TP6 : Résolution de l'équation de la chaleur (différences finies)

Compilation à l'aide d'un makefile

Nous allons nous intéresser à la compilation. Récupérer en suivant ce lien la version initiale du code de ce TP. La commande de compilation (détaillée ci-dessous) est :

```
g++ -std=c++11 -I Eigen/Eigen/ -o run Laplacian.cpp DataFile.cpp
TimeScheme.cpp Function.cpp main.cc
```

- g++ le nom du compilateur (ici *gcc* qui est *g++* en C++),
- -std=c++11 une option qui permet de préciser que l'on fait du C++11,
- -I Eigen/Eigen/ inclut (avec -I) le répertoire où sont les fichiers `Eigen`,
- -o run le nom de l'exécutable qui sera créé,
- Laplacian.cpp TimeScheme.cpp Function.cpp DataFile.cpp main.cc les fichiers à compiler.

Rappel : pour utiliser `Lapack`, il faut aussi ajouter la commande suivante :

- -llapack pour faire une édition de lien vers la librairie `Lapack` qui est déjà compilée.

Pour simplifier la compilation (qui se complexifie avec le nombre de fichiers et de librairies), ouvrir le fichier nommé par convention *Makefile* (sans extension) contenant les lignes suivantes :

```
# Compilateur utilisé
CC=g++

 # Options en mode optimisé - La variable NDEBUG est définie
 OPTIM_FLAG = -O3 -DNDEBUG -I Eigen/Eigen -std=c++11 -Wall
 # Options en mode debug - La variable NDEBUG n'est pas définie
 DEBUG_FLAG = -g -I Eigen/Eigen -std=c++11 -Wall
# On choisit comment on compile
CXX_FLAGS = $(OPTIM_FLAG)

# Le nom de l'exécutable
PROG = laplacian

# Les fichiers source à compiler
SRC = main.cc TimeScheme.cpp Laplacian.cpp Function.cpp DataFile.cpp

# La commande complète : compile seulement si un fichier a été modifié
$(PROG) : $(SRC)
	$(CC) $(SRC) $(CXX_FLAGS) -o $(PROG)
# Évite de devoir connaitre le nom de l'exécutable
all : $(PROG)

# Supprime l'exécutable, les fichiers binaires (.o) et les fichiers
# temporaires de sauvegarde  (~)
clean :
	rm -f *.o *~ $(PROG)
```

1. Deux modes de compilation sont proposés :

- Le mode *debug* sert à déboguer. Il intègre différentes fonctions qui permettent au compilateur d'avancer pas à pas. Des tests avec des messages d'erreurs peuvent être effectués.
- Le mode *release* est pour l'utilisateur. Les tests du mode *debug* ne sont plus nécessaires puisque le développeur a déjà vérifié que tout était correct. L'exécutable est plus léger et plus rapide.

Le mode se choisit sur la ligne 9 : pour le moment en mode *release*. Compiler le dans ce mode en tapant dans votre terminal :

```
make all
```

Exécuter votre code avec :

```
./laplacian data.txt
```

Vous avez un pas de temps et un pas d'espace qui sont petits et pourtant l'erreur reste importante. Si vous tracez la solution vous verrez qu'il y a un problème.

2. En effet une erreur (grossière) a été glissée dans le code. Compiler le en mode *debug* en modifiant la ligne 9. Comme votre code n'a pas été modifié et que le *Makefile* a été construit pour compiler seulement en cas de modification des fichiers sources, vous obtenez :

```
make: Nothing to be done for ‘all‘.
```

Pour lui faire oublier la précédente compilation il faut faire un :

```
make clean
```

Compiler et exécuter. En mode *debug* vous obtenez une erreur. En effet, plusieurs tests (vérifiant les tailles des matrices et des vecteurs) ont été ajoutés dans le code sous cette forme :

```
 // La macro "assert" ne vérifie que si NDEBUG n'a pas été defini (mode debug)
#ifdef DEBUG
   // Si A n'est pas une matrice carrée de la même taille que le vecteur u
   // Un message d'erreur est renvoyé avec la macro assert
   assert((A.cols() == A.rows()) && (A.cols() == u.size())
             && "Problème de taille dans la fonction DirectSolver.");
#endif //_DEBUG
```

