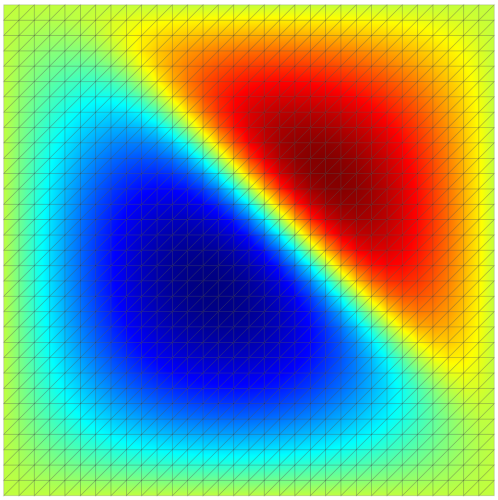
Finite elements for dummies : Convergence rate

Louis Devillez
Jérome Eertmans

Ce document a pour but d'aider 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

Cette semaine, nous allons nous éloigner de la résolution de systèmes afin d'implémenter quelques métriques utiles à l'évaluation de nos méthodes, en termes d'erreur et de convergence.

Pour ce faire, nous allons devoir implémenter 4 fonctions :

1. Une fonction qui va nous permettre de créer un maillage carré avec différentes granularités [1]

```
femMesh *femMeshCreateBasicSquare(int n);
```

2. Une fonction qui va calculer le terme source f de notre équation

```
double convergenceSource(double x, double y);
```

3. Une fonction qui va calculer les différentes erreurs des fonctions u^h et $\tilde{u}^h$

```
void femDiffusionComputeError(femDiffusionProblem *theProblem,
                              double(*soluce)(double,double,double*));
```

4. Une fonction pour estimer le taux de convergences des erreurs

```
double convergenceEstimateRate(double *errors, int n, double ratio);
```

1. Voir https://fr.wikipedia.org/wiki/Granularit%C3%A9

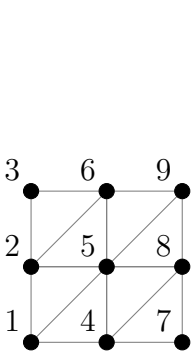
2 Résolution du problème

Pour ce devoir, la matière aquise lors des précédents devoirs devrait vous permettre de résoudre celui-ci sans trop de souci. Ici, hormis les deux dernières fonctions, vous pouvez tester chacune de vos fonctions de manière indépendante, et on vous conseille d'ailleurs de le faire afin de pouvoir valider que chaque bloc est bien fonctionnel.

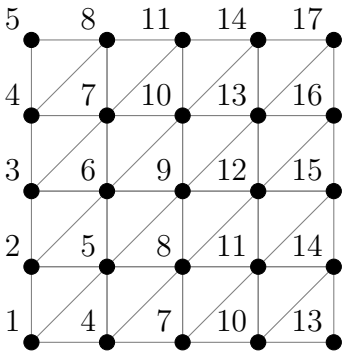
Concernant les éventuelles optimisations possibles, on vous renvoie vers nos autres solutionnaires dans lesquels nous avons soulevé plusieurs façons d'obtenir un code plus rapide. Parmi ces dernières, pour le projet, on vous conseille de prêter une attention toute particulière à l'effet bénéfique que peut avoir la mémoire cache sur la rapidité de votre code ;-)

2.1 `femMeshCreateBasicSquare`

Cette fonction va nous permettre de créer des maillages carrés dont la granularité va être paramétrisée par un entier `n`. Ce maillage carré possède $2n^2$ éléments et $(n+1)^2$ noeuds. On souhaite donc obtenir de ce genre de maillage :



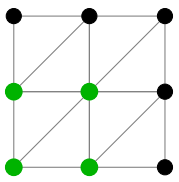
$n = 2$



$n = 4$

La numérotation des éléments n'a pas d'importance ici (vu qu'on utilise pas de solveur frontal). Il existe une pléthore de manières de créer notre maillage et nous en avons relevé 3 principales. *(i)* On pourrait avoir 2 fois 2 boucles `for` imbriquées pour créer d'abord les noeuds et, en suite, les éléments. *(ii)* On peut faire ces 2 opérations dans la même double boucle `for` en ajoutant une condition pour la création des élements : on ne va créer les 2 éléments supérieurs à droite du noeud actuel que seulement pour les noeuds verts. *(iii)* Finalement, on pourrait faire une seule grande boucle sur tous les éléments et, ensuite, déterminer les noeuds de chaque éléments.

