1;
301     tmp2 += 3.0 * f2;
302     tmp3 += 3.0 * f3;
303
304     tmp1 += 4.0 * ( f12 + f13);
305     tmp2 += 4.0 * ( f12 + f23);
306     tmp3 += 4.0 * ( f13 + f23);
307
308     tmp1 += 9.0 * f123;
309     tmp2 += 9.0 * f123;
310     tmp3 += 9.0 * f123;
311
312     f_elem(0) = tmp1;
313     f_elem(1) = tmp2;
314     f_elem(2) = tmp3;
315
316     f_elem *= area / 60.0;
317
318     return f_elem;
319 }
320
321 void solution::assign_elemental_forcing( VectorXd f_elem, ←
    int elem)
322 {
323     for( int i = 0; i < elemental_dofs; i++)
324     {
325         int row = m->get_global_id( elem, i);
326         F( row) += f_elem( i);
327     }

```

```

328     return;
329 }
330
331 double solution::force_at_point( double x, double y)
332 {
333     double ans = 0.0;
334     if( CaseNumber == 1)
335     {
336         ans = 2.0;
337     }
338     else if( CaseNumber == 2)
339     {
340         ans = 2.0 * x;
341     }
342     else if( CaseNumber == 3)
343     {
344         ans = 2.0 * y;
345     }
346     else if( CaseNumber == 4)
347     {
348         double a = 2.0 * y * y * y;
349         double b = -18.0 * y;
350         double c = 152.0 * std::sin( 5.0 * ( x + y));
351         double d = -2.0 * std::exp( x);
352
353         ans = a + b + c + d;
354     }
355     else
356     {
357         std::cout
358         << "Unrecognized CaseNumber. CaseNumber = "
359         << CaseNumber
360         << std::endl;
361
362         std::abort();
363     }
364
365     return ans;
366 }
367

```



```

368 void solution::solve_system()
369 {
370     K.makeCompressed();
371     Eigen::SparseLU< SparseMatrix< double> > solver;
372
373     solver.analyzePattern( K);
374
375     solver.factorize( K);
376
377
378     timespec ts;
379     clock_gettime( CLOCK_REALTIME, &ts);
380
381     U = solver.solve( F);
382
383     timespec tf;
384     clock_gettime( CLOCK_REALTIME, &tf);
385
386     double b = 1.0e9;
387     double solve_time = b * (tf.tv_sec - ts.tv_sec) + tf.tv_↵
        tv_nsec - ts.tv_nsec;
388     solve_time /= b;
389
390     std::cout << std::scientific
391         << "Solve_time:_"
392         << solve_time
393         << "_seconds."
394         << std::endl;
395
396     return;
397 }
398
399
400 void solution::compute_errors()
401 {
402     compute_L2_error();
403     compute_H1_error();
404
405     return;
406 }

```

```

407
408 void solution::compute_L2_error()
409 {
410     double error = 0.0;
411     for( int elem = 0; elem < numElems; elem++)
412     {
413         error += get_elemental_error_L2( elem);
414     }
415
416     L2_error = error;
417     L2_error = std::sqrt( L2_error);
418
419     error = std::sqrt( error);
420     std::cout << std::scientific
421         << "L2_Error_="
422         << L2_error
423         << std::endl;
424
425     return;
426 }
427
428 double solution::get_elemental_error_L2( int elem)
429 {
430     double area = m->get_elem_area( elem);
431
432     double x1 = m->get_pos( elem, 0, 0);
433     double y1 = m->get_pos( elem, 0, 1);
434
435     double x2 = m->get_pos( elem, 1, 0);
436     double y2 = m->get_pos( elem, 1, 1);
437
438     double x3 = m->get_pos( elem, 2, 0);
439     double y3 = m->get_pos( elem, 2, 1);
440
441     double x12 = average( x1, x2);
442     double x13 = average( x1, x3);
443     double x23 = average( x2, x3);
444
445     double y12 = average( y1, y2);
446     double y13 = average( y1, y3);

```

