# SIM201 - Examen
# Résolution du problème de Helmholtz

E. Lunéville

*Durée :* ***3h***
*Documents autorisés : polycopié, transparents du cours, TP réalisés auparavant, documentation en ligne du C++ (`https://cplusplus.com/reference/stl/`)*
*L'utilisation de programme ou module de* ***génération de code utilisant de l'intelligence artificielle (Copilot, ChatGPT,...) est interdite****. Il s'agit d'un* ***travail individuel****, toute copie ou plagiat sera sanctionné. Il est conseillé de* ***valider progressivement*** *son travail à l'aide du programme principal fourni. La* ***présentation*** *et le fait que* ***le programme compile et s'exécute*** *est noté sur* ***3 pts****.* Le nombre de points affecté à chacune des parties est prévisionnel.

L'objectif de ce TP est la réalisation d'un programme C++ permettant de résoudre par éléments finis d'ordre 1, le problème de Helmholtz dans un domaine borné $\Omega$ de dimension 1 ou 2 :

$$\begin{cases} \Delta u + k^2 u = f & \text{dans } \Omega \qquad (f \in L^2(\Omega)) \\ \dfrac{\partial u}{\partial n} = 0 & \text{sur } \partial\Omega. \end{cases}$$

On rappelle que la discrétisation de la formulation variationnelle de ce problème dans l'espace éléments finis

$$V_h = \left\{ v \in C^0(\bar{\Omega});\ v_{|E_\ell} \in P^1(E\ell), \forall \ell = 1, L \right\}$$

où $\cup_{\ell=1,L} E_\ell = \bar{\Omega}$ ($E_\ell$ un segment ou un triangle) définit un maillage admissible dont les nœuds sont désignés par $(M_i)_{i=1,N}$, conduit au système linéaire

$$(\mathbb{K} - k^2\mathbb{M})U = \mathbb{M}F,$$

$$\text{avec} \quad \left| \begin{array}{l} \mathbb{K}_{ij} = \displaystyle\int_\Omega \nabla w_i.\nabla w_j \, d\Omega, \quad \mathbb{M}_{ij} = \int_\Omega w_i \, w_j \, d\Omega, \quad F_i = f(M_i) \quad \forall i,j = 1,N \\ w_i \in V_h \text{ telle que } w_i(M_j) = \delta_{ij} \quad \forall i,j = 1,N. \end{array} \right.$$

Il s'agit donc de fabriquer les matrices $\mathbb{K}$, $\mathbb{M}$ et de résoudre le système linéaire. Pour ce faire, on aura besoin

— d'une classe `Point` permettant de gérer des points 1D ou 2D
— d'une classe `Maillage` dont hériteront les classes `Maillage1D`, `Maillage2D` pour gérer, respectivement, des géométries 1D (segment $[a,b]$) et des géométries 2D (rectangle $[a,b] \times [c,d]$)
— d'une classe `EF` dont hériteront les classes `EF1D`, `EF2D` pour gérer, respectivement, les calculs éléments finis 1D et 2D
— d'une classe `Sparse` gérant des matrices creuses qui est fournie (fichier *sparse.hpp*)
— d'une classe permettant de gérer des vecteurs : on utilisera la classe `vector` de la STL pour laquelle les opérations algébriques `+, -, *, /, |` et la fonction `norme` sont également fournies (fichier *sparse.hpp*).

## Partie 1 : la classe Point (5 pts)

On propose d'utiliser une classe `Point` gérant explicitement l'abcisse `x` et l'ordonnée `y` d'un point, ainsi que la dimension `dim` permettant de distinguer s'il s'agit d'un point 1D ou d'un point 2D :

```
class Point
{public:
  int dim;                     //dimension du point (1 ou 2)
  double x, y;                 //abcisse , ordonnée
  Point();                     //constructeur par défaut (0D)
  Point(double a);             //constructeur 1D
  Point(double a, double b);   //constructeur 2D
  ...
};
```

Dans ce qui suit, `P`,`Q` désignent des points, `a` un scalaire et `out` un objet de la classe `ostream`. On prendra en compte la dimension du point dans tous les implémentations.