Par facilité, nous avons choisi la méthode *(ii)*, mais rien ne nous dit que c'est la plus rapide ! Il faudrait comparer le code dé-assemblé afin d'en être sûr.



$n = 2$

Quoi qu'il arrive, nous vous conseillons fortement de faire un petit dessin à côté pour faciliter votre implémentation.

```c
femMesh *femMeshCreateBasicSquare(int n)
{
  // allocation des variables
  femMesh *theMesh = malloc(sizeof(femMesh));
  theMesh->nNode = (n+1)*(n+1);
  theMesh->X = malloc(sizeof(double)*theMesh->nNode);
  theMesh->Y = malloc(sizeof(double)*theMesh->nNode);
  double *X = theMesh->X;
  double *Y = theMesh->Y;

  theMesh->nElem = 2*n*n;
  theMesh->nLocalNode = 3;
  theMesh->elem = malloc(sizeof(int)*3*theMesh->nElem);
  int *elem = theMesh->elem;
  int i, j, k, iElem, iNode;
  double x, y;

  // Le pas h entre deux noeuds
  double h = 1.0 / n;
  // Indice du noeud en diagonale
  int diag = n + 2;

  // Nombre de noeud sur une colonne
  int above = n + 1;

  for (i = 0, x = 0.0; i <= n; i++, x+=h) {
    for (j = 0, y = 0.0; j <= n; j++, y+=h) {
      // Node coordinates
      iNode = i * above + j;
      X[iNode] = x;
      Y[iNode] = y;

      // Ne regarder que les noeuds verts
      if ((i < n) && (j < n)) {
        // Node assignation
        // Triangle Sud-Est
        iElem = (i * n + j) << 1; // Division par 2 car 2 triangles par carre
        elem[3*iElem  ] = iNode;
        elem[3*iElem+1] = iNode + diag;
        elem[3*iElem+2] = iNode + 1;

        // Triangle Nord-Ouest
        iElem++;
        elem[3*iElem  ] = iNode + above;
        elem[3*iElem+1] = iNode + diag;
        elem[3*iElem+2] = iNode;
      }
    }
  }

  theMesh->number = malloc(sizeof(int)*theMesh->nNode);
  for (int i = 0; i < theMesh->nNode; i++)
    theMesh->number[i] = i;
  return theMesh;
}
```

2.2 `convergenceSource`

Ici, on doit calculer le laplacien de la fonction u. Obtenir cette fonction est assez calculatoire alors on va profiter de la librairie de calcul symbolique de `Python` : `SymPy`. Pour rappel, le laplacien d'une fonction scalaire u est défini comme :

$$\nabla^2 u(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{1}$$

On voit donc vite que, avec un u compliqué, calculer ses dérivées secondes va être rapidement sujet à de bêtes erreurs de caculs ou de recopiage :-)

$$u(x,y) = xy\,(1-x)\,(1-y)\arctan\left(20\frac{(x+y)}{\sqrt{(2)}} - 16\right) \tag{2}$$

Heureusement, `SymPy` fournit une fonction, `ccode`, qui donne directement le code `C` nécessaire. En plus, il est conseillé de d'abord utiliser la fonction `simplifiy` qui tente de simplifier l'expression symbolique en mettant en évidence les termes redondants ce qui permet de réduire la taille de code ainsi que les recalculs inutiles. Il est toutefois possible d'améliorer encore plus le code, mais il faut pour cela vous-mêmes séparer le cacul sur plusieurs lignes afin de réduire le nombre de calculs au minimum.

```
from sympy import *  # Importe atan, sqrt, diff, etc.
from sympy.abc import x, y  # x et y sont des "Symbol"

u = x*y*(1-x)*(1-y)*atan(20*(x+y)/sqrt(2) - 16)

ddudx = diff(u, x, 2)  # Derivee seconde % a x
ddudy = diff(u, y, 2)  # Derivee seconde % a y

print(ccode(simplify(ddudx+ddudy)))
```

Afin de valider vos expressions, `SymPy` possède un mode `pprint` (*pretty print*) qui permet de mieux visualiser vos expressions symboliques. Encore mieux : les Jupyter Notebooks intègrent très bien ces expressions en les affichants dans un joli mode maths :-)

Pour finir, il est important de bien noter que f vaut l'opposé du laplacien précédemment calculé, d'où le `-` dans le code ci-dessous.

