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.

$$-(p(x)u'(x))' + q(x)u(x) = f(x) \tag{1}$$
$$u(0) = \alpha, u(1) = \beta \tag{2}$$

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.

$$u(x) \in C^2[0,1] \tag{3}$$
$$f, q \in C^0[0,1] \qquad q \geq 0 \tag{4}$$
$$p \in C^1[0,1] \qquad p > 0 \tag{5}$$

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.

$$\Delta x = \frac{1}{N+1} \tag{6}$$

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.

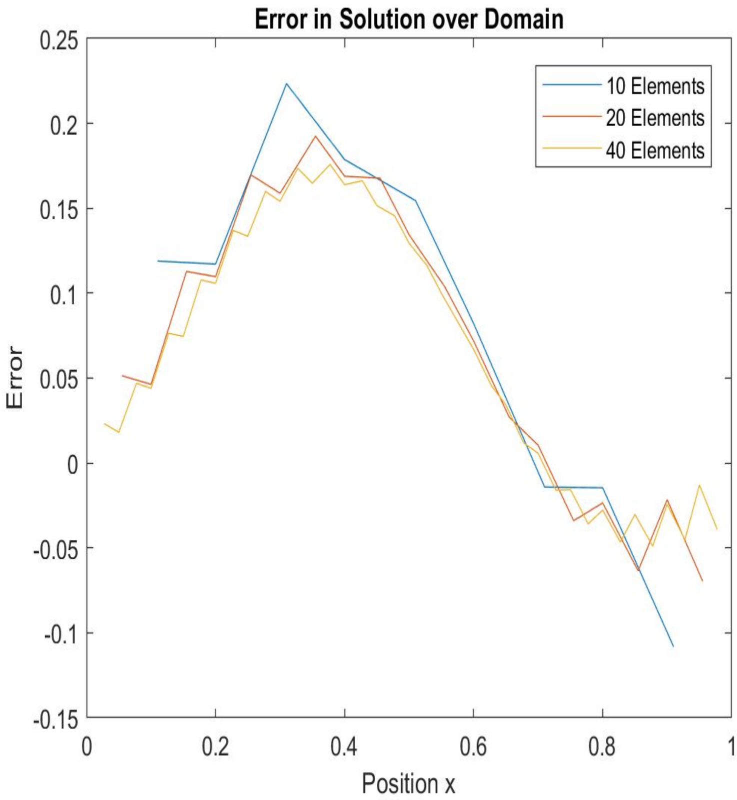

Figure 1: Error in solution over domain of problem.

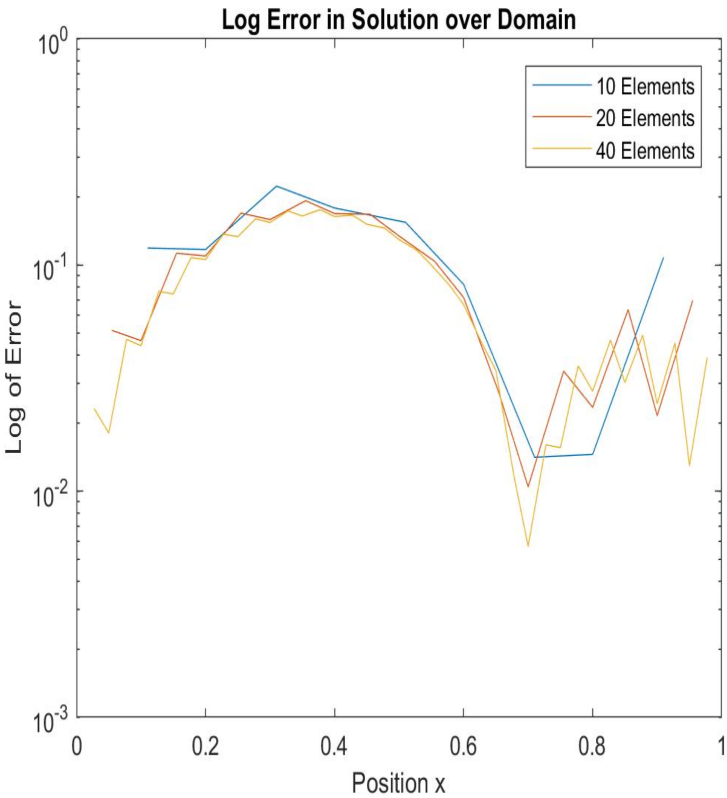

Figure 2: Log of error in solution over domain of problem.

