

Part II: Implementation

The purpose of this exercise was to use the Finite Element Method (FEM) to approximate, and study, the solution to the problem presented in Eq. 1.

$$-(p(x)u'(x))' + q(x)u(x) = f(x) \quad (1)$$

$$u(0) = \alpha, u(1) = \beta \quad (2)$$

This equation was solved on the domain $x \in [0, 1]$. The analytical solution was assumed to exist uniquely. The variables in Eq. 1 are as denoted below in Eq. 3.

$$u(x) \in C^2[0, 1] \quad (3)$$

$$f, q \in C^0[0, 1] \quad q \geq 0 \quad (4)$$

$$p \in C^1[0, 1] \quad p > 0 \quad (5)$$

The problem domain of one unit in one dimension was divided into a nonuniform but structured mesh. The rule for determining element size is shown in Eq. .

$$h_j = \begin{cases} 0.9\Delta x & \text{for odd } j \\ 1.1\Delta x & \text{for even } j \end{cases}$$

The Δx referred to in Eq. is the average size of any given element. Specifically, it is calculated by use of Eq. 6.

$$\Delta x = \frac{1}{N + 1} \quad (6)$$

Here, N is the number of degrees of freedom; $N + 1$ is the number of elements. Tests were conducted for $N = 10, 20, 40, 80, 160, 320$. These six mesh size tests were conducted for six different cases which are described in Table .

Case Number	α	β	p	q	u
1	0	0	3	2	$u = x(x - 1)(\sin(5x) + 3e^x)$
2	0	0	$p = 1 + x$	0	$u = x(x - 1)(\sin(5x) + 3e^x)$
3	4	4	3	2	$u = 4$
4	-2	-1	3	2	$u = x - 2$
5	-3	-2	3	2	$u = x^2 - 3$

A computer code was written to evaluate the posed problems. The results follow this section and the code itself is appended to this report. Plots of the error for $N = 10, 20, 40$ with respect to location are presented in Figures 1 and Figure 2.

Tabulated data, in the form of error norms and convergence order are presented in Tables 1 through 5.

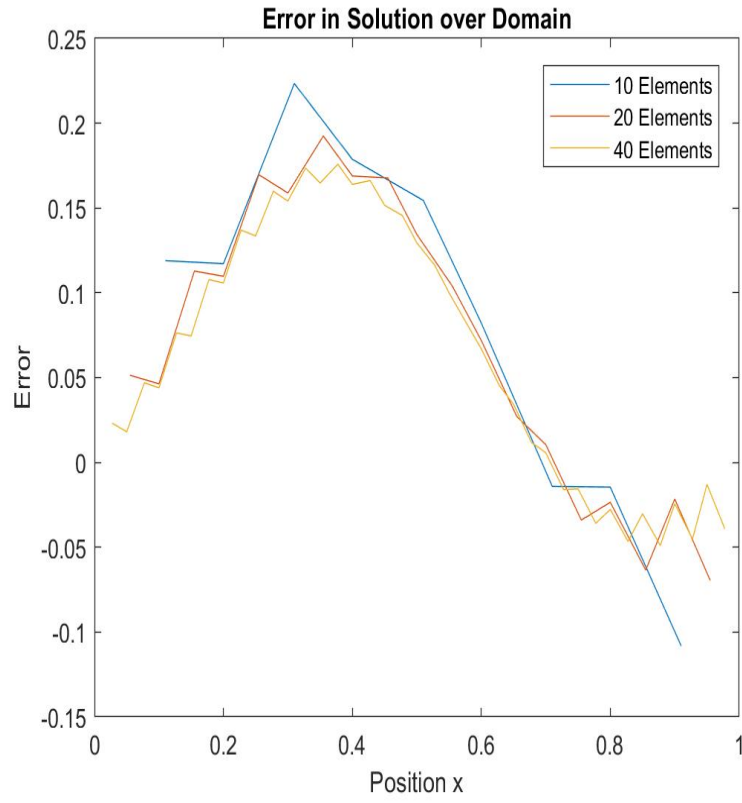


Figure 1: Error in solution over domain of problem.

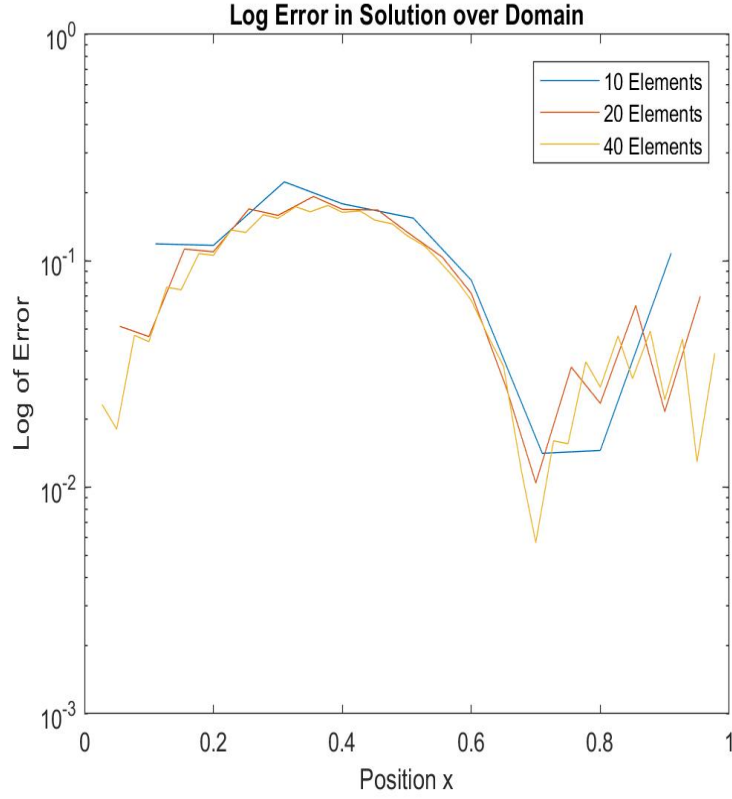


Figure 2: Log of error in solution over domain of problem.

