Universitat Politècnica de Catalunya

Programming for Engineers and Scientists

Master's Degree in Numerical Methods in Engineering

Design of a Finite-Element code

Authors:
Pau Márquez
Iván Pérez
Diego Roldán

Supervisor:
Prof. M. De Corato

Academic Year 2019-2020

Contents

1 Introduction **1**

2 Methodology discussion **2**

3 Code design **11**

4 Concluding remarks **14**

1 Introduction

In this report, it is discussed the methodology procedures to implement a finite-element code to solve the Poisson equation in an arbitrary domain. In it, it is developed a design that will allow us to solve to problem taking into account a number of features to add generality and make it suitable for solving a variety of geometries (both in 2D and 3D) and conditions in the boundary.

The motivation of this project comes from the fact that, for two and three-dimensional domains, the need of a numerical solution for the majority of practical problems involving partial differential equations becomes essential. In fact, only the simplest domains and boundary conditions may be solved with an exact solution. It will be seen that the use of piece-wise linear functions with triangular or tetrahedral sub-domains (for two dimensions) is a good choice when solving the Poisson equation, in which C^0 continuity of the shape functions is demanded [1]. Moreover, higher order shape functions will also be used in such domains.

The mathematical statement of the problem to be solved is that of the steady-state Poisson equation in three dimensions (or two) in equation 1

$$\frac{\partial}{\partial x}\left(k_x\frac{\partial \phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k_y\frac{\partial \phi}{\partial y}\right)+\frac{\partial}{\partial z}\left(k_z\frac{\partial \phi}{\partial z}\right)+Q=0 \quad in \quad \Omega \tag{1}$$

With the boundary conditions

$$\begin{cases} \phi=\overline{\phi} \quad on \quad \Gamma_\phi \\ k\frac{\partial \phi}{\partial n}=-\overline{q} \quad on \quad \Gamma_q \end{cases} \tag{2}$$

As in the general case, the unknown function ϕ will be the variable to be determined by the finite element code. In addition, if we assume isotropic material $k_x=k_y=k_z=k$, the weak form of the weighted residual statement, using the trial functions as weighting functions yields to a system of algebraic equations that can be constructed by summing the individual element contributions (equations (3) and (4)).

$$K_{ij}^e=\int_{\Omega^e}\left(\frac{\partial N_i^e}{\partial x}k\frac{\partial N_j^e}{\partial x}+\frac{\partial N_i^e}{\partial y}k\frac{\partial N_j^e}{\partial y}+\frac{\partial N_i^e}{\partial z}k\frac{\partial N_j^e}{\partial z}\right)dxdydz \tag{3}$$

$$f_i^e=\int_{\Omega^e}QN_i^e dxdydz-\int_{\Gamma_q^e}N\overline{q}d\Gamma \tag{4}$$

The latter means that the integral over Γ_q^e will only appear on those elements which are adjacent to the boundary of the whole domain, whereas Ω^e is the surface of the particular element. Now equations (3) and (4) provide a general formulation which is valid any type of shape function and element. This is the starting point to formulating the whole procedure. In particular, the code is intended to solve the problem using the following elements:

- 2D: Triangles and quadrilateral elements.
- 3D: Tetrahedral and hexahedral elements.

Moreover, the user will also have the choice to select linear, quadratic or cubic shape functions.

Before stepping into the general design of the code, which steps will have and the main variables involved, let us define which will be description of the shape functions to be used and other methodological aspects on computations.

2 Methodology discussion

In this section, it is discussed the procedure that the FEM code has to follow in order to implement the finite element method in a specific geometry.

2.1 Geometry

First and foremost, it is required to define the geometry which has to be analyzed to study the behaviour of the variables of interest.

As it is shown in figure 1, it will be developed an app to make easier and user-friendly this FEM code which will be implemented.

