



# Langage C++ et calcul scientifique

Pierre Saramito

## ► To cite this version:

Pierre Saramito. Langage C++ et calcul scientifique. DEA. Grenoble, France, 2005, pp.151. cel-00573975v3

**HAL Id: cel-00573975**

**<https://cel.hal.science/cel-00573975v3>**

Submitted on 11 Dec 2012 (v3), last revised 16 Dec 2013 (v4)

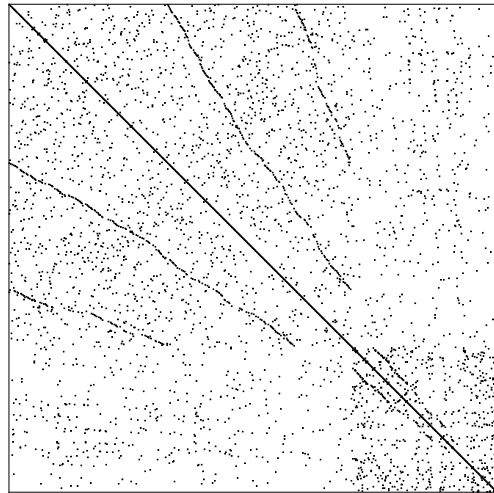
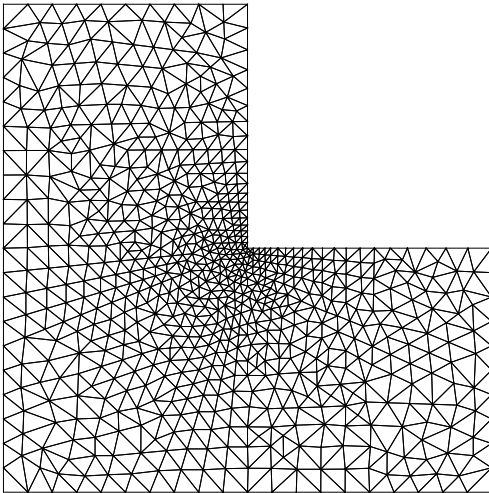
**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

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 11 décembre 2012*



Copyright (c) 2003-2012 Pierre Saramito

Permission vous est donnée de copier, distribuer et/ou modifier ce document selon les termes de la licence *GNU Free Documentation License*, version 1.3 ou ultérieure, publiée par la Free Software Foundation, avec le texte de première et quatrième de couverture. Une copie de cette licence figure dans la section Annexes de ce document (texte original en anglais de la licence GNU FDL).

# Préface

La simulation numérique est devenue essentielle dans de nombreux domaines tels que la mécanique des fluides et des solides, la météo ou l'évolution du climat, la biologie ou les semi-conducteurs. Elle permet de comprendre et de prévoir, d'accéder là où les instruments de mesures s'arrêtent.

Ce livre présente les méthodes performantes du calcul scientifique : matrices creuses, résolution performante des grands systèmes linéaires, ainsi que de nombreuses applications à la résolution par éléments finis, différences finies et méthodes spectrales. Alternant algorithmiques et applications, les programmes sont directement présentés en langage C++. Les codes présentés sont volontairement concis et clairs, et utilisent largement les notions de classe et de généricité du langage C++.

Le contenu de ce livre a fait l'objet de cours de troisième année à l'école nationale supérieure d'informatique et de mathématiques appliquées de Grenoble (ENSIMAG) ainsi qu'au master de mathématiques appliquées de l'université Joseph Fourier. Des connaissances de base d'algèbre matricielle et de programmation sont recommandés. La maîtrise du contenu de ce livre permet de maîtriser les principaux paradigmes de programmation du calcul scientifique et de les appliquer à des problèmes d'intérêt pratique, comme la résolution des équations aux dérivées partielles. La diversité des sujets abordés, l'efficacité des algorithmes présentés et leur écriture directe en langage C++ font de ce livre un recueil fort utile dans la vie professionnelle d'un ingénieur.

Le premier chapitre présente les fondamentales de base pour la suite : présentation du langage C++ à travers la conception d'une classe de quaternions et outils d'analyse asymptotique du temps de calcul des algorithmes. Le second chapitre aborde l'algorithme de transformée de Fourier rapide et développe deux de ses applications à la discrétisation d'équations aux dérivées partielles par la méthode des différences finies. Le troisième chapitre est dédié aux matrices creuses et à l'algorithme du gradient conjugué, et en développe une application à la méthode des éléments finis. En annexe sont groupés des exemples de génération de maillage et de visualisation graphique.

S'il est cependant recommandé de maîtriser les notions du premier chapitre pour aborder le reste du livre, les chapitres deux et trois sont complètement indépendants et peuvent être abordés séparément. Ces chapitres sont complétés par des exercices qui en constituent des développements, ainsi que des notes bibliographiques, retraçant l'historique des travaux et fournissant des références sur des logiciels et librairies récents implémentant ou étendant les algorithmes présentés.

Les codes C++ présentés au long de ce livre sont librement disponibles à l'adresse <http://www-ljk.imag.fr/membres/Pierre.Saramito/books> sous la licence GNU public licence.

# Table des matières

<b>1</b>	<b>Introduction à l'algorithmique numérique en C++</b>	<b>5</b>
1.1	Quaternions . . . . .	5
1.2	Analyse asymptotique des algorithmes . . . . .	14
<b>2</b>	<b>Transformée de Fourier et applications</b>	<b>27</b>
2.1	Transformée de Fourier . . . . .	27
2.2	Discrétisation de problèmes aux limites . . . . .	33
2.3	Application aux différences finies multi-dimensionnelles . . . . .	47
<b>3</b>	<b>Matrices creuses et méthode des éléments finis</b>	<b>61</b>
3.1	Algorithme du gradient conjugué . . . . .	61
3.2	Matrices creuses . . . . .	67
3.3	Maillages . . . . .	78
3.4	Méthode des éléments finis . . . . .	87
<b>A</b>	<b>Pré- et post-traitements</b>	<b>105</b>
A.1	Ordonnancement et visualisation des matrices creuses . . . . .	105
A.2	Génération et visualisation de maillages . . . . .	106
A.3	Visualisation des solutions de type éléments finis . . . . .	110
<b>B</b>	<b>Corrigé des exercices</b>	<b>113</b>
<b>C</b>	<b>GNU free documentation license</b>	<b>133</b>
	<b>Bibliographie</b>	<b>142</b>
	<b>Liste des fichiers d'exemples</b>	<b>146</b>
	<b>Index</b>	<b>149</b>

# Chapitre 1

## Introduction à l’algorithmique numérique en C++

### 1.1 Quaternions

#### 1.1.1 Concept

L’objectif de premier chapitre est d’introduire, à travers un exemple concret, un certain nombre de notions clef du langage C++ qui seront utilisées tout au long de ce livre : conception de classe, classe paramétrée par un type et généricité, surcharge d’opérateur, librairie standard. Pour cela, intéressons nous à une classe représentant des quaternions, une extension des nombres complexes. 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 [45]. 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  $\bar{h} = a\mathcal{U} - b\mathcal{I} - c\mathcal{J} - d\mathcal{K}$ .

### 1.1.2 Implémentation de la classe

La classe `quaternion` peut 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;
};
```

Le premier constructeur prend quatre 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.

Nous avons choisi de représenter un quaternion par deux nombres complexes `zc` et `wc` plutôt que par quatre réels : ce choix permet d'écrire les opérations algébriques de façon plus compacte. Notre choix a été également guidé par la volonté de construire une classe `quaternion` ayant le même type d'interface, et compatible avec la classe `complex` de la librairie standard C++ :

`<complex>`

```
template <typename 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);
    T& real();
    T& imag();
    const T& real() const;
    const T& imag() const;
protected:
    T re, im;
};
```

Remarquons tout d'abord le préfixe `std::` devant `complex` : cette classe est accessible dans la librairie standard via le domaine de nom `std` après inclusion du fichier d'entête correspondant. La classe est paramétrée par le type `T` qui représente le type à virgule flottante approchant les nombres réels, et utilisé pour les parties réelles et imaginaires. Dans le langage C++, il existe pour cela trois types prédéfinis : `float`, `double` et `long double`. Le type `float` est limitée à une précision de sept décimales et le type `double` à quinze décimales. Le type `long double` dépend de la machine, mais a au moins quinze décimales. Sur certaines machines, un `long double` peut représenter une triple ou quadruple précision. Le paramétrage de la classe quaternion par un type `T` nous permet également d'utiliser d'autres classe de nombre à virgule flottante que celles prédéfinies par le langage C++ : de nombreuses librairie C++ proposent des nombres à virgule flottante ayant des propriétés très variées. Ainsi, la librairie `qd` propose des précisions quadruple et octuple [29] très performantes. Dans la librairie GNU multi-précision `gmp`, la précision des nombres à virgule flottante peut être arbitrairement fixée [26] par l'utilisateur, et un grand nombre de décimales deviennent accessibles. Le paramétrage par un type flottant nous permet de donner un caractère plus général à notre classe `quaternion` : il ne sera plus nécessaire de la ré-écrire pour chaque nouveau type de nombre à virgule flottante : c'est le principe de la *généricité*. La classe `complex` est complétée par les opérations d'algèbre usuelles `+`, `-`, `*`, `/` ainsi que `conj(z)` qui renvoie le conjugué `abs(z)` qui renvoie le module et `norm(z)` qui renvoie le carré du module.

Revenons à présent à l'implémentation de notre classe `quaternion`. Les constructeurs et opérateurs d'affectation s'écrivent :

#### quaternion.h (suite)

```
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;
}
```

Étudions à présent les fonctions d'accès aux données `z` et `w` de la classe. Le langage C++ distingue l'accès en *lecture seule* de l'accès en *lecture et écriture*. L'accès en lecture seule est agrémenté du mot-clef `const`, et renvoie une référence constante sur un `complex<T>`. L'accès en lecture et écriture renvoie une référence sur un `complex<T>` sans la restreindre à être constante : ce type d'accès permettra de modifier les données `zc` et `wc` contenues dans la classe. Les fonctions d'accès aux



données `z` et `w` s'écrivent :

quaternion.h (suite)

```
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; }
```

Le langage C++ permet aux utilisateurs de définir eux-même des notations infixées du type `h1+h2`. Ceci se fait à travers la définition d'une fonction appelée `operator+` et l'expression infixée du type `h1+h2` s'écrit également de façon préfixée `operator+(h1,h2)`. L'addition entre deux quaternions est définie simplement par :

quaternion.h (suite)

```
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;
}
```

Notez l'utilisation des accès en lecture seule pour les opérandes `h` et `m`, qui sont situés dans les membres de droite des affectations, ainsi que les accès en lecture et écriture pour `r`, qui est situé dans les membres de gauche des affectations. Les opérateurs de soustraction et de multiplication sont analogues :

quaternion.h (suite)

```
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  $z$ . La fonction `abs` renvoie le module et la fonction `norm` renvoie le carré du module : ceci sert à ne pas inutilement extraire des racines carrées pour ensuite élever au carré. Ces conventions ont été adoptées

pour la classe `complex` et nous les étendons simplement ici aux quaternions.

quaternion.h (suite)

```
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>
T norm (const quaternion<T>& h) {
    return norm(h.z()) + norm(h.w());
}
template <typename T>
T abs (const quaternion<T>& h) {
    return sqrt(norm(h));
}
```

Nous avons  $h\bar{h} = |h|^2$  et l'inverse s'écrit :  $h^{-1} = \bar{h}/|h|^2$ . Ceci va nous permettre d'introduire la division entre deux quaternions :

quaternion.h (suite)

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

La classe `complex` a pour convention d'écrire les parties réelle et imaginaire entre parenthèses et séparées par une virgule. Ainsi,  $z = 3 + 2i$  sera formaté (3,2). Étendons aux quaternions cette convention de la façon suivante : nous écrivons le quaternion  $h = 3 + 2i + 5j + 7k$  en écrivant ses deux nombres complexes, parenthésés et séparés par des virgules, soit ((3,2),(5,7)). La fonction d'écriture écrit alors simplement :

quaternion.h (suite)

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

Remarquez l'utilisation de la classe `ostream` de la librairie standard du C++ : cette classe décrit un flot de sortie. Cette notion étend de façon très souple la notion de descripteur de fichier. L'écriture s'effectue via l'utilisation de l'opérateur infixé `<<` :

nous définissons ici la fonction `operator<<` entre une variable de type `ostream` et un quaternion. Pour implémenter cette fonction, nous utilisons la fonction `operator<<` entre une variable de type `ostream` et un nombre complexe. L'opérateur infixé `<<` a l'avantage de pouvoir s'utiliser en cascade, ce qui permet une programmation très lisible de la fonction d'écriture.

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.

quaternion.h (suite)

```
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;
}
```

Ainsi se termine l'implémentation de la classe quaternion.

### 1.1.3 Utilisation de la classe

Écrivons un petit programme qui teste les fonctionnalités de notre classe :

quaternion\_tst.cc

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

et le test sont données par :

```

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

```

dont le résultat est :

```

h1 = ((1,1), (7,9))
h2 = ((1,-1), (-7,-9))
h1+h2 = ((2,0), (0,0))
h1-h2 = ((0,2), (14,18))
h1*h2 = ((132,0), (0,0))
h1/h2 = ((-0.984848,0.0151515), (0.106061,0.136364))
(h1/h2)*h2 = ((1,1), (7,9))

```

Par la suite, la compilation sera facilitée par l'utilisation de la commande `make` à l'aide du `Makefile` suivant :

#### Makefile

```

CXX      = c++
CXXFLAGS = -O2 -std=c++11

```

Ceci permet d'entrer une fois pour toute la commande de compilation et ses options. L'option `-O2` active l'optimiseur de code tandis que l'option `-std=c++11` précise que le langage C++ est celui décrit par la révision du standard de 2011. La commande précédente devient :

```

make quaternion_tst
./quaternion_tst

```

### 1.1.4 Exercices

#### EXERCICE 1. (*Application aux rotations*)

L'objectif de ce problème est de présenter l'utilisation des quaternions pour représenter les rotations de  $\mathbb{R}^3$ . Fixons-nous un quaternion particulier  $p = a + ib + jc + kd$ , et introduisons l'application  $C_p$  des quaternions vers les quaternions, définie pour tout quaternion  $r$  par  $C_p(r) = pr\bar{p}$ .

- 1) Montrer que la composition est  $C_p \circ C_q = C_{pq}$  pour deux quaternions  $p$  et  $q$ .
- 2) L'application  $C_p$  s'identifie à une application de  $\mathbb{R}^4$  dans  $\mathbb{R}^4$ . Par un calcul un peu long, montrer que la matrice de cette application est :

$$M_p = \begin{pmatrix} a^2 + b^2 + c^2 + d^2 & 0 & 0 & 0 \\ 0 & a^2 + b^2 - c^2 - d^2 & 2bc - 2ad & 2bd + 2ac \\ 0 & 2bc + 2ad & a^2 - b^2 + c^2 - d^2 & 2cd - 2ab \\ 0 & 2bd - 2ac & 2cd - 2db & a^2 - b^2 - c^2 + d^2 \end{pmatrix}$$

- 2) Montrer que  $\det(M_p) = |p|^8$ .
- 3) On s'intéresse au cas où  $|p| = 1$  : la première ligne de  $M_p$  est  $(1, 0, 0, 0)$ . Montrer que la sous-matrice  $3 \times 3$ , notée  $\tilde{M}_p$ , obtenue en enlevant la première ligne et la première colonne est orthogonale et que son déterminant vaut 1.

4) On s'intéresse à représenter les rotations par des quaternions  $p$  de norme 1 via la matrice  $M_p$ . Soit  $R = \tilde{M}_p$  la rotation obtenue pour le quaternion  $p$ . Montrer que  $M_{-p} = R$  et que toute rotation a une représentation unique par un quaternion, au signe près.

5) Montrer que l'axe de la rotation est le vecteur de  $\mathbb{R}^3$  de coordonnées  $(b, c, d)$  et que l'angle de rotation  $\theta$  autour de cet axe vérifie :

$$\cos(\theta/2) = a \quad \text{et} \quad \sin^2(\theta/2) = b^2 + c^2 + d^2$$

### EXERCICE 2. (*Promotion de type*)

Nous avons vu que le langage C++ possède trois types à virgule flottante, par ordre de précision croissante : `float`, `double` et `long double`, qui représentent respectivement la précision simple, la précision double et une précision supérieure ou égale à la précision double. Ces types peuvent être combinées entre eux dans des expressions lorsqu'il n'y a pas perte de précision :

```
float    x1 = 1.0;
double   x2 = 2.0;
double   x3 = x1 + x2;
```

1) Considérer l'extrait de code suivant :

```
quaternion<float>  h1 (0.0, 1.0, 0.0, 0.0);
quaternion<double> h2 (1.0, 2.0, 0.0, 0.0);
quaternion<double> h3 = h1 + h2;
```

Expliquez pourquoi, avec la classe développée dans ce chapitre, ce code conduit à un échec à la compilation. Pour la même raison que la classe `quaternion`, la classe `complex` de la librairie `std` du C++ souffre d'un manque de souplesse en terme de promotion de type à virgule flottante : nous allons remédier à cela.

2) La *promotion de type* dans une expression à virgule flottante est définie par une relation de la forme :

$$\text{float} + \text{double} \longrightarrow \text{double}$$

Spécifiez complètement la promotion des types à virgule flottante dans le tableau suivant :

+	float	double	long double
float			
double			
long double			

3) La promotion de type entre deux types `T1` et `T2` sera définie par `typename promote<T1,T2>::type`. La classe `promote` est définie par :

```
struct undefined {};
```

```
template <typename T1, typename T2>
struct promote {
    typedef undefined type;
};
```

Par défaut, la promotion entre deux types quelconques est `undefined`. Les promotions effectives sont définies par des *spécialisations* de la classe `promote`, de la forme :

```
template<> struct promote<float, double> {
    typedef double type;
};
// ...
```

Notez que la spécialisation s'effectue à l'aide de la déclaration `template<>` et en précisant explicitement les types `T1` et `T2`. Complétez la classe `promote`, de façon à ce que les neuf combinaisons du tableau répondent correctement.

4) Définir l'opérations d'addition générale  $z_1 + z_2$  entre deux nombres complexes  $z_1$  et  $z_2$  utilisant des représentations flottantes a priori différentes, de façon à ce que le code de la question 1 fonctionne.

5) De même, définir l'opérations d'addition entre deux quaternions utilisant des type de virgule flottante différents.

6) Considérons une classe de nombre à virgule flottante définie indépendamment : pour fixer les idées, choisissons la classe `qd_real`, définie dans la librairie [29] et qui implémente quatre fois la double précision, soit huit fois la précision d'un `float`. Citer quelles seraient les modifications supplémentaires à apporter pour que les classes `complex` et `quaternion` puissent se combiner avec à la fois les précisions standards (`float`, `double`, `long double`) et cette classe `qd_real` ?

### 1.1.5 Notes

Les quaternions ont été introduits en 1853 par Hamilton. Représenter des rotations avec des quaternions de norme unité est une astuce ancienne en physique : Shoemaker [45] a été un pionnier pour son utilisation en géométrie algorithmique, pour tracer des courbes en rotation. Ces méthodes sont actuellement massivement utilisées, notamment dans la librairie `opengl`.

Il existe une autre application de la rotation  $\tilde{M}_p$  introduite à l'exercice 1 : cette application concerne la transformation de Lorentz en relativité. Pour cela on utilise une extension des quaternions, où le type `T` est un nombre complexe au lieu d'un réel : ces quaternions sont appelés quaternions complexes. Le but de ce chapitre étant de se familiariser au langage C++ et à la généricité par l'exemple, nous n'approfondissons pas les applications des quaternions au delà des exercices proposés.

La promotion de type présentée à l'exercice 2 est une technique de programmation typique du langage C++ : elle a été introduite par Veldhuizen [54] et est actuellement disponible dans diverses librairies C++ comme `blitz++` [53] ou `boost/promote` [9].

## 1.2 Analyse asymptotique des algorithmes

Le but de l'algorithmique en général, et de l'algorithmique numérique en particulier, est de concevoir les algorithmes les plus efficaces possibles pour résoudre les problèmes posés. Pour cela, il est nécessaire de mesurer l'efficacité d'un algorithme afin de pouvoir comparer les algorithmes entre eux. Ce chapitre introduit quelques notions de base relative à l'analyse des algorithmes.

### 1.2.1 Quelques concepts théoriques de l'algorithmique

Le choix d'un algorithme n'est pas crucial lorsque la taille des données à traiter reste petite : l'analyse de l'algorithme n'entre en jeu qu'à partir du moment où cette taille devient grande. Nous nous intéressons à la croissance du temps de calcul et de la place mémoire avec la taille des données, notée  $n$ , tend vers l'infini.