Table 1: Errors and convergence orders for Case 1.

$N + 1$	$\ e_h(\cdot)\ _{L^2([0,1])}$	Order	$\ e_h(\cdot)\ _{L^\infty([0,1])}$	Order	$\ e_h(\cdot)\ _h$	Order
10	2.072428e-02	0.00	2.233885e-01	0.00	1.884026e-01	0.00
20	9.421318e-03	1.13	1.924613e-01	0.21	1.712967e-01	0.14
40	3.740229e-03	1.33	1.758183e-01	0.13	1.360083e-01	0.33
80	1.398561e-03	1.41	1.685136e-01	0.06	1.017135e-01	0.42
160	5.082905e-04	1.46	1.646495e-01	0.03	7.393317e-02	0.46
320	1.821839e-04	1.48	1.627100e-01	0.02	5.299896e-02	0.48

The following would be done to incorporate non-homogeneous Dirichlet boundary conditions. First, the only change to the weak form of the problem would be separating the trial and test spaces; the solution must have a non-zero value on the specified boundary whereas the trial function must be zero on the boundaries. For the FEM formulation, the contributions of the non-zero boundary conditions are

Table 2: Errors and convergence orders for Case 2.

$N + 1$	$\ e_h(\cdot)\ _{L^2([0,1])}$	Order	$\ e_h(\cdot)\ _{L^\infty([0,1])}$	Order	$\ e_h(\cdot)\ _h$	Order
10	2.343497e-02	0.00	3.718547e-01	0.00	2.130452e-01	0.00
20	2.133812e-02	0.14	6.910094e-01	-0.89	3.879659e-01	-0.86
40	1.097459e-02	0.96	9.852965e-01	-0.51	3.990759e-01	-4.07
80	4.620878e-03	1.24	1.175112e+00	-0.25	3.360638e-01	0.25
160	1.780002e-03	1.37	1.278376e+00	-0.12	2.589093e-01	0.37
320	6.566801e-04	1.43	1.332562e+00	-0.06	1.910342e-01	0.44

Table 3: Errors and convergence orders for Case 3.

$N + 1$	$\ e_h(\cdot)\ _{L^2([0,1])}$	Order	$\ e_h(\cdot)\ _{L^\infty([0,1])}$	Order	$\ e_h(\cdot)\ _h$	Order
10	7.423628e-01	0.00	3.896897e+00	0.00	6.748753e+00	0.00
20	5.327322e-01	0.48	3.945805e+00	-0.02	9.686040e+00	-0.52
40	3.797393e-01	0.49	3.972235e+00	-0.01	1.380870e+01	-0.51
80	2.696418e-01	0.43	3.985950e+00	-0.01	1.961031e+01	-0.51
160	1.910725e-01	0.49	3.992933e+00	-0.01	2.779236e+01	-0.50
320	1.352541e-01	0.50	3.996456e+00	-0.01	3.934665e+01	-0.50

Table 4: Errors and convergence orders for Case 4.

$N + 1$	$\ e_h(\cdot)\ _{L^2([0,1])}$	Order	$\ e_h(\cdot)\ _{L^\infty([0,1])}$	Order	$\ e_h(\cdot)\ _h$	Order
10	2.006381e-01	0.00	1.847478e+00	0.00	1.823982e+00	0.00
20	1.385033e-01	0.53	1.922485e+00	-0.05	2.518242e+00	-0.47
40	9.681655e-02	0.52	1.960923e+00	-0.02	3.520602e+00	-0.48
80	6.807593e-02	0.51	1.980381e+00	-0.01	4.950977e+00	-0.49
160	4.800344e-02	0.50	1.990170e+00	-0.01	6.982319e+00	-0.49
320	3.389673e-02	0.50	1.995080e+00	-0.01	9.860868e+00	-0.50

taken account for in the forcing vector. No changes need to be made to represent U_h on *interior* elements. However, on the boundary, additional hat functions are used to recover the assigned boundary values. The matrix \mathbf{A} is not changed as the original PDE does not change. The forcing vector \mathbf{F} includes the addition of the boundary terms on the first and last entry.

While the L_2 error norm decreases for all tests, the same is not true for the L_∞ or the energy norm. Appended to the code is additional work showing the hand calculations to derive the forcing functions and their approximated equivalents. The **Eigen** software package was used as the linear solver for this project. The

Table 5: Errors and convergence orders for Case 5.

$N + 1$	$\ e_h(\cdot)\ _{L^2([0,1])}$	Order	$\ e_h(\cdot)\ _{L^\infty([0,1])}$	Order	$\ e_h(\cdot)\ _h$	Order
10	3.831430e-01	0.00	2.837704e+00	0.00	3.483118e+00	0.00
20	2.704644e-01	0.50	2.917944e+00	-0.04	4.917534e+00	-0.49
40	1.912960e-01	0.49	2.958735e+00	-0.02	6.956220e+00	-0.50
80	1.353213e-01	0.49	2.979307e+00	-0.09	9.841546e+00	-0.50
160	9.571245e-02	0.49	2.989638e+00	-0.04	1.392181e+01	-0.50
320	6.768923e-02	0.49	2.994815e+00	-0.02	1.969141e+01	-0.50

matrix inversion method used was Householder-QR with pivoting.