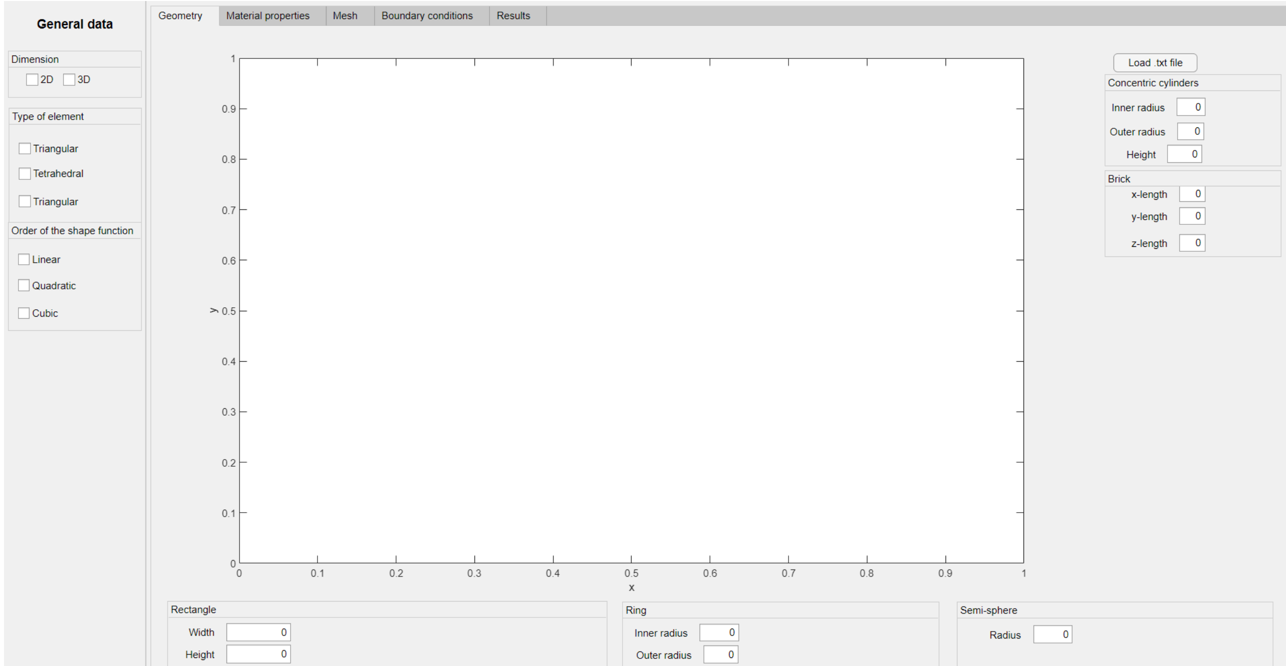


Figure 1: Section of the app where the geometry is defined.

In order to save computational cost, it is created a data structure to represent the description

of our Problem. All the variables referred to the definition of the Problem are gathered in the structure `Problem`.

First, it is required to define the dimension of our domain: 2D or 3D. To do so, it is created the variable `dimension = Problem.dimension` (possible values $\in$ [2,3]), where 2 is referred to a 2D Problem and 3 to a 3D Problem.

In this code, it is provided several predefined geometries for the user. These geometries are the basic ones which it is possible to specify their dimensions in the code without drawing in any CAD tool. They are the following with these parameters necessary to define in the variable `geometry = Problem.geometry` (possible values $\in$ [1,2,3]), where each geometry is referred to a number in brackets below. For 2D domain:

- 1 $\longrightarrow$ Rectangle. Parameters: width and height.
- 2 $\longrightarrow$ Ring. Parameters: Inner and outer radius. If it is necessary to create a circle, the inner radius would be 0.

For 3D domain:

- 1 $\longrightarrow$ Concentric cylinders. Parameters: Inner and outer radius and height.
- 2 $\longrightarrow$ Brick. Parameters: X,Y,Z length.
- 3 $\longrightarrow$ Semi-sphere. Parameters: Radius.