Les fonctions usuelles  $1, \log n, n, n \log n, n^2, n^3, \dots, 2^n$ , dont les ordres de grandeur forment une suite croissante, peuvent constituer une échelle de comparaison pour les algorithmes. Si un premier algorithme a un temps de calcul  $T_1(n) = c_1 n^2$  et un second algorithme un temps  $T_2(n) = c_2 n$ , ceci pour effectuer le même calcul, où  $c_1$  et  $c_2$  sont deux constantes positives strictement, alors pour  $n$  assez grand on aura  $T_1(n) \geq T_2(n)$ . Nous dirons alors que le second algorithme est *asymptotiquement* plus rapide que le premier. Ceci reste vrai même si  $c_1$  est beaucoup plus grand que  $c_2$ . Ceci reste encore vrai, par exemple, si le second algorithme est exécuté sur une machine lente, avec un langage de programmation peu performant et a été codé par un programmeur débutant, il sera quand même, pour des données assez grandes, plus rapide que le premier algorithme, même si il est exécuté sur un super-calculateur, écrit en langage assembleur très optimisé et codé par un programmeur expert.

L'analyse asymptotique des performances permet donc de s'affranchir de considération particulières et nous allons comparer les algorithmes sur la machine abstraite.

Les performances d'un algorithme peuvent se mesurer en pratique en terme de temps de calcul et de place mémoire nécessaires à l'exécution de l'algorithme : plutôt que de considérer un programme particulier qui code cet algorithme sur une machine donnée, par un programmeur expérimenté ou non, avec un langage de programmation performant ou non, imaginons plutôt une machine abstraite idéale et définissons-lui son unité de temps et de mémoire.

Le modèle de calculateur sous-jacent à toutes les analyses d'algorithmes présentés dans ce livre est connu sous le nom de *calculateur réel à accès aléatoire* (voir [7] par exemple). Pour simplifier, nous supposons que la machine est capable de travailler en précision infinie : autrement dit, nous ne considérons pas la question de la gestion des arrondis du point de vue de la place mémoire ou des temps de calculs. Ce modèle suppose que chaque unité de mémoire peut contenir la représentation d'un nombre réel et que l'accès en lecture ou en écriture à une unité de mémoire prend un temps constant, c'est-à-dire indépendant de l'unité de mémoire accédée. L'ensemble des opérations élémentaires, qui seront exécutées en une unité de temps par notre

machine abstraite comprend les quatre opérations algébriques  $+$ ,  $-$ ,  $*$ ,  $/$  ainsi que la comparaison entre deux nombres, le calcul de toutes les fonctions mathématiques usuelles comme  $\log$ ,  $\exp$ ,  $\sin$ ,  $\cos$ , et partie entière.

Les notations suivantes sont particulièrement utiles pour comparer les ordres de grandeur de différentes fonctions asymptotiques.

Soient  $f$  et  $g$  deux fonctions de la variable entière  $n$  et à valeur réelle.

- Nous dirons que  $f$  est un *grand o* de  $g$ , et nous noterons  $f(n) = \mathcal{O}(g(n))$ , si et seulement si il existe une constante  $C_1 > 0$  et un entier  $n_0$  tels que

$$f(n) \leq C_1 g(n), \quad \forall n \geq n_0$$

- Nous dirons que  $f$  est un *grand oméga* de  $g$ , et nous noterons  $f(n) = \Omega(g(n))$ , si et seulement si il existe une constante  $C_2 > 0$  et un entier  $n_0$  tels que

$$C_2 g(n) \leq f(n), \quad \forall n \geq n_0$$

- Nous dirons que  $f$  est un *grand thêta* de  $g$ , et nous noterons  $f(n) = \Theta(g(n))$ , si et seulement si il existe deux constantes  $C_1 > 0$  et  $C_2 > 0$  et un entier  $n_0$  tels que

$$C_2 g(n) \leq f(n) \leq C_1 g(n), \quad \forall n \geq n_0$$

### 1.2.2 Exemple : le calcul de $x^n$

L'idée la plus simple pour calculer  $x^n$  est de dérouler une boucle et d'effectuer  $n$  multiplications :

pow\_linear.cc

```
#include <iostream>
using namespace std;
template <class T>
T pow_linear (const T& x, size_t n) {
    T y = 1;
    for (size_t i = 1; i <= n; i++) y *= x;
    return y;
}
int main(int argc, char** argv) {
    double x = (argc > 1) ? atof(argv[1]) : 2;
    size_t n = (argc > 2) ? atoi(argv[2]) : 10;
    cout << pow_linear(x,n) << endl;
}
```

Pour exécuter ce programme, rien de plus simple :

```
make pow_linear
./pow_linear 2 8
./pow_linear 3.1 500
```

Le coût en temps de calcul est évidemment  $\Theta(n)$ .



Pouvons-nous faire mieux ? Si  $n$  est pair, nous avons la possibilité de décomposer le calcul en  $x^{n/2}x^{n/2}$  : au lieu de  $n$  multiplications, nous n'en effectuons à présent plus que  $n/2 + 1$  pour obtenir le même résultat. De même, si  $n/2$  est pair, nous pouvons re-diviser le produit de  $x^{n/2}$  en  $x^{n/4}x^{n/4}$ , si bien que, appliqué récursivement, nous effectuons beaucoup moins de multiplications. Si  $n$  est impair, on il est possible de diviser le produit en  $x^{(n-1)/2}x^{(n-1)/2}x$  : là aussi, appliqué récursivement, nous obtenons une réduction importante du nombre d'opérations.

pow\_recursive.cc

```
#include <iostream>
using namespace std;
template <class T>
T pow_recursive (const T& x, size_t n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    T y = pow_recursive(x, n/2);
    return (n % 2 == 0) ? y*y : y*y*x;
}
int main(int argc, char** argv) {
    double x = (argc > 1) ? atof(argv[1]) : 2;
    size_t n = (argc > 2) ? atoi(argv[2]) : 10;
    cout << pow_recursive(x, n) << endl;
}
```

Nous pouvons tester cette nouvelle version :

```
make pow_linear
./pow_recursive 2 8
./pow_recursive 3.1 500
```

Quel est le coût en temps de calcul  $T(n)$  de cet algorithme ? Il est donné par la formule de récurrence :

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Cette relation exprime que le nombre d'opérations nécessaire pour calculer  $x^n$  est celui pour calculer  $x^{n/2}$  plus un nombre d'opérations  $\Theta(1)$  indépendant de  $n$ . Ces dernières opérations incluent tout ce qui n'est pas l'appel récursif. Par récurrence, nous pouvons montrer sans difficulté que

$$T(n) = \Theta(\log n)$$

Nous avons par conséquent amélioré de façon très importante notre algorithme de  $x^n$ .

Cette approche de réduction du temps de calcul peut s'étendre à un très grand nombre de situations.

### 1.2.3 Résolution des récurrences

Plutôt que de résoudre, pour chaque algorithme, des récurrences similaires afin d'obtenir son temps de calcul, nous avons intérêt à le résoudre une fois pour toute.

Supposons que nous disposons d'un algorithme, dont le temps de calcul vérifie la relation de récurrence :

$$T(n) = aT(\lfloor n/b \rfloor) + f(n)$$

où  $a \geq 1$  et  $b > 1$  sont deux constantes, et  $f$  est une fonction réelle de l'entier  $n$ . La notation  $\lfloor n/b \rfloor$  représente la partie entière de  $n/b$ . Ce type de relation est très courant dans les algorithmes récursifs et nous en verrons un exemple à la section suivante, pour la multiplication de deux matrices. Cormen, Leiserson, Rivest et Stein [14, p. 86] proposent la résolution cette relation de récurrence en considérant différents cas possibles :

- 1) Supposons que  $f(n) = \mathcal{O}(n^{\log_b(a)+\lambda})$  pour une certaine constante  $\lambda > 0$ . Alors  $T(n) = \Theta(n^{\log_b(a)})$ .
- 2) Supposons que  $f(n) = \Theta(n^{\log_b(a)})$ . Alors  $T(n) = \Theta(n^{\log_b(a)})$ .
- 3) Supposons que  $f(n) = \Omega(n^{\log_b(a)+\lambda})$  pour une certaine constante  $\lambda > 0$ . Supposons également qu'il existe une constante  $0 < C < 1$  et un entier  $n_0$  tels que

$$af(n/b) \leq Cf(n), \quad \forall n \geq n_0$$

Alors  $T(n) = \Theta(f(n))$ .

### 1.2.4 Algorithme de Strassen pour la multiplication matricielle

Vous savez sans doute déjà comment multiplier deux matrices  $A = (a_{i,j})_{1 \leq i,j \leq n}$  et  $B = (b_{i,j})_{1 \leq i,j \leq n}$  : le produit  $C = AB$  est défini par ses coefficients  $(c_{i,j})_{1 \leq i,j \leq n}$  :

$$c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}$$

Il est clair que le calcul de chaque coefficient  $c_{i,j}$  nécessite  $n$  multiplications : le nombre total de multiplications pour calculer  $C = AB$  est donc  $\Theta(n^3)$ . Vous pouvez penser que cette façon de faire ne saurait être améliorée, c'est à dire effectuée asymptotiquement en moins d'itérations : vous avez tort. L'algorithme étudié ici, proposé par Strassen [48] a suscité un grand émoi lors de sa publication en 1969.

Supposons que  $n$  est pair et partitionnons :

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

où les sous-blocs sont de taille  $n/2$ . Lorsque  $n$  est impair, nous pouvons également partitionner en deux parties, par exemple  $(n-1)/2$  et  $(n+1)/2$ , aussi, pour simplifier l'exposé, et sans perte de généralité, nous allons supposer que  $n$  est une puissance de

2. Le produit  $C = AB$  se développe en :

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Ce calcul nécessite huit multiplications de matrices de taille  $n/2$ . L'idée, nullement évidente, de Strassen, est que ce calcul peut être effectué en sept multiplications au lieu de huit. Pour cela, calculons tout d'abord les sommes :

$$\begin{array}{l|l} s_1 = b_{12} - b_{22} & s_6 = b_{11} - b_{22} \\ s_2 = a_{11} + a_{12} & s_7 = a_{12} - a_{22} \\ s_3 = a_{21} + a_{22} & s_8 = b_{21} + b_{22} \\ s_4 = b_{21} - b_{11} & s_9 = a_{11} - a_{21} \\ s_5 = a_{11} + a_{22} & s_{10} = b_{11} + b_{12} \end{array}$$

Ensuite viennent les produits :

$$\begin{array}{l|l} p_1 = a_{11} s_1 & p_5 = s_5 s_6 \\ p_2 = s_2 b_{22} & p_6 = s_7 s_8 \\ p_3 = s_3 b_{11} & p_7 = s_9 s_{10} \\ p_4 = a_{22} s_4 & \end{array}$$

Nous pouvons vérifier, par un simple développement, que :

$$\begin{aligned} c_{11} &= p_3 + p_4 - p_2 + p_6 \\ c_{12} &= p_1 + p_2 \\ c_{21} &= p_3 + p_4 \\ c_{22} &= p_5 + p_1 - p_3 - p_7 \end{aligned}$$

Le temps de calcul  $T(n)$  de cet algorithme vérifie la relation de récurrence :

$$T(n) = 7 T(n/2) + c n^2$$

où  $c > 0$  est une constante, indépendante de  $n$ . Ce dernier terme inclut toutes les opérations non-récurrentes telles que les sommes et soustractions de deux matrices. Sans difficulté, nous pouvons montrer par récurrence sur  $T(n)$  que

$$T(n) = \Theta(n^{\log_2 7})$$

Autrement dit, il est possible de calculer le produit de deux matrices carré de taille  $n$  en  $\Theta(n^{\log_2 7})$  multiplications, avec  $\log_2 7 \approx 2.81$ , ce qui améliore l'algorithme en  $\Theta(n^3)$  multiplications précédent.

### 1.2.5 Application à l'inversion d'une matrice

Supposons qu'une matrice  $C$  soit l'inverse d'une autre matrice  $A$ . La méthode la plus naturelle pour calculer  $C$  à partir de  $A$  est de résoudre  $n$  systèmes linéaires :

$$Ax_i = b_i, \quad 1 \leq i \leq n$$

avec pour le second membre  $b_i$  un vecteur ne contenant que des zéros sauf son  $i$ -ième élément qui vaudra un. Alors le vecteur  $x_i$  représente la  $i$ -ième colonne de la matrice  $C$ . Pour résoudre les  $n$  systèmes linéaires, on a recours à une factorisation  $A = LU$  où  $L$  est une matrice triangulaire inférieure et  $U$  est triangulaire supérieure unitaire. Dans le cas où  $A$  est symétrique, les variantes sont la factorisation de Choleski  $A = LL^T$ , où  $L$  est triangulaire inférieure, ou bien la factorisation  $A = LDL^T$ , avec  $D$  diagonale et  $L$  triangulaire inférieure unitaire. (voir par exemple [32], page 266). Rappelons que cette dernière variante, très classique, s'écrit en pseudo-code :

```

algorithme : factorisation  $A = LDL^T$ 
  entrée :  $A$  matrice  $n \times n$  symétrique définie positive
  sortie :  $L$  triangulaire inférieure unitaire et  $D$  diagonale
début
  pour  $i = 1 : n$ 
     $L(i, i) = 1$ 
    pour  $j = 1 : i - 1$ 
       $L(i, j) = A(i, j)$ 
      pour  $k = 1 : j - 1$ 
         $L(i, j) = L(i, j) - L(i, k) * D(k, k) * L(j, k)$ 
       $L(i, j) = L(i, j) / D(j, j)$ 
     $D(i, i) = A(i, i)$ 
    pour  $j = 1 : i - 1$ 
       $D(i, i) = D(i, i) - L(i, j)^2 * D(j, j)$ 
  fin

```

Le pseudo-code précédent adopte les conventions de notations et de syntaxe de **octave** [18] et **matlab**, et nous avons introduit l'indentation pour marquer la limite des blocs d'instructions afin d'alléger les notations. Il y a trois boucles imbriquées, et le coût de cette factorisation est clairement  $\Theta(n^3)$ . Il est possible d'améliorer légèrement cet algorithme, en utilisant deux fois moins de multiplications et en effectuant l'algorithme *sur place*, c'est-à-dire en écrivant directement le résultat  $L$  et  $D$  dans le tableau servant à l'entrée  $A$  (voir par exemple [32], page 268). Le temps de calcul restera cependant  $\Theta(n^3)$ . La résolution de  $Ax = b$  équivaut à  $LL^Tx = b$  et s'effectue en deux étapes : tout d'abord  $Ly = b$  puis  $L^Tx = y$ . Chacune de ses deux étapes fait intervenir une matrice triangulaire. Ainsi, une fois  $L$  construite, par une descente, puis remontée, la résolution d'un système linéaire peut s'effectuer en  $\Theta(n^2)$  opérations. Ayant  $n$  systèmes linéaires à résoudre, nous obtenons par cette méthode un coût total de la construction de l'inverse  $C = A^{-1}$  en  $\Theta(n^3)$  opérations.

Revenons à présent à l'approche récursive adoptée pour le produit de deux matrices. En partitionnant  $A$  et  $C$ , avec les notations de la section précédente, nous avons, par

une généralisation de l'inverse d'une matrice  $2 \times 2$  :

$$\begin{aligned} c_{11} &= a_{11}^{-1} + (a_{11}^{-1} a_{12}) (a_{22} - a_{21} a_{11}^{-1} a_{12})^{-1} (a_{21} a_{11}^{-1}) \\ c_{12} &= - (a_{11}^{-1} a_{12}) (a_{22} - a_{21} a_{11}^{-1} a_{12})^{-1} \\ c_{21} &= - (a_{22} - a_{21} a_{11}^{-1} a_{12})^{-1} (a_{21} a_{11}^{-1}) \\ c_{22} &= (a_{22} - a_{21} a_{11}^{-1} a_{12})^{-1} \end{aligned}$$

Nous voyons apparaître un grand nombre de facteurs répétés, que nous pouvons ne calculer qu'une seule fois :

$$\begin{array}{l|l} r_1 = a_{11}^{-1} & c_{12} = r_3 r_6 \\ r_2 = a_{21} r_1 & c_{21} = r_6 r_2 \\ r_3 = r_1 a_{12} & r_7 = r_3 c_{21} \\ r_4 = a_{21} r_3 & c_{11} = r_1 - r_7 \\ r_5 = r_4 - a_{22} & c_{22} = -r_6 \\ r_6 = r_5^{-1} & \end{array}$$

Ainsi, pour calculer l'inverse d'une matrice de taille  $n$ , nous n'avons besoin que de deux inverses de matrices de taille  $n/2$  et de six produits de matrices de taille  $n/2$ . Le coût d'une opération d'inversion est donné par :

$$T(n) = 2T(n/2) + c n^{\log_2 7}$$

Par récurrence, nous montrons que :

$$T(n) = \Theta(n^{\log_2 7})$$

Autrement dit, le coût de l'inversion d'une matrice est dominé par le coût du produit entre deux matrices.

Le calcul du déterminant suit également cette réduction : De la formule :

$$\det(A) = \det(a_{22}) \det(a_{11} - a_{12} a_{22}^{-1} a_{21})$$

nous déduisons que le calcul du déterminant d'une matrice d'ordre  $n$  se réduit à deux calculs de déterminants d'ordre  $n/2$  plus une inversion et deux produits de matrices d'ordre  $n/2$ . Autrement dit, nous retrouvons un coût en  $\Theta(n^{\log_2 7})$ .

### 1.2.6 La classe `valarray<T>`

Afin de représenter des données de grandes tailles, nous allons avoir recours, tout au long de ce livre, à la classe `valarray<T>`. Cette classe, de type tableau, est disponible dans la librairie standard du C++ et représente un vecteur de données numériques. Cette classe est définie sous l'espace de nom `std` dans l'entête `<valarray>` :

## &lt;valarray&gt;

```

template <typename T>
class valarray {
public:
    valarray();
    valarray (size_t n);
    valarray (const T& initval, size_t n);
    valarray (const valarray<T>& y);
    valarray<T>& operator= (const valarray<T>& y);
    size_t size() const;
    const T& operator[] (size_t i) const;
    T& operator[] (size_t i);
    // ...
};

```

Cette classe est complétée par des opérations d'algèbre linéaire usuelles pour les vecteurs de  $\mathbb{R}^n$ .

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 <typename Vec1, typename Vec2>
typename Vec1::value_type dot (const Vec1& x, const Vec2& y) {
    typename Vec1::value_type sum = 0;
    for (size_t i = 0; i < x.size(); i++)
        sum += x[i]*y[i];
    return sum;
}
template <typename Vec>
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 [49, 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 de **octave** [18] et **matlab**, nous avons introduit, pour notre confort, la fonction **range**, très commode avec la classe **valarray**, et définie par :

## range.h

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

Lorsque le pas ne vaut pas un, nous avons :

## range.h (suite)

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

Nous avons établi la correspondance entre deux syntaxes :

matlab	C++
u (first:last-1)	u [range(first,last)]
u (first:step:last-1)	u [range(first,step,last)]

En clair, `u [range(first,last)]` considère l'intervalle `[first,last[` fermé à gauche et ouvert à droite, si bien que le nombre d'éléments est `last-first`.

### 1.2.7 Exercices

EXERCICE 3. (*Factorisation  $LDL^T$* )

La matrice

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$$

possède-t'elle une factorisation  $LDL^T$  ?

EXERCICE 4. (*Factorisation  $LDL^T$  d'une matrice bande*)

1) Soit  $b$  un entier dans l'intervalle  $[0, n-2]$ . Montrer que si une matrice symétrique  $A$  est  $(2b+1)$ -diagonale, c'est-à-dire que si  $A_{i,j} = 0$  pour  $|i-j| \geq b+1$ , alors la factorisation  $A = LDL^T$  est telle que  $L$  est  $(b+1)$ -diagonale, c'est-à-dire  $L_{i,j} = 0$  pour  $i-j \geq b+1$ . L'entier  $b$  est appelé largeur de bande de la matrice  $A$  et on dit que la factorisation de Choleski respecte la structure bande de  $A$ .

2) Écrire une variante de l'algorithme de factorisation  $LDL^T$  qui exploite le fait que les éléments de  $A$  hors de la bande sont nuls. Montrer que le temps de calcul de cet algorithme est

$$T(n) = \Theta(b^2 n)$$

et que le temps de résolution des systèmes triangulaires, une fois la factorisation formée, est  $\Theta(bn)$ .