A Source Code and Headers

A.1 fea_hw1.cpp

```
1 #include "driver.hpp"
2 #include <iostream>
3
4 int main( int argc, char** argv)
5 {
6     // Define Mesh paramters, testing size
7     double oddSize      = 0.9;
8     double evenSize     = 1.1;
9     int     NplusOneArray[] = {10, 20, 40, 80, 160, 320};
10    //int     NplusOneArray[] = {10, 10, 10, 10, 10, 10};
11    int     numTests      = 6;
12
13    for ( int i = 1; i < 6; i++)
14    {
15        drive_problem( oddSize, evenSize, NplusOneArray, ↵
16                      numTests, i );
17    }
18    return 0;
19 }
```

A.2 driver.cpp

```
1 #include "driver.hpp"
2
3 void drive_problem( double oddSize,
4                   double evenSize,
5                   int*   NplusOneArray,
6                   int    numTests,
7                   int    caseNumber )
8 {
9     std::vector<VectorXd> solutions;
10    std::vector<mesh1D*> meshes;
11
12    for( int i = 0; i < numTests; i++)
13    {
```

```

14 // First construct the mesh
15 int NplusOne = NplusOneArray[i];
16 mesh1D* mesh = new mesh1D( oddSize, evenSize, ←
    NplusOne);
17 meshes.push_back( mesh);
18
19 // Second construct stiffness matrix
20 springFactory* sf = new springFactory( mesh, ←
    caseNumber);
21 sf->create_stiffness();
22 MatrixXd K = sf->getStiffness();
23
24 // Third construct forcing vector
25 forcingFactory* ff = new forcingFactory( mesh, ←
    caseNumber);
26 ff->create_forcing();
27 VectorXd F = ff->getForcing();
28
29 // Calculate and store solution
30 solutions.push_back( K.colPivHouseholderQr().solve( F←
    ));
31
32 delete ff;
33 delete sf;
34 }
35 errorCalcs( solutions, meshes, NplusOneArray, ←
    caseNumber);
36 solutions.clear();
37 for (int i = 0; i < meshes.size(); i++)
38 {
39     delete meshes[i];
40 }
41 meshes.clear();
42
43 return;
44 }

```

A.3 driver.hpp

```

1 #ifndef DRIVER_HPP

```

```

2 #define DRIVER_HPP
3
4 #include "eig_wrap.hpp"
5 #include "mesh1D.hpp"
6 #include "stiffness.hpp"
7 #include "forcing.hpp"
8 #include "errorCalcs.hpp"
9
10 void drive_problem( double oddSize ,
11                   double evenSize ,
12                   int*   NplusOneArray ,
13                   int    numTests ,
14                   int    caseNumber );
15
16 #endif

```

A.4 element1D.cpp

```

1 #include "element1D.hpp"
2
3 #include <cstdlib>
4 #include <iostream>
5
6 elem::elem( double leftPos_ , double rightPos_ )
7 {
8     rightPos = rightPos_;
9     leftPos = leftPos_;
10
11     length = rightPos - leftPos;
12
13     if ( length <= 0.0 )
14     {
15         std::cout << "Degenerate element construction ↵
16             attempted. ↵"
17             << "Attempted length = ↵"
18             << length << std::endl;
19
20         std::abort();
21     }
22 }

```



```

22
23 elem::~~elem()
24 {
25
26 }
27
28 double elem::getLength()
29 {
30     return length;
31 }
32
33 double elem::getLeftPos()
34 {
35     return leftPos;
36 }
37
38 double elem::getRightPos()
39 {
40     return rightPos;
41 }

```

A.5 element1D.hpp

```

1 #ifndef ELEMENT1D_HPP
2 #define ELEMENT1D_HPP
3
4 class elem
5 {
6
7     public:
8         // Gets the length of an element.
9         double getLength();
10
11         // Gets the left Node Position.
12         double getLeftPos();
13
14         // Gets the right Node Position.
15         double getRightPos();
16
17         // Constructor

```

```

18     elem( double leftPos_ , double rightPos_);
19
20     // Destructor
21     ~elem();
22
23     private:
24         // The length of the element.
25         double length;
26
27         // The coordinate of the left node.
28         double leftPos;
29
30         // The coordinate of the right node.
31         double rightPos;
32 };
33
34 #endif

```

A.6 errorCalcs.cpp

```

1 #include "errorCalcs.hpp"
2
3 #include <vector>
4 #include <cmath>
5 #include <iostream>
6 #include <sstream>
7 #include <fstream>
8 #include <string>
9 #include <cstdlib>
10
11 void errorCalcs( std::vector<VectorXd> solutions ,
12                 std::vector<mesh1D*> meshes ,
13                 int* NA,
14                 int caseNumber)
15 {
16     EC* ec = new EC( solutions , meshes , NA, caseNumber);
17
18     ec->create_analytics();
19     ec->calc_errors();
20     ec->write();

```

```

21
22     delete ec;
23     return;
24 }
25
26 void EC::create_analyticals()
27 {
28     for( int i = 0; i < ms.size(); i++)
29     {
30         int numDofs = na[i] - 1;
31         VectorXd U = VectorXd::Zero( numDofs);
32         // For each mesh, we need to calculate the exact ↵
33         nodal
34         // solutions to use a the analytic solution
35         for( int j = 0; j < numDofs; j++)
36         {
37             double xj = ms[i]->getElem(j).getRightPos();
38             if (( cn == 1) || ( cn == 2) )
39             {
40                 // This is the p3q2 case and the p(x), q0 case
41                 U( j) = xj * (xj - 1.0) * ( std::sin( 5.0 * xj) +↵
42                     3.0 * std::exp( xj));
43             }
44             else if ( cn == 3)
45             {
46                 U( j) = 4.0;
47             }
48             else if ( cn == 4)
49             {
50                 U( j) = xj - 2.0;
51             }
52             else if ( cn == 5)
53             {
54                 U( j) = xj * xj - 3.0;
55             }
56         }
57         us.push_back( U);
58     }
59     return;
60 }