À l'aide du message d'erreur que vous obtenez, vous pouvez trouver l'erreur et la corriger. Compiler de nouveau (en mode *debug*) et exécuter. Cette fois-ci votre code vous donne un résultat qui semble correct.
Rq : *Assert* est une macro (et non une fonction), c'est donc le preprocesseur qui la remplace par ce qu'il faut. Si *NDEBUG* a été défini, cette macro est remplacé par rien, donc il n'y a pas de "surcout" a mettre un assert et il n'est pas nécessaire de le protéger.
Rq : Il est possible d'utiliser *gdb* pour trouver l'erreur plus facilement. Pour cela il faut taper :

```
// Dans le terminal
gdb ./laplacian
// gdb s'ouvre. Pour faire tourner gdb :
run data.txt
// pour remonter jusqu'à  l'erreur, faire successivement :
up
```

Bilan :

- L'utilisation d'un *Makefile* et des commandes *make* simplifie la phase de compilation.
- L'utilisation régulière de la fonction *assert* en mode *debug* est recommandée et permet de faciliter le déboggage.
- Un code doit toujours être compilé en mode *debug* pendant la phase d'implémentation.
- Le mode *release* doit être choisi quand le code a été validé afin de gagner du temps.

1 - Prise en main du code

Le code que vous venez de compiler et d'exécuter permet de résoudre l'équation de la chaleur en 1D avec conditions aux bord de Dirichlet homogène :

$$\left\{\begin{array}{rcll} \partial_t u(t,x) - \sigma \partial_{xx} u(t,x) & = & f(t,x), & \forall x \in]x_{min}, x_{max}[,\ \forall t \in [t_0, T_{final}], \\ u(t,x_{min}) = u(t,x_{max}) & = & 0, & \forall t \in [t_0, T_{final}], \\ u(0,x) & = & u_0(x), & \forall x \in]x_{min}, x_{max}[, \end{array}\right.$$

avec $\sigma \in \mathbb{R}^{+*}$ le coefficient de diffusion. Ce code s'appuie sur le TP 4 pour la matrice du laplacien et sur le TP 5 pour la résolution en temps du système. La classe **TimeScheme** est couplée à la classe **Laplacian** (au lieu de la classe *OdeSystem* dans le TP 5).

Afin de valider le code une solution exacte a été considérée :

$$u_{ex}(t,x) = \sin(x - x_{max}) \sin(x - x_{min}) e^{-t}.$$

Toutes les fonctions nécessaires :

- condition initiale,
- terme source
- et solution exacte si elle est connue,

sont implémentées dans la classe **Function**.

Nous utilisons la discrétisation de l'opérateur laplacien présentée dans le TP 4 et un schéma d'Euler explicite. On discrétise l'intervalle $[x_{min}, x_{max}]$ et l'intervalle $[t_0, T_{final}]$ par

$$(x_i)_{i=0,N_x+1} \text{ avec } x_i = x_{min} + i h_x \text{ et } h_x = \frac{x_{max} - x_{min}}{N_x + 1},$$
$$(t_n)_{n=0,N_{it}} \text{ avec } t_n = t_0 + n\,\Delta t \text{ et } \Delta t = \frac{T_{final} - t_0}{N_{it}}.$$

Soient u_i^n une approximation de $u(t_n, x_i)$, $f_i^n = f(t_n, x_i)$ et $u_{0,i} = u_0(x_i)$, on considère le schéma :

$$\left\{\begin{array}{ll} \dfrac{u_i^{n+1} - u_i^n}{\Delta t} + \sigma \dfrac{2u_i^n - u_{i+1}^n - u_{i-1}^n}{h_x^2} = f_i^n, & i = 1 \dots N_x,\ n = 0 \dots N_{it} - 1, \\ u_0^n = u_{N_x+1}^n = 0 & n = 1 \dots N_{it}, \\ u_i^0 = u_{0,i} & i = 0 \dots N_x + 1 \end{array}\right.$$