EXERCICE 5. (*Algorithme de Strassen*)

Utilisez l'algorithme de Strassen pour le produit matriciel

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

EXERCICE 6. (*Multiplication de polynômes*)

Montrez comment multiplier deux polynômes réels  $p_1(x) = ax + b$  et  $p_2(x) = cx + d$  en trois multiplications entre réels au lieu de quatre. Indication : une des multiplications est  $(a + b)(c + d)$ .

EXERCICE 7. (*Multiplication de nombres complexes*)

Montrez comment multiplier deux nombres complexes  $z_1 = a + ib$  et  $z_2 = c + id$  en trois multiplications entre réels au lieu de quatre.

EXERCICE 8. (*Matrices denses*)

Nous nous proposons d'étudier la programmation la plus simple possible de matrices *denses* rectangulaires. Ces matrices ne font pas partie de la librairie standard actuelle. Cependant, différents projets récents de standardisation (`lapack++`, `boost/ublas`) proposent un interface du type :

dmatrix.h

```
#include <valarray>
template <class T>
class dmatrix {
public:
    dmatrix (size_t nr = 0, size_t nc = 0);
    void resize (size_t nr, size_t nc);
    size_t nrow () const;
    size_t ncol () const;
    T& operator() (size_t i, size_t j);
    T operator() (size_t i, size_t j) const;
protected:
    size_t nr, nc;
    std::valarray<T> v;
};
```

Le tableau de type `valarray` dans la zone de données contient les coefficients de la matrice, rangés colonne par colonne.

- 1) Écrire le code du constructeur ainsi que des quatre fonctions d'accès.
- 2) Écrire le code du produit matrice-vecteur suivant :

```
valarray<T> operator* (const dmatrix<T>& a, const valarray<T>& x);
```

Quel est le coût en nombre de multiplication de cette opération ?

- 3) Écrire le code du produit matrice-matrice suivant :

```
dmatrix<T> operator* (const dmatrix<T>& a, const dmatrix<T>& b);
```

L'algorithme sera le plus simple possible, sur la base de trois boucles imbriquées. Montrez que le coût en nombre de multiplication de cette opération  $\Theta(n^3)$  pour une matrice carrée  $n \times n$ .



### 1.2.8 Notes

Les ouvrages de calcul scientifique actuels n'abordent pas, ou que très succinctement, l'analyse asymptotique des algorithmes numériques : il est alors généralement difficile de comparer différentes méthodes pour résoudre un même problème. Inversement, les ouvrages d'algorithmique actuels à vocation pédagogique et qui abordent la l'analyse asymptotique des algorithmes ne traitent pas de calcul numérique, c'est-à-dire avec des nombres à virgule flottante pour approcher les nombres réels : les algorithmes qui y sont présentés concernent alors plutôt les tris de tableaux ou de listes, ou encore des structures de données de type arbres pour représenter des dictionnaires.

La présentation des notions de l'analyse asymptotique des algorithmes de ce chapitre est volontairement réduite au strict minimum. Le lecteur qui souhaiterait plus de détails est invité à se reporter à des ouvrages classiques. Une présentation récente et assez complète de l'algorithmique et de l'analyse asymptotique est l'ouvrage de Cormen, Leiserson, Rivest et Stein [14] : le lecteur y trouvera également au chapitre 30 quelques algorithmes numériques comme la transformée de Fourier rapide et le calcul avec des polynômes. Knuth [30], plus ancien, reste une référence classique. Dans un second volume [31], l'auteur introduit l'analyse des calculs en virgule flottante ainsi que le calcul avec des polynômes. Enfin, Boissonat et Yvinec [7] proposent une analyse asymptotique de nombreux algorithmes pour la géométrie et les maillages.

L'algorithme de Strassen pour la multiplication matricielle a proposé en 1969 [48] : il a suscité un grand émoi lors de sa publication. Jusque là, peu de gens pensaient qu'il pût exister un algorithme asymptotiquement plus rapide que la procédure en trois boucles imbriquées et avec un temps de calcul en  $\Theta(n^3)$ . Strassen a proposé un algorithme en  $\Theta(n^{\log_2(7)})$ , avec  $\log_2(7) \approx 2.81 < 3$  et, de plus, a montré que l'inversion d'une matrice n'était également pas une opération en  $\Theta(n^3)$  mais est dominé par le coût asymptotique de la multiplication de deux matrices. L'algorithme original de Strassen a été depuis amélioré à plusieurs reprises depuis : l'algorithme le plus performant actuellement est celui proposé par Coppersmith et Winograd, en  $\Theta(n^\omega)$  avec  $\omega \approx 2.376$ , et la meilleure borne inférieure connue pour le produit de deux matrices n'est autre que  $\Theta(n^2)$ , car il faut bien remplir les  $n^2$  éléments du résultat. Le lecteur intéressé pourra lire l'article [39] qui fait une synthèse récente des différentes avancées sur le produit de deux matrices, ainsi que des commentaires dans [14] au chapitre 4 ou encore dans [36], paragraphe 2.11 pour l'inversion des matrices.

Dans la pratique, la constante cachée dans la notation  $\Theta(\cdot)$  reste relativement grande, même pour des implémentations très optimisées, et la procédure classique en  $\Theta(n^3)$  est plus rapide pour des  $n$  suffisamment petits, disons  $n < n_0$ . Les implémentations de la méthode essaient de déterminer ce  $n_0$ , appelé point d'équilibre, et qui peut être finalement relativement grand, suivant le type de machine considéré. Les implémentations actuelles dans la librairie d'algèbre linéaire `atlas`, ce  $n_0$  peut être évalué dynamiquement [15], par un étalonnage sur une machine spécifique lorsque la librairie est installée. Ensuite, à chaque appel et pour un  $n$  donné, la procédure la plus rapide est alors sélectionnée.

L'exercice 4, page 22, a présenté des optimisations possibles pour des matrices

présentant une structure bande : ce type d'optimisation est disponible dans la librairie **lapack** [2], écrite en langage fortran, et qui dispose d'interfaces pour d'autres langages tels que le C ou le C++.

Au chapitres suivants, et tout au long de cet ouvrage, aborderons des méthode radicalement différentes à celle présentée ici : ces méthodes nous permettrons de diminuer encore de façon très importante le coût asymptotique de la résolution d'un système linéaire : elles nous permettrons de passer en dessous de la barre  $\Theta(n^2)$  pour la résolution d'un système matriciel. Ces méthodes s'appliqueront non plus à une matrice générale mais à des classe de matrices particulières. Cette perte de généralité est compensée par une motivation majeure : les classes de matrices considérées tout au long de cet ouvrage apparaissant très fréquemment dans les applications du calcul scientifique, et nous en étudierons plusieurs exemples concrets.



## Chapitre 2

# Transformée de Fourier et applications

### 2.1 Transformée de Fourier

Les transformées de Fourier servent intensivement dans le traitement du signal. Le signal est défini dans un domaine de temps, à savoir une fonction qui, à un temps associe une amplitude. L'analyse de Fourier permet de caractériser le signal en termes de fréquences, sous la forme de modes et de poids associés. Parmi les nombreuses applications récentes de la transformée de Fourier, citons les techniques de compression audio et vidéo, dont les fichiers `mp3` et `jpeg`. De nouvelles applications font leur apparition, comme le produit de deux polynômes ou celui de deux nombres entiers ou à virgule flottante en précision élevée, et dont les notes en fin de chapitre donnent un aperçu. Dans cet esprit, le chapitre 2.3 présentera une autre application à la résolution très rapide de solutions approchées d'équations aux dérivées partielles en dimension deux d'espace.

#### 2.1.1 La transformée de Fourier discrète

Soit  $f$  une fonction de  $\mathbb{R}$  dans  $\mathbb{C}$ . La fonction  $f$  est supposée périodique de période  $T > 0$  et est appelée *signal*. Ce signal se développe en série de Fourier suivant :

$$f(x) = \sum_{p \in \mathbb{Z}} c_p e^{2i\pi p x/T} \quad \text{avec} \quad c_p = \frac{1}{T} \int_0^T f(x) e^{-2i\pi p x/T} dx \quad (2.1)$$

Supposons qu'on ne connaisse qu'un nombre fini  $n$  de valeurs de  $f$  sur  $[0, T[$  en des points régulièrement espacés :  $(f(kT/n))_{0 \leq k \leq n-1}$  et qu'on appelle *échantillon*. Nous souhaitons déduire de cette information une approximation des coefficients de Fourier

$(c_p)_{p \in \mathbb{Z}}$  de  $f$  :

$$f\left(\frac{kT}{n}\right) = \sum_{p \in \mathbb{Z}} c_p e^{2i\pi pk/n} = \sum_{p \in \mathbb{Z}} c_p w_n^{pk}, \quad 0 \leq k \leq n-1,$$

où nous avons noté  $w_n = e^{2i\pi/n}$ , appelé racine  $n$ -ième de l'unité, car il vérifie  $w_n^n = 1$ . Ayant  $n$  données, il est logique de ne chercher à calculer que  $n$  des coefficients  $c_p$ ,  $p \in \mathbb{Z}$ . De plus, ces coefficients tendent vers zéro quand  $n$  devient grand, aussi nous chercherons à les approcher pour  $p \in \{-n/2, \dots, n/2-1\}$  en supposant  $n$  pair. Dans le cas où  $n$  est impair, nous pourrions prendre un intervalle centré. Notons  $\left(c_p^{(n)}\right)_{-n/2 \leq p \leq n/2-1}$  les coefficients de la série tronquée, et caractérisés par :

$$f\left(\frac{kT}{n}\right) = \sum_{p=-n/2}^{n/2-1} c_p^{(n)} w_n^{pk}, \quad 0 \leq k \leq n-1.$$

Il s'agit d'un système de  $n$  équations à  $n$  inconnues, et nous allons voir que ce système admet une solution unique. Pour une raison de commodité, réarrangeons ce système en numérotant les inconnues de 0 à  $n-1$  de la façon suivante :

$$F_p = \begin{cases} c_p^{(n)} & \text{si } 0 \leq p \leq n/2-1 \\ c_{p-n}^{(n)} & \text{si } n/2 \leq p \leq n-1 \end{cases}$$

et posons  $f_k = f(kT/n)$ . Le système devient :

$$\sum_{p=0}^{n-1} w_n^{pk} F_p = f_k, \quad 0 \leq k \leq n-1.$$

Notons  $A$  la matrice du système. Elle est symétrique :

$$A = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & \dots & w_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)^2} \end{pmatrix}$$

Par un algorithme direct classique de type Gauss ou Choleski, la résolution du système linéaire serait en  $\Theta(n^3)$  opérations. L'intérêt de ce système vient du fait que l'inverse de  $A$  admet une forme explicite connue :

$$A^{-1} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n^{-1} & w_n^{-2} & \dots & w_n^{-n+1} \\ 1 & w_n^{-2} & w_n^{-4} & \dots & w_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & w_n^{-n+1} & w_n^{-2(n-1)} & \dots & w_n^{-(n-1)^2} \end{pmatrix}$$

Par conséquent, le système se résout explicitement :

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} w_N^{-nk} f_n, \quad 0 \leq k \leq N-1. \quad (2.2)$$

Par un produit matrice-vecteur avec  $A^{-1}$ , le calcul de  $(F_k)_{0 \leq k \leq n-1}$  reste cependant en  $\Theta(n^2)$  opérations. Bien que ce soit déjà bien plus compétitif que l'algorithme de Strassen vu précédemment, nous allons voir qu'il est possible de faire ici encore beaucoup mieux en exploitant la forme particulière de la matrice  $A$  et en faisant des hypothèses sur la parité de  $n$ .

### 2.1.2 Algorithme par bisection

Supposons que  $n$  est pair, et dans (2.2), séparons les termes d'indices pairs et impairs :

$$F_k = \frac{1}{2} (P_k + w^{-k} I_k), \quad 0 \leq k \leq n-1,$$

avec

$$\begin{aligned} P_k &= \frac{2}{n} (F_0 + w_n^{-2k} F_2 + \dots + w_n^{-(n-2)k} F_{n-2}) \\ I_k &= \frac{2}{n} (F_1 + w_n^{-2k} F_3 + \dots + w_n^{-(n-2)k} F_{n-1}) \end{aligned}$$

Supposons ensuite  $n$  divisible par 4. Ces sommes peuvent alors toutes deux à leur tour être séparées en termes pairs et impairs. Supposant enfin  $n = 2^r$ , nous pouvons poursuivre la dichotomie jusqu'à ne plus avoir qu'un seul terme dans la somme. Nous obtenons alors un algorithme récursif appelé *transformée de Fourier rapide*. Il présente l'intérêt de se programmer en suivant la démarche que nous avons adopté.

fft.h

```
#include <valarray>
template <typename T>
void fft (const T& w,
         const std::valarray<T>& f, std::valarray<T>& F)
{
    size_t n = f.size();
    if (n == 0) return;
    if (n == 1) { F[0] = f[0]; return; }
    size_t m = n/2;
    std::valarray<T> f0(m), f1(m), F0(m), F1(m);
    for (size_t k = 0; k < m; k++) {
        f0[k] = f[2*k];
        f1[k] = f[2*k+1];
    }
    fft (w*w, f0, F0);
    fft (w*w, f1, F1);
    T wk = 1;
    for (size_t k = 0; k < m; k++) {
        F[k] = 0.5*(F0[k] + wk*F1[k]);
        F[m+k] = 0.5*(F0[k] - wk*F1[k]);
        wk = w*wk;
    }
}
```

Un programme d'appel de cette fonction est :

fft.cc

```
#include "fft.h"
#include "valarray_io.h"
#include <complex>
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
int main() {
    valarray<complex<double>> x;
    cin >> x;
    size_t n = x.size();
    assert(fabs(log2(n)-int(log2(n))) < 1e-10);
    const double pi = acos(-1.0);
    const complex<double> i (0,1);
    complex<double> w = exp((2*pi/n)*i);
    valarray<complex<double>> y(n);
    fft (1./w, x, y);
    cout << setprecision(16) << y;
}
```

Ce programme lit simplement les données sur le standard d'entrée, appelle la transformée de Fourier rapide, puis écrit le résultat sur le standard de sortie. Pour illustrer la transformée de Fourier rapide, calculons les amplitudes des harmoniques après un pincement de corde de guitare : nous pinçons la corde en son milieu pour avoir un beau son, bien propre.

guitar.cc

```
#include <complex>
#include <iostream>
using namespace std;
complex<double> f (double x) { return (x < 0.5) ? x : 1-x; }
int main(int argc, char** argv) {
    size_t n = (argc > 1) ? atoi(argv[1]) : 16;
    cout << n << endl;
    for (size_t k = 0; k < n; k++)
        cout << f(1.*k/n) << endl;
}
```

La première valeur  $F_0^{(n)}$  est l'amplitude de la déformation de la corde, qui vaut  $1/4$  ici, indépendamment de  $n$  :

```
make fft guitar
./guitar 16 | ./fft
./guitar 32 | ./fft
```

Les autres valeurs de  $F_k^{(n)}$ ,  $k \geq 1$  correspondent aux amplitudes des harmoniques successives de la corde de guitare. La quantité  $F_1^{(n)}$  correspond à l'approximation de  $F_1$ , qui est l'amplitude de la note fondamentale de la corde, et est réelle. Le calcul de cette amplitude est d'autant plus précis que  $n$  est grand : La pente sur la Fig. 2.2, en axes log-log, suggère que l'erreur commise sur l'amplitude de la première harmonique

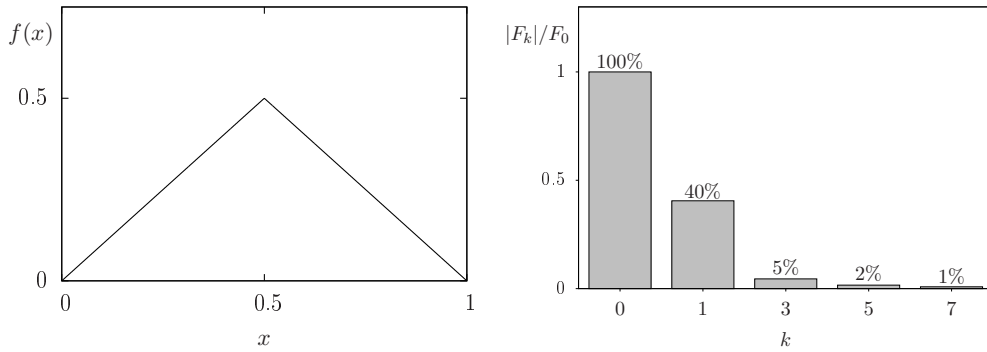


FIGURE 2.1 – Déformation de la corde de guitare et décroissance correspondante des amplitudes des harmoniques.

en approchant la transformée de Fourier par un développement tronqué à l'ordre  $n$ , notée  $\left|F_1^{(n)}\right|$  décroît comme  $\mathcal{O}(n^{-2})$ .

### 2.1.3 Coût en temps de calcul

Pour déterminer le temps de calcul de la fonction `fft`, remarquons tout d'abord, que si on omet les appels récursifs, le coût est  $\Theta(n)$ . Comme il y a deux appels récursifs, la récurrence pour le temps de calcul est donc :

$$T(n) = 2T(n/2) + \theta(n)$$

Un rapide calcul par induction (voir par exemple l'exercice 1.2.3) conduit à :

$$T(n) = \Theta(n \log n)$$

Ainsi, en exploitant la forme de la matrice dense  $A$ , nous avons réduit le temps de résolution de  $\Theta(n^3)$  à  $\Theta(n \log n)$ .

### 2.1.4 Exercices

EXERCICE 9. (*Où pincer une corde de guitare ?*)

Le but de cet exercice est d'approfondir ses connaissances sur la transformée de Fourier à travers l'exemple de la corde de guitare. Le son est d'autant plus propre que la note fondamentale se dégage nettement des harmoniques suivantes, dont l'amplitude décroît le plus vite possible.

1) Pour une corde pincée en son milieu, la déformation est donnée par :

$$f(x) = \begin{cases} x & \text{si } 0 \leq x \leq 1/2 \\ 1 - x & \text{si } 1/2 \leq x \leq 1 \end{cases}$$



$n$	$F_1^{(n)}$
4	0.125
8	0.1066941
16	0.1026333
32	0.1016473
64	0.1014026
128	0.1013415
256	0.1013262
512	0.1013224
1024	0.1013215

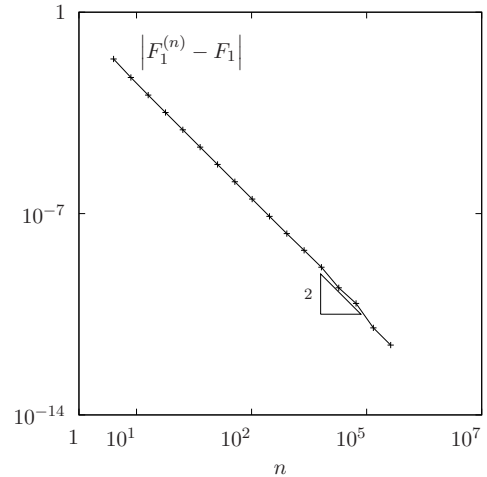


FIGURE 2.2 – Convergence de la première harmonique  $F_1$  en fonction de  $n$ .

Les coefficients  $c_p$  sont données de façon exacte par (2.1). Calculer  $c_0 = F_0$ , qui correspond à la moyenne de  $f$ .

2) Calculer l'amplitude de la note fondamentale de la corde  $c_1 = F_1$ .

3) Montrer que  $c_2 = 0$  et calculer l'amplitude de la première harmonique de la corde  $c_3$ .

4) Montrer que  $c_4 = 0$  et calculer l'amplitude de la seconde harmonique de la corde  $c_5$ .

5) Montrer que  $c_p = 0$  si  $p$  est pair et calculer  $c_p$  quand  $p$  est impair. Comment les amplitudes  $|c_p|$  des harmoniques successives décroissent-elles ?

6) Nous allons à présent pincer la corde en un point  $x_0 \in ]0, 1[$ , qui pourra être autre que le milieu de la corde, correspondant à  $x_0 = 1/2$ . La déformation est donnée par :

$$f(x) = \begin{cases} 2x(1 - x_0) & \text{si } 0 \leq x \leq x_0 \\ 2x_0(1 - x) & \text{si } x_0 \leq x \leq 1 \end{cases}$$

Vérifier que la déformation moyenne  $c_0$  de la corde est inchangée. Calculer  $c_1$  et plus généralement les coefficients  $c_p$ . Comment les amplitudes  $|c_p|$  des harmoniques successives décroissent-elles à présent ? Est-il préférable de pincer la corde au milieu ou bien de façon décentrer pour entendre la note fondamentale la plus pure possible ?

