



**HAL**  
open science

## Langage C++ et calcul scientifique

Pierre Saramito

► **To cite this version:**

Pierre Saramito. Langage C++ et calcul scientifique. DEA. Grenoble, France, 2005, pp.102. cel-00573975v1

**HAL Id: cel-00573975**

**<https://cel.hal.science/cel-00573975v1>**

Submitted on 6 Mar 2011 (v1), last revised 16 Dec 2013 (v4)

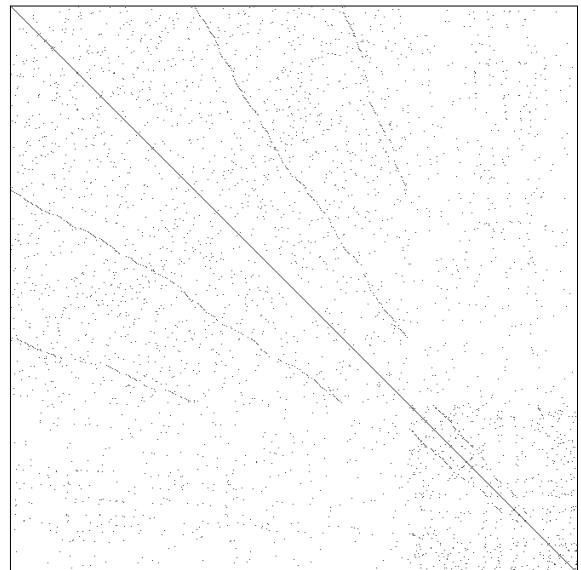
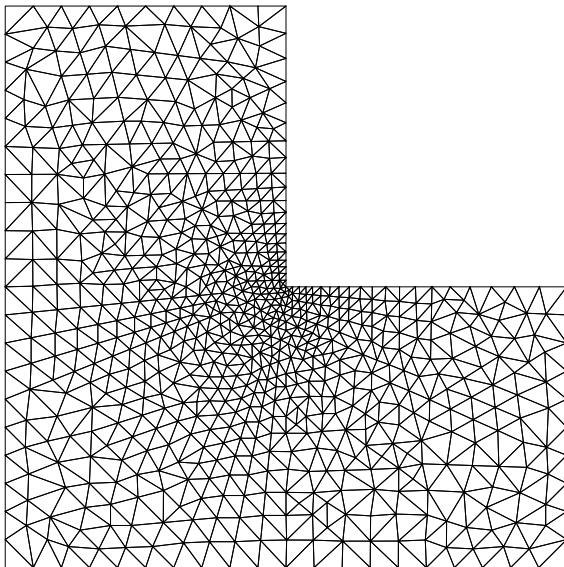
**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Langage C++ et calcul scientifique

**Pierre Saramito**

mardi 14 février 2006



Copyright (c) 2003-2006 Pierre Saramito

Copyright : cette œuvre est libre, vous pouvez la redistribuer et/ou la modifier selon les termes de la Licence Art Libre. Vous trouverez un exemplaire de cette Licence sur le site d'Art Libre <http://www.artlibre.org> ainsi que sur d'autres sites.

# Chapitre 1

## Introduction : les quaternions

Ce chapitre a pour but d'introduire les concepts de base utilisés par ce cours de C++ en calcul scientifique : conception de classe, classe paramétrée par un type, ainsi que la surcharge d'opérateur.

Les quaternions ont été introduits en 1853 par HAMILTON. Ils plus tard été utilisés en mécanique quantique, et, plus récemment, en animation 3D, pour calculer des rotations d'axes.

Les quaternions sont des nombres *hypercomplexes* qui forment un groupe non commutatif. Ils peuvent être représentés à l'aide des matrices complexes  $2 \times 2$  :

$$h = \begin{pmatrix} z & w \\ -\bar{w} & \bar{z} \end{pmatrix} = \begin{pmatrix} a + ib & c + id \\ -c + id & a - ib \end{pmatrix} = a\mathcal{U} + b\mathcal{I} + c\mathcal{J} + d\mathcal{K}$$

avec

$$\mathcal{U} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{I} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, \quad \mathcal{J} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad \mathcal{K} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

et  $\mathcal{I}^2 = \mathcal{J}^2 = \mathcal{K}^2 = -\mathcal{U}$  généralisent les nombres imaginaires purs. La norme de  $h$  est définie par  $|h| = \sqrt{|z|^2 + |w|^2}$  et le conjugué de  $h$  est  $\bar{h} = a\mathcal{U} - b\mathcal{I} - c\mathcal{J} - d\mathcal{K}$ .

L'objectif de cette partie est de construire une classe `quaternion` ayant le même type d'interface, et compatible avec la classe `complex` de la librairie standard C++ :

```
<complex>
```

---

```

template <class T>
class complex {
  public :
    complex(const T& a=0, const T& b=0);
    complex(const complex<T>& z);
    complex<T>& operator= (const complex<T>& z);
    ~complex();
    T& real();
    T& imag();
    const T& real() const;
    const T& imag() const;
  protected :
    T re, im;
};

```

La classe est paramétrée par le type `T` qui représente le type flottant (`float`, `double`, `long double`...) approchant les nombres réels, et utilisé pour les parties réelles et imaginaires. Ainsi le type `float` est limitée à une précision de sept décimales et le type `double` à quinze décimales. La classe `quaternion` est complété un ensemble de fonctions décrivant l'algèbre standard :

```

template<class T> istream& operator >> (ostream& in, quaternion<T>& h);
template<class T> ostream& operator << (ostream& out, const quaternion<T>& h);
template<class T> complex<T> operator+ (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator- (const complex<T>& z); // moins unaire
template<class T> complex<T> operator- (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator* (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/ (const complex<T>&, const complex<T>&);
template<class T> T conj (const complex<T>&);
template<class T> T abs (const complex<T>&);

```

La classe `quaternion` peut ainsi s'écrire :

## quaternion.h

```

#include <complex>
template <typename T>
class quaternion {
public :
    quaternion (const T& a=0, const T& b=0, const T& c=0, const T& d=0);
    quaternion (const std::complex<T>& z, const std::complex<T>& w = std::complex<T>());
    quaternion (const quaternion<T>& h);
    quaternion<T>& operator= (const quaternion<T>& h);
    std::complex<T>& z();
    std::complex<T>& w();
    const std::complex<T>& z() const;
    const std::complex<T>& w() const;
protected :
    std::complex<T> zc, wc;
};

```

On a choisi ici de représenter un quaternion par deux nombres complexes plutôt que par quatre réels : cela permettra d'écrire les opérations algébriques de façon plus compacte. De même que pour la classe `complex`, la classe `quaternion` est paramétrée par le type `T` qui représente l'approximation par des nombres flottants des nombres réels. Remarquons le préfixe `std::` devant `complex` : cette class est accédée dans la librairie standard via le domaine de nom `std` après inclusion du fichier d'entête correspondant.

Les constructeurs et opérateurs d'affectation s'écrivent :

```

template <typename T>
quaternion<T>::quaternion (const T& a, const T& b, const T& c, const T& d)
    : zc(a,b), wc(c,d) {}
template <typename T>
quaternion<T>::quaternion (const std::complex<T>& z, const std::complex<T>& w) : zc(z), wc(w) {}
template <typename T>
quaternion<T>::quaternion (const quaternion<T>& h) : zc(h.zc), wc(h.wc) {}
template <typename T>
quaternion<T>& quaternion<T>::operator= (const quaternion<T>& h) {
    z = h.z; w = h.w; return *this; }

```

Le premier constructeur prend quatres réels en arguments. Ces arguments ont tous des valeurs par défaut, si bien qu'il est possible de déclarer un quaternion sans préciser de valeur : ce sera zéro. Nous avons affaire au *constructeur par défaut*. Lorsque le constructeur est appelé avec un seul argument de type flottant, il convertit cette valeur en quaternion : nous avons affaire à une *conversion implicite de type*. Nous avons affaire au *constructeur par défaut*. De même, le second constructeur peut n'être appelé qu'avec un seul nombre complexe, le second étant nul par défaut : nous retrouvons la conversion implicite de type, ici du type complexe au type quaternion. Le troisième constructeur est le *constructeur de copie*. L'opérateur d'affectation `operator=` prend également en argument un quaternion.

Les accesseurs aux données `z` et `w` de la classe sont :

```

template <typename T> std::complex<T>& quaternion<T>::z() { return zc; }
template <typename T> std::complex<T>& quaternion<T>::w() { return wc; }
template <typename T> const std::complex<T>& quaternion<T>::z() const { return zc; }
template <typename T> const std::complex<T>& quaternion<T>::w() const { return wc; }

```

Remarquons que on distingue les accès en lecture seule, agrémenté du mot-clef `const`, et qui renvoient une référence constante sur un `complex<T>`, des accès en lecture et écriture, qui renvoie une référence `complex<T>` sans la restreindre à être constante.

L'addition entre deux quaternions s'écrit simplement :

```

template <typename T>
quaternion<T> operator+ (const quaternion<T>& h, quaternion<T> m) {
    quaternion<T> r;
    r.z() = h.z() + m.z();
    r.w() = h.w() + m.w();
    return r;
}

```

Les opérateurs de soustraction et de multiplication sont assez analogues :

```

template <typename T>
quaternion<T> operator- (const quaternion<T>& h, quaternion<T> m) {
    quaternion<T> r;
    r.z() = h.z() - m.z();
    r.w() = h.w() - m.w();
    return r;
}
template <typename T>
quaternion<T> operator* (const quaternion<T>& h1, quaternion<T> h2) {
    quaternion<T> r;
    r.z() = h1.z()*h2.z() - h1.w()*conj(h2.w());
    r.w() = h1.z()*h2.w() + h1.w()*conj(h2.z());
    return r;
}

```

Le module d'un quaternion est  $|h| = \sqrt{|z|^2 + |w|^2}$  où  $|z|$  et  $|w|$  sont les modules des nombres complexes  $z$  et  $w$ . Le quaternion conjugué est noté  $\bar{h} = \bar{z} - w$  où  $\bar{z}$  est simplement le nombre complexe conjugué de  $z$ . Ainsi  $h\bar{h} = |h|^2$  et l'inverse s'écrit :  $h^{-1} = \bar{h}/|h|^2$ . Ceci va nous permettre d'introduire la division entre deux quaternions :

```

template <typename T>
T abs (const quaternion<T>& h) {
    return sqrt(abs(h.z())*abs(h.z()) + abs(h.w())*abs(h.w()));
}
template <typename T>
quaternion<T> conj (const quaternion<T>& h) {
    quaternion<T> r;
    r.z() = conj(h.z());
    r.w() = -h.w();
    return r;
}
template <typename T>
quaternion<T> operator/ (const quaternion<T>& h1, quaternion<T> h2) {
    quaternion<T> r = h1*conj(h2);
    T deno = abs(h2)*abs(h2);
    r.z() /= deno;
    r.w() /= deno;
    return r;
}

```

La fonction d'écriture écrit enfin :

```

#include <iostream>
template<typename T>
std::ostream& operator<< (std::ostream& out, const quaternion<T>& h) {
    out << "(" << h.z() << ", " << h.w() << ")";
    return out;
}

```

Ainsi, un quaternion  $h = 3 + 2i + 5j + 7k$  sera formaté suivant deux nombres complexes et parenthésé :  $((3,2), (5,7))$ . La fonction de lecture associé s'écrit :



```

template<typename T>
std::istream& operator>> (std::istream& is, quaternion<T>& h) {
    std::complex<T> z, w;
    char c;
    is >> c;
    if (c == '(') {
        is >> z >> c;
        if (c == ',') {
            is >> w >> c;
            if (c == ')')
                h = quaternion<T>(z, w);
            else
                is.setstate(std::ios_base::failbit);
        } else {
            if (c == ')')
                h = z;
            else
                is.setstate(std::ios_base::failbit);
        }
    } else {
        is.putback(c);
        is >> z;
        h = z;
    }
    return is;
}

```

Les fonctions de lecture sont souvent plus longues et compliquées de les fonctions d'écriture, car elles testent différentes variantes d'écriture et détectent les erreurs de format d'entrée. Ainsi se termine la classe quaternion. Écrivons un petit programme qui teste ses fonctionnalités :

quaternion\_tst.cc

---

```

#include "quaternion.h"
using namespace std;
int main(int argc, char**argv) {
    quaternion<double> h1 (1,1,7,9);
    quaternion<double> h2 (1,-1,-7,-9);
    cout << "h1 = " << h1 << endl;
    cout << "h2 = " << h2 << endl;
    cout << "h1+h2 = " << h1+h2 << endl;
    cout << "h1-h2 = " << h1-h2 << endl;
    cout << "h1*h2 = " << h1*h2 << endl;
    cout << "h1/h2 = " << h1/h2 << endl;
    cout << "(h1/h2)*h2 = " << (h1/h2)*h2 << endl;
    return 0;
}

```

La compilation et le test sont donnés par :

```

c++ quaternion_tst.cc -o quaternion_tst
./quaternion_tst

```

# Chapitre 2

## Introduction aux algorithmes génériques

	cg.h
<pre>algorithme gc   entrée <math>A, x^{(0)}, b, M</math>   sortie <math>(x^{(i)})_{i \geq 1}</math>  début   <math>r^{(0)} := b - Ax^{(0)}</math>   <math>z^{(0)} := M^{-1}r^{(0)}</math>   <math>p^{(1)} := z^{(0)}</math>   <math>\rho_0 := (r^{(0)}, z^{(0)})</math>    pour <math>i := 1, 2, \dots</math>     <math>q^{(i)} := Ap^{(i)}</math>     <math>\alpha_i := \rho_{i-1} / (p^{(i)}, q^{(i)})</math>     <math>x^{(i)} := x^{(i-1)} + \alpha_i q^{(i)}</math>     <math>r^{(i)} := r^{(i-1)} - \alpha_i q^{(i)}</math>     si <math>\ r^{(i)}\  &lt; \epsilon</math> alors arrê      <math>z^{(i)} := M^{-1}r^{(i)}</math>     <math>\rho_i := (r^{(i)}, z^{(i)})</math>     <math>\beta_i := \rho_i / \rho_{i-1}</math>     <math>p^{(i+1)} := z^{(i)} + \beta_i p^{(i)}</math>    fin fin</pre>	<pre>template &lt;typename Mat, typename Vec,           typename Precond, typename Real&gt; int cg (const Mat &amp;A, Vec &amp;x, const Vec &amp;b,         const Precond &amp;M, int maxiter, Real tol) {   Real normb = norm(b);   if (normb == 0) { x = 0; return 0; }   Vec r = b - A*x;   Vec z = M.solve(r);   Vec p = z;   Real rho = dot(r, z);   Real rho_prev = rho;   std::clog &lt;&lt; "pcg : 0 " &lt;&lt; norm(r)/normb &lt;&lt; std::endl;   for (int i = 1; i &lt; maxiter; i++) {     Vec q = A*p;     Real alpha = rho / dot(p, q);     x += alpha * p;     r -= alpha * q;     Real resid = norm(r) / normb;     std::clog &lt;&lt; "pcg : "&lt;&lt; i &lt;&lt; " " &lt;&lt; resid &lt;&lt; std::endl;     if (resid &lt;= tol) return 0;     z = M.solve(r);     rho = dot(r, z);     Real beta = rho/rho_prev;     rho_prev = rho;     p = z + beta*p;   }   return 1; }</pre>

## 2.1 L'algorithme du gradient conjugué

Le langage C++ permet la généricité. Ainsi, en programmant la résolution de  $Ax = b$  par l'algorithme du gradient conjugué, nous pouvons disposer d'une seule version du code pour tous les types de matrices.

Ceci nécessite que les types `Mat`, `Vec` et `Real` partagent un interface commun :

```
Vec ← Real * Vec
Vec ← Mat * Vec
Real ← dot(Vec, Vec)
Real ← norm(Vec)
Vec ← Vec ± Vec
```

Le type `Precond` modélise un *préconditionneur* noté  $M$ . On remplace alors la résolution de  $Ax = b$  par celle de  $M^{-1}Ax = M^{-1}b$ , où  $M$  est une matrice inversible, a priori assez facile à inverser, et afin que la résolution du système ainsi transformé converge plus rapidement.

La classe `Precond` devra satisfaire avec la classe `Vec` une opération :

```
Vec ← Precond.solve(Vec)
```

Le plus simple des préconditionneurs est la matrice identité :

`eye.h`

---

```
class eye {
public :
    template <typename Vec> const Vec& solve (const Vec& x) const { return x; }
    template <typename Vec> const Vec& operator* (const Vec& x) const { return x; }
};
```

Un appel au gradient conjugué sans préconditionnement aura la forme :

```
cg (A, x, b, eye(), maxiter, tol);
```

Il est bien sûr possible d'élaborer des préconditionneurs plus élaborés : préconditionneur diagonal, de GAUSS-SEIDEL ou SSOR, de CHOLESKI incomplet, etc. En s'appuyant sur une analyse des algorithmes itératifs de résolution [BBC<sup>+</sup>94], Dongarra *et al.* ont proposé dans cet esprit la librairie IML++ [DLPR01], librairie générique en C++ et contenant une grande variété de méthodes itératives, notamment :

- gradient conjugué préconditionné (`cg`)
- gradient bi-conjugué stabilisé (`bicgstab`)
- résidu minimal généralisé (`gmres`)
- résidu quasi-minimal (`qmr`)

Le paragraphe suivant va définir une possibilité de choix pour la classe `Vec`.

## 2.2 La classe `valarray` $\langle T \rangle$

La classe `valarray` $\langle T \rangle$ , proposée dans la librairie standard du C++, représente un vecteur de numérique avec l'arithmétique associée. Cette classe est définie sous l'espace de nom `std` dans l'entête `valarray` :

---

```
<valarray>
```

---

```

template <class T>
class valarray {
  public :
    valarray(); // constructeur de taille nulle
    valarray (size_t n); // de taille n
    valarray (const T& initval, size_t n); // de taille n avec initialisation
    valarray (const valarray<T>& y); // constructeur de copie
    valarray<T>& operator= (const valarray<T>& y); // affectation
    size_t size() const; // renvoie la taille
    T& operator [] (size_t i); // accès en lecture-écriture
    const T& operator [] (size_t i) const; // accès en lecture seule
    // ...
};

```

Un exemple simple d'utilisation de cette classe est :

---

```
valarray_tst.cc
```

---

```

#include <valarray>
#include <iostream>
using namespace std;
int main () {
  valarray<float> u(10);
  valarray<float> v(3.1, 10);
  cout << v[0] << endl;
  return 0;
}

```

Les fonctions `dot(.)` et `norm(.,.)` dont nous avons besoin se définissent simplement dans le fichier entête '`valarray_util.h`' :

---

```
valarray_util.h
```

---

```

#include <valarray>
template <class Vec1, class Vec2>
typename Vec1 : :value_type dot (const Vec1& x, const Vec2& y) {
  typename Vec1 : :value_type sum = 0;
  for (unsigned int i = 0; i < x.size(); i++)
    sum += x[i]*y[i];
  return sum;
}
template <class Vec>
inline typename Vec : :value_type norm (const Vec& x) {
  return sqrt(dot(x,x));
}

```

Nous utiliserons aussi souvent la fonction membre `shift` qui réalise un décalage logique (voir [Str01, p. 740]). L'instruction `v=u.shift(d)` renvoie dans `v` un objet de la classe `valarray<T>` de même taille que `u`, et dont le  $i$ -ème élément `v[i]` est `u[i+d]` si  $i+d$  est dans l'intervalle  $0..u.size()-1$ , et zéro autrement. Ainsi, une valeur positive de `d` décale les éléments vers la gauche de `d` places, avec un remplissage par des zéro. Par exemple :

$$\begin{aligned} \mathbf{u} &= (u_0, u_1, \dots, u_{n-2}, u_{n-1}) \\ \mathbf{u.shift}(1) &= (u_1, u_2, \dots, u_{n-1}, 0) \\ \mathbf{u.shift}(-1) &= (0, u_0, \dots, u_{n-3}, u_{n-2}) \end{aligned}$$

En s'inspirant de la syntaxe du célèbre langage `matlab`, nous avons introduit, pour notre confort, la fonction `range`, très commode avec la classe `valarray`, et définie par :

---

`range.h`

---

```
#include <valarray>
inline std::slice range (size_t first, size_t last) {
    return std::slice (first, last-first+1, 1);
}
```

Lorsque le pas ne vaut pas un, nous avons :

```
std::slice range (size_t first, size_t step, size_t last) {
    return std::slice (first, 1+(last-first)/step, step);
}
```

Nous avons établi la correspondance entre deux syntaxes :

matlab	C++
u (debut :fin)	u [range(debut,fin)]
u (debut :pas :fin)	u [range(debut,pas,fin)]

Certain tableaux à un indice ont une interprétation bidimensionnelle, comme  $(u_{i,j})_{0 \leq i,j \leq n-1}$  dans les schémas aux différences finies. Afin de faciliter cette interprétation, nous introduisons les utilitaires suivants :

```

class col_type {
public :
    col_type (size_t n1) : n(n1) {}
    std::slice operator() (size_t i) { return range (i, n, n*n - 1); }
protected :
    size_t n;
};
class row_type {
public :
    row_type (size_t n1) : n(n1) {}
    std::slice operator() (size_t j) { return range (j*n, (j+1)*n-1); }
protected :
    size_t n;
};
class index_type {
public :
    index_type (size_t n1) : n(n1) {}
    size_t operator() (size_t i, size_t j) { return j+i*n; }
protected :
    size_t n;
};
};

```

Voici un exemple d'utilisation :

range\_tst.h

---

```

#include "range.h"
#include <iostream>
using namespace std;
int main (int argc, char** argv) {
    size_t n = (argc > 1)? atoi(argv[1]) : 10;
    size_t i = (argc > 2)? atoi(argv[2]) : 0;
    size_t j = (argc > 3)? atoi(argv[3]) : 0;
    valarray<double> u (0., n*n);
    row_type row (n);
    col_type col (n);
    index_type elt (n);
    u [row(i)] = 1.0;
    u [col(j)] = 2.0;
    u [elt(i,j)] = 3.0;
    for (size_t j = 0; j < n; j++) {
        for (size_t i = 0; i < n; i++)
            cout << u [elt(i,j)] << " ";
        cout << endl;
    }
}

```

La compilation et le test sont :

```

c++ range_tst.cc -o range_tst
range_tst 5 2 3

```

ce qui a pour effet d'imprimer :

```
0 0 1 0 0
0 0 1 0 0
0 0 1 0 0
2 2 3 2 2
0 0 1 0 0
```

Rapellons que  $(u_{i,j})_{0 \leq i,j \leq n-1}$  n'est pas une *matrice* : les expressions algébriques telles que  $\sum_j u_{i,j} x_j$  n'a pas de sens. Cette structure bidimensionnelle est issue du maillage utilisé pour la discrétisation, et disparaît lorsque ces maillages sont plus généraux.

# Chapitre 3

## Problème de POISSON en dimension un

### 3.1 Les différences finies pour le problème de POISSON

#### 3.1.1 Présentation du problème

Considérons le problème de POISSON avec conditions aux bord homogènes de DIRICHLET :

$(P)$  : trouver  $u$ , définie de  $[-1, 1]$  dans  $\mathbb{R}$ , telle que

$$\begin{aligned} -u'' &= f \text{ dans } ]-1, 1[ \\ u(-1) &= 0 \\ u(1) &= 0 \end{aligned} \tag{3.1}$$

où  $f$  est une fonction donnée, de  $[-1, 1]$  dans  $\mathbb{R}$ . Ce problème est à la base de nombreuses modélisations en mécanique, physique, chimie, thermique. Par exemple, il permet de décrire la vitesse d'un fluide dans une conduite ou un canal, la forme d'un pont ployant sous une charge.

La solution du problème  $(P)$  n'est connue explicitement que pour quelques cas particuliers.

