Introduction à FREEFEM++

Nicolas KIELBASIEWICZ*

19 avril 2007

FREEFEM++est un freeware développé au Laboratoire Jacques-Louis Lions de l'Université Pierre et Marie Curie, porté sous Windows, Unix et Mac OS et dédié à la résolution d'équations aux dérivées partielles par des méthodes de type éléments finis.

Pour le télécharger et obtenir de plus amples informations, consulter :

http ://www.freefem.org/ff++

L'objectif de ce petit document n'est pas de donner une liste exhaustive des diverses fonctionnalités du langage FREEFEM++(basé sur C++) — la documentation disponible sur le lien précédent est faite pour ça —, mais plutôt de donner un aperçu des fonctions les plus couramment utilisées, ainsi que les fonctionnalités que j'ai personnellement eu du mal à trouver dans les docs disponibles sur le net, dans l'esprit d'un aide-mémoire.

Table des matières

*Unité de Mathématiques Appliquées, École Nationale Supérieure de Techniques Avancées

1 Les types de données

1.1 Variables globales ou variables réservées

En FREEFEM++, il existe un certain nombre de variables globales dont voici les plus courantes :
- **x**, **y** et **z** : les coordonnées du point courant. Pour l'instant, **z** n'est pas encore utilisable, mais il est réservé pour un usage futur.
- **label** : le numéro de référence de la frontière dans laquelle se situe le point courant, O sinon.
- **P** : le point courant.
- **N** : le vecteur normal sortant unitaire au point courant s'il se situe sur une frontière.
- **cin**, **cout** et **endl** : les commandes d'affichage/récupération de données issues de C++, utilisées avec << et >>.
- **pi** : le nombre π
- **true** et **false** : les booléens.
- **i** : le nombre imaginaire ($\sqrt{-1}$).

1.2 Les types basiques

1.2.1 Les booléens

Il s'agit du type **bool**.
Les opérateurs logiques sont :

symbole	signification de l'opérateur
==	égal à
<=	inférieur ou égal à
>=	supérieur ou égal à
<	inférieur strictement à
>	supérieur strictement à
!=	différent de

1.2.2 Les entiers

Il s'agit du type **int**.

1.2.3 Les réels

Il s'agit du type **real**.

1.2.4 Les complexes

Il s'agit du type **complex**. A l'affichage, un complexe $x+iy$ est remplacé par le couple (x, y). Quelques fonctions élémentaires associées : **real**, **imag** et **conj**.

1.2.5 Les chaînes de caractères

Il s'agit du type **string**. Les chaines de caractères sont définies avec des doubles guillemets :

```
string toto "this is a string"
```

1.3 Les tableaux et les matrices

Il existe deux types de tableaux, ceux avec des indices entiers, et ceux dont les indices sont des chaines de caractères. Les éléments sont de type **int**, **complex** ou **real**.

On peut aussi préférer des tableaux à deux indices plutôt que le type **matrix**.

```
real[int] a(n);
a[3]=2;
```

2 Les fonctions

2.1 Les fonctions mathématiques prédéfinies

Voici quelques unes des fonctions mathématiques les plus courantes :

- +, -, * et /.
- **cos**, **sin**, **tan**, **acos**, **asin** et **atan**.
- **cosh**, **sinh**, **acosh** et **asinh**.
- **log**, **log10** et **exp**.
- **sqrt** et ^.

2.2 Définir une fonction à une variable

```
func type nom_fct(type var)
{
instruction 1;
...
...
instruction n;
return outvar;
}
```

2.3 Les formules

Il s'agit de fonctions dépendant des deux variables d'espace **x** et **y** et sont définies à partir des fonctions élémentaires vues précédemment. On les définit de la façon suivante :

```
func outvar = expression(x,y);
```

Exemples :

```
func c=x+y*1i;
func f=imag(sqrt(z));
```

2.4 Les fonctions dépendant du maillage

Il s'agit d'un cas particulier des fonctions précédentes, dans la mesure où on va évaluer une formule sur les noeuds du maillage. La procédure est donc la suivante :

```
fespace espace_name(maillage, type_elements_finis);
func fctoutvar = expression(x,y);
espace_name FEfctoutvar = fctoutvar;
```

De plus amples explications sont données en Section 6.
On notera que l'on peut également définir un tableau de fonctions dépendant du maillage.

