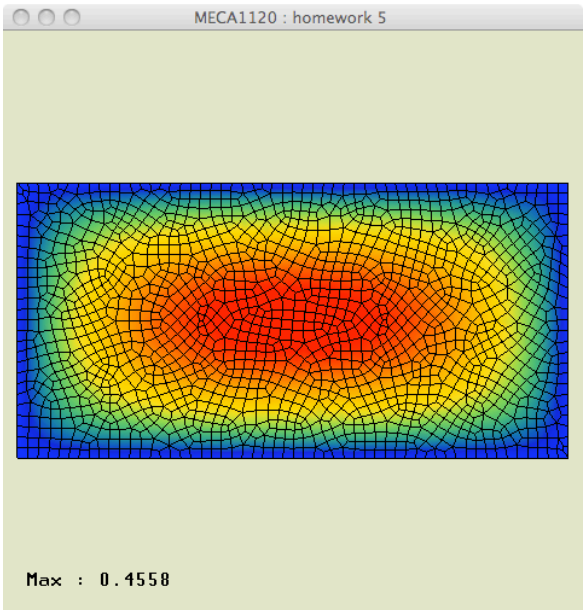
Finite elements for dummies : Résolution d'un système linéaire creux...

Nous allons reprendre notre premier programme d'éléments finis et tenter de le rendre un peu plus efficace ! Nous utilisons toujours des éléments triangulaires linéaires continus ou des éléments quadrilatères bilinéaires continus pour résoudre le problème suivant :

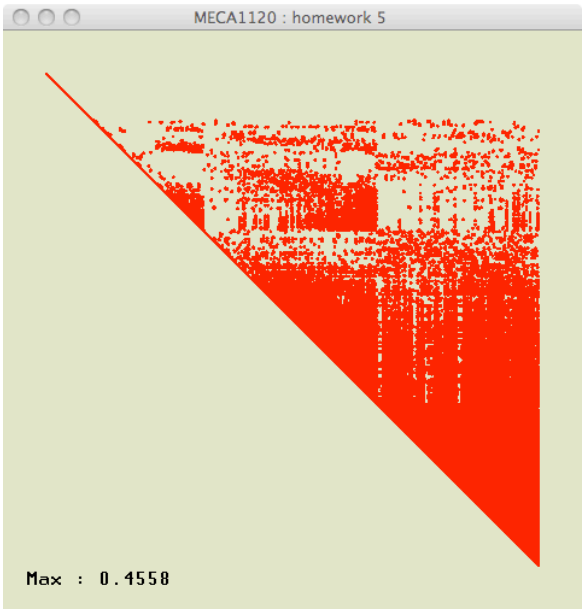


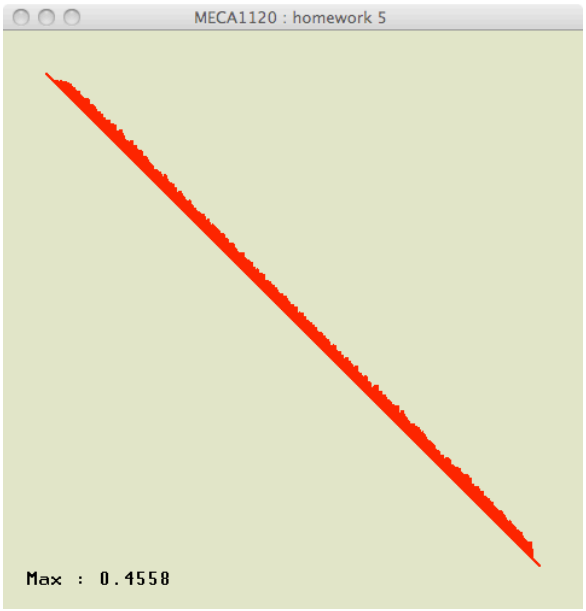
Trouver $u(x,y)$ tel que

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

Il s'agit maintenant de comparer les performances de divers solveurs pour la résolution du système linéaire discret. Nous allons comparer les performances d'un solveur direct plein, du solveur direct bande et d'un solveur itératif, à savoir la méthode des gradients conjugués.

En utilisant le clavier comme sur une console de jeu élémentaire, vous allez pouvoir de manière interactive changer le solveur utilisé, et d'algorithme de renumérotation sans devoir recompiler le code. 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 d'un solveur itératif efficace. N'hésitez donc pas à utiliser le programme avec les maillages `triangles_166.txt`, `triangles_166_xopt.txt` et `triangles_166_yopt.txt`. où trois numérotations distinctes ont été utilisées dans la numérotation de noeuds. En comparant les résultats obtenus avec le solveur bande, on observe que bien renuméroter les noeuds est critique.





Mais pour les autres maillages, ce sera votre travail de renuméroter les noeuds. Par contre, l'implémentation du solveur bande vous est fournie : en d'autres mots, la plus grosse partie du travail a été effectuée par nos soins.

