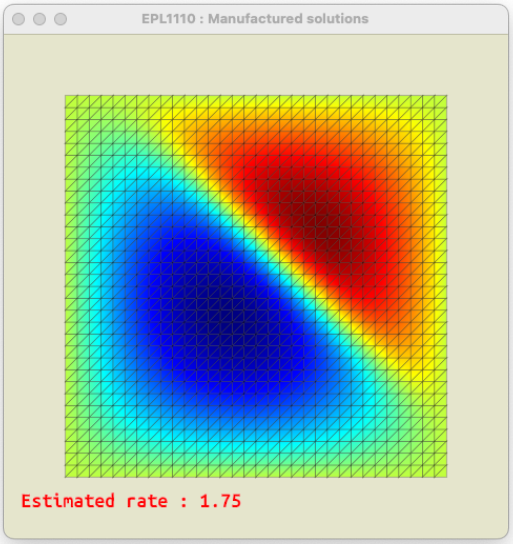
Finite elements for dummies : Solution manufacturée !



Nous allons ici illustrer comment on peut vérifier l'ordre de convergence théorique de la méthode des éléments finis. C'est très souvent utile pour s'assurer qu'un code d'éléments finis a un comportement correct et qu'il ne reste plus de petites erreurs perfides : les fameux *bugs* :-) On considère toujours notre problème elliptique modèle :

Trouver $u(x,y)$ tel que

$$\begin{aligned} \boldsymbol{\nabla}^2 u(x,y) + f(x,y) &= 0, \qquad \forall (x,y) \in \Omega, \\ u(x,y) &= 0, \qquad \forall (x,y) \in \partial\Omega, \end{aligned}$$

Nous allons sélectionner une solution analytique dont l'expression est suffisamment compliquée pour qu'elle n'appartienne pas à l'espace discret $\mathcal{U}^h$:

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

et on va résoudre l'équation de Poisson :

$$\boldsymbol{\nabla}^2 u(x,y) + f(x,y) = 0$$

sur le carré $\Omega =]0,1[\times]0,1[$ et en choisissant comme terme source $f(x,y)$ précisément le laplacien de la fonction $u(x,y)$ de sorte que celle-ci soit la solution de notre problème de Poisson. Il s'agit d'une *solution manufacturée* et le choix peut être totalement arbitraire. Ici, nous avons sélectionné une fonction proposée par André Fortin afin de représenter au mieux les comportements observés sur les vrais problèmes. On voit immédiatement que les conditions frontières essentielles homogènes sont bien satisfaites.

Pour estimer expérimentalement l'ordre de convergence de notre programme, nous allons créer une suite régulière de maillages en diminuant successivement la taille représentative des éléments. En partant d'un maillage 2×2 avec 8 éléments, on considère successivement des maillages 4×4 ... 64×64, 128×128 et 256×256. On va obtenir la solution discrète u^h sur chaque maillage et calculer les normes de l'erreur :

$$\|u - u^h\|_0 = \sqrt{\int_\Omega \Big(u(x,y) - u^h(x,y)\Big)^2 d\Omega}$$

$$\|u - u^h\|_1 = \sqrt{\int_\Omega \Big(u(x,y) - u^h(x,y)\Big)^2 + \Big(u_{,x}(x,y) - u^h_{,x}(x,y)\Big)^2 + \Big(u_{,y}(x,y) - u^h_{,y}(x,y)\Big)^2 d\Omega}$$

Et on effectuera le même calcul pour l'interpolation $\widetilde{u}^h$ obtenue en prenant la solution analytique pour toutes les valeurs nodales $\widetilde{U}_i = u(X_i, Y_i)$.

Le programme permet toujours de sélectionner le solveur plein bande ou les gradients conjugés. Avec les deux flèches à droite et à gauche (mais, pas les caractrères ¡ et ¿), on peut raffiner ou déraffiner le maillage pour observer la convergence de notre méthode d'éléments finis qui va progressivement atteindre son taux quadratique théorique de convergence asymptotique de la norme L_2 de l'erreur.

Plus concrètement, il faut implémenter les fonctions suivantes :

1. Tout d'abord, à titre d'échauffement, il vous est demandé d'écrire une petite fonction qui génére les maillages réguliers de triangles illustrés dans l'énoncé.