Dans le cas général, il est cependant possible d'approcher la solution  $u(x)$  par la méthode des différences finies. Soit  $n$  un entier et considérons le problème suivant :

$(P)_h$  : trouver  $(u_i)_{0 \leq i \leq n}$  telle que

$$\begin{aligned} \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} &= f(x_i) \text{ pour } 1 \leq i \leq n-1 \\ u_0 &= 0 \\ u_n &= 0 \end{aligned} \tag{3.2}$$



où  $h = 2/n$ ,  $x_i = -1 + ih$ ,  $0 \leq i \leq n$ . Les inconnues  $u_i$  *approchent*  $u(x_i)$  d'autant mieux que  $n$  est grand :

$$u_i \longrightarrow u(x_i) \text{ lorsque } n \longrightarrow +\infty$$

Le problème peut se mettre sous la forme suivante :

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} h^2 f(x_1) \\ \vdots \\ h^2 f(x_{n-1}) \end{pmatrix} \quad (3.3)$$

Il s'agit d'un système linéaire de taille  $n - 1$ . La matrice est notée  $A = (a_{i,j})_{1 \leq i,j \leq n-1}$  et le second membre  $b = (b_i)_{1 \leq i \leq n-1}$ ,  $b_i = h^2 f(x_i)$ .

### 3.1.2 La résolution du système tridiagonal

La manière la plus efficace de résoudre un système tridiagonal est d'utiliser une méthode directe. Si nous avons à résoudre un problème de même matrice avec successivement plusieurs seconds membres, comme ce sera le cas dans la suite, le plus efficace est de factoriser la matrice, par exemple sous la forme  $A = LDL^T$ , où  $D$  est diagonale et  $L$  triangulaire inférieure (voir par exemple [LT93], tome 1, page 268).

La fonction suivante effectue la factorisation en place, au sens où le même tableau `d` contient `diag A` en entrée et `D` en sortie. De même, le tableau `l` contient `tril A`, la partie triangulaire inférieure de  $A$ , en entrée, et `L` en sortie.

```
#include <valarray>
template <class T>
void ldlt_1d (std::valarray<T>& d, std::valarray<T>& l) {
    for (size_t i = 1; i < d.size(); i++) {
        T s = l[i-1] / d[i-1];
        d[i] -= s * l[i-1];
        l[i-1] = s;
    }
}
```

Remarquons le type générique `T` qui peut être le type `float` ou `double`, mais aussi pour utiliser des nombres complexes ou des classes de précision étendue.

Voici la résolution de  $LDL^T x = b$  :

## ldlt\_solve\_1d.h

---

```

template <class T>
void ldlt_solve_1d (const std::valarray<T>& d, const std::valarray<T>& l,
                  std::valarray<T>& b) {
    for (size_t i = 1; i < d.size(); i++)
        b [i] -= l [i-1] * b [i-1];
    for (size_t i = 0; i < d.size(); i++)
        b [i] /= d [i];
    for (size_t i = d.size()-1; i > 0; i--)
        b [i-1] -= l [i-1] * b [i];
}

```

La solution  $x$  prend également la place du second membre  $b$ .

### 3.1.3 Le programme de tests

Considérons le code suivant :

## fd\_uniform\_tst.cc

---

```

#include "ldlt_1d.h"
#include <iostream>
using namespace std;
double u (double x) { return (1-x*x)/2; }
int main (int argc, char** argv) {
    size_t n = (argc > 1) ? atoi(argv[1]) : 11;
    valarray<double> a (2.0, n-1);
    valarray<double> al (-1.0, n-2);
    ldlt_1d (a, al);
    double h = 2.0/n;
    valarray<double> b (h*h, n-1);
    ldlt_solve_1d (a, al, b);
    valarray<double> pi_h_u (n-1);
    for (size_t i = 0; i < n-1; i++)
        pi_h_u [i] = u (-1+(i+1)*h);
    valarray<double> uerr = abs(pi_h_u - b);
    double err = uerr.max();
    cerr << "err " << err << endl;
    return err > 1e-8 ? 1 : 0;
}

```

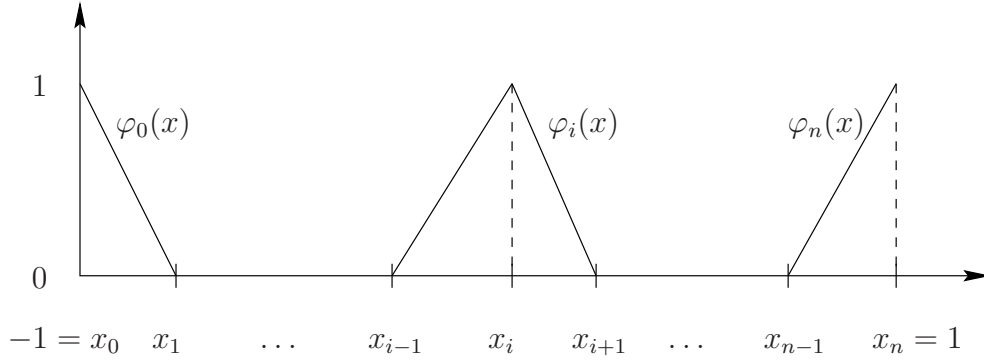
Pour un second membre  $f = 1$ , la solution est connue explicitement :  $u = x(1 - x)/2$ . La compilation et le test sont donnés par :

```

c++ fd_uniform_tst.cc -o fd_uniform_tst
fd_uniform_tst

```

La solution exacte étant polynômiale de degré deux, la solution approchée coïncide donc avec l'interpolée de la solution exacte. L'erreur sera donc nulle – à la précision machine près – quelque soit  $h$ .

FIG. 3.1 – Fonction de base  $\varphi_i$ .

## 3.2 Le cas d'une subdivision non-régulières

### 3.2.1 L'approximation variationnelle

Sans perte de généralité, nous pouvons supposer que  $I = ]-1, 1[$ . Introduisons les formes bilinéaires :

$$m(u, v) = \int_{-1}^1 u(x) v(x) dx, \quad \forall u, v \in L^2(I),$$

$$a(u, v) = \int_{-1}^1 u'(x) v'(x) dx. \quad \forall u, v \in H_0^1(I).$$

Les formes  $a(.,.)$  et  $m(.,.)$  sont habituellement appelées formes d'énergie et de masse, respectivement. La formulation variationnelle de (3.1) est :

(FV) : trouver  $u \in H_0^1(I)$  telle que

$$a(u, v) = m(f, v), \quad \forall v \in H_0^1(I). \quad (3.4)$$

Soit  $-1 = x_0 < x_1 < \dots < x_n = 1$  une subdivision de  $I$ . Posons  $h_i = x_{i+1} - x_i$ ,  $0 \leq i \leq n-1$  et  $h = \max\{h_i, 0 \leq i \leq n-1\}$ . Soit  $X_h$  l'ensemble des fonctions continues de  $I$  dans  $\mathbb{R}$  et affines dans chaque intervalle  $[x_i, x_{i+1}]$ . Introduisons  $V_{0,h} = X_h \cap H_0^1(I)$  l'espace des fonctions de  $X_h$  et s'annulant en  $x = \pm 1$ .

Nous considérons le problème approché suivant :

(FV)<sub>h</sub> : trouver  $u_h \in V_{0,h}$  telle que

$$a(u_h, v_h) = m(f, v_h), \quad \forall v_h \in V_{0,h}. \quad (3.5)$$

L'espace  $X_h$  est de dimension finie :  $\dim V_h = n + 1$ . Nous pouvons choisir pour base de  $X_h$  les fonctions  $(\varphi_i)_{0 \leq i \leq n}$  valant un au  $i$ -ème sommet  $x_i$  et zéro aux autres sommets (voir Fig. 3.1). L'opérateur d'interpolation de lagrange  $\pi_h$  correspond à la fonction `interpolate` et est donnée par :

## interpolate\_1d.h

```

template <class Array, class Function>
Array interpolate (const Array& x, Function v) {
    Array vh (x.size());
    for (unsigned int i = 0; i < x.size(); i++)
        vh [i] = v (x [i]);
    return vh;
}

```

Décomposons  $u_h$  dans cette base :

$$u_h(x) = \sum_{j=0}^n u_j, \varphi_j(x)$$

En choisissant  $v_h = \varphi_i$  dans (3.5), la formulation variationnelle conduit à :

$$\sum_{j=0}^n a(\varphi_i, \varphi_j) u_j = \sum_{j=0}^n m(\varphi_i, \varphi_j) f(x_j), \quad 1 \leq i \leq n-1,$$

$$u_0 = u_n = 0.$$

Cependant, il est difficile d'évaluer le second membre de façon exacte pour une fonction  $f$  générale. Aussi, nous approchons les intégrales apparaissant dans les formes bilinéaire par la formule de quadrature des trapèzes :

$$\int_{x_{i-1}}^{x_i} f(x) dx \approx h_i \frac{f(x_{i-1}) + f(x_i)}{2}$$

Ceci nous conduit à introduire le produit scalaire discret :

$$(u, v)_h = \sum_{i=1}^n h_i \frac{u(x_{i-1})v(x_{i-1}) + u(x_i)v(x_i)}{2} \quad \forall u, v \in L^2(I),$$

et à définir les formes bilinéaires  $m(\cdot, \cdot)$  et  $a_h(\cdot, \cdot)$  approchées :

$$m_h(u, v) = (u, v)_h, \quad \forall u, v \in L^2(I),$$

$$a_h(u_h, v_h) = (u'_h, v'_h)_h, \quad \forall u_h, v_h \in X_h,$$

Nous considérons alors le nouveau problème approché suivant :

$$(\widetilde{FV})_h : \text{trouver } \tilde{u}_h \in V_{0,h} \text{ telle que}$$

$$a_h(\tilde{u}_h, v_h) = m_h(f, v_h), \quad \forall v_h \in V_{0,h}. \quad (3.6)$$

Afin de ne pas allourdir les notations, nous omettrons par la suite les tildes, et noterons  $u_h$  la solution de (3.6). En choisissant  $v_h = \varphi_i$  dans (3.6), la formulation variationnelle (3.6) conduit à :

$$\sum_{j=0}^n a_h(\varphi_i, \varphi_j) \tilde{u}_j = \sum_{j=0}^n m_h(\varphi_i, \varphi_j) f(x_j), \quad 1 \leq i \leq n-1,$$

$$\tilde{u}_0 = \tilde{u}_n = 0.$$

D'une part, la formule est exacte pour les polynômes de degré un, et, d'autre part,  $u'_h v'_h$  est constant par intervalle pour  $u_h$  et  $v_h \in X_h$ , si bien que

$$a_h(u_h, v_h) = a(u_h, v_h), \quad \forall u_h, v_h \in X_h.$$

Il ne reste plus qu'à calculer les coefficients des matrices  $A = (a_h(\varphi_i, \varphi_j))_{1 \leq i, j \leq n-1}$  et  $M = (m_h(\varphi_i, \varphi_j))_{1 \leq i, j \leq n-1}$ . La matrice  $M$  est clairement diagonale :

$$M = \begin{pmatrix} \frac{h_0 + h_1}{2} & 0 & \dots & 0 & 0 \\ 0 & \frac{h_1 + h_2}{2} & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{h_{n-2} + h_{n-1}}{2} & 0 \\ 0 & 0 & \dots & 0 & \frac{h_{n-1} + h_n}{2} \end{pmatrix}$$

Ceci conduit à la fonction :

`fd_mass_1d.h`

```
template <class T>
void fd_mass_1d (const std::valarray<T>& x, std::valarray<T>& m) {
    for (size_t i = 0; i < x.size()-2; i++)
        m [i] = (x[i+2]-x[i])/2;
}
```

Un calcul élémentaire donne :

$$\varphi_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}} & \text{si } i \geq 1 & \text{et } x \in ]x_{i-1}, x_i[, \\ -\frac{x_{i+1} - x}{x_{i+1} - x_i} & \text{si } i \leq n-1 & \text{et } x \in ]x_i, x_{i+1}[, \\ 0 & \text{sinon.} \end{cases}$$

ainsi que

$$\varphi'_i(x) = \begin{cases} 1/h_{i-1} & \text{si } i \geq 1 & \text{et } x \in ]x_{i-1}, x_i[, \\ -1/h_{i+1} & \text{si } i \leq n-1 & \text{et } x \in ]x_i, x_{i+1}[, \\ 0 & \text{sinon.} \end{cases}$$

et finalement :

$$A = \begin{pmatrix} \frac{1}{h_0} + \frac{1}{h_1} & -\frac{1}{h_0} & \dots & 0 & 0 \\ -\frac{1}{h_1} & \frac{1}{h_1} + \frac{1}{h_2} & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{1}{h_{n-3}} + \frac{1}{h_{n-2}} & -\frac{1}{h_{n-2}} \\ 0 & 0 & \dots & -\frac{1}{h_{n-2}} & \frac{1}{h_{n-1}} + \frac{1}{h_{n-2}} \end{pmatrix}$$

La fonction suivante initialise la matrice  $A$  :

## fd\_energy\_1d.h

```

#include <valarray>
template <class T>
void fd_energy_1d (const std::valarray<T>& x, std::valarray<T>& D, std::valarray<T>& L) {
    for (size_t i = 0; i < x.size()-2; i++)
        D [i] = 1/(x[i+1]-x[i]) + 1/(x[i+2]-x[i+1]);
    for (size_t i = 0; i < x.size()-3; i++)
        L [i] = - 1/(x[i+2]-x[i+1]);
    L [x.size()-3] = 0;
}

```

Les tableaux `d` et `l`, sont la diagonale, taille  $n - 1$ , et la sous-diagonale, de taille  $n - 2$ . Pour des raisons de commodité, le tableau `l` sera également de taille de  $n - 1$ , avec comme convention `l [l.size()-1] == 0`.

Remarquons que le problème s'interprète comme un schéma aux différences finies :

$$\begin{aligned}
 -\frac{u_{i-1}}{h_{i-1}} + \left( \frac{1}{h_i} + \frac{1}{h_{i-1}} \right) u_i - \frac{u_{i+1}}{h_i} &= \frac{h_i + h_{i-1}}{2} f(x_i), & 1 \leq i \leq n-1 \\
 u_0 &= u_n = 0
 \end{aligned}$$

En particulier, lorsque la subdivision est choisie uniforme, nous retrouvons le schéma aux différences finies (3.2), page 15. Ainsi la méthode des éléments finis apparaît comme une généralisation de la méthode des différences finies.

Nous pourrions à présent écrire un programme, comme nous l'avons fait dans le cas d'une subdivision régulière. Cependant, le code perdrait en lisibilité, du fait des tableaux (`d,l`) de la matrice tridiagonale. ce code deviendrait difficilement réutilisable pour d'autres applications.

Nous allons voir dans la suite :

- comment rendre le code plus agréable à lire, et donc à maintenir ;
- comment le code sera réutilisable pour en faire un préconditionneur d'un problème plus complexe, dans le cadre des méthodes spectrales. (voir tome 2 du livre).

Pour cela, nous allons construire une classe regroupant les tableaux (`d,l`) et gérant la factorisation et la résolution.

### 3.3 La classe des matrices tridiagonales symmétriques

strid.h

---

```

#include <valarray>
#include <iostream>
template <class T>
class strid {
public :
    typedef T value_type;
    strid (size_t = 0);
    strid (const strid<T>&);
    strid<T>& operator= (const strid<T>&);
    const std : :valarray<T>& diag () const;
    const std : :valarray<T>& tril () const;
    std : :valarray<T>& diag ();
    std : :valarray<T>& tril ();
    std : :valarray<T> operator* (const std : :valarray<T>& x) const;
    void factorize ();
    std : :valarray<T> solve (const std : :valarray<T>&) const;
protected :
    std : :valarray<T> _d;
    std : :valarray<T> _l;
};
template <class T> void factorize (const strid<T>& a, strid<T>& fact);

```

Pour coder le produit matrice-vecteur, utilisons la fonctions `shift` de la classe `valarray<T>`, qui réalise un décalage logique (voir section 2.2, page 2.2).

```

template <class T>
inline std : :valarray<T> strid<T> : :operator* (const std : :valarray<T>& x) const {
    return _l.shift(-1)*x.shift(-1) + _d*x + _l*x.shift(1);
}

```

Cette façon d'utiliser les décalages demande en contrepartie que les deux tableaux `d` et `l` aient la même longueur, avec comme convention `l[l.size()-1] == 0`. La factorisation  $LDL^T$  fait appel aux fonctions précédentes :

```

#include "ldlt_1d.h"
template <class T> inline void strid<T> : :factorize () { ldlt_1d (_d, _l); }
template <class T> inline void factorize (const strid<T>& a, strid<T>& fact) {
    fact = a;
    fact.factorize ();
}
template <class T>
inline std : :valarray<T> strid<T> : :solve (const std : :valarray<T>& b) const {
    std : :valarray<T> x = b;
    ldlt_solve_1d (_d, _l, x);
    return x;
}

```

L'implémentation des autres fonctions membres est assez classique. Les constructeurs de copie et d'affectation doivent être explicitement spécifiés, car la classe `strid<T>` contient deux `valarray<T>` qui utilisent l'allocation dynamique de mémoire.

```

template <class T> inline strid<T> : :strid (size_t n) : _d(n), _l(n) {}
template <class T> inline strid<T> : :strid (const strid<T>& a) : _d(a._d), _l(a._l) {}
template <class T> inline strid<T>& strid<T> : :operator= (const strid<T>& a) {
    _d.resize (a._d.size());
    _l.resize (a._l.size());
    _d = a._d;
    _l = a._l;
    return *this;
}

```

Enfin, les accès en lecture et écriture aux données sont immédiats :

```

template <class T> inline const std : :valarray<T>& strid<T> : :diag () const { return _d; }
template <class T> inline const std : :valarray<T>& strid<T> : :tril () const { return _l; }
template <class T> inline std : :valarray<T>& strid<T> : :diag () { return _d; }
template <class T> inline std : :valarray<T>& strid<T> : :tril () { return _l; }

```

L'assemblage de la matrice d'énergie devient :

#### fd\_energy\_1d.h (suite)

```

#include "strid.h"
template <class T>
inline void fd_energy_1d (const std : :valarray<T>& x, strid<T>& a) {
    fd_energy_1d (x, a.diag(), a.tril());
}

```

Une façon commode d'imbriquer les inclusions des fichiers d'entête, sans repasser deux fois sur les mêmes déclarations, est de les encadrer par des directives du pré-processeur compilateur. Par exemple :

```

#ifndef _HAVE_STRID_H
#define _HAVE_STRID_H
template <class T>
class strid {
    // ...
};
#endif // _HAVE_STRID_H

```

Le paragraphe suivant illustre l'utilisation de la classe `strid` et des subdivisions non-uniformes.



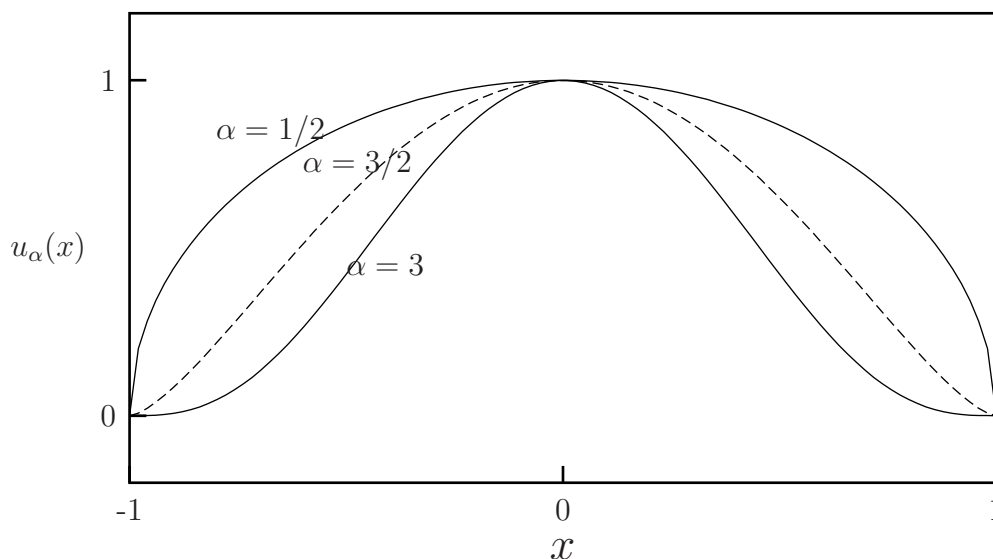


FIG. 3.2 – Aspect de la solution  $u_\alpha(x)$  sur  $[-1, 1]$  suivant  $\alpha$ .

### 3.4 De l'intérêt des subdivisions non-uniformes

`u_exact_singular.h`

```

template <class T> struct u_fct {
    T operator() (const T& x) const { return pow (1-x*x, a); }
    u_fct (const T& alpha) : a(alpha){}
    T a;
};
template <class T> inline struct u_fct<T> u (const T& a) { return u_fct<T>(a); }

template <class T> struct f_fct {
    T operator() (const T& x) const {
        return (x*x == 1) ? 0 : 2*a*(pow (1-x*x, a-1) - 2*(a-1)*x*x*pow (1-x*x, a-2)); }
    f_fct (const T& alpha) : a(alpha){}
    T a;
};
template <class T> inline struct f_fct<T> f (const T& a) { return f_fct<T>(a); }

template <class T> inline
T chi (const T& b, const T& t) { return (b*t - 1 - pow (t, b) + pow(1-t, b))/(b-2); }

```

Ce code a pour but l'étude d'une solution de la forme :

$$u_\alpha(x) = (1 - x^2)^\alpha,$$

avec  $\alpha > 1/2$ . La solution est peu régulière en  $x = \pm 1$  (voir figure 3.2). Un rapide calcul permet de préciser ce manque de régularité :  $u_\alpha \in H^{\alpha+1/2-\varepsilon}(I)$ , pour tout  $\varepsilon > 0$ , mais

$u_\alpha \notin H^{\alpha+1/2}(I)$ . En effet, la dérivée d'ordre  $s > 0$ ,  $u^{(s)}$  est en  $(1+x)^{\alpha-s}$  au voisinage de  $x = -1$ . Cette dérivée n'est de carré sommable dans ce voisinage que pour  $s < \alpha + 1/2$ .

D'autre part, pour  $h > 0$ , et sur une subdivision uniforme, l'erreur d'interpolation  $|u - \pi_h u|$  en  $x = -1 + h$  varie comme :

$$h^2 |u''(-1 + h)| \approx h^2 \times h^{\alpha-2} = h^\alpha.$$

Ainsi, l'erreur dans ce voisinage est majorée par un terme en  $h^\alpha$ . Loin des bords, l'erreur est en  $h^2$ , si bien que le mieux que nous puissions avoir sur une subdivision uniforme est  $\mathcal{O}(h^{\min(\alpha, 2)})$ .

Afin de resserrer la subdivision près des bords, cherchons une distribution des sommets de la forme  $x_i = 2\chi(i/n) - 1$ ,  $0 \leq i \leq n$  avec :

$$\chi(t) = \frac{\beta t - 1 + (1-t)^\beta - t^\beta}{\beta - 2},$$

et où  $\beta \geq 1$ . Nous retrouvons la subdivision uniforme pour  $\beta = 1$ . Le pas de la subdivision  $h_i = x_{i+1} - x_i$  évolue comme  $\chi'(i/n)$ . Ainsi, l'erreur d'interpolation  $h_i^2 |u''(x_i)|$  au voisinage de  $x = -1$  va se comporter comme  $\chi'(t)^2 \chi(t)^{\alpha-2}$  au voisinage de  $t = 0$ .

$$\chi'(t)^2 \chi(t)^{\alpha-2} \approx t^{\alpha\beta-2} \quad \text{au voisinage de } t = 0.$$

Cette erreur sera équirépartie dans ce voisinage si  $\alpha\beta - 2 = 0$ , c'est-à-dire  $\beta = 2/\alpha$ . Nous appellerons *subdivision adaptée* la subdivision obtenue.