```

```

59
60 void EC::calc_errors()
61 {
62     getDiff();
63     L_2();
64     L_inf();
65     Energy();
66     return;
67 }
68
69 void EC::getDiff()
70 {
71     for( int i = 0; i < us.size(); i++)
72     {
73         ehs.push_back( sols[i] - us[i] );
74     }
75     return;
76 }
77
78 double EC::e_interp( double ea,
79                     double eb,
80                     double xa,
81                     double xb,
82                     double h,
83                     double x)
84 {
85     return ea * (xb - x) / h + eb * (x - xa) / h;
86 }
87
88 void EC::L_2()
89 {
90     double tmp = 0.0;
91     for( int i = 0; i < ehs.size(); i++)
92     {
93         VectorXd e = ehs[i];
94         mesh1D* m = ms[i];
95
96         for( int j = 0; j < e.size(); j++)
97         {
98             double hj = m->getElem( j ).getLength();

```

```

99     double xa = m->getElem( j ).getLeftPos();
100    double xb = m->getElem( j ).getRightPos();
101    double ea = 0.0;
102    if( j != 0)
103    {
104        double ea = e(j-1);
105    }
106    double eb = e(j);
107    // M perscribed in problem handout
108    int M = 3;
109    for ( int m = 1; m <= M; m++)
110    {
111        double x = xa + hj * m / M;
112        double e_intp = e_interp( ea, eb, xa, xb, hj, x);
113        tmp = e_intp * e_intp;
114    }
115    tmp *=  hj / M;
116    }
117    L2.push_back( std::sqrt( tmp));
118 }
119
120 std::cout << "L2_Error = " << std::endl;
121 for ( int i = 0; i < L2.size(); i++)
122 {
123     std::cout << L2[i] << std::endl;
124 }
125
126 for ( int i = 0; i < L2.size(); i++)
127 {
128     if ( i == 0)
129     {
130         L2_order.push_back( 0.0);
131     }
132     else
133     {
134         double top      = std::log( L2[ i-1] / L2[i]);
135         double bottom = std::log( 2.0);
136         double tmp      = top / bottom;
137         L2_order.push_back( tmp);
138     }

```

```

139     }
140     return;
141 }
142
143 void EC::L_inf()
144 {
145     double tmp = 0.0;
146     for( int i = 0; i < ehs.size(); i++)
147     {
148         VectorXd e = ehs[i];
149         mesh1D* m = ms[i];
150
151         for( int j = 0; j < e.size(); j++)
152         {
153             double hj = m->getElem( j ).getLength();
154             double xa = m->getElem( j ).getLeftPos();
155             double xb = m->getElem( j ).getRightPos();
156             double ea = 0.0;
157             if( j != 0)
158             {
159                 double ea = e(j-1);
160             }
161             double eb = e(j);
162             // M perscribed in problem handout
163             int M = 3;
164             for ( int m = 1; m <= M; m++)
165             {
166                 double x = xa + hj * m / M;
167                 double e_intp = std::abs( e_interp( ea, eb, xa, ←
168                     xb, hj, x) );
169                 if ( tmp < e_intp)
170                 {
171                     tmp = e_intp;
172                 }
173             }
174             Linf.push_back( tmp);
175             tmp = 0.0;
176         }
177     }

```

```

178     std::cout << "Linf_Error_=" << std::endl;
179     for ( int i =0; i < Linf.size(); i++)
180     {
181         std::cout << Linf[i] << std::endl;
182     }
183
184     for ( int i = 0; i < Linf.size(); i++)
185     {
186         if ( i == 0)
187         {
188             L_inf_order.push_back( 0.0);
189         }
190         else
191         {
192             double top      = std::log( Linf[ i-1] / Linf[i]);
193             double bottom = std::log( 2.0);
194             double tmp      = top / bottom;
195             L_inf_order.push_back( tmp);
196         }
197     }
198     return;
199 }
200
201 void EC::Energy()
202 {
203     double tmp = 0.0;
204     for( int i = 0; i < ehs.size(); i++)
205     {
206         VectorXd e = ehs[i];
207         mesh1D* m = ms[i];
208
209         for( int j = 0; j < e.size(); j++)
210         {
211             double hj = m->getElem( j).getLength();
212             double xa = m->getElem( j).getLeftPos();
213             double xb = m->getElem( j).getRightPos();
214             double ea = 0.0;
215             if( j != 0)
216             {
217                 double ea = e(j-1);

```

```

218     }
219     double eb = e(j);
220     // M perscribed in problem handout
221     int M = 3;
222     for ( int m = 1; m <= M; m++)
223     {
224         // Since we have a linear basis, the
225         // derivatives are constant inside each element
226         double dedx = (eb - ea) / hj;
227         tmp = dedx * dedx;
228     }
229     tmp *=  hj / M;
230 }
231 NRG.push_back( std::sqrt( tmp));
232 }
233
234 std::cout << "NRG_Error_=" << std::endl;
235 for ( int i =0; i < NRG.size(); i++)
236 {
237     std::cout << NRG[i] << std::endl;
238 }
239
240 for ( int i = 0; i < NRG.size(); i++)
241 {
242     if ( i == 0)
243     {
244         NRG_order.push_back( 0.0);
245     }
246     else
247     {
248         double top      = std::log( NRG[ i-1] / NRG[i]);
249         double bottom = std::log( 2.0);
250         double tmp      = top / bottom;
251         NRG_order.push_back( tmp);
252     }
253 }
254 return;
255 }
256
257 void EC::write()