- Pour la résolution du système linéaire, vous disposerez de trois solveurs avec une interface commune. Il est donc possible d'avoir une implémentation unique du problème, en pouvant faire appel à n'importe lequel des trois solveurs et pouvant modifier ce choix sans devoir recompiler le programme. Toutes les fonctions permettant l'allocation, l'initialisation, l'assemblage et la résolution d'un système linéaire ont été ajoutées dans le module `fem`. Il est aussi possible de contraindre la valeur d'un noeud ou d'imprimer la forme courante du système linéaire.

```
femSolver*           femSolverFullCreate(int size);
femSolver*           femSolverBandCreate(int size,int band);
femSolver*           femSolverIterativeCreate(int size);
void                 femSolverFree(femSolver* mySolver);
void                 femSolverInit(femSolver* mySolver);
void                 femSolverPrint(femSolver* mySolver);
void                 femSolverPrintInfos(femSolver* mySolver);
double*              femSolverEliminate(femSolver* mySolver);
void                 femSolverConstrain(femSolver* mySolver, int myNode, double value);
void                 femSolverAssemble(femSolver* mySolver, double *Aloc, double *Bloc, double *Uloc, int *map, int nLoc
double               femSolverGet(femSolver* mySolver, int i, int j);
int                  femSolverConverged(femSolver *mySolver);

femFullSystem*       femFullSystemCreate(int size);

femBandSystem*       femBandSystemCreate(int size, int band);

femIterativeSolver*  femIterativeSolverCreate(int size);
```

  Il s'agit d'une implémentation de l'héritage dans la syntaxe du C. On observe ainsi que la structure `femSolver` permet d'accèder de manière transparente à l'un des trois solveurs sans que le problème doive connaître a priori la structure utilisée.

Plus précisément, on vous demande de concevoir, d'écrire ou de modifier cinq fonctions.

1. Tout d'abord, vous mettrez au point une fonction

```
int femDiffusionComputeBand(femDiffusionProblem *theProblem)
```

   qui calcule la largeur de bande de la matrice du système discret. Pour rappel, la largeur de bande d'une matrice A_{ij} est la plus grande distance à la diagonale que peut atteindre un élément non nul de la matrice, augmentée d'une unité, ce que nous écrirons

$$\beta(\mathbf{A}) - 1 = max_{ij} \left\{ |i-j|, \ \ \forall (i,j) \text{ tels que } a_{ij} \neq 0 \right\}.$$

   Pour le moment, la version actuelle de la fonction renvoie simplement, la taille de la matrice. Ce n'est évidemment pas la meilleure manière de tirer profit du solveur bande. Comme l'assemblage se fait avant l'application des conditions aux frontières, il n'est pas possible de tenir compte que la largeur de bande est éventuellement réduite lors de l'application des conditions frontières : il faut donc juste appliquer la formule telle que donnée ci-dessus.

2. Ensuite, il s'agira d'optimiser la numérotation pour réduire la largeur de bande.
   C'est le rôle de la fonction qui doit calculer les indices de variables que l'on va associer à chaque noeud.

```
void femDiffusionRenumber(femDiffusionProblem *theProblem, femRenumType renumType)
```

   Les indices de variables que l'on va associer à chaque noeud seront incluses dans le tableau `myProblem->number` qui est initialisé par défaut comme suit :

```
for (i = 0; i < theProblem->mesh->nNode; i++)
      theProblem->number[i] = i;
```

   Cela correspond donc à ne pas renuméroter les noeuds ! Trois stratégies de renumérotations seront considérées en fonction de `renumType`.

   - Ne rien faire (`FEM_NO`),
   - Numéroter les noeuds en ordre croissant des abscisses (`FEM_XNUM`),
   - Numéroter les noeuds en ordre croissant des ordonnées (`FEM_YNUM`).

   Afin de pouvoir vérifier votre algorithme de renumérotation, nous avons fourni le maillage en y ayant appliqué les trois renumérotations : il s'agit `triangles_166.txt`, `triangles_166_xopt.txt` et `triangles_166_yopt.txt`. En n'appliquant aucune renumérotation sur un maillage déjà numéroté de manière optimale, vous obtiendrez la même largeur de bande que sur un maillage non numéroté auquel vous avez appliqué une renumérotation : c'est simple, non ?

3. Finalement, il s'agira d'implémenter la méthode des gradients conjugués en partant du solveur itératif fourni et en modifiant les trois fonctions suivantes.

```
double* femIterativeSolverEliminate(femIterativeSolver* mySolver);
void    femIterativeSolverConstrain(femIterativeSolver* mySolver, int myNode, double myValue);
void    femIterativeSolverAssemble(femIterativeSolver* mySolver, double *Aloc, double *Bloc,
                                   double *Uloc, int *map, int nLoc);
```

   La version fournie dans l'ébauche du programme contient un algorithme très élémentaire : on progresse dans la direction de la plus grande pente avec un facteur constant $\alpha = 1/5$. A chaque itération, on effectue l'opération suivante :

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha \left(\mathbf{A}\mathbf{x}^k - \mathbf{b} \right)$$

   Il ne vous est pas permis de modifier les autres fonctions de la librairie. En étant attentifs, vous observerez que 4 vecteurs ont été alloués dans la structure, vous pouvez donc assembler quatre vecteurs pour effectuer chaque itération. La méthode des gradients conjugués est définie dans une note disponible sur le site du cours.

4. 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.