ASE 384P.4/EM 394F/PGE 383/CSE 393F

Coding Project 2

Due Nov 15, 2017

The objective of this assignment is to give you a taste of developing a versatile finite element code and to familiarize you with modern programming practices and paradigms in finite elements. This is a more substantial assignment than coding project 1 and therefore you are encouraged to start early and use the TA and Canvas discussion boards as resources.

1.) Consider the following partial differential equation:

$$\begin{aligned} \nabla \cdot \boldsymbol{u} &= q \quad \text{for} \quad \boldsymbol{x} \in \Omega, \\ p &= g_D \quad \text{for} \quad \boldsymbol{x} \in \Gamma_D, \\ \boldsymbol{u} \cdot \boldsymbol{n} &= g_N \quad \text{for} \quad \boldsymbol{x} \in \Gamma_N, \end{aligned}$$

where

$$\boldsymbol{u} = -\boldsymbol{K} \nabla p.$$

Use the **deal.ii** library (`http://dealii.org`) to write a program that solves the equation for a non-constant diffusion coefficient $\boldsymbol{K}$ and a spatially varying source q in 2D. Observation of good programming practices will enhance the readability of your code and help you remain organized in your work. You may use any of the deal.ii tutorial programs as a starting point. However we require that you strip the file of all comments and replace them with your own as you proceed. Please note that while comments are essential for assessment of your work, keeping them concise and specific is just as crucial. The finished solution should be capable of handing:

- 2D grids
- Non-constant diffusion coefficient $\boldsymbol{K}$
- Spatially varying source term $q(\boldsymbol{x})$
- Non-homogeneous, spatially varying Dirichlet boundary conditions $g_D(\boldsymbol{x})$
- Non-homogeneous, spatially varying Neumann boundary conditions $g_N(\boldsymbol{x})$

Your program should be contained in a single .cc file and must produce point-centered VTK data readable by a visualization package (like ParaView). The following specifications and hints will aid you in your implementation:

- Use the `GridGenerator::hyper_cube` function to generate your mesh. This function will generate either a 2D or 3D grid, depending on the dimensionality of the Triangulation object that is passed to it.
- Use a conjugate gradient solver with a tolerance of 10^{-12}.
- Consider creating a derived class each, for the diffusion coefficient, the forcing function and the Dirichlet boundary values. The base class for these classes would be the Function class defined in deal.ii. Defining virtual member functions of the derived class would be an ideal way of implementing spatially varying quantities. For the diffusion coefficient, deriving a class from TensorFunction will provide maximum flexibility as it would allow the specification of a full 2-tensor.

Set up your code in 2D for a unit square with vertices at $(0, 0)$ and $(1, 1)$. Other inputs are:

$$\boldsymbol{K} = \begin{bmatrix} x_1 & 0 \\ 0 & x_2 \end{bmatrix}$$

$$q = (x_1 + x_2)\cos(x_1 - x_2) \quad \text{for} \quad \boldsymbol{x} \in \Omega = [0, 1]^2$$

$$g_D = \begin{cases} \cos(x_1) & \text{for} \quad \boldsymbol{x} \in (\cdot, 0) \\ \cos(x_1 - 1) & \text{for} \quad \boldsymbol{x} \in (\cdot, 1) \end{cases}$$

$$g_N = \begin{cases} 0 & \text{for} \quad \boldsymbol{x} \in (0, \cdot) \\ \sin(1 - x_2) & \text{for} \quad \boldsymbol{x} \in (1, \cdot) \end{cases}$$

2.) One of the most important steps in the development of predictive modeling software is verification. It usually involves running a prediction scheme for a problem with well understood behavior - such a problem is often called a benchmark. Researchers seek to replicate the known solution using their code and a mismatch indicates the presence of bugs. Once a code can reliably handle a suite of benchmark problems, it is ready to be applied to practical, real-world cases.

The BVP described above admits the exact solution:

$$p = \cos(x_1 - x_2)$$

Further, we can define the L^2 error:

$$||\boldsymbol{e}||_{L^2} = ||p - p_h||_{L^2}.$$

Write a function that uses your computed solution and the analytical solution provided above to compute the L^2 error for a given refinement level.