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 was 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

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

int main( int argc, char** argv)
{
  // Define Mesh paramters, testing size
  double oddSize          = 0.9;
  double evenSize         = 1.1;
  int    NplusOneArray[]  = {10, 20, 40, 80, 160, 320};
  //int    NplusOneArray[] = {10, 10, 10, 10, 10, 10};
  int    numTests         = 6;

  for ( int i = 1; i < 6; i++)
  {
    drive_problem( oddSize, evenSize, NplusOneArray, numTests, i);
  }

  return 0;
}
```

A.2 driver.cpp

```
#include "driver.hpp"

void drive_problem( double oddSize,
                    double evenSize,
                    int*   NplusOneArray,
                    int    numTests,
                    int    caseNumber)
{
  std::vector<VectorXd> solutions;
  std::vector<mesh1D*> meshes;

  for( int i = 0; i < numTests; i++)
  {
```

```
    // First construct the mesh
    int NplusOne = NplusOneArray[i];
    mesh1D* mesh = new mesh1D( oddSize, evenSize, NplusOne);
    meshes.push_back( mesh);

    // Second construct stiffness matrix
    springFactory* sf = new springFactory( mesh, caseNumber);
    sf->create_stiffness();
    MatrixXd K = sf->getStiffness();

    // Third construct forcing vector
    forcingFactory* ff = new forcingFactory( mesh, caseNumber);
    ff->create_forcing();
    VectorXd F = ff->getForcing();

    // Calculate and store solution
    solutions.push_back( K.colPivHouseholderQr().solve( F));

    delete ff;
    delete sf;
  }
  errorCalcs( solutions, meshes, NplusOneArray, caseNumber);
  solutions.clear();
  for (int i = 0; i < meshes.size(); i++)
  {
    delete meshes[ i];
  }
  meshes.clear();

  return;
}
```

A.3 driver.hpp

```
#ifndef DRIVER_HPP
```

```
#define DRIVER_HPP

#include "eig_wrap.hpp"
#include "mesh1D.hpp"
#include "stiffness.hpp"
#include "forcing.hpp"
#include "errorCalcs.hpp"

void drive_problem( double oddSize,
                    double evenSize,
                    int*   NplusOneArray,
                    int    numTests,
                    int    caseNumber);

#endif
```

A.4 element1D.cpp

```
#include "element1D.hpp"

#include <cstdlib>
#include <iostream>

elem::elem( double leftPos_, double rightPos_)
{
  rightPos = rightPos_;
  leftPos = leftPos_;

  length = rightPos - leftPos;

  if ( length <= 0.0)
  {
    std::cout << "Degenerate element construction attempted. "
              << "Attempted length = "
              << length << std::endl;

    std::abort();
  }
}
```

```cpp

elem::~elem()
{

}

double elem::getLength()
{
  return length;
}

double elem::getLeftPos()
{
  return leftPos;
}

double elem::getRightPos()
{
  return rightPos;
}
```

A.5 element1D.hpp

```cpp
#ifndef ELEMENT1D_HPP
#define ELEMENT1D_HPP

class elem
{

  public:
    // Gets the length of an elment.
    double getLength();

    // Gets the left Node Position.
    double getLeftPos();

    // Gets the right Node Position.
    double getRightPos();

    // Constructor
```

```
    elem( double leftPos_, double rightPos_);

    // Destructor
    ~elem();

  private:
    // The length of the element.
    double length;

    // The coordinate of the left node.
    double leftPos;

    // The coordinate of the right node.
    double rightPos;
};

#endif
```

A.6 errorCalcs.cpp

```
#include "errorCalcs.hpp"

