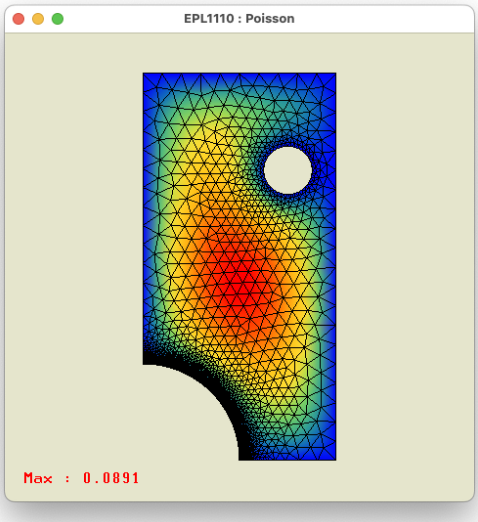
Finite elements for dummies :
Un petit Poisson :-)

Nous allons écrire notre premier vrai programme d'éléments finis ! Evidemment, ce sera encore un problème assez simple puisque nous allons utiliser des éléments triangulaires linéaires continus ou des éléments quadrilatères bilinéaires continus pour résoudre le problème suivant :



Trouver $u(x,y)$ tel que

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

Il s'agit donc de construire le système linéaire et de le résoudre après avoir imposé les conditions essentielles homogènes sur la frontière.

Pour la gestion et la résolution du système linéaire, un ensemble de fonctions ont été présentées lors du dernier cours et ont été incorporées dans la librairie. Il faudra donc uniquement assembler la matrice de raideur et le vecteur des forces, imposer les conditions aux limites et ensuite effectuer l'élimination de Gauss. Par ailleurs, nous allons utiliser une élimination de Gauss sur une matrice pleine : ce qui n'est pas la meilleure idée car la matrice obtenue est une matrice creuse et utiliser un solveur creux serait nettement plus efficace. C'est aussi la raison pour laquelle vous êtes invités à ne pas utiliser des maillages de trop grande taille, en particulier si votre ordinateur n'est pas une bête de course.

- Toutes les fonctions permettant l'allocation, l'initialisation 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.

```
femFullSystem* femFullSystemCreate(int size);
void           femFullSystemFree(femFullSystem* mySystem);
void           femFullSystemPrint(femFullSystem* mySystem);
void           femFullSystemInit(femFullSystem* mySystem);
double*        femFullSystemEliminate(femFullSystem* mySystem);
void           femFullSystemConstrain(femFullSystem* mySystem, int myNode, double value);
```

  Observer que la fonction `femFullSystemPrint` est particulièrement utile pour la mise au point de votre petit programme.

- Pour préparer progressivement la suite de nos aventures, les fonctions de forme sont également fournies dans la petite bibliothèque fournie. Il faut faire appel impérativement aux fonctions de forme fournies, car nous allons tester votre code en modifiant ces fonctions de forme (par exemple, en modifiant la numérotation locale des fonctions de forme). En conséquence, recoder en interne les fonctions de forme ne vous permettra pas de réussir le devoir.

```
femDiscrete* femDiscreteCreate(int n, femElementType type);
void         femDiscreteFree(femDiscrete* mySpace);
void         femDiscretePrint(femDiscrete* mySpace);
void         femDiscreteXsi(femDiscrete* mySpace, double *xsi, double *eta);
void         femDiscretePhi(femDiscrete* mySpace, double xsi, double eta, double *phi);
void         femDiscreteDphi(femDiscrete* mySpace, double xsi, double eta, double *dphidxsi, double *dphideta);
```

- Les fichiers `tiny.txt` et `example.txt` sont les maillages élémentaires des exemples du cours et du syllabus. C'est une bonne idée de les utiliser pour mettre au point votre programme.

La définition complète du problème aux conditions aux limites se fait dans la structure suivante :

```
typedef struct {
    femGeo *geo;
    femDiscrete *space;
    femIntegration *rule;
    femFullSystem *system;
} femPoissonProblem;
```

En bref, il faut une géométrie, un espace de discrétisation avec des fonctions de forme, une règle d'intégration et un solveur linéaire pour résoudre le système ainsi construit.

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

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

   ```
   femPoissonFindBoundaryNodes(femPoissonProblem *theProblem)
   ```

   qui construit un nouveau domaine contenant les noeuds de toute la frontière. Ces noeuds seront stockés comme une liste d'un nouveau domaine contenant des éléments de taille zéro (les noeuds) et le domaine s'appellera `Boundary`. Attention, la table `map` contiendra les indices des variables liés à un élément.

2. Ensuite, vous mettrez au point une fonction :

   ```
   void femPoissonLocal(femPoissonProblem *theProblem, const int i, int *map, double *x, double *y)
   ```

   qui copie les indices de variables et les coordonnées des sommets pour un élément `i` dans les tableaux `map`, `x` et `y`. Attention, la table `map` contiendra les indices des variables liés à un élément.

3. La plus grosse partie du devoir consistera ensuite à intégrer, assembler et résoudre le problème discret correspondant au problème de Poisson.

```
void femPoissonSolve(femPoissonProblem *theProblem)
```

   qui résout le problème de Poisson sur le maillage `theProblem->mesh` en imposant une condition essentielle sur les noeuds des segments frontières contenus dans `theProblem->edges`. Le système linéaire est contenu dans la structure `theProblem->system`. A la fin du processus d'assemblage, il faut évidemment résoudre le système linéaire.
   La solution finale se trouvera alors dans `theProblem->system->B`.

   Pour chaque élément, le système linéaire local est ajouté au contenu courant dans la structure `theProblem->system`. Il s'agit d'assembler la matrice et le membre de droite ! La règle d'intégration et les fonctions de forme qui définissent l'espace discret sont définies dans `theProblem->rule` et `theProblem->space` respectivement. Le constructeur et le destructeur de la structure de votre problème sont fournis : il est utile d'observer que l'on détruit les objets dans l'ordre inverse de la création de ceux-ci, puisqu'on commence à vider l'intérieur de la structure avant de détruire le contenant, tandis que c'est exactement l'inverse lors de la création de la structure du problème.

4. Finalement, il s'agira de désallouer toutes les structures crées pour résoudre le problème.
   C'est le rôle de la fonction :

```
void femPoissonFree(femPoissonProblem *theProblem)
```

   Si vous avez bien implémenté cette dernière fonction, le check de la désallocation de la mémoire doit être parfaitement réussi avec `valgrind` sur le serveur.

5. Vos quatre 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.

6. Afin de pouvoir effectuer un test distinct de vos fonctions, des instructions de compilation ont été mis dans le fichier `homework.c`. Il est IMPERATIF de ne pas les retirer, ni de les modifier...