On définit H la matrice du laplacien, U^n les vecteurs solutions et F^n les vecteurs sources pour $n = 0, \dots, N_{it}$ par :

$$H = \frac{1}{h_x^2} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \quad U^n = \begin{pmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_N^n \end{pmatrix} \quad F^n = \begin{pmatrix} f_1^n \\ f_2^n \\ \vdots \\ f_N^n \end{pmatrix},$$

la discrétisation devient :

$$U^{n+1} = U^n + \Delta t(-\sigma\, H\, U^n + F^n).$$

La résolution de ce système nécessite de nombreux paramètres :

```
x_min x_max hx
sigma
t0 tfinal dt
scheme // choix du schéma en temps
results // nom du dossier solution
```

Pour ne pas avoir à compiler de nouveau à chaque fois que l'on modifie un paramètre, un fichier de données (ici *data.txt*) est lu par la classe **DataFile**. C'est pourquoi il faut taper :

```
./laplacian data.txt
```

Pour qu'un paramètre soit lu il faut bien mettre sa valeur sur la ligne en dessous du nom du paramètre. Bien sur il est possible d'adapter la classe **DataFile** si on souhaite utiliser un autre style de fichier de données.

1. **Regarder attentivement la structure du code et les fonctionnalités contenues.**
Rappel : Le code est constitué de 4 classes :

- **TimeScheme :** pour le schéma en temps,
- **Laplacian :** pour construire la matrice,
- **Function :** pour toutes les fonctions,
- **DataFile :** pour les paramètres.

Important : Partir du fichier *main.cc* et faire un schéma du code (liens entre les différentes classes).

2. Afficher avec `Gnuplot` la solution obtenue.

3. Que se passe t-il si un des paramètres n'est pas donné dans le fichier de données ? Il faut retirer les deux lignes : le nom du paramètre et sa valeur.

4. La résolution est lente. Que se passe t-il si vous augmentez le pas de temps ?

5. Pour pouvoir utiliser le schéma d'Euler explicite, une condition CFL doit être vérifiée. Pour la déterminer, il faut repartir du schéma. Vérifier qu'il se réécrit :

$$u_i^{n+1} = \Big(1 - \frac{2\Delta t\sigma}{h_x^2}\Big)u_i^n + \frac{\Delta t\sigma}{h_x^2}u_{i+1}^n + \frac{\Delta t\sigma}{h_x^2}u_{i-1}^n + \Delta t f_i^n.$$

La condition CFL s'écrit donc :

$$\frac{2\Delta t\sigma}{h_x^2} < 1.$$

Redéfinir le pas de temps afin que la condition CFL soit toujours vérifiée **pour le schéma d'Euler Explicite** à la fin de la fonction *ReadDataFile* du fichier *DataFile.cpp* juste avant que le nombre d'itérations soit calculé. Informer l'utilisateur que le pas de temps est fixé avec la condition CFL.

2 - Résolution à l'aide d'un schéma implicite

Un schéma implicite permet de diminuer fortement les temps de calcul.

1. Implémenter le schéma implicite :

$$U^{n+1} = (Id + \Delta t\,\sigma H)^{-1}(U^n + \Delta t F^n),$$

en créant une classe dérivée de la classe *TimeScheme* qui sera nommée *ImplicitEulerScheme*. Cette fois-ci, la résolution d'un système linéraire est nécessaire. Rappel : pour utiliser le solveur direct *SimplicialLLT* de *Eigen* il faut faire :

```
// Définir un solveur
SimplicialLLT<SparseMatrix<double> > solver_direct;
// Étape de factorisation (à  faire qu'une seule fois !)
solver_direct.compute(A); // où A est une matrice creuse SparseMatrix<double>
// Résolution (que l'on fait pour chaque vecteur u)
VectorXd sol = solver_direct.solve(u); // où u est vecteur dense VectorXd
```

La difficulté ici réside dans le fait qu'il ne faut pas factoriser la matrice à chaque itération. Cette étape doit être faite qu'une seule fois dans la fonction *Initialize*. Ce qui implique qu'il faut définir :

- une variable privée dans la classe dérivée *ImplicitEulerScheme*

```
Eigen::SimplicialLLT<Eigen::SparseMatrix<double> > _solver_direct;
```