$$f(x,y) = -\nabla^2 u(x,y) \tag{3}$$

```
double convergenceSource(double x, double y)
{
    double f = 2*(x*(x - 1)*(-400*y*(y - 1)*(5*M_SQRT2*(x + y) - 8) + 10*M_SQRT2
   *(2*y - 1)*(4*pow(5*M_SQRT2*(x + y) - 8, 2) + 1) + pow(4*pow(5*M_SQRT2*(x + y) -
    8, 2) + 1, 2)*atan(10*M_SQRT2*(x + y) - 16)) + y*(y - 1)*(-400*x*(x - 1)*(5*
   M_SQRT2*(x + y) - 8) + 10*M_SQRT2*(2*x - 1)*(4*pow(5*M_SQRT2*(x + y) - 8, 2) +
   1) + pow(4*pow(5*M_SQRT2*(x + y) - 8, 2) + 1, 2)*atan(10*M_SQRT2*(x + y) - 16)))
   /pow(4*pow(5*M_SQRT2*(x + y) - 8, 2) + 1, 2);
    return -f;
}
```

Une fois cette fonction implémentée, vous devriez avoir un programme fonctionnel où seuls l'erreur affichée et le taux de convergence sont faux.

2.3 femDiffusionComputeError

La fonction la plus compliquée de ce devoir, en grosse partie à cause de sa longueur, consiste en le calcul de plusieurs métriques d'erreurs. La première étape est de comprendre la différence entre u^h et $\tilde{u}^h$. *(i)* u^h représente l'approximation par les éléments finis. On trouve les valeurs nodales via un système d'équations et on utilise les fonctions de formes pour avoir les valeurs entre les noeuds. *(ii)* $\tilde{u}^h$ va représenter l'interpolation avec les fonctions de forme définies sur notre maillage mais avec les valeurs nodales données par la fonction analytique $u(x, y)$.

Concrètement, il va falloir calculer les normes L^2 et H^1 de l'erreur, à savoir $\|u - u^h\|_0$ et $\|u - u^h\|_1$ (resp. $\|u - \tilde{u}^h\|_0$ et $\|u - \tilde{u}^h\|_1$), pour la solution u^h (resp. $\tilde{u}^h$). Il va donc falloir calculer 4 intégrales différentes sur le maillage, en utilisant la règle d'intégration spécifée dans la structure `theProblem`.

Le code n'est pas très compliqué mais est plutôt long : le plus simple est de s'inspirer de ce qui a déjà fait dans les autres devoirs et de recopier :-) Ici, on va simplement commenter la solution mais n'hésitez pas à aller relire les autres solutionnaires pour plus de détails !

Attention : Pour rappel, il faut bien différencier les noeuds (sommets) sur lesquels on connait u et u^h, des points d'intégration pour lesquels il va falloir interpoler les valeurs.

```
void femDiffusionComputeError(femDiffusionProblem *theProblem,
                                    double(*soluce)(double,double,double*))
{
  femMesh *theMesh = theProblem->mesh;
  femIntegration *theRule = theProblem->rule;
  femDiscrete *theSpace = theProblem->space;

  // Que des triangles ou quadrilateres
  if (theSpace->n > 4) Error("Unexpected discrete space size !");
  double Xloc[4],Yloc[4],phi[4],dphidxsi[4],dphideta[4];
  double Uloc[4],Utilt[4],uRef[3];
  int iEdge,iElem,iInteg,i,j,map[4],ctr[4];

  // On initialise nos 4 normes d'erreur
  theProblem->errorSoluceL2 = 0.0;
  theProblem->errorSoluceH1 = 0.0;
  theProblem->errorInterpolationL2 = 0.0;
  theProblem->errorInterpolationH1 = 0.0;

  // Sur chaque element, on integre
  for (iElem = 0; iElem < theMesh->nElem; iElem++) {
    femDiffusionMeshLocal(theProblem,iElem,map,ctr,Xloc,Yloc,Uloc);
    for (i = 0; i < theSpace->n; i++)
      Utilt[i] = soluce(Xloc[i],Yloc[i],uRef);
    // On integre sur chaque point de la methode
    for (iInteg=0; iInteg < theRule->n; iInteg++) {
      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 x = 0;
      double y = 0;
      double u = 0;
```

```
      double utilt = 0;
      double dudx = 0;
      double dudy = 0;
      double dutiltdx = 0;
      double dutiltdy = 0;
      double dxdxsi = 0;
      double dxdeta = 0;
      double dydxsi = 0;
      double dydeta = 0;
      // On interpole les valeurs de u sur le point
      // actuel de la methode
      for (i = 0; i < theSpace->n; i++) {
        x      += Xloc[i]*phi[i];
        y      += Yloc[i]*phi[i];
        u      += Uloc[i]*phi[i];  // u^h
        utilt  += Utilt[i]*phi[i]; // \tilde{u}^h
        dxdxsi += Xloc[i]*dphidxsi[i];
        dxdeta += Xloc[i]*dphideta[i];
        dydxsi += Yloc[i]*dphidxsi[i];
        dydeta += Yloc[i]*dphideta[i];
      }
      double jac = fabs(dxdxsi * dydeta - dxdeta * dydxsi);
      // On calcule les derivees en ce point
      for (i = 0; i < theSpace->n; i++) {
        double dphidx = (dphidxsi[i] * dydeta - dphideta[i] * dydxsi) / jac;
        double dphidy = (dphideta[i] * dxdxsi - dphidxsi[i] * dxdeta) / jac;
        dudx += Uloc[i]*dphidx;
        dudy += Uloc[i]*dphidy;
        dutiltdx += Utilt[i]*dphidx;
        dutiltdy += Utilt[i]*dphidy;
      }
      // uRef = {u, dudx, dudy}
      soluce(x,y,uRef);
      // On calculer l'erreur sur chaque composante
      // 1. Sur u^h
      double e     = (u - uRef[0]);
      double dedx  = (dudx - uRef[1]);
      double dedy  = (dudy - uRef[2]);
      theProblem->errorSoluceL2 += jac * e*e * weight;
      theProblem->errorSoluceH1 += jac * (e*e + dedx*dedx + dedy*dedy) * weight;

      // 2. Sur \tilde{u}^h
      e    = (utilt - uRef[0]);
      dedx = (dutiltdx - uRef[1]);
      dedy = (dutiltdy - uRef[2]);
      theProblem->errorInterpolationL2 += jac * e*e * weight;
      theProblem->errorInterpolationH1 += jac * (e*e + dedx*dedx + dedy*dedy) *
   weight;
    }
  }
  // Finalement, on prend la racine carree pour avoir les bonnes unites
  theProblem->errorSoluceL2 = sqrt(theProblem->errorSoluceL2);
  theProblem->errorSoluceH1 = sqrt(theProblem->errorSoluceH1);
  theProblem->errorInterpolationL2 = sqrt(theProblem->errorInterpolationL2);
  theProblem->errorInterpolationH1 = sqrt(theProblem->errorInterpolationH1);
}
```

