Universitat Politècnica de Catalunya
Numerical Methods in Engineering
Programming for Engineers and Scientists

FEM Code Pre-planning

March 3, 2020

Sebastian Ares de Parga	aresdepargas@gmail.com
Eduard Gómez	eduard.gomez.escandell@gmail.com
Federico Valencia	fvalenciao95@gmail.com

Contents

1 Introduction

The Finite Element Method (FEM) is a numerical method that is implemented for the approximation of solutions of complex differential equations associated with a physical problem about complicated geometries. FEM is designed to be implemented in computers to obtain very approximated numerical solutions on a body, structure or domain on which certain differential equations are defined in a weak form that characterize the physical behavior of the problem by dividing it into a large number of sub-domains called Finite Elements.

The aim of this assignment is to design a FEM code which will be able to solve a variety of engineering problems by using several different types of elements in the most flexible way possible. The code will be implemented on Matlab, with the idea of applying Object Oriented Programming (OOP) to make the code efficient and clean. This report will contain an initial draft design of the program which will be developed throughout the course.

The code will be able to solve problems presenting the following characteristics:

- 2D/3D domains
- Triangular/Quadrilateral/Tetrahedral/Hexahedral elements
- Dirichlet/Neumann boundary conditions
- Material properties depending on space location

This report includes a description of:

- Objects that will be implemented throughout the code.
- Attributes and methods contained in the objects.
- General structure of the program
- Input and output channels

1.1 Work methodology

Because this project is rather open ended, our idea is not to set goals and divide the workload amongst us, but rather to discuss problems as they come and solve them in group for high level decisions, and individually for small ones. We have set out a unrealistic goal: solving any linear problem in 1, 2 and 3 dimensions. Our real goal is to be able to solve as wide a variety of this sort of problems as possible. To do this we'll make the code flexible and particularize only when necessary.

To mantain good versioning control and have a centralized storage space for the project, we decided to open a Github repository.

2 Objects

This section features all the classes we thought could be necessary to fulfil the aforementioned specifications, as well as all the methods and properties we thought of. Note that although variables types need not be declared in Matlab, we decided to specify them here for easier understanding.

All our classes are handle classes. This is detailed in subsection 2.2, but first giving a look at what objects we'll be using makes understanding the next section easier.

2.1 Classes

2.1.1 Node

This class is very simple and only has the following attributes:

- `real[] X` is an array with the coordinates of a specific node.
- `char[] BC_type` is an array with the boundary condition types in each axis
- `real[] BC_value` is an array with the boundary condition value
- `int[] BC_id` is an array with the ID of the nodes when the matrices and vectors are split as explained in subsection 2.1.4.

2.1.2 Element

This class defines a single element and its attributes are:

- `@Node{} nodes` contains handles to the nodes it is connected to
- `@Material material` contains a handle to the material it's made out of
- `real[][] jacobian` contains the jacobian matrix of the element. If during the development of the code we learn that the jacobian is only needed once, we'll save memory and not store it here, instead calculating it only when needed.
- `real area` contains the area of the element

2.1.3 Domain

Contains a lot of information about the geometry of the domain. The attributes are:

- `Node{} nodes` contains all the nodes. Their index in the cell array is considered its `id`.
- `Element{} elems` contains all the elements. Their index in the cell array is considered its `id`.
- `Material{} materials` contains all the materials. Their index in the cell array is considered its `id`.
- `int n_nodes` is the number of nodes
- `int n_elems` is the number of elements
- `int n_dimensions` is the number of dimensions
- `int nodes_per_elem` is the number of nodes per element

- `int DOF_per_node` is the number of degrees of freedom per node

This class will also have some public methods:

- `readFromFile(fileName)` will read the problem data from a text file.This is further specified in section 3.2.
- `I = local2global(elem, i)` will take the element or element id, and the local coordinate of a node and output the global coordinate of the node or a handle to the node. We'll decide depending on what is more useful to use.
- `new_material(id, properties, ...)` will create a new material and append it to the materials cell array. it will also increase `n_materials` by one.
- `new_element(id, node_ids, material_id...)` will act similarly for elements.
- `new_node(id, coordinates,...)` will act similarly for nodes.

We might add some more methods to improve readability.