## fd\_adapt\_tst.cc

```

#include "fd_energy_1d.h"
#include "fd_mass_1d.h"
#include "interpolate_1d.h"
#include "u_exact_singular.h"
#include "range.h"
#include "valarray_util.h"
#include <iostream>
using namespace std;
int main (int argc, char** argv) {
    size_t n = (argc > 1)? atoi(argv[1]) : 11;
    double alpha = (argc > 3)? atof(argv[3]) : 0.5;
    double beta = (argc > 2)? atof(argv[2]) : 2/alpha;
    valarray<double> x (n+1);
    for (size_t i = 0; i < n+1; i++)
        x [i] = -1 + 2*chi (beta, 1.0*i/n);
    strid<double> a (n-1);
    valarray<double> m (n-1);
    fd_energy_1d (x, a);
    fd_mass_1d (x, m);
    strid<double> fact (n-1);
    factorize (a, fact);
    valarray<double> pi_h_f = interpolate (x, f(alpha));
    valarray<double> b = m * valarray<double>(pi_h_f [range(1,n-1)]);
    valarray<double> uh (0.0, n+1);
    uh [range(1,n-1)] = fact.solve (b);
    valarray<double> pi_h_u = interpolate (x, u(alpha));
    valarray<double> u_err = valarray<double>(pi_h_u - uh)[range(1,n-1)];
    valarray<double> uerr_linf = abs(u_err);
    cerr << "alpha  " << alpha << endl
         << "beta   " << beta << endl
         << "n      " << n << endl
         << "err_l2 " << sqrt(dot (u_err, m*u_err)) << endl
         << "err_linf " << uerr_linf.max() << endl
         << "err_h1 " << sqrt(dot (u_err, a*u_err)) << endl;
}

```

La compilation et l'exécution du code sont :

```

c++ fd_adapt_tst.cc -o fd_adapt_tst
fd_adapt_tst 1000 0.5 1
fd_adapt_tst 1000 0.5 4

```

Le premier argument de la ligne d'appel est  $n$ . Les suivants sont  $\alpha$  et  $\beta$ . Ces arguments sont optionnels.

La figure 3.3 compare les solutions approchées sur maillages uniformes et adaptés ainsi que la solution exacte, ceci pour  $n = 40$  et  $\alpha = 1/2$ . Le maximum de l'erreur, obtenu en  $x = 0$ , est de 12 % sans adaptation et tombe à 2 % avec un maillage adapté. Ainsi, pour

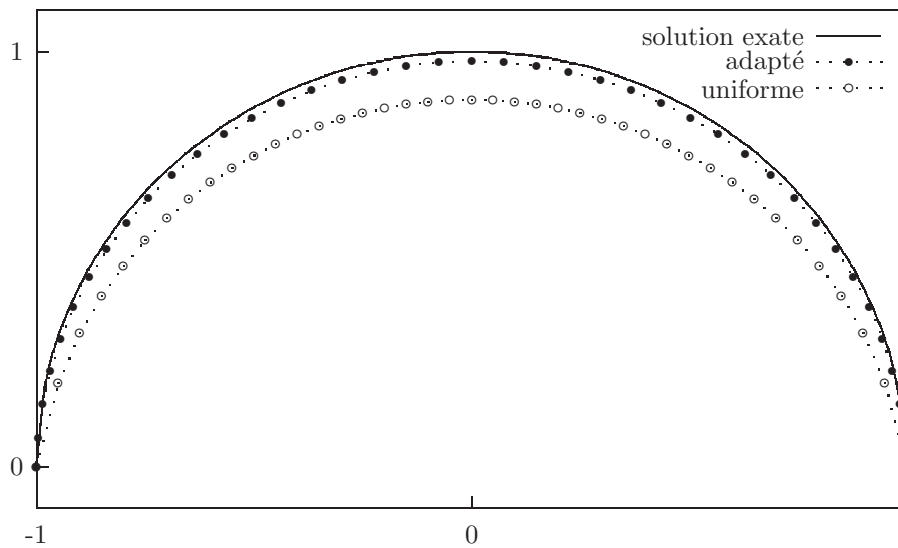


FIG. 3.3 – Approximation sur les subdivisions uniformes et adaptées ( $n = 40$ ,  $\alpha = 1/2$ ).

un même nombre de points de la subdivision, la solution approchée sur maillage adapté est nettement plus précise. Cet écart se creuse encore plus lorsque  $n$  augmente. La figure 3.4 présente, pour  $\alpha = 1/2$ , l'erreur  $\|u_h - \pi_h u\|_{L^\infty}$  selon  $n$  pour une subdivision uniforme ( $\beta = 1$ ) et adaptée ( $\beta = 4$ ). Dans le cas uniforme, l'erreur est en  $\mathcal{O}(h^{1/2}) = \mathcal{O}(n^{-1/2})$ . Dans le cas adapté, l'erreur est en  $\mathcal{O}(n^{-1})$  : la convergence est donc bien plus rapide. La norme  $L^2$  de l'erreur a un comportement similaire. Enfin, pour  $\alpha = 1/2$ , l'erreur ne converge pas en norme  $H^1$  car la solution n'appartient pas à cet espace.

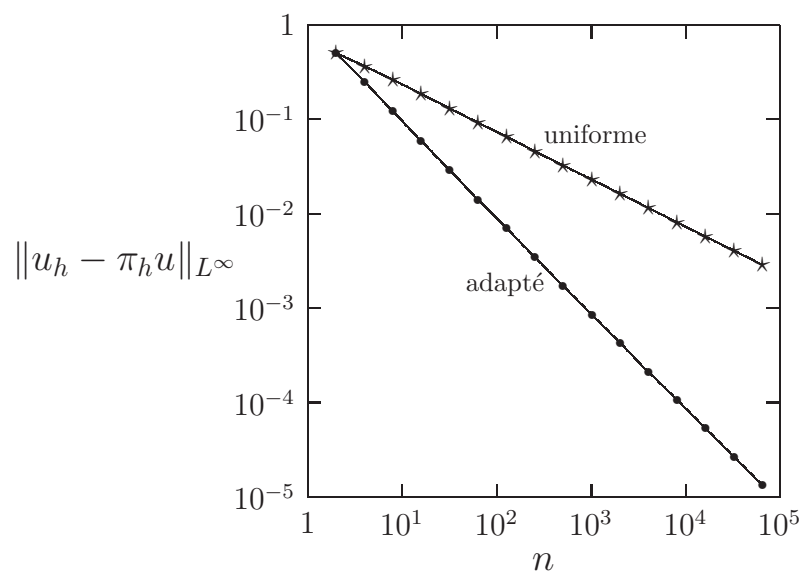


FIG. 3.4 – Convergence vers  $u(x) = (1 - x^2)^{1/2}$  pour les subdivisions uniformes et adaptées.

# Chapitre 4

## Problème de POISSON en dimension deux

### 4.1 Les différences finies en grilles régulières

Soit  $\Omega$  une région bornée du plan  $\mathbb{R}^2$ , de frontière  $\partial\Omega$  régulière. Considérons le problème suivant :

( $Q$ ) : trouver  $u$ , définie de  $\Omega$  dans  $\mathbb{R}$ , telle que

$$\begin{aligned} -\Delta u &= f \text{ dans } \Omega \\ u &= 0 \text{ sur } \partial\Omega \end{aligned}$$

où  $f$  est une fonction donnée, de  $\Omega$  dans  $\mathbb{R}$ .

Résoudre ce problème permet par exemple de connaître la déformation  $u$  d'une membrane subissant la charge  $f$ , ou bien la vitesse  $u$  d'un fluide en écoulement établi dans une section de forme  $\Omega$  sous la poussée  $f$ .

La solution du problème ( $Q$ ) n'est généralement pas connue explicitement sauf pour quelques cas très particuliers. Dans le cas où  $\Omega$  est rectangulaire, nous allons approcher la solution  $u$  par la technique des différences finies. Sans perte de généralité, nous pouvons alors supposer  $\Omega = ]-1, 1[^2$ . Le problème s'énonce :

( $Q$ ) <sub>$h$</sub>  : trouver  $(u_{i,j})_{0 \leq i,j \leq n}$  telle que

$$-u_{i,j+1} - u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j-1} = h^2 f(x_i, x_j), \text{ pour } 1 \leq i, j \leq n-1 \quad (4.1)$$

$$u_{i,0} = u_{i,n+1} = 0, \text{ pour } 0 \leq i \leq n \quad (4.2)$$

$$u_{0,j} = u_{n+1,j} = 0, \text{ pour } 1 \leq j \leq n-1 \quad (4.3)$$

où  $h = 1/n$  et  $x_i = ih$ ,  $0 \leq i \leq n$ . Numérotons les inconnues  $(u_{i,j})_{1 \leq i,j \leq n-1}$  par ligne d'abord, puis par colonne ensuite :  $\mathcal{I}(i, j) = (n-1) * (j-1) + i - 1 \in [0, N-1]$  pour

$1 \leq i, j \leq n$ , avec  $\mathbf{N} = (n - 1)^2$  la taille du système. L'interpolée  $\pi_h v$  d'une fonction  $v$  sur la grille s'obtient par :

interpolate\_2d.h

---

```
#include <valarray>
template <class T>
std : :valarray<T> interpolate (size_t n, T (*v)(const T&, const T&)) {
    std : :valarray<T> pi_h_v ((n+1)*(n+1));
    T h = 2.0/n;
    for (size_t j = 0; j < n+1; j++)
        for (size_t i = 0; i < n+1; i++)
            pi_h_v [i+j*(n+1)] = v (-1+i*h, -1+j*h);
    return pi_h_v;
}
```

Le problème peut se mettre sous la forme d'un système linéaire de matrice

$$A = \begin{pmatrix} \tilde{C} & -I & & & \\ -I & \tilde{C} & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & \tilde{C} & -I \\ & & & -I & \tilde{C} \end{pmatrix}$$

où  $I$  la matrice identité d'ordre  $n - 1$  et  $\tilde{C} = 4.I - C_0$  est la matrice, d'ordre  $n - 1$ , avec :

$$C_0 = \begin{pmatrix} 0 & 1 & & & \\ 1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 1 \\ & & & & 1 & 0 \end{pmatrix}$$

Nous allons à présent étudier deux façon de résoudre ce système linéaire. La première méthode est itérative, non optimale, mais assez simple à programmer et très courante d'utilisation : il s'agit de la méthode du gradient conjugué préconditionné. Ensuite nous étudierons une méthode directe de résolution, de plus quasi-optimale (très rapide), quoique plus compliquée à programmer.

## 4.2 Une résolution itérative du problème

### 4.2.1 La méthode du gradient conjugué

Afin de calculer les termes  $u_{i+1,j}$  et  $u_{i-1,j}$ , le produit matrice-vecteur utilisera la classe `gslice` [Str01, p. 754] de la librairie standard C++, et qui permettent de décaler les indices dans ces tableaux de type `valarray`.

```

class finite_differences_2d {
public :
    finite_differences_2d (size_t n);
    template <class T>
        std::valarray<T> operator* (const std::valarray<T>& u) const;
protected :
    size_t _n;
    std::gslice _shift_left;
    std::gslice _shift_right;
};

```

Le codage de cette classe est :

```

finite_differences_2d::finite_differences_2d (size_t n) : _n(n) {
    std::valarray<size_t> length (2);
    length [0] = n-1;
    length [1] = n;
    std::valarray<size_t> stride (2);
    stride [0] = 1;
    stride [1] = n;
    _shift_left = std::gslice (0, length, stride);
    _shift_right = std::gslice (1, length, stride);
}
template <class T>
std::valarray<T> finite_differences_2d::operator* (const std::valarray<T>& u) const {
    // rhs(i,j) := -u(i,j-1) + 4*u(i,j) - u(i,j+1)
    std::valarray<T> rhs = - u.shift(-_n) + 4.0*u - u.shift(_n);
    // rhs(i,j) -= u(i+1,j)
    rhs [_shift_left] -= u [_shift_right]; // u(i+1,j)
    // rhs(i,j) -= u(i-1,j)
    rhs [_shift_right] -= u [_shift_left]; // u(i-1,j)
    return rhs;
}

```

Avec pour second membre  $f(x, y) = x(1 - x) + y(1 - y)$ , la solution exacte est connue explicitement  $u(x, y) = x(1 - x)y(1 - y)/2$ , ce qui permet de calculer l'erreur. D'où le programme de test :



---

 poisson\_2d\_tst.cc
 

---

```

// dans ]-1,1[ avec n*n elements
#include "finite_differences_2d.h"
#include "fd_ssor_2d.h"
#include "eye.h"
#include "cg.h"
#include "valarray_util.h"
#include "sqr.h"
#include <iostream>
using namespace std;
double u (const double& x, const double& y) { return (1-sqr(x))*(1-sqr(y))/2; }
double f (const double& x, const double& y) { return 2-sqr(x)-sqr(y); }

int main (int argc, char** argv) {
    size_t n = (argc > 1)? atoi(argv[1]) : 10;
    double h = 2.0/n;
    valarray<double> pi_h_u = interpolate (n, u);
    valarray<double> pi_h_f = interpolate (n, f);
    valarray<double> b ((n-1)*(n-1));
    for (size_t i = 0; i < n-1; i++)
        for (size_t j = 0; j < n-1; j++)
            b [i+(n-1)*j] = h*h*pi_h_f [(i+1)+(n+1)*(j+1)];
    valarray<double> x (0.0, (n-1)*(n-1));
    finite_differences_2d A (n-1);
    cg (A, x, b, eye(), 2*sqr(n-1), 1e-17);
    valarray<double> uh (0.0, (n+1)*(n+1));
    for (size_t i = 0; i < n-1; i++)
        for (size_t j = 0; j < n-1; j++)
            uh [(i+1)+(n+1)*(j+1)] = x [i+(n-1)*j];
    double err = abs(pi_h_u-uh).max();
    cerr << "err " << err << endl;
    return err > 1e-7? 1 : 0;
}

```

a compilation et les test sont donnés par :

```

c++ poisson_2d_tst.cc -o poisson_2d_tst
poisson_2d_tst 5
poisson_2d_tst 10
poisson_2d_tst 20
...

```

Ici encore, la solution exacte étant polynômiale de degré deux par rapport à chaque variable, la solution approché  $u_{i,j}$  coïncide avec l'interpolée de la solution exacte sur la grille.

## 4.2.2 Préconditionnement

Le préconditionneur SSOR d'EVAN s'écrit de la façon suivante :

## fd\_ssor\_2d.h

---

```

#include <valarray>
struct fd_ssor_2d {
    template <class T>
    std::valarray<T> solve (const std::valarray<T>& b) const;
    fd_ssor_2d (size_t n);
    size_t n;
};
inline fd_ssor_2d::fd_ssor_2d (size_t n1) : n(n1) {}

template <class T>
std::valarray<T> fd_ssor_2d::solve (const std::valarray<T>& b) const {
    std::valarray<T> x (b.size());
    x[0] = b[0];
    for (size_t i = 1; i < n; i++)
        x[i] = b[i] + x[i-1]/2;
    for (size_t j = 1; j < n; j++) {
        size_t nj = n*j;
        size_t nj1 = n*(j-1);
        x[nj] = b[nj] + x[nj1]/2;
        for (size_t i = 1; i < n; i++)
            x[i+nj] = b[i+nj] + (x[i+nj1] + x[i-1+nj])/2;
    }
    size_t nj = n*(n-1);
    x[n-1+nj] /= 2;
    for (size_t i = n-1; i > 0; i--)
        x[i-1+nj] = (x[i-1+nj] + x[i+nj])/2;
    for (size_t j = n-1; j > 0; j--) {
        x[n-1+n*(j-1)] = (x[n-1+n*(j-1)] + x[n-1+n*j])/2;
        for (size_t i = n-1; i > 0; i--)
            x[i-1+n*(j-1)] = (x[i-1+n*(j-1)] + x[i+n*(j-1)] + x[i-1+n*j])/2;
    }
    return x;
}

```

## poisson\_2d\_pcg\_tst.cc

```

// dans ]-1,1[ avec n*n elements
#include "finite_differences_2d.h"
#include "fd_ssor_2d.h"
#include "cg.h"
#include "valarray_util.h"
#include "sqr.h"
#include <iostream>
using namespace std;
double u (const double& x, const double& y) { return (1-sqr(x))*(1-sqr(y))/2; }
double f (const double& x, const double& y) { return 2-sqr(x)-sqr(y); }

int main (int argc, char** argv) {
    size_t n = (argc > 1)? atoi(argv[1]) : 10;
    double h = 2.0/n;
    valarray<double> pi_h_u = interpolate (n, u);
    valarray<double> pi_h_f = interpolate (n, f);
    valarray<double> b ((n-1)*(n-1));
    for (size_t i = 0; i < n-1; i++)
        for (size_t j = 0; j < n-1; j++)
            b [i+(n-1)*j] = h*h*pi_h_f [(i+1)+(n+1)*(j+1)];
    valarray<double> x (0.0, (n-1)*(n-1));
    finite_differences_2d A (n-1);
    fd_ssor_2d m (n-1);
    cg (A, x, b, m, 2*sqr(n-1), 1e-17);
    valarray<double> uh (0.0, (n+1)*(n+1));
    for (size_t i = 0; i < n-1; i++)
        for (size_t j = 0; j < n-1; j++)
            uh [(i+1)+(n+1)*(j+1)] = x [i+(n-1)*j];
    double err = abs(pi_h_u-uh).max();
    cerr << "err " << err << endl;
    return err > 1e-7? 1 : 0;
}

```

Soit  $r(n, i)$  le résidu de l'algorithme du gradient conjugué préconditionné à l'itération  $i$  pour le problème de POISSON sur une grille  $n \times n$ . Nous observons sur la figure 4.1 que ce résidu décroît suivant :

$$r(n, i) \approx c_n e^{-a_n i}$$

où  $a_n$  et  $c_n$  sont des constantes dépendant de  $n$ . Soit  $\varepsilon > 0$ . Le nombre  $\mathcal{N}(n, \varepsilon)$  d'itération minimum nécessaire pour obtenir  $r(n, i) < \varepsilon$  est alors :

$$\mathcal{N}(n, \varepsilon) = \frac{\log(c_n \varepsilon)}{a_n}$$

Or, nous observons sur la figure 4.2 que pour  $n$  grand :

$$\mathcal{N}(n, \varepsilon) \approx k(\varepsilon) n^{2/3}$$

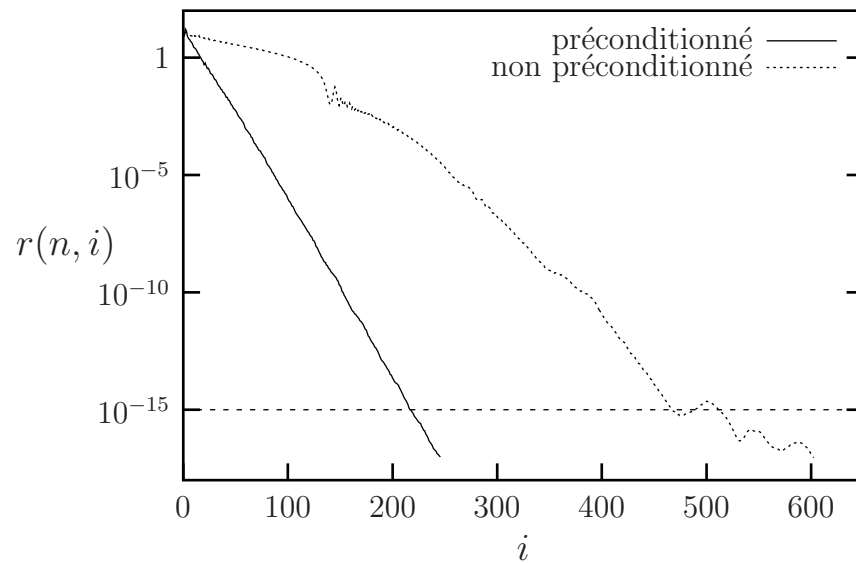


FIG. 4.1 – Gradient conjugué : effet du préconditionnement sur la convergence du résidu du problème de POISSON pendant les itérations (grille  $200 \times 200$  : 40 000 inconnues).

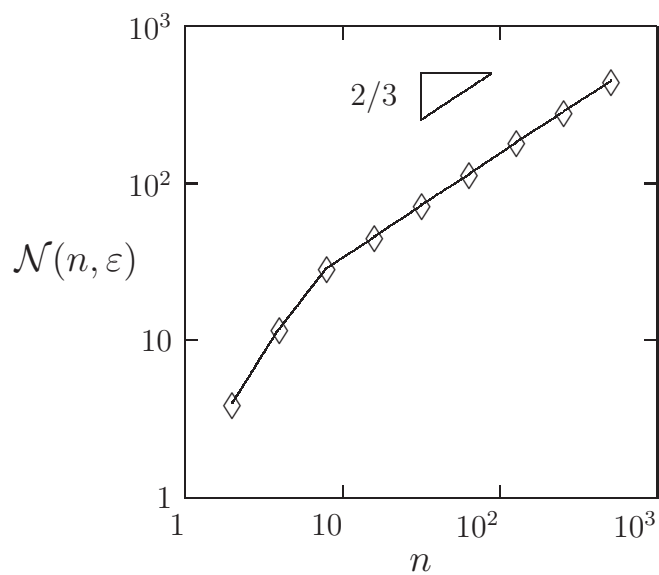


FIG. 4.2 – Gradient conjugué préconditionné : nombre d'itérations  $\mathcal{N}(n, \varepsilon)$  nécessaire, sur une grille  $n \times n$ , pour obtenir la précision  $\varepsilon = 10^{-15}$ .

En identifiant les deux relations précédentes, il vient  $a_n = a^* n^{2/3}$ ,  $k(\varepsilon) = k^* |\log \varepsilon|$ , et  $c_n = c^*$  est donc indépendante de  $n$ . D'où, finalement :

$$\begin{aligned} r(n, i) &\approx c^* e^{-a^* n^{2/3} i} \\ \mathcal{N}(n, \varepsilon) &\approx k^* |\log \varepsilon| n^{2/3} \end{aligned}$$

Or, le coût d'une itération du gradient conjugué est  $\mathcal{O}(n^2)$  opérations. Le coût total de la résolution est  $\mathcal{O}(\log \varepsilon n^{8/3})$ , soit encore  $\mathcal{O}(\log \varepsilon \mathbf{N}^{4/3})$  où  $\mathbf{N} = n^2$  est la taille du système linéaire.

## 4.3 Méthode directe par décomposition

### 4.3.1 Analyse de FOURIER

Cherchons les valeurs propres de la matrice  $\tilde{C}$ . Remarquons que matrice du problème en dimension un, apparaissant dans (3.3), page 16, est obtenue de  $\tilde{C}$  en remplaçant les 4 sur la diagonale par des 2 :  $C = 2.I - C_0$ .

PROPRIÉTÉ 4.3.1 *La matrice d'ordre  $n - 1$*

$$\text{tridiag}_{n-1}(\alpha, \beta) = \begin{pmatrix} \alpha & \beta & & & \\ \beta & \alpha & \beta & & \\ & \ddots & \ddots & \ddots & \\ & & \beta & \alpha & \beta \\ & & & \beta & \alpha \end{pmatrix}$$

*a pour valeurs propres*

$$\lambda_k = \alpha + 2\beta \cos\left(\frac{k\pi}{n}\right), \quad 1 \leq k \leq n - 1$$

*et pour vecteurs propres unitaires associés :*

$$u_k = \sqrt{\frac{2}{n}} (\sin(kj\pi/n))_{1 \leq j \leq n-1}$$

*Démonstration :*  $\lambda$  est valeur propre de  $\text{tridiag}_{n-1}(\alpha, \beta)$  si et seulement si  $\det((\lambda - \alpha)I - \beta C_0) = 0$ . Autrement dit, si  $\mu$  est valeur propre de  $C_0$ , alors  $\lambda = \alpha + \beta\mu$  sera valeur propre de  $\text{tridiag}_{n-1}(\alpha, \beta)$ .

