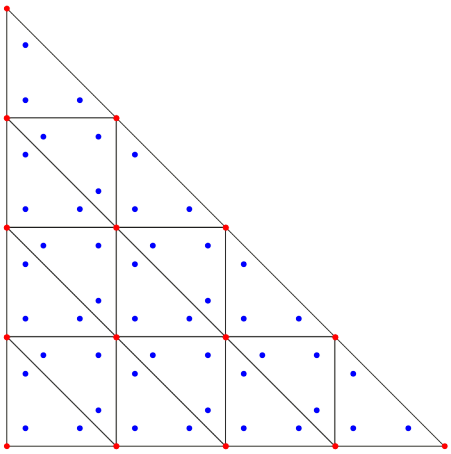
Finite elements for dummies : Integrate

Louis Devillez
Jérome Eertmans
Gilles Poncelet

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

L'objectif de ce devoir est double. Dans un premier temps, il est demandé d'implémenter la fonction

```
double integrate(double x[3], double y[3], double(*f) (double,double));
```

qui estime l'intégrale d'une fonction `f` sur un domaine Ω triangulaire. Dans un second temps, il faut implémenter la fonction

```
double integrateRecursive(double x[3], double y[3], double (*f)(double,double), int
    n);
```

qui appliquera de manière récursive la première méthode en subdivisant le triangle courant en d'autres plus petits.

1.1 Les arguments des fonctions

- `x` et `y` : des tableaux (vecteurs) unidimensionnels de longueur 3
- `f` : une fonction qui prend 2 arguments scalaires en entrée pour en retourner un seul
- `n` : un entier donnant le nombre de récursions à appliquer

1.2 La règle d'intégration de Hammer d'ordre 2 et ses poids

Cette méthode donne une approximation de l'intégration d'une fonction sur un triangle **parent** :

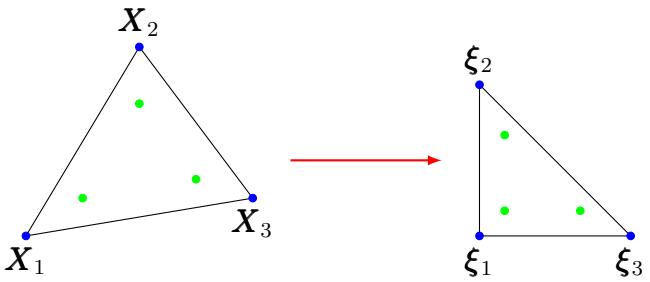
$$\underbrace{\int_{\hat{\Omega}} f(x,y)\,dx\,dy}_{I} \approx \underbrace{\sum_{k=1}^{3} w_k\, f(x_k,y_k)}_{I_h} \tag{1}$$

Les poids w_k sont donnés dans le tableau suivant pour l'élément parent, le triangle $\hat{\Omega}$, dont les sommets sont (0,0), (1,0) et (0,1) dans le plan (ξ,η) :

ξ_k	η_k	w_k
1/6	1/6	1/6
1/6	2/3	1/6
2/3	1/6	1/6

Afin de passer de l'élément parent à un élément **quelconque** dont les sommets sont (X_i, Y_i) avec $i = 1...3$, on utilise le changement de variable suivant :

$$x(\xi,\eta) = \sum_{i=1}^{3} X_i\,\phi_i(\xi,\eta) \qquad \text{et} \qquad y(\xi,\eta) = \sum_{i=1}^{3} Y_i\,\phi_i(\xi,\eta) \tag{2}$$



Il ne faut pas oublier d'adapter les poids w_k de la règle de Hammer, car ceux repris ci-dessus sont valables uniquement pour l'élément parent (c'est-à-dire un triangle formé par les sommets considérés précédemment) ! Pour ce faire, on utilise le jacobien de la transformation, dont la valeur absolue peut être interprétée comme étant un rapport d'aire entre le triangle parent et le triangle quelconque (cf. APE 1). Le jacobien étant le déterminant du gradient de la transformation, il se calcule comme suit :

$$J = det\left(\frac{\partial(x,y)}{\partial(\xi,\eta)}\right) = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix} = (X_2 - X_1)(Y_3 - Y_1) - (X_3 - X_1)(Y_2 - Y_1) \tag{3}$$

Le détail du calcul a été évité étant donné sa longueur mais il suffit d'appliquer les définitions de x et de y pour trouver le même résultat.