Record and tabulate the error decreasing element sizes (at least 4 times) and also calculate the rate of convergence. Compare this with the expected convergence rate (conforming Galerkin with bi-linear elements) and report the performance of your code.

You may want to consult the documentation of the `VectorTools::integrate_difference` function in order to implement this feature. Alternatively (and preferably), you may write the integration routine by hand over each cell while relying on `FEValues::get_function_values` to obtain the solution values at quadrature points. The manual integration technique is encouraged as it is more transparent and allows for the future possibility of computing custom norms (e.g the energy norm in a discontinuous Galerkin method).

Possible extensions for bonus points (optional)

Leveraging templates for a dimension-independent implementation (10 points): Use the dimension as a parameter template to instantiate classes for different dimensions from the same source code. Note that even using this approach will not subvert the need to write code specific to 2-D or 3-D cases - it will simply minimize the extra work. The converse idea would be to not use templates at all and create two separate classes for 2-D and 3-D. This will result in a program nearly double the length of the one with templates, and hence twice as likely to have undiscovered bugs.

After you add this feature, you can verify your code with a 3D benchmark problem. Set up your code in 3D for a unit cube with vertices at $(0,0,0)$ and $(1,1,1)$. Other inputs are:

$$\boldsymbol{K} = \begin{bmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \end{bmatrix}$$

$$q = -\sin(x_1 - x_2 - x_3) + (x_1 + x_2 + x_3)\cos(x_1 - x_2 - x_3) \quad \text{for} \quad \boldsymbol{x} \in \Omega = [0,1]^3$$

$$g_D = \begin{cases} \cos(x_1 - x_3) & \text{for} \quad \boldsymbol{x} \in (\cdot, 0, \cdot) \\ \cos(x_1 - 1 - x_3) & \text{for} \quad \boldsymbol{x} \in (\cdot, 1, \cdot) \\ \cos(x_1 - x_2) & \text{for} \quad \boldsymbol{x} \in (\cdot, \cdot, 0) \\ \cos(x_1 - x_2 - 1) & \text{for} \quad \boldsymbol{x} \in (\cdot, \cdot, 1) \end{cases}$$

$$g_N = \begin{cases} 0 & \text{for} \quad \boldsymbol{x} \in (0, \cdot, \cdot) \\ \sin(1 - x_2 - x_3) & \text{for} \quad \boldsymbol{x} \in (1, \cdot, \cdot) \end{cases}$$

The BVP described above admits the exact solution:

$$p = \cos(x_1 - x_2 - x_3)$$

Report the errors and the convergence rates of your code.

Submission guidelines

1.) Submit your code

Add a multi-line comment to the top of your source code that contains the following information:

- Your personal information - name, UTEID
- Line numbers where the following inputs are specified:
  - Diffusion coefficient
  - Forcing function
  - Dirichlet boundary values
  - Reference finite element
  - Quadrature rule
- A summary of features and capabilities which may extend beyond what is specified in the mandatory part

Rename your source code in the following format - [eid].cc - and upload it to Canvas by 11:59 PM on 11/15/2017.

2.) Submit your report of the results
Summarize your results in a report including:

- A convergence table showing the number of cells, the number of degrees of freedom, the mesh size, the L^2 error and the convergence rate.
- A plot showing the mesh size and the L^2 error in log-log scale.
- Solution plots with high resolution for the benchmark problem. The mesh needs to be 64 by 64 at least.

The template of the convergence table is shown as below.

Table 1: Spatial convergence rate study

cycle	mesh size	# cells	# dofs	L^2-error	spatial convergence rate
1					
2					
3					
4					

Submit your report in class on 11/15/2017.

Option: use other programming languages

If you do not want to learn and use **deal.ii** library, you can choose to write your code from scratch using your favorite programming language such as `Matlab`. Your code needs to have the same capabilities described above. You can also get bonus points if you solve the 3D problem.