ME 257: Finite element method
Homework 5.

Instructions:

- This HW is worth 50 points.
- Submit your code as a .cpp file and a summary of your results as a pdf file.
- Upload your zipped folder at this link by 9 PM on Saturday, May 29: `https://www.dropbox.com/request/xD05Dm9dhqhePKcUaZmk`
- Document your code.
- Late submissions incur a penalty of 1 point/hour.
- Make use of the help sessions after each lecture.
- **Start early.** Do not expect to be able to complete this HW in a day. This is practically a mini project.
- **Do not plagiarize**. Your submissions will be run through JPlag. You cannot beat the software.

In this HW, you will implement a finite element method for approximating the 2D model problem we covered in class:

$$-\text{Div}(\nabla u) = f \quad \text{over } \Omega \subset \mathbb{R}^2,$$

where $\Omega = [-1, 1] \times [-1, 1]$ is a square-shaped domain. Assume homogeneous boundary conditions along $\partial\Omega$. Follow the steps outlined.

Programming (25 points)

1. *Creating a mesh:* A set of triangle meshes over Ω is provided along with this HW. Each mesh is specified by:

   - A file `coordinates.dat` containing a list of nodal coordinates. The first line contains the x and y coordinates of the first vertex, and so on.
   - A file `connectivity.dat` contianing a list of element connectivities. The first line contains the node numbers of the three vertices of the first triangle, and so on. Notice that nodes are numbered starting from 0.

   The following function is provided to you in the header file `ReadMesh.h` to read in a mesh:

```
void ReadMesh(std::string coord_filename, std::vector<double>& coordinates,
              std::string conn_filename, std::vector<int>& connectivity);
```

Include the header file in your code to use it. Sample usage:

```
ReadMesh("mesh1/coordinates.dat", coordinates,
         "mesh1/connectivity.dat", connectivity);
```

2. *Shape functions and derivatives over the parametric element:* Declare and define:

```
void GetShapeFunction(const double* xi, int a, double& val, double* dval);
```

Details:

```
$\boldsymbol{\xi}$: Input. Coordinates in the parametric triangle, i.e., the pair $(\xi, \eta)$
a: Input. Shape function number, possible values 0, 1, 2
val: Output. Value of the shape function $\hat{N}_a(\boldsymbol{\xi})$
dval: Output. Derivatives of the shape function $\nabla\hat{N}_a(\boldsymbol{\xi})$
```

3. *Shape functions and derivatives in the physical element:* Declare and define:

```
void GetShapeFunction(const double* xi, int a, const double* nodal_coords,
                      double& val, double& dval);
```

```
xi: Input. Coordinates of a quadrature point in the parametric triangle
a: Input. Local shape function number in an element.  Possible values: 0, 1, 2
nodal_coords: Input. Cartesian coordinates of the 3 nodes of a triangle element
val: Output. Value of the shape function $N_a(\varphi(\boldsymbol{\xi}))$
dval: Output. Derivative of the shape function $\nabla N_a(\varphi(\boldsymbol{\xi}))$
```

Suggested pseudo-code for this routine:

```
– Compute shape functions & derivatives over the parametric element:
      GetShapeFunction(xi, a, $\hat{N}_a$, $\nabla\hat{N}_a$)
– Set shape function value $N_a(\varphi(\boldsymbol{\xi})) = \hat{N}_a(\boldsymbol{\xi})$
– Evaluate $\nabla\varphi$ for the given set of nodal coordinate
– Evaluate $(\nabla\varphi)^{-T}$
– Transform derivatives: $\nabla N_a = (\nabla\varphi)^{-T}\nabla\hat{N}_a$
```

Take advantage of the fact that the Jacobian $\nabla\varphi$ is a constant. Hence, you can directly code the 3x3 matrix $(\nabla\varphi)^{-T}$ in terms of the nodal coordinates.

