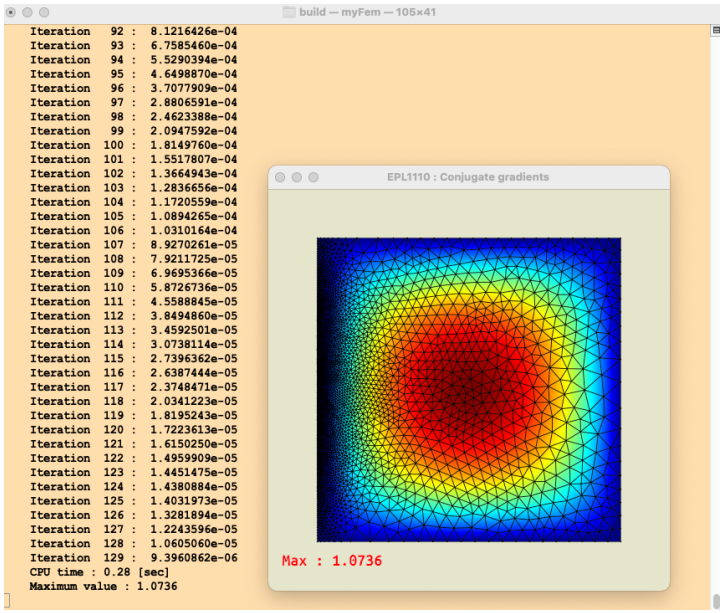
Finite elements for dummies : Méthode des gradients conjugés...

Nous allons maintenant implémenter une version *matrix-free* de la méthode des gradients conjugés pour résoudre ce problème d'élements finis



Trouver $u(x,y)$ tel que

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

L'algorithme des gradients conjugés est défini par l'itération suivante que Hestenes et Stiefel ont rendu célèbre dans le monde de l'analyse numérique.

$$\begin{aligned} \alpha^k &= \frac{\mathbf{r}^k \cdot \mathbf{r}^k}{\mathbf{A}\mathbf{d}^k \cdot \mathbf{r}^k} \\ \mathbf{x}^{k+1} &= \mathbf{x}^k + \alpha^k \mathbf{d}^k \\ \mathbf{r}^{k+1} &= \mathbf{r}^k - \alpha^k \mathbf{A}\mathbf{d}^k \\ \beta^k &= \frac{\mathbf{r}^{k+1} \cdot \mathbf{r}^{k+1}}{\mathbf{r}^k \cdot \mathbf{r}^k} \\ \mathbf{d}^{k+1} &= \mathbf{r}^{k+1} + \beta^k \mathbf{d}^k \end{aligned}$$

en démarrant le calcul avec $\mathbf{x}^0 = 0$ et $\mathbf{d}^0 = \mathbf{r}^0 = \mathbf{b} - \mathbf{A}\mathbf{x}0$.

Attention, on n'effectue pas le produit matriciel $\mathbf{A}\mathbf{d}^k$, mais on assemble un vecteur $\mathbf{s}^k = \mathbf{A}\mathbf{d}^k$. Tout l'intérêt d'un algorithme itératif est de ne pas nécessiter le stockage d'une matrice de grande taille : c'est une implémentation *matrix-free* qu'on veut réaliser :-)

Il s'agira d'implémenter deux fonctions. Tout d'abord, il s'agira de faire l'assemblage du terme $\mathbf{A}\mathbf{d}$. Et ensuite d'écrire les opérations qu'il convient de faire à chaque itération de nos gradients conjugés. Pour rendre plus heureuse Julia, je vais essayer d'expliquer le plus clairement possible ce que doivent faire ces deux fonctions. Ceci dit, il est aussi permis d'imaginer que pour 350 étudiants travaillant de manière collaborative et solidaire, il soit possible de décrypter ce que j'ai imaginé comme devoir.

Beh oui : c'est cela la vrai vie des ingénieurs !
Trouver la solution d'un problème compliqué d'un enseignant trouve simple et explique mal :-)

- Pour implémenter les gradients conjugés, nous avons créé un ensemble de fonctions afin d'avoir exactement le même interface (API) pour les éléments finis qu'avec le solveur plein et le solveur bande.

```
typedef struct {
    double *R;
    double *D;
    double *S;
    double *X;
    double error;
    int size;
    int iter;
} femIterativeSolver;


femIterativeSolver*  femIterativeSolverCreate(int size);
void                 femIterativeSolverFree(femIterativeSolver* mySolver);
void                 femIterativeSolverInit(femIterativeSolver* mySolver);
void                 femIterativeSolverPrint(femIterativeSolver* mySolver);
void                 femIterativeSolverPrintInfos(femIterativeSolver* mySolver);
double*              femIterativeSolverEliminate(femIterativeSolver* mySolver);
void                 femIterativeSolverAssemble(femIterativeSolver* mySolver, double *Aloc,
                                                double *Bloc, double *Uloc, int *map, int nLoc);
double               femIterativeSolverGet(femIterativeSolver* mySolver, int i, int j);
int                  femIterativeSolverConverged(femIterativeSolver *mySolver);
```

