TP4 : 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 la librairie d'algèbre linéaire `Lapack` (très utile pour inverser des systèmes de manière très efficace !), 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 = $(DEBUG_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 ceux temporaires  (~)
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 *debug*.

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.

Équation de la chaleur

Nous allons résoudre l'équation de la chaleur en 2D avec des conditions aux bords de type Dirichlet homogène :

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

avec $\sigma \in \mathbb{R}^{+*}$ le coefficient de diffusion.

On discrétise les intervalles $[x_{min}, x_{max}]$ et $[y_{min}, y_{max}]$ et l'intervalle $[t_0, T_{final}]$ par

$$(x_i)_{i=0,N_x+1} \text{ avec } x_i = x_{min} + i h_x \text{ et } h_x = \frac{x_{max} - x_{min}}{N_x + 1},$$

$$(y_j)_{j=0,N_y+1} \text{ avec } y_j = y_{min} + j h_y \text{ et } h_y = \frac{y_{max} - y_{min}}{N_y + 1},$$

$$(t_n)_{n=0,N_{it}} \text{ avec } t_n = t_0 + n\,\Delta t \text{ et } \Delta t = \frac{T_{final} - t_0}{N_{it}}.$$

Soient $u_{i,j}^n$ une approximation de $u(t_n, x_i, y_j)$, $f_{i,j}^n = f(t_n, x_i, y_j)$, on considère le schéma, pour tout $i = 1 \dots N_x,\ j = 1 \dots N_y,\ n = 0 \dots N_{it} - 1$:

$$\left\{\begin{array}{c} \dfrac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + \sigma \dfrac{2u_{i,j}^n - u_{i+1,j}^n - u_{i-1,j}^n}{h_x^2} + \sigma \dfrac{2u_{i,j}^n - u_{i,j+1}^n - u_{i,j-1}^n}{h_y^2} = f_{i,j}^n, \\ u_{0,j}^n = u_{N_x+1,j}^n = u_{i,0}^n = u_{i,N_x+1}^n = 0. \end{array}\right.$$

On note H la matrice du Laplacien :

$$H = \begin{pmatrix} \mathbb{B} & \mathbb{T} & & & \\ \mathbb{T} & \mathbb{B} & \mathbb{T} & & \\ & \ddots & \ddots & \ddots & \\ & & \mathbb{T} & \mathbb{B} & \mathbb{T} \\ & & & \mathbb{T} & \mathbb{B} \end{pmatrix} \text{ avec } \mathbb{B} = \begin{pmatrix} \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & & & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} \\ & & & -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} \end{pmatrix},$$

et U^n le vecteur solution et F^n le vecteur source :

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

la discrétisation devient :

$$U^{n+1} = U^n + \Delta t\,(-\sigma\, H\, U^n + F^n) \text{et en implicite : } (Id + \Delta t \sigma H) U^{n+1} = U^n + \Delta t F^{n+1}.$$