### 2.1.5 Notes

C'est en 1965 que Cooley et Tukey publient cette méthode, qui deviendra bientôt l'algorithme le plus célèbre du calcul scientifique. Cependant, il a été découvert par la

suite que l'algorithme avait déjà été inventé en 1805 par Gauss et adapté à plusieurs reprises par la suite [28] sous des formes différentes.

L'ouvrage de van Loan est la référence classique pour la présentation de la transformée de Fourier rapide, et cet auteur n'hésite pas à dire que cet algorithme a changé la face des sciences à un tel point qu'il n'est pas exagéré de dire que le monde tel que nous le connaissons serait bien différent sans `fft`.

L'implémentation présentée ici de la transformée de Fourier rapide est la plus simple qui soit et avec un coût en  $\Theta(n \log n)$ . Les applications massives de cet algorithme dans de nombreux domaines ont conduit à diverses optimisations qui tendent à diminuer autant que possible la constante cachée dans la notation  $\Theta(\cdot)$  du coût de calcul. Une implémentation plus efficace élimine la récursivité pour obtenir une version itérative directe : le lecteur intéressé pourra consulter [14] à la section 30.3. Il est également possible d'étendre la transformation du cas où  $n$  est une puissance de 2 au cas où  $n$  est une puissance de 3, d'une puissance ou d'un produit de puissance quelconque, et finalement de pouvoir choisir  $n$  quelconque, jours avec un coût  $\Theta(n \log n)$ .

La librairie `fftpack` [50], en langage fortran, écrite dans les années 1980, propose ces fonctionnalités. Plus récente, la librairie `fftw` [21], écrite en langage C propose des optimisations de performance plus poussées, adaptées aux architectures des ordinateurs actuels.

La transformée de Fourier rapide suppose que l'entrée représente des points uniformément espacés. Cependant, il existe des techniques pour approcher la transformation à partir d'un échantillonnage non-uniforme : l'article de Ware [55] présente un comparatif des différents algorithmes possibles.

## 2.2 Discrétisation de problèmes aux limites

### 2.2.1 Présentation du problème

Considérons le très classique problème de Poisson avec conditions aux bord homogènes de Dirichlet :

$$\begin{aligned} &\text{trouver } u, \text{ définie de } [-1, 1] \text{ dans } \mathbb{R}, \text{ telle que} \\ &\quad \begin{aligned} -u'' &= f \text{ dans } ]-1, 1[ \\ u(-1) &= u(1) = 0 \end{aligned} \end{aligned} \tag{2.3}$$

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 ou la déformation d'une membrane ou d'une voile sous l'effet du vent.

La solution du problème (2.3) 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 :

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

$$\begin{aligned} \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} &= f(x_i) \text{ pour } 1 \leq i \leq n-1 \\ u_0 &= u_n = 0 \end{aligned} \quad (2.4)$$

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 :

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} h^2 f(x_1) \\ \vdots \\ h^2 f(x_{n-1}) \end{pmatrix} \quad (2.5)$$

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  $b = (b_i)_{1 \leq i \leq n-1}$ ,  $b_i = h^2 f(x_i)$ . La matrice symétrique  $A$  a une forme très particulière : elle n'a d'éléments non-nuls que sur la diagonales et sur les deux diagonales juste au dessus et au dessous. Elle est dite tridiagonale, et nous allons voir comment résoudre très efficacement le système linéaire associé.

### 2.2.2 La résolution du système tridiagonal

Nous allons voir que 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 second 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 [32], page 268).

Reprenons l'algorithme général de factorisation sous la forme  $A = LDL^T$ , et présenté page 19, et adaptions-le ici au cas d'une matrice tridiagonale, en prenant soin de n'effectuer aucune multiplication par zéro. La fonction suivante effectue la factorisation *sur place*, au sens où les deux tableaux  $D$  et  $L$  contiennent les coefficients non-nuls de  $A$  en entrée, et fournissent en sortie les coefficients non-nuls de la factorisée.

inplace\_factorize.h

```
#include <valarray>
template <typename T>
void inplace_factorize(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 enfin la résolution de  $LDL^T x = b$  :

#### inplace\_solve.h

```
template <typename T>
void inplace_solve (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 (long i = D.size()-1; i > 0; i--)
        b[i-1] -= L[i-1]*b[i];
}
```

La résolution s'effectue également sur place : le tableau **b** contient le second membre en entrée et fournit la solution en sortie.

### 2.2.3 Le programme de tests

Choisissons pour second membre  $f(x) = 1$  : la solution est connue explicitement :  $u(x) = x(1-x)/2$ . Le code suivant va permettre de tester notre résolution du problème de Poisson en dimension un :

#### fish1d.cc

```
#include <iostream>
#include "inplace_factorize.h"
#include "inplace_solve.h"
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;
    double h = 2./n;
    valarray<double> D( 2, n-1), L(-1, n-2);
    inplace_factorize (D, L);
    valarray<double> b(h*h, n-1);
    inplace_solve (D, L, 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);
    cout << "err = " << uerr.max() << endl;
    return (uerr.max() < 1e-8) ? 0 : 1;
}
```

La compilation et l'exécution sont donnés par :

```
make fish1d
./fish1d
```

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

### 2.2.4 Le cas d'une subdivision non-uniforme

Dans la section précédente, la subdivision était uniforme : le pas de discrétisation  $h$  était constant. Nous allons voir comment le faire varier et comment cela peut-être très intéressant quand la solution exacte  $u$  est peu régulière. Pour cela, nous allons introduire la méthode des éléments finis et étudier comment elle généralise la méthode des différences finis du paragraphe précédent.

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

$$\begin{aligned} m(u, v) &= \int_{-1}^1 u(x) v(x) dx, \quad \forall u, v \in L^2(I), \\ a(u, v) &= \int_{-1}^1 u'(x) v'(x) dx. \quad \forall u, v \in H_0^1(I). \end{aligned}$$

Les formes  $a(.,.)$  et  $m(.,.)$  sont habituellement appelées formes d'énergie et de masse, respectivement. La formulation variationnelle de (2.3) est :

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

$$a(u, v) = m(f, v), \quad \forall v \in H_0^1(I). \quad (2.6)$$

Pour obtenir cette formulation variationnelle, nous avons multiplié par une fonction-test  $v$  qui s'annule aux bords  $x = \pm 1$ , puis intégré entre  $-1$  et  $1$ , et enfin intégré par parties.

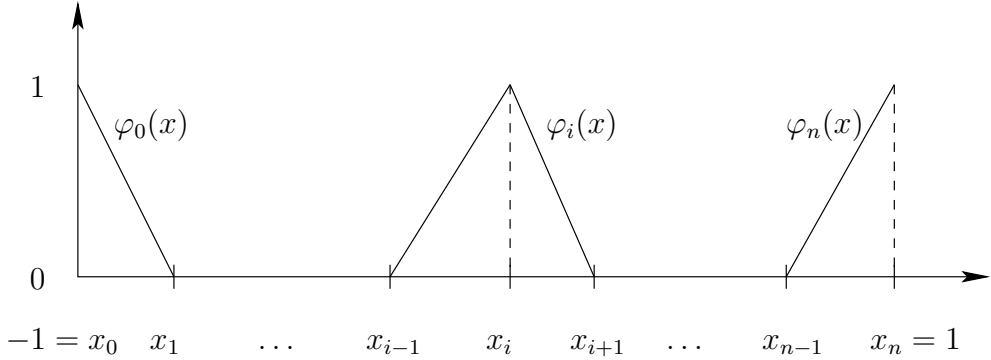


FIGURE 2.3 – Fonction de base  $\varphi_i$ .

Passons à présent à la discrétisation. 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  $V_h = X_h \cap H_0^1(I)$  l'espace des fonctions de  $X_h$  et s'annulant en  $x = \pm 1$ . Nous considérons le problème approché suivant :

trouver  $u_h \in V_h$  telle que

$$a(u_h, v_h) = m(f, v_h), \quad \forall v_h \in V_h. \quad (2.7)$$

L'espace  $X_h$  est de dimension finie :  $\dim(X_h) = n + 1$  et l'espace  $V_h$  est un sous-espace de  $X_h$  qui contraint ses éléments à s'annuler aux deux extrémités  $x = \pm 1$  : nous avons  $\dim(V_h) = n - 1$ . 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. 2.3). L'opérateur d'interpolation de Lagrange, noté  $\pi_h$ , associe à toute fonction  $u$  continue son interpolée  $\pi_h(u) \in X_h$  telle que  $u$  et  $\pi_h(u)$  coïncident aux  $n + 1$  points  $x_i$ ,  $0 \leq i \leq n$ . La fonction `interpolate` suivante implémente l'opérateur  $\pi_h$  :

`interpolate_1d.h`

```
#include <valarray>
template <typename T, typename Function>
std::valarray<T>
interpolate (const std::valarray<T>& x, Function v) {
    std::valarray<T> vh (x.size());
    for (size_t 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 (2.7), la formulation variationnelle conduit à :

$$\begin{aligned} \sum_{j=0}^n a(\varphi_j, \varphi_i) u_j &= m(f, \varphi_i), \quad 1 \leq i \leq n-1, \\ u_0 &= u_n = 0. \end{aligned}$$

Cependant, il est difficile d'évaluer le second membre  $m(f, \varphi_i)$  de façon exacte pour une fonction  $f$  générale :

$$m(f, \varphi_i) = \int_{-1}^1 f(x) \varphi_i(x) dx, \quad 1 \leq i \leq n-1$$

Cette intégrale ne peut-être connue exactement que pour des fonctions  $f$  particulières. Aussi, nous décomposons cette intégrale sur  $[-1, 1]$  en une somme d'intégrale sur chaque élément  $[x_i, x_{i+1}]$  et nous approchons chaque intégrale par la formule de quadrature des trapèzes :

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

En introduisant la forme bilinéaire, définie pour tout  $u, v \in L^2(I)$  par :

$$m_h(u, v) = \sum_{i=1}^n h_i \frac{u(x_{i-1})v(x_{i-1}) + u(x_i)v(x_i)}{2}$$

ceci revient à remplacer le problème approché précédent par le problème suivant :

trouver  $\tilde{u}_h \in V_h$  telle que

$$a(\tilde{u}_h, v_h) = m_h(f, v_h), \quad \forall v_h \in V_h. \quad (2.8)$$

Cette fois, nous n'avons plus aucun problème pour évaluer le second membre. Afin de ne pas alourdir les notations, nous omettrons par la suite les tildes, et noterons  $u_h$  la solution de (2.8). En choisissant  $v_h = \varphi_i$  dans (2.8), la formulation variationnelle (2.8) conduit à :

$$\begin{aligned} \sum_{j=0}^n a(\varphi_i, \varphi_j) u_j &= \sum_{j=0}^n m_h(\varphi_i, \varphi_j) f(x_j), \quad 1 \leq i \leq n-1, \\ u_0 &= u_n = 0. \end{aligned}$$

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 :

mass.h

```
template <typename T>
void mass (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 :

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

ainsi que

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

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  $A$  :

energy.h

```
template <typename T>
void energy (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, comme précédemment, la diagonale, de 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 que son dernier élément  $L[L.size()-1]$  ne sera pas utilisé.

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

$$-\frac{u_{i-1}}{h_{i-1}} + \left( \frac{1}{h_i} + \frac{1}{h_{i-1}} \right) u_i - \frac{u_{i+1}}{h_i} = \frac{h_i + h_{i-1}}{2} f(x_i), \quad 1 \leq i \leq n-1$$

$$u_0 = u_n = 0$$

En particulier, lorsque la subdivision est choisie uniforme de pas  $h$ , nous retrouvons le schéma aux différences finies (2.4), page 34. 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. Pour augmenter en lisibilité et faciliter la réutilisation de notre code, nous allons encapsuler les deux tableaux ( $D, L$ ) dans une classe représentant une matrice tridiagonale. En effet, nous avons pour ambition de réutiliser notre code pour résoudre ensuite des problèmes en dimension deux.



### 2.2.5 La classe des matrices tridiagonales symétriques

Voici une classe qui représente les matrices symétriques et tridiagonales :

tridiag.h

```
#include <valarray>
template <typename T>
class tridiag {
public:
    tridiag (size_t n=0, const T& a=0, const T& b=0);
    tridiag (const tridiag<T>&);
    tridiag<T>& operator= (const tridiag<T>&);
    std::valarray<T> operator* (const std::valarray<T>& x) const;
    template <class U>
    friend void energy (std::valarray<U>& x, tridiag<U>& A);
protected:
    std::valarray<T> D, L;
};
```

La classe contient en interne les deux tableaux D et L précédent et propose un interface permettant le produit matrice-vecteur  $A * x$  ainsi qu'une fonction *amie*, qui assemblera la matrice d'énergie, et aura ainsi la permission d'accéder aux tableaux D et L internes à la classe.

L'assemblage de la matrice d'énergie se réduit à :

tridiag.h (suite)

```
#include "energy.h"
template <class T>
void energy (std::valarray<T>& x, tridiag<T>& A) {
    energy (x, A.D, A.L);
}
```

Pour coder le produit matrice-vecteur, utilisons la fonctions `shift` de la classe `valarray<T>`, qui réalise un décalage logique (voir section 1.2.6, page 1.2.6).

tridiag.h (suite)

```
template <typename T>
std::valarray<T>
tridiag<T>::operator* (const std::valarray<T>& x) const {
    return L.shift(-1)*x.shift(-1) + D*x + L*x.shift(1);
}
```

Cette façon concise 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`.

Enfin, passons aux fonctions-membres de la classe qui en administrent la gestion : ce sont, dans l'ordre, un constructeur, le constructeur de copie, et l'opérateur d'affectation. Le constructeur admet trois paramètres qui permettent des initialisations simples :  $n$ , la taille de la matrice et  $a$  et  $b$  permettent d'initialiser à des constantes les coefficient de la diagonale et des sur- et sous-diagonales, respectivement. Par défaut,  $a$  et  $b$  sont à zéro, ainsi que  $n$ . Notons que le constructeur prend soin d'initialiser à zéro le dernier élément de  $L$ . Les constructeurs de copie et d'affectation sont ici

assez similaires sur le principe de recopier leur argument mais sont assez différents dans leur écriture. Ils explicitement spécifiés ici, car la classe `tridiag<T>` contient deux tableaux `valarray<T>`, de taille variable  $n$ , déterminée dynamiquement lors de l'exécution. Ces tableaux doivent être alloués et recopiés par les opérateurs de copie et d'affectation.

`tridiag.h` (suite)

```
template <typename T>
tridiag<T>::tridiag (size_t n, const T& a, const T& b)
    : D(a,n), L(b,n)
{
    if (n > 0) L[n-1] = 0;
}
template <typename T>
tridiag<T>::tridiag (const tridiag<T>& A)
    : D(A.D), L(A.L)
{
}
template <typename T>
tridiag<T>& tridiag<T>::operator= (const tridiag<T>& A) {
    D.resize (A.D.size());
    L.resize (A.L.size());
    D = A.D;
    L = A.L;
    return *this;
}
```

Nous allons maintenant représenter la factorisation  $LDL^T$  par une classe qui va prendre en argument une matrice tridiagonale, la factoriser, et proposer un interface concis et clair pour la résolution du système linéaire :

`tridiag_ldlt.h`

```
#include <iostream>
#include "tridiag.h"
template <typename T>
class tridiag_ldlt : protected tridiag<T> {
public:
    tridiag_ldlt (const tridiag<T>& A);
    std::valarray<T> solve (const std::valarray<T>& b) const;
};
```

Le constructeur effectue la factorisation :

`tridiag_ldlt.h` (suite)

```
#include "inplace_factorize.h"
template <typename T>
tridiag_ldlt<T>::tridiag_ldlt (const tridiag<T>& A)
    : tridiag<T>(A)
{
    inplace_factorize (tridiag<T>::D, tridiag<T>::L);
}
```

La résolution s'écrit alors simplement :

tridiag\_ldlt.h (suite)

```
#include "inplace_solve.h"
template <typename T>
std::valarray<T>
tridiag_ldlt<T>::solve (const std::valarray<T>& b) const {
    std::valarray<T> x = b;
    inplace_solve (tridiag<T>::D, tridiag<T>::L, x);
    return x;
}
```

Ces deux classes sont à présent complètes : nous sommes prêt à les utiliser. Le paragraphe suivant en illustre l'utilisation avec des subdivisions non-uniformes.

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

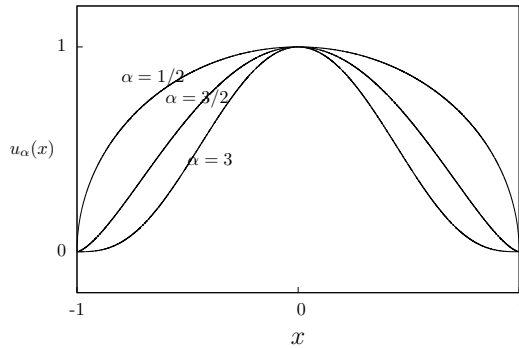


FIGURE 2.4 – Aspect de la solution  $u_\alpha(x)$  sur  $[-1, 1]$  suivant  $\alpha$ .

Ce paragraphe a pour but l'étude du problème modèle de Poisson lorsque la solution est peu régulière. Quitte à ajuster la donnée  $f$  de façon adéquate, nous pouvons choisir la solution de la forme :

$$u_\alpha(x) = (1 - x^2)^\alpha, \text{ avec } \alpha > 1/2.$$

Cette fonction, dépendant d'un paramètre, peut être décrite en C++ de façon élégante à l'aide d'une *classe-fonction* :

u\_alpha.h

```
struct u {
    u (double a1) : a(a1) {}
    double operator() (double x) const { return pow (1-x*x, a); }
    double a;
};
```

Cette implémentation permettra en particulier de passer une instance de la classe-fonction à l'opérateur d'interpolation de Lagrange, exactement comme s'il s'agissait d'une fonction usuelle.

La Fig. 2.4 représente  $u_\alpha(x)$  : nous pouvons observer que  $u'_\alpha(x)$  peut devenir infini  $x = \pm 1$  pour certaines valeurs de  $\alpha$ . 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  $s < \alpha + 1/2$ .

Avec une subdivision uniforme de pas  $h > 0$ , 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 une erreur d'interpolation en  $\mathcal{O}(h^{\min(\alpha, 2)})$ .

L'idée est 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, en ce sens qu'elle est adaptée à la solution à calculer. La fonction  $\chi_\beta(t)$  s'écrit très simplement :

chi\_beta.h

```
double chi (double b, double t) {
    return (b*t - 1 - pow (t, b) + pow(1-t, b))/(b-2);
}
```

Nous disposons à présent de tous les éléments pour écrire un programme de calcul :

## fish1d.adapt.cc

```

#include <iostream>
#include "valarray_util.h"
#include "range.h"
#include "mass.h"
#include "tridiag_ldlt.h"
#include "u_alpha.h"
#include "f_alpha.h"
#include "chi_beta.h"
#include "interpolate_1d.h"
using namespace std;
int main (int argc, char** argv) {
    size_t n = (argc > 1) ? atoi(argv[1]) : 11;
    double alpha = (argc > 2) ? atof(argv[2]) : 0.5,
           beta = (argc > 3) ? atof(argv[3]) : 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);
    valarray<double> M(n-1);
    tridiag<double> A(n-1);
    mass (x, M);
    energy (x, A);
    tridiag_ldlt<double> C(A);
    valarray<double> pi_h_f = interpolate (x, f(alpha)),
                    b = M*valarray<double>(pi_h_f[range(1,n)]),
                    uh (0.0, n+1);
    uh [range(1,n)] = C.solve (b);
    valarray<double> pi_h_u = interpolate (x, u(alpha)),
                    u_err = (pi_h_u - uh)[range(1,n)],
                    uerr_linf = abs(u_err);
    cerr << "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;
}

```

Le programme prend en entrée, sur la ligne de commande, les données  $n$ ,  $\alpha$  et  $\beta$ . Après avoir résolu le problème sur la subdivision adaptée, nous obtenons en sortie l'erreur  $\pi_h(u_\alpha) - u_h$  entre l'interpolée de la solution exacte et la solution approchée, ceci dans les normes  $L^2$ ,  $L^\infty$  et dans la semi-norme de l'énergie, équivalente à la norme  $H^1$  dans  $H_0^1$ .

## f\_alpha.h

```

struct f {
    f (double a1) : a(a1) {}
    double operator() (double 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));
    }
    double a;
};

```

La donnée  $f$  en second membre du problème de Poisson a été ajusté de façon à ce que  $u_\alpha$  soit solution, c'est-à-dire en choisissant  $f = -u''_\alpha$ .

