UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master on Numerical Methods in Engineering

# HOMEWORK 2

Course: Programming for Engineers and Scientists

Name: Yonatan Lisne Luque Apaza

May 2017

## 1. Introduction

This report accompanies the solver developed in c++ for the solution of the poisson equation in 2D and 3D, according to the course requirements.
The implemented code is located in the source folder, and the objects compiled into the build folder.

## 2. Theoretical formulation

### 2.1. Poisson Equation

The poisson equation is given by:

$$\boldsymbol{\nabla}.(D\boldsymbol{\nabla}\phi) + Q = 0 \qquad in\ \Omega$$

$$\phi - \bar{\phi} = 0 \qquad on\ \Gamma_d$$

$$\boldsymbol{n}.\boldsymbol{q} = q_n + \alpha(\phi - \phi_q) \qquad on\ \Gamma_n$$

Where:

$$\boldsymbol{q} = -D\boldsymbol{\nabla}\phi$$

### 2.2. Weak form

According to Galerkin's formulation, the weak form is:

$$\int_{\Omega} (\boldsymbol{\nabla}N_i).\left(D(\boldsymbol{\nabla}N_j)\phi_j\right) d\Omega + \int_{\Gamma_n} N_i \alpha N_j \phi_j \, d\Gamma = \int_{\Omega} N_i Q \, d\Omega - \int_{\Gamma_n} N_i(q_n - \alpha\phi_q) d\Gamma + r_{\Gamma_d}$$

Where:

$$\phi = \sum_{j}^{n} N_j \phi_j = N_j \phi_j$$

$$\boldsymbol{\nabla}\phi = \sum_{j}^{n} (\boldsymbol{\nabla}N_j)\phi_j = (\boldsymbol{\nabla}N_j)\phi_j$$

For each element:

$$K_{ij}^{(e)} = \int_{\Omega^{(e)}} D\left(\boldsymbol{\nabla}N_i^{(e)}\right).\left(\boldsymbol{\nabla}N_j^{(e)}\right) d\Omega + \int_{\Gamma_n^{(e)}} \alpha N_i^{(e)} N_j^{(e)} \, d\Gamma$$

$$f_i^{(e)} = \int_{\Omega^{(e)}} N_i^{(e)} Q \, d\Omega - \int_{\Gamma_n^{(e)}} N_i^{(e)}(q_n - \alpha\phi_q) d\Gamma + r_{\Gamma_d}^{(e)}$$

### 2.3. Matrix form

#### 2.3.1. Basic Matrices

Considering an element of $ne$ nodes, the values will be stored in the following matrices:

$$\boldsymbol{X}^{(e)} = \begin{bmatrix} x_1^{(e)} & y_1^{(e)} & z_1^{(e)} \\ x_2^{(e)} & y_2^{(e)} & z_2^{(e)} \\ \vdots & \vdots & \vdots \\ x_{ne}^{(e)} & y_{ne}^{(e)} & z_{ne}^{(e)} \end{bmatrix} \qquad \boldsymbol{\phi}^{(e)} = \begin{bmatrix} \phi_1^{(e)} \\ \phi_2^{(e)} \\ \vdots \\ \phi_{ne}^{(e)} \end{bmatrix}$$

Global shape functions are stored in the following matrices:

$$\boldsymbol{N}^{(e)} = \begin{bmatrix} N_1^{(e)} & N_2^{(e)} & \dots & N_{ne}^{(e)} \end{bmatrix}$$

$$\boldsymbol{B}^{(e)} = \boldsymbol{\nabla N}^{(e)} = \begin{bmatrix} \frac{dN_1^{(e)}}{dx} & \frac{dN_2^{(e)}}{dx} & \cdots & \frac{dN_{ne}^{(e)}}{dx} \\ \frac{dN_1^{(e)}}{dy} & \frac{dN_2^{(e)}}{dy} & \cdots & \frac{dN_{ne}^{(e)}}{dy} \\ \frac{dN_1^{(e)}}{dz} & \frac{dN_2^{(e)}}{dz} & \cdots & \frac{dN_{ne}^{(e)}}{dz} \end{bmatrix}$$

Local shape functions are stored in the following matrices:

$$\boldsymbol{N}_{\xi\eta\zeta}^{(e)} = \begin{bmatrix} N_1^{(e)} & N_2^{(e)} & \dots & N_{ne}^{(e)} \end{bmatrix}$$

$$\boldsymbol{B}_{\xi\eta\zeta}^{(e)} = \boldsymbol{\nabla}_{\xi\eta\zeta}\boldsymbol{N}^{(e)} = \begin{bmatrix} \frac{\partial N_1^{(e)}}{\partial\xi} & \frac{\partial N_2^{(e)}}{\partial\xi} & \cdots & \frac{\partial N_{ne}^{(e)}}{\partial\xi} \\ \frac{\partial N_1^{(e)}}{\partial\eta} & \frac{\partial N_2^{(e)}}{\partial\eta} & \cdots & \frac{\partial N_{ne}^{(e)}}{\partial\eta} \\ \frac{\partial N_1^{(e)}}{\partial\zeta} & \frac{\partial N_2^{(e)}}{\partial\zeta} & \cdots & \frac{\partial N_{ne}^{(e)}}{\partial\zeta} \end{bmatrix}$$

