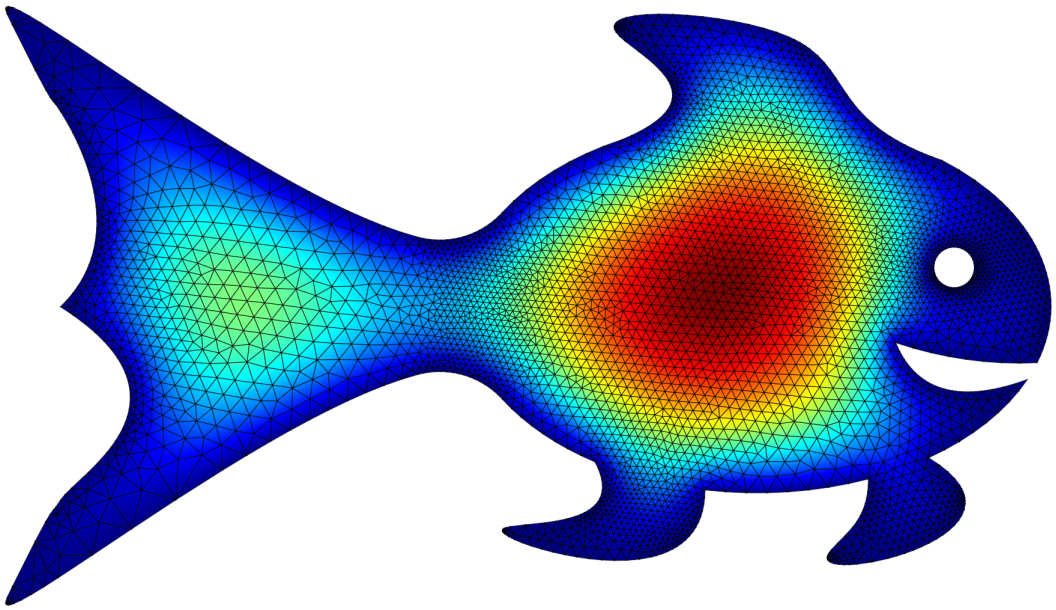
Finite elements for dummies : Poisson

Louis Devillez
Jérome Eertmans

Ce document a pour but daider les étudiants du cours d'Éléments Finis, donné par MM. Vincent Legat et Jean-François Remacle à l'Ecole Polytechnique de Louvain, en leur offrant une solution détaillée aux devoirs.



1 Le problème à résoudre

Comme annoncé par votre professeur, cette semaine marque votre premier "vrai" programme d'éléments finis ! Lors des séances 3 et 4 de TPs, vous avez appris à construire la matrice d'éléments finis afin de résoudre l'équation de Poisson sur un maillage. Cependant, faire cela à la main devient très vite long et calculatoire, c'est là que la programmation prend tout son intérêt ;-)

Votre contribution sera répartie sur deux fonctions :

1. La première, très courte, sera utile lors de notre résolution pour lier notations locales et globales

```
void femMeshLocal(const femMesh *theMesh, const int i, int *map, double *x,
double *y);
```

2. La seconde est LA fonction de ce devoir où on va enfin résoudre un problème avec des élements finis, en passant de l'assemblage de la matrice à sa résolution

```
void femPoissonSolve(femPoissonProblem *theProblem);
```

2 Résolution du problème

Ce devoir visant surtout à une bonne compréhension de l'assemblage de la matrice d'éléments finis, et non à diverses facéties du langage `C`, nous allons directement rentrer dans la résolution des 2 fonctions à implémenter.

2.1 `femMeshLocal`

Comme cela a été vu au cours, il est souvent plus judicieux d'établir toutes les fonctions de formes de manière locale, pour ensuite transposer leur application dans le problème global. Ceci permet, en autres, de traiter tous les éléments de la même manière, et est illustré sur la Figure 1.