4. *Element stiffness matrix:* Declare and define:

```
void GetElementStiffnessMatrix(const double* nodal_coords,
                               double kmat[][3]);

nodal_coords: Input. Nodal coordinates of the element
kmat: Output. Element stiffness matrix. Size 3x3
```

Suggested pseudo-code for this routine:

```
– For each quadrature point $\boldsymbol{\xi}_q$ in the parametric element and
  each local dof $a$, evaluate & store $\nabla N_a(\varphi(\boldsymbol{\xi}_q))$: GetShapeFunction($\boldsymbol{\xi}_q$, a, $N_a$, $\nabla N_a$)
– Store the quadrature weight over the parametric element as $w_q$

– Initialize $\mathbf{k} = 0$
– Compute $\mathbf{k}_{ab} = \sum_q w_q \, det(\nabla\varphi(\boldsymbol{\xi}_q)) \, \nabla N_a(\varphi(\boldsymbol{\xi}_q)) \cdot \nabla N_b(\varphi(\boldsymbol{\xi}_q))$
```

5. *Element force vector:* Declare and define:

```
void GetElementForceVector(const double* nodal_coords, double* fvec);

nodal_coords: Input. Nodal coordinates of the element
fvec: Output. Element force vector
```

Suggested pseudo-code for this routine:

```
– For each quadrature point $\boldsymbol{\xi}_q$ in the parametric element and
  each local dof $a$, evaluate & store $N_a(\varphi(\boldsymbol{\xi}_q))$: GetShapeFunction($\boldsymbol{\xi}_q$,a,nullptr)
– Store the quadrature weight over the parametric element as $w_q$

– Initialize $\mathbf{f} = 0$
– Compute $\mathbf{f}_a = \sum_q w_q\, det(\nabla\varphi(\boldsymbol{\xi}_q))\, f(\varphi(\boldsymbol{\xi}_q)) N_a(\varphi(\boldsymbol{\xi}_q))$
```

6. *Main routine:* Suggested pseudo-code:

- Read the mesh
- Compute sizes required for the (sparse) global stiffness matrix $\mathbf{K}$ and the global force vector $\mathbf{F}$. Use Eigen's sparse matrix and vector data structures for this purpose.
- For each element, invoke the `GetElementStiffnessMatrix` and the `GetElementForceVector` routines to compute $\mathbf{k}^e$ and $\mathbf{f}^e$
- Assemble elemental contributions into $\mathbf{K}$ and $\mathbf{F}$. Use the element node numbers to define the local to global map.
- Enforce homogeneous Dirichlet BCs along nodes lying on the boundary. As we will discuss during the help session, you can do this by setting

$$\mathbf{F}_n = 0 \quad \text{and} \quad \begin{cases} \mathbf{K}_{n,j} = 0 \text{ if } j \neq n \\ \mathbf{K}_{n,n} = 1 \end{cases}$$

- Compute the LU factorization of $\mathbf{K}$
- Solve $\mathbf{KU} = \mathbf{F}$.

Questions (25 points):

1. To help test the correctness of your code, pick forcing functions f such that the exact solution u belongs to the finite element space. In such cases, check that your solution u_h equals u (upto numerical precision). Report the functions f with which you tested, and the error metric

$$\sum_{\text{nodes } n} |u(x_n) - u_h(x_n)|^2.$$

2. Pick forcing functions f such that the exact solution u can be derived and does not belong to the finite element space. Plot the L^2 norm of the error $(u - u_h)$ for a series of successively refined meshes as a function of the mesh size h on a log-log plot. For this, you will need to code a function that computes the L^2 error as

$$\|u - u_h\| = \left(\int_\Omega (u - u_h)^2\, d\Omega\right)^{1/2}$$

What do you observe?

3. For your choice of functions in part 2, compare the error norms $\|\nabla u - \nabla u_h\|$ with $\|\nabla u - \nabla(\pi u)\|$, where the norm is defined as

$$\|\nabla u - \nabla u_h\| = (\nabla(u - u_h) \cdot \nabla(u - u_h))^{1/2}\,.$$

   What do you expect? And what do you find?

4. Consider the functional

$$\mathrm{I}[v] = \int_\Omega \left(\frac{1}{2}|\nabla v|^2 - fv\right)\, d\Omega.$$

   Code a routine to compute I. Compare values of $\mathrm{I}[u], \mathrm{I}[u_h]$ and $\mathrm{I}[\pi u]$ for a few different choice of f and meshes. What do you expect? What do you find?

5. Identify aspects of the code that can make be made more efficient.