Introduisons le polynôme caractéristique de la matrice  $C_0$  d'ordre  $k \geq 2$  :

$$p_k(\mu) = \det(\mu - C_0) = \begin{pmatrix} \mu & -1 & & & \\ -1 & \mu & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & \mu & -1 \\ & & & -1 & \mu \end{pmatrix}.$$

Un développement du déterminant par rapport à la première colonne conduit à la formule de récurrence

$$p_k(\mu) = \mu p_{k-1}(\mu) - p_{k-2}(\mu), \quad k \geq 3.$$

D'autre part,

$$p_1(\mu) = \mu \quad \text{et} \quad p_2(\mu) = \mu^2 - 1.$$

Par commodité de notations, introduisons

$$p_{-1}(\mu) = 0 \quad \text{et} \quad p_0(\mu) = 1.$$

si bien que la formule de récurrence précédente est vérifiée dès que  $k \geq 1$ . Posons à présent :

$$\mu = 2\nu \quad \text{et} \quad q_k(\nu) = p_k(\mu), \quad k \geq -1.$$

La formule de récurrence change légèrement :

$$\begin{aligned} q_{-1}(\nu) &= 0, \\ q_0(\nu) &= 1, \\ q_k(\nu) &= 2\nu q_{k-1}(\nu) - q_{k-2}(\nu), \quad k \geq 1. \end{aligned}$$

Le facteur deux devant le premier terme du membre de droite fait toute la différence : nous reconnaissons là les polynômes de Tchebyschev. En effet, posons à présent  $\nu = \cos \varphi$ . D'une part, la formule de récurrence devient

$$q_k(\cos \varphi) = 2 \cos \varphi q_{k-1}(\cos \varphi) - q_{k-2}(\cos \varphi), \quad k \geq 1. \quad (4.4)$$

D'autre part, nous avons la formule trigonométrique :

$$\sin \{(k+1)\varphi\} = 2 \cos \varphi \sin(k\varphi) - \sin \{(k-1)\varphi\}.$$

Ceci nous conduit à poser

$$q_k(\nu) = \frac{\sin \{(k+1)\varphi\}}{\sin \varphi}.$$

Le dénominateur sert à assurer les conditions pour  $k = -1$  et  $k = 1$  :

$$\begin{aligned} q_{-1}(\nu) &= \frac{\sin 0}{\sin \varphi} = 0 \\ q_0(\nu) &= \frac{\sin \varphi}{\sin \varphi} = 1 \end{aligned}$$

Ainsi,  $\mu$  est valeur propre de  $C_0$  si et seulement si  $q_{n-1}(\nu) = 0$ , soit encore ;

$$\frac{\sin n\varphi}{\sin \varphi} = 0, \quad \text{avec } \varphi = \arccos \nu = \arccos \mu/2$$

Ceci équivaut à  $\varphi \notin \{0, \pi\}$  et  $\exists k \in \mathbb{Z}$  tel que  $n\varphi = k\pi$ . Les valeurs propre de  $C_0$ , d'ordre  $n - 1$ , sont donc données en ordre croissant par :

$$\mu = \mu_k \stackrel{\text{déf}}{=} 2 \cos \left( \frac{k\pi}{n} \right), \quad 1 \leq k \leq n - 1$$

Recherchons à présent les vecteurs propres de  $C_0$ . Remarquons que les  $n - 1$  premières relations de récurrences (4.3.1), avec  $k = 1, \dots, n-1$  s'écrivent aussi sous la forme : s'écrivent :

$$\mu \begin{pmatrix} p_0(\mu) \\ p_1(\mu) \\ \dots \\ p_{n-3}(\mu) \\ p_{n-2}(\mu) \end{pmatrix} = \begin{pmatrix} 0 & 1 & & & \\ 1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 1 \\ & & & & 1 & 0 \end{pmatrix} \begin{pmatrix} p_0(\mu) \\ p_1(\mu) \\ \dots \\ p_{n-3}(\mu) \\ p_{n-2}(\mu) \end{pmatrix} + p_{n-1}(\mu) \begin{pmatrix} 0 \\ 0 \\ \dots \\ 0 \\ 1 \end{pmatrix}$$

Posons  $v(\mu) = (p_0(\mu), \dots, p_{n-2}(\mu))^T$ . Si  $\mu = \mu_k$  est une racine de  $p_{n-1}$ , alors

$$\mu_k v(\mu_k) = C_0 v(\mu_k)$$

et donc  $v(\mu_k)$  est un vecteur propre associé. Nous avons

$$\|v(\mu_k)\|^2 = \frac{1}{\sin^2(k\pi/n)} \sum_{j=1}^{n-1} \sin^2(jk\pi/n).$$

*pourquoi ?* Or

$$\sum_{j=1}^{n-1} \sin^2(jk\pi/n) = \frac{n}{2}$$

d'où le résultat.  $\square$  En particulier, les valeurs propres de  $\tilde{C}$  sont donc  $\lambda_k = 4 - 2\mu_k = 4 - 2 \cos k\pi/n$ ,  $0 \leq k \leq n - 1$ . Introduisons la matrice unitaire  $U = (U_{k,l})_{1 \leq k, l \leq n-1}$  des vecteurs propres :

$$U_{k,l} = \left( \frac{2}{n} \right)^{1/2} \sin \left( \frac{kl\pi}{n} \right), \quad 1 \leq k, l \leq n - 1.$$

Ainsi

$$UCU^T = U^T C U = \Lambda,$$

où  $\Lambda$  désigne la matrice diagonale des valeurs propres  $\lambda_k$ ,  $1 \leq k \leq n - 1$ .

Revenons à présent au système de POISSON (4.1)-(4.3), page 29. Ce système peut s'écrire par blocs :

$$\begin{cases} C u_1 - u_2 = f_1, \\ -u_{j-1} + C u_j - u_{j+1} = f_j, & 2 \leq j \leq n - 2, \\ -u_{n-2} + C u_{n-1} = f_{n-1}, \end{cases}$$

avec  $u_j = (u_{k,j})_{1 \leq k \leq n-1}$  et  $f_j = (f_{k,j})_{1 \leq k \leq n-1}$ . Passons dans la base associée aux vecteurs propres :

$$u_j = U\tilde{u}_j \quad \text{et} \quad f_j = U\tilde{f}_j, \quad 1 \leq j \leq n-1.$$

Alors le système devient :

$$\begin{cases} \Lambda\tilde{u}_1 & - \tilde{u}_2 & = \tilde{f}_1, \\ -\tilde{u}_{j-1} + \Lambda\tilde{u}_j & - \tilde{u}_{j+1} & = \tilde{f}_j, & 2 \leq j \leq n-2, \\ -\tilde{u}_{n-2} + \Lambda\tilde{u}_{n-1} & & = \tilde{f}_{n-1}. \end{cases}$$

Redéveloppons à présent les blocs. Nous avons pour tout  $k \in \{1, \dots, n-1\}$  :

$$\begin{cases} \lambda_k\tilde{u}_{k,1} & - \tilde{u}_{k,2} & = \tilde{f}_{k,1}, \\ -\tilde{u}_{k,j-1} + \lambda_k\tilde{u}_{k,j} & - \tilde{u}_{k,j+1} & = \tilde{f}_{k,j}, & 2 \leq j \leq n-2, \\ -\tilde{u}_{k,n-2} + \lambda_k\tilde{u}_{k,n-1} & & = \tilde{f}_{k,n-1}. \end{cases}$$

Ainsi, ces  $n-1$  systèmes linéaires sont découplés. De plus, à  $k$  fixé, nous sommes en présence d'un système tridiagonal de matrice  $A_k = \lambda_k - C_0$ , que nous savons résoudre efficacement par une méthode directe (voir section 3.1.2, page 16).

**algorithme** POISSON bidimensionnel sur grille uniforme

**entrée**  $(f_{k,j})_{1 \leq k, j \leq n-1}$

**sortie**  $(f_{k,j})_{1 \leq k, j \leq n-1}$

**début**

**pour**  $j := 1 \dots n-1$

$$\tilde{f}_{k,j} := \frac{2}{n} \sum_{p=1}^{n-1} \sin\left(\frac{kp\pi}{n}\right) f_{p,j}, \quad k = 1 \dots n-1$$

**fin pour**  $j$

**pour**  $k := 1 \dots n-1$

$\lambda_k := 4 - 2 \cos k\pi/n$

trouver  $\tilde{u}_k = (\tilde{u}_{p,j})_{1 \leq p \leq n-1}$  tel que

$$(\lambda_k I - C_0)\tilde{u}_k = \tilde{f}_k$$

**fin pour**  $k$

**pour**  $j := 1 \dots n-1$

**pour**  $k := 1 \dots n-1$

$$u_{k,j} := \sum_{p=1}^{n-1} \sin\left(\frac{kp\pi}{n}\right) \tilde{u}_{p,j}, \quad k = 1 \dots n-1$$

**fin pour**  $k$

**fin pour**  $j$

**fin algo**

Si nous faisons le calcul du temps de calcul, nous trouvons  $\mathcal{O}(n^3)$ . La méthode n'est *a priori* pas très attractive, car il n'y a que  $(n-1)^2$  inconnues. Nous allons voir que ce temps se ramène à  $\mathcal{O}(n^2 \log n)$  à condition d'utiliser la transformée de FOURIER rapide.



La classe assurant la résolution sera :

---

poisson\_2d\_sinus.h

---

```
#include <valarray>
class poisson_2d_sinus {
public :
    poisson_2d_sinus (size_t n);
    template <class T> void solve (const std : :valarray<T>& b, std : :valarray<T>& u) const ;
    template <class T> std : :valarray<T> solve (const std : :valarray<T>& b) const ;
protected :
    size_t n;
};
```

L'implémentation des fonctions membres est :

```

#include "range.h"
#include "sin_tranformation.h"
#include "numeric_constant.h"
#include "strid.h"
#include "valarray_util.h"

inline poisson_2d_sinus::poisson_2d_sinus (size_t n1) : n(n1) {}

template <class T>
void
poisson_2d_sinus::solve (const std::valarray<T>& b, std::valarray<T>& u) const
{
    col_type col (n);
    u.resize (n*n);
    std::valarray<T> b_hat (n*n);
    std::valarray<T> u_hat (n*n);
    T c = 2.0/(n+1);
    for (size_t j = 0; j < n; j++) {
        b_hat [col(j)] = c*sin_tranformation (b [col (j)]);
        std::valarray<T> err_j = sin_tranformation (b_hat [col (j)]) - b [col (j)];
    }
    const T pi = numeric_constant<T>::pi ();
    strid<T> poisson_1d (n);
    strid<T> factorized_1d (n);
    row_type row (n);
    for (size_t i = 0; i < n; i++) {
        poisson_1d.tril() = -1;
        poisson_1d.tril()[n-1] = 0;
        poisson_1d.diag() = 4 - 2*cos((i+1)*pi/(n+1));
        factorize (poisson_1d, factorized_1d);
        u_hat [row(i)] = factorized_1d.solve (b_hat [row(i)]);
        std::valarray<T> err_i = poisson_1d*u_hat [row(i)] - std::valarray<T>(b_hat [row(i)]);
    }
    for (size_t j = 0; j < n; j++) {
        u [col (j)] = sin_tranformation (u_hat [col(j)]);
        std::valarray<T> err_j = c*sin_tranformation (u [col (j)]) - std::valarray<T>(u_hat [col (j)]);
    }
}

template <class T>
inline std::valarray<T> poisson_2d_sinus::solve (const std::valarray<T>& b) const {
    std::valarray<T> u (n*n);
    solve (b, u);
    return u;
}

```

### 4.3.2 La transformée en sinus

Considérons la transformée en sinus :

$$\tilde{u}_k = \sum_{p=1}^{n-1} \sin\left(\frac{kp\pi}{n}\right) u_p, \quad k = 1 \dots n-1.$$

Étendons  $u_p$  à  $p = 2n$  en un échantillon périodique, impair à  $p = n$ , par :

$$\begin{aligned} u_{2n-p} &= -u_p, \quad p = 1 \dots n-1, \\ u_0 &= u_n = u_{2n} = 0, \end{aligned}$$

et observons la transformée de FOURIER :

$$\hat{u}_k = \sum_{p=0}^{2n-1} e^{2ikp\pi/(2n)} u_p, \quad k = 0 \dots 2n-1.$$

La moitié de cette somme, de  $p = n$  à  $2n-1$ , peut se réécrire avec  $p' = 2n-p$  :

$$\sum_{p=n}^{2n-1} e^{2ikp\pi/(2n)} u_p = \sum_{p'=1}^n e^{2ik(2n-p')\pi/(2n)} u_p = - \sum_{p'=1}^{n-1} e^{-2ikp'\pi/(2n)} u_p$$

Ainsi :

$$\begin{aligned} \hat{u}_k &= \sum_{p=0}^{n-1} \{e^{2ikp\pi/(2n)} - e^{-2ikp\pi/(2n)}\} u_p \\ &= 2i \sum_{p=1}^{n-1} \sin(kp\pi/n) u_p \\ &= 2i\tilde{u}_k \end{aligned}$$

Nous savons à présent calculer cette somme en un temps  $\mathcal{O}(n \log n)$ . Cependant, la partie réelle de la transformée de FOURIER est uniformément nulle, et donc la moitié du temps de calcul serait pris pour calculer des zéros !

Dans [PTVF94], paragraphe 12.4, les auteurs proposent une solution à ce délicat problème.

## 4.4 Exercices

### EXERCICE 4.4.1 (convergence)

On se place dans le cas de la dimension un.

1. Pour le second membre  $f(x) = \sin(\pi x)/\pi^2$ , la solution est  $u(x) = \sin(x)$ . Cette solution n'est pas polynômiale, et la solution calculée ne coïncide pas avec l'interpolée  $\pi_h u$  de la solution exacte.
2. Écrire la fonction `interpolate`, et interpoler  $u$  et  $f$ .
3. Calculer l'erreur  $\max_i |u_i - u(ih)|$  en faisant varier  $n$  suivant 5, 10, 20, ... 160.
4. Tracer l'erreur en fonction de  $h$  en diagramme log-log.
5. En déduire que l'erreur se comporte en  $\mathcal{O}(h^\alpha)$  et calculer  $\alpha$ .

#### EXERCICE 4.4.2 (dimension trois)

1. Écrire la classe `finite_difference_3d` pour la dimension trois.
2. Utiliser une solution polynômiale et valider ce calcul.
2. Utiliser une solution non polynômiale et calculer l'indice  $\alpha$  de convergence de la méthode, en procédant comme à l'exercice 4.4.1.

#### EXERCICE 4.4.3 (sortie graphique 1d)

Cet exercice a pour but de visualiser les solutions avec `gnuplot` [WKL<sup>+</sup>98].

1. Écrire la fonction `plot1d(n,uh)` qui crée un fichier `output.dat` contenant en première colonne  $x_i$  et en deuxième colonne  $u_i$ . Prendre soin d'ajouter les conditions aux bord : le fichier débutera par 0 0 et finira par 1 0.
2. Ajouter dans la fonction `plot1d(n,uh)` la création du fichier de lancement `output.plot` :  

```
plot "output.dat" with linespoints
pause -1 "<retour>"
```

et le lancement proprement dit par la fonction `system` du C++.
3. Effacer les fichiers temporaires en fin de fonction `plot1d`.

#### EXERCICE 4.4.4 (sortie graphique 2d)

Cet exercice a pour but de visualiser les solutions en dimension deux avec `gnuplot` [WKL<sup>+</sup>98]. S'inspirant de l'exercice 4.4.3, écrire la fonction `plot2d(n,uh)` en utilisant la commande `splot`. Utiliser l'option `set contour` de `gnuplot` pour visualiser également les iso-contours.

# Chapitre 5

## Matrices creuses

### 5.1 La structure de données

Pour aborder des système matrice-vecteurs généraux, nous devons cependant représenter les matrices en mémoire. Un cas particulier important représente les matrices creuses  $n \times m$ , pour lesquelles le nombre d'éléments non-nuls

$$\text{nnz}(A) = \text{card}\{(i, j) \in [1, n] \times [1, m]; A_{i,j} \neq 0\}$$

est petit devant le nombre total  $n \times m$  d'éléments de  $A$ . Ainsi, la matrice  $n \times n$  du paragraphe précédent contient seulement  $3n - 2$  éléments non-nuls, et est donc très creuse. Si on ne conserve que les éléments non-nuls en mémoire, le gain est important lorsque  $n$  devient grand : penser à  $n = 10^6$  !

Considérons l'exemple suivant :

$$A = \begin{pmatrix} 1 & & 2 & 3 & \\ & 4 & & & 5 \\ 6 & & 7 & 8 & \\ 9 & & & 10 & \\ & 11 & & & 12 \end{pmatrix}$$

Nous allons conserver les valeurs des coefficients non nuls dans un tableau `val` de taille `nnz`, ordonnés par indices de ligne puis de colonne croissants. Pour chaque coefficient non-nul, nous rangerons l'indice de colonne correspondant dans un tableau `idx` de taille également `nnz`. Enfin, `start[i]` sera l'indices de début de la  $i$ -ème ligne dans les tableaux précédents,  $0 \leq i \leq n - 1$ . Nous conviendrons de plus que `start[n] = nnz`, si bien que `start` est un tableau de taille  $n + 1$ . Pour la matrice précédente, nous avons :

val	1	2	3	4	5	6	7	8	9	10	11	12
-----	---	---	---	---	---	---	---	---	---	----	----	----

idx	0	2	3	1	4	0	2	3	0	3	2	4
-----	---	---	---	---	---	---	---	---	---	---	---	---

start	0	3	5	8	10	12
-------	---	---	---	---	----	----

Ainsi, le nombre de coefficients non-nuls dans la ligne  $i$  est  $\text{start}[i + 1] - \text{start}[i]$ . La déclaration de classe ressemble à :

---

matrix.h

---

```

#include <valarray>
#include <iostream>
template <class T>
class matrix {
public :
    matrix ();
    matrix (const matrix<T>&);
    matrix<T>& operator= (const matrix<T>&);
    size_t nrow () const;
    size_t ncol () const;
    size_t nnz () const;
    template <class U> friend std::istream& operator >> (std::istream&, matrix<U>&);
    template <class U> friend std::ostream& operator << (std::ostream&, const matrix<U>&);
    void resize (size_t nrow, size_t ncol, size_t nnz);
    std::valarray<T> operator* (const std::valarray<T>&) const;
protected :
    std::valarray<size_t> _start;
    std::valarray<size_t> _idx;
    std::valarray<T> _val;
    size_t _ncol;
};

```

Le constructeur par défaut, l'allocateur et les accès en lecture-écriture s'écrivent simplement :

```

template<class T>
inline matrix<T> : :matrix () : _start(0), _idx(0), _val(0), _ncol(0) { }

template<class T>
inline matrix<T> : :matrix (const matrix<T>& a)
    : _start(a._start), _idx(a._idx), _val(a._val), _ncol(a._ncol) { }

template<class T>
inline
matrix<T>&
matrix<T> : :operator= (const matrix& a) {
    resize (a.nrow(), a.ncol(), a.nnz());
    _start = a._start;
    _idx = a._idx;
    _val = a._val;
    _ncol = a._ncol;
    return *this;
}
template<class T>
inline void matrix<T> : :resize (size_t nrow, size_t ncol, size_t nnz) {
    _start.resize (nrow+1);
    _ncol = ncol;
    _idx.resize (nnz);
    _val.resize (nnz);
}
template<class T>
inline size_t matrix<T> : :nrow () const { return _start.size() - 1; }

template<class T>
inline size_t matrix<T> : :ncol () const { return _ncol; }

template<class T>
inline size_t matrix<T> : :nnz () const { return _idx.size(); }

```

Les entrée-sortie suivent :

```

#include <assert.h> // the assert(test) macro-function
template<class T>
std::ostream& operator << (std::ostream& os, const matrix<T>& a) {
    os << "%MatrixMarket\n"
        << a.nrow() << " " << a.ncol() << " " << a.nnz() << "\n";
    for (size_t i = 0; i < a.nrow(); i++)
        for (size_t p = a._start [i]; p < a._start [i+1]; p++)
            os << i+1 << " " << a._idx [p]+1 << " " << a._val [p] << "\n";
    return os;
}
template<class T>
std::istream& operator >> (std::istream& is, matrix<T>& a) {
    is >> std::ws;
    char c = is.peek();
    if (c == '%') while (is.good() && (c != '\n')) is.get(c);
    size_t nrow, ncol, nnz;
    is >> nrow >> ncol >> nnz;
    a.resize (nrow, ncol, nnz);
    a._start = 0;
    size_t i_prec = 0, j_prec = 0;
    for (size_t p = 0; p < nnz; p++) {
        size_t i;
        is >> i >> a._idx [p] >> a._val [p];
        i--;
        a._idx [p]--;
        if (p == 0 || i != i_prec) {
            assert (p == 0 || i_prec < i); // input may be sorted by row index
            i_prec = i;
            j_prec = 0;
        } else {
            assert (j_prec < a._idx [p]); // input may be sorted by column index
            assert (a._idx[p] < ncol); // input has invalid column index
            j_prec = a._idx [p];
        }
        a._start [i+1]++;
    }
    for (size_t i = 0; i < nrow; i++)
        a._start [i+1] += a._start [i];
    return is;
}

```

Remarquons les tests en lectures, afin de vérifier que l'entrée est bien triée par ordre croissant de lignes et colonnes. Le format de fichier utilisé ici est le standard d'échange dans ce domaine. Le site internet *matrix market* [Deu02] gère un grand nombre de telles matrices.

En regroupant les déclarations précédentes dans un fichier d'en-tête 'matrix.h', un programme de test, ainsi que qu'une entrée associé à la matrice creuse de l'exemple précédent, sont donnés par :



matrix_tst.cc	a.mm
<pre> #include "matrix.h" using namespace std; int main() {     matrix&lt;double&gt; a;     cin &gt;&gt; a;     cout &lt;&lt; a; } </pre>	<pre> %%MatrixMarket 5 5 12 1 1 1 1 3 2 1 4 3 2 2 4 2 5 5 3 1 6 3 3 7 3 4 8 4 1 9 4 4 10 5 3 11 5 5 12 </pre>

La compilation et le test sont donnés par :

```

c++ matrix_tst.cc -o matrix_tst
matrix_tst < a.mm

```

## 5.2 Le produit matrice-vecteur

Le produit matrice-vecteur cumule sur chaque ligne les produits non-nuls et les range dans le vecteur résultat :

matrix.h (suite)

```

template <class T>
std::valarray<T> matrix<T>::operator* (const std::valarray<T>& x) const {
    assert (ncol() == x.size()); // sizes may match
    std::valarray<T> y (T(0), nrow());
    for (size_t i = 0; i < nrow(); i++)
        for (size_t p = _start [i]; p < _start [i+1]; p++)
            y [i] += _val [p]*x [_idx [p]];
    return y;
}

```

## 5.3 Application au gradient conjugué

Voici un programme de test du gradient conjugué avec des matrices creuses.

matrix_cg_tst.cc	as.mm
<code>#include "matrix.h"</code>	
<code>#include "cg.h"</code>	
<code>#include "eye.h"</code>	<code>%%MatrixMarket</code>
<code>using namespace std;</code>	<code>4 4 9</code>
<code>int main () {</code>	<code>1 1 1</code>
<code>matrix&lt;double&gt; a;</code>	<code>1 2 2</code>
<code>cin &gt;&gt; a;</code>	<code>2 1 2</code>
<code>valarray&lt;double&gt; x(1.0, a.nrow());</code>	<code>2 3 3</code>
<code>valarray&lt;double&gt; b = a*x;</code>	<code>3 2 3</code>
<code>valarray&lt;double&gt; xi(0.0, a.nrow());</code>	<code>3 3 4</code>
<code>cg (a, xi, b, eye(), 1000, 1e-7);</code>	<code>3 4 5</code>
<code>valarray&lt;double&gt; xerr = xi-x;</code>	<code>4 3 5</code>
<code>double err = norm(xerr);</code>	<code>4 4 6</code>
<code>cerr &lt;&lt; "err " &lt;&lt; err &lt;&lt; endl;</code>	
<code>return (err &lt; 1e-7)? 0 : 1;</code>	
<code>}</code>	

Le gradient conjugué nécessitant pour converger que la matrice soit symétrique, une telle matrice est également présentée. La compilation et le test sont donnés par :

```

c++ matrix_cg_tst.cc -o matrix_cg_tst
matrix_cg_tst < as.mm

```

## 5.4 Exercices

### EXERCICE 5.4.1 (sortie graphique)

Cet exercice a pour but de visualiser les matrices creuses. Pour cela, on générera un fichier postscript que on visualisera à l'aide de `ghostview` [Deu01]. Écrire une fonction `plotps(ostream&, const matrix<T>&)` ; et un programme `mm2ps.cc` qui lit une matrice sur le standard d'entrée et écrit le postscript sur le standard de sortie.

```

mm2ps < a.mm > a.ps
ghostview a.ps

```

On pourra s'inspirer du code `sparskit` [saa94] de Y. saad, fonction `pspltm` du fichier `INOUT/inout.f`. Tester la visualisation avec des matrices du site internet *matrix market* [Deu02].

