Une méthode de résolution numérique de l'équation de Poisson

Mourad Ismail

4 octobre 2012

Table des matières

L'équation de Poisson intervient dans plusieurs projets proposés cette année. On présente ici une méthode numérique pour la résolution d'une telle équation en utilisant la méthode des différences finies et la transformée de Fourier rapide. Ce document synthétique est en grande partie extrait du cours *Computational Physics* de Richard Fitzpatrick (University of Texas at Austin) [1]

1. http://farside.ph.utexas.edu/

1 Équation de Poisson en une dimension

On considère le problème modèle suivant : étant donnés deux constantes u_g et u_d et une fonction f (supposée assez régulière), trouver une fonction u solution de l'équation de Poisson

$$\frac{d^2u}{dx^2}(x) = f(x), \quad \forall x \in]x_g, x_d[, \tag{1}$$

avec des conditions aux limites de Dirichlet

$$u(x_g) = u_g \text{ et } u(x_d) = u_d. \tag{2}$$

Nous commençons par discrétiser l'intervalle de résolution $]x_g, x_d[$ en $N+1$ subdivisions $]x_i, x_i + 1[$ tel que

$$\forall i \in \{0, \cdots, N+1\}, \quad x_i = x_g + i\frac{x_d - x_g}{N+1} = x_g + ih_x. \tag{3}$$

Ensuite, nous écrivons la version discrète du problème (1)-(2) en utilisant les points x_i et une approximation de la dérivée seconde de u basée sur un développement de Taylor d'ordre 2. On obtient

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h_x^2} = f_i, \quad \forall i \in \{1, \cdots, N\}, \tag{4}$$

$$u_0 = u_g \text{ et } u_{N+1} = u_d, \tag{5}$$

où u_i désigne $u(x_i)$ et f_i désigne $f(x_i)$.

Le problème discret (4)-(5) se réduit à la résolution du système linéaire

$$\mathbf{AU} = \mathbf{F}. \tag{6}$$

Avec

– $\mathbf{A}$ est la matrice de $\mathcal{M}_N(\mathbb{R})$ donnée par

$$\frac{1}{h_x^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & \ddots & & \vdots \\ 0 & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & 1 & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & -2 \end{pmatrix} \tag{7}$$

– $\mathbf{U}$ et $\mathbf{F}$ sont les vecteurs de $\mathbb{R}^N$ définis par

$$\mathbf{U} = \begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ u_N \end{pmatrix} \text{ et } \mathbf{F} = \begin{pmatrix} f_1 \\ \vdots \\ \vdots \\ f_N \end{pmatrix} \tag{8}$$

Théoriquement, la solution $\mathbf{U}$ du système linéaire (6) est bien évidemment définie par $\mathbf{U} = \mathbf{A}^{-1}\mathbf{F}$ mais en pratique on ne calcule **jamais** l'inverse d'une matrice. En effet, cette opération est très coûteuse en temps de calcul et contrairement à cet exemple simple où $\mathbf{A}$ est symétrique et tridiagonale (donc facilement inversible), dans le cas générale et surtout en dimensions supérieures, la matrice $\mathbf{A}$ n'a pas forcément une forme simple. Il existe plusieurs méthodes numériques pour résoudre des systèmes linéaires sans calculer explicitement l'inverse d'une matrice. Nous présentons en annexe une méthode de résolution de système linéaire dont la matrice est tridiagonale.

2 Généralisation en deux dimensions

Notons par $\Omega =]x_g, x_d[\times]0, L[$ un domaine rectangulaire de $\mathbb{R}^2$. À présent on considère le problème suivant : étant données des constantes α_g, α_d, β_g, β_d, deux fonctions γ_g, γ_d et une fonction « assez régulière » f, trouver une fonction u telle que

$$\Delta u(x, y) = f(x, y), \quad \forall (x, y) \in \Omega, \tag{9}$$

avec les conditions aux limites suivantes :

$$\alpha_g u(x_g, y) + \beta_g \frac{\partial u}{\partial x}(x_g, y) = \gamma_g(y), \quad \forall y \in]0, L[, \tag{10}$$