```



```

258 {
259     std::string filename = "errorCalcs_";
260     if( cn == 1)
261     {
262         filename = filename + "1.txt";
263     }
264     else if ( cn == 2)
265     {
266         filename = filename + "2.txt";
267     }
268     else if ( cn == 3)
269     {
270         filename = filename + "3.txt";
271     }
272     else if ( cn == 4)
273     {
274         filename = filename + "4.txt";
275     }
276     else if ( cn == 5)
277     {
278         filename = filename + "5.txt";
279     }
280     else
281     {
282         std::cout << "Bad_case_Number_" << cn << std::endl;
283         std::abort();
284     }
285     std::ofstream file;
286     file << std::scientific;
287     file.open( filename.c_str());
288
289     for( int i = 0; i < ehs.size(); i++)
290     {
291         file << "Error_with_" << na[i] << "_elements:" << std::
292             ::endl;
293         file << ehs[i] << std::endl;
294         file << "Abs(Error)_with_" << na[i] << "_elements:" <<
295             << std::endl;
296         file << ehs[i].cwiseAbs() << std::endl;
297         file << "Node_Locations:" << std::endl;

```

```

296     for( int j = 0; j < (ms[i]->getNumElems() - 1); j++)
297     {
298         file << ms[i]->getElem(j).getRightPos() << std::endl;
299     }
300
301     file << "L2_Error:" << std::endl;
302     for( int j = 0; j < L2.size(); j++)
303     {
304         file << L2[j] << std::endl;
305     }
306     file << "L2_Order:" << std::endl;
307     for( int j = 0; j < L2_order.size(); j++)
308     {
309         file << L2_order[j] << std::endl;
310     }
311
312     file << "Linf_Error:" << std::endl;
313     for( int j = 0; j < Linf.size(); j++)
314     {
315         file << Linf[j] << std::endl;
316     }
317     file << "Linf_Order:" << std::endl;
318     for( int j = 0; j < Linf_order.size(); j++)
319     {
320         file << Linf_order[j] << std::endl;
321     }
322
323     file << "Energy_Error:" << std::endl;
324     for( int j = 0; j < NRG.size(); j++)
325     {
326         file << NRG[j] << std::endl;
327     }
328     file << "NRG_Order:" << std::endl;
329     for( int j = 0; j < NRG_order.size(); j++)
330     {
331         file << NRG_order[j] << std::endl;
332     }
333 }
334

```

```

335     file.close();
336
337     return;
338 }
339
340 EC::EC( std::vector<VectorXd> solutions ,
341         std::vector<mesh1D*> meshes ,
342         int* NA,
343         int caseNumber)
344 {
345     sols = solutions;
346     ms   = meshes;
347     na   = NA;
348     cn   = caseNumber;
349 }

```

A.7 errorCalcs.hpp

```

1  #ifndef ERRORCALCS_HPP
2  #define ERRORCALCS_HPP
3
4  #include "eig_wrap.hpp"
5  #include "mesh1D.hpp"
6
7  void errorCalcs( std::vector<VectorXd> solutions ,
8                  std::vector<mesh1D*> meshes ,
9                  int* NA,
10                 int caseNumber);
11
12 class EC
13 {
14     public:
15         // Constructor
16         EC( std::vector<VectorXd> solutions ,
17            std::vector<mesh1D*> meshes ,
18            int* NA,
19            int caseNumber);
20
21         // Create the analytical solution, dependent on case
22         void create_analytics();

```

```

23
24 // Calculates the L2, L_inf, and Energy; includes  $\leftarrow$ 
    convergence
25 void calc_errors();
26
27 // Writes calculation results to file
28 void write();
29
30 private:
31 // vector of FE solution vectors
32 std::vector<VectorXd> sols;
33
34 // The Array of NplusOne values
35 int* na;
36
37 // The Case Number for this problem
38 int cn;
39
40 // The vector of analytical solutions
41 std::vector<VectorXd> us;
42
43 // The vector of mesh pointers
44 std::vector<mesh1D*> ms;
45
46 // Calculate the actual error as a difference
47 void getDiff();
48
49 // A vector of error vectors
50 std::vector<VectorXd> ehs;
51
52 // Calculates the L2 norm and order
53 void L_2();
54
55 // The vector of L_2 norms
56 std::vector<double> L2;
57
58 // The vector of L_2 orders
59 std::vector<double> L_2_order;
60
61 // Calculates the L_inf norm and order

```

```

62     void L_inf();
63
64     // The vector of L_inf norms
65     std::vector<double> Linf;
66
67     // The vector of L_inf orders
68     std::vector<double> L_inf_order;
69
70     // Calculates the Energy norm and order
71     void Energy();
72
73     // The vector of NRG norms
74     std::vector<double> NRG;
75
76     // The vector of NRG orders
77     std::vector<double> NRG_order;
78
79     // An intra element interpolation of the error
80     double e_interp( double ea,
81                     double eb,
82                     double xa,
83                     double xb,
84                     double h,
85                     double x);
86 };
87
88 #endif

```

A.8 forcing.cpp

```

1  #include "forcing.hpp"
2
3  #include <cmath>
4  #include <iostream>
5
6  forcingFactory::forcingFactory( mesh1D* mesh, int ↵
    caseNumber_)
7  {
8      m = mesh;
9      caseNumber = caseNumber_;