- la fonction *Initialize* doit être virtuelle (et non virtuelle pure !) pour pouvoir ajouter l'étape de factorisation. Pour ne pas répéter le code qui se trouve dans *TimeScheme : :Initialize*, il est possible de l'appeler directement dans *ImplicitEulerScheme : :Initialize* :

```
void ImplicitEulerScheme::Initialize(DataFile* data_file, Laplacian* lap)
{
  TimeScheme::Initialize(data_file, lap);

  // Créer la matrice ImpMatrix = Id + delta * sigma * H
  // La factoriser avec _solver_direct.compute(ImpMatrix)
}
```

2. L'utilisateur doit pouvoir indiquer le schéma souhaité dans le fichier de données. Le schéma par défaut sera le schéma implicite. Pour trouver ce qu'il faut modifier faire une recherche dans *DataFile.cpp* du mot *EulerExplicite*.

3. Que pensez-vous du schéma implicite dans ce cas ? Si votre résolution de système est trop lente, le schéma implicite perd beaucoup de son intérêt. Qu'en est-il ici ?

3 - Différentes conditions limites

L'objectif de cette section est de proposer d'autres conditions limites pour ce problème. Nous rappelons les conditions aux limites **Dirichlet non homogène** :

$$u(t, x_{min}) = \delta_L(t) \text{ et } u(t, x_{max}) = \delta_R(t) \ \forall t \in [t_0, T_{final}].$$

et les conditions aux limites **Neumann non homogène** :

$$\partial_x u(t, x_{min}) = \eta_L(t) \text{ et } \partial_x u(t, x_{max}) = \eta_R(t) \ \forall t \in [t_0, T_{final}].$$

Les conditions de Dirichlet homogène considérées précédemment correspondent à $\delta_L \equiv 0,\ \delta_R \equiv 0$. Nous voulons pouvoir proposer les 4 cas suivants :

- Dirichlet à droite, Dirichlet à gauche
- Neumann à droite, Dirichlet à gauche

- Dirichlet à droite, Neumann à gauche
- Neumann à droite, Neumann à gauche

1. a) Pour des conditions de Dirichlet à droite et de Neumann à gauche, le fichier de données sera :

```
# Boundary conditions
LeftBoundCond
Dirichlet
RightBoundCond
Neumann
```

Construire deux variables privées *string* que l'on notera $\_LBC$ et $\_RBC$ dans la classe *DataFile* ainsi que deux booléens $\_if\_LBC$ et $\_if\_RBC$. Ils sont remplis comme les autres variables en lisant le fichier de données. Suivre l'exemple de la variable $\_results$.
b) Initialiser les variables par défaut pour des conditions de Dirichlet à droite et à gauche.
c) Ne pas oublier de faire une fonction Get_LBC_x (resp. Get_RBC_x) pour accéder à $\_LBC$ (resp. $\_RBC$) dans le fichier *DataFile.h*.

2. Construire les 4 fonctions suivantes dans la classe *Fonction* :

```
double DirichletLeftBC(const double t) const;
double DirichletRightBC(const double t) const;
double NeumannLeftBC(const double t) const;
double NeumannRightBC(const double t) const;
```

Pour le moment afin de conserver la solution exacte proposée précédemment, implémenter ces fonctions comme identiquement nulles.

3. À présent, vous êtes prêts à implémenter les conditions non homogènes de Dirichlet et de Neumann. Quelques conseils sont donnés ci-dessous.
a) **Dirichlet non homogène :** La matrice n'est pas modifiée, seulement le terme source. Pour trouver le terme à ajouter, écrire le schéma suivant :

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}+\sigma\,\frac{2u_i^n-u_{i+1}^n-u_{i-1}^n}{h_x^2}=f_i^n$$

sur une feuille de papier pour $i=1$ et pour $i=N_x$.
b) **Neumann non homogène :** La matrice et le vecteur source doivent être modifiés. Les conditions de Neumann à droite et à gauche sont respectivement :

$$\frac{u_1^n-u_0^n}{h_x}=\eta_L^n \quad \text{et} \quad \frac{u_{N_x+1}^n-u_{N_x}^n}{h_x}=\eta_R^n,$$

ce qui peut se réécrire sous la forme suivante :

$$u_0^n=u_1^n-h_x\,\eta_L^n \quad \text{et} \quad u_{Nx+1}^n=u_{N_x}^n+h_x\,\eta_R^n.$$