The Jacobian matrix is stored as the following matrix:

$$\boldsymbol{J}^{(e)} = \begin{bmatrix} \frac{\partial x}{\partial\xi} & \frac{\partial y}{\partial\xi} & \frac{\partial z}{\partial\xi} \\ \frac{\partial x}{\partial\eta} & \frac{\partial y}{\partial\eta} & \frac{\partial z}{\partial\eta} \\ \frac{\partial x}{\partial\zeta} & \frac{\partial y}{\partial\zeta} & \frac{\partial z}{\partial\zeta} \end{bmatrix}$$

Where:

$$\boldsymbol{J}^{(e)} = \boldsymbol{B}_{\xi\eta\zeta}^{(e)}\boldsymbol{X}^{(e)}$$

The inverse of the Jacobian matrix depends on the dimension of the element and is stored as the following matrix:

$$\boldsymbol{J^{-1}}_{(3D)}^{(e)} = \begin{bmatrix} \frac{\partial x}{\partial\xi} & \frac{\partial y}{\partial\xi} & \frac{\partial z}{\partial\xi} \\ \frac{\partial x}{\partial\eta} & \frac{\partial y}{\partial\eta} & \frac{\partial z}{\partial\eta} \\ \frac{\partial x}{\partial\zeta} & \frac{\partial y}{\partial\zeta} & \frac{\partial z}{\partial\zeta} \end{bmatrix}^{\boldsymbol{-1}}$$

$$\boldsymbol{J^{-1}}_{(2D)}^{(e)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial\xi} & \frac{\partial y}{\partial\xi} \\ \frac{\partial x}{\partial\eta} & \frac{\partial y}{\partial\eta} \end{bmatrix}^{\boldsymbol{-1}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}^T$$

$$\boldsymbol{J^{-1}}_{(1D)}^{(e)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial\xi} \end{bmatrix}^{\boldsymbol{-1}} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}^T$$

The Jacobian will be determined according to the dimension of the element:

$$|\boldsymbol{J}|_{3D}^{(e)} = \boldsymbol{det}(\boldsymbol{J}^{(e)})$$

$$|\boldsymbol{J}|_{2D}^{(e)} = \left\| \left(\frac{\partial x}{\partial \xi}, \frac{\partial y}{\partial \xi}, \frac{\partial z}{\partial \xi}\right) \times \left(\frac{\partial x}{\partial \eta}, \frac{\partial y}{\partial \eta}, \frac{\partial z}{\partial \eta}\right) \right\|_{\mathbf{2}}$$

$$|\boldsymbol{J}|_{1D}^{(e)} = \left\| \left(\frac{\partial x}{\partial \xi}, \frac{\partial y}{\partial \xi}, \frac{\partial z}{\partial \xi}\right) \right\|_{\mathbf{2}}$$

Other equations:

$$[x \quad y \quad z]^{(e)} = \boldsymbol{N}^{(e)}\boldsymbol{X}^{(e)}$$
$$\nabla\phi^{(e)} = \boldsymbol{B}^{(e)}\boldsymbol{\phi}^{(e)}$$
$$\phi^{(e)} = \boldsymbol{N}^{(e)}\boldsymbol{\phi}^{(e)}$$

Where:

$$\boldsymbol{B}^{(e)} = \boldsymbol{J^{-1}}^{(e)}\boldsymbol{B}_{\xi\eta\zeta}^{(e)}$$

### 2.3.2. Assembly matrix and vector

For the assembly of the matrices in each element we can consider dividing the matrix and vector of each element for the global assembly and then the boundary conditions:

$$\boldsymbol{K}^{(e)} = \boldsymbol{K}_{\boldsymbol{\Omega}}^{(e)} + \boldsymbol{K}_{\Gamma_n}^{(e)}$$
$$\boldsymbol{f}^{(e)} = \boldsymbol{f}_{\boldsymbol{\Omega}}^{(e)} + \boldsymbol{f}_{\Gamma_n}^{(e)} + \boldsymbol{r}_{\Gamma_d}^{(e)}$$

According to the matrix form and considering the integration by quadrature with points $\boldsymbol{z}_g$ and weights $w_g$, the matrices and vectors of each element

$$\boldsymbol{K}_{\boldsymbol{\Omega}}^{(e)} = \int_{\Omega_{\xi\eta\zeta}^{(e)}} D\left(\boldsymbol{B}^{(e)}\right)^{\boldsymbol{T}}\left(\boldsymbol{B}^{(e)}\right) \left|\boldsymbol{J}^{(e)}\right| d\Omega_{\xi\eta\zeta} = \sum_{g=1}^{n_g} w_g\, D_{(\boldsymbol{z}_g)} \left(\boldsymbol{B}^{(e)}{}_{(\boldsymbol{z}_g)}\right)^{\boldsymbol{T}} \left(\boldsymbol{B}^{(e)}{}_{(\boldsymbol{z}_g)}\right) \left|\boldsymbol{J}^{(e)}\right|_{(\boldsymbol{z}_g)}$$

$$\boldsymbol{K}_{\Gamma_n}^{(e)} = \int_{\Gamma_{n_{\xi\eta\zeta}}^{(e)}} \alpha(\boldsymbol{N}^{(e)})^{\boldsymbol{T}}(\boldsymbol{N}^{(e)}) \left|\boldsymbol{J}^{(e)}\right| d\Gamma = \sum_{g=1}^{n_g} w_g \alpha_{(\boldsymbol{z}_g)} (\boldsymbol{N}^{(e)}{}_{(\boldsymbol{z}_g)})^{\boldsymbol{T}} (\boldsymbol{N}^{(e)}{}_{(\boldsymbol{z}_g)}) \left|\boldsymbol{J}^{(e)}\right|_{(\boldsymbol{z}_g)}$$