3 Les boucles et les instructions de contrôle

3.1 La boucle for

```
for (init,cond,incr) {
...
}
```

3.2 La boucle while

```
while (cond) {
...
}
```

3.3 Les instructions de contrôle

```
if (cond) {
...
}
```

```
else {
...
}
```

4 Les entrées / sorties

4.1 Ouvrir un fichier

Pour ouvrir un fichier en lecture :

```
ifstream name(nom_fichier);
```

Pur ouvrir un fichier en écriture :

```
ofstream name(nom_fichier);
```

4.2 Lire dans un fichier

On utilise $>>$.

4.3 Ecrire dans un fichier

On utilise $<<$.

4.4 Fermer un fichier

Il n'existe pas de commandes pour fermer un fichier, comme c'est le cas dans la plupart des langages de programmation. L'astuce consiste à utiliser l'allocation/désallocation dynamique de la mémoire et la notion de variables locales en C++. Concrètement, cela revient à procéder de la façon suivante :

```
{
ouverture du fichier
séquence d'instructions de lecture/écriture
} /* ceci ferme le fichier car en sortant du bloc délimité par les
accolades, la variable du fichier est détruite */
```

5 Définir un maillage

5.1 Maillage triangulaire régulier dans un domaine rectangulaire

On considère le domaine $]x0, x1[x]y0, y1[$. Pour générer un maillage régulier nxm, on utilise la commande suivante :

```
mesh nom_maillage = square(n,m,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
```

A noter qu'il faut éviter d'avoir des triangles trop allongés, comme il est précisé dan la théorie de la triangulation. On choisira donc en conséquence les valeurs n et m.

5.2 Maillage triangulaire non structuré défini à partir de ses frontières

Pour définir les frontières, on utilise la commande **border** :

```
border name(t=deb,fin){x=x(t);y=y(t);label=num_label};
```

On définit ainsi l'ensemble des frontières du domaine. Il faut néanmoins faire attention à l'orientation de la frontière :

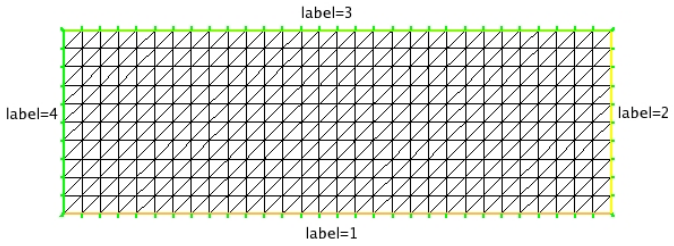

FIG. 1 – Maillage triangulaire avec la numérotation des frontières

1. Les frontières extérieures sont définies dans le sens trigonométrique
2. Si le domaine comporte une frontière intérieure fermée, il y a deux possibilités. Si la frontière intérieure est définie dans le sens trigonométrique, alors tout le domaine sera rempli, et le domaine àl'intérieur de la frontière intérieure aura une référence différente. Si au contraire la frontière intérieure est parcourue dans le sens horaire, alors le domaine aura un «trou ».

Pour définir un maillage à partir de ses frontières, on utilise la commande **buildmesh** :

```
buildmesh nom_maillage=buildmesh(b1(z1)+b2(z2)+...+bk(zk));
```

où les b_i sont les frontières définies avec la commande **border** et z_i un entier relatif dont la valeur absolue représente le nombre de noeuds sur la frontière b_i. Si z_i est négatif, alors l'orientation de la bordure est inversée.

Conseil : Puisqu'on peut utiliser des entiers relatifs, alors autant définir toutes les frontières dans le sens trigonométrique et utiliser des z_i négatifs quand on a besoin de changer le sens d'orientation d'une ou plusieurs frontières. Cela rend, à mon avis, le code davantage lisible.