Écrire le schéma suivant :

$$\frac{u_i^{n+1}-u_i^n}{\Delta t}+\sigma\,\frac{2u_i^n-u_{i+1}^n-u_{i-1}^n}{h_x^2}=f_i^n$$

pour $i=1$ et pour $i=N_x$ **sur une feuille de papier** dans le cas des conditions de Neumann pour trouver les modifications à effectuer.

4. Valider vos résultats avec la solution exacte suivante :

$$u(t,x) = e^{-t}\sin\Big(4\pi(x - x_{max}) + \frac{\pi}{2}\Big)\Big(\frac{x_{max} + x_{min}}{2} - x\Big).$$

Pour ne pas perdre de temps, voici les quelques lignes de code des fonctions dont vous avez besoin :

```
SourceFunction
 -sin(M_PI*(4.*(x-_xmax)+0.5))*exp(-t)*((_xmax+_xmin)/2.-x)
 +_sigma*(8.*M_PI*cos(M_PI*(4.*(x-_xmax)+0.5))*exp(-t)
 +16.*pow(M_PI,2)*sin(M_PI*(4.*(x-_xmax)+0.5))*exp(-t)*((_xmax+_xmin)/2.-x));

ExactSolution
 sin(M_PI*(4*(x-_xmax)+0.5))*exp(-t)*((_xmax+_xmin)/2-x);

DerivativeOfExactSolution
-sin(M_PI*(4*(x-_xmax)+0.5))*exp(-t)
    +4.*M_PI*cos(M_PI*(4*(x-_xmax)+0.5))*exp(-t)*((_xmax+_xmin)/2 - x);
```

Avec le schéma **Euler Implicite** et les paramètres suivants :

$$x_{min} = 0,\ x_{max} = 0.5,\ h_x = 10^{-5},\ \sigma = 1.2,\ t_0 = 0,\ T_{final} = 0.1,\ \Delta t = 0.01$$

l'erreur est de l'ordre de 1.38×10^{-6} avec des conditions de Dirichlet à gauche et à droite et de l'ordre de 4.95×10^{-5} avec des conditions de Dirichlet à droite et de Neumann à gauche. Pourquoi l'erreur avec des conditions de Neumann est-elle plus importante ?

Cas 2D

Nous souhaitons à présent pouvoir résoudre l'équation de la chaleur en 2D. Copier le code dans un nouveau dossier afin de conserver le code en 1D. On discrétise cette fois-ci en 2D

$$(x_i)_{i=0,N_x+1} \text{ avec } x_i = x_{min} + ih_x \text{ et } h_x = \frac{x_{max} - x_{min}}{N_x + 1},$$
$$(y_j)_{j=0,N_y+1} \text{ avec } y_j = y_{min} + jh_y \text{ et } h_y = \frac{y_{max} - y_{min}}{N_y + 1}.$$

Soient $u_{i,j}^n$ une approximation de $u(t_n, x_i, y_j)$, $f_{i,j}^n = f(t_n, x_i, y_j)$ et $u_{0,i,j} = u_0(x_i, y_j)$, on considère le schéma (ici implicite mais la version explicite est possible aussi) :

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + \sigma \frac{2u_{i,j}^{n+1} - u_{i+1,j}^{n+1} - u_{i-1,j}^{n+1}}{h_x^2} \quad + \sigma \frac{2u_{i,j}^{n+1} - u_{i,j+1}^{n+1} - u_{i,j-1}^{n+1}}{h_y^2} = f_{i,j}^n,$$