$$\boldsymbol{f}_{\boldsymbol{\Omega}}^{(e)} = \int_{\Omega_{\xi\eta\zeta}^{(e)}} (\boldsymbol{N}^{(e)})^{\boldsymbol{T}} Q \left|\boldsymbol{J}^{(e)}\right| d\Omega_{\xi\eta} = \sum_{g=1}^{n_g} w_g\, (\boldsymbol{N}^{(e)}{}_{(\boldsymbol{z}_g)})^{\boldsymbol{T}} Q_{(\boldsymbol{z}_g)} \left|\boldsymbol{J}^{(e)}\right|_{(\boldsymbol{z}_g)}$$

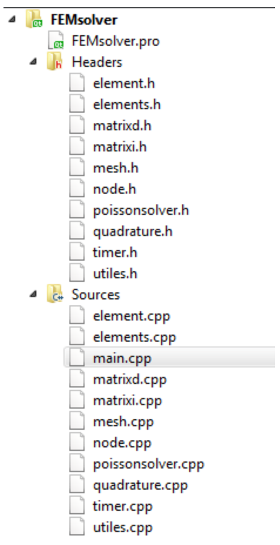
$$\boldsymbol{f}_{\Gamma_n}^{(e)} = -\int_{\Gamma_{n_{\xi\eta\zeta}}^{(e)}} (\boldsymbol{N}^{(e)})^{\boldsymbol{T}} \left(q_n - \alpha\phi_q\right) \left|\boldsymbol{J}^{(e)}\right| d\Gamma = -\sum_{g=1}^{n_g} w_g\, (\boldsymbol{N}^{(e)}{}_{(\boldsymbol{z}_g)})^{\boldsymbol{T}} \left(q_{n_{(\boldsymbol{z}_g)}} - \alpha_{(\boldsymbol{z}_g)} \phi_{q_{(\boldsymbol{z}_g)}}\right) \left|\boldsymbol{J}^{(e)}\right|_{(\boldsymbol{z}_g)}$$

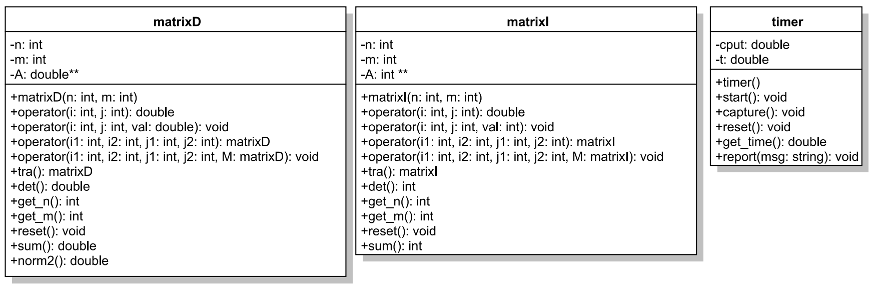
Not necessary to obtain the vector $\boldsymbol{r}_{\Gamma_d}^{(e)}$, because we will replace the equations where the variables $\phi_j$ are previously established by equations of equality $\phi_j = \overline{\phi}_j$, Then we will replace these variables in the global vector to preserve the symmetric global matrix..