### EXERCICE 5.4.2 (produit transposé)

Écrire la fonction `A.trans_mult(x)` qui renvoie le produit  $A^x$ .

**EXERCICE 5.4.3** (intégrité)

Écrire une fonction membre `matrix<T>::check()` const qui vérifie l'intégrité des données, de façon analogue à celle effectuée lors de la lecture.

**EXERCICE 5.4.4** (diagonale de matrice)

Écrire la fonction `diag(x)` qui renvoie la diagonale de  $A$  sous la forme d'un tableau `valarray`. Lorsque la matrice  $A$  est rectangulaire, le tableau sera de taille `min(nrow, ncol)`.

**EXERCICE 5.4.5** (triangulaire)

Calculer `triu(A)` et `tril(A)`, les triangulaires supérieures et inférieures, respectivement, (sans la diagonale) d'une matrice creuse  $A^x$ .

**EXERCICE 5.4.6** (somme)

Calculer la somme de deux matrices creuses. Utiliser le fait que les coefficients sont triés par indices de colonne croissants pour effectuer  $A + B$  en un temps de calcul  $\mathcal{O}(\text{nnz } A + \text{nnz } B)$ .

**EXERCICE 5.4.7** (transposée [difficile])

Calculer la transposée d'une matrice creuse.

**EXERCICE 5.4.8** (produit [difficile])

Calculer le produit de deux matrices creuses.

# Chapitre 6

## Maillages

La résolution de problèmes multi-dimensionnels sur des géométries générales utilise généralement un maillage. Un maillage est défini comme une liste d'éléments :

$$\mathcal{T}_h = \{K_0, K_1, \dots, K_{m-1}\}$$

Le maillage contiendra également la liste des coordonnées des sommets.

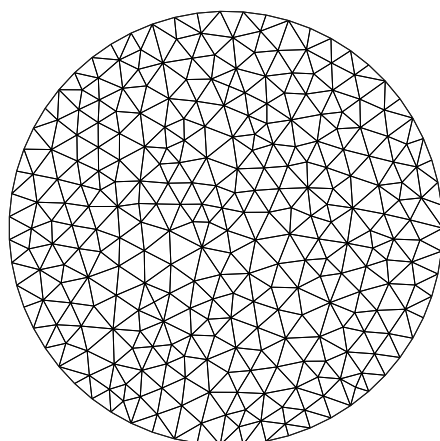


FIG. 6.1 – Maillage d'un cercle.

## fixed\_array.h

---

```

#include <iostream>
template <class T, int N>
class fixed_array {
protected :
    T _x [N];
public :
    fixed_array() {
        for (size_t i = 0; i < N; i++) _x[i] = T();
    }
    const T& operator[] (size_t i) const { return _x [i]; }
    T& operator[] (size_t i) { return _x [i]; }
    friend std :: istream& operator >> (std :: istream& is, fixed_array<T,N>& a) {
        for (size_t i = 0; i < N; i++)
            is >> a._x[i];
        return is;
    }
    friend std :: ostream& operator << (std :: ostream& os, const fixed_array<T,N>& a) {
        for (size_t i = 0; i < N; i++) {
            os << a._x[i];
            if (i!= N-1) os << " ";
        }
        return os;
    }
};

```

## 6.1 Les éléments

Nous commencerons par les triangles en dimension deux. Un triangle du maillage est représenté par la classe `element` :

## element.h

---

```

#include "fixed_array.h"
class element : public fixed_array<size_t,3> {
public :
    friend std :: ostream& operator << (std :: ostream& os, const element& K) {
        return os << K[0]+1 << " " << K[1]+1 << " " << K[2]+1; }
    friend std :: istream& operator >> (std :: istream& is, element& K) {
        is >> K[0] >> K[1] >> K[2];
        K[0]--; K[1]--; K[2]--;
        return is;
    }
};

```

La programmation des fonctions membres est également simple :

```

#ifdef TO_CLEAN
#include <iostream>
inline size_t element : :operator[] (size_t i) const { return _vertice_idx [i]; }
inline size_t& element : :operator[] (size_t i) { return _vertice_idx [i]; }
inline std : :istream& operator >> (std : :istream& is, element& K) {
    is >> K[0] >> K[1] >> K[2]; K[0]--; K[1]--; K[2]--; return is; }
inline std : :ostream& operator << (std : :ostream& os, const element& K) {
    return os << K[0]+1 << " " << K[1]+1 << " " << K[2]+1; }
#endif // TO_CLEAN

```

## 6.2 Les sommets

Un sommet du plan sera décrit par la classe `point` :

point.h

---

```

#include "fixed_array.h"
template <class T>
class point : public fixed_array<T,2> {
#ifdef TO_CLEAN
public :
    explicit point (const T& x = 0, const T& y = 0) { _x[0] = x; _x[1] = y; }
#endif // TO_CLEAN
};

```

## 6.3 La classe maillage

La structure de données du maillage est décrite par la classe `mesh` :

## mesh.h

```

#include <vector>
#include "point.h"
#include "element.h"
template <class T>
class mesh : public std::vector<element> {
public :
    mesh ();
    mesh (const mesh<T>&);
    mesh<T>& operator= (const mesh<T>&);
    size_t n_vertice () const;
    size_t n_internal_vertice () const;
    const point<T>& vertice (size_t i) const;
    point<T>& vertice (size_t i);
    bool is_boundary (size_t i) const;
    bool is_internal (size_t i) const;
    T meas (size_t l) const;
    template <class U> friend std::istream& operator >> (std::istream&, mesh<U>&);
    template <class U> friend std::ostream& operator << (std::ostream&, const mesh<U>&);
protected :
    std::vector<point<T> > _vertice;
    std::vector<size_t> _boundary;
    size_t _n_internal_vertice;
};

```

Remarquons l'utilisation de la classe `vector<T>` (voir [Fon97, p. 53] ou [Str01, p. 523]), de la librairie standard C++. La classe `mesh` hérite de la classe `vector<element>` : c'est une *classe dérivée*. La fonction membre `meas(j)` revoie l'aire du  $j$ -ème triangle. Cette fonction sera décrite et utilisée plus loin (page 7.3.1).

La programmation des fonctions membres suit :

```

#ifdef TO_CLEAN
#include <iostream>
inline size_t element::operator[] (size_t i) const { return _vertice_idx [i]; }
inline size_t& element::operator[] (size_t i) { return _vertice_idx [i]; }
inline std::istream& operator >> (std::istream& is, element& K) {
    is >> K[0] >> K[1] >> K[2]; K[0]--; K[1]--; K[2]--; return is; }
inline std::ostream& operator << (std::ostream& os, const element& K) {
    return os << K[0]+1 << " " << K[1]+1 << " " << K[2]+1; }
#endif // TO_CLEAN

```

Les entrées-sorties permettent de fixer un format de fichier :

```

template <class T> std::ostream& operator << (std::ostream& os, const mesh<T>& m) {
    os << m.n_vertice() << std::endl;
    for (size_t i = 0; i < m.n_vertice(); i++)
        os << m.vertice(i) << " " << m._boundary[i] << std::endl;
    os << m.size() << std::endl;
    for (size_t i = 0; i < m.size(); i++)
        os << m[i] << " 1" << std::endl;
    return os;
}
template <class T> std::istream& operator >> (std::istream& is, mesh<T>& m) {
    size_t nvert;
    is >> nvert;
    m._vertice.resize (nvert);
    m._boundary.resize (nvert);
    m._n_internal_vertice = 0;
    for (size_t i = 0; i < nvert; i++) {
        is >> m._vertice[i] >> m._boundary[i];
        if (m._boundary[i] == 0) m._n_internal_vertice++;
    }
    size_t nelt;
    is >> nelt;
    m.resize (nelt);
    for (size_t i = 0; i < nelt; i++) {
        int domain; // not used
        is >> m[i] >> domain;
        m [i];
    }
    return is;
}

```

Le format de fichier utilisé ici est très proche de celui utilisé par de nombreux générateurs de maillages, tel [Hec97]. Il n'existe malheureusement pas de conventions de format de fichier dans ce domaine.

Les sommets internes sont suivis de l'indicateur 0 et ceux situés sur la frontière, d'un indicateur non-nul pouvant être un numéro de domaine frontière. Les éléments sont suivis de l'indicateur de sous domaine, qui ne sera pas utilisé par la suite.

## 6.4 Test des entrées-sorties

Voici à présent un petit programme de test :



mesh_tst.cc	square.mail
	4
<code>#include "mesh.h"</code>	0 0 1
<code>using namespace std;</code>	1 0 1
<code>int main() {</code>	1 1 1
<code>mesh&lt;double&gt; m;</code>	0 1 1
<code>cin &gt;&gt; m;</code>	2
<code>cout &lt;&lt; m;</code>	1 2 3 1
<code>}</code>	1 3 4 1

```

c++ mesh_tst.cc -o mesh_tst
mesh_tst < square.mail

```

## 6.5 Exercices

**EXERCICE 6.5.1** (sortie graphique)

Cet exercice a pour but de visualiser les maillages avec `gnuplot` [WKL<sup>+</sup>98]. S'inspirant de l'exercice 4.4.4 page 43, écrire la fonction `plot(Th)`. Note : on utilisera la commande `set size ratio 1` pour avoir des échelles égales en  $x$  et  $y$ .

**EXERCICE 6.5.2** (longueurs extrêmes des arêtes)

Écrire les fonctions membres de la classe

```

T h_max() const;
T h_min() const;

```

et qui calculent  $h$  et  $h_{\min}$  les longueur maximales et minimales, respectivement, d'une triangulation  $\mathcal{T}_h$ .

# Chapitre 7

## Méthode des éléments finis

Le but de ce chapitre est de présenter un aperçu de la programmation des éléments finis en C++. Pour une utilisation plus poussée, la librairie `rheolef` [SR02a, SR02b] offre un environnement très complet dans ce domaine.

### 7.1 Une interpolation par éléments

Soit  $\Omega$  un ouvert borné de  $\mathbb{R}^2$ . Reprenons le problème de POISSON avec conditions aux limites homogènes, déjà abordé page 4.1 :

*(Q) : trouver  $u$ , définie de  $\Omega$  dans  $\mathbb{R}$ , telle que*

$$\begin{aligned} -\Delta u &= f \text{ dans } \Omega \\ u &= 0 \text{ sur } \partial \Omega \end{aligned}$$

où  $f$  est une fonction donnée, de  $\Omega$  dans  $\mathbb{R}$ .

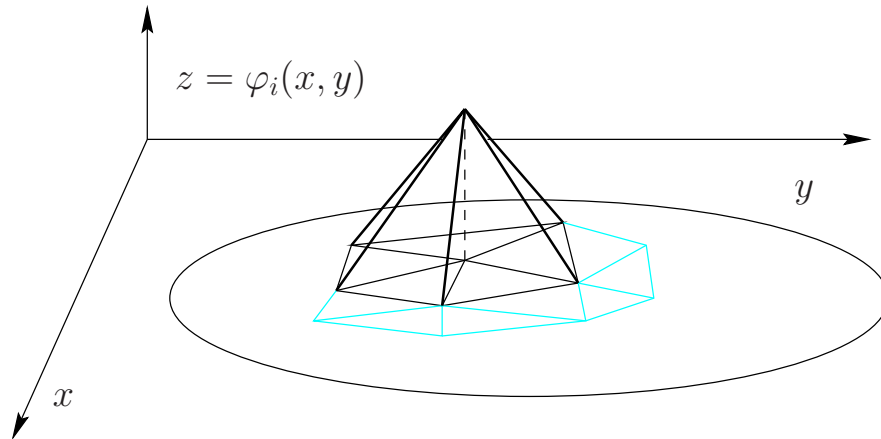
La méthode des différences finies nous restreignait au cas où  $\Omega$  était rectangulaire. Ici, nous pourrions aborder des domaines  $\Omega$  beaucoup plus généraux : nous allons approcher la solution  $u(x)$  par la technique des éléments finis. Soit  $X_h$  l'ensemble des fonctions continues de  $\Omega$  dans  $\mathbb{R}$  affines dans chaque élément triangulaire  $K$  d'un maillage donné. Le maillage est noté  $\mathcal{T}_h$ .

Nous considérons alors le problème suivant :

*(Q)<sub>h</sub> : trouver  $u_h \in V_{0,h}$  telle que*

$$\int_{\Omega} \nabla u_h(x, y) \cdot \nabla v_h(x, y) \, dx \, dy = \int_{\Omega} f(x, y) v_h(x, y) \, dx \, dy, \quad \forall v_h \in V_{0,h} \quad (7.1)$$

où  $V_{0,h} = X_h \cap H_0^1(\Omega)$  représente l'ensemble des fonctions continues de  $\Omega$  dans  $\mathbb{R}$  s'annulant sur la frontière  $\partial\Omega$  et affines dans chaque élément triangulaire  $K$  d'un maillage donné.

FIG. 7.1 – Fonction de base  $\varphi_i$ .

L'espace  $X_h$  est de dimension finie. Soit  $n = \dim V_h$ . Elle est égale au nombre de sommets de la triangulation  $\mathcal{T}_h$ .

Nous pouvons choisir pour base de  $X_h$  les fonctions  $(\varphi_i)_{0 \leq i \leq n-1}$  valant 1 au  $i$ -ème sommet  $(x_i, y_i)$  de  $\mathcal{T}_h$  et zéro aux autres sommets (voir Fig. 7.1). Nous pouvons décomposer  $u_h$  sur cette base :

$$u_h(x, y) = \sum_{j=0}^{n-1} u_j \varphi_j(x, y)$$

. où  $u_i = u_h(x_i, y_i)$  sont les composantes de  $u_h$  dans  $X_h$ , appelées aussi *degrés de liberté*.

L'opérateur d'interpolation  $\pi_h$  dans  $X_h$  se code très simplement :

**interpolate.h**

---

```

template <class T>
std : :valarray<T> interpolate (const mesh<T>& Th, T (*v)(const point<T>&)) {
    std : :valarray<T> vh (Th.n_vertice());
    for (size_t i = 0; i < Th.n_vertice(); i++)
        vh [i] = v (Th.vertice (i));
    return vh;
}

```

En choisissant  $v_h = \varphi_i$  dans (7.1), le problème se ramène à

trouver  $(u_i)_{1 \leq i \leq n} \in \mathbb{R}^n$  telle que

$$\sum_{j=1}^n \left( \int_{\Omega} \nabla \varphi_i(x, y) \cdot \nabla \varphi_j(x, y) \, dx \, dy \right) u_j = \sum_{j=1}^n \left( \int_{\Omega} \varphi_i(x, y) \varphi_j(x, y) \, dx \, dy \right) f(x_j, y_j), \quad \forall j \in \text{int}(\mathcal{T}_h) \quad (7.3)$$

$$u_j = 0, \quad \forall j \in \{1, \dots, n\} \setminus \text{int}(\mathcal{T}_h).$$

où  $\text{int}(\mathcal{T}_h)$  désigne l'ensemble des indices des sommets de  $\mathcal{T}_h$  appartenant à l'intérieur du domaine  $\Omega$ .

Introduisons les matrices :

$$A_{i,j} = \int_{\Omega} \nabla \varphi_i(x, y) \cdot \nabla \varphi_j(x, y) \, dx \, dy,$$

$$M_{i,j} = \int_{\Omega} \varphi_i(x, y) \varphi_j(x, y) \, dx \, dy,$$

Remarquons que  $A = (A_{i,j})$  et  $M = (M_{i,j})$  se décomposent en :

$$A = \sum_{k=1}^m A^{(k)}$$

$$M = \sum_{k=1}^m M^{(k)}$$

où  $A^{(k)}$  et  $M^{(k)}$  sont les matrices élémentaires relatives à l'élément  $K_k$  de  $\mathcal{T}_h$  :

$$A_{i,j}^{(k)} = \int_{K_k} \nabla \varphi_i(x, y) \cdot \nabla \varphi_j(x, y) \, dx \, dy,$$

$$M_{i,j}^{(k)} = \int_{K_k} \varphi_i(x, y) \cdot \varphi_j(x, y) \, dx \, dy,$$

Les matrices  $A^{(k)}$  et  $M^{(k)}$  ne font intervenir que les trois sommets du triangle  $K_k$ , n'ont donc au plus que neuf éléments non-nuls. Les matrices  $A$  et  $M$ , de taille  $n \times n$  ont donc au plus  $9m$  éléments non-nuls. Or, pour un maillage classique,  $m = \mathcal{O}(n)$ , et ainsi  $A$  et  $M$  sont très creuses.

## 7.2 La re-numérotation des sommets

Afin de bien imposer la condition aux bords  $u_h = 0$  sur  $\partial\Omega$ , nous allons séparer les degrés de liberté en deux catégories : (i) ceux associés à un sommet du maillage interne à  $\Omega$ ; (ii) ceux associés à un sommet sur la frontière de  $\Omega$ . L'algorithme de renumérotation est :

renumbering.h

---

```

template<class T>
void renumbering (const mesh<T>& Th, std::valarray<size_t>& num) {
    num.resize (Th.n_vertice());
    size_t i_bdr = 0;
    size_t i_int = 0;
    for (size_t i = 0; i < Th.n_vertice(); i++)
        if (Th.is_boundary (i))
            num [i] = i_bdr++;
        else
            num [i] = i_int++;
}

```

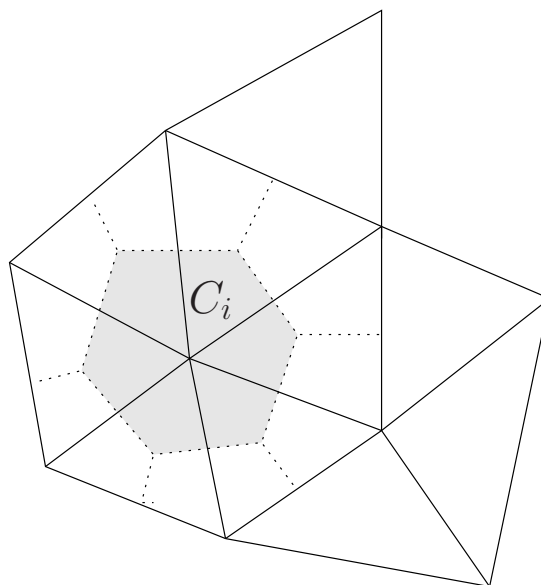


FIG. 7.2 – Volume fini  $C_i$  centré autour du sommet  $i$ .

## 7.3 Le calcul des matrices élémentaires

### 7.3.1 Calcul de la matrice de masse $M$

Utilisons la formule des trapèzes pour évaluer  $M_{i,j}^{(l)}$  :

$$M_{i,j}^{(l)} = \frac{\text{mes}(K_l)}{3} \sum_{k \in \text{sommet}(K_l)} \varphi_i(x_k, y_k) \varphi_j(x_k, y_k)$$

Si  $i \neq j$  ou si  $i$  ou  $j$  ne sont pas des indices de sommets de  $K_l$ , alors  $M_{i,j}^{(l)} = 0$ , et autrement,  $i = j \in \text{sommet}(K_l)$  et  $M_{i,i}^{(l)} = \text{mes}(K_l)/3$ . Ainsi  $M_{i,i}^{(l)}$  et donc  $M^{(l)}$  sont diagonales.

$$M_{i,i} = \sum_{l/i \in \text{sommet}(K_l)} \frac{\text{mes}(K_l)}{3}$$

La valeur  $M_{i,i}$  s'interprète comme étant l'aire du *volume fini*  $C_i$  centré autour du sommet  $i$ , et constitué des portions d'éléments adjacents en reliant le barycentre de ces éléments aux milieux des arêtes (voir Fig. 7.2). Un algorithme pour assembler  $M$  est :

## assembly\_mass.h

```

template <class T>
void assembly_mass (const mesh<T>& Th, const std::valarray<size_t>& num,
                   std::valarray<T>& M) {
    M.resize (Th.n_internal_vertice());
    M = 0;
    for (size_t l = 0; l < Th.size(); l++) {
        T area = Th.meas (l);
        for (size_t r = 0; r < 3; r++) {
            size_t i = Th [l][r];
            if (Th.is_internal (i))
                M [num [i]] += area/3;
        }
    }
}

```

La fonction membre `meas(l)` qui renvoie l'aire d'un triangle est donnée par

## mesh.h (suite)

```

template<class T> T mesh<T>::meas (size_t l) const {
    const element& K = operator[] (l);
    size_t i = K[0];
    size_t j = K[1];
    size_t k = K[2];
    return ((_vertice[j][0] - _vertice[i][0])*(_vertice[k][1] - _vertice[i][1])
           - (_vertice[j][1] - _vertice[i][1])*(_vertice[k][0] - _vertice[i][0]))/2;
}

```

### 7.3.2 Calcul de la matrice d'énergie $A$

Les fonctions de base  $\varphi_i$  sont affines dans chaque élément  $K$ . Par conséquent, leur gradient  $y$  est constant :

$$A_{i,j}^{(l)} = \nabla \varphi_i \cdot \nabla \varphi_j \text{mes}(K_l)$$

Ce coefficient ne peut être non-nul que si  $i$  et  $j$  sont des indices de sommets de  $K$  : ils sont alors soit égaux, soit adjacents sur une arête de  $K$ . Ainsi  $A_{i,j} \neq 0$  dès que  $i$  et  $j$  sont des sommets d'un même élément.

Soit  $K_l$  un élément contenant le sommet d'indice  $i$  et  $p$  et  $q$  les indices des deux sommets autres que  $i$ . Quitte à permuter le rôle de  $p$  et  $q$ , nous supposons que  $(i, p, q)$  tourne dans le sens positif sur  $\partial K_l$  (voir Fig. 7.3). Un rapide calcul donne la restriction à  $K_l$  de  $\varphi_i$  :

$$\varphi_{i|K_l}(x, y) = \frac{(x_p - x)(y_q - y_p) - (x_q - x_p)(y_p - y)}{2\text{mes}(K_l)}$$

si bien que

$$\frac{\partial \varphi_{i|K_l}}{\partial x} = -\frac{y_q - y_p}{2\text{mes}(K_l)} \quad \text{et} \quad \frac{\partial \varphi_{i|K_l}}{\partial y} = \frac{x_q - x_p}{2\text{mes}(K_l)}$$

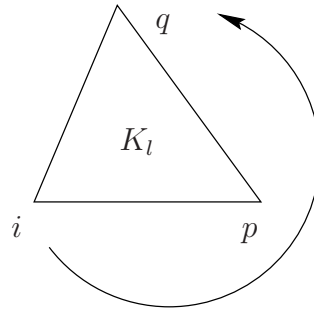


FIG. 7.3 – Rotation autour de  $K_l$  suivant  $(i, p, q)$  dans le sens positif.