In order to study more complex geometries, it is provided two options in the code. On one hand, it is possible to import a file `.txt` which contains the coordinates that define the geometry. For 2D, with x and y coordinates; for 3D, with x, y and z coordinates. On the other hand, it is possible to import more complex geometries in 2D or 3D created in CAD software with the extension `.stl` using the Matlab built-in function `X = importGeometry(model,geometryfile)`; the output is `X` as the geometry description and the inputs are the model (which is the Problem to analyse) and `geometryfile` (which is the path to the `.stl` file). The `X` output is a Discrete Geometry. At the end, it is obtained a variable `X = Problem.X` which defines the coordinates of the geometry to, a posteriori, generate the mesh with the function `mesh_generator`.

2.2 Mesh

The next step is to create a mesh for the geometry in order to be able to study the Problem of interest with the finite element method. In figure 2 it is shown the section of the app which the user can see the mesh generated and specify the element size of the elements (`Mesh.h`).

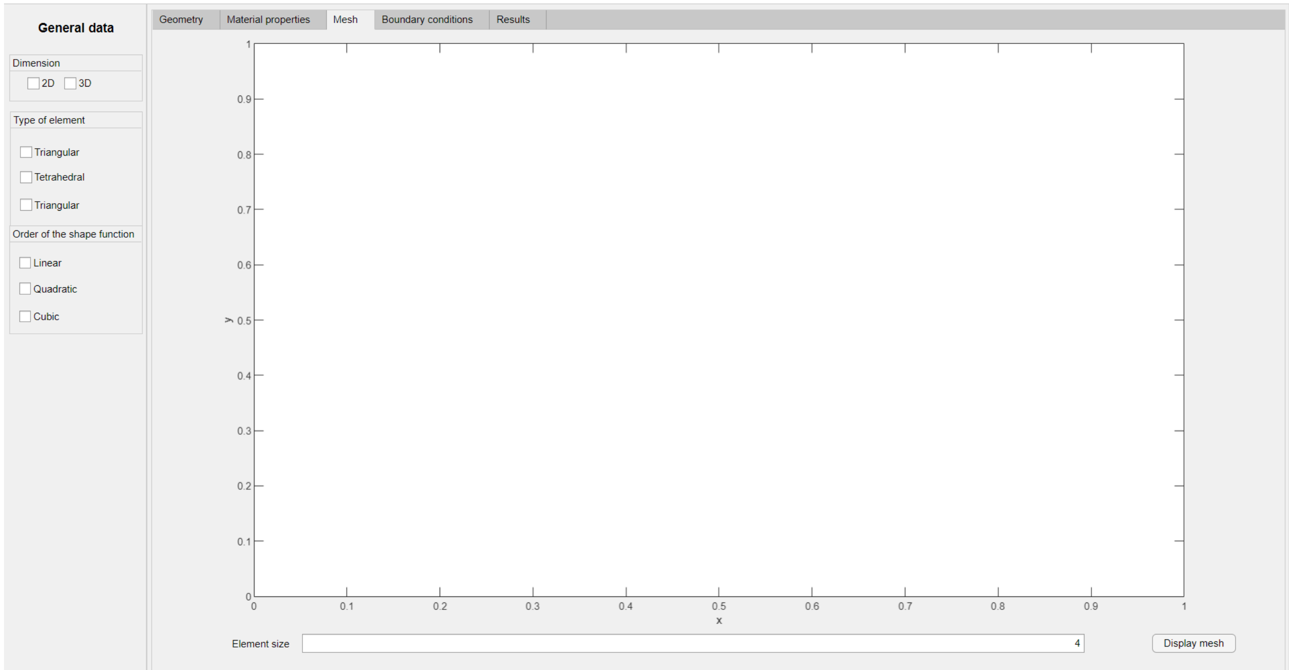


Figure 2: Section of the app where the mesh is defined.

As it is done before, it is created a data structure to represent the mesh of our Problem. All the variables referred to the definition of the mesh are gathered in the structure `mesh`.