#include <vector>
#include <cmath>
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <cstdlib>

void errorCalcs( std::vector<VectorXd> solutions,
                 std::vector<mesh1D*> meshes,
                 int* NA,
                 int caseNumber)
{
  EC* ec = new EC( solutions, meshes, NA, caseNumber);

  ec->create_analyticals();
  ec->calc_errors();
  ec->write();
```

```

  delete ec;
  return;
}

void EC::create_analyticals()
{
  for( int i = 0; i < ms.size(); i++)
  {
    int numDofs = na[i] - 1;
    VectorXd U = VectorXd::Zero( numDofs);
    // For each mesh, we need to calculate the exact nodal
    //  solutions to use a the analytic solution
    for( int j = 0; j < numDofs; j++)
    {
      double xj = ms[i]->getElem(j).getRightPos();
      if (( cn == 1) || ( cn == 2) )
      {
        // This is the p3q2 case and the p(x), q0 case
        U( j) = xj * (xj - 1.0) * ( std::sin( 5.0 * xj) + 3.0 * std::exp( xj));
      }
      else if ( cn == 3)
      {
        U( j) = 4.0;
      }
      else if ( cn == 4)
      {
        U( j) = xj - 2.0;
      }
      else if ( cn == 5)
      {
        U( j) = xj * xj - 3.0;
      }
    }
    us.push_back( U);
  }
  return;
}
```

```
void EC::calc_errors()
{
  getDiff();
  L_2();
  L_inf();
  Energy();
  return;
}

void EC::getDiff()
{
  for( int i = 0; i < us.size(); i++)
  {
    ehs.push_back( sols[i] - us[i] );
  }
  return;
}

double EC::e_interp( double ea,
                     double eb,
                     double xa,
                     double xb,
                     double h,
                     double x)
{
  return ea * (xb - x) / h + eb * (x - xa) / h;
}

void EC::L_2()
{
  double tmp = 0.0;
  for( int i = 0; i < ehs.size(); i++)
  {
    VectorXd e = ehs[i];
    mesh1D* m = ms[i];

    for( int j = 0; j < e.size(); j++)
    {
      double hj = m->getElem( j ).getLength();
```

```
      double xa = m->getElem( j ).getLeftPos();
      double xb = m->getElem( j ).getRightPos();
      double ea = 0.0;
      if( j != 0)
      {
        double ea = e(j-1);
      }
      double eb = e(j);
      // M perscribed in problem handout
      int M = 3;
      for ( int m = 1; m <= M; m++)
      {
        double x = xa + hj * m / M;
        double e_intp = e_interp( ea, eb, xa, xb, hj, x);
        tmp = e_intp * e_intp;
      }
      tmp *=  hj / M;
    }
    L2.push_back( std::sqrt( tmp));
  }

  std::cout << "L2 Error = " << std::endl;
  for ( int i =0; i < L2.size(); i++)
  {
    std::cout << L2[i] << std::endl;
  }

  for ( int i = 0; i < L2.size(); i++)
  {
    if ( i == 0)
    {
      L_2_order.push_back( 0.0);
    }
    else
    {
      double top    = std::log( L2[ i-1] / L2[i]);
      double bottom = std::log( 2.0);
      double tmp    = top / bottom;
      L_2_order.push_back( tmp);
    }
```

```
  }
  return;
}

void EC::L_inf()
{
  double tmp = 0.0;
  for( int i = 0; i < ehs.size(); i++)
  {
    VectorXd e = ehs[i];
    mesh1D* m = ms[i];

    for( int j = 0; j < e.size(); j++)
    {
      double hj = m->getElem( j).getLength();
      double xa = m->getElem( j).getLeftPos();
      double xb = m->getElem( j).getRightPos();
      double ea = 0.0;
      if( j != 0)
      {
        double ea = e(j-1);
      }
      double eb = e(j);
      // M perscribed in problem handout
      int M = 3;
      for ( int m = 1; m <= M; m++)
      {
        double x = xa + hj * m / M;
        double e_intp = std::abs( e_interp( ea, eb, xa, xb, hj, x) );
        if ( tmp < e_intp)
        {
          tmp = e_intp;
        }
      }
    }
    Linf.push_back( tmp);
    tmp = 0.0;
  }

```