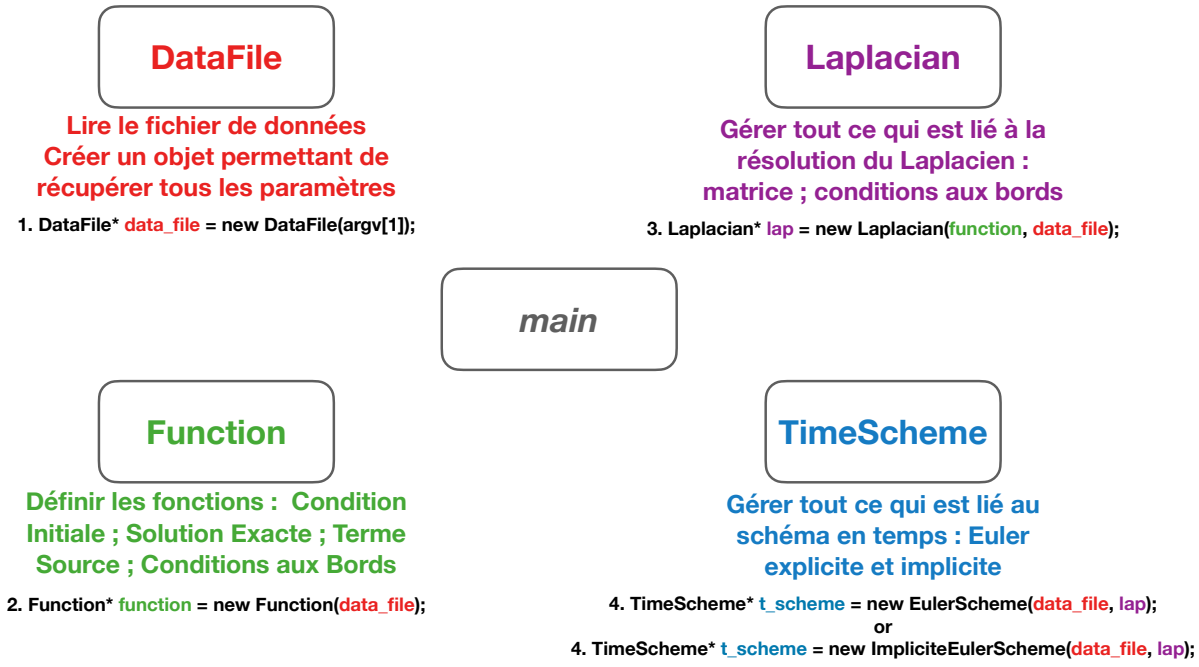

Figure 1 – Schéma du code

Le code est structuré en 4 classes :

- La classe *DataFile* permet de lire un fichier de données (contenant tous les paramètres) et de les placer dans un objet.
- La classe *Function* permet de donner tous les fonctions essentielles aux bon fonctionnement du code : la condition initiale ; le terme source ; la solution exacte (si elle existe) et les conditions aux bords (dans le cas où on ne considère pas seulement du Dirichlet homogène).
- La classe *Laplacian* permet de construire la matrice du Laplacien ; le terme source et de gérer la sauvegarde de la solution.
- La classe *TimeScheme* permet de construire tout ce qui est lié au schéma en temps. Deux types de schéma devront être implémentés : schéma d'Euler Explicite - *ExplicitEuler* et schéma d'Euler Implicite - *ImplicitEuler*.

La figure 1 résume cette décomposition. Les définitions des différents objets de ces classes sont explicitées permettant de mettre en avant les relations entre les classes.

Documentation de la classe DataFile

La résolution de ce système nécessite de nombreux paramètres :

```
x_min x_max hx y_min y_max hy
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 *dataSmallCase.toml*) est lu par la classe **DataFile** à travers la librairie `Toml` (voir https ://github.com/toml-lang/toml). C'est pourquoi il faut taper :

```
./laplacian dataSmallCase.toml
```

La librairie `Toml` impose la structure de fichier de données.

1. Construction d'un objet de la classe *DataFile* et lecture du fichier :

```
DataFile* data_file = new DataFile("params.toml");
```

2. Récupération des paramètres :

Nom de la Fonction	Type renvoyé	Variable
Get_xmin()	double	x_{min}
Get_xmax()	double	x_{max}
Get_hx()	double	h_x
Get_Nx()	int	N_x
Get_ymin()	double	y_{min}
Get_ymax()	double	y_{max}
Get_hy()	double	h_y
Get_Ny()	int	N_y
Get_sigma()	double	σ
Get_t0()	double	t_0
Get_tfinal()	double	T_{final}
Get_dt()	double	Δt
Get_scheme()	string	Schémas : ExplicitEuler ; ImplicitEuler
Get_results()	string	Nom de dossier *résultats*
Get_LBC_x()	string	Type de conditions aux bords : Dirichlet (ou Neumann)
Get_RBC_x()	string	Type de conditions aux bords : Dirichlet (ou Neumann)
Get_DBC_y()	string	Type de conditions aux bords : Dirichlet (ou Neumann)
Get_UBC_y()	string	Type de conditions aux bords : Dirichlet (ou Neumann)

avec un exemple d'utilisation :

