One-dimensional Finite Element Coding Project

Instructor: Debanjan Mukherjee *debanjan@Colorado.Edu*
TA: Zachariah Irwin *Zachariah.Irwin@Colorado.Edu*
Due Date: March 13, 2020 **Total Points:** 100 points at 100%

Office hours:

Debanjan:	Tues 09:00am-10:30am + Thurs 3:30pm-5:00pm	[275 ECME]
Zachariah:	Mon 10:00am-11:00am + Wed 3:00pm-4:00pm	[218 ECOT]

Assignment Instructions:

- ***This project duration spans 5 weeks and is due on March 13 at 05:00 pm on Canvas.***
- ***The project is mandatory for 5228 level students – aiming to provide them with background on advanced understanding and handling of numeric arrays for FEM calculations.***
- ***Download the project files onto a local folder on your computer.***
- ***Start implementing your code for the various functions by following the instructions step-by-step/part-by-part.***
- ***Implement the code directly in the various template files provided.***
- ***Save your report for the analysis in Part 10 in this same folder.***
- ***When completed, this folder should have:***
  - ***FEM_1D_Functions.py***
  - ***Second_Order_BVP.py***
  - ***SolverMain.py***
  - ***project_part_1.py***
  - ***project_part_2.py***
  - ***project_part_3.py***
  - ***project_part_4.py***
  - ***Analysis-Report.pdf (or whatever name you give it).***
- ***Zip this folder, and upload the folder onto Canvas submission site.***
- ***If you score all parts correctly, and the analysis is complete you will be scored at 100 points.***
- ***Standard homework submission and late submission policies as outlined in the Syllabus will apply.***

1. Project objectives:

The overall goal of this project is to introduce student to the nuances and intricacies of implementing the finite element method – some of which we have seen already during the lectures. Through this project, students will be acquainted with understanding and handling of numeric arrays for finite element method computations, and implementing basic algorithmic pieces. This project will also serve as an introduction to scientific computing using Python – for those who have not been working with Python yet. The several step-by-step outline provided here to solve a boundary value problem using finite element methods should give you an overview of the overall flow of method and how the mathematics derived in class translates into computer code to perform actual computations.

2. General structure of the project:

If you follow the instructions presented here, you should be able to create three main Python scripts.

The first, a set of functions that implement the basic ingredients of the finite element method for 1-dimensional problems – we will place these in a file named **`FEM_1D_Functions.py`**

The second, a set of functions that implement the computation of discretized matrices and vectors, performs global assembly, and solves the resultant system. These are specific to a boundary value and its weak form, hence we will assume a template boundary value problem provided in Section 3. We will place these functions in a file named **`Second_Order_BVP.py`**

Finally, you will use the functions in **`FEM_1D_Functions.py`** and **`Second_Order_BVP.py`** to create a solver script **`SolverMain.py`** where you will enter specific problem parameter inputs and solve a boundary value problem.

As you work through the project, we have provided instructions for you to test your code with specified inputs. You should use these instructions to check your code with provided benchmark outputs wherever relevant. This should help you find any bugs you may have in your implementation.

We will also use these instructions to evaluate and assign grades to your work. The inputs we will use to test the codes will be specified wherever relevant.

For any of these checks, when you compose your script: for example **`project_part_1.py`**, you can import all the functions from your main Python code **`FEM_1D_Functions.py`** or **`Second_Order_BVP.py`** using the standard Python import command (for example):

```
from FEM_1D_Functions import *
```

3. The template boundary value problem:

For this programming project we will assume a template boundary value problem as follows (this is the same 1-dimensional problem which we started with as we discussed finite element method details, except that for simplicity we have a Dirichlet condition specified at both ends):

$$\frac{\partial^2 u}{\partial x^2} = f(x) \ \forall \ x \in [a, b]$$
$$u = u_a \text{ at } x = a$$
$$u = u_b \text{ at } x = b$$
$$f(x) = k^2 \cos\left(\frac{\pi k x}{L}\right) + \alpha\left(1 - k^2\right) \sin\left(\frac{2\pi k x}{L}\right)$$

For this assignment, we will assume the following parameter values:

a = 0, b = 1, L = 1.0, and u_a = 0.0, u_b = 1.0

We will also assume that α = 5.0 here.