1) Écrire l'implémentation des constructeurs de la classe `Point`.
2) Proposer la surcharge de l'opérateur `<<` permettant de réaliser l'affichage d'un point à l'aide de la commande `out<<P` sous la forme `(x)` ou `(x,y)`.
3) Écrire les surcharges d'opérateurs permettant de réaliser les opérations `P+=Q, P-=Q, P*=a, P/=a`.
4) Écrire les surcharges d'opérateurs permettant de réaliser les opérations `P+Q, P-Q, a*p, P*a, P/a, P|Q` et la fonction `norme(P)`.
5) Écrire la fonction `mesure` permettant de calculer la surface d'un triangle donné par ses 3 sommets $A, B, C$ (utiliser le produit vectoriel : $\text{mes}(A,B,C) = \frac{1}{2}|AB \times AC|$).

## Partie 2 : les classes Maillage (5 pts)

On rappelle qu'un maillage est une collection d'éléments (segments en 1D, triangles en 2D) que l'on peut décrire par un vecteur de points (sommets des éléments comptés une seule fois) et une liste des numéros des sommets de chacun des éléments (pair en 1D, triplet en 2D). On définit ainsi la classe `Maillage` :

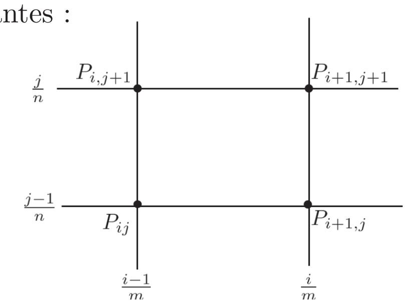
```
typedef vector<int> Numeros;
class Maillage
{public:
  int dim;                          // dimension de la géométrie (1,2)
  string geometrie;                 // nom de la géométrie maillée
  vector<Point> noeuds;             // liste des noeuds comptés une seule fois
  list<Numeros> numelts;            // liste des numéros des noeuds
  Maillage(int d=0, const string& geo="");  // constructeur
  void print(ostream& out=cout) const;      // affichage du maillage
  ...
}
```

6) Proposer un constructeur de la classe `Maillage`, initialisant `dim` et `geometrie`.
7) Écrire l'implémentation de la fonction `print` affichant le maillage sous la forme :

```
Maillage du segment [0,1]
Liste des noeuds (11 points)
(0),(0.1),...
Liste des elements (10 segments)
(0,1),(1,2),...
```

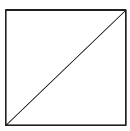
   et proposer la surcharge de l'opérateur `<<` pour un objet de la classe `ostream` et un objet de la classe `Maillage`.
8) Proposer une classe `Maillage1D` héritant de la classe `Maillage` proposant le constructeur :
   `Maillage1D(double a, double b, int m)`
   réalisant le maillage du segment $[a,b]$ en $m$ intervalles.
9) Le carré unité $[0,1] \times [0,1]$ est maillé en se basant sur un découpage en $m > 0$ segments suivant l'axe $x$ et en $n > 0$ segments suivant l'axe $y$. Ce découpage conduit à $m \times n$ rectangles de taille $1/m$ par $1/n$. En parcourant les rectangles de gauche à droite et de bas en haut, les coordonnées des rectangles (noeuds du maillage) sont données par les formules suivantes :

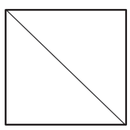
$$\forall i = 1,m,\ j = 1,n$$
$$P_{ij} = \left(\tfrac{i-1}{m}, \tfrac{j-1}{n}\right),\ P_{i+1,j} = \left(\tfrac{i}{m}, \tfrac{j-1}{n}\right),$$
$$P_{i,j+1} = \left(\tfrac{i-1}{m}, \tfrac{j}{n}\right),\ P_{i+1,j+1} = \left(\tfrac{i}{m}, \tfrac{j}{n}\right)$$




   Chacun des rectangles est ensuite découpé en deux triangles de façon aléatoire[1] :

$(P_{i+1,j}, P_{i+1,j+1}, P_{ij})$
$(P_{i,j+1}, P_{ij}, P_{i+1,j+1})$





$(P_{ij}, P_{i+1,j}, P_{i,j+1})$
$(P_{i+1,j+1}, P_{i,j+1}, P_{i+1,j})$

1. voir **rand** et **srand**, entête *cstdlib*

Proposer une classe `Maillage2D` héritant de la classe `Maillage` proposant la fonction :

`void maille_carre_unite(int m,int n)`

réalisant le maillage du carré unité.

10) Proposer un constructeur réalisant un maillage $(m \times n)$ du carré unité ainsi qu'un constructeur proposant la réalisation d'un maillage $(m \times n)$ du rectangle $[a, b] \times [c, d]$ s'appuyant sur une transformation affine du maillage d'un carré unité.