```

447  double y23 = average( y2, y3);
448
449  double x123 = average( x1, x2, x3);
450  double y123 = average( y1, y2, y3);
451
452  double u1 = get_exact_solution( x1, y1);
453  double u2 = get_exact_solution( x2, y2);
454  double u3 = get_exact_solution( x3, y3);
455
456  double u12 = get_exact_solution( x12, y12);
457  double u13 = get_exact_solution( x13, y13);
458  double u23 = get_exact_solution( x23, y23);
459
460  double u123 = get_exact_solution( x123, y123);
461
462  double uh1 = U( m->get_global_id( elem, 0));
463  double uh2 = U( m->get_global_id( elem, 1));
464  double uh3 = U( m->get_global_id( elem, 2));
465
466  double uh12 = average( uh1, uh2);
467  double uh13 = average( uh1, uh3);
468  double uh23 = average( uh2, uh3);
469
470  double uh123 = average( uh1, uh2, uh3);
471
472  double e1 = (u1 - uh1) * (u1 - uh1);
473  double e2 = (u2 - uh2) * (u2 - uh2);
474  double e3 = (u3 - uh3) * (u3 - uh3);
475
476  double e12 = (u12 - uh12) * (u12 - uh12);
477  double e23 = (u23 - uh23) * (u23 - uh23);
478  double e13 = (u13 - uh13) * (u13 - uh13);
479
480  double e123 = (u123 - uh123) * (u123 - uh123);
481
482  double tmp1 = e1 + e2 + e3;
483  double tmp2 = e12 + e13 + e23;
484  double tmp3 = e123;
485
486  double error = 3.0 * tmp1 + 8.0 * tmp2 + 27 * tmp3;

```

```

487
488     error *= area / 60.0;
489
490     return error;
491 }
492
493 void solution::compute_H1_error()
494 {
495     double error = 0.0;
496     for( int elem = 0; elem < numElems; elem++)
497     {
498         error += get_elemental_error_H1( elem);
499     }
500
501     H1_error = error + L2_error * L2_error;
502     H1_error = std::sqrt( H1_error);
503
504     std::cout << std::scientific
505         << "H1_Error = "
506         << H1_error
507         << std::endl;
508     return;
509 }
510
511 double solution::get_elemental_error_H1( int elem)
512 {
513     double area = m->get_elem_area( elem);
514
515     double x1 = m->get_pos( elem, 0, 0);
516     double y1 = m->get_pos( elem, 0, 1);
517
518     double x2 = m->get_pos( elem, 1, 0);
519     double y2 = m->get_pos( elem, 1, 1);
520
521     double x3 = m->get_pos( elem, 2, 0);
522     double y3 = m->get_pos( elem, 2, 1);
523
524     double x12 = average( x1, x2);
525     double x13 = average( x1, x3);
526     double x23 = average( x2, x3);

```

```

527
528 double y12 = average( y1, y2);
529 double y13 = average( y1, y3);
530 double y23 = average( y2, y3);
531
532 double x123 = average( x1, x2, x3);
533 double y123 = average( y1, y2, y3);
534
535 double ux1 = get_exact_solution_grad( x1, y1, 0);
536 double ux2 = get_exact_solution_grad( x2, y2, 0);
537 double ux3 = get_exact_solution_grad( x3, y3, 0);
538 double uy1 = get_exact_solution_grad( x1, y1, 1);
539 double uy2 = get_exact_solution_grad( x2, y2, 1);
540 double uy3 = get_exact_solution_grad( x3, y3, 1);
541
542 double ux12 = get_exact_solution_grad( x12, y12, 0);
543 double ux23 = get_exact_solution_grad( x23, y23, 0);
544 double ux13 = get_exact_solution_grad( x13, y13, 0);
545 double uy12 = get_exact_solution_grad( x12, y12, 1);
546 double uy13 = get_exact_solution_grad( x13, y13, 1);
547 double uy23 = get_exact_solution_grad( x23, y23, 1);
548
549 double ux123 = get_exact_solution_grad( x123, y123, 0);
550 double uy123 = get_exact_solution_grad( x123, y123, 1);
551
552 double gx1 = (y2 - y3) / (2.0 * area);
553 double gy1 = (x3 - x2) / (2.0 * area);
554
555 double gx2 = (y3 - y1) / (2.0 * area);
556 double gy2 = (x1 - x3) / (2.0 * area);
557
558 double gx3 = (y1 - y2) / (2.0 * area);
559 double gy3 = (x2 - x1) / (2.0 * area);
560
561 double uh1 = U( m->get_global_id( elem, 0));
562 double uh2 = U( m->get_global_id( elem, 1));
563 double uh3 = U( m->get_global_id( elem, 2));
564
565 double uhx1 = uh1 * gx1;
566 double uhx2 = uh2 * gx2;

```