It is required to define two more variables of the Problem. First, the type of element of the mesh with the variable `type_element = Problem.type_element` (possible values $\in$ [1,2]). For a 2D Problem, 1 is referred to as triangular element and 2 for a quadrilateral element. For a 3D Problem, 1 is referred to a tetrahedral element and 2 to a hexahedral element. Second, it is necessary to define the order of the shape functions of the element with the variable `order = Problem.order` (possible values $\in$ [1,2,3]); where 1 is linear, 2 is quadratic and 3 is cubic order.

Then, it is created a function in order to generate the mesh for the geometry as it is described below. It has an input the variable `Problem.X`, created in the geometry step and as outputs, two new variables: `nodes = mesh.nodes` and `elements = Mesh.elements`.

1

mesh_generator

Brief description: Computes the mesh of a given 2D or 3D geometry:
Input variables: Problem structure (`Problem`). Contains coordinates of the geometry (`Problem.X`), element size (`Problem.h`), element type (`Problem.type_element`) and order of the shape functions (`Problem.order`), among others.
Output variables: Mesh structure (`mesh`). Contains the connectivity matrix (`Mesh.elements`) and the coordinates of the nodes (`Mesh.nodes`), among others.
Matlab built-in functions used: `decsg`, `geometryFromEdges`, `pdegplot`, `generateMesh`, `pdeMesh`.

In order to analyse the quality of the mesh, it is used the Matlab built-in function `Q = meshQuality(mesh)`, where returns a row vector of numbers from 0 through 1 representing shape quality of all elements of the mesh. Here, 1 corresponds to the optimal shape of the element.

Finally, it used the Matlab built-in function `pdemesh(mesh.Nodes,mesh.Elements)` in order to graph the mesh in the geometry.

2.3 Material properties

Representing the particularity of each media through the PDE of Poisson's equation is possible by changing the k_i therm for the Poisson's equation (1).

In order to apply different material properties through the domain it has been created a function where two kind of materials can be prescribed:

- **Isotropic material**: the material properties of the region are equal in all directions, $k_x = k_y = k_z = k$.
- **Anisotropic material**: the material properties of the region are not equal in all directions, $k_x \neq k_y \neq k_z$.

2

apply_material

Brief description: modify the list of nodal material properties `'Problem.material'`, where default value is `NaN`.
Input variables: `Problem.dimension` , users list of regions, type of material (`isotropic`,`anisotropic`), value (`scalar`, `vector`).
Output variables: `Problem.material`
Matlab built-in functions used: `findNodes`.

Note that for anisotropic material k_x, k_y and k_z are not equal and need to be introduced in vector form.

Furthermore, it will be possible to mix different materials in the domain Problem region, by applying `apply_material` function as many times as the number of materials.

Nevertheless for more than one material the internal boundary region between two given materials needs to be specified to average the two material properties on the nodes of the borders.

3 `apply_boundaryMaterial`

Brief description: modify the list of nodal material properties `'Problem.material'`, with the averaged property for each material.
Input variables: `Problem.dimension` , users list of regions, type of material 1 (`isotropic`,`anisotropic`), value (`scalar`, `vector`), type of material 2 (`isotropic`,`anisotropic`), value (`scalar`, `vector`).
Output variables: `Problem.material`
Matlab built-in functions used: `findNodes`.

2.4 Boundary Conditions

The boundary conditions of the Problem are key to solving the Problem with a sufficient degree of generality. In like manner, an strategy to define various boundary conditions at different segment of the Problem boundary is necessary.