## Partie 3 : les classes éléments finis (2 pts)

On rappelle que les matrices de masse et de rigidité élémentaires sur l'élément $E^\ell$ de sommets $M_p^\ell$ sont données par

$$\mathbb{M}_{pq}^\ell = \int_{E_\ell} \tau_p^\ell \, \tau_q^\ell \, d\Omega, \quad \mathbb{K}_{pq}^\ell = \int_{E_\ell} \nabla \tau_p^\ell . \nabla \tau_q^\ell \, d\Omega \quad \text{où } \tau_p^\ell \in P^1(T^\ell) \text{ et } \tau_p^\ell(M_q^\ell) = \delta_{pq} \ \forall p, q.$$

On introduit la classe abstraite

```
class EF
{public :
  EF(int d): dim(d){}
  int dim;  //dimension de l'élément (1 ou 2)
  virtual void masseP1(const vector<Point>&, Matrix&) const =0; // matrice de masse élémentaire
  virtual void rigidP1(const vector<Point>&, Matrix&) const =0; // matrice de rigidité élémentaire
};
```

11) Proposer les classes `EF1D` et `EF2D` héritant de la classe `EF` et fournissant le calcul des matrices élémentaires dont les expressions sont données ci-après en fonction des sommets $S_1^\ell$, $S_2^\ell$ d'un segment et $S_1^\ell$, $S_2^\ell$, $S_3^\ell$ d'un triangle ($s_{ij} = S_j^\ell - S_i^\ell$, $m = \text{mes}(S_1^\ell, S_2^\ell, S_3^\ell) = \frac{1}{2}|s_{12} \times s_{13}|$) :

$$\text{segment : } \mathbb{M}^\ell = \frac{|s_{12}|}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad \mathbb{K}^\ell = \frac{1}{|s_{12}|} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

$$\text{triangle : } \mathbb{M}^\ell = \frac{m}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \quad \mathbb{K}^\ell = \frac{1}{4\,m} \begin{bmatrix} s_{23}|s_{23} & -s_{13}|s_{23} & s_{12}|s_{23} \\ -s_{13}|s_{23} & s_{13}|s_{13} & -s_{12}|s_{13} \\ s_{12}|s_{23} & -s_{12}|s_{13} & s_{12}|s_{12} \end{bmatrix}.$$

## Partie 4 : la classe Helmholtz (5 pts)

Afin de résoudre le problème de Helmholtz, on introduit la classe *template* `Helmholtz` dont les types abstraits sont une classe de maillage (`MT`) et une classe d'élément fini (`EFT`). Elle pilote les données du problème : maillage (pointeur), nombre d'onde $k$, les fonctions $f$,$u_{ex}$ (pointeurs de fonction), le calcul des matrices ainsi que la résolution du système linéaire issu de la discrétisation par éléments finis :

```
template <typename MT, typename EFT>
class Helmholtz
{protected:
  const MT* mail_ = nullptr;          //pointeur sur un maillage de type MT
  EFT ef_;                            //élément fini de type EFT
  double k_;                          //nombre d'onde du problème
  fun f_, uex_;                       //fonction f et uex si connue
 public:
  Sparse M,K;                         //matrices K et M
  Vecteur sol;                        //vecteur solution
  Helmholtz(const MT& m, double k, fun f, fun uex=nullptr);
  void assembleMatrices();            //calcul assemblé des matrices K et M
  Vecteur& resoudre();                //résolution du systeme (K-k^2*M)*U=M*F
};
```

Les matrices globales $\mathbb{M}$ et $\mathbb{K}$ s'obtiennent en réalisant une boucle sur tous les éléments, en calculant pour chaque élément $T^\ell$ les matrices élémentaires $\mathbb{M}^\ell$ et $\mathbb{K}^\ell$ et en assemblant ces matrices dans les matrices globales à l'aide de la liste `Maillage::numelts` qui relie le numéro local d'un nœud sur un élément à son numéro global dans la liste `Maillage::noeuds`.

12) Écrire l'implémentation du constructeur de la classe `Helmholtz`.

13) Écrire l'implémentation de la fonction `Helmholtz<MT,EFT>::assembleMatrices`.