La compilation et l'exécution du code sont :

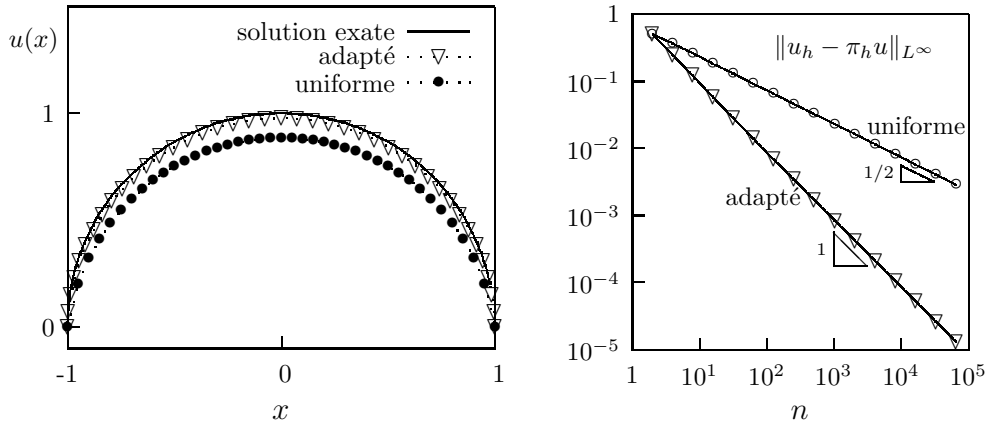


FIGURE 2.5 – Approximation sur les subdivisions uniformes et adaptées : à gauche, solutions pour  $n = 40$  et  $\alpha = 1/2$  ; à droite, convergence vers  $u(x) = (1 - x^2)^{1/2}$  en fonction de  $n$  pour  $\alpha = 1/2$  avec la norme  $L^\infty$ .

```
make fish1_adapt
./fish1d_adapt_tst 1000 0.5 1
./fish1d_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. Le premier calcul avec  $\beta = 1$  correspond à la subdivision uniforme, le second, avec  $\beta = 4$ , à l'adaptation.

La figure 2.5.a compare ces deux solutions approchées avec 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 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 2.5.b 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  $\mathcal{O}(n^{-1/2}) = \mathcal{O}(h^{1/2})$ . Dans le cas adapté, l'erreur est en  $\mathcal{O}(n^{-1}) = \mathcal{O}(h)$ . La convergence est donc bien plus rapide pour un maillage adapté. 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.

## 2.2.7 Exercices

EXERCICE 10. (*Convergence de la méthode des différences finies*)

On se place dans le cas de la dimension un. Ajuster le second membre  $f(x)$  pour que la solution du problème de Poisson dans  $] -1, 1[$  soit  $u(x) = \cos(\pi x/2)$ . Cette solution n'est pas polynomiale, et la solution calculée par différences finies ne coïncide donc pas avec l'interpolée  $\pi_h u$  de la solution exacte, comme c'était le cas dans l'exemple

au début de ce chapitre.

- 1) Modifier le code `fish1d.cc` de ce chapitre, pour qu'il calcule interpolée de Lagrange de  $u$  et celle de  $f$  à l'aide de la fonction `interpolate` du fichier `interpolate_1d.h`.
- 2) Résoudre le problème de Poisson et calculer l'erreur  $e_h = \max_i |u_i - u(x_i)|$ .
- 3) Faire varier la taille du problème  $n$  géométriquement suivant  $n = 2^k$ , pour  $k = 1, 2, 3 \dots 15$ . Tracer l'erreur  $e_h$  en fonction de  $h$  en axes logarithmiques. En déduire que l'erreur se comporte comme une fonction puissance de  $h$ , de la forme  $e_h = ch^\alpha$  et estimer les valeurs de  $c$  et  $\alpha$ .

EXERCICE 11. (*Factorisation  $LDL^T$  avec structure bande*)

Cet exercice utilise le résultat sur les matrices à structure bande introduit à l'exercice 4, page 22.

- 1) L'approximation par éléments finis de degré  $P_k$  (voir par exemple dans [37] pour les détails de cette méthode) en dimension un d'espace conduit à un système linéaire dont la largeur de bande est  $b = k$ . En déduire le temps asymptotique d'exécution de la résolution.
- 2) L'approximation par différences finies en dimension  $d$  d'espace avec un pas de discrétisation  $h = 1/m$  conduit à une matrice de taille totale  $n = (m+1)^d$  et à une largeur de bande  $b = \Theta(m^{d-1})$ . Montrer que le temps de factorisation  $LDL^T$  exploitant la structure bande est

$$T(n) = \Theta\left(n^{3-2/d}\right)$$

et que le temps de résolution des systèmes triangulaires est  $\Theta(n^{2-1/d})$ .

### 2.2.8 Notes

La résolution du problème de Poisson en dimension un est un sujet extrêmement classique, qui est traité dans de nombreux manuels. Pour des compléments théoriques, tels que la preuve de l'existence et de l'unicité de la solution ainsi que de la solution du problème approché, le lecteur pourra consulter [37]. La régularité de la solution exacte dépend de celle de la donnée  $f$ , et cette régularité influe sur les propriétés de convergence de la solution approchée vers la solution exacte : pour présenter ces notions a priori assez abstraites de façon plus concrète, nous avons présenté un exemple et illustré la dégradation des propriétés de convergence, ainsi que leur restauration par l'utilisation d'un maillage adapté. Cette notion d'adaptation de maillage s'étend au cas des problèmes en dimension deux ou trois : la librairie [43], propose également une implémentation C++ de ces méthodes.

Dans ce livre, nous aurons l'occasion d'étudier plusieurs approches du problème de Poisson pour des dimensions supérieures en espace : le chapitre 3.4 présentera la méthode des éléments finis et le chapitre 2.3 une résolution rapide dans le cadre de la méthode des différences finies.

Le concept de classe-fonction présenté dans l'exemple `u_alpha.h`, page 42, dans le contexte du calcul scientifique a été introduite de façon plus générale dans la *standard template library*, abrégée en STL [35] et incluse aujourd'hui dans la librairie standard du langage C++.

## 2.3 Application aux différences finies multi-dimensionnelles

### 2.3.1 Les différences finies en grilles régulières

Soit  $\Omega$  un ouvert borné du plan  $\mathbb{R}^2$ , de frontière  $\partial\Omega$ . Considérons le problème suivant :

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

$$\begin{aligned} -\Delta u &= f \text{ dans } \Omega \\ u &= 0 \text{ sur } \partial\Omega \end{aligned} \tag{2.9}$$

où  $f$  est une fonction donnée de  $\Omega$  dans  $\mathbb{R}$ . Il s'agit d'une extension du problème du chapitre précédent à la dimension deux en espace. Rappelons que résoudre ce problème permet par exemple de connaître la déformation  $u$  d'une membrane ou d'une voile subissant la charge  $f$ , ou bien la vitesse  $u$  d'un fluide en écoulement établi dans une section de forme  $\Omega$  sous la poussée  $f$ , ou encore de résoudre divers problèmes de thermique.

La solution du problème (2.9) n'est généralement pas connue explicitement sauf pour quelques cas très particuliers de géométrie  $\Omega$  et de donnée  $f$ . Dans le cas où  $\Omega$  est rectangulaire, et pour une donnée  $f$  générale, 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 discrétisé par différences finies s'énonce :

*trouver  $(u_{i,j})_{0 \leq i,j \leq m}$  telle que*

$$\begin{aligned} -u_{i,j+1} - u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j-1} &= h^2 f(x_i, x_j), \quad 1 \leq i, j \leq m-1 \\ u_{i,0} &= u_{i,m} = 0, \quad 0 \leq i \leq m \\ u_{0,j} &= u_{m,j} = 0, \quad 1 \leq j \leq m-1 \end{aligned}$$

où  $h = 1/m$  et  $x_i = ih$ ,  $0 \leq i \leq m$ . Notons  $n$  taille de ce système linéaire : nous avons  $n = (m-1)^2$ . Numérotons ensuite les inconnues  $(u_{i,j})_{1 \leq i,j \leq m-1}$  par ligne d'abord, puis par colonne ensuite :  $\mathcal{I}(i,j) = (m-1)(j-1) + i - 1 \in [0, n-1]$  pour  $1 \leq i, j \leq m$ . L'interpolée  $\pi_h v$  d'une fonction  $v$  sur la grille s'obtient par :



## interpolate\_2d.h

```

#include <valarray>
template <typename 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  $m-1$  et  $\tilde{C} = 4.I - C_0$  est la matrice, d'ordre  $m-1$ , 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.

### 2.3.2 Estimation du coût pour différentes résolutions classiques

Passons en revue les différentes méthodes que nous avons abordé jusqu'à présent et qui permettrait de résoudre ce problème.

Nous pouvons aborder la résolution de ce système par une méthode directe : une résolution directe par application brutale de la factorisation  $LDL^T$  présentée à la section 1.2.5, page 19 conduit à un coût en temps de calcul :

$$T(n) = \Theta(n^3)$$

Nous avons vu dans ce même paragraphe que l'algorithme récursif, de type Strassen, conduirait à une amélioration, en réduisant le coût à

$$T(n) = \Theta(n^{\log_2(7)})$$

avec  $\log_2(7) \approx 2.81$ .

Remarquons que la matrice  $A$  a une structure bande. En exploitant celle-ci, comme cela a été analysé à l'exercice 11 page 46, nous obtenons, pour la dimension deux d'espace, un coût en temps de calcul de la factorisation  $LDL^T$  :

$$T(n) = \Theta(n^2)$$

La prise en compte de cette structure bande conduit donc à un gain notable.

Ne nous arrêtons pas là : nous pouvons également aborder la résolution de ce système par une méthode itérative en utilisant l'algorithme du gradient conjugué (voir au chapitre 3.1). La matrice  $A$  étant très creuse, avec au plus cinq coefficients non-nuls par ligne, un produit matrice-vecteur aura un coût en temps de calcul  $\Theta(n)$  si nous prenons soin de d'effectuer les opérations que sur les coefficients non-nuls. Par conséquent, exécuter  $n_{\text{iter}}$  itérations de l'algorithme du gradient conjugué auront un coût en temps de calcul  $T(n) = \Theta(n_{\text{iter}}n)$ . Utilisons à présent la relation (3.2), page 65, pour estimer le nombre d'itérations nécessaires pour atteindre une précision  $\varepsilon$  donnée. En supposant la précision  $\varepsilon$  fixée indépendamment de  $n$ , nous avons  $n_{\text{iter}} = \Theta(\sqrt{\text{cond}(A)})$ . Sachant que  $\text{cond}(A) = \Theta(1/h^2)$  pour notre matrice de différences finies (voir [3, p. 238]) et puisque  $h = 1/m = \Theta(1/\sqrt{n})$ , il vient finalement :

$$T(n) = \Theta(n^{3/2})$$

C'est encore mieux que précédemment, par méthode directe exploitant la structure bande.

Nous pouvons encore améliorer cet algorithme itératif en utilisant un préconditionneur. Supposons par exemple que nous disposons d'un bon préconditionneur  $M$ , tel que  $\text{cond}(M^{-1}A) = \Theta(1/h)$ . Pour des exemples de tels préconditionneurs, voir par exemple [3, p. 356]. Après un calcul analogue au précédent, nous obtenons

$$T(n) = \Theta(n^{5/4})$$

Ce temps de calcul est encore meilleur, et tout à fait intéressant dans la pratique.

Nous allons cependant voir que nous pouvons faire encore mieux, et obtenir un coût

$$T(n) = \Theta(n \log n)$$

en utilisant plus finement la structure très particulière de la matrice  $A$ . Pour cela, nous allons commencer par calculer les valeurs propres de la matrice  $\tilde{C}$ . Ensuite, nous verrons comment la résolution se ramène à  $m$  applications de la transformation de Fourier rapide.

### 2.3.3 Valeurs et vecteurs propres des matrices tridiagonales

Rappelons qu'une matrice de Toeplitz est une matrice qui a ses coefficients constants sur chacune de ses diagonales. La matrice  $\tilde{C}$  est une matrice de Toeplitz symétrique et

tridiagonale. Cherchons les valeurs propres de la matrice  $\tilde{C}$ . Remarquons que matrice du problème de Poisson en dimension un, apparaissant dans (2.5), page 34, est obtenue de  $\tilde{C}$  en remplaçant les 4 sur la diagonale par des 2 :  $C = 2I - C_0$ . Nous allons montrer la propriété bien utile suivante.

**PROPRIÉTÉ 1** *La matrice d'ordre  $m - 1$*

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

*a pour valeurs propres*

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

*et pour vecteurs propres unitaires associés :*

$$v_k = \left( \sqrt{\frac{2}{m}} \sin\left(\frac{jk\pi}{m}\right) \right)_{1 \leq j \leq m-1}$$

*Démonstration :* Remarquons que tout d'abord que  $\lambda$  est valeur propre de  $\text{tridiag}_{m-1}(\alpha, \beta)$  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  $\text{tridiag}_{m-1}(\alpha, \beta)$ . Il nous suffit donc de trouver les valeurs propres de  $C_0$ . Introduisons le polynôme caractéristique de la matrice  $C_0$  d'ordre  $k \geq 2$  :

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

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

$$p_k(\mu) = \mu p_{k-1}(\mu) - p_{k-2}(\mu), \quad k \geq 3. \quad (2.10)$$

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  $k \geq 1$ . Effectuons ensuite un changement de variable. Posons :

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

Avec ce changement de variable, 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 à présent la formule de récurrence caractéristique des polynômes de Tchebychev. Un second changement de variable va nous permettre de résoudre cette récurrence : posons  $\nu = \cos \varphi$ . La formule de récurrence devient

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

et comparons-la à la formule trigonométrique suivante :

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

Ceci nous conduit à poser

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

Le dénominateur sert à assurer l'amorçage de la récurrence, 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  $q_{m-1}(\nu) = 0$ , soit encore ;

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

Ceci équivaut à  $\varphi \notin \{0, \pi\}$  et  $\exists k \in \mathbb{Z}$  tel que  $m\varphi = k\pi$ . Les  $m-1$  valeurs propres  $(\mu_k)_{1 \leq k \leq m-1}$  de  $C_0$  sont donc données, en ordre croissant, par :

$$\mu_k = 2 \cos \left( \frac{k\pi}{m} \right), \quad 1 \leq k \leq m-1$$

Recherchons à présent les vecteurs propres associés. Remarquons que les  $m-1$  premières relations de récurrences (2.10), avec  $k = 1, \dots, m-1$  s'écrivent aussi sous la forme matricielle suivante :

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

Posons  $v(\mu) = (p_0(\mu), \dots, p_{m-2}(\mu))^T$ . En choisissant  $\mu = \mu_k$ , qui est une racine de  $p_{m-1}$ , il vient :

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

Ainsi,  $v(\mu_k)$  est un vecteur propre associé à la valeur propre  $\mu_k$ . Il nous reste à normaliser ce vecteur propre. Nous avons

$$\|v(\mu_k)\|^2 = \sum_{j=0}^{m-2} p_j(\mu_k)^2 = \frac{\sum_{j=1}^{m-1} \sin^2\left(\frac{jk\pi}{m}\right)}{\sin^2\left(\frac{k\pi}{m}\right)}$$

Une inspection de la somme donne successivement :

$$\begin{aligned} \sum_{j=1}^{m-1} \sin^2\left(\frac{jk\pi}{m}\right) &= \frac{1}{4} \sum_{j=1}^{m-1} \left(2 - e^{2ijk\pi/m} - e^{-2ijk\pi/m}\right) \\ &= \frac{m-1}{2} - \sum_{j=1}^{m-1} e^{2ijk\pi/m} + e^{-2ijk\pi/m} \end{aligned}$$

En rappelant que  $1 + \xi + \dots + \xi^{m-1} = (1 - \xi^m)/(1 - \xi)$ , puis, avec  $\xi = e^{\pm 2ik\pi/m}$  et en remarquant que  $\xi^m = 1$ , il vient finalement :

$$\|v(\mu_k)\|^2 = \frac{m}{2 \sin^2\left(\frac{k\pi}{m}\right)}$$

d'où le résultat annoncé.  $\square$

Nous allons à présent appliquer ce résultat à la résolution du problème de Poisson en dimension deux.

### 2.3.4 Résolution directe par transformée de Fourier

Les valeurs propres de  $\tilde{C} = 4.I - C_0$ , de taille  $m-1$  sont  $\lambda_k = 4 - 2\mu_k = 4 - 2\cos(k\pi/m)$ ,  $1 \leq k \leq m-1$ . Afin d'effectuer un changement de base, introduisons la matrice  $V = (V_{k,l})_{1 \leq k, l \leq m-1}$  dont les colonnes sont les vecteurs propres associés :

$$V_{l,k} = \sqrt{\frac{2}{m}} \sin\left(\frac{kl\pi}{m}\right), \quad 1 \leq k, l \leq m-1. \quad (2.11)$$

La matrice  $\tilde{C}$  étant symétrique, les vecteurs sont orthogonaux entre eux. De plus, ces vecteurs sont de norme unité, si bien que  $V^T V = V V^T = I$ , soit encore  $V^{-1} = V^T$ , autrement dit la matrice de changement de base est unitaire. Ainsi  $V^T \tilde{C} V = \Lambda$ , où  $\Lambda$  désigne la matrice diagonale composée des valeurs propres :  $\Lambda_{k,k} = \lambda_k$ ,  $1 \leq k \leq m-1$ . De plus, de par sa définition (2.11), remarquons que la matrice  $V$  est ici symétrique,

si bien que  $V = V^T = V^{-1}$  et  $VV = I$ . Nous avons à présent des outils pour revenir au système de Poisson (2.3.1), page 47. Ce système peut s'écrire par blocs :

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

avec  $u_j = (u_{k,j})_{1 \leq k \leq m-1}$  et  $f_j = (f_{k,j})_{1 \leq k \leq m-1}$ . Décomposons  $u_j$  et  $f_j$  dans la base associée aux vecteurs propres :  $u_j = V^T \tilde{u}_j$  et  $f_j = V^T \tilde{f}_j$ ,  $1 \leq j \leq m-1$ . Alors le système devient :

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

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

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

Ainsi, ces  $m-1$  systèmes linéaires sont découplés : pour chaque  $k$  fixé, nous sommes en présence d'un système tridiagonal de matrice  $A_k = \lambda_k I - C_0$ , que nous savons résoudre par une méthode directe en un temps de calcul optimal (voir section 2.2.2, page 34). Remarquons que la matrice  $A_k$  est elle aussi une matrice de Toeplitz symétrique et tridiagonale. Nous pouvons proposer un premier algorithme de résolution :

**algorithme** : Poisson bidimensionnel sur grille uniforme  
**entrée**  $(f_{k,j})_{1 \leq k,j \leq m-1}$   
**sortie**  $(u_{k,j})_{1 \leq k,j \leq m-1}$   
**début**  
  **pour**  $j := 1 \dots m-1$   
    **pour**  $k := 1 \dots m-1$   
       $\tilde{f}_{k,j} := \frac{2}{m} \sum_{p=1}^{m-1} \sin\left(\frac{kp\pi}{m}\right) f_{p,j}$   
    **pour**  $k := 1 \dots m-1$   
       $\lambda_k := 4 - 2 \cos k\pi/m$   
      trouver  $\tilde{u}_k = (\tilde{u}_{k,j})_{1 \leq j \leq m-1}$  tel que  $(\lambda_k I - C_0)\tilde{u}_k = \tilde{f}_k$   
    **pour**  $j := 1 \dots m-1$   
      **pour**  $k := 1 \dots m-1$   
         $u_{k,j} := \sum_{p=1}^{m-1} \sin\left(\frac{kp\pi}{m}\right) \tilde{u}_{p,j}$   
  **fin**

Faisons une rapide évaluation du temps de calcul pour cet algorithme. La première boucle contient une sous-boucle qui, elle-même contient une somme : son temps de calcul est  $\Theta(m^3) = \Theta(n^{3/2})$ , car  $n = (m-1)^2$  est la taille du système linéaire à

résoudre. La seconde boucle contient la résolution du système tridiagonal : chaque système tridiagonal est résolu de façon optimale en  $\Theta(m)$  si bien que le coût de cette deuxième boucle est  $\Theta(m^2) = \Theta(n)$ . La troisième boucle est similaire à la première et son temps de calcul est également  $\Theta(n^{3/2})$ . La méthode n'est donc *a priori* pas très attractive : nous avons au paragraphe 2.3.2 que l'algorithme du gradient conjugué sans préconditionnement conduisait à ce coût asymptotique, et qu'un préconditionnement permettait de l'abaisser encore.

Nous allons voir que nous pouvons optimiser les première et troisième boucles par l'utilisation de la transformée en sinus, une variante de la transformée de Fourier, pour calculer les sommes, et obtenir ainsi un temps de calcul très attractif  $T(n) = \Theta(n \log n)$ .