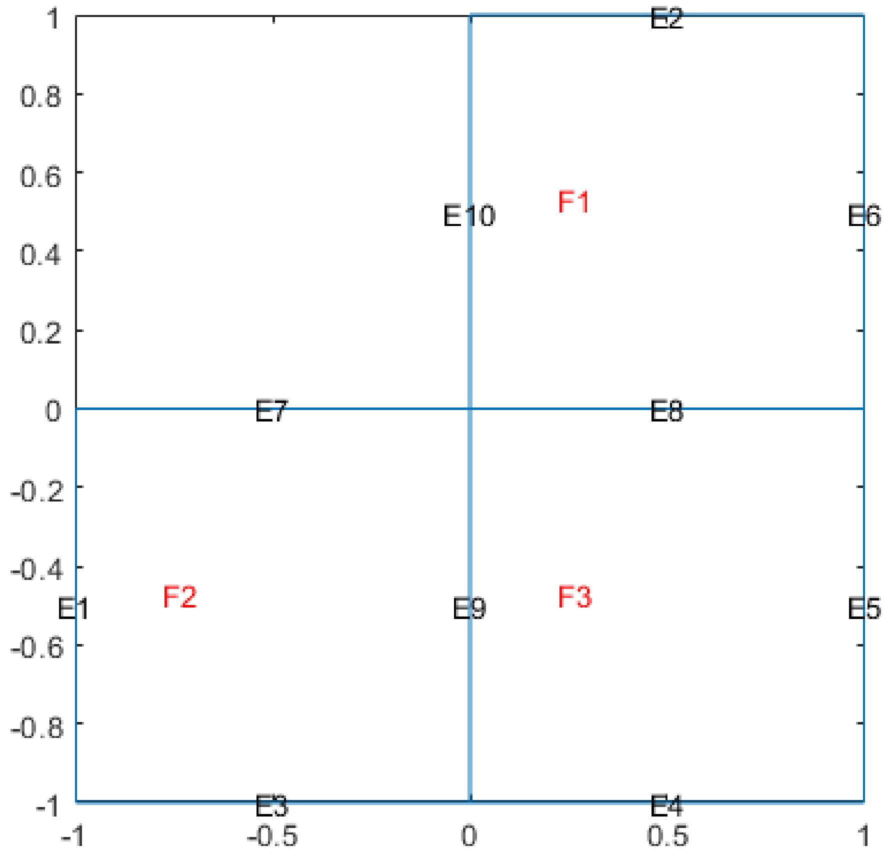


Figure 3: Output figure of the geometry with edge labels for use in the boundary condition definition.

Using the plotted geometry and the name of its parts, the boundary nodes can be find and selected to introduce the 2 types of boundary conditions and its initial values:

- **Dirichlet:** it is prescribed a *fixed* or *non-constant* values for each boundary.
- **Neummann:** it is prescribed a *fixed* or *non-constant* values of fluxes for each boundary

Note that in 2-D elements the boundaries are `Edges` and for 3-D elements referenced boundaries are `Faces`.

Matlab include several built-in functions to allocate nodes in a mesh.

- `nodes = findNodes(mesh,'region',RegionType,RegionID)`: returns the IDs of the mesh nodes that belong to the specified geometric region.

In order to apply the boundary conditions to a selected nodes the following function has been created:

4

`apply_BCs`

Brief description: create the list of nodes with the boundary conditions.
Input variables: `Problem.dimension` , users list of regions, type of BC (`Dirichlet`,`Neummann`), value.
Output variables: `'Problem.bc_dirichlet'`,`'Problem.bc_neummann'`
Matlab built-in functions used: `findNodes`.

2.5 Loads

For instance, in a 2-D heat transfer Problem mathematically represented in 5, the Q is the source therm with units of $\left[\frac{W}{m^2}\right]$.

$$\frac{\partial}{\partial x}\left(k\frac{\partial \phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial \phi}{\partial y}\right)+Q=0 \quad in \quad \Omega \tag{5}$$

If the user desire to apply some kind of load to each element (e.g. a distributed heat source), in the same way that we used to apply the boundary conditions can be used.

5

`apply_source`

Brief description: create the list of nodes with its heat source.
Input variables: `Problem.dimension` , users list of regions, value.
Output variables: `'Problem.loads'`
Matlab built-in functions used: `findNodes`.

It should be apply as many times as desired if different heat sources want to be used.