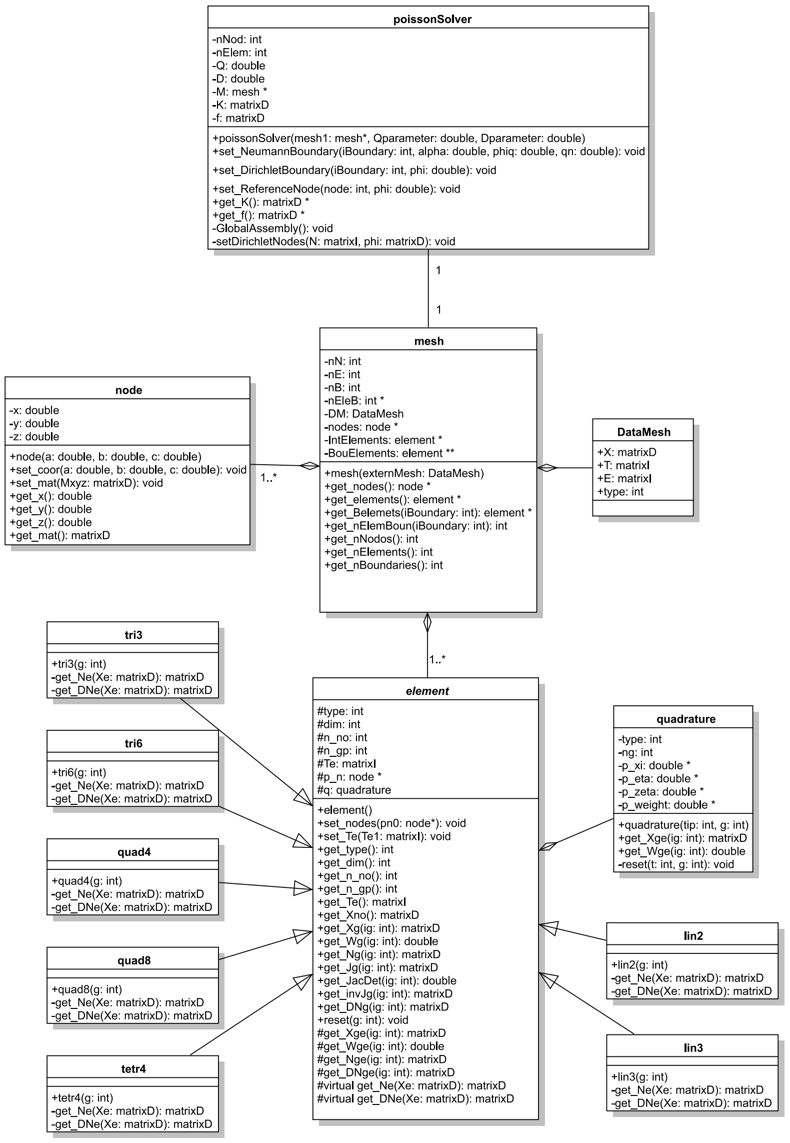
## 3. FEM C++ program

The program implemented having the structure shown in the right image.

The structure of the classes is shown in the lower UML diagram.

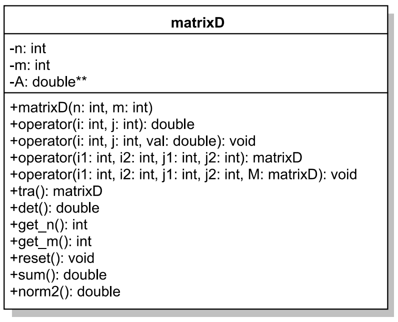



matrixD
-n: int
-m: int
-A: double**
+matrixD(n: int, m: int)
+operator(i: int, j: int): double
+operator(i: int, j: int, val: double): void
+operator(i1: int, i2: int, j1: int, j2: int): matrixD
+operator(i1: int, i2: int, j1: int, j2: int, M: matrixD): void
+tra(): matrixD
+det(): double
+get_n(): int
+get_m(): int
+reset(): void
+sum(): double
+norm2(): double
matrixI
-n: int
-m: int
-A: int **
+matrixI(n: int, m: int)
+operator(i: int, j: int): double
+operator(i: int, j: int, val: int): void
+operator(i1: int, i2: int, j1: int, j2: int): matrixI
+operator(i1: int, i2: int, j1: int, j2: int, M: matrixI): void
+tra(): matrixI
+det(): int
+get_n(): int
+get_m(): int
+reset(): void
+sum(): int
timer
-cput: double
-t: double
+timer()
+start(): void
+capture(): void
+reset(): void
+get_time(): double
+report(msg: string): void


poissonSolver
-nNod: int
-nElem: int
-Q: double
-D: double
-M: mesh *
-K: matrixD
-f: matrixD
+poissonSolver(mesh1: mesh*, Qparameter: double, Dparameter: double)
+set_NeumannBoundary(iBoundary: int, alpha: double, phiq: double, qn: double): void
+set_DirichletBoundary(iBoundary: int, phi: double): void
+set_ReferenceNode(node: int, phi: double): void
+get_K(): matrixD *
+get_f(): matrixD *
-GlobalAssembly(): void
-setDirichletNodes(N: matrixI, phi: matrixD): void
1
1
mesh
-nN: int
-nE: int
-nB: int
-nEleB: int *
-DM: DataMesh
-nodes: node *
-IntElements: element *
-BouElements: element **
+mesh(externMesh: DataMesh)
+get_nodes(): node *
+get_elements(): element *
+get_Belemets(iBoundary: int): element *
+get_nElemBoun(iBoundary: int): int
+get_nNodos(): int
+get_nElements(): int
+get_nBoundaries(): int
node
-x: double
-y: double
-z: double
+node(a: double, b: double, c: double)
+set_coor(a: double, b: double, c: double): void
+set_mat(Mxyz: matrixD): void
+get_x(): double
+get_y(): double
+get_z(): double
+get_mat(): matrixD
1..*
DataMesh
+X: matrixD
+T: matrixI
+E: matrixI
+type: int
1..*
element
#type: int
#dim: int
#n_no: int
#n_gp: int
#Te: matrixI
#p_n: node *
#q: quadrature
+element()
+set_nodes(pn0: node*): void
+set_Te(Te1: matrixI): void
+get_type(): int
+get_dim(): int
+get_n_no(): int
+get_n_gp(): int
+get_Te(): matrixI
+get_Xno(): matrixD
+get_Xg(ig: int): matrixD
+get_Wg(ig: int): double
+get_Ng(ig: int): matrixD
+get_Jg(ig: int): matrixD
+get_JacDet(ig: int): double
+get_invJg(ig: int): matrixD
+get_DNg(ig: int): matrixD
+reset(g: int): void
#get_Xge(ig: int): matrixD
#get_Wge(ig: int): double
#get_Nge(ig: int): matrixD
#get_DNge(ig: int): matrixD
#virtual get_Ne(Xe: matrixD): matrixD
#virtual get_DNe(Xe: matrixD): matrixD
quadrature
-type: int
-ng: int
-p_xi: double *
-p_eta: double *
-p_zeta: double *
-p_weight: double *
+quadrature(tip: int, g: int)
+get_Xge(ig: int): matrixD
+get_Wge(ig: int): double
-reset(t: int, g: int): void
tri3
+tri3(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
tri6
+tri6(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
quad4
+quad4(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
quad8
+quad8(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
tetr4
+tetr4(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
lin2
+lin2(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD
lin3
+lin3(g: int)
-get_Ne(Xe: matrixD): matrixD
-get_DNe(Xe: matrixD): matrixD