### 2.3.5 La transformée en sinus

Considérons la transformée en sinus : d'un vecteur  $v = (v_p)_{1 \leq p \leq m-1}$  :

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

Étendons  $v_p$  à  $p = 2m$  en un échantillon périodique, impair à  $p = m$ , par :

$$\begin{aligned} v_{2m-p} &= -v_p, \quad p = 1 \dots m-1, \\ v_0 &= v_m = u_{2m} = 0, \end{aligned}$$

et observons la transformée de Fourier :

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

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

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

Ainsi :

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

Nous savons à présent calculer cette somme en un temps  $\mathcal{O}(m \log m)$ . La fonction suivante implémente la transformée en sinus, les tableaux  $v$  et  $\tilde{v}$  ayant été renumérotés

à partir de zéro pour suivre la convention du langage C++.

fft\_sin.h

```
#include "fft.h"
#include <valarray>
#include <complex>
#include <cassert>
template <typename T>
std::valarray<T> fft_sin (const std::valarray<T>& v) {
    using namespace std;
    size_t m = v.size()+1;
    valarray<complex<T>> v_ext(2*m), hat_v(2*m);
    v_ext[0] = 0;
    for (size_t p = 1; p < m; ++p) {
        v_ext[p] = v[p-1];
        v_ext[2*m-p] = -v[p-1];
    }
    const T pi = acos(T(-1));
    complex<T> i (0,1), w = exp((pi/m)*i);
    assert(fabs(log2(m)-int(log2(m))) < 1e-10);
    fft (w, v_ext, hat_v);
    std::valarray<T> tilde_v(m-1);
    for (size_t p = 1; p < m; ++p)
        tilde_v[p-1] = m*hat_v[p].imag();
    return tilde_v;
}
```

Pour augmenter la lisibilité du code, qui utilise les tableaux unidimensionnels `valarray<T>` pour ranger des données bidimensionnelles  $u_{k,j}$ , nous avons utilisé les aides suivantes, qui permettent d'accéder à une colonne ou une ligne spécifique :

row\_col.h

```
#include <valarray>
struct row_type {
    row_type (size_t n1) : n(n1) {}
    std::slice operator() (size_t j) const {
        return std::slice (n*j, n, 1); }
protected:
    size_t n;
};
struct col_type {
    col_type (size_t n1) : n(n1) {}
    std::slice operator() (size_t i) const {
        return std::slice (i, n, n); }
protected:
    size_t n;
};
```

Dans le même esprit, nous pouvons accéder à un élément particulier à l'aide de la petite classe suivante :

index.h

```
struct index_type {
    index_type (size_t n1) : n(n1) {}
    size_t operator() (size_t i, size_t j) const { return j+i*n; }
protected:
    size_t n;
};
```



### 2.3.6 Un algorithme de résolution rapide

L'algorithme en trois boucles précédent s'écrit alors :

fish2d\_fft.h

```
#include "fft_sin.h"
#include "tridiag_ldlt.h"
#include "row_col.h"
template <typename T>
std::valarray<T> fish2d_fft(size_t m, const std::valarray<T>& b) {
    using namespace std;
    valarray<T> tilde_b(b.size()), tilde_u(b.size()), u(b.size());
    col_type col(m-1);
    for (size_t j = 0; j < m-1; j++) {
        valarray<T> bj = b[col(j)];
        tilde_b[col(j)] = (2./m)*fft_sin(bj);
    }
    row_type row(m-1);
    const T pi = acos(T(-1));
    for (size_t k = 0; k < m-1; k++) {
        tridiag<T> poisson_1d (m-1, 4 - 2*cos((k+1)*pi/m), -1);
        tridiag_ldlt<T> factorized_1d (poisson_1d);
        tilde_u[row(k)] = factorized_1d.solve (tilde_b [row(k)]);
    }
    for (size_t j = 0; j < m-1; j++) {
        valarray<T> tilde_uj = tilde_u[col(j)];
        u[col(j)] = fft_sin(tilde_uj);
    }
    return u;
}
```

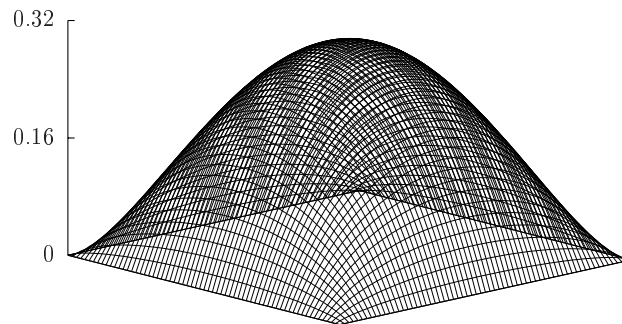


FIGURE 2.6 – Solution du problème de Poisson en dimension deux : vue en élévation  $z = u_h(x, y)$ .

Voici un programme qui appelle cette résolution rapide :

fish2d\_fft.cc

```

#include "fish2d_fft.h"
#include "interpolate_2d.h"
#include "index.h"
#include "valarray_io.h"
using namespace std;
double f (const double& x, const double& y) { return 1; }
int main (int argc, char** argv) {
    size_t m = (argc > 1) ? atoi(argv[1]) : 64;
    double h = 2./m;
    index_type elt (m+1), num (m-1);
    valarray<double> pi_h_f = interpolate(m, f), b ((m-1)*(m-1));
    for (size_t i = 0; i < m-1; i++)
        for (size_t j = 0; j < m-1; j++)
            b [num(i,j)] = h*h*pi_h_f [elt (i+1,j+1)];
    valarray<double> uh(0.,(m+1)*(m+1)), x = fish2d_fft(m, b);
    for (size_t i = 0; i < m-1; i++)
        for (size_t j = 0; j < m-1; j++)
            uh [elt(i+1, j+1)] = x[num(i,j)];
    cout << uh;
}

```

```

make fish2d_fft
./fish2d_fft 64 > uh.data

```

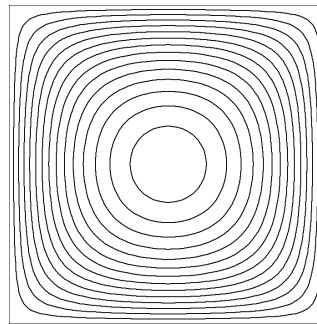
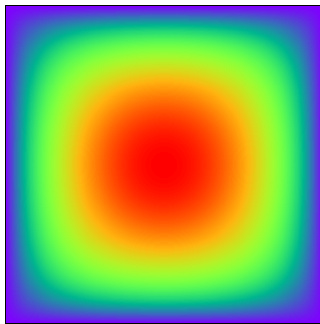


FIGURE 2.7 – Solution du problème de Poisson en dimension deux : (à gauche) vue avec une palette de couleurs; (à droite) avec des lignes isovaleurs.

La solution est écrite dans le fichier `uh.data` que nous pouvons convertir au format `.gdat` afin de le visualiser avec `gnuplot` [56] (voir Fig. 2.6) :

```

make data2gdat
./data2gdat < uh.data > uh.gdat
gnuplot

```

Une fois sous l'interpréteur de commande `gnuplot`, nous entrons simplement :

splot 'uh.gdat' with lines

et la solution apparaît en élévation  $z = u_h(x, y)$ . Le code de conversion, très simple, est :

data2gdat.cc

```
#include "valarray_io.h"
#include "index.h"
using namespace std;
int main (int argc, char** argv) {
    valarray<double> u;
    cin >> u;
    size_t m = sqrt(u.size()) - 1;
    index_type elt(m+1);
    double h = 2./m;
    for (size_t i = 0; i < m+1; i++) {
        for (size_t j = 0; j < m+1; j++)
            cout << -1+i*h << " " << -1+j*h << " " << u[elt(i,j)] << endl;
        cout << endl;
    }
}
```

Une séquence de commandes plus élaborées permet de visualiser les isovaleurs de la solution à l'aide d'une palette de couleurs (voir Fig. 2.7) :

```
set size ratio -1
set view map
set palette rgbformulae 33,13,-4
set pm3d interpolate 10,10 corners2color mean
splot 'uh.gdat' with pm3d
```

Une variante permet de visualiser les lignes isovaleurs une à une :

```
set size ratio -1
set view map
set palette rgbformulae 0,0,0
set cntrparam levels 20
set contour base
unset surface
unset colorbox
splot "uh.gdat" notitle with lines palette
```

### 2.3.7 Exercices

EXERCICE 12. (*Problème de Poisson par Fourier en dimension trois*)

On s'intéresse à la discrétisation par différences finies du problème aux limites de

Poisson en dimension  $d$  dans le domaine  $\Omega = ]0, 1[^3$  :

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

1) Écrire le schéma de général de différences finies d'indices  $(i, j, k)$  avec un pas  $h = 1/m$  constant en dimension trois, ainsi que les conditions aux limites sur les six bords du domaine.

2) En éliminant les inconnues sur le bord, qui sont fixées, quelle est la taille du système réduit ?

3) Écrire le système ainsi réduit à l'aide de blocs matriciels de taille  $m - 1$  sur l'indice  $i$  : le schéma obtenu n'utilisera que les indices  $j$  et  $k$ .

par blocs de taille  $m - 1$ .

2) Effectuer le changement de base, comme pour le cas de la dimension deux. Obtenir  $m - 1$  systèmes linéaires découplés de matrices  $A_k$ ,  $1 \leq k \leq m - 1$ , où la matrice  $A_k$  est de taille  $(m - 1)^2 \times (m - 1)^2$ . Quelle est la structure de la matrice  $A_k$  ? Est-elle symétrique ? Est-elle de Toeplitz ?

3) Proposer une méthode de gradient conjugué pour résoudre ce problème et calculer le temps de calcul global de la méthode obtenue.

4) En utilisant le fait que  $A_k$  est également une matrice de Toeplitz, calculer les valeurs propres de  $A_k$ . Effectuer un second changement de base et découpler une seconde fois le système linéaire. En déduire une nouvelle façon de résoudre ce système linéaire et calculer le temps de calcul global de la méthode obtenue.

### 2.3.8 Notes

Golub et Meurant présentent dans [25] (voir également [34]) l'application de la transformée de Fourier à la résolution rapide du problème de Poisson en dimension deux. Dans ce même ouvrage, les auteurs proposent également une seconde méthode rapide pour résoudre ce même problème : la méthode de réduction cyclique.

La librairie `fishpack` [51] propose un ensemble de fonctions écrites en langage fortran pour résoudre le problème de Poisson en dimensions deux ou trois et étend la méthode au système de coordonnées polaires et au cas de conditions aux bords variées.

On trouvera dans [5] un préconditionneur, construit à partir des propriétés des matrices de Toeplitz, et qui permet aux méthodes itératives de rivaliser avec les méthodes directes tout en ayant des propriétés algorithmiques plus favorables pour le calcul parallèle et à mémoire distribuée.



## Chapitre 3

# Matrices creuses et méthode des éléments finis

### 3.1 Algorithme du gradient conjugué

#### 3.1.1 Présentation pratique de l'algorithme

L'algorithme du gradient conjugué est une méthode efficace pour résoudre les systèmes linéaires d'équations de la forme  $Ax = b$ , dont la matrice  $A$  est symétrique définie positive. Cette méthode procède de façon itérative en construisant une suite  $(x^{(i)})_{i \geq 0}$  qui converge vers la solution. L'idée est de minimiser la fonction quadratique  $J(x) = (Ax)^T x - b^T x$ . Observons que le minimum satisfait l'équation à résoudre et que la fonction  $J$  est convexe : son minimum est unique et est donc bien la solution recherchée. La minimisation est effectuée par une méthode de descente à pas variable  $\alpha_i$  :

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$$

Les directions de descente successives  $p^{(i)}$  sont choisies orthogonales entre elles dans un certain sens que nous allons préciser : nous disons pour cela que les directions sont *conjuguées*. Bien que le nombre d'itérations peut être parfois grand, seulement un petit nombre de vecteurs sont simultanément nécessaires en mémoire, aussi le coût en place mémoire est faible pour cet algorithme. À chaque itération, nous recalculons le résidu  $r^{(i)} = Ax^{(i)} - b$  de l'équation à résoudre. Par récurrence, il peut être remis à jour par la relation :

$$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$$

où on a posé  $q^{(i)} = Ap^{(i)}$ . Parmi tous les choix possibles pour le pas de descente  $\alpha_i$ , le choix  $\alpha_i = r^{(i)T} r^{(i)} / p^{(i)T} q^{(i)}$  minimise la quantité  $r^{(i)T} A^{-1} r^{(i)}$ . La direction de descente est calculé également par récurrence :

$$p^{(i+1)} = r^{(i)} + \beta_i p^{(i)}$$

Le choix  $\beta_i = r^{(i)T} r^{(i)} / r^{(i-1)T} r^{(i-1)}$  garantit que  $p^{(i)}$  et  $Ap^{(i-1)}$  sont orthogonales. Avec ce choix, les résidus successifs  $r^{(i)}$  et  $r^{(i-1)}$  sont également orthogonaux. On peut montrer également qu'avec ce choix de  $\beta_i$ , la direction de descente  $p^{(i)}$  et le résidu  $r^{(i)}$  sont orthogonaux à tous les  $Ap^{(j)}$  et  $r^{(j)}$  précédents,  $j < i$ . Le pseudo-code pour cet algorithme est donné ci-après, avec à droite son implémentation en C++.

	cg.h
<pre> <b>algorithme</b> cg   entrée <math>A, x^{(0)}, b, M</math>   sortie <math>(x^{(i)})_{i \geq 1}</math> <b>début</b>   <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>    <b>pour</b> <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 p^{(i)}</math>     <math>r^{(i)} := r^{(i-1)} - \alpha_i q^{(i)}</math>     si <math>\ r^{(i)}\  &lt; \epsilon</math> alors arrêt      <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>   <b>fin</b> <b>fin</b> </pre>	<pre> #include&lt;iostream&gt; 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) {   using namespace std;   Real nb = norm(b);   if (nb == 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;   clog&lt;&lt;"pcg: 0 "&lt;&lt;norm(r)/nb&lt;&lt;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 nr = norm(r)/nb;     clog&lt;&lt;"pcg: "&lt;&lt;i&lt;&lt;" "&lt;&lt;nr&lt;&lt;endl;     if (nr &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>

Observez tout d'abord l'efficacité du codage, qui permet une très grande lisibilité, et évite ainsi de faire des erreurs. De plus, 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 que nous allons considérer au long de cet ouvrage. La généricité de nécessite que les types **Mat**, **Vec** et **Real** présentent un interface minimal. En particulier, la classe **Vec** doit fournir les fonctionnalités suivantes :

```

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

```

Pour la classe `Mat`, il est seulement requis un produit matrice-vecteur :

$$\text{Vec} \leftarrow \text{Mat} * \text{Vec}$$

Cet algorithme est utilisable pour des matrices denses : de nombreuses bibliothèques C++ proposent une implémentation de telles matrices, telle `blitz++` [53] ou `boost/ublas` [9]. La classe `dmatrix`, proposée dans l'exercice 8, peut également être utilisée. Mais aussi nous pourrions utiliser des matrices creuses, comme la classe `matrix` proposée au chapitre 3.2, et où, pour `Vec`, nous utiliserons la classe `valarray`. Enfin, nous verrons dans la troisième partie de ce livre une implémentation dite *sans matrice* de la résolution des systèmes linéaires : dans ce cas, une classe spécifique renvoie l'effet du produit matrice-vecteur sans que les coefficients de la matrice aient besoin d'être stockés en mémoire : la matrice ayant une structure simple, les coefficients sont simples à re-calculer pour chaque application du produit matrice-vecteur. Dans tous ces cas, il n'est nullement nécessaire de re-programmer les algorithmes de résolution : nous pouvons utiliser le même code pour l'algorithme du gradient conjugué.

Dans l'algorithme précédent, 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 :

$$\text{Vec} \leftarrow \text{Precond.solve}(\text{Vec})$$

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 : les notes en fin de chapitre présentent des lectures sur ce sujet.

### 3.1.2 Propriétés de convergence

L'algorithme du gradient conjugué construit le  $i$ -ème itéré dans l'espace vect  $(r^{(0)}, \dots, A^{(i-1)}r^{(0)})$  et de sorte que  $(x^{(i)} - x)^T A (x^{(i)} - x)$  soit minimum, où  $x$  est la solution, vérifiant  $Ax = b$ . Une condition suffisante pour que ce minimum existe est que  $A$  soit symétrique définie positive. La version préconditionnée de l'algorithme fonctionne sur le même principe, en remplaçant  $A$  par  $M^{-1}A$  et le second membre  $b$



par  $M^{-1}b$  : ceci nécessite que le préconditionneur  $M$  soit également symétrique défini positif. Remarquons que dans ce cas, la minimisation de l'erreur  $x^{(i)} - x$  équivaut à ce que les résidus successifs  $r^{(i)} = Ax^{(i)} - b$  soient orthogonaux entre eux pour la métrique  $M^{-1}$ , ce qui s'écrit  $r^{(i)T} M^{-1} r^{(j)} = 0$  pour  $j \neq i$ .

Soit  $n$  la taille de la matrice carrée  $A$ . Supposons que les résidus successifs pendant les  $n$  premières itérations soient non-nuls : si tel était le cas, la solution serait déjà atteinte. Alors, à la  $n$ -ième itération, l'espace engendré par les résidus, tous non-nuls et orthogonaux entre eux, est l'espace  $\mathbb{R}^n$  tout entier, et ainsi  $x^{(n)} = x$ . Autrement dit, l'algorithme du gradient conjugué converge en au plus  $n$  itérations : l'algorithme peut-être vu comme une méthode directe de résolution.

Pour une matrice dense  $n \times n$ , le coût en temps de calcul d'une itération se réduit à celui d'un produit matrice-vecteur, soit  $\Theta(n^2)$ . Par conséquent, l'algorithme permettant d'obtenir la solution en au plus  $n$  itérations, nous obtenons une première borne supérieure pour la résolution du système linéaire :  $T_{\text{cg}}(n) = \mathcal{O}(n^3)$ . Ainsi, l'algorithme du gradient conjugué, en tant que méthode directe de résolution, n'est pas plus performante que la méthode directe de pivotage de Gauss ou de factorisation de Choleski.

L'intérêt de cet algorithme apparaît lorsqu'il est considéré en tant que méthode itérative : dans la pratique, la taille des matrices peut devenir très grande, de l'ordre de plusieurs millions, et nous souhaitons interrompre les itérations bien avant. Axelsson et Barker [3, p. 14] donnent une majoration de l'erreur commise : à la  $i$ -ième itération, nous disposons d'une solution approchée  $x^{(i)} \neq x$  qui vérifie :

$$\|x^{(i)} - x\|_{M^{-1}A} \leq 2 \left( \frac{\sqrt{\text{cond}(M^{-1}A)} - 1}{\sqrt{\text{cond}(M^{-1}A)} + 1} \right)^i \|x^{(0)} - x\|_{M^{-1}A} \quad (3.1)$$

où  $\|y\|_{M^{-1}A} = (y^T M^{-1} A y)^{1/2}$  et  $\text{cond}(M^{-1}A) = \lambda_{\max}(M^{-1}A) / \lambda_{\min}(M^{-1}A)$  est le conditionnement de  $M^{-1}A$ , c'est-à-dire le rapport de sa plus grande valeur propre sur sa plus petite. L'algorithme converge d'autant plus vite que le conditionnement est proche de l'unité. Par conséquent, un bon préconditionnement  $M$  sera tel que les valeurs propres de  $M$  soient proches de celles de  $A$ . La relation précédente est seulement d'une majoration : dans la pratique, l'erreur est souvent bien meilleure que cela et une prédiction précise de la convergence du gradient conjugué reste difficile.

De (3.1), nous obtenons l'expression du nombre d'itérations  $n_{\text{iter}}$  qui garantit une erreur relative plus petite que  $\varepsilon > 0$  :

$$n_{\text{iter}} = \left\lceil \frac{\log\left(\frac{2}{\varepsilon}\right)}{\log\left(\frac{\sqrt{\text{cond}(M^{-1}A)} + 1}{\sqrt{\text{cond}(M^{-1}A)} - 1}\right)} + 1 \right\rceil$$

Par ailleurs, le coût en temps de calcul d'une itération du gradient conjugué est dominé par le coût du produit matrice-vecteur, noté  $T_{A*x}(n)$  et celui, éventuellement, du préconditionneur, noté  $T_{M^{-1}}(n)$ . En effet, les autres opérations sont des combinaisons

linéaires de vecteurs et des calculs de produits scalaires, toutes ces opérations étant effectués avec un coût optimal  $\Theta(n)$ . Ainsi, le coût de l'algorithme pour une erreur relative plus petite que  $\varepsilon$  s'écrit :

$$T_{\text{cg}}(n) = n_{\text{iter}} \times (T_{A*x}(n) + T_{M^{-1}}(n))$$

Il est possible de diminuer le coût du produit matrice-vecteur lorsque la matrice  $A$  contient un grand nombre de coefficients  $a_{i,j}$  nuls : on prendra alors soin de n'effectuer que les produits avec les coefficients non-nuls. Nous étudierons une structure de donnée et des algorithmes adéquats au chapitre 3.2. Cette situation est très courante lorsque la matrice est issue de la discrétisation d'équations différentielles ou aux dérivées partielles par des méthodes de différences finies ou d'éléments finis, et ces applications seront explorées en détail dans ce livre.