2.6 Finite elements and approximation

2.6.1 Mapping

The global coordinates x and y can only be used when the geometry is simple, meaning that the basic elements such as a rectangle or a brick restrict the application of the above-mentioned formula to solve Problems of arbitrary geometry. Therefore it becomes essential to map the complex geometries of the elements produced by the mesh in the global coordinates to the local element coordinates (ξ, η). Once a relation has been found between coordinates, the transformation can be carried out. Therefore, the shape functions written in the local element space can be used to represent the function variation over the element in the global coordinates. With the proper procedures, the integral that was written in (3) can be rewritten in terms of integration over the local element as

$$K_{ij}^e = \int_{-1}^{1}\int_{-1}^{1}\left(\frac{\partial N_i^e}{\partial x}k\frac{\partial N_j^e}{\partial x} + \frac{\partial N_i^e}{\partial y}k\frac{\partial N_j^e}{\partial y}\right) det(\mathbf{J}) d\xi d\eta \tag{6}$$

Where the derivatives with respect to the global coordinates can be obtained in terms of the local coordinates by means of the Jacobian. It is therefore necessary to know the shape functions in terms of the local coordinates and also the proper parametric mapping, which relates the global coordinates to the local coordinates through the following relationship valid for a finite element shape function of a (M+1) noded element.

$$\begin{cases} x = N_0^e x_0 + N_1^e x_1 + ... + N_M^e x_M \\ y = N_0^e y_0 + N_1^e y_1 + ... + N_M^e y_M \end{cases} \tag{7}$$

Where (x_i, y_i) are the global coordinates of the point into which we want to map node i of the element into the local space. Note also that $N_i^e = N_i^e(\xi, \eta)$.

In this way all the element matrices required can be evaluated if the integrals are simple enough, which may not be the case as the code is intended for a general case. In such a case it will also be necessary to employ numerical integration, where the integrals are approximated by using a simple summation of terms at specific points of the domain and multiplied by specific weights.

2.6.2 Numerical integration

Integrating shape functions of high order may be a difficult task, hence making precise the use of Gauss quadrature, which can integrate exactly polynomials of a certain degree. Since the element type and the order of the shape functions is already known, a function (2.6.2) may be implemented that returns the weights and the coordinated of the integration points that integrate exactly the required integral.

6

`gauss_quadrature`

Brief description: Returns the number of integration Gauss points, their coordinates and their weights based on the user's choices.
Input variables: Type of element (`Problem.elem_type`), order of the shape functions (`Problem.order`).
Output variables: `gauss` structure, containing the points of integration and the weights, `Gauss.points` and `Gauss.weights`
Matlab built-in functions used:.

2.6.3 Resolution

The resolution of the system of equations is achieved once all the components of equation (3) are known. In this sense, algorithm 1 summarizes the main procedures to compute the stiffness matrix of each element. This algorithm could be implemented inside function . a very similar algorithm could be implemented for the vector forces, and defined inside the same function.

7

`k_matrix`

Brief description: Computes the stiffness matrix of the elements.
Input variables: Shape functions (`N`), connectivity matrix (`T`), element number, conductivity coefficient (`k`).
Output variables: Elemental matrix (`K`).
Matlab built-in functions used:.
Function calls to: `gauss_quadrature`.

Then, to complete the process, function 2.6.3 would have the task to allocate each elemental matrix to a global matrix provided the connectivity matrix.

8

`allocation`

Brief description: Allocates the elemental matrix coefficients to their proper positions in the master stiffness matrix .
Input variables: Elemental matrix (`K`), connectivity matrix(`T`), element number, master stiffness matrix (`Global_K`).
Output variables: Updated master stiffness matrix (`Global_K`)
Matlab built-in functions used:.

The sampling points and their weights have to be determined previously, before entering the iterating loop, once the choices of the user have been made regarding the order and the type of element.