$$\alpha_d u(x_d, y) + \beta_d \frac{\partial u}{\partial x}(x_d, y) = \gamma_d(y), \quad \forall y \in]0, L[, \tag{11}$$

$$u(x, 0) = u(x, L), \quad \forall x \in]x_d, x_g[. \tag{12}$$

Nous réécrivons le problème (9)-(12) en cherchant une solution u qui admet ce développement en série de Fourier (selon la variable y) :

$$u(x, y) = \sum_{j=0}^{\infty} U_j(x) \sin(j\pi \frac{y}{L}). \tag{13}$$

Dans ces conditions f admet un développement analogue :

$$f(x, y) = \sum_{j=0}^{\infty} F_j(x) \sin(j\pi \frac{y}{L}). \tag{14}$$

Avec

$$U_j(x) = \frac{2}{L}\int_0^L u(x,y)\sin(j\pi\frac{y}{L})dy, \tag{15}$$

$$F_j(x) = \frac{2}{L}\int_0^L f(x,y)\sin(j\pi\frac{y}{L})dy. \tag{16}$$

Ainsi, l'équation (9) devient

$$\frac{d^2U_j}{dx^2}(x) - \frac{j^2\pi^2}{L^2}U_j(x) = F_j(x), \quad \forall x \in]x_g, x_d[\text{ et } j \in \mathbb{N}. \tag{17}$$

Nous commençons d'abord par discrétiser l'équation (17) dans la direction des y en tronquant les développements en séries de Fourier. Ce qui revient à remplacer la somme infinie sur j par une somme finie sur les indices j dans $\{0, \cdots, J\}$. Plus précisément, ce procédé est équivalent à la discrétisation de l'intervalle $[0, L]$ en $J+1$ points $y_j = j\frac{L}{J}$, $\quad j \in \{0, \cdots, J\}$. Ainsi, la version discrète de l'équation (17) s'écrit

$$\frac{d^2U_j}{dx^2}(x) - \frac{j^2\pi^2}{L^2}U_j(x) = F_j(x), \quad \forall x \in]x_g, x_d[\text{ et } j \in \{1, \cdots, J-1\}. \tag{18}$$

Avec les conditions aux limites suivantes

$$\alpha_g U_j(x_g) + \beta_g \frac{dU_j}{dx}(x_g) = \Gamma_{gj}, \tag{19}$$

$$\alpha_d U_j(x_d) + \beta_d \frac{dU_j}{dx}(x_d) = \Gamma_{dj}, \tag{20}$$

où

$$\Gamma_{gj} = \frac{2}{L}\int_0^L \gamma_g(y)\sin(j\pi\frac{y}{L})dy, \tag{21}$$

$$\Gamma_{dj} = \frac{2}{L}\int_0^L \gamma_d(y)\sin(j\pi\frac{y}{L})dy. \tag{22}$$

$$\tag{23}$$

Notons que la condition de périodicité (12) est automatiquement vérifiée par U_j puisque $\sin(j\pi\frac{y}{L})$ s'annule en $y = 0$ et en $y = L$.

Ensuite, nous écrivons la discrétisation de l'équation (18) selon la variable x. Pour ce faire, nous introduisons la suite $(x_i)_{0\leq i\leq N+1}$ définie par $x_i = x_g + ih_x$ ($h_x = \frac{x_d - x_g}{N+1}$) et nous remplaçons x par x_i dans (18). Il en résulte l'équation discrète suivante

$$\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{h_x^2} - \frac{j^2\pi^2}{L^2}U_{i,j} = F_{i,j}, \quad \forall i \in \{1, \cdots, N\} \text{ et } j \in \{1, \cdots, J-1\}, \tag{24}$$

où l'on a noté $U_j(x_i)$ par $U_{i,j}$ et $F_j(x_i)$ par $F_{i,j}$. L'équation (24) peut aussi s'écrire sous cette forme

$$\begin{aligned} U_{i-1,j} - (2 + j^2\kappa^2)U_{i,j} + U_{i+1,j} &= F_{i,j}h_x^2, \\ \forall \quad i \in \{1, \cdots, N\} \text{ et } j &\in \{1, \cdots, J-1\}. \end{aligned} \tag{25}$$

Avec $\kappa = \pi\frac{h_x}{L}$.Ce qui nous donne, pour chaque $j \in \{1, J-1\}$, un système tridiagonale de la même forme que celui décrit dans la section 1. On pourrait ainsi utiliser la même technique (et le même programme) de résolution.