Une autre façon de faire diminuer le coût de l'algorithme est d'élaborer un préconditionneur : cette approche est plus difficile car si  $n_{\text{iter}}$  diminue, il faut prendre garde à ce que  $T_{M^{-1}}(n)$  n'augmente pas trop en contrepartie. Les notes en fin de chapitre présentent des lectures sur ce sujet.

### 3.1.3 Exercices

EXERCICE 13. (*Majoration du nombre d'itérations*)

Le but de cet exercice est d'obtenir une majoration plus simple du nombre d'itérations  $n_{\text{iter}}$  de l'algorithme, et qui soit asymptotiquement aussi fin que celui obtenu dans ce chapitre lorsque le conditionnement de la matrice est grand.

Définissons  $f(x) = (x-1)/(x+1)$  et  $g(x) = \exp(-2/x)$  pour tout  $x > 1$ .

1) Montrer que  $f(x) \leq g(x)$  pour tout  $x > 1$ . Indication : on pourra étudier un développement limité en  $1/x$  au voisinage de zéro.

2) De l'estimation d'erreur (3.1) et du résultat de la question précédente, en déduire la majoration d'erreur suivante :

$$\|x^{(i)} - x\|_{M^{-1}A} \leq 2 \exp\left(\frac{-2i}{\sqrt{\text{cond}(M^{-1}A)}}\right) \|x^{(0)} - x\|_{M^{-1}A}$$

3) De la question précédente, en déduire l'estimation plus simple suivante du nombre d'itération de l'algorithme du gradient conjugué :

$$n_{\text{iter}} = \left\lceil \frac{\sqrt{\text{cond}(M^{-1}A)}}{2} \log\left(\frac{2}{\varepsilon}\right) + 1 \right\rceil \quad (3.2)$$

EXERCICE 14. (*Problème de Poisson multi-dimensionnel*)

On s'intéresse au conditionnement d'une matrice  $A$  issue d'une discrétisation par différences finies du problème aux limites dit de Poisson en dimension  $d$  dans le

domaine  $\Omega = ]0, 1[^d$  :

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

Dans la pratique  $d = 1, 2$  ou  $3$  est la dimension spatiale.

- 1) On discrétise le domaine  $\Omega$  en  $m + 1$  points dans chaque direction spatiale, si bien que  $h = 1/m$ . Rappeler le schéma de différences finies pour le problème précédent en dimensions  $d = 1, 2$  et  $3$ .
- 2) On élimine du système linéaire les valeurs aux bords, qui sont fixées. Calculer le nombre d'inconnues  $n$  du problème ainsi réduit en fonction de  $m$  et  $d$ .
- 3) Montrer que le nombre d'éléments non nuls par ligne de la matrice  $A$  du système est majoré par une constante qui ne dépend que de  $d$ , et non de  $n$ , et que on explicitera. En déduire le coût asymptotique d'un produit matrice-vecteur lorsque  $n$  devient grand, en supposant que seuls les coefficients non nuls de  $A$  interviennent.
- 4) Nous admettons ici que  $\text{cond}(A) = \mathcal{O}(h^{-2})$  lorsque  $h \rightarrow 0$ , où  $h$  est le pas de discrétisation spatiale (voir [3, p. 238]). Exprimer  $h$  en fonction de  $n$  et  $d$ . En déduire une expression asymptotique du conditionnement de  $A$  qui ne dépende que de  $n$  et  $d$ .
- 5) Montrer l'estimation asymptotique suivante du nombre d'itérations  $n_{\text{iter}}$  nécessaires pour garantir une erreur relative plus petite que  $\varepsilon > 0$  par l'algorithme du gradient conjugué (sans préconditionnement) :

$$n_{\text{iter}} = \mathcal{O}\left(n^{1/d} \log(\varepsilon^{-1})\right) \quad (3.3)$$

où  $d$  est la dimension spatiale. Pour cela, on utilisera le résultat (3.2) de l'exercice 13.

- 6) En déduire une majoration du coût asymptotique de l'algorithme du gradient conjugué pour cette matrice, en fonction de  $n$ ,  $d$  et  $\varepsilon$ .
- 7) Supposons que nous disposions d'un préconditionnement  $M$  pour  $A$  tel que  $\text{cond}(M^{-1}A) = \sqrt{\text{cond}(A)}$ . Les notes en fin de chapitre donnent des références sur des préconditionneurs vérifiant cette propriété. Calculer alors une majoration asymptotique de  $n_{\text{iter}}$  dans ce cas. En supposant que le coût asymptotique de ce préconditionneur soit au plus de l'ordre d'un produit matrice-vecteur pour  $A$ , en déduire le coût de l'algorithme du gradient conjugué avec ce préconditionnement.
- 8) Comparer le coût de la résolution par cette méthode avec celles vues au chapitre 1 : la méthode directe pour  $d = 1$  ou par la méthode utilisant la transformation de Fourier, pour  $d = 2$ . Suivant  $d$ , quelle est la plus performante pour résoudre le problème issu de la discrétisation en différences finie avec une grille uniforme ?

### 3.1.4 Notes

L'ouvrage de Golub et van Loan [24] est une excellente référence pour les algorithmes itératifs tandis que celui d'Axelsson et Barker [3] affine certains résultats théoriques

en vue de les appliquer à la méthode des éléments finis. Une synthèse pragmatique de ces résultats est présentée dans [4]. En s'appuyant sur cette synthèse, Dongarra *et al.* ont introduit en 1994 l'idée d'une librairie générique en C++ d'algorithmes pour la résolution de systèmes linéaires d'équations par des méthodes itératives : la librairie **IML++** [17]. Cette librairie permet d'étendre le domaine d'application des méthodes itératives de résolution du type gradient conjugué aux matrices non symétriques :

- **bicgstab** : gradient bi-conjugué stabilisé
- **gmres** : résidu minimal généralisé
- **qmr** : résidu quasi-minimal

Pour les matrices symétriques mais non définies, c'est-à-dire ayant des valeurs propres qui s'annulent ou changent de signe, on a recours à l'algorithme du gradient minimal **minres**, qui minimise la fonction quadratique  $J(x) = \|Ax - b\|^2$ . Enfin, la librairie [43], plus récente, propose également une implémentation générique en C++ de cette collection d'algorithmes itératifs.

Benzi présente un panorama des préconditionneurs les plus classiques [6]. Axelsson et Barker [3, p. 14] donnent également des estimations asymptotiques du nombre d'itérations pour des problèmes de type éléments finis et pour certains préconditionneurs classiques. La factorisation incomplète de Choleski est une méthode de préconditionnement très utilisée, notamment sa variante avec compensation de la diagonale, généralement désignée par le sigle **mic0**, et qui permet de garantir que  $\text{cond}(M^{-1}A) = \mathcal{O}(h^{-1})$  pour les matrices issues de la méthode des éléments finis. Un excellent ouvrage d'introduction au préconditionnement est celui de Saad [41], avec un intérêt pour le cas des matrices non symétriques. La librairie **sparskit** [40], écrite en langage fortran par Saad, propose de nombreux préconditionneurs présentés dans [41]. Plus récente et écrite en langage C++, la librairie **ifpack** [42] de l'environnement **trilinos** présente également une grande variété de préconditionneurs.

## 3.2 Matrices creuses

### 3.2.1 La structure de données

Pour aborder des système matrice-vecteurs généraux, nous devons 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 `start[i + 1] - start[i]`. La déclaration de classe ressemble à :

matrix.h

```
#include <valarray>
#include <iostream>
#include <list>
#include <vector>
template <typename T>
class matrix {
public:
    matrix ();
    matrix (const matrix<T>&);
    matrix<T>& operator= (const matrix<T>&);
    void resize (const std::vector<std::list<size_t> >& s, size_t lncol);
    size_t nrow () const;
    size_t ncol () const;
    size_t nnz () const;
    void resize (size_t nrow, size_t ncol, size_t nnz);
    std::valarray<T> operator* (const std::valarray<T>&) const;
    const T& operator() (size_t i, size_t j) const;
    T& entry (size_t i, size_t j);
    template <typename U> friend
    std::istream& operator>> (std::istream&, matrix<U>&);
    template <typename U> friend
    std::ostream& operator<< (std::ostream&, const matrix<U>&);
protected:
    std::valarray<size_t> ptr;
    std::valarray<size_t> idx;
    std::valarray<T> val;
    size_t idxmax;
};
```

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

matrix.h (suite)

```
template<typename T>
matrix<T>::matrix () : ptr(0), idx(0), val(0), idxmax(0) {}
template<typename T>
matrix<T>::matrix (const matrix<T>& a)
: ptr(a.ptr), idx(a.idx), val(a.val), idxmax(a.idxmax) { }
template<typename T>
matrix<T>& matrix<T>::operator= (const matrix& a) {
    resize (a.nrow(), a.ncol(), a.nnz());
    ptr = a.ptr;
    idx = a.idx;
    val = a.val;
    idxmax = a.idxmax;
    return *this;
}
template<typename T>
void matrix<T>::resize (size_t nrow, size_t ncol, size_t nnz) {
    ptr.resize (nrow+1);
    idx.resize (nnz);
    val.resize (nnz);
    idxmax = ncol;
}
template<typename T>
size_t matrix<T>::nrow () const { return ptr.size() - 1; }
template<typename T>
size_t matrix<T>::ncol () const { return idxmax; }
template<typename T>
size_t matrix<T>::nnz () const { return idx.size(); }
```

Les entrée-sortie suivent. Tout d'abord l'écriture en sortie d'une matrice creuse, relativement simple :

matrix.h (suite)

```
template<typename T>
std::ostream& operator<< (std::ostream& s, const matrix<T>& a) {
    using std::endl;
    s << "%MatrixMarket matrix coordinate real general" << endl
    << a.nrow() << " " << a.ncol() << " " << a.nnz() << endl;
    for (size_t i = 0; i < a.nrow(); ++i)
        for (size_t p = a.ptr [i]; p < a.ptr [i+1]; ++p)
            s << i+1 << " " << a.idx [p]+1 << " " << a.val [p] << endl;
    return s;
}
```

La lecture en entrée suit :

## matrix.h (suite)

```

#include<cassert>
template<typename 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.ptr = 0;
    for (size_t i_prec = 0, p = 0; p < nnz; ++p) {
        size_t i;
        is >> i >> a.idx [p] >> a.val [p];
        i--;
        a.idx [p]--;
        assert (a.idx[p] < ncol);
        if (p == 0 || i != i_prec) {
            assert (p == 0 || i_prec < i);
            i_prec = i;
        }
        a.ptr [i+1]++;
    }
    for (size_t i = 0; i < nrow; i++)
        a.ptr [i+1] += a.ptr [i];
    return is;
}

```

Remarquons les tests en lectures, afin de vérifier que l'entrée est bien correcte et que les indices de ligne se suivent par ordre croissant. Le format de fichier utilisé ici est le standard d'échange dans ce domaine. Le site internet *matrix market* [8] gère un grand nombre de telles matrices dans ce format et en présente diverses représentations graphiques.

Un programme de test est donné par :

## matrix\_tst.cc

```

#include "matrix.h"
using namespace std;
int main() {
    matrix<double> a;
    cin >> a;
    cout << a;
}

```

L'entrée associée à la matrice creuse de l'exemple précédent,

```

a.mtx
%%MatrixMarket matrix coordinate real general
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

```

La compilation et le test sont donnés par :

```

make matrix_tst
./matrix_tst < a.mtx

```

Le programme re-écrit la matrice qu'il vient de lire : ceci ne nous apporte pas grand chose, sinon que notre code est correct.

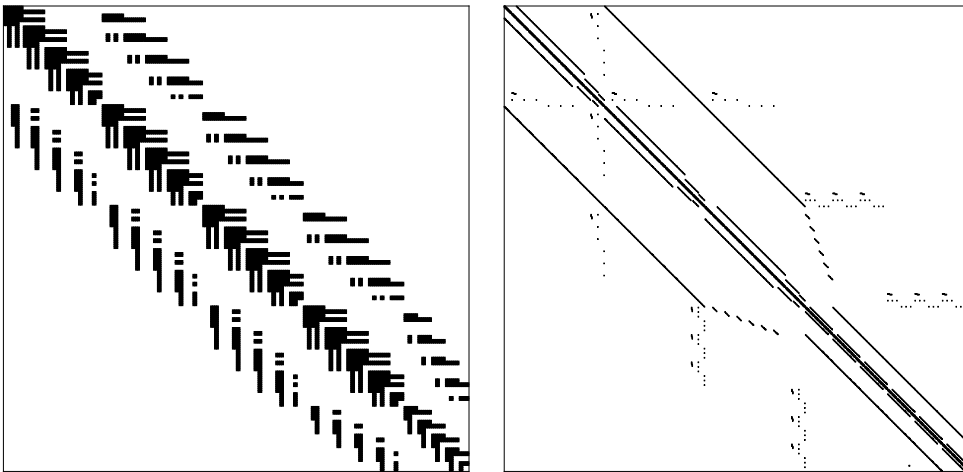


FIGURE 3.1 – Visualisation de matrices creuses de la collection [8] : à gauche `e05r0500`, d'un problème de mécanique des fluides ; à droite `pores_2`, d'un problème simulation de réservoirs.

Passons à quelque chose de plus intéressant à présent. Le code de visualisation `mtx2plot.cc` présenté en annexe A.1.2, dont la compilation et l'exécution s'écrivent :

```

make mtx2plot

```



```
./mtx2plot < a.mtx > a.plot
gnuplot a.plot
```

Des matrices issues de la collection [8] sont représentées sur la Fig. 3.1. Nous pouvons observer clairement les motifs de la structure creuse : les zones noires représentent les coefficients non-nuls et les zones blanches, ceux qui sont nuls. Nous aurons, au cours des chapitres suivants, l'occasion de construire nos propres matrices creuses et de les visualiser.

### 3.2.2 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 <typename T>
std::valarray<T>
matrix<T>::operator* (const std::valarray<T>& x) const {
    std::valarray<T> y (T(0), nrow());
    for (size_t i = 0; i < nrow(); i++)
        for (size_t p = ptr [i]; p < ptr [i+1]; p++)
            y [i] += val [p]*x [idx [p]];
    return y;
}
```

### 3.2.3 Application au gradient conjugué

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

matrix\_cg\_tst.cc

```
#include "matrix.h"
#include "valarray_util.h"
#include "cg.h"
#include "eye.h"
using namespace std;
int main () {
    matrix<double> a;
    cin >> a;
    valarray<double> x(1.0, a.nrow());
    valarray<double> b = a*x;
    valarray<double> xi(0.0, a.nrow());
    cg (a, xi, b, eye(), 10000, 1e-14);
    return (norm(xi-x) < 1e-7) ? 0 : 1;
}
```

as.mtx

```
%%MatrixMarket matrix coordinate real general
4 4 9
1 1 1
1 2 2
2 1 2
2 3 3
3 2 3
3 3 4
3 4 5
4 3 5
4 4 6
```

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 :

```
make matrix_cg_tst
./matrix_cg_tst < as.mtx
```

### 3.2.4 Accès aux coefficients

Nous allons écrire une fonction d'accès en lecture à un coefficient  $a_{ij}$  qui renvoie la valeur ou bien zéro si ce coefficient n'est pas représenté dans la structure creuse.

matrix.h (suite)

```
template<typename T>
const T& matrix<T>::operator() (size_t i, size_t j) const {
    for (size_t p = ptr[i]; p != ptr[i+1]; ++p)
        if (idx[p] == j) return val[p];
    return T(0);
}
```

Une variante est un accès en lecture-écriture : cette fonction nous sera très utile pour assembler les matrices creuses dans la méthode des éléments finis. Dans ce cas, le coefficient doit être déjà représenté par une entrée  $(i, j)$  dans la structure creuse, et nous renvoyons une référence sur la valeur.

matrix.h (suite)

```
template<typename T>
T& matrix<T>::entry (size_t i, size_t j) {
    for (size_t p = ptr[i]; p != ptr[i+1]; ++p)
        if (idx[p] == j) return val[p];
    static T zero = 0;
    assert(0); return zero;
}
```

Remarquons l'appel à `assert(0)` qui interrompt le programme : la valeur de retour n'est jamais utilisée, elle est déclarée uniquement par la nécessité dans le langage C++ de renvoyer une valeur.

### 3.2.5 Exercices

EXERCICE 15. (*Intégrité d'une matrice creuse*)

Écrire une fonction membre `a.check()` qui vérifie l'intégrité des données d'une matrice creuse `a`.

EXERCICE 16. (*Information sur une matrice*)

1) Écrire une fonction membre `a.frobenius_norm()` entrée une matrice creuse  $n \times m$ , de type `matrix` qui renvoie la norme de Frobenius d'une matrice creuse :

$$\|a\|_F = \left( \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |a_{i,j}|^2 \right)^{1/2}$$

2) Écrire une fonction `a.average_dist_diag()` qui renvoie la moyenne de la distance des éléments à la diagonale.

3) Une matrice est dite à *diagonale dominante* si

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$$

et strictement à diagonale dominante si l'inégalité précédente est stricte. Écrire une fonction `a.diagonal_dominant()` qui renvoie  $\max_i |a_{ii}| - \sum_{j \neq i} |a_{ij}|$ . La propriété sera déduite du signe de la quantité renvoyée.

EXERCICE 17. (*Multiplications par une matrice diagonale*)

1) Écrire une procédure de multiplication à gauche d'une matrice creuse par une matrice diagonale. Pour cela on implémentera l'opération dite *sur place*, qui place le résultat directement dans  $A$ , c'est-à-dire  $A \leftarrow DA$ , sous la forme :

```
matrix<T>& matrix<T>::inplace_left_mult (const valarray<T>& D);
```

où la matrice  $D$  diagonale est représentée par un tableau `valarray<T>` et qui renvoie le résultat  $A$  de la même façon que l'opérateur d'affectation `operator=` de la classe.

2) De même, écrire la fonction  $AD$  de multiplication *sur place* à droite d'une matrice creuse  $A$  par une matrice diagonale  $D$  sous la forme :

```
matrix<T>& matrix<T>::operator*= (const valarray<T>& D);
```

c'est-à-dire  $A \leftarrow AD$ .

3) Écrire un code qui, à partir des fonctions précédentes, calcule  $B = AD$  puis  $C = DA$  sans modifier  $A$ . Indication : penser à l'opérateur de copie ou d'affectation des matrices creuses.

EXERCICE 18. (*Extraction de la diagonale*)

Écrire la fonction `diag(A)` 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 19. (*Ajout d'une matrice diagonale*)

Écrire la fonction d'addition *sur place* d'une matrice diagonale  $A \leftarrow A + D$  sous la forme :

```
matrix<T>& matrix<T>::operator+= (const valarray<T>& D);
```

On suppose que la structure creuse de  $A$  représente déjà les coefficients diagonaux : si un de ceux-ci n'était pas représenté, la fonction émet une erreur.

EXERCICE 20. (*Produit transposé*)

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

EXERCICE 21. (*Extraction d'une partie triangulaire*)

Soit  $A$  une matrice creuse. Calculer `tril(A)` et `triu(A)`, les fonctions d'extractions des parties triangulaires inférieure et supérieure strictes, c'est-à-dire sans la diagonale. Indication : l'extraction s'effectuera en deux passages sur la structure creuse de  $A$  : lors du premier passage, on effectuera un comptage des éléments non-nuls de la partie triangulaire. Ensuite, après dimensionnement, lors du second passage, on effectuera la copie des coefficients concernés.

EXERCICE 22. (*Vecteurs creux*)

De façon analogue aux matrices creuses, nous pouvons définir des *vecteurs creux*. Ils représentent en quelque sorte la ligne ou la colonne d'une matrice creuse : seuls les éléments non-nuls seront représentés. Ce type de vecteur est particulièrement utile lorsque nous travaillons dans un espace de dimension  $n$  de grande taille (un million par exemple) et que seul un petit nombre, noté  $\text{nnz}(x)$ , ( $\text{nnz}(x) = 5$  ou  $10$  par exemple) des éléments du vecteur  $x$  sont non nuls.

$$\text{nnz}(x) = \text{card}\{i \in \{0, \dots, n-1\}; x_i \neq 0\}$$

Ce type de situation arrive fréquemment en calcul scientifique : météo, mécanique des fluides, etc. Dans ce cas, nous souhaitons que les calcul de  $z = x + y$  ou  $z = \lambda x$  prennent de l'ordre de 5 ou 10 opérations au lieu d'un millions d'additions d'éléments presque tous nuls... Pour cela nous introduisons la classe suivante :