  La structure contient 4 vecteurs qui seront respectivement à chaque itération :

  - Le vecteur `R` contiendra la valeur courante du residu $\mathbf{r}^k$.
  - Le vecteur `D` contiendra la valeur courante de la direction congugée $\mathbf{d}^k$.
  - Le vecteur `S` contiendra la valeur courante de $\mathbf{s}^k = \mathbf{A}\mathbf{d}^k$.
  - Le vecteur `X` contiendra l'incrément de la solution $\mathbf{x}^{k+1} - \mathbf{x}^k$.

  On stocke également un réel `error` qui contient une estimation de l'erreur courante, un entier `iter` qui contient le numéro k de l'itération courante et finalement la taille `size` du problème.

Malencontreusement, un lutin un brin taquin a chapardé `fem...Eliminate` et `fem...Assemble`.

Votre mission consiste donc à rétrouver les bribes manquantes. Un assistant pas très doué a réussi péniblement à proposer quelque chose de pas très efficace qui implémente un algorithme de la plus grand pente avec un facteur $\alpha = 0.2$ constant. Cela converge, mais c'est affreusement lent !
En d'autres mots, la version fournie dans l'ébauche du programme contient un algorithme très élémentaire qui permet d'obtenir la solution mais avec beaucoup de patience...

A chaque itération, on effectue l'opération suivante en appelant la fonction `fem...Eliminate`[1] :

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

après avoir assemblé le résidu dans la fonction `fem...Assemble`.

Il ne vous est pas permis de modifier les autres fonctions de la librairie !
Il ne vous est pas permis d'allouer d'autres vecteurs pour faire votre calcul !

[1] On pourrait discuter de l'usage du mot `Eliminate` dans le nom de la fonction, car on effectue plutôt une itération `Iterate`. C'est un choix effectué pour avoir la même API pour les solveurs directs et itératifs. D'une certaine manière, un solveur direct est un solveur itératif qui converge en une unique itération ! Ainsi, on aurait peut-être pu plutôt utiliser `Iterate` pour nos solveurs directs.... mais tout cela est un débat sans fin

Plus concrètement, il faut implémenter les deux fonctions suivantes :

1. Tout d'abord, il faudra écrire la fonction qui assemble

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

   Cette fonction assemblera le résidu pour la première itération. Pour les autres itérations, il assemblera le vecteur $\mathbf{s}^k = \mathbf{A}\mathbf{d}^k$ dans le tableau `S` à partir de la matrice locale et du vecteur disponible `D` à cet instant. On supposera évidemment que les vecteurs `R` et `S` auront été initialisés à zéro avant d'entamer la procédure de leur assemblage.

2. Ensuite, il faudra simplement implémenter une itération des gradients conjugés dans la fonction suivante :

```
double* femIterativeSolverEliminate(femIterativeSolver* mySolver);
```

   Pour la première itération, on initialise $\mathbf{x}^0 = 0$ et $\mathbf{d}^0 = \mathbf{r}0$. On a bien assemblé le résidu dans la fonction d'assemblage.

   Pour les autres itérations, on calcule α^k, on incrémente $\mathbf{r}^k$ pour obtenir $\mathbf{r}^{k+1}$, on calcule β^k, on calcule $\mathbf{d}^k$. Et on ré-initialise le tableau `S` pour qu'il soit prêt pour le nouvel assemblage.

   L'implémentation fournie de l'algorithme de la plus grande pente peut vous servir d'inspiration.

   Pour toutes itérations, on estimera[2] l'erreur commise en calculant $\sqrt{\mathbf{r}^k \cdot \mathbf{r}^k}$. Cette valeur sera mise dans l'attribut `error` de notre structure de donnée. Et finalement, on renverra l'incrément $\mathbf{x}^{k+1} - \mathbf{x}^k$ comme argument de sortie.

   L'implémentation fournie de l'algorithme de la plus grande pente devrait à nouveau vous servir d'inspiration.

3. Et les gradients conjugés préconditionnés ?
   Cela sera un des objectifs pour le projet final !
   Ne soyez donc pas impatients :-)

4. Vos deux 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.

[2] On peut remarquer qu'estimer l'erreur à chaque itération est une question vraiment difficile, je vous invite à ce propos à relire la section consacrée à cette question dans le texte de Jonathan Shewchuk :-)
C'est vraiment une question difficile que Jean-François n'a pas abordé au passage !