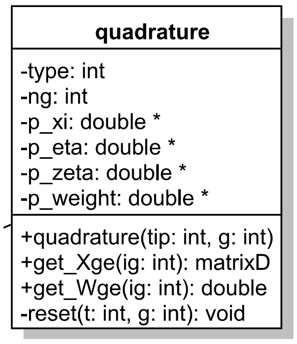
## 3.1. Main classes

### 3.1.1. MatrixD and matrixI


matrixD
-n: int
-m: int
-A: double**
+matrixD(n: int, m: int)
+operator(i: int, j: int): double
+operator(i: int, j: int, val: double): void
+operator(i1: int, i2: int, j1: int, j2: int): matrixD
+operator(i1: int, i2: int, j1: int, j2: int, M: matrixD): void
+tra(): matrixD
+det(): double
+get_n(): int
+get_m(): int
+reset(): void
+sum(): double
+norm2(): double

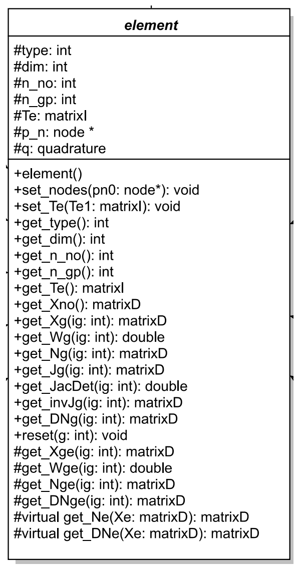
- They are basic classes and store matrices with values type double and int respectively.
- They are used throughout the program.
- Main functions:
  - $\det() \Rightarrow$ determinant.
  - $\text{sum}() \Rightarrow$ Sum of all elements.
  - $\text{norm}2() \Rightarrow$ Euclidean norm.

### 3.1.2. Quadrature


quadrature
-type: int
-ng: int
-p_xi: double *
-p_eta: double *
-p_zeta: double *
-p_weight: double *
+quadrature(tip: int, g: int)
+get_Xge(ig: int): matrixD
+get_Wge(ig: int): double
-reset(t: int, g: int): void

- Stores quadrature points and weights.
- Initialized with element type and quadrature number.
- Main functions:
  - $\text{get\_Xge}(i_g) \Rightarrow [\xi_{i_g} \quad \eta_{i_g} \quad \zeta_{i_g}]$.
  - $\text{get\_Wge}(i_g) \Rightarrow w_{i_g}$, weight in $i_g$.
- $i_g$ Is the quadrature point index.

### 3.1.3. Element


element
#type: int
#dim: int
#n_no: int
#n_gp: int
#Te: matrixI
#p_n: node *
#q: quadrature
+element()
+set_nodes(pn0: node*): void
+set_Te(Te1: matrixI): void
+get_type(): int
+get_dim(): int
+get_n_no(): int
+get_n_gp(): int
+get_Te(): matrixI
+get_Xno(): matrixD
+get_Xg(ig: int): matrixD
+get_Wg(ig: int): double
+get_Ng(ig: int): matrixD
+get_Jg(ig: int): matrixD
+get_JacDet(ig: int): double
+get_invJg(ig: int): matrixD
+get_DNg(ig: int): matrixD
+reset(g: int): void
#get_Xge(ig: int): matrixD
#get_Wge(ig: int): double
#get_Nge(ig: int): matrixD
#get_DNge(ig: int): matrixD
#virtual get_Ne(Xe: matrixD): matrixD
#virtual get_DNe(Xe: matrixD): matrixD

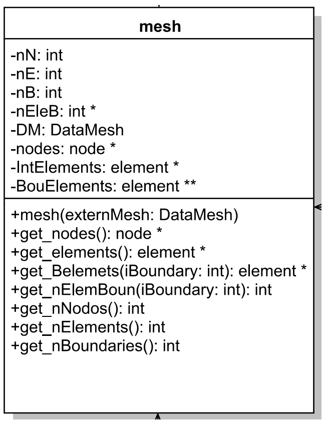
- Abstract class, has the attributes and functions of a general element.
- All defined element types inherit from this class. The attributes defined when initializing the inherited classes are according to table 1.
- All inherited elements initialize with the quadrature number.
- Main Attributes:
  - $type$: Type of element. Matches the VTK format.
  - $dim$: Dimension of the element.
  - $n\_no$: Number of element nodes.
  - $n\_gp$: Number of quadrature points.
- Main functions:
  - $\text{set\_nodes}() \Leftarrow$ Initial node pointer.
  - $\text{set\_Te}() \Leftarrow [n_1^{(e)} \quad n_2^{(e)} \quad \cdots \quad n_{ne}^{(e)}]$
  - $\text{get\_Te}() \Rightarrow [n_1^{(e)} \quad n_2^{(e)} \quad \cdots \quad n_{ne}^{(e)}]$
  - $\text{get\_Xno}() \Rightarrow \boldsymbol{X}^{(e)}$
  - $\text{get\_Ng}(i_g) \Rightarrow \boldsymbol{N}^{(e)}$ in $i_g$
  - $\text{get\_DNg}(i_g) \Rightarrow \boldsymbol{B}^{(e)}$ in $i_g$
  - $\text{get\_Jg}(i_g) \Rightarrow \boldsymbol{J}^{(e)}$ in $i_g$
  - $\text{get\_JacDet}(i_g) \Rightarrow |\boldsymbol{J}^{(e)}|$ in $i_g$

