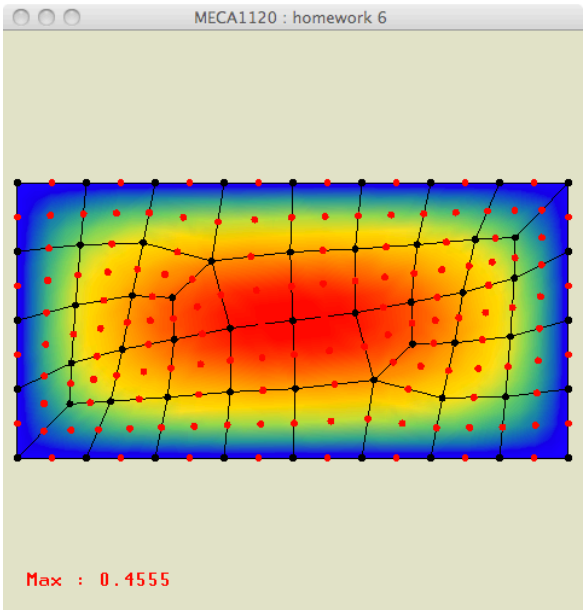
Finite elements for dummies : Numérotation pour des éléments (bi)-quadratiques et (bi)-cubiques...



Nous allons maintenant incorporer des fonctions de forme de degré deux et de degré trois à notre programme d'élements finis. On résout toujours le même problème modèle :

Trouver $u(x,y)$ tel que

$$\begin{aligned} \boldsymbol{\nabla}^2 u(x,y) + 1 &= 0, \quad \forall (x,y) \in \Omega, \\ u(x,y) &= 0, \quad \forall (x,y) \in \partial\Omega, \end{aligned}$$

Il s'agit maintenant de comparer la précision obtenue avec une méthode d'ordre plus élevé. Nous allons observer qu'à nombre égal de degrés de liberté, il peut être plus intéressant d'utiliser un plus petit nombre d'éléments avec davantage de noeuds par éléments.

En utilisant le clavier, vous pouvez toujours de manière interactive changer de solveur et d'algorithme de renumérotation sans devoir recompiler le code. Le solveur itératif est le solveur des gradients conjugués[1] qui est vraiment efficace pour le problème considéré. Il sera toujours possible de voir la solution ou d'inspecter l'allure de la matrice. L'exercice consiste à programmer, mais aussi à manipuler votre programme pour comprendre tout l'intérêt d'un solveur bande avec une bonne numérotation de noeuds ou des gradients conjugués. Mais, maintenant, il est aussi possible de choisir des fonctions de forme d'ordre un, deux ou trois. L'algorithme de renumérotation est désormais appliqué sur l'ensemble de noeuds. La visualisation a aussi été légèrement modifiée pour tenir compte des éléments d'ordre plus élevé... On a aussi ajouté une option permettant de visualiser l'ensemble de noeuds en dessinant en noir les noeuds linéaires et en rouge les autres noeuds.

Comme il y a maintenant davantage d'inconnues pour des interpolations quadratiques et cubiques que de sommets du maillage, il faudra introduire une nouvelle numérotation des variables comme suit :

- On numérote tout d'abord tous les sommets.
- On numérote ensuite les noeuds associés aux arêtes Typiquement, on ajoutera une inconnue par segment pour une interpolation quadratique et deux inconnues pour une interpolation cubique. Les noeuds seront distribués de manière uniforme le long du segment. Attention, l'ordre de ces inconnues n'est pas innocent lorsqu'on ajoute deux inconnues sur un segment.
- Finalement, on numérote les noeuds associés à l'intérieur de l'élément. Pour une interpolation biquadratique sur un quadrilatère ou une interpolation cubique sur un triangle, on ajoutera un noeud. Pour une interpolation bicubique sur un quadrilatère, on ajoutera 4 inconnues.

[1] Mais, oui c'était possible d'obtenir et d'implémenter cette méthode avec quatre vecteurs de travail... Si,si on l'expliquera, vendredi :-)

Les coordonnées des noeuds supplémentaires sont obtenues en effectuant une interpolation (bi)-linéaires des quatre sommets pour les valeurs (ξ, η) qui correspondent à la définition formelle du noeud dans l'élément parent.