```
  std::cout << "Linf Error = " << std::endl;
  for ( int i =0; i < Linf.size(); i++)
  {
    std::cout << Linf[i] << std::endl;
  }

  for ( int i = 0; i < Linf.size(); i++)
  {
    if ( i == 0)
    {
      L_inf_order.push_back( 0.0);
    }
    else
    {
      double top    = std::log( Linf[ i-1] / Linf[i]);
      double bottom = std::log( 2.0);
      double tmp    = top / bottom;
      L_inf_order.push_back( tmp);
    }
  }
  return;
}

void EC::Energy()
{
  double tmp = 0.0;
  for( int i = 0; i < ehs.size(); i++)
  {
    VectorXd e = ehs[i];
    mesh1D* m = ms[i];

    for( int j = 0; j < e.size(); j++)
    {
      double hj = m->getElem( j).getLength();
      double xa = m->getElem( j).getLeftPos();
      double xb = m->getElem( j).getRightPos();
      double ea = 0.0;
      if( j != 0)
      {
        double ea = e(j-1);
```

```
        }
        double eb = e(j);
        // M perscribed in problem handout
        int M = 3;
        for ( int m = 1; m <= M; m++)
        {
          // Since we have a linear basis, the
          //  derivatives are constant inside each element
          double dedx = (eb - ea) / hj;
          tmp = dedx * dedx;
        }
        tmp *=  hj / M;
      }
      NRG.push_back( std::sqrt( tmp));
    }

    std::cout << "NRG Error = " << std::endl;
    for ( int i =0; i < NRG.size(); i++)
    {
      std::cout << NRG[i] << std::endl;
    }

    for ( int i = 0; i < NRG.size(); i++)
    {
      if ( i == 0)
      {
        NRG_order.push_back( 0.0);
      }
      else
      {
        double top    = std::log( NRG[ i-1] / NRG[i]);
        double bottom = std::log( 2.0);
        double tmp    = top / bottom;
        NRG_order.push_back( tmp);
      }
    }
    return;
}

void EC::write()
```

```
{
  std::string filename = "errorCalcs_";
  if( cn == 1)
  {
    filename = filename + "1.txt";
  }
  else if ( cn == 2)
  {
    filename = filename + "2.txt";
  }
  else if ( cn == 3)
  {
    filename = filename + "3.txt";
  }
  else if ( cn == 4)
  {
    filename = filename + "4.txt";
  }
  else if ( cn == 5)
  {
    filename = filename + "5.txt";
  }
  else
  {
    std::cout << "Bad case Number " << cn << std::endl;
    std::abort();
  }
  std::ofstream file;
  file << std::scientific;
  file.open( filename.c_str());

  for( int i = 0; i < ehs.size(); i++)
  {
    file << "Error with " << na[i] << " elements:" << std::endl;
    file << ehs[i] << std::endl;
    file << "Abs(Error) with " << na[i] << " elements:" << std::endl;
    file << ehs[i].cwiseAbs() << std::endl;
    file << "Node Locations: " << std::endl;
```

```
    for( int j = 0; j < (ms[i]->getNumElems() - 1); j++)
    {
      file << ms[i]->getElem(j).getRightPos() << std::endl;
    }

    file << "L2 Error: " << std::endl;
    for( int j = 0; j < L2.size(); j++)
    {
      file << L2[j] << std::endl;
    }
    file << "L2 Order: " << std::endl;
    for( int j = 0; j < L_2_order.size(); j++)
    {
      file << L_2_order[j] << std::endl;
    }

    file << "Linf Error: " << std::endl;
    for( int j = 0; j < Linf.size(); j++)
    {
      file << Linf[j] << std::endl;
    }
    file << "Linf Order: " << std::endl;
    for( int j = 0; j < L_inf_order.size(); j++)
    {
      file << L_inf_order[j] << std::endl;
    }

    file << "Energy Error: " << std::endl;
    for( int j = 0; j < NRG.size(); j++)
    {
      file << NRG[j] << std::endl;
    }
    file << "NRG Order: " << std::endl;
    for( int j = 0; j < NRG_order.size(); j++)
    {
      file << NRG_order[j] << std::endl;
    }
  }

```