Algorithm 1 Computer programming of $\mathbf{K^e}$

1: **procedure** (for n:1:Ne) ▷ Loop over all elements
2: **procedure** (for i,j:1:n_p, n_q) ▷ Loop over all integration points
3: $\frac{\partial N_i}{\partial \xi}$
4: $\frac{\partial N_i}{\partial \eta}$ ▷ Compute the shape functions and natural derivatives at integration points
5: $\mathbf{J^e}$ ▷ Compute the Jacobian matrix at each integration points
6: $\frac{\partial N_i}{\partial x}$
7: $\frac{\partial N_i}{\partial y}$ ▷ Compute the cartesian derivatives of the shape functions at integration points
8: $\mathbf{K^e_{ij}} = \sum_{j=0}^{n} \left(W_j \sum_{i=0}^{n} W_i G(\xi_i, \eta_j)\right)$
9: **end procedure**
10: **end procedure**

2.7 Post-processing

Once the results are obtained, the characteristic function can be plotted in the results window of the application. This function would simply get as input the solution (`Problem.solution`) and plot is using a Matlab function such as `pcolor` or `surf`.

3 Code design

3.1 Structures

A Structure is a named collection of data representing a single idea or "object". For anything in a computer more complicated than a list of numbers, structures can be used. Inside a structure are a list of fields each being a variable name for some sub-piece of data. Structures are similar to arrays in that they contain multiple data, but the main difference is, instead of an Index to each piece of data, we have a "name"; and instead of every piece of data being the same type, we can have a different type for each "field".

Problem structure

- **`dimension = Problem.dimension(possible values ∈ [2,3])`**, where 2 is referred to a 2D problem and 3 to a 3D problem.
- **`geometry = Problem.geometry(possible values ∈ [1,2,3])`**. For 2D domain: Rectangle (1), Ring(2). For 3D domain: Concentric cylinders (1), Brick (2), Semi-sphere (3)
- **`type_element = Problem.type_element(possible values ∈ [1,2])`** For a 2D problem, 1 is referred to triangular element and 2 for a quadrilateral element. For a 3D problem, 1 is referred to a tetrahedral element and 2 to a hexahedral element.
- **`order = Problem.order(possible values ∈ [1,2,3])`**; where 1 is linear, 2 is quadratic and 3 is cubic order.
- **`X = Problem.X`**. Variable to define the coordinates of the geometry.
- **`h = Problem.h`**. Variable to define the element size of the mesh.
- **`nodesBC_dir = Problem.bc_dirichlet`** (list of the nodes with its field BC value)
- **`nodesBC_neu = Problem.bc_neummann`** (list of the nodes with its field BC value)
- **`nodes_loads = Problem.loads`** (list of the nodes with its sources or loads)
- **`mat_properties = Problem.material`** (constant value or function(position,u field) of the material properties)

- **`solution = Problem.solution`** (Vector of solution of the system of equations, $\mathbf{a} = \mathbf{K}^{-1}\mathbf{f}$)