2.4 `convergenceEstimateRate`

Cette fonction très simple qui doit vous rappeler le cours de méthodes numériques : l'idée [2] est modéliser l'erreur ϵ_h comme étant une fonction du pas h, d'un coefficient γ et d'un certain n représentant l'ordre de l'erreur.

$$\epsilon_h \sim \gamma h^n \tag{4}$$

Afin de déterminer n, sans connaître γ, on peut utiliser deux mesures d'erreurs, pour deux valeurs de pas différentes, et calculer leur ratio :

$$\frac{\epsilon_{h_i}}{\epsilon_{h_{i+1}}} = \underbrace{\left(\frac{h_i}{h_{i+1}}\right)^n}_{\alpha} \tag{5}$$

$$\log\left(\frac{\epsilon_{h_i}}{\epsilon_{h_{i+1}}}\right) = n \log(\alpha) \tag{6}$$

$$\Rightarrow n = \frac{\log\left(\frac{\epsilon_{h_i}}{\epsilon_{h_{i+1}}}\right)}{\log(\alpha)} \tag{7}$$

Finalement, il peut être intéressant de faire une moyenne de plusieurs estimations de n afin d'obtenir un résultat plus fidèle.

```c
double convergenceEstimateRate(double *errors, int n, double ratio)
{
  double rate = 0;
  for(int i = 0; i < n - 1; i++) {
    rate += log(errors[i]/errors[i+1]);
  }
  return rate / ((n - 1) * log(ratio)); // Moyenne sur n-1 mesures
}
```

3 Comment peut-on vous aider ?

À tous ceux qui lisent nos solutionnaires, tout d'abord : merci ! Ensuite, on aimerait vraiment savoir comment vous aider pour bien maîtriser ce cours, réussir le projet, aller plus loin en `C` ou autre :-)

Pour ce faire, n'hésitez pas à nous envoyer vos retours sur ce qu'on pourrait améliorer et aussi sur ce que vous souhaiteriez voir apparaître dans nos documents / vidéos / TPs / etc. On arrive à la fin des TPs et des devoirs, mais on reste présents pour vous aider à comprendre le cours ainsi qu'à cartonner dans votre projet ! Vous pouvez simplement nous envoyer un mail ou à nous contacter via Teams ;-)

2. Pour rappel, cette idée ne vient pas de nulle part : dans le développement en série de Taylor, le terme d'erreur principal a bien cette forme ;-) On néglige donc souvent les termes d'ordres supérieurs.