```
double xmin = data_file->Get_xmin();
```

Documentation de la classe Function

D'autres informations ont besoin d'être données pour pouvoir résoudre le problème :

- la condition initiale $(x, y) \mapsto u_0(x, y)$,
- le terme source $(t, x, y) \mapsto f(t, x, y)$,
- la solution exacte $(t, x, y) \mapsto u_{exacte}(t, x, y)$ si elle existe,
- les conditions aux bords (dans le cas où on ne considère pas que du Dirichlet Homogène).

Toutes ces informations sont données dans la classe *Function*. Pour pouvoir valider le code que vous avez créé, nous allons construire une solution exacte. Il nous faut une fonction qui s'annule aux 4 bords quand on considère des conditions de Dirichlet homogènes. Elle ne faut pas qu'elle soit trop simple (en particulier pour ne pas tomber sur un cas où la discrétisation est exacte) :

$$u(t, x, y) = (y - y_{min})(x - x_{min}) \sin(y - y_{max}) \sin(2\pi(x - x_{max}))(t + 1)^2,$$

ce qui veut dire que la condition initiale vaut : $u_0(x, y) = u(0, x, y)$. Pour calculer le terme source, nous allons tout simplement calculer : $f = \partial_t u - \sigma \Delta u$. La fonction u proposée vérifie donc bien : l'équation de la chaleur ; la solution initiale et les conditions aux bords.

Vérifier que toutes ces fonctions ont été implémentées dans la classe *Function*.

Remarque : Le contenu du fichier *Function.cpp* est identique au contenu du fichier *FunctionDirichlet.cpp* (c'est une sauvegarde). Il correspond à un exemple où la solution exacte vaut 0 à chaque bord (conditions de Dirichlet homogènes). Un autre fichier pour des conditions aux bords de type Dirichlet ou Neumann non homogènes est donné dans le fichier *FunctionBC.cpp*. Il suffira de copier son contenu dans le fichier *Function.cpp* pour l'utiliser.

Quelques explications sur la fonction main

Tout d'abord la lecture du nom de fichier qui contient les paramètres :

```
if (argc < 2)
{
  cout << "Please, enter the name of your data file." << endl; abort();
}
DataFile* data_file = new DataFile(argv[1]);
```

Concernant le schéma en temps, un *switch* est proposé pour gérer le choix du schéma en temps :

```
TimeScheme* time_scheme = NULL;
int int_euler_scheme(1);
if (data_file->Get_scheme() == "ImplicitEuler") {int_euler_scheme = 2;}
switch(int_euler_scheme)
{
  case 1:
    time_scheme = new EulerScheme(data_file, lap);
    break;
  case 2:
    time_scheme = new ImplicitEulerScheme(data_file, lap);
    break;
}
```

Enfin, il ne faut pas oublier de libérer la mémoire en supprimant tous les pointeurs créés :

```
  delete data_file; delete function; delete lap; delete time_scheme;
```

Implémentation du Code

Vous devez maintenant implémenter les deux classes *TimeScheme* et *Laplacian*. Quelques règles ou astuces :

- Faire un code **propre** avec des fonctions pour les différentes actions. Par exemple :