Enfin, la discrétisation (selon x) des conditions aux limites (19) et (20) est donnée par

$$U_{0,j} = \frac{\Gamma_{gj}h_x - \beta_g U_{1,j}}{\alpha_g h_x - \beta_g}, \tag{26}$$

$$U_{N+1,j} = \frac{\Gamma_{dj}h_x - \beta_d U_{N,j}}{\alpha_d h_x - \beta_d}. \tag{27}$$

Annexes

A Programmes

Si vous utilisez les programmes présentés dans cette section, vous n'aurez pas à implémenter des classes pour modéliser les tableaux. Nous vous proposons d'utiliser les classes de la librairie `blitz++`. [2]

A.1 Équation de Poisson en une dimension

Cette section contient tous les programmes nécessaires pour la résolution du problème présenté à la section 1 avec des conditions aux limites plus générales. Plus précisément on résout

$$\frac{d^2u}{dx^2}(x) = f(x), \quad \forall x \in]x_g, x_d[, \tag{28}$$

$$\alpha_g u(x_g) + \beta_g \frac{du}{dx}(x_g) = \gamma_g, \tag{29}$$

$$\alpha_d u(x_d) + \beta_d \frac{du}{dx}(x_d) = \gamma_d. \tag{30}$$

2. http://www.oonumerics.org/blitz/

Script makefile

```
IDIR = /usr/include/
ODIR = .
Cxx = g++
CFLAGS = -I$(IDIR) -Wall
LIBS = -L/usr/lib -lblitz
OBJ = main.o Poisson1D.o Tridiagonal.o

%.o: %.cpp
        $(Cxx) $(CFLAGS) -c -o $@ $<
lapl1D : $(OBJ)
        $(Cxx)  $^ -o $@ $(LIBS)



        @echo
        @echo "################################################"
        @echo "tapez ./lapl1D pour executer votre programme"
        @echo "################################################"
        @echo

clean:
        @echo
        @echo "Nettoyage du repertoire. On efface les .o"
        rm -f $(ODIR)/*.o *~ core a.out lapl1D
```

Il faudrait sans doute modifier le script ci-dessus en fonction de votre environnement, notamment le compilateur et le chemin des librairies.