Pour  $i = j$  nous obtenons :

$$A_{i,j}^{(l)} = \frac{(x_q - x_p)^2 + (y_q - y_p)^2}{4\text{mes}(K_l)}$$

tandis-que  $i \neq j$  conduit à :

$$A_{i,j}^{(l)} = \frac{(x_q - x_p)(x_i - x_p) + (y_q - y_p)(x_i - x_p)}{4\text{mes}(K_l)}$$

## 7.4 L'assemblage des matrices creuses

La matrice  $A$  n'est donc pas diagonale. De plus, le nombre d'arrête arrivant à un sommet  $i$  n'est pas fixé *a priori*. En moyenne, il est de l'ordre de 6 pour un maillage régulier bidimensionnel formé de triangle, et de l'ordre de 30 dans le cas tridimensionnel.

Une première idée serait de cumuler les matrices élémentaires dans un tableau de listes. Chaque liste d'indice  $i$  dans le tableau contient la liste des paires  $(j, A_{i,j})$  de coefficients de la matrice en cours de construction. Ce tableau de liste est ensuite converti en une structure de donnée `matrix`.

Nous avons à rechercher dans la liste pour voir si un coefficient  $A_{i,j}$  est déjà représenté ou bien s'il faut en créer une représentation. Nous pouvons accélérer cette recherche en maintenant la liste triée par ordre croissant des indices  $j$ . La librairie standard C++ propose la *table de correspondance* `map<Key,T>` (voir [Fon97, p. 88] ou [Str01, p. 534]), qui proposent un rangement ordonné du type de celui d'un *dictionnaire*. La structure de donnée `y` est celle d'un arbre. Si la  $i$ -ème ligne de la matrice creuse contient  $\text{nnz}(i)$  éléments non-nuls, la recherche d'un élément prendra un temps de calcul en  $\mathcal{O}(\log \text{nnz}(i))$ . Le rangement des  $\text{nnz}(i)$  éléments de la ligne prendra donc le temps  $\mathcal{O}(\text{nnz}(i) \log \text{nnz}(i))$ , et celui de tous les coefficients de la matrice le temps  $\mathcal{O}(\text{nnz} \log \text{nnz})$ .

La classe `matrix_store` hérite de la classe `vector<map<size_t,T>>` :

## matrix\_store.h

---

```

#include <vector>
#include <map>
template<class T>
class matrix_store : public std::vector<std::map<size_t, T>> {
public:
    typedef T value_type;
    typedef typename std::map<size_t,T>::iterator iterator;
    typedef typename std::map<size_t,T>::const_iterator const_iterator;
    matrix_store (size_t nrow = 0, size_t ncol = 0);
    size_t nrow() const;
    size_t ncol() const;
    size_t nnz() const;
    std::map<size_t, T>& row (size_t i);
    const std::map<size_t, T>& row (size_t i) const;
    T& entry (size_t i, size_t j);
    T operator() (size_t i, size_t j) const;
protected:
    size_t _ncol;
    size_t _nnz;
};

```

Le codage des fonctions membres est :

L'extraction d'un élément utilise des actions spécifiques de la classe `map<Key,T>`. On vérifie tout d'abord s'il n'existe pas déjà, via

```
map<Key,T>::find (Key);
```

Cette fonction renvoie un itérateur (voir [Fon97, p. 128], [MS96, p. 49] ou bien [SGI02]) sur une paire de type `pair_type<Key,T>`. Si cet itérateur est valide, c'est que l'élément existe déjà, et alors nous renvoyons une référence sur le second objet de la paire, c'est à dire sur la valeur numérique du coefficient de la matrice. Dans le cas contraire, la fonction

```
T& map<Key,T>::operator [] (Key);
```

appelée par `row(i)[j]` insère un élément avec la valeur zéro et en renvoie la référence.

Il s'agit enfin de convertir une `matrix_store<T>` en une `matrix<T>`. Pour cela, on ajoute dans 'matrix.h' la fonction de conversion :

## matrix.h (suite)

---

```

#include "matrix_store.h"
template <class T>
class matrix {
public:
    //...
    friend void convert (const matrix_store<T>&, matrix<T>&);
};

```



Le codage est :

## 7.5 Retour sur la matrice d'énergie $A$

Nous pouvons à présent écrire l'algorithme d'assemblage de  $A$  :

```

assembly_energy.h
-----
template <class T>
inline T contribution (const mesh<T>& Th, size_t i1, size_t j1, size_t k1,
    size_t i2, size_t j2, size_t k2) {
    return (Th.vertice(k1)[0]-Th.vertice(j1)[0])*(Th.vertice(k2)[0]-Th.vertice(j2)[0])
        + (Th.vertice(k1)[1]-Th.vertice(j1)[1])*(Th.vertice(k2)[1]-Th.vertice(j2)[1]);
}
template <class T>
void assembly_energy (const mesh<T>& Th, const std::valarray<size_t>& num, matrix<T>& a) {
    matrix_store<T> as (Th.n_internal_vertice(), Th.n_internal_vertice());
    for (size_t l = 0; l < Th.size(); l++) {
        T coef = 1/(4*Th.meas(l));
        for (size_t r = 0; r < 3; r++) {
            size_t i = Th[l][r];
            size_t j = Th[l][(r+1) % 3];
            size_t k = Th[l][(r+2) % 3];
            if (Th.is_internal(i)) {
                as.entry (num[i],num[i]) += coef*contribution (Th, i,j,k, i,j,k);
                if (Th.is_internal(j))
                    as.entry (num[i],num[j]) += coef*contribution (Th, i,j,k, j,k,i);
                if (Th.is_internal(k))
                    as.entry (num[i],num[k]) += coef*contribution (Th, i,j,k, k,i,j);
            }
        }
    }
    convert (as, a);
}

```

et un programme de test :

---

assembly\_energy\_tst.c

---

```
#include <valarray>
#include "valarray_util.h"
#include "mesh.h"
#include "matrix.h"
#include "renumbering.h"
#include "assembly_energy.h"
using namespace std;
int main() {
    mesh<double> Th;
    cin >> Th;
    valarray<size_t> num;
    renumbering (Th, num);
    matrix<double> A;
    assembly_energy (Th, num, A);
    cout << A;
}
```

La compilation et le lancement sont :

```
c++ assembly_energy_tst.cc -o assembly_energy_tst
assembly_energy_tst < L-20-adapt.mail > L-20-adapt.mm
matrix_ps_tst < L-20-adapt.mm > L-20-adapt.ps
ghostview L-20-adapt.ps
```

où le fichier 'L-20-adapt.mail' correspond au maillage de la figure 7.4.a. La visualisation de la matrice creuse Fig. 7.4.b est obtenue à partir du programme `matrix_ps_tst` présenté dans l'exercice 5.4.1 et du logiciel `ghostview` [Deu01]. Le maillage 'L-20-adapt.mail' a été obtenu avec le logiciel `bang` [Hec97].

## 7.6 Résolution du système linéaire

Passons à la résolution du système linéaire (7.2)-(7.3) : issu de la discrétisation éléments finis.

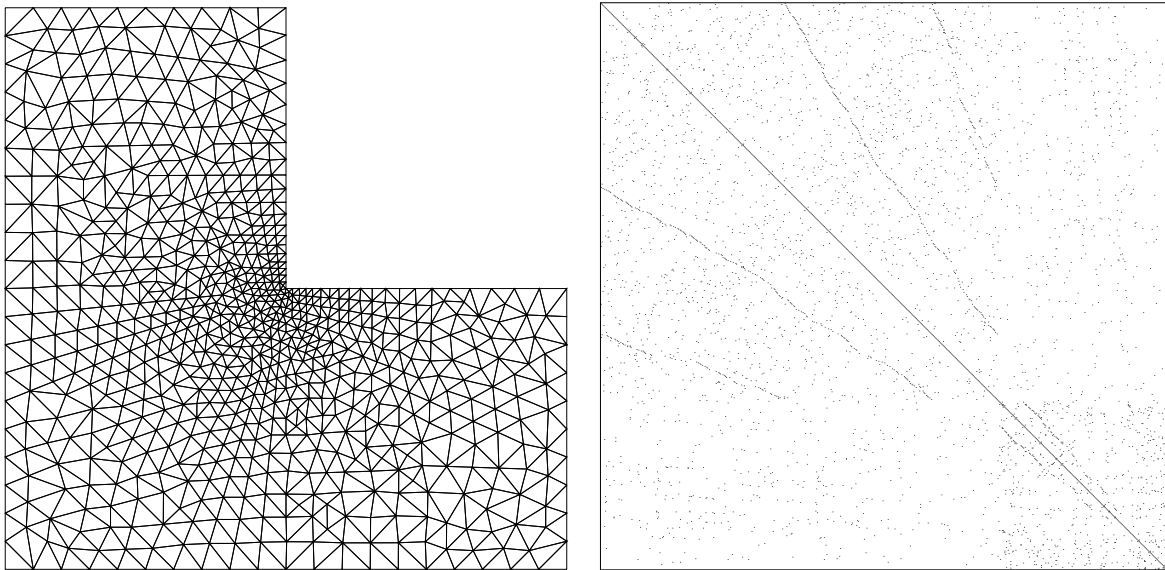


FIG. 7.4 – Maillage et structure creuse de la matrice d'énergie associée.

---

dirichlet.h

---

```

#include "mesh.h"
#include "matrix.h"
#include "renumbering.h"
#include "assembly_mass.h"
#include "assembly_energy.h"
#include "interpolate.h"
#include "valarray_util.h"
#include "cg.h"
#include "eye.h"
template <class T>
std::valarray<T> dirichlet (const mesh<T>& Th, const std::valarray<T>& fh) {
    std::valarray<size_t> num;
    std::valarray<T> M;
    matrix<T> A;
    renumbering (Th, num);
    assembly_mass (Th, num, M);
    assembly_energy (Th, num, A);
    std::valarray<bool> internal (Th.n_vertice());
    for (size_t i = 0; i < Th.n_vertice(); i++)
        internal [i] = Th.is_internal (i);
    std::valarray<T> b = M*std::valarray<T>(fh [internal]);
    std::valarray<T> x (0.0, Th.n_internal_vertice());
    cg (A, x, b, eye(), 1000, 1e-10);
    std::valarray<T> uh (0.0, Th.n_vertice());
    uh [internal] = x;
    return uh;
}

```

Notez l'utilisation du masque `internal` pour sélectionner les sommets internes du maillage, correspondant aux inconnues du système linéaire.

Avec pour second membre  $f(x, y) = x(1 - x) + y(1 - y)$ , la solution exacte est connue explicitement  $u(x, y) = x(1 - x)y(1 - y)/2$ , ce qui permet de calculer l'erreur. Voici donc le programme de test :

---

```
finite_element_tst.cc
```

---

```
#include "dirichlet.h"
using namespace std;
double f (const point<double>& x) { return x[0]*(1-x[0])+x[1]*(1-x[1]); }
double u (const point<double>& x) { return x[0]*(1-x[0])*x[1]*(1-x[1])/2; }
int main() {
    mesh<double> Th;
    cin >> Th;
    valarray<double> uh = dirichlet (Th, interpolate (Th, f));
    valarray<double> pi_h_u = interpolate (Th, u);
    valarray<double> err = abs(pi_h_u-uh);
    cerr << "err " << err.max() << endl;
}
```

La compilation et le lancement sont :

```
c++ finite_element_tst.cc -o finite_element_tst
finite_element_tst < square-10.mail
```

La figure 7.5.a trace l'erreur en norme  $L^\infty$ , en fonction du pas  $h$  du maillage. Nous pouvons observer que  $\|u - \pi_h u\|_{\infty, \Omega} = \mathcal{O}(h^2)$ . La figure 7.5.b trace le nombre d'itérations, par l'algorithme du gradient conjugué, nécessaire pour obtenir un résidu de  $10^{-10}$  en fonction du nombre d'inconnues  $n_{\text{int}}$ . Ce nombre d'itération croît comme  $\mathcal{O}(n_{\text{int}}^{1/2})$ . Or le coût d'une itération du gradient conjugué est  $\mathcal{O}(n_{\text{int}})$ , si bien que le coût total de la résolution est  $\mathcal{O}(n_{\text{int}}^{3/2})$ .

## 7.7 Exercices

**EXERCICE 7.7.1** (sortie graphique)

Cet exercice a pour but de visualiser les fonctions de type éléments finis avec `gnuplot` [WKL<sup>+</sup>98]. S'inspirant de l'exercice 4.4.4 page 43, écrire la fonction `plot(Th, uh)` en utilisant la commande `splot` de `gnuplot`.

**EXERCICE 7.7.2** (condition aux bords non-homogène)

Modifier la fonction `assembly_energy` pour accepter une condition aux bords non-homogène.

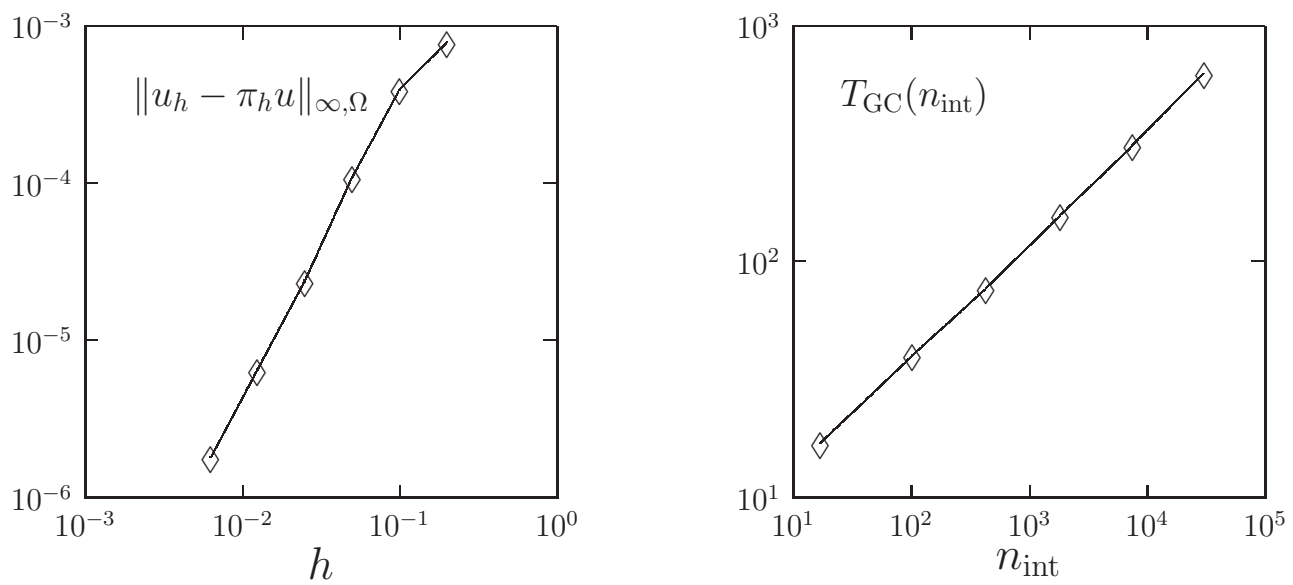


FIG. 7.5 – Convergence de la méthode des éléments finis (a); nombre d'itération du GC en fonction de la taille du problème (b).

# Chapitre 8

## Introduction au calcul formel

Dans cette partie, nous nous proposons d'étudier la conception d'un système de calcul formel (voir par exemple pour compléments [Bro97]) en C++ par la programmation d'une classe de polynômes.

Un polynôme est représenté ses coefficients  $(c_i)_{0 \leq i \leq n-1}$  :

$$p(X) = \sum_{k=0}^{n-1} c_k X^k, \quad c_{n-1} \neq 0.$$

Le degré du polynôme est alors  $n - 1$ . Le polynôme nul correspond à  $n = 0$  et est alors par convention de degré égal à  $-1$ .

### 8.1 Définition d'une classe pour les polynômes

L'indéterminée  $X$  peut être définie simplement comme suit :

polynom.h

---

```
class symbol {  
  public :  
    symbol () {}  
};
```

Cette classe ne contient aucune donnée : elle n'est utilisée que comme marque, pour définir un nouveau type. La classe `polynom` contient le tableau des coefficients :

```

#include <vector>
#include <iostream>
template <typename T>
class polynom {
public :
    polynom(const T& = T());
    polynom(const symbol&);
    polynom(const polynom<T>&);
    polynom<T>& operator= (const polynom<T>&);
    int degree() const;
    const T& coefficient(size_t i) const;
    T operator() (const T& x_value) const;
    polynom<T> operator+ (const polynom<T>&) const;
    polynom<T> operator- (const polynom<T>&) const;
    polynom<T> operator* (const polynom<T>&) const;
    template<typename U> friend std::ostream& operator<< (std::ostream&, const polynom<U>&);
    template<typename U> friend polynom<U> pow (const symbol&, int);
protected :
    std::vector<T> c;
    void check_resize();
};

```

La classe `vector` fait partie de la librairie standard du langage C++. Un certain nombre de fonctions membres et amies y sont déclarées. Nous allons les passer en revue. Tout d'abord, les constructeurs et opérateurs d'affectation standards s'écrivent :

```

template <typename T>
polynom<T>::polynom(const T& c0) : c(1,c0) { if (c0 == T()) c.resize(0); }
template <typename T>
polynom<T>::polynom(const symbol& X1) : c(2,T()) { c[1] = 1; }
template <typename T>
polynom<T>::polynom(const polynom<T>& q) : c(q.c) {}
template <typename T>
polynom<T>& polynom<T>::operator= (const polynom<T>& q) {
    c.resize(q.c.size());
    c = q.c;
    return *this;
}

```

Ceci permet d'initialiser un polynôme à partir d'une constante  $c_0$ , d'une indéterminée  $X$ , ou bien encore par copie d'un autre polynôme. Notez ici le cas particulier lorsque la constante à l'initialisation est nulle : ceci est en accord avec la définition de la fonction degré :

```

template <typename T>
int polynom<T>::degree() const { return c.size() - 1; }

```

## 8.2 Addition, soustraction et multiplication

L'opérateur d'addition est relativement simple :

```

template <typename T>
polynom<T> polynom<T>::operator+ (const polynom<T>& q) const {
    polynom<T> r;
    r.c.resize (std::max(c.size(), q.c.size()), T());
    for (size_t i = 0; i < c.size(); i++) r.c[i] = c[i];
    for (size_t i = 0; i < q.c.size(); i++) r.c[i] += q.c[i];
    r.check_resize();
    return r;
}

```

Il faut cependant prendre soin de traiter le cas où les coefficients de plus haut degré deviennent s'annulent :

```

template <typename T>
void polynom<T>::check_resize () {
    size_t new_size = c.size();
    for (; new_size > 0; new_size--) if (c[new_size-1] != T()) break;
    if (new_size == c.size()) return;
    std::vector<T> tmp(new_size);
    for (size_t i = 0; i < new_size; i++) tmp[i] = c[i];
    c.resize(new_size);
    for (size_t i = 0; i < new_size; i++) c[i] = tmp[i];
}

```

La soustraction est très proche de l'addition :

```

template <typename T>
polynom<T> polynom<T>::operator- (const polynom<T>& q) const {
    polynom<T> r;
    r.c.resize (std::max(c.size(), q.c.size()), T());
    for (size_t i = 0; i < c.size(); i++) r.c[i] = c[i];
    for (size_t i = 0; i < q.c.size(); i++) r.c[i] -= q.c[i];
    r.check_resize();
    return r;
}

```

Il est également facile de définir le produit  $p * q$  de deux polynômes entre eux :

```

template <typename T>
polynom<T> polynom<T>::operator* (const polynom<T>& q) const {
    polynom<T> r;
    r.c.resize(c.size() + q.c.size() - 1, T());
    for (size_t i = 0; i < c.size(); i++)
        for (size_t j = 0; j < q.c.size(); j++)
            r.c[i+j] += c[i]*q.c[j];
    return r;
}

```

Nous souhaitons également reconnaître des expressions plus complexes, contenant des termes de la forme  $c * p$ , où  $c$  est un scalaire :

```

double          c = 2;
polynom<double> X = symbol();

```



```

polynom<double> p = X*X;
polynom<double> q = c*X*p;

```

L'expression  $c * X * p$  sera évaluée de la gauche vers la droite comme indiqué sur la Fig. 8.1. Les combinaisons de la forme  $c * p$  et  $p * c$  sont traités de la façon suivante :

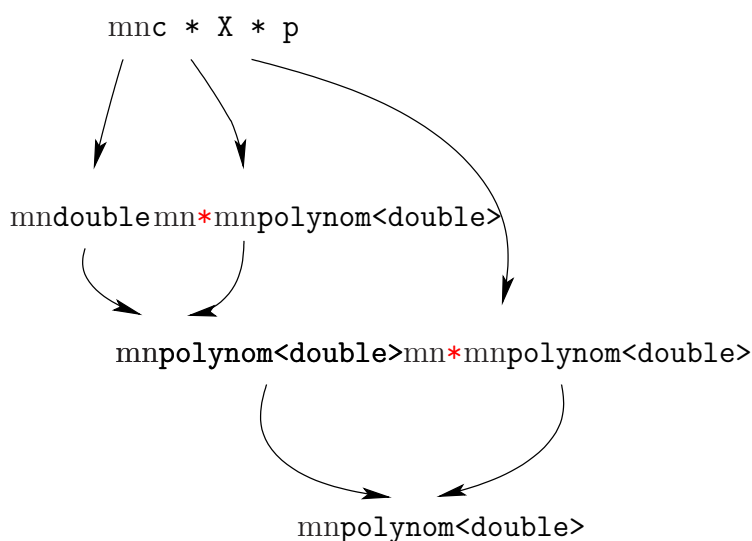


FIG. 8.1 – Évaluation d'une expression polynomiale de la forme  $c * X * p$ .

---

#### polynom.h (suite)

---

```

template <typename T1, typename T2>
polynom<T2> operator* (const T1& c0, const polynom<T2>& p) { return polynom<T2>(c0)*p; }

template <typename T1, typename T2>
polynom<T1> operator* (const polynom<T1>& p, const T2& c0) { return p*polynom<T1>(c0); }

```

Notez l'utilisation du constructeur `polynom<T>(c0)` qui convertit son opérande en polynôme, afin de se ramener à des produits entre deux polynômes. L'utilisation de deux types `T1` et `T2` dans la déclaration `template` permet de reconnaître des produits de la forme  $2 * p$  où `T1=int` et `p` est du type `polynom<double>`.

Enfin, les combinaisons entre scalaires et polynômes s'étendent aux opérateurs d'additions et de soustraction :

```

template <typename T1, typename T2>
polynom<T2> operator+ (const T1& c0, const polynom<T2>& p) { return polynom<T2>(c0)+p; }

template <typename T1, typename T2>
polynom<T1> operator+ (const polynom<T1>& p, const T2& c0) { return p+polynom<T1>(c0); }

template <typename T1, typename T2>
polynom<T2> operator- (const T1& c0, const polynom<T2>& p) { return polynom<T2>(c0)-p; }

template <typename T1, typename T2>
polynom<T1> operator- (const polynom<T1>& p, const T2& c0) { return p-polynom<T1>(c0); }

template <typename T>
polynom<T> operator- (const polynom<T>& p) { return polynom<T>()-p; }

```

La dernière fonction correspond au moins unaire, par exemple dans  $q = -p$ .