1.3 Les fonctions de formes

La manipulation des fonctions de forme est une partie importante de la matière de ce cours, étant donné leur utilité dans le calcul des éléments finis. Rappelons que lors de calculs numériques, un ordinateur aura très facile à calculer des additions et des produits, ceci expliquant pourquoi on approche souvent des fonctions compliquées par le biais de polynômes. Nous vous conseillons vivement de lire la partie du syllabus sur ce sujet (chapitre 1).

Les fonctions de formes peuvent ici être vues comme l'application multidimensionnelle du principe des polynômes de Lagrange. Leur construction est toujours la même : à chaque noeud, on associe une fonction qui vaut 1 en ce noeud et 0 lorsqu'elle est évaluée sur les autres noeuds. Ces fonctions sont de degré p et, dans le cadre de ce devoir, on a choisit $p = 1$.

Pour ceux qui ont un peu de difficultés avec les fonctions de formes, vous êtes invités à lire l'annexe (4).

2 Les importations

En `C`, l'importation de fonctions se fait différemment qu'en `Python` : vous devez *(*i) indiquer au compilateur les différents fichiers que vous souhaitez lier à votre projet, *(ii)* déclarer les fonctions que vous souhaitez partager dans un fichier header (`.h`) et *(iii)* inclure le header de chacun de ces fichiers dans le fichier `.c` qui souhaite utilise les-dites fonctions [1].

Cette procédure vous semble longue et fastidieuse ? Ne vous inquiétez pas trop, vous ne devez a priori pas vous en souciez pour ce devoir (et sûrement pour les prochains), mais cela reste important d'essayer de créer des fichiers supplémentaires et de les inclures à votre projet.

Note sur l'optimisation

Contrairement à `Python`, les boucles en `C` sont très rapides. Vectoriser vos opérations n'a plus trop de sens ici [2], sauf si vous voulez utiliser de la parallélisation, c'est qui bien plus complexe.

Pour optimiser votre code, vous devez donc réflechir en termes d'espace mémoire et de nombre d'opérations. Dans ce document, nous avons surtout mis l'accent sur la simplicité du code mais, à l'avenir, on pourra vous proposer quelques optimisations possibles de votre code. Il y a vraiment plein de trucs chouettes à apprendre là-dessus et une pléthore de façons d'optimiser votre code ! Une méthode simple est de laisser le compilateur optimiser votre code pour même en ajoutant le flag `-O3` à votre compilation pour accélerer votre code sans devoir le changer.

1. Ceci dit, il existe aussi d'autres solutions... :-)
2. Et oui, quand on parlait de vectorisation en `Python`, on parlait en fait d'utiliser un code `C` pour effectuer nos calculs

3 Résolution du problème

3.1 `integrate`

3.1.1 Explications

Comme le définit la règle de Hammer, une approximation de l'intégrale de la fonction f sur le domaine considéré est simplement la somme pondérées de l'évaluation de f aux point considérés. Nous connaissons les coordonnées des point d'intégration sur le triangle parent, mais pas sur le triangle quelconque. Pour pouvoir évaluer la fonction en ces points, on va devoir utiliser le changement de variable permettant de localiser ces points dans le triangle quelconque : c'est ici qu'interviennent les fonctions de forme. Le but est donc de calculer les coordonnées (X_k, Y_k) de ces points dans le triangle quelconque. On obtient donc

$$X_k(\xi_k, \eta_k) = (1 - \xi_k - \eta_k)\, X_0 + \xi_k\, X_1 + \eta_k\, X_2 \tag{4}$$

$$Y_k(\xi_k, \eta_k) = (1 - \xi_k - \eta_k)\, Y_0 + \xi_k\, Y_1 + \eta_k\, Y_2 \tag{5}$$

Une fois ces trois points trouvés, on somme le résultat de la fonction f en ces points (X_k, Y_k), en multipliant par la pondération (qui est toujours **ici** de $\frac{1}{6}$). Comme dit plus haut, il faut encore calculer et multiplier par le Jacobien de la transformation, qui peut se calculer en une ligne de code. Il faut tout de même faire attention à ce que ce Jacobien soit positif : on prend donc la valeur absolue de celui-ci.