5.3 Entrées / sorties fichiers

1. `savemesh(nom_maillage,nom_fichier); // permet de sauver le maillage au format .msh`

2. `readmesh(nom_fichier); // permet de lire un maillage à partir d'un fichier .msh`

5.4 Autres fonctions sur les maillages

1. `mesh mail2 = movemesh(mail1,[f1(x,y),f2(x,y)]); /* permet de déformer le maillage mail1 et de le stocker dans mail2 */`

2. `mesh mail2 = adaptmesh(mail1,var) /* permet de raffiner le maillage mail1 dans les zones de fortes variations de var et de stocker le résultat dans mail2 */`

5.5 Lire les données d'un maillage

Maintenant que nous avons vu les commandes usuelles pour générer un maillage, je vais énumérer ici les commandes pour accéder à certaines informations du maillage

Th.nt le nombre de triangles

Th.nv le nombre de noeuds

Th[i][j] le sommet j du triangle i

6 Résoudre une EDP

6.1 Définition de l'espace d'approximation

On utilise la commande **fespace**.

```
fespace nom_espace(nom_maillage,type_elements_finis);
```

Le type d'éléments finis est un mot-clé dans la liste suivante : P0, P1, P1dc (P1 discontinu), P1b (P1 bulle), P2, P2b, P2dc, RT0 (Raviart-Thomas), P1inc (P1 non conforme).

L'espace ainsi défini est à son tour un type de données pour définir les variables de type éléments finis.

6.2 Définir le problème variationnel

De manière générale, on définit un problème variationnel de la façon suivante :

```
problem pb_name(u,v) =
        a(u,v)-l(v)
        + (conditions aux limites);
```

Pour résoudre un problème variationnel, il suffit de taper la commande :

```
pb_name;
```

6.2.1 Dérivées

On utilise les commandes **dx** et **dy**, qui ne s'appliquent qu'à des variables de type éléments finis.
Exemple :

```
mesh Th=square(20,20,[x,y]);
fespace Vh(Th,P1);
Vh uh;
func u=x+y;
```

On peut donc écrire **dx**(uh) mais pas **dx**(u). Si toutefois on a besoin d'effectuer ce genre d'opération, il faut utiliser la fonction de type éléments finis associée :

```
Vh ue=u;
```

6.2.2 Formes bilinéaires

int1d(T_h,n_1,n_2,...,n_k)(A*u*v)	$\sum_{T\in\mathcal{T}_h}\int_{(\partial T\bigcup\Gamma)\bigcap(\bigcup_{i=1}^{k}\Gamma_{n_i})} Auv$
int2d(T_h[,k])(A*u*v)	$\sum_{T\in\mathcal{T}_h[\bigcap\Omega_k]}\int_T Auv$
intalledges(T_h[,k])(A*u*v)	$\sum_{T\in\mathcal{T}_h[\bigcap\Omega_k]}\int_{\partial T} Auv$

6.2.3 Formes linéaires

int1d(T_h,n_1,n_2,...,n_k)(A*v) $\displaystyle\sum_{T\in\mathcal{T}_h}\int_{(\partial T\bigcup\Gamma)\bigcap(\bigcup_{i=1}^{k}\Gamma_{n_i})} Av$

intalledges(T_h[,k])(f*v) $\displaystyle\sum_{T\in\mathcal{T}_h[\bigcap\Omega_k]}\int_{\partial T} Auv$

6.2.4 Conditions aux limites de Dirichlet

On utilise la commande **on** sous la forme :

```
on(num1, numk, u=g);
```

6.2.5 Quelques autres possibilités

On peut définir une formulation variationnelle avec le type **varf**, et ce de la même manière que le type **problem**. Cela présente un grand intérêt dans le cas où l'on veut résoudre plusieurs problèmes présentant une partie commune. Le deuxième intérêt est de pouvoir raisonner en terme de produits matriciels. Pour définir la matrice associée à une forme variationnelle, on procède comme suit :

```
matrix name = varf_name(espace1,espace2);
```

Les diverses commandes présentées ont également des options qui servent dans la résolution de problèmes plus élaborés (condensation de masse, algorithme de résolution, ...).

6.2.6 Conventions d'écriture

Au-delà des types des arguments des fonctions FREEFEM++, il y a deux conventions d'écriture à respecter afin d'éviter un message d'erreur pas forcément compréhensible :

- On n'effectue pas d'opérations à l'intérieur d'une dérivée. Par exemple, au lieu d'écrire **dx**(2.0*u-v), on écrira 2.0***dx**(u) - **dx**(v).
- De même, les opérations s'effectuent à l'intérieur des commandes **int1d** et **int2d**. Seul le signe peut être mis avant. Par exemple, au lieu d'écrire 2.0***int2d**(T_h)(u), on écrira **int2d**(T_h)(2.0*u).