### 8.3 Polynômes de LEGENDRE

Nous avons à présent suffisamment d'éléments pour utiliser notre classe `polynom` dans une application : les polynômes de Legendre sont donnés par la formule de récurrence :

$$\begin{aligned}
 P_0(X) &= 1 \\
 P_1(X) &= X \\
 (n+1)P_{n+1}(X) &= (2n+1)XP_n(X) - nP_{n-1}, \quad n \geq 1
 \end{aligned}$$

Ces polynômes ont des propriétés remarquables et sont très largement utilisés en analyse numérique, en particulier pour l'élaboration de formules d'intégration numérique approchées. Écrivons une fonction qui imprime les polynômes de Legendre.

`polynom_legendre.h`

---

```

#include "polynom.h"
#include <vector>
template <typename T>
void polynom_legendre (std::vector<polynom<T> >& P) {
    if (P.size() < 2) return;
    polynom<T> X = symbol();
    P[0] = 1;
    P[1] = X;
    for (size_t n = 2; n < P.size(); n++)
        P[n] = (2.0*n-1)/n*X*P[n-1] - (n-1.0)/n*P[n-2];
}

```

Des termes de la forme  $c * X * p$  et  $c * p$  apparaissent au second membre de l'expression au second membre de l'affectation dans la boucle : nous avons déjà les traiter. Notez également l'utilisation de la classe `vector` pour ranger les données. Un petit programme de test s'écrira simplement :

---

polynom\_legendre\_tst.cc

---

```
#include "polynom_legendre.h"
using namespace std;
int main(int argc, char** argv) {
    size_t n = (argc > 1) ? atoi(argv[1]) : 9;
    vector<polynom<double> > P(n+1);
    polynom_legendre(P);
    for (size_t i = 0; i < P.size(); i++)
        cout << "P" << i << "=" << P[i] << endl;
}
```

Nous avons utilisé une d'une fonction d'impression d'un polynôme et qui s'écrit :

polynom.h (suite)

---

```
template <class T>
std::ostream& operator<< (std::ostream& out, const polynom<T>& p) {
    if (p.c.size() == 0) return out << "0";
    bool prem = true;
    for (int i = p.c.size()-1; i >= 0; i--) {
        if (p.c[i] == 0) continue;
        if (!prem && p.c[i] > 0) out << "+";
        prem = false;
        if (i == 0) {
            out << p.c[i];
            continue;
        }
        if (p.c[i] != 1) {
            if (p.c[i] == -1)
                out << "-";
            else
                out << p.c[i] << "*";
        }
        out << "X";
        if (i == 1) continue;
        out << "^" << i;
    }
    return out;
}
```

La compilation et le test sont :

```
c++ polynom_legendre_tst.cc -o polynom_legendre_tst
./polynom_legendre_tst 15
```

## 8.4 Évaluation

Afin de mélanger calcul formel et calcul numérique, nous allons à présent évaluer les polynômes avec des notations du type  $p(\text{valeur})$  en définissant l'opérateur de parentésage `operator()` :

```

template <typename T>
T polynom<T>::operator() (const T& x) const {
    T v = 0;
    for (size_t i = c.size(); i > 0; i--)
        v = v*x + c[i-1];
    return v;
}

```

Les polynômes peuvent ainsi être appelés avec la même syntaxe que les fonctions. Les classes disposant d'une définition de `operator()` sont appelées *classes fonctions*.

## 8.5 Sortie graphique

Disposant d'une fonction d'évaluation, nous pouvons échantillonner un polynôme sur un intervalle  $[a, b]$  et passer ces valeurs sur un programme de sortie graphique :

polynom\_plot.h

---

```

#include <fstream>
template<class T>
void plot (const polynom<T>& p, const T& a = -1, const T& b = 1) {
    size_t n = 1000;
    T h = (b-a)/n;
    std::ofstream data ("p.dat");
    for (size_t i = 0; i <= n; i++)
        data << a+i*h << " " << p(a+i*h) << std::endl;
    data.close();
    std::ofstream gpl ("p.gpl");
    gpl << "plot [" << a << " : " << b << "] 'p.dat' w l" << std::endl;
    gpl.close();
    system ("gnuplot -persist p.gpl");
}

```

Le programme de test suivant imprimera la  $n$ -ième polynôme de LEGENDRE :

polynom\_legendre\_plot\_tst.cc

---

```

#include "polynom_legendre.h"
#include "polynom_plot.h"
using namespace std;
int main(int argc, char** argv) {
    size_t n = (argc > 1)? atoi(argv[1]) : 9;
    vector<polynom<double> > P(n+1);
    polynom_legendre(P);
    plot(P[n]);
}

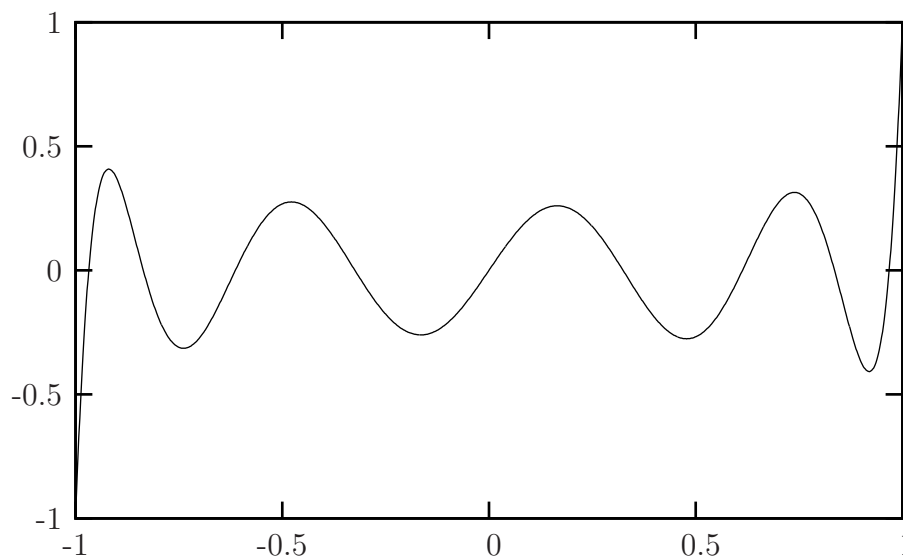
```

La compilation et le test sont :

```

c++ polynom_legendre_plot_tst.cc -o polynom_legendre_plot_tst
./polynom_legendre_plot_tst 3

```

FIG. 8.2 – Polynôme de LEGENDRE pour  $n = 9$ .

```
./polynom_legendre_plot_tst 15
./polynom_legendre_plot_tst 40
```

Le résultat est représenté sur la Fig. 8.2. Au delà de  $n = 50$ , les erreurs d'arrondis se propagent et dénaturent le résultat : l'évaluation peut se poursuivre encore en remplaçant `double` par `long double` dans le code précédent, voire en utilisant des classes de précision arbitraire pour les nombres flottants [FSF05].

## 8.6 Division euclidienne

Pour deux polynômes  $a$  et  $b$ , avec  $b \neq 0$ , on note le  $q$  et  $r$  le quotient et le reste, respectivement, de la division euclidienne, tels que  $a = b * q + r$ , avec soit  $r = 0$  soit  $\text{degré}(r) < \text{degré}(b)$ . L'algorithme suivant donne  $q$  et  $r$  :

```
algo division euclidienne
   $q := 0$ 
   $r := a$ 
  tantque  $r \neq 0$  et  $d := \text{degré}(r) - \text{degré}(b) \geq 0$  faire
     $t := lc(r)/lc(b) * X^d$ 
     $q := q + t$ 
     $r := r - b * t$ 
fin
```

où la notation  $lc(p)$  représente le coefficient du terme de plus haut degré du polynôme  $p$ .

Notre classe polynôme est presque prête pour implémenter un tel algorithme : écrivons-le.

## polynom\_div.h

---

```

#include "polynom.h"
template <typename T>
void div(const polynom<T>& a, const polynom<T>& b, polynom<T>& q, polynom<T>& r) {
    symbol X;
    q = 0;
    r = a;
    int d = 0;
    while (r.degree() != -1 && (d = r.degree() - b.degree()) >= 0) {
        polynom<T> t = lc(r)/lc(b)*pow<T>(X,d);
        q = q + t;
        r = r - b*t;
    }
}

```

Il reste à implémenter  $pow(X, d)$  qui renvoie le monôme  $X^d$  et s'écrit :

## polynom.h (suite)

---

```

template <typename T>
polynom<T> pow (const symbol& X, int d) {
    polynom<T> p;
    p.c.resize (d+1, T());
    p.c[d] = 1;
    return p;
}

```

ainsi que la fonction  $lc(p)$  :

```

template <typename T>
const T& lc(const polynom<T>& p) { return p.coefficient (p.degree()); }

```

qui elle-même nécessite un accès en lecture aux valeurs des coefficients :

```

template <typename T>
const T& polynom<T>::coefficient (size_t i) const { return c[i]; }

```

Voici un exemple de programme d'appel qui imprime le quotient et le reste de la division de  $a = 3X^3 + X^2 + X + 5$  par  $b = 5X^2 - 3X + 1$ .

## polynom\_div\_tst.cc

---

```

#include "polynom_div.h"
using namespace std;
int main() {
    polynom<double> X = symbol(),
        a = 3*X*X*X + X*X + X + 5,
        b = 5*X*X - 3*X + 1,
        q, r;
    div (a,b,q,r);
    cout << "a = " << a << endl
        << "b = " << b << endl
        << "q = " << q << endl
        << "r = " << r << endl
        << "b*q+r = " << b*q+r << endl;
}

```

## 8.7 Exercices

EXERCICE 8.7.1 (dérivation des polynômes)

- a) **Définir** la fonction membre `diff()` de la classe `polynom`, qui renvoient la dérivée d'un polynôme.
- b) **Adapter** le programme `polynom_legendre_tst`, pour imprimer les dérivées des polynômes de Legendre.

EXERCICE 8.7.2 (intégration des polynômes)

- a) **Définir** la fonction membre `integrate()` de la classe `polynom`, qui renvoient une primitive d'un polynôme, associée à une constante d'intégration nulle.
- b) **Définir** la fonction `integrate(p,a,b)` qui intègre un polynôme dans l'intervalle  $[a, b]$ .
- c) **Adapter** le programme `polynom_legendre_tst`, pour imprimer les quantités

$$\int_{-1}^1 L_i(x) L_j(x) dx, \quad 0 \leq i, j \leq n$$

et vérifier que cela fait  $2/(2i+1)\delta_{i,j}$ . Pour une démonstration de cette propriété, on pourra consulter par exemple [CM86, p. 21], exercice 1.6-2.

EXERCICE 8.7.3 (optimisation de la classe `polynom` - difficile)

La représentation du monôme  $X^{500}$  nécessite un tableau contenant 499 zéros pour un coefficient non-nul. Une représentation informatique plus efficace ne contient que les coefficients non-nuls sous la forme d'une liste de paires  $\{(i, c_i), 0 \leq i \leq n \text{ et } c_i \neq 0\}$ . Ce type de structure de données compressé a été adoptée pour les matrices creuses. La structure `list<T>` de la librairie standard du langage C++ permet de représenter un tel ensemble :

```
template <typename T>
class polynom {
public:
// ..
protected:
    std::list<std::pair<size_t, T> > c;
};
```

La classe `pair<T1,T2>` fait également partie de la librairie standard. **Re-écrire** les fonctions membres et amies de la classe polynôme : constructeurs, affectation, arithmétique... **Tester** les programme d'affichage des polynômes de LEGENDRE ainsi que de la division euclidienne. Afin d'assurer des performances aux opérations d'addition, soustraction et multiplication, nous maintiendrons la liste des coefficients ordonnée par indices  $i$  croissants.

# Chapitre 9

## Différentiation automatique

On considère le problème de minimisation :

(P) : trouver  $u \in \mathbb{R}$  solution de

$$\min_{v \in \mathbb{R}} J(v)$$

avec

$$\begin{aligned} F(v) &= 2u(u + 1) \\ J(v) &= F(v) \times (F(v) + \sin(v)) \end{aligned}$$

Comme nous pouvons l'observer sur la Fig. 9.1, la fonction  $J(v)$  est assez difficile à minimiser. En particulier, elle présente ici des minimums locaux en  $u = 0$  et  $u \approx -1.20662$ , le second étant un minimum global. Le codage de cette fonction est :

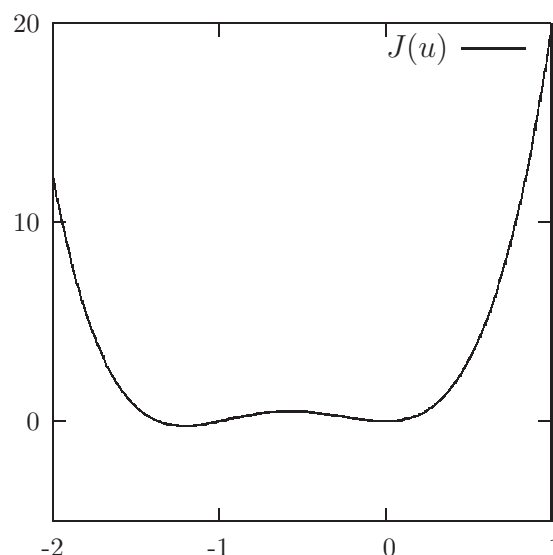


FIG. 9.1 – La fonction  $J(u)$ .

j.h

---

```
struct J {
  template <class T>
  T operator() (const T& u) const {
    T f = 2*u*(u+1);
    return f*(f + sin(u));
  }
};
```

Un petit programme d'affichage s'écrira :



---

j\_tst.cc

---

```

#include "j.h"
#include <cmath>
#include <iostream>
using namespace std;
int main() {
    J j;
    double u = -2;
    for (size_t i = 0; i <= 300; i++, u = u + 0.01)
        cout << u << " " << j(u) << endl;
}

```

La compilation et le lancement sont :

```

c++ j_tst.cc -o j_tst
./j_tst > j.dat
gnuplot
> plot "j.dat" w lp

```

La méthode de NEWTON permet de rechercher un zéro de  $J'(v)$  en construisant une suite  $(u_n)_{n \geq 0}$  selon :

$n = 0 : u_0 \in \mathbb{R}$  donné  
 $n \geq 0 : u_n \in \mathbb{R}$  connu, calculer  
 $u_{n+1} := u_n - (J''(u_n))^{-1} \times J'(u_n)$

Cette méthode converge donc vers un des minimums locaux de  $J$ , selon le choix initial de  $u_0$ . Une programmation générique de cette méthode est :

---

newton.h

---

```

#include <iostream>
template <typename T, typename F1, typename F2>
int newton (const F1& f, const F2& df_dx, T& x, size_t n_max, const T& epsilon) {
    for (size_t n = 0; n < n_max; n++) {
        T dx = - f(x)/df_dx(x);
        std::clog << "newton : " << n << " " << dx << std::endl;
        if (fabs(dx) < epsilon) return 0;
        x += dx;
    }
    return 1;
}

```

Cet algorithme cherche  $x$  solution de  $f(x) = 0$ . Les fonctions **f** et **df\_dx** sont passées en argument, l'algorithme s'arrête en au plus  $n_{max}$  itérations, et le test d'arrêt  $|u_{n+1} - u_n| < \epsilon$ .

Il restera donc à appeler cette méthode avec **f**= $J'$  et **df\_dx**= $J''$ . Une façon de calculer  $J'(u)$  est de l'approcher par la différence  $(J(u + \epsilon) - J(u)) / \epsilon$ . Cependant, cette quantité est sensible au choix de  $\epsilon$ , et l'algorithme résultant ne sera pas très robuste, d'autant plus qu'il faudra également évaluer  $J''(u)$  de façon approchée. Les erreurs peuvent s'accumuler.

Dans la pratique, l'algorithme est beaucoup plus robuste lorsque nous évaluons exactement  $J'(u)$  et  $J''(u)$ . Une meilleure solution est donc de calculer explicitement  $J'$  et  $J''$ , puis

d'écrire les classes-fonctions associées, comme cela a été fait pour la fonction  $J$  :

---

dj\_du.h

---

```

struct dJ_du {
    template <class T>
    T operator() (const T& u) const {
        T f = 2*u*(u+1);
        T df_du = 4*u+2;
        return df_du*(f + sin(u)) + f*(df_du + cos(u));
    }
};

```

---

d2j\_du2.h

---

```

struct d2J_du2 {
    template <class T>
    T operator() (const T& u) const {
        T f = 2*u*(u+1);
        T df_du = 4*u+2;
        T d2f_du2 = 4;
        return d2f_du2*(f + sin(u)) + 2*df_du*(df_du + cos(u)) + f*(d2f_du2 - sin(u));
    }
};

```

Enfin l'appel de la minimisation de  $J$  par la méthode de NEWTON s'écrit alors :

---

newton\_tst.h

---

```

#include "newton.h"
#include "j.h"
#include "dj_du.h"
#include "d2j_du2.h"
#include <cmath>
using namespace std;
int main(int argc, char**argv) {
    double u = (argc > 1) ? atof(argv[1]) : -2;
    newton (dJ_du(), d2J_du2(), u, 100, 1e-7);
    J j;
    cout << "u = " << u << endl
         << "J(u) = " << j(u) << endl;
    return 0;
}

```

La compilation et le test sont :

```

c++ newton_tst.cc -o newton_tst
./newton_tst

```

Cependant, l'expression de  $J$  peut être arbitrairement complexe, des milliers de lignes de code dans la pratique, et issu d'un calcul numérique, différences finies, éléments finis, ... Dans ce cas, le calcul de  $J'$  et  $J''$  sera fastidieux et source d'erreur : pour chaque

modification de  $J$ , lors de mise à jours du code, il faudra les répercuter dans les codes de calcul de  $J'$  et de  $J''$ . Une première technique, en vogue ces derniers temps, consiste à générer automatiquement les fichiers `dj_du.h` et `d2j_du2.h` à partir du fichier `j.h`. Un tel programme de génération automatique sera relativement complexe, car il devra reconnaître toutes les constructions du langage C++, puis dériver toutes les expressions. Ainsi, le logiciel TAPENADE (précédemment appelé ODYSSEE) réalise ce type de conversion. La procédure reste longue et pénible.

Nous allons utiliser une autre technique, plus récente, de dérivation automatique de code qui s'applique sur chaque instruction : si une fonction est décrite par son implémentation numérique, alors ses dérivées seront calculables exactement. Pour cela, considérons la classe :

`ad.h`

---

```
#include <iostream>
template <typename T>
class ad {
public :
    ad (const T& x = T(), const T& dx = T());
    ad<T>& operator= (const ad<T>&);
    const T& diff() const;
    void declare_variable();
    ad<T> operator- () const;
    ad<T> operator+ (const ad<T>&) const;
    ad<T> operator+ (double g) const;
    ad<T> operator* (const ad<T>&) const;
    template <typename U> friend ad<U> operator* (double, const ad<U>&);
    template <typename U> friend ad<U> sin (const ad<U>&);
    template <typename U> friend ad<U> cos (const ad<U>&);
    template <typename U> friend ad<U> exp (const ad<U>&);
    template <typename U> friend std::ostream& operator<< (std::ostream&, const ad<U>&);
protected :
    T x, dx;
};
```

La *surcharge* de l'opérateur de multiplication `operator*` entre deux variables  $f$  et  $g$  de type `ad<double>` entrainera que, chaque fois que  $f \times g$  apparaît, nous calculerons simultanément  $fg = f.x * g.x$  et  $f'g + fg' = f.dx * g.x + f.x * g.dx$ . Un objet de type `ad<double>` est par défaut une constante. L'instruction `x.is_variable()` affecte 1 au champs `x.dx` : la dérivée de  $x$  par rapport à  $x$  vaut 1. Par exemple, le code :

`ad_tst.cc`

---

```
#include "ad.h"
using namespace std;
int main() {
    ad<double> x = 3;
    x.declare_variable();
    ad<double> f = 2*x*(x+1);
    cout << f << " " << f.diff() << endl;
}
```

affichera les valeurs de  $f(x)$  et de sa dérivée  $f'(x)$  en  $x = 3$ , soit 24 et 14, respectivement. La compilation et le test sont :

```
c++ ad_tst.cc -o ad_tst
./ad_tst
```

Avant de pouvoir compiler et exécuter ce code, il va falloir cependant compléter notre classe `ad<T>`. Les constructeurs et opérateurs d'affectation s'écrivent :

ad.h (suite)

---

```
template <class T> ad<T>::ad (const T& y, const T& dy) : x(y), dx(dy) {}
template <class T> ad<T>& ad<T>::operator= (const ad<T>& f) {
    x = f.x;
    dx = f.dx;
    return *this;
}
```

Les fonctions d'accès aux données sont :

```
template <class T> const T& ad<T>::diff() const { return dx; }
template <class T> void ad<T>::declare_variable() { dx = T(1); }
```

S'appuyant sur les formules de dérivation de  $(f+g)'$  de l'addition, nous pouvons commencer à compléter le code des fonctions de la classe `ad<T>` :

```
template <class T>
ad<T> ad<T>::operator+ (const ad<T>& g) const { return ad<T>(x + g.x, dx+g.dx); }
template <class T>
ad<T> ad<T>::operator+ (double g) const { return ad<T>(x+g, dx); }
template <class T>
ad<T> ad<T>::operator- () const { return ad<T>(-x, -dx); }
```

La formule de dérivation pour  $(f \times g)'$  permet de définir `operator=` :

```
template <class T>
ad<T> ad<T>::operator* (const ad<T>& g) const { return ad<T>(x*g.x, dx*g.x + x*g.dx); }
template <class T>
ad<T> operator* (double f, const ad<T>& g) { return ad<T>(f*g.x, f*g.dx); }
```

La librairie mathématique courante est aussi introduite :

```
#include <cmath>
template <class T>
ad<T> sin (const ad<T>& f) { return ad<T>(sin(f.x), f.dx*cos(f.x)); }
template <class T>
ad<T> cos (const ad<T>& f) { return ad<T>(cos(f.x), -f.dx*sin(f.x)); }
template <class T>
ad<T> exp (const ad<T>& f) { return ad<T>(exp(f.x), f.dx*exp(f.x)); }
```

Enfin, la sortie standard :

```
template <class T>
std::ostream& operator<< (std::ostream& os, const ad<T>& f) {
    return os << f.x;
}
```

Cet inventaire n'est certes pas complet, mais sera suffisant pour nous permettre de tester quelques fonctionnalités de notre classe. Le petit programme suivant affiche sur trois colonnes les valeurs de  $(u, J(u), J'(u))$  entre  $-2$  et  $1$  par pas de  $0.01$  :

---

ad\_j1\_tst.cc

---

```
#include "j.h"
#include "ad.h"
using namespace std;
int main() {
    J j;
    ad<double> u = -2;
    u.declare_variable();
    for (size_t i = 0; i <= 300; i++, u = u + 0.01) {
        ad<double> ju = j(u);
        cout << u << " " << ju << " " << ju.diff() << endl;
    }
}
```

Afin d'utiliser, lors des appels la fonction  $J$ , à l'aide d'un argument  $u$  de type `ad<double>`, nous l'avons définie une fois pour toute à l'aide d'un type générique  $T$ . D'autre part, nous n'avons à aucun moment à utiliser explicitement les expressions des dérivées de  $J$ . La compilation et le test sont :

```
c++ ad_j1_tst.cc -o ad_j1_tst
./ad_j1_tst > j1.dat
gnuplot
> plot "j1.dat" w lp, "j1.dat" u 1:3 w lp
```

Ceci va nous permettre de calculer les dérivées secondes, voire d'ordre supérieur, de façon automatique : pour cela, nous utiliserons la capacité de *récurssion* les modèles de types : `ad<ad<double> >`, `ad<ad<ad<double> > >`...