3.1.2 Le code

Une règle de bonnes pratiques, c'est de commencer par des choses simples : ici, on peut commencer par le changement de variable. Et, vu que ce bout de code risque d'être réutilisé, il est intéressant de l'écrire dans une fonction à part entière. Ce qu'on veut c'est donner un tableau de 3 éléments, un ξ et un η à notre fonction et qu'elle retourne le point interpolé dans le domaine quelconque.

```
double interpolate(double u[3], double xi, double eta){
  return u[0]*xi + u[1]*eta + (1-xi-eta)*u[2];
}
```

On aurait pu aussi écrire une fonction pour interpoler directement un tableau de points.

```
double interpolate_array(double u[3], double xi[3], double eta[3], double *v){
  for(int i=0; i<3 ; i++){
    v[i] = interpolate(u, xi[i], eta[i]);
  }
}
```

Une autre partie simple à écrire est le calcule du jacobien, pour lequel il ne faut pas oublier la valeur absolue.

```
double jac = fabs((x[0]-x[1])*(y[0]-y[2]) - (x[0]-x[2])*(y[0]-y[1]));
```

Maintenant ce qu'il nous reste c'est calculer notre intégrale.

```
double I = 0;
for(int i=0 ; i<3 ; i++){
  I += f(xloc[i],yloc[i]) * weight[i];
}
I *= jac;
```

On peut désormais tout remettre ensemble.

```
double interpolate(double u[3], double xi, double eta)
{
  return u[0]*xi + u[1]*eta + u[2]*(1.0-xi-eta);
}


double integrate(double x[3], double y[3], double (*f) (double, double))
{
  double I = 0;
  const double xi[3]    = {0.166666666666667,0.666666666666667,0.166666666666667};
  const double eta[3]    = {0.166666666666667,0.166666666666667,0.666666666666667};
  const double weight[3] = {0.166666666666667,0.166666666666667,0.166666666666667};
  double jac = fabs((x[0]-x[1]) * (y[0]-y[2]) - (x[0]-x[2]) * (y[0]-y[1]));

  double xLoc[3];
  double yLoc[3];

  for (int i=0 ; i<3 ; i++) {
    double xiLoc = xi[i];
    double etaLoc = eta[i];
    xLoc[i] = interpolate(x, xiLoc, etaLoc);
    yLoc[i] = interpolate(y, xiLoc, etaLoc);
    I += f(xLoc[i], yLoc[i]) * weight[i];
  }

  return I*jac;
}
```

C'est toujours bien de définir ce qui est constant avec le mot clé `const`. On aurait pu tout aussi bien écrire tout ce qui est dans la boucle `for` en une seule ligne :

```
for(int i=0 ; i<3 ; i++)
  I += f(interpolate(x, xi[i], eta[i]), interpolate(y, xi[i], eta[i])) * weight[i];
```

Mais le code devient toute de suite moins lisible sans réel gain en performances. Un petit changement qu'on pourrait réaliser : ici, on a déclaré `xloc[3]` et `yloc[3]` mais, en tant que tel, on a juste besoin de `xloc[i]` et `yloc[i]`, les valeurs que l'on vient de calculer. On aurait donc ce code ci :

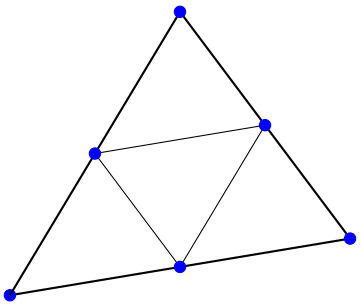
```
double xLoc;
double yLoc;

for (int i=0 ; i<3 ; i++) {
  double xiLoc = xi[i];
  double etaLoc = eta[i];
  xLoc = interpolate(x, xiLoc, etaLoc);
  yLoc = interpolate(y, xiLoc, etaLoc);
  I += f(xLoc, yLoc) * weight[i];
}

return I*jac;
```

3.2 `integrateRecursive`

3.2.1 Explications

Le principe fondamental de cette fonction est la récursion. Il faut tout d'abord établir la condition de récursion : ici, cette condition est $n > 0$. Si la condition n'est pas respectée, alors on va simplement

calculer l'intégrale sur le triangle formé par les vecteurs x et y sur la fonction f. Il suffit donc de retourner l'appel à la fonction `integrate` avec ces paramètres.