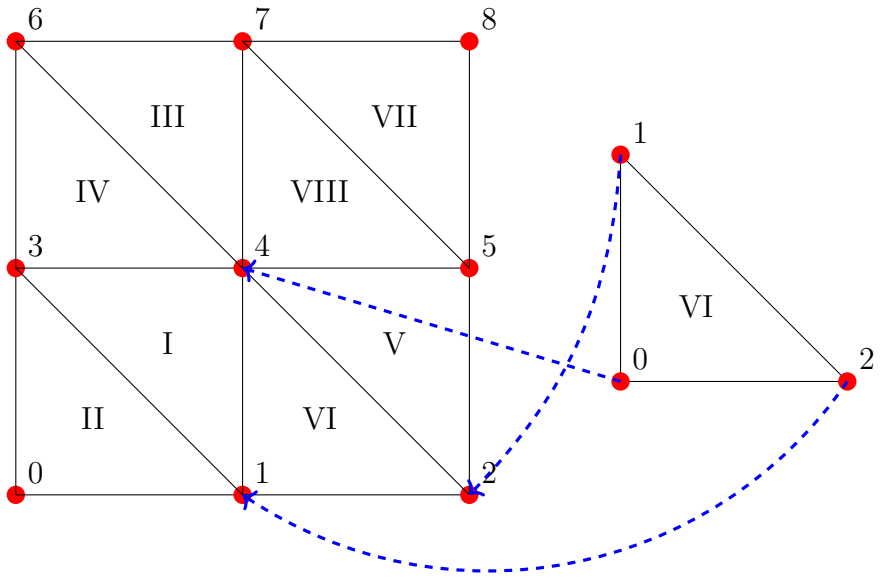


Figure 1 – Illustration de la notation globale (gauche) et locale (droite) des différents noeuds. Les flèches bleues indiquent un *mapping* possible entre les noeuds locaux et globaux.

La première fonction sert donc, pour un élément `i` (VI dans notre exemple), à indiquer, pour chaque noeud local `j`, les coordonnées `X`, `Y`, ainsi que sa numérotation globale `map`. Ici, la subtilité est qu'il faut bien utiliser la valeur de `nLocalNode` vu qu'on peut travailler avec des triangles ou des quadrilatères (cf. APE 4).

```
void femMeshLocal(const femMesh *theMesh, const int i, int *map, double *x,
double *y) {
  for(int j = 0; j < theMesh->nLocalNode; j++){
    map[j] = theMesh->elem[theMesh->nLocalNode*i+j];
    x[j] = theMesh->X[map[j]];
    y[j] = theMesh->Y[map[j]];
  }
}
```

2.2 `femPoissonSolve`

Maintenant que nous avons notre fonction de *mapping*, il va nous falloir assembler notre matrice, pour ensuite la résoudre. Le système linéaire peut se noter comme suit :

$$\boldsymbol{A} \times \boldsymbol{u} = \boldsymbol{B} \tag{1}$$

La matrice $\boldsymbol{A}$ est de dimension (`nElem * nLocalNode`) $\times$ (`nElem * nLocalNode`), et les vecteurs $\boldsymbol{u}$ et $\boldsymbol{B}$ sont de dimension (`nElem * nLocalNode`) $\times 1$.

Comme cela a été vu au cours, les éléments de $\boldsymbol{A}$ et $\boldsymbol{B}$ peuvent simplement être décrits comme valant :

$$A_{ij} = < (\nabla \tau_i) \cdot (\nabla \tau_j) > \tag{2}$$
$$B_i = < \tau_i > \tag{3}$$

Cependant, comme cela a été dit plus haut, les matrices et vecteurs du systèmes vont êtres construits comme la somme de contributions d'éléments locaux. Pour un noeud, selon signifie que l'on va sommer la contribution de toutes les éléments contenant ce noeud. En reprenant notre example, voir FIGURE 2, le noeud 7 va donc recevoir les contributions des éléments III, VII et VIII.

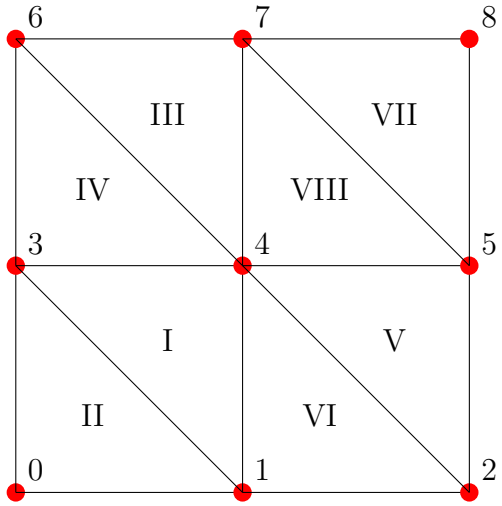


FIGURE 2 – Example basique de maillage avec des éléments triangulaires.

Les matrices et vecteurs locaux, notés $\boldsymbol{A}^e$ et $\boldsymbol{B}^e$, seront respectivement de dimensions (`nLocalNode`) $\times$ (`nLocalNode`) et (`nLocalNode`) $\times 1$. Au lieu d'utiliser les fonctions de formes globales, nous allons utiliser les fonctions de formes locales, fournies avec la règle d'intégration `theProblem->rule`. En faisant cela, il faudra bien faire attention à multiplier par le jacobien de la transformation !