3.a. Getting started: Obtaining the exact solution for this problem:

For the given form of this function, you can obtain an analytical solution for u(x) by integrating the differential equation subject to the two boundary conditions.

Compute this solution and keep it ready for later.

3.b. Getting started: Planning out the pseudocode:

Use the notes and derivations from class, and the functions you have defined thus far, to construct a pseudocode/algorithm for computing the element stiffness matrix and the element loading vector, assembling it over a given mesh globally, and implementing the Dirichlet boundary condition for the template boundary value problem defined here.

Note: You are recommended to draft this cleanly and submit along with the rest of the project (you can type or write/scan this part). You will not be graded on this. However, as you may have already realized, you need to have this correctly formulated if you want to program the matrix computations anyways. Hence, we are mandating that you prepare this first before moving on to the next part.

4. Programming 1D meshing and mesh data-structures:

4.a. Creating mesh node array:

Start by composing a Python function **`generateMeshNodes`** with the following specifications:

input-args: *domain interval (in a form [a,b]);*
polynomial degree;
element size

output-args: *array of node coordinates;*
number of nodes;
number of elements

Note: It is required to use a Numpy array to store and handle the array of node coordinates.

As an example, you might think of using the following form of Python function signature:

```
def generateMeshNodes(a_Domain, a_Degree, a_Size):

      [function body that you need to program]

return [numElements, numNodes, nodes]
```

4.b. Creating element connectivity:

Next, compose a Python function **`generateMeshConnectivity`** with the following specifications:

input-args: *number of elements;*
polynomial degree

output-args: *element connectivity matrix*

Note: It is required to use a Numpy array to store and handle the mesh connectivity array.

As an example, you might think of using the following form of Python function signature:

```
def generateMeshConnectivity(a_NumElements, a_Degree):

      [function body that you need to program]

return connectivity
```

4.c. Testing your code & submission evaluation:

Create a script **`project_part_1.py`**

In this script, import your functions from **`FEM_1D_Functions.py`** using:
`from FEM_1D_Functions import *`

For Benchmarking:

Use the two functions **`generateMeshNodes`** and **`generateMeshConnectivity`** to create the mesh data structures for the following input.

- Domain Interval = [0.0,1.0]; Polynomial Order = 2; Size = 0.1

You should get the results stated below for checking.

```
The Number of Elements: 10
The Number of Nodes: 10
The List Of Nodes:
[0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65 0.7  0.75
0.8  0.85 0.9  0.95 1.  ]
The Connectivity Matrix:
[0 1 2]
[2 3 4]
[4 5 6]
[6 7 8]
[ 8  9 10]
[10 11 12]
[12 13 14]
[14 15 16]
[16 17 18]
[18 19 20]
```

For Evaluation:

We will run it with the following arguments. If we get the correct outputs, you get full points for this segment.

- Domain interval = [0.0, 1.0]; Polynomial order = 1; Size = 0.1.
- Domain interval = [0.0, 1.0]; Polynomial order = 1; Size = 0.1.

5. Programming Isoparametric 1D elements:

5.a. Computing nodes for reference element of a given polynomial order:

Having completed the meshing functions, next we will compose functions that implement the mathematics of linear isoparametric 1-dimensional elements. Start by compose a Python function **`referenceElementNodes`** in the **`FEM_1D_Functions.py`** file which generates all the reference element nodal coordinates for a given polynomial degree as input.

input-args: *polynomial degree of the finite element*
output-args: *array of reference element node locations (in the range `[-1.0, 1.0]`)*

Note: The output of the function has to be in form of a Numpy array.

As an example, you might think of using the following form of Python function signature:

```
def referenceElementNodes(a_Degree):

     [function body that you need to program]

return refNodes
```

5.b. Element shape functions and their derivatives:

Next, compose a Python function named **`lineElementShapeFunction`** in **`FEM_1D_Functions.py`** that implements the shape functions for 1-dimensional line elements. Specifically, this function will take a reference coordinate value and output the shape function value at that reference coordinate location.

input-args: *coordinate in the local coordinate system where shape function is to be evaluated*
polynomial degree/order;
id of the local node for which shape function is computed;

output-args: *a single scalar value for the basis function evaluated in local coordinate system.*

Note: This function should be able to compute shape functions for any polynomial degree/order.