Si la condition est respectée, alors il faut subdiviser le triangle courant (toujours formé par les vecteurs x et y) en 4 sous-triangles. La question est désormais de savoir comment subdiviser le triangle quelconque en 4 sous-triangles. La réponse est toute simple : de la même manière que pour trouver les points d'intégration dans la fonction précédente. Pour cela, partons du triangle parent : pour subdiviser celui-ci, on a besoin de 3 points supplémentaires, un sur le milieu de chaque segment du triangle. Pour passer de chaque sous-triangle de l'élément triangulaire parent au sous-triangle quelconque, on réutilise les équations 4 et 5.

Une fois ces coordonnées trouvées, il suffit de rappeler la fonction récursive sur chaque sous-triangle, en décrémentant évidemment la valeur de n pour arriver à la fin de la récursion. Bien évidemment, il faut sommer la valeur de retour (qui est l'intégrale sur chaque sous-triangle) pour obtenir la valeur totale de l'intégrale.

3.2.2 Le code

On peut commencer par le cas le plus simple qui est `n = 0`

```
double integrateRecursive(double x[3], double y[3], double (*f)(double, double),
    int n) {
  // Toujours bien d'avoir un cas general
  if(n <= 0) {
    return integrate(x, y, f);
  }
}
```

Pour la partie `n > 0` on doit tout d'abord générer les nouveaux points. On pourrait écrire à la main tous les calculs mais il y a plus malin :

```
const int nodes[4][3] = {{0,3,5},{3,1,4},{5,4,2},{3,4,5}};
const double   xi[6] = {0.0,1.0,0.0,0.5,0.5,0.0};
const double   eta[6] = {0.0,0.0,1.0,0.0,0.5,0.5};
double xLoc[3];
double yLoc[3];

for (int i=0 ; i<4 ; i++) { // 4 sous-triangles par triangle
  for (int j=0 ; j<3 ; j++) { // 3 points par triangle
    double xiLoc = xi[nodes[i][j]];
    double etaLoc = eta[nodes[i][j]];
    xLoc[j] = interpolate(x, xiLoc, etaLoc);
    yLoc[j] = interpolate(y, xiLoc, etaLoc);
```

```
  }
}
```

On définit un array `nodes` qui va contenir les informations sur chacun des triangles, plus particulièrement contenir l'id de chacun des noeuds qui composent un triangle.

Dans `xi` et `eta` on va créer des nouveaux points d'interpolations : les anciens sommets et les points milieux des arrêtes.

En utilisant les valeurs encodées dans `nodes` et ces nouveaux points d'interpolation on a nos triangles. On finir par intégrer chacun de ces triangles.

```
double integrateRecursive(double x[3], double y[3], double (*f)(double, double),
    int n) {
  int i,j;
  const int nodes[4][3] = {{0,3,5},{3,1,4},{5,4,2},{3,4,5}};
  const double   xi[6] = {0.0,1.0,0.0,0.5,0.5,0.0};
  const double   eta[6] = {0.0,0.0,1.0,0.0,0.5,0.5};
  double xLoc[3];
  double yLoc[3];

  if (n <= 0) return integrate(x, y, f, window);

  double I = 0.0;
  for (i=0; i<4; i++) {
    for (j=0; j<3; j++) {
      double xiLoc = xi[nodes[i][j]];
      double etaLoc = eta[nodes[i][j]];
      xLoc[j] = interpolate(x, xiLoc, etaLoc);
      yLoc[j] = interpolate(y, xiLoc, etaLoc);
      }
    I += integrateRecursive(xLoc, yLoc, f, n-1, window);
    }
  return I;
}
```

4 Annexe

4.1 Exemple de calcul de fonctions de formes

Dans les cas étudiés, nous prendrons le cas d'un espace bi-dimensionnel, donc avec 2 variables, dans lequel on veut appliquer les méthodes aux éléments finis sur un triangle parent de sommets (0,0), (1,0) et (0,1).