```

```

10  int numNodes = m->getNumNodes();
11  // Assuming that only the interior nodes are
12  // degrees of freedom
13  numDofs = numNodes - 2;
14  F = VectorXd::Zero( numDofs);
15  }
16
17  void forcingFactory::create_forcing()
18  {
19      for ( int i = 0; i < numDofs; i++)
20      {
21          assign_force( i);
22      }
23
24      std::cout << "F_=" << std::endl;
25      std::cout << F << std::endl;
26
27      return;
28  }
29
30  VectorXd forcingFactory::getForcing()
31  {
32      return F;
33  }
34
35  void forcingFactory::assign_force( int row)
36  {
37      if ( caseNumber == 1)
38      { // A caseNumber of 1 is for the  $p = 3, q = 2$  problem
39          p3q2force( row);
40      }
41      else if ( caseNumber == 2)
42      { // A casenumber of 2 is for the  $p = 1+x, q=0$  problem
43          pXq0force( row);
44      }
45      else if ( caseNumber == 3)
46      {
47          case3force( row);
48      }
49      else if ( caseNumber == 4)

```

```

50     {
51         case4force( row);
52     }
53     else if ( caseNumber == 5)
54     {
55         case5force( row);
56     }
57     else
58     {
59         std::cout << "Unrecognized_caseNumber_=" << caseNumber << std::endl;
60         std::abort();
61     }
62     return;
63 }
64
65 void forcingFactory::p3q2force( int row)
66 {
67     double hi    = m->getElem( row).getLength();
68     double hip1  = m->getElem( row + 1).getLength();
69
70     double xim1  = m->getElem( row).getLeftPos();
71     double xi    = m->getElem( row).getRightPos();
72     double xip1  = m->getElem( row + 1).getLeftPos();
73
74     double fim1  = analytic_p3q2( xim1);
75     double fi    = analytic_p3q2( xi);
76     double fip1  = analytic_p3q2( xip1);
77
78     F( row) = fim1 * hi / 6.0
79             + fi * ( hi + hip1) / 3.0
80             + fip1 * hip1 / 6.0;
81
82     return;
83 }
84
85 void forcingFactory::case3force( int row)
86 {
87     double alpha = 4.0;
88     double beta  = 4.0;

```

```

89  double p      = 3.0;
90  double q      = 2.0;
91
92  double hi     = m->getElem( row).getLength();
93  double hip1  = m->getElem( row + 1).getLength();
94
95  double xim1 = m->getElem( row).getLeftPos();
96  double xi   = m->getElem( row).getRightPos();
97  double xip1 = m->getElem( row + 1).getLeftPos();
98
99  double fim1 = analytic_3( xim1);
100 double fi   = analytic_3( xi);
101 double fip1 = analytic_3( xip1);
102
103 if ( row == 1)
104 {
105     F( row) += alpha * ( - p / hi + q * hi / 6.0);
106 }
107 else if (row == F.size())
108 {
109     F( row) += beta * ( - p / hi + q * hi / 6.0);
110 }
111
112 F( row) = fim1 * hi / 6.0
113          + fi * ( hi + hip1) / 3.0
114          + fip1 * hip1 / 6.0;
115 return;
116 }
117
118 double forcingFactory::analytic_3( double x)
119 {
120     return 8.0;
121 }
122
123 void forcingFactory::case4force( int row)
124 {
125     double alpha = -2.0;
126     double beta  = -1.0;
127     double p     = 3.0;
128     double q     = 2.0;

```



```

129
130     double hi    = m->getElem( row).getLength();
131     double hip1 = m->getElem( row + 1).getLength();
132
133     double xim1 = m->getElem( row).getLeftPos();
134     double xi   = m->getElem( row).getRightPos();
135     double xip1 = m->getElem( row + 1).getLeftPos();
136
137     double fim1 = analytic_4( xim1);
138     double fi   = analytic_4( xi);
139     double fip1 = analytic_4( xip1);
140
141     if ( row == 1)
142     {
143         F( row) += alpha * ( - p / hi + q * hi / 6.0);
144     }
145     else if (row == F.size())
146     {
147         F( row) += beta * ( - p / hi + q * hi / 6.0);
148     }
149
150     F( row) = fim1 * hi / 6.0
151              + fi * ( hi + hip1) / 3.0
152              + fip1 * hip1 / 6.0;
153     return;
154 }
155
156 double forcingFactory::analytic_4( double x)
157 {
158     double f = 0.0;
159     f = 2.0 * x - 4.0;
160     return f;
161 }
162
163 void forcingFactory::case5force( int row)
164 {
165     double alpha = -3.0;
166     double beta  = -2.0;
167     double p     = 3.0;
168     double q     = 2.0;

```

```

169
170 double hi    = m->getElem( row).getLength();
171 double hip1  = m->getElem( row + 1).getLength();
172
173 double xim1 = m->getElem( row).getLeftPos();
174 double xi   = m->getElem( row).getRightPos();
175 double xip1 = m->getElem( row + 1).getLeftPos();
176
177 double fim1 = analytic_5( xim1);
178 double fi   = analytic_5( xi);
179 double fip1 = analytic_5( xip1);
180
181 if ( row == 1)
182 {
183     F( row) += alpha * ( - p / hi + q * hi / 6.0);
184 }
185 else if (row == F.size())
186 {
187     F( row) += beta * ( - p / hi + q * hi / 6.0);
188 }
189
190 F( row) = fim1 * hi / 6.0
191          + fi * ( hi + hip1) / 3.0
192          + fip1 * hip1 / 6.0;
193 return;
194 }
195
196 double forcingFactory::analytic_5( double x)
197 {
198     double f = 0.0;
199     f = -6.0 + 2.0 * (x*x - 3.0);
200     return f;
201 }
202
203 void forcingFactory::pXq0force( int row)
204 {
205     double hi    = m->getElem( row).getLength();
206     double hip1  = m->getElem( row + 1).getLength();
207
208     double xim1 = m->getElem( row).getLeftPos();