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

   Cette fonction va créer une structure de maillage pour un maillage de $2n^2$ éléments de $(n+1)^2$ noeuds pour le carré $[0,1]\times[0,1]$. Les noeuds seront numérotés, colonne par colonne avec le premier noeud en $(0,0)$ et le dernier noeud en $(1,1)$. Il s'agit donc de généraliser le mallage de 8 éléments construits dans le canevas de départ. Pour vous aider, nous avons fourni les maillages obtenus avec $n = 2, 4, 8$ dans les fichiers `msh2.txt`, `msh4.txt` et `msh8.txt`. Il suffit donc de comparer vos maillages obtenus avec ces fichiers de référence[1] pour vérifier que votre implémentation respecte parfaitement les spécifications requises.

2. Ensuite, il s'agit d'implémenter la solution manufacturée, ses dérivées premières et son laplacien. Comme l'équipe enseignante est vraiment trop gentille, nous vous avons fourni l'implémentation de la solution manufacturée et du gradient. Il ne reste plus qu'à obtenir l'expression (un brin compliquée :-) du laplacien.

   ```
   double convergenceSource(double x, double y);
   double convergenceSoluce(double x, double y, double *u);
   ```

   La fonction `convergenceSoluce` doit renvoyer le terme source f pour notre problème manufacturé d'éléments finis. Ici, il s'agit donc de calculer le laplacien de notre solution manufacturée et de ne pas oublier de lui adjoindre un petit signe négatif ! Comment faire ? Il suffit d'obtenir cette expression : ceci n'est donc pas un exercice de programmation, mais un simple calcul algébrique un peu compliqué[2].

   Comme l'équipe enseignante est vraiment trop gentille, on vous a fourni l'implémentation de la fonction `convergenceSoluce`. Bien observer ce qu'il faut fournir en entrée : un pointeur `u*` du vecteur de taille trois dans lequel la fonction va écrire les 3 valeurs demandées : $u(x,y)$, $u_{,x}(x,,y)$ et $u_{,y}(x,y)$. De manière redondante[3], la valeur de $u(x,y)$ est également renvoyée par la fonction.

3. Et maintenant, quelque chose d'un fifrelin plus long à réaliser !

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

   On vous demander de calculer les normes de L^2 et H^1 de l'erreur de la solution éléments finis $\|u - u^h\|_0$ et $\|u - u^h\|_1$. On effectuera le même calcul, pour l'interpolation de la solution exacte aux valeurs nodales $\|u - \widetilde{u}^h\|_0$ et $\|u - \widetilde{u}^h\|_1$. Pour calculer les intégrales sur chaque élément, on

[1] Observer que la fonction `femMeshWrite` permet de sauver un maillage dans un fichier : ce qui peut être utile pour faire cette comparaison. La commande unix `diff` est aussi particulièrement utile pour comparer deux fichiers :-)

[2] Utiliser `sympy` est évidemment permis. Les instructions `simplify` et `ccode` sont assez magiques, à tout hasard :-)

[3] Mon copain Jonathan trouve que c'est pas joli et idéal, mais moi, j'aime bien :-) Il trouve aussi que je respecte pas les règles usuelles pour le positionnement des crochets, mais j'aime bien ne pas faire comme tout le monde et les spécifications du langage C n'exigent rien à ce propos : donc, je suis libre de rester un divergent et na !

fera usage de la règle d'intégration définie pour le problème `theProblem`. Le second argument de la fonction contient le pointeur de la fonction permettant d'obtenir les valeurs de la fonction et de ses dérivées partielles : fonction qui a été fournie par nos bons soins !

Les quatre valeurs numériques seront stockées dans les attributs ad-hoc
de la structure `femDiffusionProblem`

```
theProblem->errorSoluceL2 = errorSoluceL2;
theProblem->errorSoluceH1 = erroSoluceH1
theProblem->errorInterpolationL2 = errorInterpolationL2;
theProblem->errorInterpolationH1 = errorInterpolationH1;
```

4. Et à titre de dessert, un tout petit dernier exercice pour la route :

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

Il s'agit d'obtenir une estimation du taux de convergence si nous disposons d'un vecteur contenant des erreurs e_i pour des valeurs successives de

$$h_{i+1} = \frac{h_i}{\alpha} \qquad\qquad i = 1, \cdots, n$$

avec un h_0 arbitraire. Le premier argument `errors` est un pointeur vers $n+1$ cases mémoire consécutives. Les deux autres arguments `n` et `ratio` sont respectivement n et α.