4.1.1 Fonction du premier degré

Une fonction du premier degré peut s'écrire en toutes généralité :

$$f(\xi, \eta) = a\xi + b\eta + c \tag{6}$$

On retrouve 3 degrés de libertés : il nous faut donc 3 noeuds.

La méthode utilisée pour calculer les fonctions de formes étant essentiellement la même quelque soit l'ordre, nous ne la détaillerons que pour les fonctions de formes d'ordre 2. Vous devez néanmoins retrouver, en reprenant la notation du cours :

$$\phi_1 = 1 - \xi - \eta \tag{7}$$
$$\phi_2 = \xi \tag{8}$$
$$\phi_3 = \eta \tag{9}$$

4.1.2 Fonction du second degré

Une fonction du deuxième degré peut s'écrire en toutes généralité :

$$f(\xi, \eta) = a\xi + b\eta + c + d\xi^2 + e\eta^2 + f\xi\eta \tag{10}$$

On retrouve 6 degrés de libertés : il nous faut donc 6 noeuds.

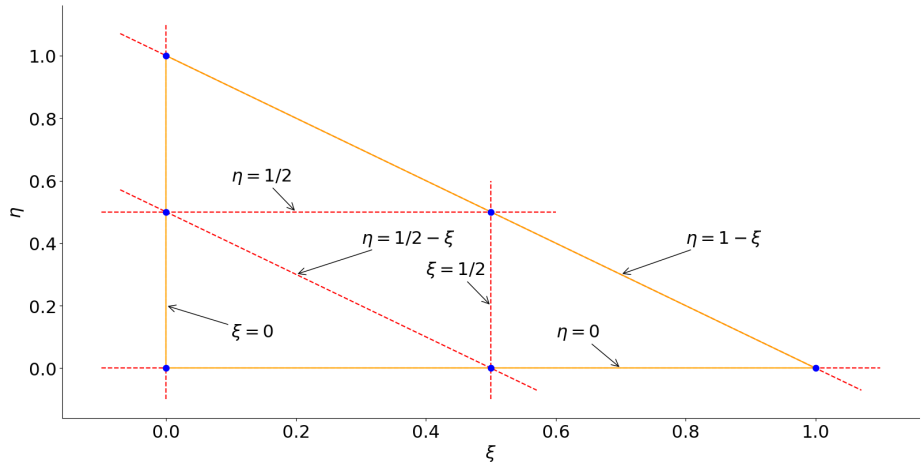


FIGURE 1 – Illustration de la méthode pour les fonctions de degré 2.

On va choisir les trois points formant le triangle, ainsi que le milieu de chacun des segments de celui-ci. Calculer directement une fonction de degré deux telle qu'on a 1 au point considéré et 0 aux autres points n'est pas directement visible (sauf peut-être pour certains d'entre vous :-). Heureusement, on peut se rappeler qu'une fonction telle que la fonction f est équivalente à un produit de fonctions (bi-)linéaires. Nous pouvons dès lors chercher à simplement multiplier plusieurs fonctions simples entre elles pour donner obtenir la fonction de forme voulue.
Pour une fonction de degré p à une variable, on obtient la forme suivante

$$f(x) = (x - x_1)(x - x_2)...(x - x_p)$$

où chaque terme du produit est une racine de f. C'est exactement ce que nous allons faire ici : chercher ces fonctions (bi-)linéaires telles que les racines du polynôme formé sont exactement les coordonnées des points que nous cherchons justement à annuler dans la fonction de forme. Néanmoins, deux contraintes se présentent :

— Dans notre cas, les fonctions de forme sont de degré 2, pas plus
— En notre point considéré, la fonction de forme doit valoir 1