```
  file.close();

  return;
}

EC::EC( std::vector<VectorXd> solutions,
        std::vector<mesh1D*> meshes,
        int* NA,
        int caseNumber)
{
  sols = solutions;
  ms   = meshes;
  na   = NA;
  cn   = caseNumber;
}
```

A.7 errorCalcs.hpp

```
#ifndef ERRORCALCS_HPP
#define ERRORCALCS_HPP

#include "eig_wrap.hpp"
#include "mesh1D.hpp"

void errorCalcs( std::vector<VectorXd> solutions,
                 std::vector<mesh1D*> meshes,
                 int* NA,
                 int caseNumber);

class EC
{
  public:
    // Constructor
    EC( std::vector<VectorXd> solutions,
        std::vector<mesh1D*> meshes,
        int* NA,
        int caseNumber);

    // Create the analytical solution, dependent on case
    void create_analyticals();
```

```
    // Calculates the L2, L_inf, and Energy; includes convergence
    void calc_errors();

    // Writes calculation results to file
    void write();

  private:
    // vector of FE solution vectors
    std::vector<VectorXd> sols;

    // The Array of NplusOne values
    int* na;

    // The Case Number for this problem
    int cn;

    // The vector of analytical solutions
    std::vector<VectorXd> us;

    // The vector of mesh pointers
    std::vector<mesh1D*> ms;

    // Calculate the actual error as a difference
    void getDiff();

    // A vector of error vectors
    std::vector<VectorXd> ehs;

    // Calculates the L2 norm and order
    void L_2();

    // The vector of L_2 norms
    std::vector<double> L2;

    // The vector of L_2 orders
    std::vector<double> L_2_order;

    // Calculates the L_inf norm and order
```

```
    void L_inf();

    // The vector of L_inf norms
    std::vector<double> Linf;

    // The vector of L_inf orders
    std::vector<double> L_inf_order;

    // Calculates the Energy norm and order
    void Energy();

    // The vector of NRG norms
    std::vector<double> NRG;

    // The vector of NRG orders
    std::vector<double> NRG_order;

    // An intra element interpolation of the error
    double e_interp( double ea,
                     double eb,
                     double xa,
                     double xb,
                     double h,
                     double x);
};

#endif
```

A.8 forcing.cpp

```
#include "forcing.hpp"

#include <cmath>
#include <iostream>

forcingFactory::forcingFactory( mesh1D* mesh, int caseNumber_)
{
  m = mesh;
  caseNumber = caseNumber_;
```

```
  int numNodes = m->getNumNodes();
  // Assuming that only the interior nodes are
  //  degrees of freedom
  numDofs = numNodes - 2;
  F = VectorXd::Zero( numDofs);
}

void forcingFactory::create_forcing()
{
  for ( int i = 0; i < numDofs; i++)
  {
    assign_force( i);
  }

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

  return;
}

VectorXd forcingFactory::getForcing()
{
  return F;
}

void forcingFactory::assign_force( int row)
{
  if ( caseNumber == 1)
  { // A caseNumber of 1 is for the p = 3, q = 2 problem
    p3q2force( row);
  }
  else if ( caseNumber == 2)
  { // A casenumber of 2 is for the p = 1+x, q=0 problem
    pXq0force( row);
  }
  else if ( caseNumber == 3)
  {
    case3force( row);
  }
  else if ( caseNumber == 4)
```

```
  {
    case4force( row);
  }
  else if ( caseNumber == 5)
  {
    case5force( row);
  }
  else
  {
    std::cout << "Unrecognized caseNumber = " << caseNumber << std::endl;
    std::abort();
  }
  return;
}

void forcingFactory::p3q2force( int row)
{
  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row + 1).getLeftPos();

  double fim1 = analytic_p3q2( xim1);
  double fi   = analytic_p3q2( xi);
  double fip1 = analytic_p3q2( xip1);

  F( row) = fim1 * hi / 6.0
          + fi * ( hi + hip1) / 3.0
          + fip1 * hip1 / 6.0;

  return;
}

void forcingFactory::case3force( int row)
{
  double alpha = 4.0;
  double beta  = 4.0;
```

```
  double p      = 3.0;
  double q      = 2.0;