On définit H la matrice du laplacien 2D par :

$$H = \begin{pmatrix} \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & \cdots & -\frac{1}{h_y^2} & \cdots & \cdots & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & \cdots & -\frac{1}{h_y^2} & \cdots & \cdots \\ & \ddots & \ddots & \ddots & \cdots \ddots \cdots & & \\ & & \ddots & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{Id_{N_x}}{h_y^2} & & \\ & & & -\frac{Id_{N_x}}{h_y^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & & \end{pmatrix}$$

puis U^n les vecteurs solution et F^n les vecteurs sources pour tout $n = 0, \ldots, N_{it}$:

$$U^n = \begin{pmatrix} u_{1,1}^n \\ u_{2,1}^n \\ \vdots \\ u_{N_x,1}^n \\ u_{1,2}^n \\ u_{2,2}^n \\ \vdots \\ u_{N_x,2}^n \\ \vdots \\ u_{1,N_y}^n \\ u_{2,N_y}^n \\ \vdots \\ u_{N_x,N_y}^n \end{pmatrix} \quad F^n = \begin{pmatrix} f_{1,1}^n \\ f_{2,1}^n \\ \vdots \\ f_{N_x,1}^n \\ f_{1,2}^n \\ f_{2,2}^n \\ \vdots \\ f_{N_x,2}^n \\ \vdots \\ f_{1,N_y}^n \\ f_{2,N_y}^n \\ \vdots \\ f_{N_x,N_y}^n \end{pmatrix}.$$

Le problème se réécrit alors :

$$U^{n+1} = U^n + \Delta t(-\sigma\, H\, U^n + F^n).$$

1 - Implémentation

Implémenter le cas 2D à l'aide des remarques et conseils suivants.

1. **Classe DataFile :**

- Pour prendre en compte les nouveaux paramètres, définir les variables privées suivantes : $\_hy$, $\_Ny$, $\_y\_min$, $\_y\_max$, $\_DBC$ (Down BC), $\_UBC$ (Up BC),
  les booléens : $\_if\_hy$, $\_if\_y\_min$, $\_if\_y\_max$, $\_if\_DBC$, $\_if\_UBC$
  les fonctions : Get_hy, Get_Ny, Get_y_min, Get_y_max, Get_DBC, Get_UBC.
- Construire $\_Ny$ à la fin du fichier *DataFile.cpp* et adapter $\_hy$.
- Pour le schéma explicite, ne pas oublier de changer la condition CFL (écrire sur un papier la CFL 2D).

2. **Classe Function :**

- Modifier les fonctions *InitialCondition*, *SourceFunction* et *ExactSolution* pour ajouter les dépendances en y.
- Modifier les fonctions des conditions aux limites à gauche et à droite pour ajouter les dépendances en y.
- Ajouter les fonctions pour les conditions aux limites en haut et en bas. Ne pas oublier leur dépendance dans la variable x.

3. **Classe Laplacian :**

- Faire des sorties *.vtk* pour afficher la solution dans `Paraview` :

```
ofstream solution;
solution.open(name_file, ios::out);
solution.precision(7);
solution << "# vtk DataFile Version 3.0" << endl;
solution << "sol" << endl;
solution << "ASCII" << endl;
solution << "DATASET STRUCTURED_POINTS" << endl;
solution << "DIMENSIONS " << _Nx << " " << _Ny << " " << 1 << endl;
solution << "ORIGIN " << _xmin << " " << _ymin << " " << 0 << endl;
solution << "SPACING " << _hx << " " << _hy << " " << 1 << endl;;
solution << "POINT_DATA " << _Nx*_Ny << endl;
solution << "SCALARS sol float" << endl;
solution << "LOOKUP_TABLE default" << endl;
for(int j=0; j<_Ny; ++j)
{
  for(int i=0; i<_Nx; ++i)
  {
    solution << float(sol(i+j*_Nx)) << " ";
  }
  solution << endl;
}
solution.close();
```

- Construire la matrice et le vecteur source en 2D. Ne pas oublier de mettre les conditions aux limites.

4. **Function Main :** Ne pas oublier d'adapter l'affichage pour vérifier la discrétisation en y.