```
Laplacian::InitialCondition()
Laplacian::BuildLaplacianMatrix()
Laplacian::BuildSourceTerm(double t)
TimeScheme::SaveSol(Eigen::VectorXd sol, std::string name_sol, int n)
EulerScheme::Integrate()
ImplicitEulerScheme::Integrate()
```

- Une comparaison entre les schémas d'Euler Explicite et Implicite doit être faite. 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 :

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

  La condition CFL s'écrit donc :

$$\Delta t < \frac{h_x^2h_y^2}{2\sigma(h_x^2+h_y^2)}.$$

  Le pas de temps pour le schéma d'Euler Explicite est défini afin que la condition CFL soit toujours vérifiée (voir ligne 265 du fichier *DataFile.cpp*). Montrer que cela implique des temps de calcul prohibitifs (avec des valeurs grandes de σ par exemple). (Attention à stopper le calcul si le pas de temps est vraiment trop petit pour ne pas surcharger votre espace de sauvegarde!!!).

- Pour le schéma d'Euler Implicite, il s'écrit :

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

  ce qui demande l'inversion d'un système linéaire. Concernant le *solveur*, `Eigen` en propose plusieurs dont une description est donnée sur cette page où il est expliqué comment choisir son solveur en fonction de la matrice : carrée ou rectangulaire, symétrique ou non, très creuse ou non, pattern irrégulier ou non etc ... Il est important de remarquer qu'il est aussi possible d'utiliser des solveurs extérieurs à `Eigen` en faisant appel à d'autres librairies (comme par exemple `Lapack`). Vous pouvez commencer par utiliser un solveur direct de type *Cholesky* (classe : SimplicialLLT) mais quelque soit le solveur, voici la marche à suivre pour résoudre un système sous la forme $Ax = b$ où A est une *SparseMatrix<double>* et b est un *VectorXd* :

```
//Adapter SolverClass en fonction du solveur
// (par exemple SolverClass = SimplicialLLT)
SolverClass <SparseMatrix<double> > solver;
solver.compute(A);
VectorXd x = solver.solve(b);
```

Remarque : l'étape qui est **coûteuse** en temps de calcul est celle qui consiste à faire la décomposition de Cholesky (ou une autre décomposition ou un préconditionnement en cas de solveur itératif). Cette étape est cachée dans : *solver.compute(A)*. Si la matrice ne change pas, il n'y a pas besoin de faire cette étape plusieurs fois !

- Pour construire la matrice vous pouvez utiliser l'objet *triplets* :

```
SparseMatrix<double> S(10,10);
vector<Triplet<double>> triplets;
for (int i=0; i<S.rows(); ++i)
{
  triplets.push_back({i,i,1.});
  if (i > 0)
  triplets.push_back({i,i-1,-2.});
  if (i < S.rows()-1)
  triplets.push_back({i,i+1,2.});
}
S.setFromTriplets(triplets.begin(), triplets.end());
```

- On va sauvegarder la solution dans des fichiers `Paraview` (comme c'est la première fois que vous créez des fichiers de ce style, vous pouvez utiliser la fonction suivante) :

```
void TimeScheme::SaveSol(VectorXd sol, string n_sol, int n)
{
  string n_file = _df->Get_results() + "/" + n_sol + to_string(n) + ".vtk";
  ofstream solution;
  solution.open(n_file, ios::out);
  int Nx(_df->Get_Nx()), Ny(_df->Get_Ny());
  double xmin(_df->Get_xmin()), ymin(_df->Get_ymin());
  double hx(_df->Get_hx()), hy(_df->Get_hy());

  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 << sol(i+j*Nx) << " ";
    }
    solution << endl;
  }
  solution.close();
}
```

- Pour valider votre code, ajouter les lignes suivantes dans le *main.cc* :

```
VectorXd exact_sol = lap->ExactSolution(df->Get_tfinal());
VectorXd approx_sol = time_scheme->GetSolution();
double error = ((approx_sol-exact_sol).array().abs()).maxCoeff();
cout << "Erreur = " << error << endl;
```

On obtient une erreur de 0.0062. La matrice du Laplacien est donnée par :

```
208 -100 -4 0 0 0
-100 208 0 -4 0 0
-4 0 208 -100 -4 0
0 -4 -100 208 0 -4
0 0 -4 0 208 -100
0 0 0 -4 -100 208
```

Et le terme source à t = 0 vaut :