  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row + 1).getLeftPos();

  double fim1 = analytic_3( xim1);
  double fi   = analytic_3( xi);
  double fip1 = analytic_3( xip1);

  if ( row == 1)
  {
    F( row) += alpha * (- p / hi + q * hi / 6.0);
  }
  else if (row == F.size())
  {
    F( row) += beta * (- p / hi + q * hi / 6.0);
  }

  F( row) = fim1 * hi / 6.0
            + fi * ( hi + hip1) / 3.0
            + fip1 * hip1 / 6.0;
  return;
}

double forcingFactory::analytic_3( double x)
{
  return 8.0;
}

void forcingFactory::case4force( int row)
{
  double alpha = -2.0;
  double beta  = -1.0;
  double p      = 3.0;
  double q      = 2.0;
```

```

  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row + 1).getLeftPos();

  double fim1 = analytic_4( xim1);
  double fi   = analytic_4( xi);
  double fip1 = analytic_4( xip1);

  if ( row == 1)
  {
    F( row) += alpha * (- p / hi + q * hi / 6.0);
  }
  else if (row == F.size())
  {
    F( row) += beta * (- p / hi + q * hi / 6.0);
  }

  F( row) = fim1 * hi / 6.0
            + fi * ( hi + hip1) / 3.0
            + fip1 * hip1 / 6.0;
  return;
}

double forcingFactory::analytic_4( double x)
{
  double f = 0.0;
  f = 2.0 * x - 4.0;
  return f;
}

void forcingFactory::case5force( int row)
{
  double alpha = -3.0;
  double beta  = -2.0;
  double p     = 3.0;
  double q     = 2.0;
```

```

  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row + 1).getLeftPos();

  double fim1 = analytic_5( xim1);
  double fi   = analytic_5( xi);
  double fip1 = analytic_5( xip1);

  if ( row == 1)
  {
    F( row) += alpha * (- p / hi + q * hi / 6.0);
  }
  else if (row == F.size())
  {
    F( row) += beta * (- p / hi + q * hi / 6.0);
  }

  F( row) = fim1 * hi / 6.0
          + fi * ( hi + hip1) / 3.0
          + fip1 * hip1 / 6.0;
  return;
}

double forcingFactory::analytic_5( double x)
{
  double f = 0.0;
  f = -6.0 + 2.0 * (x*x - 3.0);
  return f;
}

void forcingFactory::pXq0force( int row)
{
  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
```

```
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row + 1).getLeftPos();

  double fim1 = analytic_pXq0( xim1);
  double fi   = analytic_pXq0( xi);
  double fip1 = analytic_pXq0( xip1);

  F( row) = fim1 * hi / 6.0
          + fi * ( hi + hip1) / 3.0
          + fip1 * hip1 / 6.0;
  return;
}

double forcingFactory::analytic_pXq0( double x)
{
  double f = 0.0;
  f += std::sin( 5.0 * x) * ( -9.0 * x - 1.0);
  f += std::cos( 5.0 * x) * ( -17.0 * x*x - x + 6.0);
  f += std::exp( x) * (- 3.0 * x*x*x - 15.0 * x*x - 15.0 * x +6.0);
  return f;
}

double forcingFactory::analytic_p3q2( double x)
{
  double f = 0.0;
  f += (17.0 * x * (x-1.0) - 6.0) * std::sin( 5.0 * x);
  f += -3.0 * x * (x + 11) * std::exp( x);
  f += -18.0 * (2.0 * x - 1) * cos( 5.0 * x);
  return f;
}
```

A.9 forcing.hpp

```
#ifndef FORCING_HPP
#define FORCING_HPP

#include "mesh1D.hpp"
#include "eig_wrap.hpp"

```

```cpp
class forcingFactory
{
  public:
  // Constructor
  forcingFactory( mesh1D* mesh, int caseNumber_);

  // Create the forcing vector for the problem
  void create_forcing();

  // Get the forcing vector for the problem
  VectorXd getForcing();

  private:
  // Assign the row value of the forcing vector
  // for the particular case number
  void assign_force( int row);

  // Assign forcing component for p=3, q=2 case (caseNumber 1)
  void p3q2force( int row);