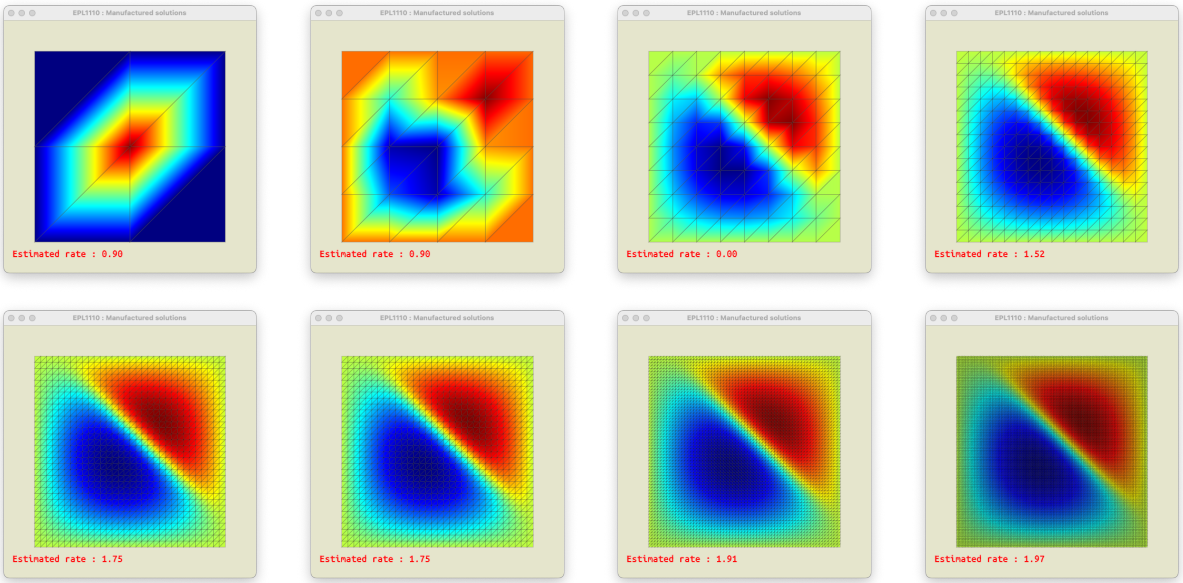
On obtiendra une estimation du taux de convergence par l'expression suivante :

$$\frac{1}{n-1} \sum_{i=1}^{n} \log\left(\frac{e_i}{e_{i+1}}\right) \frac{1}{\log(\alpha)}$$

Ah oui : on obtient comment cette expression ?

5. Vos cinq fonctions seront incluses dans un unique fichier `homework.c`, sans y adjoindre le programme de test fourni ! Ce fichier devra être soumis via le web et la correction sera effectuée automatiquement. Il est donc indispensable de respecter strictement la signature des fonctions. Votre code devra être strictement conforme au langage `C` et il est fortement conseillé de bien vérifier que la compilation s'exécute correctement sur le serveur.

Et tout cela, pourquoi ?



A l'issue de l'implémentation qu'on espère correcte de votre devoir, il est possible d'utiliser votre programme comme un tout petit laboratoire numérique pour répondre aux questions suivantes :

1. Quelle est la différence entre `Mean estimated rate` et `Last estimated rate` dans le code[4] ?
2. Observe-t-on bien les taux asymptotiques théoriques de convergence[5] ?
3. Pourquoi parler de taux asymptotique ?
4. Est-ce la solution u^h ou $\widetilde{u}^h$ qui est la représentation la plus proche de la solution ?
5. Est-ce que cela ne dépend-il pas de la manière dont on définit la solution la plus proche ?
6. Comment évolue le nombre d'itérations des gradients conjugés lorsqu'on raffine le maillage ?
7. Est-ce conforme aux résultats théoriques sur la convergence des gradients conjugés ?
8. Peut-on en déduire expérimentalement une relation entre h et le nombre de condition de la matrice du système discret ?
9. Intéger l'erreur avec un seul point d'intégration est-ce pertinent[6] ?

C'est évidemment la partie essentielle du devoir !

[4] Oui, il faut aller regarder ce que j'ai écrit dans `main.c` évidemment !
[5] Il faut faire un petite modification dans `main.c` pour obtenir le taux de convergence H^1, oui !
[6] Les petits curieux verront que j'ai ajouté une règle d'intégration avec 12 points sur les triangles dans `fem.c`. Essayer pour voir !