```
0.481892
0.863303
 0.82104
 1.46643
0.709587
 1.26302
```

Ensuite, pour valider le code sur une grille plus raffinée : avec le fichier de données *data.toml* vous devez obtenir une erreur de 0.023.

- L'objectif est d'obtenir un code permettant de choisir pour chaque bord si on considère des conditions de Dirichlet et de Neumann non homogènes. Par exemple, pour des conditions de Dirichlet non homogènes au bord gauche c'est-à-dire $x = x_{min}$: $u(t, x_{min}, y) = g_L(t, y)$, ce qui se traduit par : $u^n_{0,j} = g^n_{L,j}$. Il faut donc regarder les modifications à effectuer dans la matrice et/ou dans le vecteur source pour prendre en compte cette condition aux bords en regardant ce qui se passe pour $i = 1$ dans :

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} + \sigma \frac{2u^n_{i,j} - u^n_{i+1,j} - u^n_{i-1,j}}{h_x^2} + \sigma \frac{2u^n_{i,j} - u^n_{i,j+1} - u^n_{i,j-1}}{h_y^2} = f^n_{i,j}.$$

La fonction $(t, y) \mapsto g_L(t, y)$ est donnée par la fonction qui est dans le fichier *Function.cpp* :

```
double Function::DirichletLeftBC(const double y, const double t) const
```

Pour des conditions de Neumann non homogènes au bord gauche (c'est-à-dire $x = x_{min}$) :

$$\partial_t\, u(t, x_{min}, y) = h_L(t, y),$$

ce qui peut s'approximer à l'ordre 1 par :

$$\frac{u^n_{1,j} - u^n_{0,j}}{h_x} = h^n_{L,j}.$$

Il faut donc regarder les modifications à effectuer dans la matrice et/ou dans le vecteur source pour prendre en compte cette condition aux bords en regardant ce qui se passe pour $i = 1$ dans l'équation ci-dessous. La fonction $(t, y) \mapsto h_L(t, y)$ est donnée par la fonction (qui est dans le fichier *Function.cpp*) :

```
double Function::NeumannLeftBC(const double y, const double t) const
```

Pour valider le code, avec des conditions aux bords non homogènes, il y a un fichier *FunctionBC.cpp* (il suffit de remplacer le contenu du fichier *Function.cpp* par le contenu de ce fichier) correspond à la fonction :

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

Par exemple avec les conditions aux bords données ci dessous :

```
[BC]
LeftBoundCond = "Neumann"
RightBoundCond = "Dirichlet"
DownBoundCond = "Dirichlet"
UpBoundCond = "Neumann"
```

vous devez obtenir une erreur d'environ 0.0052 avec le fichier *data.toml*. Bien sûr, vous pouvez tester d'autres combinaisons de conditions aux bords.

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.

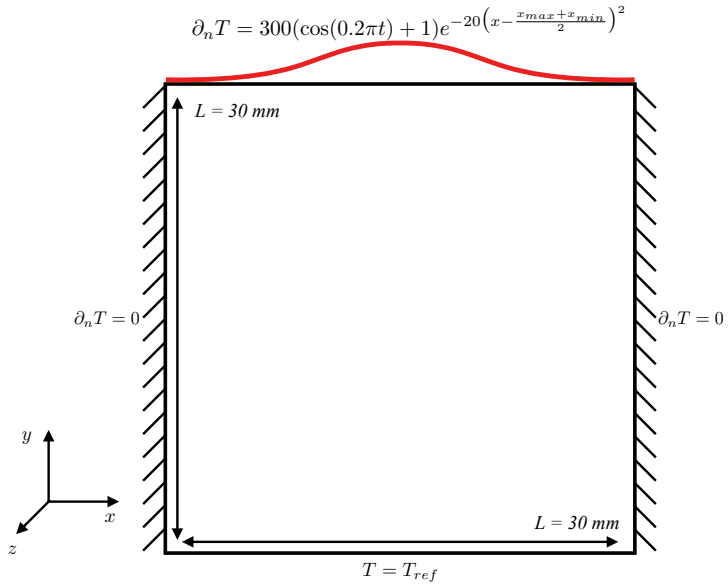

Figure 2 – Une plaque de fer sur laquelle est posé un élément chauffant.

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 ?