14) Écrire l'implémentation de la fonction `Helmholtz<MT,EFT>::resoudre` qui, à partir des matrices $\mathbb{M}$ et $\mathbb{K}$, construit la matrice et le second membre du système linéaire, puis appelle le solveur itératif `gradConj` fourni (voir fichier *sparse.hpp*).

## Questions bonus (2 pts)

15) Écrire une fonction `double Helmholtz<MT,EFT>::calculErreur()` qui calcule l'erreur $L^2$ lorsque la solution exacte est donnée. On rappelle que $\forall u_h \in V_h$ $(U = (u_h(M_i))_{i=1,N})$ :

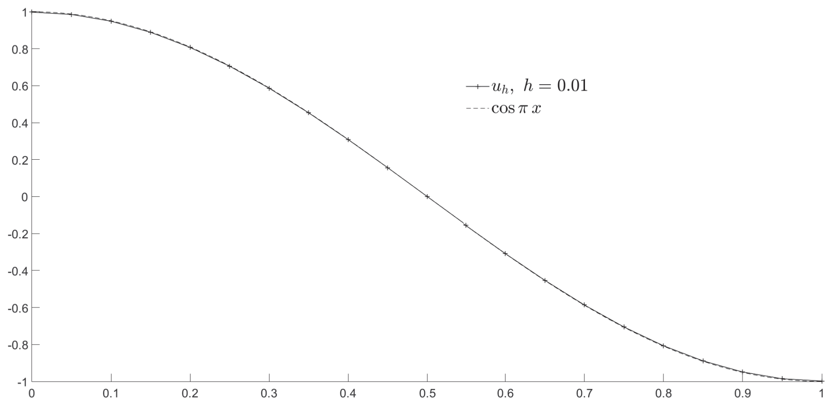
$$||u_h||_{L^2(\Omega)} = \left(\int_\Omega |u_h|^2 d\Omega\right)^{\frac{1}{2}} = \left(\sum_{i=1}^{N}\sum_{j=1}^{N} u_h(M_i)u_h(M_j)\int_\Omega w_i\, w_j\right)^{\frac{1}{2}} = (\mathbb{M}U|U)^{\frac{1}{2}}.$$

16) Écrire une fonction `void Helmholtz<MT,EFT>::exporte(const string& fn)` qui exporte dans un fichier les coordonnées des sommets du maillage et la solution obtenue en ces points ainsi que la numérotation des éléments en 2D seulement, sous la forme :

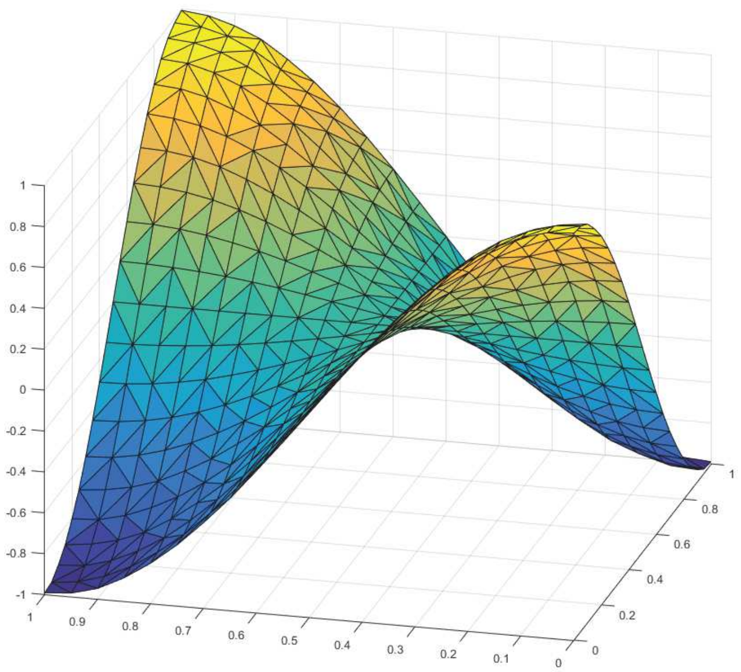
```
 en 1D
x1 sol1
x2 sol2
 ...
```

```
   en 2D
x1 y1 sol1
x2 y2 sol2
   ...
i1 j1 k1
i2 j2 k2
   ...
```

On peut aisément relire ces fichiers avec Matlab et afficher les solutions sous forme de courbe (1D) avec `plot` ou de surface (2D) avec `trisurf` :




solution du problème de Helmholtz 1D



solution du problème de Helmholtz 2D