```

567  double uhx3 = uh3 * gx3;
568  double uhy1 = uh1 * gy1;
569  double uhy2 = uh2 * gy2;
570  double uhy3 = uh3 * gy3;
571
572  double guhx = uhx1 + uhx2 + uhx3;
573  double guhy = uhy1 + uhy2 + uhy3;
574
575  double ex1  = (guhx - ux1) * (guhx - ux1);
576  double ey1  = (guhy - uy1) * (guhy - uy1);
577  double ex2  = (guhx - ux2) * (guhx - ux2);
578  double ey2  = (guhy - uy2) * (guhy - uy2);
579  double ex3  = (guhx - ux3) * (guhx - ux3);
580  double ey3  = (guhy - uy3) * (guhy - uy3);
581
582  double tmp1 = ex1 + ey1 + ex2 + ey2 + ex3 + ey3;
583
584  double ex12  = (guhx - ux12) * (guhx - ux12);
585  double ey12  = (guhy - uy12) * (guhy - uy12);
586  double ex23  = (guhx - ux23) * (guhx - ux23);
587  double ey23  = (guhy - uy23) * (guhy - uy23);
588  double ex13  = (guhx - ux13) * (guhx - ux13);
589  double ey13  = (guhy - uy13) * (guhy - uy13);
590
591  double tmp2 = ex12 + ey12 + ex13 + ey13 + ex23 + ey23;
592
593  double ex123 = (guhx - ux123) * (guhx - ux123);
594  double ey123 = (guhy - uy123) * (guhy - uy123);
595
596  double tmp3 = ex123 + ey123;
597
598  double error = 3.0 * tmp1 + 8.0 * tmp2 + 27 * tmp3;
599
600  error *= area / 60.0;
601
602  return error;
603 }
604
605 double solution::get_exact_solution( double x, double y)
606 {

```

```

607  double u = 0.0;
608
609  if( CaseNumber == 1)
610  {
611      u = 1.0;
612  }
613  else if( CaseNumber == 2)
614  {
615      u = x;
616  }
617  else if( CaseNumber == 3)
618  {
619      u = y;
620  }
621  else if( CaseNumber == 4)
622  {
623      double a = y * y * y;
624      double b = std::sin( 5.0 * (x + y));
625      double c = 2.0 * std::exp( x);
626      u = a + b + c;
627  }
628  return u;
629 }
630
631 double solution::get_exact_solution_grad( double x, ↵
        double y, int xy)
632 {
633     double du = 0.0;
634
635     if( xy == 0)
636     {
637         if( CaseNumber == 1)
638         {
639             du = 0.0;
640         }
641         else if( CaseNumber == 2)
642         {
643             du = 1.0;
644         }
645         else if( CaseNumber == 3)

```

```

646     {
647         du = 0.0;
648     }
649     else if( CaseNumber == 4)
650     {
651         double a = 0.0;
652         double b = 5.0 * std::cos( 5.0 * (x + y));
653         double c = 2.0 * std::exp( x);
654         du = a + b + c;
655     }
656 }
657 else if ( xy == 1)
658 {
659     if( CaseNumber == 1)
660     {
661         du = 0.0;
662     }
663     else if( CaseNumber == 2)
664     {
665         du = 0.0;
666     }
667     else if( CaseNumber == 3)
668     {
669         du = 1.0;
670     }
671     else if( CaseNumber == 4)
672     {
673         double a = 3.0 * y * y;
674         double b = 5.0 * std::cos( 5.0 * (x + y));
675         double c = 0.0;
676         du = a + b + c;
677     }
678 }
679 else
680 {
681     std::cout
682         << " Unrecognized _xy_option:_"
683         << xy
684         << std::endl;
685

```