- get_invJg($i_g$) $\Longrightarrow \boldsymbol{J^{-1}}^{(e)}$ in $i_g$
- get_Ne($X_e$) $\Longrightarrow$ $\boldsymbol{N}_{\xi\eta\zeta}^{(e)}$ in $[\xi \quad \eta \quad \zeta]$, defined in the inherited class.
- get_DNe($X_e$) $\Longrightarrow$ $\boldsymbol{B}_{\xi\eta\zeta}^{(e)}$ in $[\xi \quad \eta \quad \zeta]$, defined in the inherited class.

| **Class:** | **lin2** | **lin3** | **tri3** | **tri6** | **quad4** | **quad8** | **tetr4** |
|---|---|---|---|---|---|---|---|
| type | 3 | 21 | 5 | 22 | 9 | 23 | 10 |
| dim | 1 | 1 | 2 | 2 | 2 | 2 | 3 |
| n_no | 2 | 3 | 3 | 6 | 4 | 8 | 4 |
| n_gp | 1,2,3,4 | 1,2,3,4 | 1,3,4,6 | 1,3,4,6 | 1,4,9 | 1,4,9 | 1,4,5 |

**Table 1: Attributes in the derived class.**

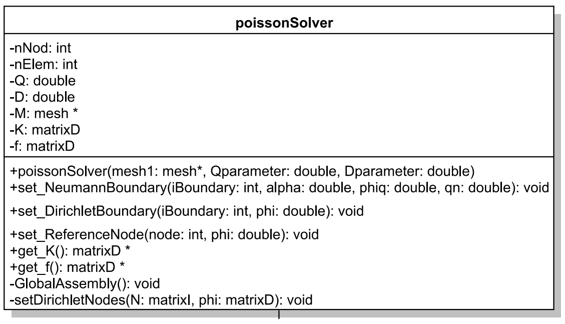
## 3.1.4. Mesh



```
mesh
-nN: int
-nE: int
-nB: int
-nEleB: int *
-DM: DataMesh
-nodes: node *
-IntElements: element *
-BouElements: element **
+mesh(externMesh: DataMesh)
+get_nodes(): node *
+get_elements(): element *
+get_Belemets(iBoundary: int): element *
+get_nElemBoun(iBoundary: int): int
+get_nNodos(): int
+get_nElements(): int
+get_nBoundaries(): int
```

- Stores all nodes and elements.
- Initializes with a DataMesh object that contains the data of the imported mesh.
- Main Attributes:
    - $nodes$: Array of node objects.
    - $IntElements$: Array of element objects, corresponding to the inner elements.
    - $BouElements$: Double array of element objects, corresponding to the contour elements.
- Main functions:
    - $get\_nBoundaries()$ $\Longrightarrow$ Number of contours.
    - $get\_nElemBoun(iBoundary)$ $\Longrightarrow$ Number of elements of the contour $i_{Boundary}$.
    - $get\_elements()$ $\Longrightarrow$ Pointer to element 0.
    - $get\_Belements(iBoundary)$ $\Longrightarrow$ Pointer to element 0 of the contour $i_{Boundary}$.

## 3.1.5. PoissonSolver



```
poissonSolver
-nNod: int
-nElem: int
-Q: double
-D: double
-M: mesh *
-K: matrixD
-f: matrixD
+poissonSolver(mesh1: mesh*, Qparameter: double, Dparameter: double)
+set_NeumannBoundary(iBoundary: int, alpha: double, phiq: double, qn: double): void
+set_DirichletBoundary(iBoundary: int, phi: double): void
+set_ReferenceNode(node: int, phi: double): void
+get_K(): matrixD *
+get_f(): matrixD *
-GlobalAssembly(): void
-setDirichletNodes(N: matrixI, phi: matrixD): void
```

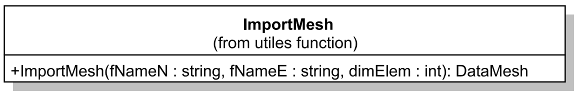
- Generate global matrix and vector K and f.
- Initialize with a mesh object and the values of Q and D.
- Main Attributes
    - $Q, D$: Parameter of the Poisson equation.
    - $M$: Mesh object pointer.
    - $K$: Global matrix.
    - $f$: Global vector.
- Main functions:
    - $set\_NeumannBoundary()$ $\Longleftarrow$ Set the boundary $i_{Boundary}$ as Neumann type with $\alpha, \phi_q, q_n$, values .

- set_DirichletBoundary() $\Longleftarrow$ Set the boundary $i_{Boundary}$ as Dirichlet type with $\bar{\phi}$ value.
- set_ReferenceNode() $\Longleftarrow$ Set reference node with $\bar{\phi}$ value.

## 3.2.Auxiliary functions

### 3.2.1. ImportMesh

Import the mesh of the stored formats for the homework and returns a DataMesh object. Divide the contours elements into three parts: elements found at x = 0, elements found at x = 1 and others.