	P_1C_0	P_2C_0	P_3C_0	Q_1C_0	Q_2C_0	Q_3C_0
Nombre d'inconnues associées aux sommets	3	3	3	4	4	4
Nombre d'inconnues associées aux arête	0	3	6	0	4	8
Nombre d'inconnues associées à a face	0	0	1	0	1	4
Nombre local par élément	3	6	10	4	9	16

Pour un maillages avec N_0 sommets, N_1 arêtes et N_2 éléments, on peut déduire facilement le nombre global d'inconnues à partir des caractéristiques topologiques globales du maillage. Cela est résumé dans le tableau ci-dessous :

	Nombre global d'inconnues pour un maillage
P_1C_0	N_0
P_2C_0	$N_1 + N_0$
P_3C_0	$N_2 + 2N_1 + N_0$
Q_1C_0	N_0
Q_2C_0	$N_2 + N_1 + N_0$
Q_3C_0	$4N_2 + 2N_1 + N_0$

Tout le code est prêt pour résoudre notre problème avec des fonctions de forme cubique et quadratique à l'exception de l'une ou l'autre fonction qu'un enseignant taquin a subtilisée. Plus précisément, on vous demande de concevoir, d'écrire ou de modifier quatre fonctions.

1. Tout d'abord, il s'agit d'écrire une fonction qui calcule les coordonnées des noeuds supplémentaires, ainsi que la numérotation des variables.

   ```
   void femDiffusionComputeMap(femDiffusionProblem *theProblem, int orderType)
   ```

   qui calcule les coordonnées de l'ensemble des noeuds et le tableau d'appartenance des variables de chaque élément. Les coordonnées seront placées dans `theProblem->X`, `theProblem->Y`. La dimension de ces vecteurs est le nombre global d'inconnues du problème. Le tableau d'appartenance sera construit dans `theProblem->map`, sur le modèle du tableau d'appartenance du maillage. La dimension de ce vecteur est le produit des nombres global et local d'inconnues.

   Pour le moment, la version actuelle de la fonction ne construit ces tables que pour un problème linéaire. Il faut donc la généraliser pour pouvoir les éléments quadratiques et cubiques. L'implémentation partielle doit toutefois vous aider à comprendre la procédure à appliquer. Bien être

attentif à l'ordre des noeuds sur un segment lorsqu'on ajoute deux noeuds sur une arête : il faut bien associer le noeud correct lors de la construction du tableau d'appartenance des variables.

Pour le moment, tous les noeuds supplémentaires sont initialisés à l'origine, c'est pourquoi vous ne verrez qu'un unique point rouge au départ.

Pour développer votre programme, une fonction `femDiffusionMapPrint` permet d'imprimer le tableau d'appartenance des inconnues que vous aurez construit. C'est une bonne idée d'en tirer profit lors de la mise au point de la fonction.

2. Finalement, il s'agira d'écrire les fonctions de forme (bi)-cubiques, ainsi que leurs dérivées dans l'espace parent. Il faut respecter strictement la numérotation définie par les fonctions `p3c0_x` et `q3c0-x` fournies qui donnent la position des abscisses d'interpolation dans l'élément parent. L'utilisation du calcul symbolique de MATLAB est une manière astucieuse d'obtenir le résultat demandé. Comme nous sommes gentils, nous vous avons fournis les fonctions de forme cubiques sur le triangle : c'était une partie d'un des problèmes de l'année passée :-)

```
void q3c0_phi(double xsi, double eta, double *phi)
void q3c0_dphidx(double xsi, double eta, double *dphidxsi, double *dphideta)
void p3c0_dphidx(double xsi, double eta, double *dphidxsi, double *dphideta)
```

   Afin de vérifier l'exactitude de vos fonctions de forme, c'est une bonne idée d'utiliser la fonction `femDiscretePrint` qui imprime l'ensemble des fonctions de forme pour les positions d'interpolation. Chaque fonction de forme doit valoir l'unité à la position associée et s'annuler pour toutes les autres abscisses. On peut aussi vérifier que la somme des dérivées doit toujours s'annuler.

3. Vos quatre fonctions seront incluses dans un unique fichier `homework.c`, sans y adjoindre le programme de test fourni ! L'obtention d'un code quadratique est assez aisé, tandis que la version cubique est un peu plus délicate... On en tiendra compte dans la correction. 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.

4. Afin de permettre aux étudiants impliqués dans l'activité socio-culturelle de la semaine prochaine, vous recevrez deux semaines pour ce problème-ci ! Il est donc possible de le faire avant ou après la revue des ingénieurs. Bon travail à tous.