As an example, you might think of using the following form of Python function signature:

```
def lineElementShapeFunction(a_Eta, a_Degree, a_LocalNode):

    [function body that you need to program]

return shapeVal
```

Next, compose a Python function named **`lineElementShapeDerivative`** in **`FEM_1D_Functions.py`** that implements the shape function derivatives for 1-dimensional line elements. Specifically, this function will take a reference coordinate value and output the derivative of the shape function evaluated at that reference coordinate location.

input-args: *coordinate in the local coordinate system where shape derivative is to be evaluated*
polynomial degree/order;
id of the local node for which shape function derivative is computed;

output-args: *a single scalar value for the basis function derivative evaluated in local coordinate system.*

Note: This function should be able to compute shape derivatives for any polynomial degree/order.

As an example, you might think of using the following form of Python function signature:

```
def lineElementShapeDerivatives(a_Eta, a_Degree, a_LocalNode):

    [function body that you need to program]

return shapeDiffVal
```

5.c. Isoparametric mapping and its gradient:

Next, compose a Python function named **`lineElementIsoparametricMap`** in **`FEM_1D_Functions.py`** that can compute the numeric value of the standard isoparametric mapping for a linear element, for a given reference coordinate location.

input-args: *element nodal coordinates;*
polynomial degree/order;
coordinate in the local coordinate system where mapping is to be evaluated

output-args: *the mapped coordinate value using shape function interpolation.*

As an example, you might think of using the following form of Python function signature:

```
def lineElementIsoparametricMap(a_ElementNodeCoordinates, a_Degree, a_Eta):

      [function body that you need to program]

return isoMap
```

Next, compose a function named **`lineElementMappingGradient`** in **`FEM_1D_Functions.py`** that can compute the numeric value of the derivative or gradient of the standard isoparameteric mapping for a line element, for a given reference coordinate location.

input-args: *element nodal coordinates;*
polynomial degree/order;
coordinate in the local coordinate system where mapping is to be evaluated.

output-args: *the gradient of the mapping/transformation.*

As an example, you might think of using the following form of Python function signature:

```
def lineElementMappingGradient(a_ElementNodeCoordinates, a_Degree, a_Eta):

      [function body that you need to program]

return dXdEta
```

5.d. Testing your code and submission evaluation:

For Benchmarking:

Create a script **`project_part_2.py`**. In this script, import functions from **`FEM_1D_Functions.py`** using: **`from FEM_1D_Functions import *`**

Use this script to create a plot of the 2-node linear element shape functions and derivatives, and a plot of the 3-node quadratic element shape functions and derivatives. Make sure your plots for the shape functions match the figures shown in lecture notes and pdf notes.

For evaluation:

We will run the same script, but now for displaying shape functions for polynomial degree 3 and 4. If this works and we get the expected output, you get full points for this segment.

6. Programming the local evaluation of element matrices:

6.a. The stiffness matrix:

Compose a Python function (using the pseudocode/algorithm developed in part 3) named **`computeElementStiffness`** inside the Python file **`Second_Order_BVP.py`** that computes the element stiffness matrix for the template boundary value problem. Use the following specifications:

input-args: *the nodal coordinates of the element;*
order of Gaussian quadrature;
polynomial degree/order.

output-args: *the fully evaluated element stiffness matrix.*

Note: For this functions, please use the functions **`getGaussQuadratureWeights`** ***and*** **`getGaussQuadraturePoints`** ***already provided at the end of this assignment.***

As an example, you might think of using the following form of Python signature:

```
def computeElementStiffness(a_ElementNodes, a_GaussOrder, a_Degree):

      [function body that you need to program]

return k
```

6.b. The element loading vector:

Compose a Python function (using the pseudocode/algorithm developed in part 3) named **`computeElementLoading`** inside the Python file **`Second_Order_BVP.py`** that computes the element loading vector for the template boundary value problem. Use the following specifications:

input-args: *the actual function f(x) to be integrated*
the nodal coordinates of the element;
order of Gaussian quadrature;
polynomial degree/order;

output-args: *the fully evaluated element loading vector.*

Looking at the argument definitions above, you might be wondering how to actually pass a function – specifically a Python function – as an argument to another Python function. Python makes this a really simple task by referencing the function as a passable object. Therefore, if you have programmed your function f(x) inside a Python function **`poissonF`** (for example) you can simply pass the name poissonF as the first argument as defined above.