Il faut donc effectuer un choix judicieux de ces fonctions de forme pour, premièrement, obtenir le degré désiré et, deuxièmement, que cette fonction de forme vaille 1 au point considéré. De manière générale, vous pouvez vous dire que si la fonction de forme est de degré p, il faut s'attendre à multiplier (au maximum) p fonctions linéaires entre elles. Pour le point $(\xi, \eta) = (0, 0)$, on voit sur la figure que ces deux fonctions bi-linéaires sont $f_1(\xi, \eta) = 1/2 - \xi - \eta$ et $f_2(\xi, \eta) = 1 - \xi - \eta$. Les multiplier entre elles donne simplement

$$\phi_1^*(\xi, \eta) = (1/2 - \xi - \eta)(1 - \xi - \eta)$$

Cette fonction vaut bien 0 pour les autres points que le point $(0, 0)$, mais elle vaut $1/2$ au point d'intérêt. Pour obtenir $f(0, 0) = 1$, on peut simplement multiplier cette fonction par un facteur 2. On obtient donc finalement

$$\begin{aligned}\phi_1(\xi, \eta) &= 2(1/2 - \xi - \eta)(1 - \xi - \eta) \\ &= 1 - 3(\xi + \eta) + 2(\xi + \eta)^2\end{aligned}$$

La marche à suivre est la même pour toutes les autres fonctions de formes. Elle peuvent être également représentées dans le plan (ξ, η, z) où z contient l'image des fonctions de formes.

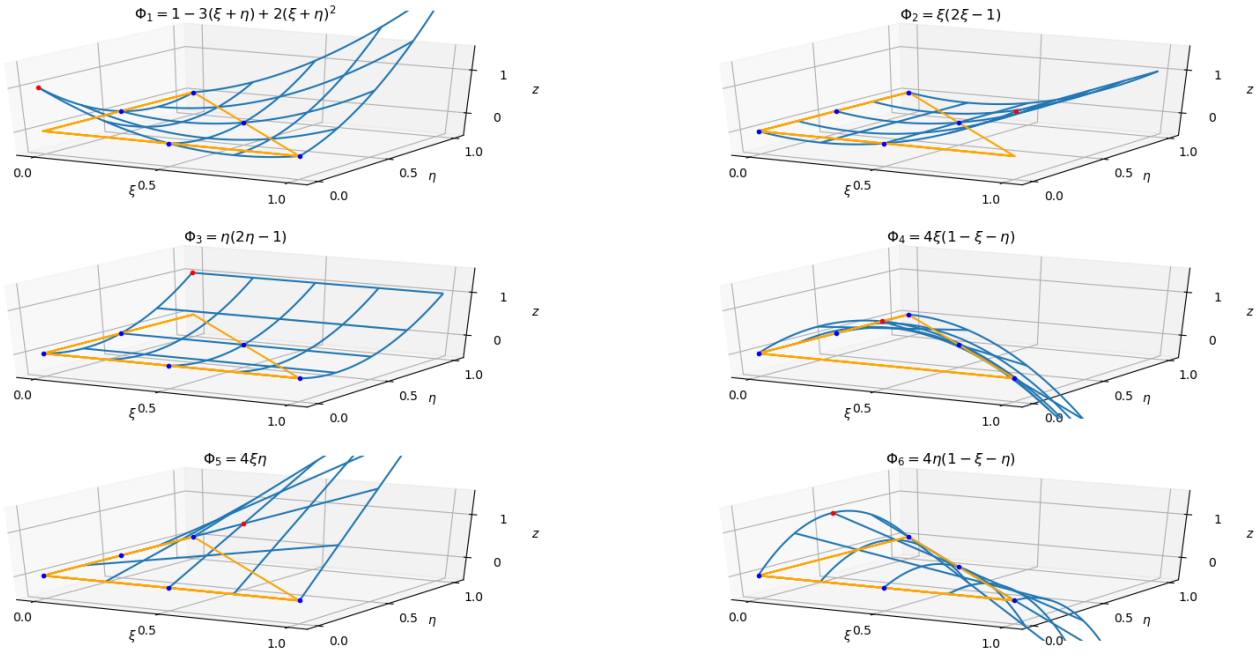


FIGURE 2 – Illustration 3D des fonctions de formes de degré 2.