  // Assign forcing component for p=x+1, q=0 case (caseNumber 2)
  void pXq0force( int row);

  // Evaluate given force function for p3q2 case
  double analytic_p3q2( double x);

  // Evaluate given force function for pXq0 case
  double analytic_pXq0( double x);

  // The forcing vector
  VectorXd F;

  // The case number for this problem
  int caseNumber;

  // Pointer to the mesh
  mesh1D* m;

```

```
  // Number of dofs
  int numDofs;

  // Force assignment for case 3
  void case3force( int row);

  // Analytical force for case 3
  double analytic_3( double x);
  // Analytical force for case 4
  double analytic_4( double x);
  // Analytical force for case 5
  double analytic_5( double x);

  // Force assignment for case 4
  void case4force( int row);

  // Force assignment for case 5
  void case5force( int row);
};
#endif
```

A.10 mesh1D.cpp

```
#include "mesh1D.hpp"

#include "iostream"
#include <cstdlib>

mesh1D::mesh1D( double oddSize_, double evenSize_, int ↩
    NplusOne)
{
  numElems = NplusOne;
  oddSize = oddSize_;
  evenSize = evenSize_;
  constructElems();
}

mesh1D::~mesh1D()
{
  for( int i = 0; i < elements.size(); i++)
```

```
  {
   delete (elements[i]);
  }
  elements.clear();
}

elem mesh1D::getElem(int i)
{
  if ( ( i >= numElems) || ( i < 0))
  {
    std::cout
      << "Attempted to access out of bounds element. "
      << "Element number " << i << " requested. "
      << "Elements are numbered 0 to " << numElems-1 << std::endl;
    std::abort();
  }
  return *elements[i];
}

void mesh1D::constructElems()
{
  numNodes = 1;
  for( int i = 0; i < numElems; i++)
  {
    constructElement( i);
    numNodes++;
  }
  return;
}

void mesh1D::constructElement( int i)
{
  // First figure out where the last element ended.
  double leftPos = 0.0;
  if ( i != 0)
  {
    leftPos = (elements[i-1])->getRightPos();
  }

```

```
  double DX = 1.0 / numElems;
  double rightPos = leftPos;

  // Create element based on even or odd number
  if ( i%2 == 0)
  { // This is an even element.
    rightPos += evenSize * DX;
  }
  else
  { // This is an odd element.
    rightPos += oddSize * DX;
  }

  elem* tmpElem = new elem( leftPos, rightPos);

  elements.push_back( tmpElem);

  return;
}

int mesh1D::getNumElems()
{
  return numElems;
}

int mesh1D::getNumNodes()
{
  return numNodes;
}
```

A.11 mesh1D.hpp

```
#ifndef MESH1D_HPP
#define MESH1D_HPP

#include <vector>
#include "element1D.hpp"

// A structure to define and interact with a ↩
    representative
```

```
// one dimensional mesh. Mesh assumed to be structured as
// defined in homework assignment.
class mesh1D
{
  public:
    // Constructor
    mesh1D( double oddSize, double evenSize, int NplusOne);

    // Destructor
    ~mesh1D();

    // Get the number of elements in the mesh
    int getNumElems();

    // Get the number of nodes in the mesh
    int getNumNodes();

    // Get the ith element of the mesh
    elem getElem( int i);

  private:

    // The number of elements in the mesh
    int numElems;

    // The number of nodes in the mesh
    int numNodes;

    // The array of elements
    std::vector<elem*> elements;

    // Element sizes
    double oddSize;
    double evenSize;

    // Constructor of the elements
    void constructElems();

    // Constructor of a single element
```

```
    void constructElement( int i);

};
#endif
```

A.12 stiffness.cpp

```
#include "stiffness.hpp"