```

686     std::abort();
687 }
688
689     return du;
690 }

```

A.7 solution.hpp

```

1  #ifndef SOLUTION_HPP
2  #define SOLUTION_HPP
3
4  #include "mesh.hpp"
5
6  class solution
7  {
8      public:
9          // Constructor
10         solution( mesh* m_, int CaseNumber_);
11
12         // Assembles the stiffness and forcing functions
13         void assemble_problem();
14
15         // Apply the Dirichlet boundary conditions
16         // for the specified case number.
17         void apply_boundary_conditions();
18
19         // Solves the system  $KU=F$  for  $U$ 
20         void solve_system();
21
22         // Compute the errors
23         void compute_errors();
24
25         // Destructor
26         ~solution();
27
28     private:
29         // Pointer to the mesh
30         mesh* m;
31
32         // PDE Parameter "p"

```

```

33  double p;
34
35  // PDE Parameter "q"
36  double q;
37
38  // Number of mesh nodes
39  int numNodes;
40
41  // Number of elements in the mesh
42  int numElems;
43
44  // Number of degrees of freedom per element
45  int elemental_dofs;
46
47  // CaseNumber
48  int CaseNumber;
49
50  // Global Reduced Stiffness Matrix
51  SparseMatrix<double> K;
52
53  // Global Reduced Forcing Vector
54  VectorXd F;
55
56  // Solution Vector
57  VectorXd U;
58
59  // Assembles only the stiffness matrix
60  void assemble_stiffness();
61
62  // Assembles only the forcing matrix
63  void assemble_forcing();
64
65  // Returns the stiffness matrix for element i
66  MatrixXd get_elemental_stiffness( int i);
67
68  // Assigns the i'th elemental stiffness k_elem
69  // to the global stiffness matrix
70  void assign_elemental_stiffness( MatrixXd k_elem, int ←
    i);
71

```

```

72 // Gets the elemental pure mass matrix
73 MatrixXd get_elemental_M( int i);
74
75 // Gets the elemental pure spring matrix
76 MatrixXd get_elemental_S( int i);
77
78 // Gets the elemental forcing function.
79 // Body forces only.
80 VectorXd get_elemental_forcing( int elem_num);
81
82 // Assigns the elemental forcing vector to the global ←
   one.
83 void assign_elemental_forcing( VectorXd f_elem , int ←
   elem);
84
85 // Gets the force at a point based on the
86 // case nuber defined at construction time.
87 double force_at_point( double x, double y);
88
89 // Adjusts the global system for boundary value bv
90 // at node i.
91 void fix_global_system( double bv, int i);
92
93 // Gets the DBC boundary value for node number i
94 // based on Case number defined at construction.
95 double get_boundary_value( int i);
96
97 // Computes the L2 error.
98 void compute_L2_error();
99
100 // Computes the H1 error.
101 void compute_H1_error();
102
103 // Gets the exact solution based on the Case
104 // number provided at construction time.
105 double get_exact_solution( double x, double y);
106
107 // Gets the L2 error for element elem.
108 double get_elemental_error_L2( int elem);
109

```

```

110 // Gets the H1 error for element elem.
111 double get_elemental_error_H1( int elem);
112
113 // Gets the exact gradient of the solution.
114 double get_exact_solution_grad( double x, double y, ↵
    int xy);
115
116 // The value of the H1 norm
117 double H1_error;
118
119 // The value of the L2 norm
120 double L2_error;
121 };
122
123 #endif

```

A.8 CMakeLists.txt

```

1 set( HW5_SOURCES
2 coord.cpp
3 coord.hpp
4 driver.cpp
5 driver.hpp
6 mesh.cpp
7 mesh.hpp
8 element.cpp
9 element.hpp
10 vertex.cpp
11 vertex.hpp
12 solution.cpp
13 solution.hpp
14 eig_wrap.hpp
15 fea_hw5.cpp)
16
17 add_executable(hw5 ${HW5_SOURCES})
18
19 target_include_directories(hw5 PUBLIC ${EIG_DIR})

```

A.9 eig_wrap.hpp

```
1 #ifndef EIG_WRAP
2 #define EIG_WRAP
3
4 // eig_wrap.hpp
5 // This wraps all needed Eigen package
6 // headers into a "convenient" header.
7 // It also defines using directives.
8
9 #include </lore/clougj/Learning_Codes/FEA/Eigen_Package/↵
    Eigen/Eigen>
10
11 using Eigen::MatrixXd;
12 using Eigen::VectorXd;
13 using Eigen::SparseMatrix;
14
15 #endif
```