---

ad\_j2\_tst.cc

---

```
#include "j.h"
#include "ad.h"
using namespace std;
int main() {
    J j;
    ad<double> u0 = -2;
    u0.declare_variable();
    ad<ad<double> > u (u0);
    u.declare_variable();
    for (size_t i = 0; i <= 300; i++, u = u + 0.01) {
        ad<ad<double> > ju = j(u);
        cout << u << " " << ju << " " << ju.diff() << " " << ju.diff().diff() << endl;
    }
}
```

La syntaxe devient alors assez complexe, et donc présente une source potentielle d'erreurs de programmation. Aussi, afin de rendre notre code plus lisible, nous allons utiliser la petite

classe de derivation automatique des classes-fonctions à une variable suivante :

---

ad\_diff.cc

---

```
#include "ad.h"
template <typename F>
class diff {
public :
    diff() : f() {}
    template<class T>
    T operator() (const T& x0) const {
        ad<T> x (x0);
        x.declare_variable();
        return f(x).diff();
    }
protected :
    F f;
};
```

Le code précédent peut se re-écrire plus clairement :

---

ad\_j2a\_tst.cc

---

```
#include "j.h"
#include "ad_diff.h"
using namespace std;
int main() {
    J j;
    diff<J> dj_du;
    diff<diff<J> > d2j_du2;
    double u = -2;
    for (size_t i = 0; i <= 300; i++, u = u + 0.01)
        cout << u << " " << j(u) << " " << dj_du(u) << " " << d2j_du2(u) << endl;
}
```

L'appel de la minimisation de  $J$  par la méthode de NEWTON s'écrit alors :

---

newton\_ad\_tst.cc

---

```
#include "ad_diff.h"
#include "newton.h"
#include "j.h"
using namespace std;
int main(int argc, char**argv) {
    double u = (argc > 1) ? atof(argv[1]) : -2;
    newton (diff<J>(), diff<diff<J> >(), u, 100, 1e-7);
    J j;
    cout << "u = " << u << endl
        << "J(u) = " << j(u) << endl;
    return 0;
}
```

La compilation et le test sont :

```
g++ newton_ad_tst.cc -o newton_ad_tst
./newton_ad_tst
```

## Notes bibliographiques

Pour en savoir plus sur ces techniques très en vogue actuellement, on pourra consulter les librairies `fabbad` [SB05] ou `adol-c` [WKG05] par exemple. Cette technique s'applique encore quand  $J$  est issue d'un calcul approché : traitement d'image, discrétisation d'équations aux dérivées partielles, etc.

# Chapitre 10

## Entiers arbitrairement grands

Nous souhaitons maintenant pouvoir effectuer des opérations arithmétiques sur des valeurs entières positives de grandes taille pour lesquelles nous ne pouvons utiliser les types `int` ou `long int` qui sont trop petits, limités à  $2^N$  où typiquement  $N = 31$  ou  $N = 63$ . Nous pourrions représenter nos grands nombres positifs en base 10, à raison d'un chiffre par élément, mais ceci ne serait pas une façon très efficace de faire les choses puisque les types numériques existent déjà et peuvent contenir des valeurs plus grandes qu'un chiffre. Nous allons découper un entier arbitrairement grand en tranche de quatre chiffres, chaque tranche étant un élément d'une liste : la représentation de l'entier 12345678901 sera la liste : (0123, 4567, 8901).

Nous allons utiliser la classe `list` de la librairie standard du C++.

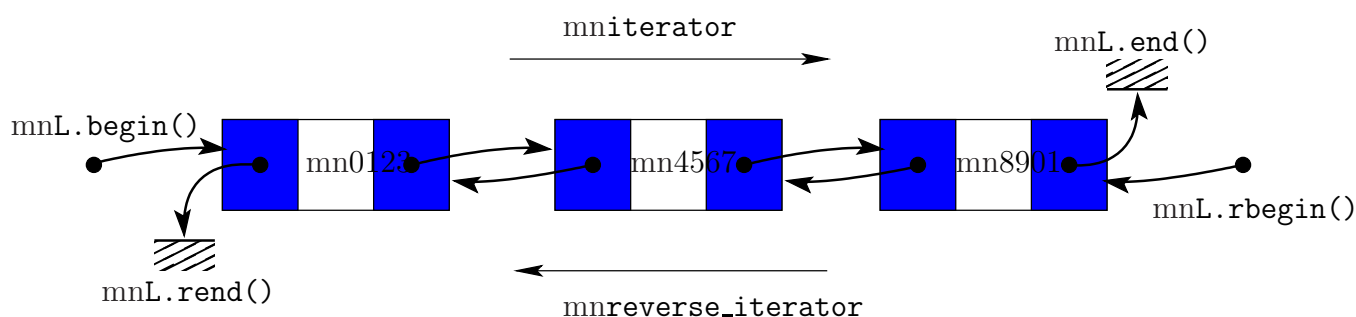


FIG. 10.1 – Liste chaînée pour les grands entiers.



## bigint.h

---

```

#include <list>
#include <string>
#include <iostream>
class bigint {
public :
    bigint();
    bigint(const char* str);
    bigint(const std::string& str);
    bigint operator+ (const bigint& j) const;
    friend std::istream& operator>> (std::istream& is, bigint& i);
    friend std::ostream& operator<< (std::ostream& os, const bigint& i);
protected :
    static const size_t n_chiffre_tranche = 4;
    static const size_t limite_tranche = 10000; // 10^(n_chiffre_tranche+1)
    void build_from_string(const std::string& str);
    std::list<size_t> l;
};

```

La classe `list` présente un interface du type :

## &lt;list&gt;

---

```

template <class T>
class list {
    list();
    list(const list<T>&);
    list<T>& operator= (const list<T>&);
    ~list();
    void push_front (const T&);
    void push_back (const T&);
    typedef iterator;
    iterator begin () const;
    iterator end () const;
    typedef const_iterator;
    const_iterator begin () const;
    const_iterator end () const;
    typedef reverse_iterator;
    const_iterator begin () const;
    reverse_iterator rbegin () const;
    reverse_iterator rend () const;
    typedef const_reverse_iterator;
    const_reverse_iterator rbegin () const;
    const_reverse_iterator rend () const;
};

```

Il existe des itérateurs qui parcourent la liste  $(i_0, \dots, i_{n-1})$  dans l'ordre normal ou inverse : `iterator` et `reverse_iterator`. Nous distinguons aussi les itérateurs qui sont capables de modifier les données de ceux qui ne travaillent qu'en lecture, et qui prennent un préfixe `const`. Ceci fait au total quatre types d'itérateurs. Le parcours d'une liste  $L$  de type `list` s'écrira :

```

    for (list<size_t>::iterator iter = L.begin(); iter != L.end(); iter++)
        cout << *iter << endl;

```

Les itérateurs reprennent la syntaxe des pointeurs dont ils sont la généralisation. Les opérations `push_front` et `push_back` nous permettrons d'ajouter un élément à la liste, respectivement en tête et en queue. Avec cette structure de données, l'addition s'écrit :

```

bigint bigint::operator+ (const bigint& j) const {
    bigint k;
    std::list<size_t>::const_reverse_iterator iter_i = l.rbegin();
    std::list<size_t>::const_reverse_iterator iter_j = j.l.rbegin();
    size_t retenue = 0;
    while (iter_i != l.rend() || iter_j != j.l.rend()) {
        size_t tranche_i = (iter_i != l.rend())? *iter_i++ : 0;
        size_t tranche_j = (iter_j != j.l.rend())? *iter_j++ : 0;
        size_t partielle = tranche_i + tranche_j + retenue;
        if (partielle > limite_tranche - 1) {
            k.l.push_front (partielle - limite_tranche);
            retenue = 1;
        } else {
            k.l.push_front (partielle);
            retenue = 0;
        }
    }
    if (retenue == 1) k.l.push_front (1);
    return k;
}

```

Le constructeur remplit la liste à partir d'une chaîne de caractère :

```

bigint::bigint() : l() {}
bigint::bigint(const char* str) : l() { build_from_string(str); }
bigint::bigint(const std::string& str) : l() { build_from_string(str); }

```

```

#include <sstream>
#include <cmath>
void bigint::build_from_string(const std::string& str) {
    size_t n = str.length();
    size_t n_tranche = size_t(std::ceil(1.0*n/n_chiffre_tranche));
    for (size_t i = 0; i < n_tranche; i++) {
        size_t start = n - std::min(n, (i+1)*n_chiffre_tranche);
        size_t last = n - i*n_chiffre_tranche;
        std::string tranche (str, start, last-start);
        std::istringstream is (tranche);
        size_t tranche_i;
        is >> tranche_i;
        l.push_front(tranche_i);
    }
}

```

La sortie standard est :

```
std::ostream& operator<< (std::ostream& os, const bigint& i) {
    size_t c = 0;
    for (std::list<size_t>::const_iterator p = i.l.begin(); p != i.l.end(); p++, c++) {
        if (c == 0) {
            os << *p;
        } else {
            size_t tranche = *p;
            size_t deno = bigint::limite_tranche/10;
            for (size_t k = bigint::n_chiffre_tranche; k > 0; k--, deno /= 10) {
                os << tranche/deno;
                tranche = tranche - deno*(tranche/deno);
            }
        }
    }
    return os;
}
```

Enfin, voici un petit programme de test :

bigint\_tst.cc

---

```
#include "bigint.h"
using namespace std;
int main (int argc, char** argv) {
    bigint i = string((argc > 1)? argv[1] : "10001");
    bigint j = string((argc > 2)? argv[2] : "10002");
    cout << i << " + " << j << " = " << i+j << endl;
}
```

# Bibliographie

- [BBC<sup>+</sup>94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems : building blocks for iterative methods*. SIAM, 1994.
- [Bro97] M. Bronstein. *Symbolic Integration*. Springer, 1997.
- [CM86] M. Crouzeix and A. L. Mignot. *Exercices d'analyse numérique des équations différentielles*. Masson, 1986.
- [Deu01] L. P. Deutsch. *Ghostsript : Postscript and PDF language interpreter and previewer*. <http://www.cs.wisc.edu/~ghost>, 2001.
- [Deu02] L. P. Deutsch. *Matrix market*. <http://math.nist.gov/MatrixMarket/>, 2002.
- [DLPR01] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. *IML++ v. 1.2 Iterative Method Library*. <http://math.nist.gov/iml++>, 2001.
- [Fon97] A. B. Fontaine. *La bibliothèque standard du C++*. InterEditions, 1997.
- [FSF05] Free Software Foundation FSF. *GMP : The GNU Multiple Precision Arithmetic Library*. <http://www.swox.com/gmp/>, 2005.
- [Hec97] F. Hecht. *Bidimensional anisotropic mesh generator*. INRIA, <http://www-rocq.inria.fr/gamma/cdrom/www/bamg/eng.htm>, 1997.
- [LT93] P. Lascaux and R. Théodor. *Analyse numérique appliquée à l'art de l'ingénieur (2 Tomes)*. Masson, 1993.
- [MS96] D. R. Musser and A. Saini. *STL tutorial and reference guide*. Addison-Wesley, 1996.
- [PTVF94] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C, second edition*. Cambridge university press, 1994.
- [saa94] Y. saad. Sparskit. <http://www.cs.umn.edu/~saad>, 1994.
- [SB05] O. Stauning and C. Bendtsen. *fadbad++ : flexible automatic differentiation using template and operator overloading in ANSI C++*. <http://www2.imm.dtu.dk/~km/FADBAD>, 2005.
- [SGI02] SGI. *Standard template library programmer's guide*. [http://www.sgi.com/tech/stl/table\\_of\\_contents.html](http://www.sgi.com/tech/stl/table_of_contents.html), 2002.

- [SR02a] Pierre Saramito and Nicolas Roquet. Rheolef home page. <http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/>, 2002.
- [SR02b] Pierre Saramito and Nicolas Roquet. Rheolef users manual. Technical report, LMC-IMAG, 2002. <http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef/usrman.ps.gz>.
- [Str01] B. Stroustrup. *Le langage C++*. CampusPress, 2001. <http://www.research.att.com/~bs/>.
- [WKG05] A. Walther, A. Kowarz, and A. Griewank. *Adol-c : a package for automatic differentiation of algorithms written in C/C++*. <http://www.math.tu-dresden.de/wir/project/adolc>, 2005.
- [WKL<sup>+</sup>98] T. Williams, C. Kelley, R. Lang, D. Kotz, J. Campbell, G. Elber, and A. Woo. *gnuplot version 3.6 pre-release*. <http://www.uni-karlsruhe.de/~ig25/gnuplot-faq/>, 1998.

# Table des matières

<b>1</b>	<b>Introduction : les quaternions</b>	<b>3</b>
<b>2</b>	<b>Introduction aux algorithmes génériques</b>	<b>9</b>
2.1	L'algorithme du gradient conjugué . . . . .	10
2.2	La classe <code>valarray&lt;T&gt;</code> . . . . .	11
<b>3</b>	<b>Problème de POISSON en dimension un</b>	<b>15</b>
3.1	Les différences finies pour le problème de POISSON . . . . .	15
3.1.1	Présentation du problème . . . . .	15
3.1.2	La résolution du système tridiagonal . . . . .	16
3.1.3	Le programme de tests . . . . .	17
3.2	Le cas d'une subdivision non-régulières . . . . .	18
3.2.1	L'approximation variationnelle . . . . .	18
3.3	La classe des matrices tridiagonales symétriques . . . . .	22
3.4	De l'intérêt des subdivisions non-uniformes . . . . .	24
<b>4</b>	<b>Problème de POISSON en dimension deux</b>	<b>29</b>
4.1	Les différences finies en grilles régulières . . . . .	29
4.2	Une résolution itérative du problème . . . . .	30
4.2.1	La méthode du gradient conjugué . . . . .	30
4.2.2	Préconditionnement . . . . .	32
4.3	Méthode directe par décomposition . . . . .	36
4.3.1	Analyse de FOURIER . . . . .	36
4.3.2	La transformée en sinus . . . . .	42

4.4	Exercices . . . . .	42
<b>5</b>	<b>Matrices creuses</b>	<b>44</b>
5.1	La structure de données . . . . .	44
5.2	Le produit matrice-vecteur . . . . .	48
5.3	Application au gradient conjugué . . . . .	48
5.4	Exercices . . . . .	49
<b>6</b>	<b>Maillages</b>	<b>51</b>
6.1	Les éléments . . . . .	52
6.2	Les sommets . . . . .	53
6.3	La classe maillage . . . . .	53
6.4	Test des entrées-sorties . . . . .	55
6.5	Exercices . . . . .	56
<b>7</b>	<b>Méthode des éléments finis</b>	<b>57</b>
7.1	Une interpolation par éléments . . . . .	57
7.2	La re-numérotation des sommets . . . . .	59
7.3	Le calcul des matrices élémentaires . . . . .	60
7.3.1	Calcul de la matrice de masse $M$ . . . . .	60
7.3.2	Calcul de la matrice d'énergie $A$ . . . . .	61
7.4	L'assemblage des matrices creuses . . . . .	62
7.5	Retour sur la matrice d'énergie $A$ . . . . .	64
7.6	Résolution du système linéaire . . . . .	65
7.7	Exercices . . . . .	67
<b>8</b>	<b>Introduction au calcul formel</b>	<b>69</b>
8.1	Définition d'une classe pour les polynômes . . . . .	69
8.2	Addition, soustraction et multiplication . . . . .	70
8.3	Polynômes de LEGENDRE . . . . .	73
8.4	Évaluation . . . . .	74
8.5	Sortie graphique . . . . .	75

<b>Langage C++ et calcul scientifique</b>	<b>95</b>
8.6 Division euclidienne . . . . .	76
8.7 Exercices . . . . .	78
<b>9 Différentiation automatique</b>	<b>79</b>
<b>10 Entiers arbitrairement grands</b>	<b>87</b>
<b>Table des matières</b>	<b>93</b>
<b>Index des concepts</b>	<b>96</b>
<b>Index des fichiers</b>	<b>98</b>
<b>Index des fonctions</b>	<b>100</b>
<b>Index des noms célèbres</b>	<b>101</b>
<b>Liste des figures</b>	<b>102</b>



# Index des concepts

`#include` imbriqués, 23

classe définit dans le livre

`ad`, 82

`bigint`, 87

`col_type`, 12

`diff`, 85

`element`, 52

`elt_type`, 12

`eye`, 10

`fd_ssor_2d`, 32

`matrix_store`, 35, 62

`matrix`, 45, 62, 63

`mesh`, 53

`point`, 53

`poisson_2d_sinus`, 40

`polynom`, 70

`quaternion`, 4

`row_type`, 12

`strid`, 22

`symbol`, 69

classe dérivée, 54

classe de la bibliothèque standard C++

`complex`, 4

`istream`, 7, 90

`istringstream`, 89

`list`, 88

`map`, 62

`ostream`, 7, 74, 83, 90

`pair_type`, 63

`valarray`, 11, 12, 22

`vector`, 54, 70, 73

classe fonction, 75

coût en temps de calcul, 67

complexe, nombre, 3

constructeur de copie, 5

constructeur et conversion implicite de  
type, 5

constructeur par défaut, 5

convergence, 42, 67

conversion implicite de type, 5

dérivation, 78

degrés de liberté, 58, 59

dictionnaire, 62

entier de taille arbitraire, 87

espace de nom `std` : `:`, 5

factorisation  $LDL^T$ , 16

formule de quadrature

formule des trapèzes, 19, 60

généricité, 10, 16

gradient conjugué, 10, 67

graphique, 43

héritage, 54

intégration, 78

itérateur, 63

liste, 87

logiciel

`IML++`, 10

`bang`, 55, 65

`gnuplot`, 43, 56, 67, 75, 80, 84

`rheolef`, 57

`sparskit`, 49

méthode

- des éléments finis, 57
- des différences finies, 12, 15, 21
- des volumes finis, 60
- méthode itérative de résolution, 10, 21, 32
- maillage, 51, 57, 65
  - adapté, 25
- maillages
  - structurés, 14
- matrice
  - élémentaire, 59
  - creuse, 44, 59, 62, 65
    - classe `matrix_store`, 62
    - classe `matrix`, 45
  - format de fichier *matrix-market* '.mm', 47, 49
  - identité, classe `eye`, 10
  - préconditionnement, 10
    - classe `eye`, 10
    - classe `fd_ssor_2d`, 32
  - triangulaire inférieure, 16
  - tridiagonale
    - classe `strid`, 22
    - factorisation  $LDL^T$ , 16
    - valeurs et vecteurs propres, 36
- polynôme
  - de TCHEBYSHEV, 37
- polynômes
  - de LEGENDRE, 73
- préconditionnement, méthode itérative de résolution, 10, 21, 32
- quaternion, nombre, 3
- récurion dans les modèles de type, 84
- sommet interne, 55, 58, 59
- table de correspondance, 62
- transformée
  - de FOURIER rapide, 39, 42
  - en sinus, 42

# Index des fichiers

<cmath>, 89  
<iostream>, 7, 74, 83, 90  
<list>, 88  
<sstream>, 89  
<valarray>, 12, 22, 30  
<vector>, 54, 70, 73  
cg.h, 9  
matrix\_cg\_tst.cc, 48  
matrix\_ps\_tst.cc, 65  
matrix\_tst.cc, 47  
mesh\_tst.cc, 55  
<complex>, 4  
<list>, 88  
<valarray>, 11  
ad.h, 82  
ad\_diff.cc, 85  
ad\_j1\_tst.cc, 84  
ad\_j2\_tst.cc, 84  
ad\_j2a\_tst.cc, 85  
ad\_tst.cc, 82  
assembly\_energy.h, 64  
assembly\_energy\_tst.c, 64  
assembly\_mass.h, 60  
bigint.h, 87  
bigint\_tst.cc, 90  
d2j\_du2.h, 81  
dirichlet.h, 65  
dj\_du.h, 81  
element.h, 52  
eye.h, 10  
fd\_adapt\_tst.cc, 25  
fd\_energy\_1d.h, 20  
fd\_mass\_1d.h, 20  
fd\_ssor\_2d.h, 32  
fd\_uniform\_tst.cc, 17  
finite\_element\_tst.cc, 67  
fixed\_array.h, 51  
interpolate.h, 58  
interpolate\_1d.h, 18  
interpolate\_2d.h, 30  
j.h, 79  
j\_tst.cc, 79  
ldlt\_solve\_1d.h, 16  
matrix.h, 45  
matrix\_store.h, 62  
mesh.h, 53  
newton.h, 80  
newton\_ad\_tst.cc, 85  
newton\_tst.h, 81  
point.h, 53  
poisson\_2d\_pcg\_tst.cc, 33  
poisson\_2d\_sinus.h, 40  
poisson\_2d\_tst.cc, 31  
polynom.h, 69  
polynom\_div.h, 76  
polynom\_div\_tst.cc, 77  
polynom\_legendre.h, 73  
polynom\_legendre\_plot\_tst.cc, 75  
polynom\_legendre\_tst.cc, 73  
polynom\_plot.h, 75  
quaternion.h, 4  
quaternion\_tst.cc, 8  
range.h, 12  
range\_tst.h, 13  
renumbering.h, 59  
strid.h, 22  
u\_exact\_singular.h, 24  
valarray\_tst.cc, 11

valarray\_util.h, 11

# Index des fonctions

assembly\_energy, 64  
assembly\_mass, 61  
cg, 9  
convert, 63  
dirichlet, 66  
fd\_energy\_1d, 20  
fd\_mass\_1d, 20  
interpolate, 18, 30, 58  
ldlt\_1d, 16  
ldlt\_solve\_1d, 16  
newton, 80, 81, 85  
polynom\_div, 76  
polynom\_legendre, 73  
range, 12  
renumbering, 59

# Index des noms célèbres

DIRICHLET, 15

EUCLIDE, 76, 78

EVAN, 32

FOURIER, 36, 39, 42

HAMILTON, 3

LEGENDRE, 73, 75, 78

NEWTON, 80

POISSON, 15, 29, 57

TSCHEBYSCHEV, 37

# Table des figures

3.1	Fonction de base $\varphi_i$ . . . . .	18
3.2	Aspect de la solution $u_\alpha(x)$ sur $[-1, 1]$ suivant $\alpha$ . . . . .	24
3.3	Approximation sur les subdivisions uniformes et adaptées ( $n = 40, \alpha = 1/2$ ). . . . .	27
3.4	Convergence vers $u(x) = (1 - x^2)^{1/2}$ pour les subdivisions uniformes et adaptées. . . . .	28
4.1	Gradient conjugué : effet du préconditionnement sur la convergence du résidu du problème de POISSON pendant les itérations (grille $200 \times 200$ : 40 000 inconnues). . . . .	35
4.2	Gradient conjugué préconditionné : nombre d'itérations $\mathcal{N}(n, \varepsilon)$ nécessaire, sur une grille $n \times n$ , pour obtenir la précision $\varepsilon = 10^{-15}$ . . . . .	35
6.1	Maillage d'un cercle. . . . .	51
7.1	Fonction de base $\varphi_i$ . . . . .	58
7.2	Volume fini $C_i$ centré autour du sommet $i$ . . . . .	60
7.3	Rotation autour de $K_l$ suivant $(i, p, q)$ dans le sens positif. . . . .	62
7.4	Maillage et structure creuse de la matrice d'énergie associée. . . . .	66
7.5	Convergence de la méthode des éléments finis (a) ; nombre d'itération du GC en fonction de la taille du problème (b). . . . .	68
8.1	Évaluation d'une expression polynomiale de la forme $c * X * p$ . . . . .	72
8.2	Polynôme de LEGENDRE pour $n = 9$ . . . . .	76
9.1	La fonction $J(u)$ . . . . .	79
10.1	Liste chaînée pour les grands entiers. . . . .	87