Attention : Comme le sens de parcours change le signe de l'intégrale, via le signe du jacobien, il est important de s'assurer de prendre la valeur absolue de ce dernier.

Pour finir, la résolution de cette fonction peut se découper de cette manière :

1. On déclares nos variables
2. On va parcourir chacun de nos éléments
3. On récupère les information de cet élément avec la première fonction
4. On va parcourir les points d'intégrations
5. On calcule nos fonctions de formes et leurs dérivées locale
6. On applique notre changement de variable
7. On calcule notre jacobienne
8. On ajoute à A_{ij}, B_i la contribution de l'intégrale
9. Ensuite, on peut imposer les conditions de bords
10. Finalement, on peut résoudre finalement le système

Ici, la résolution du système d'équations linéaires se fera simplement via l'appel de la fonction `femFullSystemEliminate`, qui effectue une simple élimination gaussienne. Cependant, vu le caractère creux de votre matrice, il est intéressant d'observer que d'autres méthodes bien plus rapides existent ;-)

```
void femPoissonSolve(femPoissonProblem *theProblem) {
  // Declaration des variables
  femMesh *theMesh = theProblem->mesh;
  femEdges *theEdges = theProblem->edges;
  femFullSystem *theSystem = theProblem->system;
  femIntegration *theRule = theProblem->rule;
  femDiscrete *theSpace = theProblem->space;

  if (theSpace->n > 4) Error("Unexpected discrete space size !");
  double x[4],y[4],phi[4],dphidxsi[4],dphideta[4],dphidx[4],dphidy[4];
  int iElem,iInteg,iEdge,i,j,map[4];

  int nLocal = theMesh->nLocalNode;

  // On parcourt nos elements
  for (iElem = 0; iElem < theMesh->nElem; iElem++) {
    // On recupere les informations globales
    femMeshLocal(theMesh,iElem,map,x,y);
    // On parcourt nos points d'integration
    for (iInteg=0; iInteg < theRule->n; iInteg++) {
      // On calcule les fonctions de formes
      double xsi    = theRule->xsi[iInteg];
      double eta    = theRule->eta[iInteg];
      double weight = theRule->weight[iInteg];
      femDiscretePhi2(theSpace,xsi,eta,phi);
      femDiscreteDphi2(theSpace,xsi,eta,dphidxsi,dphideta);
      double dxdxsi = 0;
      double dxdeta = 0;
      double dydxsi = 0;
      double dydeta = 0;
      double xloc = 0;
      double yloc = 0;
      for (i = 0; i < theSpace->n; i++) {
        xloc   += x[i]*phi[i];
        yloc   += y[i]*phi[i];
```

```
        dxdxsi += x[i]*dphidxsi[i];
        dxdeta += x[i]*dphideta[i];
        dydxsi += y[i]*dphidxsi[i];
        dydeta += y[i]*dphideta[i];
      }

      // Calcule de la jacobienne
      double jac = fabs(dxdxsi * dydeta - dxdeta * dydxsi);

      // Calcul de Ae_{ij}
      for (i = 0; i < theSpace->n; i++) {
        dphidx[i] = (dphidxsi[i] * dydeta - dphideta[i] * dydxsi);
        dphidy[i] = (dphideta[i] * dxdxsi - dphidxsi[i] * dxdeta);
      }
      double wOverJac = weight/jac; // pre-calcul
      double wJack = jac * weight;  // pre-calcul

      // Calcul des contributions de l'integrale
      for (i = 0; i < theSpace->n; i++) {
        for(j = 0; j < theSpace->n; j++) {
          theSystem->A[map[i]][map[j]] += (dphidx[i] * dphidx[j]+ dphidy[i] *
    dphidy[j]) * wOverJac;
        }
        theSystem->B[map[i]] += phi[i] * wJack;
      }
    }
  }
  // Application des conditions sur le bord
  for (iEdge= 0; iEdge < theEdges->nEdge; iEdge++) {
    if (theEdges->edges[iEdge].elem[1] < 0) {
      for (i = 0; i < 2; i++) {
        int iNode = theEdges->edges[iEdge].node[i];
        femFullSystemConstrain(theSystem,iNode,0.0);
      }
    }
  }
  // Resolution du systeme
  femFullSystemEliminate(theSystem);
}
```