2.1.4 SystemOfEquations

This holds the matrices and vectors necessary to assemble and solve the system. The system is split according to the boundary condition type on of each node:

$$\left[\begin{array}{c|c} K_{aa} & K_{ab} \\ \hline K_{ba} & K_{bb} \end{array}\right] \left[\begin{array}{c} u_a \\ \hline u_b \end{array}\right] = \left[\begin{array}{c} F_a + R \\ \hline F_b \end{array}\right] \quad \text{where} \quad \begin{array}{l} a \in \Gamma_D \\ b \in \Omega - \Gamma_D \end{array} \Bigg\} \tag{1}$$

Then, to solve the system we need only calculate the non-prescribed values:

$$u_b = K_{bb}^{-1}(F_b - K_{ba}u_a) \tag{2}$$

If we wanted to calculate the reactions we can do:

$$R = K_{aa}u_a + K_{ab}u_b - F_a \tag{3}$$

The point of this all is to justify the attributes of this class:

- `sparse Kaa, Kba, Kab, Kbb`
- `real[] Fa, Fb, ua, ub, u`
- `boolean isSymmetrical` can speed up calculations
- `boolean isSolved` to avoid reading meaningless values of `u, ua, ub`.

This function will have a couple of methods:

- `calcElement(geo)` will calculate the local stiffness matrix and force vectors.
- `add_to_system(geo, element_id, K_local, F_local)` will take the local stiffness matrix and force vector and distribute the values onto the global stiffness matrices.
- `solve()` will solve equation 2. it will also merge `ua` and `ub` into `u`.
- `solveReactions()` will solve equation 3.

A possible improvement will be to use Lagrangian multipliers instead of this method of applying boundary conditions, however we consider this is a secondary, non-critical, objective.

2.1.5 GaussPoint

This object contains the information contained in a single Gauss point. They will all be contained in a cell array of Gauss Points. Here are its attributes:

- `real w` is the weight associated to this point.
- `real[] X` are the coordinates of this Gauss Point
- `real[] N` are the values of the the shape functions at this point
- `real{}[][] gradN` are the gradients of the shape functions at this point

This class will have the following methods:

- `changeCoords(obj, domain, X_vertices)` outputs the coordinates in x-y spce instead of isoparametric.

There will also be functions that act on the cell array that contains all the gauss points:

- `loadGaussData(domain)` loads the gauss coordinates and weights from a txt file
- `calcShapeFunctions(gauss_data, domain)` calculates the values and gradients of the shape functions.

2.2 Handle vs value classes

In matlab there are two types of classes:

- Value classes are passed to functions by copy, so things like `object.increase()` do not work, so one must use `object = object.increase()`, which can be quite slow for large objects.
- Handle classes are passed by reference and can be treated similar to pointers in a language like C, which makes the code cleaner and faster.

The implications for the readability of the code also favour handle classes. For instance, to access the coordinates of second node of the third element with value classes we'd have to use:

```
geometry.nodes(geometry.elems(3).node_IDs(2)).X
```

this is because `nodes` is an attribute of `geometry`, and the `elements` can only store the IDs of the nodes, not the node itself, since that would mean storing duplicate information. With class handles we can use

```
geometry.elems(3).nodes(2).X
```

which is much cleaner and easy to read. Note that here `nodes` can be accessed both from `geometry.nodes` and `geometry.elems(i).nodes`, but both variables point to the same adress in memory, without duplicating the information.

In conclusion, all our classes are handle classes.

3 Structure of the program

3.1 Main file

This section shows how the main code is planned to work, considering the features described in the last section (Objects).

```
data_file = 'data/problem3';                % name of data input directory
out_file = 'data/problem3_result.res';      % name of data output file

domain = Domain();                          % Initialize domain
domain.readFromFile(data_file)              % Import from file

gauss_data = loadGaussData(geometry)        % Load gauss quadratures
calcShapeFunctions(gauss_data, domain);     % Calculate shape functions

syst_eq = systemOfEquations(geometry)       % Initialize stiffness matrix
                                            % and load vector

syst_eq.assemble(geometry, gauss_data)      % Assemble matrices and vectors

syst_eq.solve()                             % Solve SoE and reassemble solution vector

syst_eq.store_result(out_file)              % Write output file
```

3.2 Data loading

The data will be loaded from various files, all contained in the same directory. There will be the following files for each problem:

- The mesh will be loaded from a `msh` file generated by GiD.
- The materials definition file will contain a list of the materials and their properties.
- The boundary conditions file will contain a list of all the nodes with a boundary condition other that Neumann $\nabla u \cdot n = 0$, which will be considered the default one.
- The problem setup file will contain info such as the degree of integration or the type of problem (planar elasticity, thermal, etc.).

The program will also have its own data files independent of each problem, which contain the weights and coordinates of the gauss points.

3.3 Data exporting

The output file will be a `pos.res` file usable by GiD to visualize the post-processing.