5. Pour valider votre code, nous vous proposons d'utiliser la solution exacte suivante :

$$u(t,x) = \frac{1}{t+1}\cos(3\pi x)(y - \frac{y_{min}+y_{max}}{4})$$

```
SourceFunction
-cos(3*M_PI*x)/pow(t+1.,2)*(y-(_y_min+_y_max)/4)
    +9*_sigma*pow(M_PI,2)*cos(3*M_PI*x)*(y-(_y_max+_y_min)/4)/(t+1.);

ExactSolution
cos(3*M_PI*x)*(y-(_y_max+_y_min)/4.)/(t+1.);

PartialDerivativeOfExactSolutionX
-3*M_PI*sin(3*M_PI*x)*(y-(_y_min+_y_max)/4)/(t+1.0);

PartialDerivativeOfExactSolutionY
cos(3*M_PI*x)/(t+1.0);
```

Avec le schéma implicite et les paramètres suivants :

$$x_{min} = y_{min} = 0,\ x_{max} = 1,\ y_{max} = 0.5,\ h_x = h_y = 0.005,\ \sigma = 1.2,$$

$$t_0 = 0,\ T_{final} = 0.1,\ \Delta t = 0.01,$$

l'erreur est de l'ordre de 3.60×10^{-5} avec des conditions de Dirichlet partout et de l'ordre de 6.27×10^{-3} avec Dirichlet à gauche et en haut et de Neumann à droite et en bas.

6. Que pensez-vous du schéma explicite pour le cas 2D ?

7. Comment pourrait-on vérifier l'ordre du schéma en temps et celui du schéma en espace ?

2 - Mise en situation réaliste

Objectif : Utiliser le code que vous venez de développer sur un exemple réaliste.

Nous allons considérer un barreau de fer – infinie dans la direction z (d'où l'étude en 2D). Un élément chauffant est posé sur la plaque. Il chauffe de façon périodique en temps et non homogène en x. La plaque est isolée à gauche et à droite et est refroidie par le dessous en imposant une température T_{ref}. Au temps initial, la plaque est à 20 degrés. La figure suivante présente la situation.

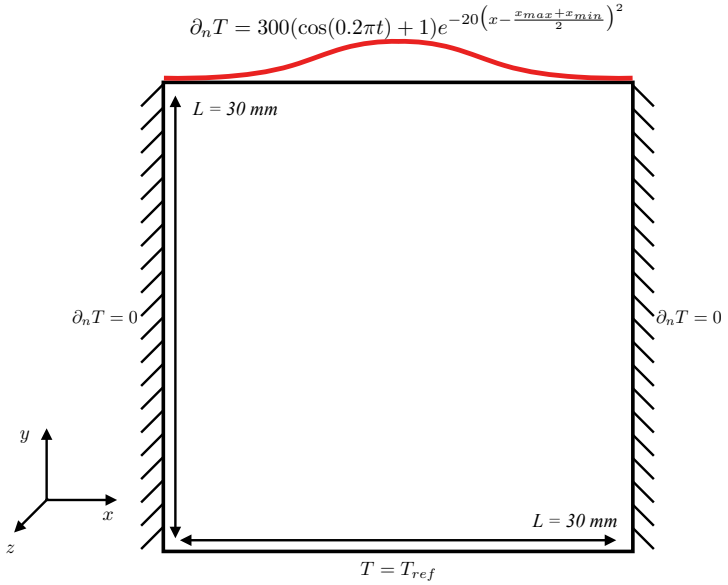

FIGURE 1 – Une plaque de fer sur laquelle est posé un élément chauffant.

Nous rappelons que

$$\sigma = \frac{k}{\rho\, c}$$

où k est la conductivité thermique, ρ la masse volumique et c la capacité massique du matériau. Le fer a une diffusivité thermique de $22.8\,\mathrm{mm}^2.\mathrm{s}^{-1}$. L'objectif est de contrôler la température de la plaque de fer. Nous ne souhaitons pas qu'elle chauffe trop.

1. Mettre en place la simulation numérique. Afficher la solution avec `Paraview`.

2. Faire une sortie de la température au centre de la plaque. Elle nous servira de référence pour voir si la plaque chauffe trop fortement.

3. Quel est l'ordre de grandeur de la température maximale T_{ref} que l'on doit imposer pour avoir une température au centre de la plaque ne dépassant pas les 70 degrés ?