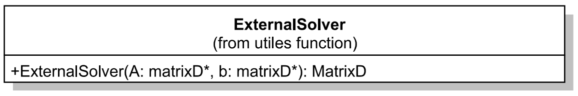


**ImportMesh**
(from utiles function)

+ImportMesh(fNameN : string, fNameE : string, dimElem : int): DataMesh

### 3.2.2. ExternalSolver

Solve a linear system $\boldsymbol{A\phi} = \boldsymbol{b}$ using eigen libraries. Returns the solution as a MatrixD object.

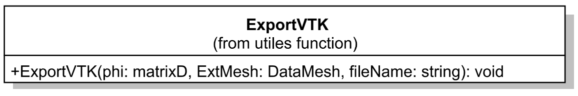


**ExternalSolver**
(from utiles function)

+ExternalSolver(A: matrixD*, b: matrixD*): MatrixD

### 3.2.3. ExportVTK

Export the mesh and solution to a vtk format.



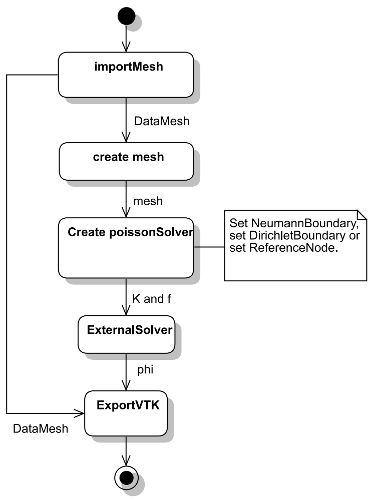
**ExportVTK**
(from utiles function)

+ExportVTK(phi: matrixD, ExtMesh: DataMesh, fileName: string): void

## 3.3.Solution process

The solution process of a problem:

- Import a mesh with function ImportMesh() .
- Create a mesh object with the DataMesh of the import.
- Create a poissonSolver object, set Neumann and Dirichlet boundary conditions.
- Solve the linear system with the function ExternalSolver() with K and f of the poissonSolver.
- Export the solution with the function ExportVTK.



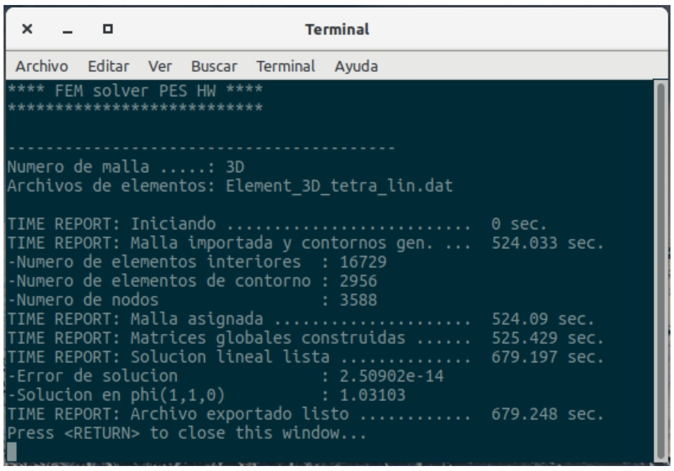
As an example of this process we create the following function that solves the homework 3D problem:

```
void HW3Dsolver()
{
    string fn, fe;
    fn="HW1 Meshes/3D-tetra-lin/Node_3D_tetra_lin.dat";
    fe="HW1 Meshes/3D-tetra-lin/Element_3D_tetra_lin.dat";
    DataMesh ExternMesh=ImportMesh(fn,fe,3);
    mesh mesh1(ExternMesh);
    poissonSolver solver1(&mesh1,0.0,1.0);
    solver1.set_NeumannBoundary(0,0,0,0);
    solver1.set_NeumannBoundary(1,0,0,1);
    solver1.set_NeumannBoundary(2,0,0,-1);
    solver1.set_ReferenceNode(getNodo000(ExternMesh.X),0);
    matrixD R=ExternalSolver(&KK,solver1.get_f());
    ExportVTK( R, ExternMesh, fileName);
}
```

Adding auxiliary objects we will have:

```cpp
void HW3Dsolver()
{
    timer t1;
    string fn, fe;
    fn="HW1 Meshes/3D-tetra-lin/Node_3D_tetra_lin.dat";
    fe="HW1 Meshes/3D-tetra-lin/Element_3D_tetra_lin.dat";
    cout<<endl<<"----------------------------------------- "<<endl;
    cout<<"Numero de malla .....: "<<"3D"<<endl;
    cout<<"Archivos de elementos: "<<"Element_3D_tetra_lin.dat"<<endl<<endl;
    t1.reset();
    t1.start();
    t1.report("Iniciando ..........................");
    DataMesh ExternMesh=ImportMesh(fn,fe,3);
    t1.capture();
    t1.report("Malla importada y contornos gen. ...");
    cout<<"-Numero de elementos interiores  : "<<ExternMesh.T.get_n()<<endl;
    cout<<"-Numero de elementos de contorno : "<<ExternMesh.E.get_n()<<endl;
    cout<<"-Numero de nodos                 : "<<ExternMesh.X.get_n()<<endl;
    mesh mesh1(ExternMesh);
    t1.capture();
    t1.report("Malla asignada .....................");
    poissonSolver solver1(&mesh1,0.0,1.0);
    solver1.set_NeumannBoundary(0,0,0,0);
    solver1.set_NeumannBoundary(1,0,0,1);
    solver1.set_NeumannBoundary(2,0,0,-1);
    solver1.set_ReferenceNode(getNodo000(ExternMesh.X),0);
    matrixD KK=*solver1.get_K();
    KK(3333,3333,1); // El nodo 3333 no esta asignado a ningun elemento en la malla
                     //es decir no existe en el archivo de elementos. K(3333,3333)=1
                     // para que el sistema lineal tenga solucion.
    t1.capture();
    t1.report("Matrices globales construidas ......");
    matrixD R=ExternalSolver(&KK,solver1.get_f());
    double err=((KK)*(R)-(*solver1.get_f())).norm2()/(*solver1.get_f()).norm2();
    t1.capture();
    t1.report("Solucion lineal lista .............");
    cout<<"-Error de solucion               : "<<err<<endl;
    cout<<"-Solucion en phi(1,1,0)          : "<<R(getNodo110(ExternMesh.X),0)<<endl;
    string fileName="HW1 Meshes/3D-tetra-lin/solucionCPP";
    ExportVTK( R, ExternMesh, fileName);
    t1.capture();
    t1.report("Archivo exportado listo ...........");
}
```