Mesh structure

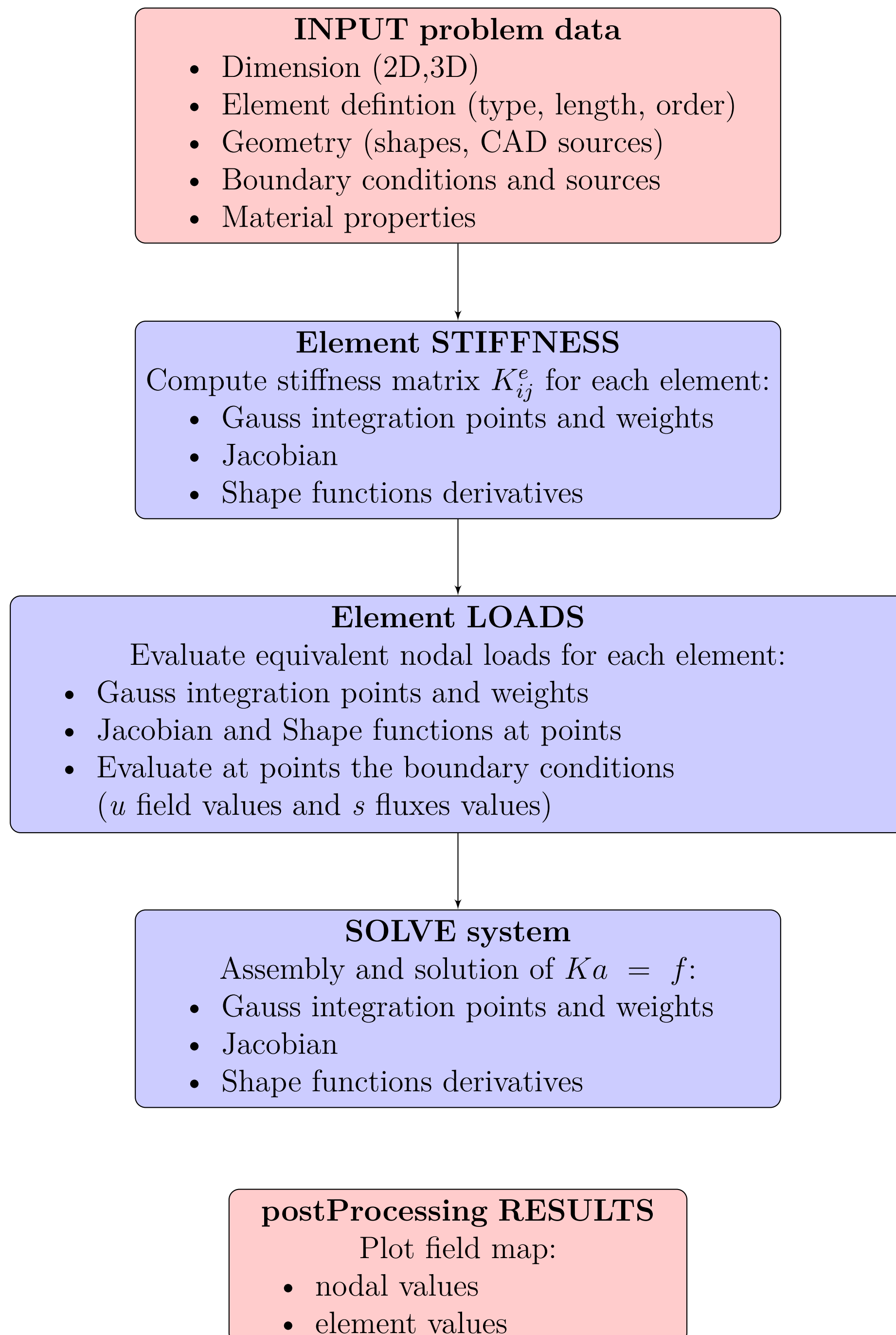
- **`nodes = Mesh.nodes`**. Variable to define the coordinates of the nodes of the mesh.
- **`elements = Mesh.elements`**. Variable to define the connectivity matrix of the mesh.

Gauss structure

- `points = Gauss.points`. Coordinates of the points of integration.
- `weights = Gauss.weights`. Weights of the points of integration.

3.2 Workflow

The above mentioned procedures can be summarized in the following workflow diagram.


INPUT problem data
Dimension (2D,3D)
Element defintion (type, length, order)
Geometry (shapes, CAD sources)
Boundary conditions and sources
Material properties
Element STIFFNESS
Compute stiffness matrix K_{ij}^{e} for each element:
Gauss integration points and weights
Jacobian
Shape functions derivatives
Element LOADS
Evaluate equivalent nodal loads for each element:
Gauss integration points and weights
Jacobian and Shape functions at points
Evaluate at points the boundary conditions
(u field values and s fluxes values)
SOLVE system
Assembly and solution of $Ka = f$:
Gauss integration points and weights
Jacobian
Shape functions derivatives
postProcessing RESULTS
Plot field map:
nodal values
element values

4 Concluding remarks

This work has established the procedure and methodology for constructing a general finite element code to solve the Poisson equation in a arbitrary domain of 2D or 3D dimensions. The functions to be used has been defined as a way to clarify all the procedure to work out the solution.

In spite of this proposed resolution for the code, some slight variations might take place when the coding process starts. There might be different efficient ways to implement the code that will be showed up once the bugging process of the code is done.

References

[1] O.C. Zienkiewics and K. Morgan, *Finite Elements and Approximation*. Dover books ISNB 0-486-4530149 (1983).