Note: For this functions, please use the functions **`getGaussQuadratureWeights`** *and* **`getGaussQuadraturePoints`** *already provided at the end of this assignment.*

As an example, you might think of using the following form of Python signature:

```
def computeElementLoading(a_Func, a_ElementNodes, a_GaussOrder, a_Degree):

      [function body that you need to program]

return f
```

6.c. Testing your code and submission evaluation:

For benchmarking:

Create a script **`project_part_3.py`**. Import functions from your main script using:

```
from FEM_1D_Functions import *
from Second_Order_BVP import *
```

Use this script to compute and display the element stiffness and loading vectors for f(x) = x using 2-node linear elements for element 1 in your mesh. Your output should match the following (for given inputs in Section 4.c.): Domain interval = [0.0, 1.0]; Polynomial order = 1; Size = 0.1.

```
The Element Stiffness Matrix:
[ 10. -10.]
[-10.  10.]
The Element Loading Vector:
[0.0025 0.0025]
```

For evaluation:

We will run your script to generate stiffness matrix for any polynomial order higher than 1. If the resulting element matrix is correct you get full points for this segment.

7. Programming the finite element assembly for the given boundary value problem:

7.a. Assembling the global stiffness matrix:

Compose a Python function (using the pseudocode/algorithm developed in part 3) named **`assembleGlobalStiffness`** inside the Python file **`Second_Order_BVP.py`** to assemble the global stiffness matrix for this boundary value problem from the local element stiffness matrices. Use the following specifications for input and output arguments:

input-args: *element nodal coordinates;*
element connectivity matrix;
order of Gaussian quadrature;
polynomial degree/order

output-args: *the global stiffness matrix*

As an example, you might think of using the following form of Python signature:

```
def assembleGlobalStiffness(a_Nodes, a_Connectivity, a_GaussOrder, a_Degree):

[function body that you have to program]

return k_g
```

7.b. Assembling the global loading vector:

Next, compose a Python function (using the pseudocode/algorithm developed in part 3) named **`assembleGlobalLoading`** inside the Python file **`Second_Order_BVP.py`** to assemble the global loading vector for this boundary value problem from the local element loading vectors. Use the following specifications for input and output arguments:

input-args: *the actual function f(x) to be integrated*
element nodal coordinates;
element connectivity matrix;
order of Gaussian quadrature;
polynomial degree/order;

output-args: *the global stiffness matrix*

As stated in Part 6, you can pass the function where you have programmed f(x) by name here as the first argument.

As an example, you might think of using the following form of Python signature:

```
def assembleGlobalLoading(a_Func, a_Nodes, a_Connectivity, a_GaussOrder, a_Degree):

      [function body that you have to program]

return f_g
```

7.c. Testing your code and submission evaluation:

For benchmarking:

Create a script **project_part_4.py**. Import functions from your main script using:

```
from FEM_1D_Functions import *
from Second_Order_BVP import *
```

Use this script to compute and display the element stiffness and loading vectors for f(x) = x using a mesh of three 2-node linear elements. Your output should match the following (for given inputs in Section 4.c.): Domain interval = [0.0, 1.0]; Polynomial order = 1; Size = 0.1

```
The Global Stiffness Matrix:
[ 2. -2.  0.]
[-2.  4. -2.]
[ 0. -2.  2.]
The Global Loading Vector:
[0.0625 0.25   0.1875]
```

For evaluation:

We will use your script to obtain the element stiffness matrices for:
a. a mesh made up of three 3-node quadratic elements
b. a mesh made up of ten 2-node linear elements.
If both resulting assembled matrices are correct you get full points for this segment.