The report shown in the terminal:

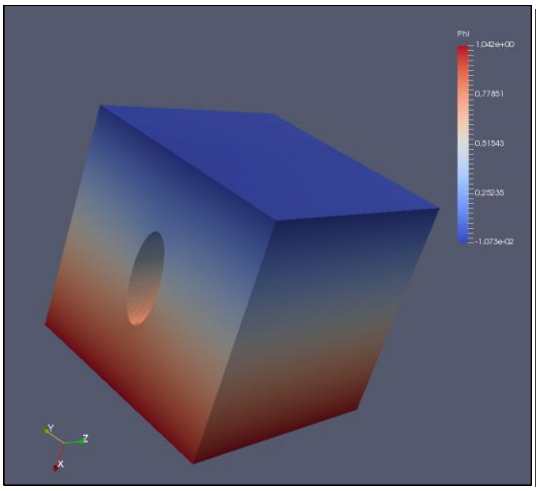


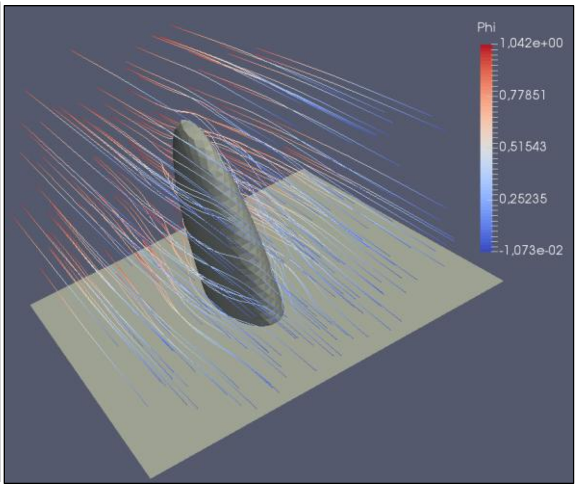
```
**** FEM solver PES HW ****
***************************


-----------------------------------------
Numero de malla .....: 3D
Archivos de elementos: Element_3D_tetra_lin.dat

TIME REPORT: Iniciando ..........................  0 sec.
TIME REPORT: Malla importada y contornos gen. ...  524.033 sec.
-Numero de elementos interiores  : 16729
-Numero de elementos de contorno : 2956
-Numero de nodos                 : 3588
TIME REPORT: Malla asignada .....................  524.09 sec.
TIME REPORT: Matrices globales construidas ......  525.429 sec.
TIME REPORT: Solucion lineal lista .............  679.197 sec.
-Error de solucion               : 2.50902e-14
-Solucion en phi(1,1,0)          : 1.03103
TIME REPORT: Archivo exportado listo ...........  679.248 sec.
Press <RETURN> to close this window...
```

Finally the post-processing can be done with paraview.





In the same way the solutions for the 2D meshes are implemented. The results of each of the meshes are attached in the same folder corresponding to the mesh of homework. The meshes and result are inside the folder:"/build/HW1 Meshes". The reports of all meshes are in the file "reporte de solucion 15-05-2017" in the same folder.

## 4. Conclusions

The implemented contour detection algorithm demands a lot of processing time. This can be improved by implementing another method.

The element class makes the handling of inherited class objects very practical.

The RAM memory used is very high. It must be implemented using the matrix type in the sparse matrix to reduce overall memory.

In the 3D-tetra-lin mesh the homework includes a node (3334) with no connection in any element, which causes the non-solubility of the linear system. We configure the globule matrix in the coordinates of this node to have a solutionable system.

In the implementation process, test functions were used to verify the behavior of the different parts.

## 5. References

*Zienkiewics, O.C., El método de Elementos Finitos, 3ra edición, Ed. Revert, Barcelona, 1979.*

*E. Oñate & P. Diez & F. Zarate & A. Larese, Introduction to the finite element method, 2008.*

*Rafael Payá Albert, Lección 8: Integrales de superficie, Asignatura: Fundamentos Matematicos I.*

*Rafael Payá Albert, Lección 4: Integrales de linea, Asignatura: Fundamentos Matematicos I.*

*Kitware. File Formats for VTK Version 4.2, 9-10.*