Fichier main.cpp

```
#include <fstream>  // pour pouvoir écrire dans des fichiers
#include <blitz/array.h> // gestions des tableaux avec blitz++

using namespace blitz;
using namespace blitz::tensor;

typedef Array<double,1> Array1D;  // un raccourci


int main()
{
  // déclaration de la fonction Poisson1D.
  // Voir Poisson1D.cpp pour l'implémentation
  void Poisson1D (Array1D& u, Array1D v,
                  double alpha_l, double beta_l, double gamma_l,
                  double alpha_h, double beta_h, double gamma_h,
```

```
                double dx);

const int N = 100; // Nombre de points de discrétisation
                   // dans la direction x
Range D(0,N+1);    // indice varaint de 0 a N+1

double L = 1.0; // notre intervalle [0,L]

double h = L / (N+1); // pas du maillage

Array1D uCal(D);  // Solution calculée
Array1D sndMbre(D); // Second membre
Array1D solExacte(D); // Solution exacte

// Exemple de solution exacte et second membre
double coeff = 250.;
sndMbre = - (2*coeff*(1.+ coeff*((i*h-0.5)*(i*h-0.5))) -
             8*coeff*coeff*(i*h-0.5)*(i*h-0.5))/
  pow((1.+ coeff*(i*h-0.5)*(i*h-0.5)),3);
solExacte = 1./(1.+ coeff*(i*h-0.5)*(i*h-0.5));

// Constantes pour les conditions aux limites
double alpha_l = 1;
double beta_l = 0;
double gamma_l = 1./(1.+coeff*0.25);
double alpha_h = 1;
double beta_h = 0;
double gamma_h = gamma_l;

// Appel de la fonction Poisson1D pour résoudre le problème
Poisson1D (uCal, sndMbre,
           alpha_l, beta_l, gamma_l,
           alpha_h, beta_h, gamma_h, h);

// Calcul de l'erreur entre solution exacte et solution calculée
cout << "Root-mean-square error:" << endl
     << "2nd order approximation: " << sqrt(mean(sqr(solExacte-uCal))
     << endl;

ofstream sortieFichier("lapl1D.dat");
cout << endl;
cout << "Creation du fichier 'lapl1D.dat' pour stocker la solution "
     << "calculee ainsi que la solution exacte." << endl;
cout << endl;

for(int i=0; i<=N+1; i++){
  sortieFichier << i*h << " "<< uCal(i) << " "
                << solExacte(i) << endl;
}
```

```
  sortieFichier.close(); // fermeture du fichier

  cout << "***" << endl;
  cout << "Pour visualiser la solution exacte et la solution que vous "
       << "venez de calculer, utilisez la ligne de commande suivante "
       <<" sous 'gnuplot' : " << endl;
  cout << "plot 'lapl1D.dat' using 1:2 title 'Ucal',"
       << "'lapl1D.dat' using 1:3 with lines title 'Uex'" << endl;
  cout << "***" << endl;
  return 0;
}
```

Fichier Poisson1D.cpp

```
// Poisson1D.cpp
// From http://farside.ph.utexas.edu
// Function to solve Poisson's equation in 1-d:
// d^2u/dx^2 = v  for x_l <= x <= x_h
// alpha_l u + beta_l du/dx = gamma_l  at  x = x_l
// alpha_h u + beta_h du/dx = gamma_h  at  x = x_h
// Arrays u and v assumed to be of extent N+2.
// Now, i^th elements of arrays correspond to
// x_i = x_l + i dx,   i = 0, ..., N+1
// Here, dx = (x_h - x_l)/(N+1)  is grid spacing.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u);

void Poisson1D (Array<double,1>& u, Array<double,1> v,
                double alpha_l, double beta_l, double gamma_l,
                double alpha_h, double beta_h, double gamma_h,
                double dx)
{
  // Find N. Declare local arrays.
  int N = u.extent(0) - 2;
  // u.extent(0) = taille de la premiere colonne de u

  Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2);

  // Initialize tridiagonal matrix
  for (int i = 2; i <= N; i++) a(i) = 1.;
  for (int i = 1; i <= N; i++) b(i) = -2.;
  b(1) -= beta_l / (alpha_l * dx - beta_l);
```

```
  b(N) += beta_h / (alpha_h * dx + beta_h);
  for (int i = 1; i <= N-1; i++) c(i) = 1.;

  // Initialize right-hand side vector
  for (int i = 1; i <= N; i++)
    w(i) = v(i) * dx * dx;
  w(1) -= gamma_l * dx / (alpha_l * dx - beta_l);
  w(N) -= gamma_h * dx / (alpha_h * dx + beta_h);

  // Invert tridiagonal matrix equation
  Tridiagonal (a, b, c, w, u);

  // Calculate i=0 and i=N+1 values
  u(0) = (gamma_l * dx - beta_l * u(1)) /
    (alpha_l * dx - beta_l);
  u(N+1) = (gamma_h * dx + beta_h * u(N)) /
    (alpha_h * dx + beta_h);
}
```