```

```

209 double xi    = m->getElem( row ).getRightPos();
210 double xip1  = m->getElem( row + 1 ).getLeftPos();
211
212 double fim1 = analytic_pXq0( xim1 );
213 double fi   = analytic_pXq0( xi );
214 double fip1 = analytic_pXq0( xip1 );
215
216 F( row ) = fim1 * hi / 6.0
217           + fi * ( hi + hip1 ) / 3.0
218           + fip1 * hip1 / 6.0;
219 return;
220 }
221
222 double forcingFactory::analytic_pXq0( double x)
223 {
224     double f = 0.0;
225     f += std::sin( 5.0 * x ) * ( -9.0 * x - 1.0 );
226     f += std::cos( 5.0 * x ) * ( -17.0 * x*x - x + 6.0 );
227     f += std::exp( x ) * ( - 3.0 * x*x*x - 15.0 * x*x - 15.0 *
        * x + 6.0 );
228     return f;
229 }
230
231 double forcingFactory::analytic_p3q2( double x)
232 {
233     double f = 0.0;
234     f += (17.0 * x * (x-1.0) - 6.0) * std::sin( 5.0 * x );
235     f += -3.0 * x * (x + 11) * std::exp( x );
236     f += -18.0 * (2.0 * x - 1) * cos( 5.0 * x );
237     return f;
238 }

```

A.9 forcing.hpp

```

1 #ifndef FORCING_HPP
2 #define FORCING_HPP
3
4 #include "mesh1D.hpp"
5 #include "eig_wrap.hpp"
6

```

```

7 class forcingFactory
8 {
9     public:
10    // Constructor
11    forcingFactory( mesh1D* mesh, int caseNumber_);
12
13    // Create the forcing vector for the problem
14    void create_forcing();
15
16    // Get the forcing vector for the problem
17    VectorXd getForcing();
18
19    private:
20    // Assign the row value of the forcing vector
21    // for the particular case number
22    void assign_force( int row);
23
24    // Assign forcing component for p=3, q=2 case ( $\leftarrow$ 
25    // caseNumber 1)
26    void p3q2force( int row);
27
28    // Assign forcing component for p=x+1, q=0 case ( $\leftarrow$ 
29    // caseNumber 2)
30    void pXq0force( int row);
31
32    // Evaluate given force function for p3q2 case
33    double analytic_p3q2( double x);
34
35    // Evaluate given force function for pXq0 case
36    double analytic_pXq0( double x);
37
38    // The forcing vector
39    VectorXd F;
40
41    // The case number for this problem
42    int caseNumber;
43
44    // Pointer to the mesh
45    mesh1D* m;

```

```

45 // Number of dofs
46 int numDofs;
47
48 // Force assignment for case 3
49 void case3force( int row);
50
51 // Analytical force for case 3
52 double analytic_3( double x);
53 // Analytical force for case 4
54 double analytic_4( double x);
55 // Analytical force for case 5
56 double analytic_5( double x);
57
58 // Force assignment for case 4
59 void case4force( int row);
60
61 // Force assignment for case 5
62 void case5force( int row);
63 };
64 #endif

```

A.10 mesh1D.cpp

```

1 #include "mesh1D.hpp"
2
3 #include "iostream"
4 #include <cstdlib>
5
6 mesh1D::mesh1D( double oddSize_, double evenSize_, int ↵
    NplusOne)
7 {
8     numElems = NplusOne;
9     oddSize = oddSize_;
10    evenSize = evenSize_;
11    constructElems();
12 }
13
14 mesh1D::~~mesh1D()
15 {
16     for( int i = 0; i < elements.size(); i++)

```

```

17     {
18         delete (elements[i]);
19     }
20     elements.clear();
21 }
22
23 elem mesh1D::getElem(int i)
24 {
25     if ( ( i >= numElems) || ( i < 0))
26     {
27         std::cout
28             << "Attempted to access out of bounds element."
29             << "Element number" << i << " requested."
30             << "Elements are numbered 0 to" << numElems-1 << "\n";
31         std::abort();
32     }
33     return *elements[i];
34 }
35
36 void mesh1D::constructElems()
37 {
38     numNodes = 1;
39     for( int i = 0; i < numElems; i++)
40     {
41         constructElement( i);
42         numNodes++;
43     }
44     return;
45 }
46
47 void mesh1D::constructElement( int i)
48 {
49     // First figure out where the last element ended.
50     double leftPos = 0.0;
51     if ( i != 0)
52     {
53         leftPos = (elements[i-1])->getRightPos();
54     }
55

```

```

56  double DX = 1.0 / numElems;
57  double rightPos = leftPos;
58
59  // Create element based on even or odd number
60  if ( i%2 == 0)
61  { // This is an even element.
62    rightPos += evenSize * DX;
63  }
64  else
65  { // This is an odd element.
66    rightPos += oddSize * DX;
67  }
68
69  elem* tmpElem = new elem( leftPos , rightPos);
70
71  elements.push_back( tmpElem);
72
73  return;
74 }
75
76 int mesh1D::getNumElems()
77 {
78   return numElems;
79 }
80
81 int mesh1D::getNumNodes()
82 {
83   return numNodes;
84 }

```

A.11 mesh1D.hpp

```

1  #ifndef MESH1D_HPP
2  #define MESH1D_HPP
3
4  #include <vector>
5  #include "element1D.hpp"
6
7  // A structure to define and interact with a ↵
   representative

```

```

8 // one dimensional mesh. Mesh assumed to be structured as
9 // defined in homework assignment.
10 class mesh1D
11 {
12     public:
13         // Constructor
14         mesh1D( double oddSize, double evenSize, int NplusOne←
15             );
16
17         // Destructor
18         ~mesh1D();
19
20         // Get the number of elements in the mesh
21         int getNumElems();
22
23         // Get the number of nodes in the mesh
24         int getNumNodes();
25
26         // Get the ith element of the mesh
27         elem getElem( int i);
28
29     private:
30
31         // The number of elements in the mesh
32         int numElems;
33
34         // The number of nodes in the mesh
35         int numNodes;
36
37         // The array of elements
38         std::vector<elem*> elements;
39
40         // Element sizes
41         double oddSize;
42         double evenSize;
43
44         // Constructor of the elements
45         void constructElems();
46
47         // Constructor of a single element