7 Visualiser les résultats

7.1 Directement avec Freefem++

On utilise la commande **plot**, qui sert non seulement à afficher des maillages, mais aussi les courbes d'isovaleurs et les champs de vecteurs. Comme pour les commandes **dx** et **dy**, la commande **plot** n'accepte pas les variables de type **func**.

```
plot(var1,[var2,var3],...[liste d'options]);
```

Les options les plus courantes sont :

- **wait**=true/false : détermine si la fenêtre graphique se ferme immédiatement ou non. Si on a choisi true, alors le programme attend une action au clavier du type :
  - **+/-** pour zoomer/dézoomer
  - **r** pour rafraîchir la fenêtre
  - **p** pour sauvegarder au format postscript

  La valeur par défault est false.
- **value**=true/false : affiche ou non la légende de couleur des isolignes de la courbe. La valeur par défaut est false.

– **fill**=num : si num=1 alors l'espace entre les isolignes est rempli de couleur.
– **ps**=nom_fichier : permet de sauvegarder la courbe au format postscript.

Dans la dernière version en date de la documentation officielle (2.11), le code permettant d'exporter vers gnuplot a été ajouté. Je vais donc parler d'un autre type d'exportation, celui qui concerne medit, où il y a un bug.

7.2 Exporter vers Medit

Dans la documentation officielle de FREEFEM++, l'exemple donné dans le paragraphe consacré à l'exportation vers medit est malheureusement obsolète. Voici donc comment se débrouiller avec la dernière version de medit :

Le fichier exemple.bb Voilà le code permettant de générer ce fichier :

```
{
ofstream file("exemple.bb");
file <<"2 1 "<<uh[].n<<" 2"<<endl;
for (int j=0;j<uh[].n;j++) {
        file <<uh[][j]<<endl;
}
}
```

Ce code mérite une petite explication, notamment la première ligne écrite dans le fichier. Le premier entier correspond à la dimension de l'espace (ici, c'est 2). Prendre 1 pour le deuxième entier signifie que les variables sont scalaires. Vient ensuite le nombre de noeuds. Le dernier entier de la ligne désigne le type (1 signifie qu'une valeur est associée à une face, et 2 qu'une valeur est associée à un noeud). Dans notre cas, il s'agit de valeurs nodales. La séquence des valeurs est écrite par la suite.

Le fichier exemple.mesh C'est là la grande différence par rapport à la version proposée dans la documentation officielle, car les fichiers .faces et .points sont maintenant regroupés dans un seul fichier dont l'extension est .mesh. Il suffit donc d'utiliser **savemesh** en donnant explicitement l'extension du fichier de sortie :

```
savemesh(Th,"exemple.mesh");
```

On génère ainsi 2 fichiers :

exemple.mesh Le fichier de maillage proprement dit au format souhaité

exemple.mesh.gmsh Le fichier concernant la géométrie du maillage utilisé directement par exemple.mesh

8 A travers des exemples

On considère les problèmes suivants :

$$\begin{cases} -\Delta u = f \quad \text{dans } \Omega \\ u = g \quad \text{sur } \partial\Omega \end{cases} \qquad \begin{cases} -\Delta u = f \quad \text{dans } \Omega \\ u = g \quad \text{sur } \Gamma_1 \\ \frac{\partial u}{\partial n} = h \quad \text{sur } \partial\Omega\backslash\Gamma_1 \end{cases}$$

Pour le premier problème, on choisit un domaine rectangulaire et la fonction f de sorte que l'on connaisse la solution exacte du problème afin de calculer l'erreur L^2. On choisira donc $f = 5\pi^2 \sin 2\pi x \sin \pi y$ de sorte que la solution est $u = \sin 2\pi x \sin \pi y$.