8. Applying Dirichlet boundary conditions:

8.a. Implementing the Dirichlet conditions at the global matrix level:

Using the techniques discussed in class, the following algorithmic approach can be employed to implement Dirichlet/Essential boundary conditions for the problem. Use this algorithm provided below to implement the Dirichlet Boundary conditions via a Python function **`applyDirichlet`** that has the following specifications for input and output arguments:

input-args: *global stiffness matrix;*
global vector;
list of Node Id's where Dirichlet values specified;
list of Dirichlet values

output-args: *modified global stiffness matrix;*
modified global vector

Note: The simplest way is to have the third and fourth arguments be lists of id's and values respectively, and loop over the list.

applyDirichlet – use the following pseudocode for your implementation:

```
i          ← dirichlet node;
u0         ← dirichlet value;
K_g[i,:]   ← 0.0;
F_g        ← F_g – K_g[:,i]*u0;
K_g[:,i]   ← 0.0;
K_g[i,i]   ← 1.0;
F_g[i]     ← u0;
```

As an example you might think of using the following form of Python signature:

```
def applyDirichlet(a_Kg, a_Fg, a_DirNodes, a_DirVals):

     [function body that you need to program]

return a_Kg, a_Fg
```

9. You have your own finite element solver coded up!!

You can now compose an entire finite element code of your own, having composed all the component functions in all the prior parts inside **`FEM_ID_Functions.py`** and **`Second_Order_BVP.py`**. Write the complete solver in Python inside a third file **`SolverMain.py`**, by identifying that the following is the format/pseudocode you need to stitch all the functions together:

```
set x0 = domain left end
set x1 = domain right end
set p = polynomial order/degree
set h = element size
set gN = order of Gaussian Quadrature
provide the function f(x) for the problem*
generate mesh nodes
generate mesh connectivity
assemble the global stiffness matrix
assemble the global loading vector
set dirNodes = list of all Dirichlet node id's
set dirVals = list of all Dirichlet values
apply Dirichlet boundary condition
solve for u**
plot u against nodal coordinates
```

Note: for this step, it is simplest to use the Python lambda function syntax as follows for f(x):

```
fun = lambda x: -1.0*( (2*2)*np.cos(np.pi*2*x/1.0) + … etc.
```

Note: For this part of the program, you do not need to code up a linear system solver. Please use the pre-built conjugate gradient solver in SciPy (for example) by using the following (assume K, F are final matrix and vector):

First import the solver from SciPy

```
from scipy.sparse.linalg import cg
```

Then use the solver to obtain the resulting nodal solution

```
[uSol, status] = cg(K, F)
```

10. Solution and analysis using the template boundary value problem:

a. For the template boundary value problem provided here, recall the analytical solution for u(x) that you derived in Part 3.

b. Using 2-node line elements, and a mesh sizing of h = 0.05, generate a numerical solution for u(x) using the full solver code, for the case where k = 2.0.

c. Plot the two solutions (exact solution from a and fem solution from b) on the same figure and show how the two solutions compare.

d. Compute an error by summing up the absolute difference between numerical and analytical solutions across the entire domain. Report this error.

e. Present an analysis on how the solution error behaves with varying element size h for k = 2, 4, 8, 16, 32.

For evaluation:

Provide your results from this analysis (as well as any of your benchmarking results from the previous steps) into a single report saved as a pdf file.

Appendix: Python functions for tabulation of weights and quadrature points for 1-dimensional Gaussian Quadrature up to order 5

```
def getGaussQuadratureWeights(a_Degree):

    import sys

    if a_Degree == 1:
        return [2.0]
    elif a_Degree == 2:
        return [1.0, 1.0]
    elif a_Degree == 3:
        return [5.0/9.0, 8.0/9.0, 5.0/9.0]
    elif a_Degree == 4:
        return [0.652145, 0.347855, 0.652145, 0.347855]
    elif a_Degree == 5:
        return [0.236987, 0.478629, 0.568889, 0.478629, 0.236927]
    else:
        sys.exit("Only upto 5th degree Gauss quadrature is implemented")

def getGaussQuadraturePoints(a_Degree):

    import sys

    if a_Degree == 1:
        return [0.0]
    elif a_Degree == 2:
        return [-0.57735, 0.57735]
    elif a_Degree == 3:
        return [-0.774597, 0.0, 0.774597]
    elif a_Degree == 4:
        return [-0.861136, -0.339981, 0.339981, 0.861136]
    elif a_Degree == 5:
        return [-0.90618, -0.538469, 0.0, 0.538469, 0.90618]
    else:
        sys.exit("Only upto 5th degree Gauss quadrature is implemented")
```