```



```

47     void constructElement( int i );
48
49 };
50 #endif

```

A.12 stiffness.cpp

```

1  #include "stiffness.hpp"
2
3  #include <iostream>
4  #include <cmath>
5
6  springFactory::springFactory( mesh1D* mesh, int ↵
    caseNumber_)
7  {
8      m = mesh;
9      caseNumber = caseNumber_;
10     int numNodes = m->getNumNodes();
11     // Assuming that only the interior nodes are
12     // degrees of freedom
13     numDofs = numNodes - 2;
14     K = MatrixXd::Zero( numDofs, numDofs);
15 }
16
17 void springFactory::create_stiffness()
18 {
19     for( int i = 0; i < numDofs; i++)
20     {
21         for( int j = 0; j < numDofs; j++)
22         {
23             assign_stiffness( i, j);
24         }
25     }
26
27     std::cout << "K_=" << std::endl;
28     std::cout << K << std::endl;
29     return ;
30 }
31
32 void springFactory::assign_stiffness( int row, int col)

```

```

33 {
34     if ( ( caseNumber == 1) ||
35         ( caseNumber == 3) ||
36         ( caseNumber == 4) ||
37         ( caseNumber == 5))
38     { // A caseNumber of 1 is for the p = 3, q = 2 problem
39       p3q2Stiffness( row, col);
40     }
41     else if ( caseNumber == 2)
42     { // A casenumber of 2 is for the p = 1+x, q=0 problem
43       pXq0Stiffness( row, col);
44     }
45     else
46     {
47         std::cout << "Unrecognized_caseNumber_=" << ↵
48         caseNumber << std::endl;
49         std::abort();
50     }
51     return;
52 }
53 MatrixXd springFactory::getStiffness()
54 {
55     return K;
56 }
57
58 void springFactory::p3q2Stiffness( int row, int col)
59 {
60     double p  = 3.0;
61     double q  = 2.0;
62
63     double hi  = m->getElem( row).getLength();
64     double hip1 = m->getElem( row + 1).getLength();
65
66     if( row == col)
67     {
68         // On the diagonal
69         K( row, col) = p * (1.0/hi + 1.0/hip1) + q/3.0 * (hi ↵
70         + hip1);
71     }

```

```

71  else if ( std::abs( row - col) == 1.0)
72  {
73      if ( (row - col) < 0.0)
74      {
75          // Inside of diagonal
76          K( row, col) = - p / hi + q * hi / 6.0;
77      }
78      else if ( (row - col) > 0.0)
79      {
80          // Inside of diagonal
81          K( row, col) = - p / hip1 + q * hip1 / 6.0;
82      }
83  }
84  return;
85  }
86
87  void springFactory::pXq0Stiffness( int row, int col)
88  {
89      double hi    = m->getElem( row).getLength();
90      double hip1  = m->getElem( row + 1).getLength();
91
92      double xim1 = m->getElem( row).getLeftPos();
93      double xi   = m->getElem( row).getRightPos();
94      double xip1 = m->getElem( row).getRightPos();
95
96      if( row == col)
97      {
98          // On the diagonal
99          double tmp1 = ( xi * xi - xim1 * xim1) / 2.0;
100         tmp1 += hi;
101         tmp1 /= (hi * hi);
102         double tmp2 = ( xip1 * xip1 - xi * xi) / 2.0;
103         tmp2 += hip1;
104         tmp2 /= (hip1 * hip1);
105         K( row, col) = tmp1 + tmp2;
106     }
107     else if ( std::abs( row - col) == 1.0)
108     {
109         if ( (row - col) < 0.0)
110         {

```

```

111     // Inside of diagonal
112     double tmp = ( xi * xi - xim1 * xim1) / 2.0;
113     tmp += hi;
114     tmp /= -1.0 * (hi * hi);
115     K( row, col) = tmp;
116 }
117 else if ( (row - col) > 0.0)
118 {
119     // Inside of diagonal
120     double tmp = ( xip1 * xip1 - xi * xi) / 2.0;
121     tmp += hip1;
122     tmp /= -1.0 * (hip1 * hip1);
123     K( row, col) = tmp;
124 }
125 }
126 return;
127 }

```

A.13 stiffness.hpp

```

1 #ifndef STIFFNESS_HPP
2 #define STIFFNESS_HPP
3
4 #include "mesh1D.hpp"
5 #include "eig_wrap.hpp"
6
7 class springFactory
8 {
9     public:
10     // Constructor
11     springFactory( mesh1D* mesh, int caseNumber_);
12
13     // Create the stiffness matrix for the problem
14     void create_stiffness();
15
16     // Get the stiffness matrix for the problem
17     MatrixXd getStiffness();
18
19     private:

```

```

20 // Assign the row and column value of the stiffness ↵
    matrix
21 // for the particular case number
22 void assign_stiffness( int row, int col);
23
24 // Assign stiffness component for p=3, q=2 case (↵
    caseNumber 1)
25 void p3q2Stiffness( int row, int col);
26
27 // Assign stiffness component for p=x+1, q=0 case (↵
    caseNumber 2)
28 void pXq0Stiffness( int row, int col);
29
30 // The stiffness matrix
31 MatrixXd K;
32
33 // The case number for this problem
34 int caseNumber;
35
36 // Pointer to the mesh
37 mesh1D* m;
38
39 // Number of dofs
40 int numDofs;
41 };
42 #endif

```

A.14 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  
14 #endif
```