Pour le second problème, on choisit un domaine rectangulaire avec un trou. Le bord Γ_1 jouera le rôle de la frontière intérieure. On garde le même f et on choisit $g = h = 0$.

```
// définition des maillages
border Gammab(t=0,1){x=2*t;y=0;label=1;};
border Gammad(t=0,1){x=2;y=t;label=2;};
border Gammah(t=0,1){x=2*(1-t);y=0;label=3;};
border Gammag(t=0,1){x=0;y=1-t;label=4;};
border Gammai(t=0,2*pi){x=1+0.5*cos(t);y=0.5+0.3*sin(t);label=5;};

mesh Th=buildmesh(Gammab(40)+Gammad(20)+Gammah(40)+Gammag(20));
mesh Mh=buildmesh(Gammab(40)+Gammad(20)+Gammah(40)+Gammag(20)+
Gammai(-30));

plot(Th,ps="maillage.ps");
plot(Mh,ps="maillagetrou.ps");

// Définition des espaces d'approximation
fespace Vh(Th,P1);
fespace Wh(Mh,P1);

// Définition des données du problème
func f=5*sin(2*pi*x)*sin(pi*y);
func g=0;
func h=0;
func u=sin(2*pi*x)*sin(pi*y);
Vh uh,vh,ue;
Wh uuh,vvh;

// Définition du problème variationnel sur Th
problem P(uh,vh) = int2d(Th)(dx(uh)*dx(vh)+dy(uh)*dy(vh))
                 - int2d(Th)(f*vh)
                 + on(1,2,3,4,uh=0);

// Définition du problème variationnel sur Mh
problem Q(uuh,vvh) = int2d(Mh)(dx(uuh)*dx(vvh)+dy(uuh)*dy(vvh))
                   - int2d(Mh)(f*vvh)
                   - int1d(Mh,1,2,3,4)(h*vvh)
                   + on(5,uuh=g);

// Résolution des problèmes variationnels
P;
Q;

// Calcul d'erreur L2 pour le problème P
ue=u;
real erreur=int2d(Th)((ue-uh)*(ue-uh));
string legende="erreur = "+erreur;

// Affichage des solutions
plot(uh,cmm=legende,wait=true,ps="lapdirichlet.ps");
plot(uuh,wait=true,fill=1,value=true,ps="lapneumanntrou.ps");
```

Voici les résultats obtenus. Pour le maillage rectangulaire, on a affiché les isolignes ainsi que la valeur de l'erreur L^2. On voit donc au passage la précision de la résolution par éléments finis.

Dans le deuxième exemple, on a ajouté les paramètres **value** et **fill** pour voir la différence au niveau graphique.

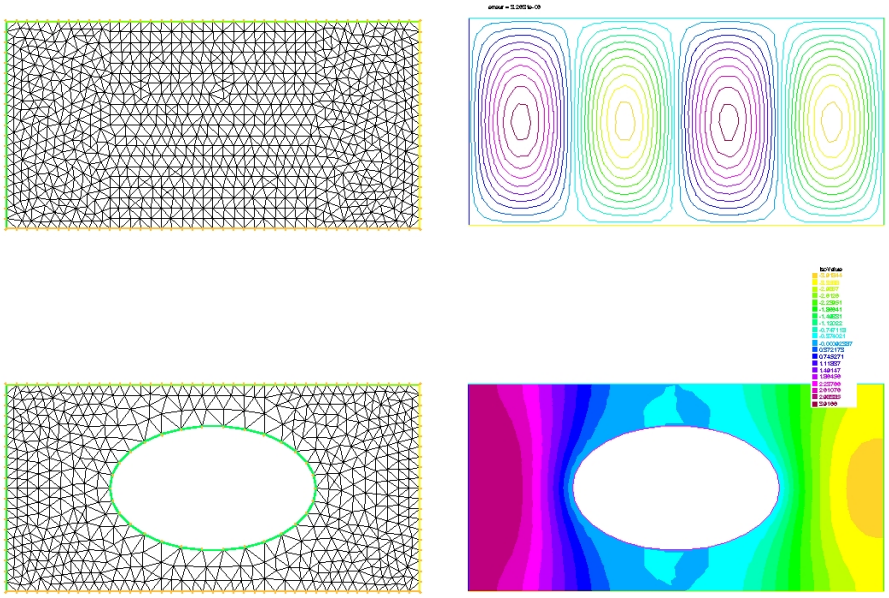

FIG. 2 – Pour les deux exemples, le maillage à gauche et la solution du problème à droite.