Fichier Tridiagonal.cpp

La fonction implémentée dans ce fichier résout un système linéaire tridiagonale

```
// Tridiagonal.cpp
// From http://farside.ph.utexas.edu
// Function to invert tridiagonal matrix equation.
// Left, centre, and right diagonal elements of matrix
// stored in arrays a, b, c, respectively.
// Right-hand side stored in array w.
// Solution written to array u.

// Matrix is N × N. Arrays a, b, c, w, u assumed to be of extent N+2,
// with redundant 0 and N+1 elements.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u)
{
  // Find N. Declare local arrays.
  int N = a.extent(0) - 2;
  Array<double,1> x(N), y(N);

  // Scan up diagonal from i = N to 1
  x(N-1) = - a(N) / b(N);
```

```
  y(N-1) = w(N) / b(N);
  for (int i = N-2; i > 0; i--)
    {
      x(i) = - a(i+1) / (b(i+1) + c(i+1) * x(i+1));
      y(i) = (w(i+1) - c(i+1) * y(i+1)) / (b(i+1) + c(i+1) * x(i+1));
    }
  x(0) = 0.;
  y(0) = (w(1) - c(1) * y(1)) / (b(1) + c(1) * x(1));

  // Scan down diagonal from i = 0 to N-1
  u(1) = y(0);
  for (int i = 1; i < N; i++)
    u(i+1) = x(i) * u(i) + y(i);
}
```

A.2 Extension en deux dimensions

Comme nous l'avons expliqué dans la section 2, la résolution de l'équation de Poisson en deux dimensions peut se faire en couplant le programme 1D avec la transformée de Fourier rapide. Pour ce faire, vous n'auriez pas à implémenter la transformée de Fourier, mais vous pouvez utiliser la fonction décrite dans le fichier suivant

Fichier FFT.cpp

```
// FFT.cpp
// From http://farside.ph.utexas.edu
// Set of functions to calculate Fourier-cosine and -sine transforms
// of real data using fftw Fast-Fourier-Transform library.
// Input/ouput arrays are assumed to be of extent J+1.
// Uses version 2 of fftw library (incompatible with vs 3).

#include <fftw.h>
#include <blitz/array.h>

using namespace blitz;

// Calculates Fourier-cosine transform of array f in array F
void fft_forward_cos (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
```

```
  c_re (ff[0]) = f(0); c_im (ff[0]) = 0.;
  c_re (ff[J]) = f(J); c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = f(j); c_im (ff[2*J-j]) = 0.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = c_re (FF[0]); F(J) = c_re (FF[J]);
  for (int j = 1; j < J; j++)
    {
      F(j) = 2. * c_re (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}

// Calculates inverse Fourier-cosine transform of array F in array f
void fft_backward_cos (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = F(0); c_im (FF[0]) = 0.;
  c_re (FF[J]) = F(J); c_im (FF[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (FF[j]) = F(j) / 2.; c_im (FF[j]) = 0.;
      FF[2*J-j] = FF[j];
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  f(0) = c_re (ff[0]); f(J) = c_re (ff[J]);
  for (int j = 1; j < J; j++)
```

```
    {
      f(j) = c_re (ff[j]);
    }
}

// Calculates Fourier-sine transform of array f in array F
void fft_forward_sin (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (ff[0]) = 0.; c_im (ff[0]) = 0.;
  c_re (ff[J]) = 0.; c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = - f(j); c_im (ff[2*J-j]) = 0.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = 0.; F(J) = 0.;
  for (int j = 1; j < J; j++)
    {
      F(j) = - 2. * c_im (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}

// Calculates inverse Fourier-sine transform of array F in array f
void fft_backward_sin (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = 0.; c_im (FF[0]) = 0.;
  c_re (FF[J]) = 0.; c_im (FF[J]) = 0.;
```

```
  for (int j = 1; j < J; j++)
    {
      c_re (FF[j]) = 0.; c_im (FF[j]) = - F(j) / 2.;
      c_re (FF[2*J-j]) = 0.; c_im (FF[2*J-j]) = F(j) / 2.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  f(0) = 0.; f(J) = 0.;
  for (int j = 1; j < J; j++)
    {
      f(j) = c_re (ff[j]);
    }
}
```

Notez que cette fonction utilise la librairie `fftw`.[3] Il faudrait penser à mettre à jour votre `makefile` en y ajoutant notamment (selon les versions de `fftw`) :

```
  LIBS = -L/usr/lib -lblitz -lfftw
```

ou

```
  LIBS = -L/usr/lib -lblitz -ldfftw
```

B Post-traitement

B.1 Présentation des résultats numériques avec Gnuplot

Une manière simple de présenter vos résultats numériques en 1D et en 2D consiste à utiliser le logiciel libre `Gnuplot`.[4] Vous enregistreriez vos résultats sous la forme d'au moins deux colonnes. La premières contient les abscisses x_i et la seconde contient les ordonnées $f(x_i)$.