#include <iostream>
#include <cmath>

springFactory::springFactory( mesh1D* mesh, int ↩
    caseNumber_)
{
  m = mesh;
  caseNumber = caseNumber_;
  int numNodes = m->getNumNodes();
  // Assuming that only the interior nodes are
  //  degrees of freedom
  numDofs = numNodes - 2;
  K = MatrixXd::Zero( numDofs, numDofs);
}

void springFactory::create_stiffness()
{
  for( int i = 0; i < numDofs; i++)
  {
    for( int j = 0; j < numDofs; j++)
    {
      assign_stiffness( i, j);
    }
  }

  std::cout << "K = " << std::endl;
  std::cout << K << std::endl;
  return ;
}

void springFactory::assign_stiffness( int row, int col)
```

```
{
  if ( ( caseNumber == 1) ||
       ( caseNumber == 3) ||
       ( caseNumber == 4) ||
       ( caseNumber == 5))
  { // A caseNumber of 1 is for the p = 3, q = 2 problem
    p3q2Stiffness( row, col);
  }
  else if ( caseNumber == 2)
  { // A casenumber of 2 is for the p = 1+x, q=0 problem
    pXq0Stiffness( row, col);
  }
  else
  {
    std::cout << "Unrecognized caseNumber = " << caseNumber << std::endl;
    std::abort();
  }
  return;
}

MatrixXd springFactory::getStiffness()
{
  return K;
}

void springFactory::p3q2Stiffness( int row, int col)
{
  double p  = 3.0;
  double q  = 2.0;

  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  if( row == col)
  {
    // On the diagonal
    K( row, col) = p * (1.0/hi + 1.0/hip1) + q/3.0 * (hi + hip1);
  }
```

```
  else if ( std::abs( row - col) == 1.0)
  {
    if ( (row - col) < 0.0)
    {
      // Inside of diagonal
      K( row, col) = - p / hi + q * hi / 6.0;
    }
    else if ( (row - col) > 0.0)
    {
      // Inside of diagonal
      K( row, col) = - p / hip1 + q * hip1 / 6.0;
    }
  }
  return;
}

void springFactory::pXq0Stiffness( int row, int col)
{
  double hi   = m->getElem( row).getLength();
  double hip1 = m->getElem( row + 1).getLength();

  double xim1 = m->getElem( row).getLeftPos();
  double xi   = m->getElem( row).getRightPos();
  double xip1 = m->getElem( row).getRightPos();

  if( row == col)
  {
    // On the diagonal
    double tmp1 = ( xi * xi - xim1 * xim1) / 2.0;
    tmp1 += hi;
    tmp1 /= (hi * hi);
    double tmp2 = ( xip1 * xip1 - xi * xi) / 2.0;
    tmp2 += hip1;
    tmp2 /= (hip1 * hip1);
    K( row, col) = tmp1 + tmp2;
  }
  else if ( std::abs( row - col) == 1.0)
  {
    if ( (row - col) < 0.0)
    {
```

```
      // Inside of diagonal
      double tmp = ( xi * xi - xim1 * xim1) / 2.0;
      tmp += hi;
      tmp /= -1.0 * (hi * hi);
      K( row, col) = tmp;
    }
    else if ( (row - col) > 0.0)
    {
      // Inside of diagonal
      double tmp = ( xip1 * xip1 - xi * xi) / 2.0;
      tmp += hip1;
      tmp /= -1.0 * (hip1 * hip1);
      K( row, col) = tmp;
    }
  }
  return;
}
```

A.13 stiffness.hpp

```
#ifndef STIFFNESS_HPP
#define STIFFNESS_HPP

#include "mesh1D.hpp"
#include "eig_wrap.hpp"

class springFactory
{
  public:
  // Constructor
  springFactory( mesh1D* mesh, int caseNumber_);

  // Create the stiffness matrix for the problem
  void create_stiffness();

  // Get the stiffness matrix for the problem
  MatrixXd getStiffness();

  private:
```

```
    // Assign the row and column value of the stiffness ↩
        matrix
    // for the particular case number
    void assign_stiffness( int row, int col);

    // Assign stiffness component for p=3, q=2 case (↩
        caseNumber 1)
    void p3q2Stiffness( int row, int col);

    // Assign stiffness component for p=x+1, q=0 case (↩
        caseNumber 2)
    void pXq0Stiffness( int row, int col);

    // The stiffness matrix
    MatrixXd K;

    // The case number for this problem
    int caseNumber;

    // Pointer to the mesh
    mesh1D* m;

    // Number of dofs
    int numDofs;
};
#endif
```

A.14 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;

#endif
```