À titre d'exemple, nous avons mis la ligne de commande suivante à la fin du fichier `main.cpp` (section A.1)

```
plot 'lapl1D.dat' using 1:2 title 'Ucal', 'lapl1D.dat' using 1:3 with lines title 'Uex'
```

3. http://www.fftw.org
4. http://www.gnuplot.info

Le fichier `lapl1D.dat` est composé de trois colonnes :

1. la première contient les abscisses des points x_i,
2. la deuxième contient les valeurs de la solution calculée aux point x_i $(Ucal(x_i))$,
3. la dernière colonne contiennent les valeurs de la solution exacte aux points x_i $(solExacte(i))$.

Par la ligne de commande ci-dessus, nous demandons à `Gnuplot` d'utiliser les deux premières colonnes pour tracer la solution calculée et en prenant `Ucal` comme légende de la courbe. Dans la deuxième partie de la ligne de commande, nous demandons à `Gnuplot` de tracer la solution exacte aux points x_i en utilisant la première et la troisième colonne. La légende sera `Uex` et les points représentant la solution exacte seront liés entre eux par des segments.

De même, on peut utiliser cet exemple de ligne de commande pour présenter des résultats en deux dimensions

```
splot 'fichier.dat' using 1:2:3 title 'mon titre' with pm3d
```

Dans ce cas, il faut avoir au moins trois colonnes : une pour les abscisses x_i, une pour les ordonnées y_i et une pour les valeurs $f(x_i, y_i)$. L'option `pm3d` signifie `palette-mapped 3d` et consiste à interpoler entre les sommets de la grille (l'équivalent de `with lines` en 2D). Les commandes `help plot` et `help splot` vous donneront plus d'information.

B.2 Enregistrements des résultats dans plusieurs fichiers

Si nous nous intéressons à un problème d'évolution, nous sommes souvent amené à sauvegarder les résultats dans plusieurs fichiers correspondant aux différents pas de temps. Ci-après une manière simple de créer automatiquement des fichiers numérotés de 0 à 999 : `fichier.000`, ..., `fichier.999`

```
char nom_fichier[12]="fichier.";
int ind1 = (t_iterator%1000 - t_iterator%100)/100;
int ind2 = (t_iterator%100  - t_iterator%10 )/10 ;
int ind3 =  t_iterator%10;
nom_fichier[8] = char(ind1 + 48);
nom_fichier[9] = char(ind2 + 48);
nom_fichier[10]= char(ind3 + 48);
ofstream sortieFichier(nom_fichier);
```

Pour pouvoir utiliser les instruction ci-dessus, n'oubliez pas `#include <fstream>`.

B.3 Scripts Gnuplot

Quant on a quelques centaines fichiers de résultats, on peut utiliser `Gnuplot` en mode script. Ci-après deux scripts `shell` permettant la création d'images `png` à partir des fichiers résultats

- ScriptGnuImages

```
#!/bin/zsh
for j in {000..999}
 do
   cat fichier.$j > file.dat
   gnuplot scriptGnuplot
   mv file.png fichier.$j.png
 done
```

- scriptGnuplot

```
set term png
set output "file.png"
splot [0:1] [0:1] [-0.1:1.2] 'file.dat'  using 1:2:3  with pm3d
```

B.4 Création d'une animation

Nous pouvons crée une animation au format `mpeg` ou `avi` à l'aide des images créées dans la section précédente.

```
ffmpeg -i fichier.%3d.png video.mpg
```