```
template <class T>
class spvalarray {
public:
    spvalarray (size_t n = 0);
    // ...
protected:
    valarray<T>      val;
    valarray<size_t> idx;
    size_t           idxmax;
};
```

Dans la classe précédente, `idxmax` représente l'indice maximal  $n$ , qui n'est pas déductible des autres données de la classe. À titre d'exemple, considérons les vecteurs suivants, pour  $n = 6$  :

$$\begin{aligned} x &= (1, 2, 0, 2, 0, 0) \\ y &= (0, 3, 5, 7, 11, 0) \end{aligned}$$

- 1) Donner pour ces deux vecteurs la représentation creuse (**val,idx**) des valeurs non nulles et des indices associés.
- 2) Écrire le constructeur dont le profil est indiqué dans la classe précédente, qui prend en argument  $n$ , l'indice maximal, et dont tous les éléments sont nuls.
- 3) Écrire le constructeur de copie et l'opérateur d'affectation de cette classe.
- 4) Écrire un constructeur pour cette classe, qui prend en argument un vecteur dense de type **valarray**, et le compresse en ne retenant que ses éléments non-nuls. Inversement, la décompression d'un vecteur creux s'effectue via la conversion de type de **spvalarray** vers **valarray**. Elle est définie dans la classe **spvalarray** par la déclaration suivante :

```
template <class T>
class spvalarray {
public:
    operator std::valarray<T>() const;
    // ...
};
```

Écrire cette fonction de conversion en **valarray**.

- 5) Définir la fonction membre **size**, qui renvoie  $n$  et la fonction **nnz** qui renvoie le nombre d'éléments non nuls du vecteur.
- 6) Écrire une fonction amie **operator\*** qui permet de calculer  $\lambda x$ , où  $x$  est un vecteur creux et  $\lambda$  un scalaire, en exactement **nnz**( $x$ ) multiplications.
- 7) On suppose à présent, et pour le reste du problème, que les vecteurs creux sont *ordonnés par indices croissant*. Donner la fonction d'accès en lecture seule à un indice  $i$ , qui permet d'écrire **x[i]** et renvoie, pour un vecteur creux **x** la valeur à l'indice  $i$  ou bien zéro si  $i$  n'est pas représenté. Indication : Exploitant le fait que les indices sont triés, on donnera un algorithme dont le temps de calcul est  $\mathcal{O}(\log \text{nnz}(x))$ .
- 8) Donner, pour les deux vecteurs  $x$  et  $y$  de la question 1, la représentation creuse (**val,idx**) du vecteur  $z = x + y$ . Combien vaut **nnz**( $z$ ) ?
- 9) Écrire une fonction membre **add\_nnz** de la classe qui calcule le nombre d'éléments non-nuls de la somme  $x + y$  de deux vecteurs creux  $x$  et  $y$  en  $\mathcal{O}(\text{nnz}(x) + \text{nnz}(y))$  opérations. Note : prendre soin de vérifier que l'algorithme proposé donne un résultat correct pour les vecteurs  $x$  et  $y$  de la question 1.
- 10) Écrire une fonction **operator+** d'addition de deux vecteurs creux  $x$  et  $y$  en  $\mathcal{O}(\text{nnz}(x) + \text{nnz}(y))$  opérations. Note : vérifier également que le résultat est correct pour les vecteurs  $x$  et  $y$  de la question 1.
- 11) Proposer un format de fichier pour les vecteurs creux, en s'inspirant du format de fichier utilisé pour les matrices creuses. Écrire les fonctions d'entrée-sortie correspondantes.

#### EXERCICE 23. (Somme de deux matrices creuses)

On suppose ici que, dans chaque ligne  $i$ , les coefficients sont ordonnés par indices de colonne croissant. 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  $\Theta(\text{nnz}(A) + \text{nnz}(B))$ . Indication : une première boucle comptera le nombre d'éléments non-nuls de  $A + B$ , qui servira à dimensionner les tableaux. Une seconde boucle effectuera le calcul.

### 3.2.6 Notes

Willoughby a organisé en 1968, au centre de recherche d'IBM de Yorktown Heights, près de New York, la première rencontre sur le thème des matrices creuses : cette rencontre marque l'émergence de ce sujet comme un champs de recherche cohérent. Cependant, la plupart des techniques importantes étaient déjà en place avant cette date : dès 1963, Sato et Tinney présentent un format de stockage compressés par ligne [52].

La communauté des matrices creuses a, dès le début, diffusé de très bonnes bibliothèques. Dès 1977, à Yale, aux États-Unis, apparaît la première bibliothèque [19], puis en 1981 **sparsepak** [11], présentée en détails dans le livre de George et Liu [22], et qui aborde la résolution directe pour les matrices creuses. Ces bibliothèques sont développées en langage fortran, ainsi que, plus tard, la bibliothèque **sparskit** [40] de Saad, qui traite essentiellement des méthodes itératives. Le lecteur intéressé pourra également consulter le très bon ouvrage de Saad [41] dédié aux matrices creuses et aux algorithmes de résolution itératifs. Il existe aujourd'hui de nombreuses bibliothèques proposant des matrices creuses : citons la bibliothèque **petsc** [10], écrite en langage C, et, dernièrement, la bibliothèque **rheolef** [43], en langage C++ et exploitant les avantages de ce langage et propose des opérations avancées sur les matrices creuses telle le produit  $C=A*B$  de deux matrices ou la transposition d'une matrice creuse. La plupart de ces bibliothèques récentes intègrent la possibilité d'effectuer des calculs distribués et massivement parallèles.

La résolution directe en matrices creuses n'est pas abordé dans ce livre : son développement utilisant des algorithmes sophistiqués de la théorie des graphes, dépasserait le cadre de ce livre. Le lecteur pourra consulter une introduction très pédagogique introduction de Stewart [47]. La bibliothèque **umfpack** [16], écrite en langage C, est un des fleurons de l'approche dite *multifrontale* dans ce domaine. Une autre approche, dite *supernodale*, est très bien représentée dans la bibliothèque **superlu** [33].

Le format de compression par ligne **csc** présenté dans ce chapitre n'est pas le seul possible : le format **coo**, pour *coordinate*, liste les triplets  $(i, j, a_{i,j})$  non nuls et est également largement utilisé, notamment lors des entrées-sorties. L'ordonnancement par lignes du format **csc** a l'avantage de faciliter certains algorithmes où le balayage par ligne est important, comme le produit matrice-vecteur.

## 3.3 Maillages

### 3.3.1 Éléments, points et maillages

La résolution de problèmes multi-dimensionnels sur des géométries générales par la méthode des éléments finis utilise un maillage du domaine de calcul. Le maillage est un recouvrement d'un domaine géométrique  $\Omega \subset \mathbb{R}^d$ , où  $d = 1, 2, 3$ , par des éléments. L'union de ces éléments donne  $\Omega$  et ces éléments ne se touchent que par les bords, ils ne se recouvrent pas entre eux. Un maillage, noté  $\mathcal{T}_h$  est défini comme une liste d'éléments :

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

La structure de donnée représentant un maillage contiendra cette liste d'éléments. Les éléments seront des simplexes en dimension  $d$ , avec  $d = 1, 2$  ou  $3$ . Un simplexe en dimension deux est simplement un triangle, et en dimension trois, un tétraèdre. Ceci est représenté par la classe `element` :

element.h

```
#include <array>
template<size_t D>
class element : public std::array<size_t, D+1> {};
```

Notez le paramètre `D` de la classe, qui correspond à la dimension  $d$  de l'espace physique. Cette description extrêmement concise exprime qu'un élément est un tableau de  $d + 1$  indices de sommets. Ces indices de sommets font référence à une table des sommets, qui sera également dans la structure de données du maillage. La classe `std::array` définit un tableau de taille fixe, ici `D+1`. Cette classe, à la fois très simple et très utile, est définie dans le fichier d'en-tête `<array>` de la librairie standard et débute de la façon suivante :

<array>

```
template <typename T, size_t N>
class array {
public:
    array() : x() {}
    size_t size() const { return N; }
    T& operator[] (size_t i) { return x[i]; }
    const T& operator[] const (size_t i) { return x[i]; }
    // ...
protected:
    T x [N];
};
```

Ce fichier a été introduit dans le standard C++ lors de la révision de 2011 : il nécessite l'option `c++ -std=c++11` lorsqu'on utilise le compilateur GNU C++.

Afin de faciliter les fonctions d'entrée-sortie des maillages, nous définissons celles pour un élément :

## element.h (suite)

```

#include <iostream>
template <size_t D>
std::ostream& operator<< (std::ostream& os, const element<D>& K) {
    for (size_t i = 0; i < D+1; ++i)
        os << K[i]+1 << " ";
    return os;
}
template <size_t D>
std::istream& operator>> (std::istream& is, element<D>& K) {
    for (size_t i = 0; i < D+1; ++i) {
        is >> K[i]; K[i]--;
    }
    return is;
}

```

Un sommet de  $\mathbb{R}^d$  sera décrit par la classe point :

## point.h

```

#include <array>
template <typename T, size_t D>
class point : public std::array<T,D> {};

```

Notez que la classe point a deux paramètres : le type flottant T et la dimension physique de l'espace D. Cette classe exprime simplement qu'un point est le tableau de ses  $d$  coordonnées. De même que pour les éléments, nous définissons les fonctions d'entrée-sortie pour un sommet :

## point.h (suite)

```

#include <iostream>
template <typename T, size_t D>
std::istream& operator>> (std::istream& is, point<T,D>& a) {
    for (size_t i = 0; i < D; i++)
        is >> a[i];
    return is;
}
template <typename T, size_t D>
std::ostream& operator<< (std::ostream& os, const point<T,D>& a) {
    for (size_t i = 0; i < D; i++) {
        os << a[i];
        if (i != D-1) os << " ";
    }
    return os;
}

```

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



## mesh.h

```

#include "point.h"
#include "element.h"
#include <vector>
template <typename T, size_t D>
class mesh : public std::vector<element<D> > {
public:
    mesh ();
    mesh (const mesh<T,D>&);
    mesh<T,D>& operator= (const mesh<T,D>&);
    size_t n_vertex () const;
    size_t n_internal_vertex () const;
    bool is_boundary (size_t i) const;
    point<T,D>& vertex (size_t i);
    const point<T,D>& vertex (size_t i) const;
    template <typename U, size_t D1> friend
    std::istream& operator>> (std::istream&, mesh<U,D1>&);
    template <typename U, size_t D1> friend
    std::ostream& operator<< (std::ostream&, const mesh<U,D1>&);
protected:
    std::vector<point<T,D> > x;
    std::vector<size_t> b;
    size_t ni;
};

```

Remarquons l'utilisation de la classe `vector<T>` (voir [20, p. 53] ou [49, 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)` renvoie l'aire du  $j$ -ème triangle. Cette fonction sera décrite et utilisée plus loin (page 3.4.3).

Le constructeur par défaut, le constructeur de copie et opérateurs d'affectation s'écrivent simplement :

## mesh.h (suite)

```

template <typename T, size_t D>
mesh<T,D>::mesh ()
: std::vector<element<D> >(), x(), b(), ni(0) {}

template <typename T, size_t D>
mesh<T,D>::mesh (const mesh<T,D>& m2)
: std::vector<element<D> >(m2), x(m2.x), b(m2.b), ni(m2.ni) {}

template <typename T, size_t D>
mesh<T,D>& mesh<T,D>::operator= (const mesh<T,D>& m2) {
    std::vector<element<D> >::operator= (m2);
    x = m2.x;
    b = m2.b;
    ni = m2.ni;
    return *this;
}

```

Ensuite, viennent les différentes fonctions d'accès en lecture s'écrivent simplement :

mesh.h (suite)

```

template <typename T, size_t D>
size_t mesh<T,D>::n_vertex () const { return x.size(); }
template <typename T, size_t D>
size_t mesh<T,D>::n_internal_vertex () const { return ni; }
template <typename T, size_t D>
bool mesh<T,D>::is_boundary (size_t i) const { return b[i] != 0; }
template <typename T, size_t D>
point<T,D>& mesh<T,D>::vertex (size_t i) { return x[i]; }
template <typename T, size_t D>
const point<T,D>&
mesh<T,D>::vertex (size_t i) const { return x[i]; }

```

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

mesh.h (suite)

```

template <typename T, size_t D>
std::ostream& operator<< (std::ostream& os, const mesh<T,D>& m) {
    using namespace std;
    os << m.x.size() << endl;
    for (size_t i = 0, n = m.x.size(); i < n; i++)
        os << m.x[i] << " " << m.b[i] << endl;
    os << m.size() << endl;
    for (size_t e = 0; e < m.size(); e++)
        os << m[e] << " 1" << endl;
    return os;
}

```

Le format de fichier utilisé ici est très proche de celui utilisé par de nombreux générateurs de maillages, tel **bamg** [27]. Il n'existe malheureusement pas de conventions de format de fichier dans ce domaine. La fonction de lecture s'écrit :

mesh.h (suite)

```

template <typename T, size_t D>
std::istream& operator>> (std::istream& is, mesh<T,D>& m) {
    size_t nx;
    is >> nx;
    m.x.resize (nx);
    m.b.resize (nx);
    m.ni = 0;
    for (size_t i = 0; i < nx; i++) {
        is >> m.x[i] >> m.b[i];
        if (m.b[i] == 0) m.ni++;
    }
    size_t ne, domain;
    is >> ne;
    m.resize (ne);
    for (size_t e = 0; e < ne; e++)
        is >> m[e] >> domain;
    return is;
}

```

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

sont suivis de l'indicateur de sous domaine, qui ne sera pas utilisé par la suite.

### 3.3.2 Test des entrées-sorties

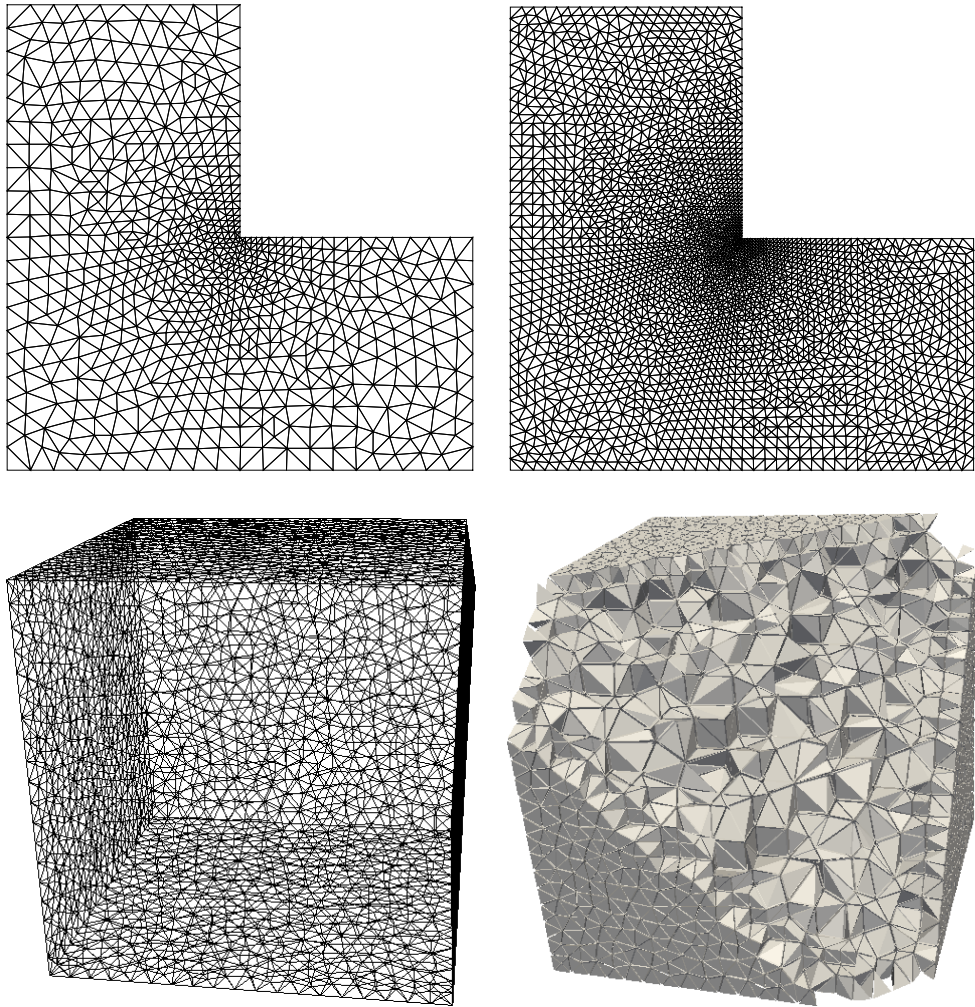


FIGURE 3.2 – Visualiation de maillages.

Voici à présent un petit programme de test des fonctions d'entrée et de sortie :

mesh\_tst.cc

```
#include "mesh.h"
using namespace std;
int main() {
    mesh<double,2> m;
    cin >> m;
    cout << m;
}
```

square.mesh

```
4
0 0      1
1 0      1
1 1      1
0 1      1
2
1 2 3    1
1 3 4    1
```

```
make mesh_tst
./mesh_tst < square.mesh
```

Les programmes `mesh` et `mesh3d` présentés en annexe [A.2.2](#) et [A.2.3](#) permettent de visualiser les maillages comme sur la Fig. [3.2](#).

### 3.3.3 Mesure des éléments

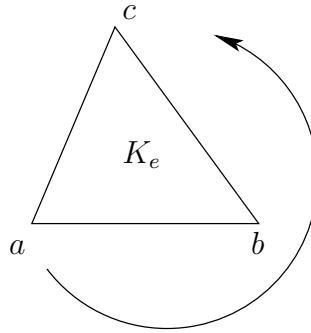
La fonction `meas(Th,1)` renvoie l'aire de l'élément d'indice  $e$  dans le maillage : il s'agit d'une fonction très utile dans de nombreuses applications. Nous aurons, au cours du chapitre suivant, l'occasion d'utiliser cette fonction dans le cadre de la méthode des éléments finis.

En dimension un, la mesure d'un élément est simplement la longueur d'une arête :

meas.h (suite)

```
template<typename T>
T meas (const mesh<T,1>& Th, size_t e) {
    const element<1>& K = Th[e];
    size_t ia = K[0], ib = K[1];
    const point<T,1>& a = Th.vertex(ia), b = Th.vertex(ib);
    return b[0]-a[0];
}
```

En dimension deux, la mesure d'un élément est l'aire d'un triangle :

FIGURE 3.3 – Rotation autour de  $K_e$  suivant  $(i, j, k)$  dans le sens positif.

meas.h (suite)

```
template<typename T>
T meas (const mesh<T,2>& Th, size_t e) {
    const element<2>& K = Th[e];
    size_t ia = K[0], ib = K[1], ic = K[2];
    const point<T,2>& a = Th.vertex(ia), b = Th.vertex(ib),
                    c = Th.vertex(ic);
    return ((b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0]))/2;
}
```

Soient  $a$ ,  $b$  et  $c$  les sommets du triangle  $K_e$ . Quitte à permuter l'ordre dans lequel sont donné ces sommets, nous supposons que  $(a, b, c)$  est orienté dans le sens positif sur  $\partial K_e$  (voir Fig. 3.3) : autrement dit, l'aire définie par cette fonction est positive. C'est d'ailleurs ainsi que, par convention, les éléments sont décrits dans les fichiers de maillage.

Enfin, en dimension trois, la mesure d'un élément est le volume d'un tétraèdre :

meas.h (suite)

```
template<typename T>
T meas (const mesh<T,3>& Th, size_t e) {
    const element<3>& K = Th[e];
    size_t ia = K[0], ib = K[1], ic = K[2], id = K[3];
    const point<T,3>& a = Th.vertex(ia), b = Th.vertex(ib),
                    c = Th.vertex(ic), d = Th.vertex(id);
    return dot(b-a, wedge(c-a, d-a))/6;
}
```

De même, nous supposons que l'ordre des sommets du tétraèdre est tel que le volume ainsi calculé est positif. Rappelons que le volume d'un tétraèdre de sommets  $(a, b, c, d)$  est un sixième du produit mixte des trois vecteurs directeurs  $\vec{ab}$ ,  $\vec{ac}$  et  $\vec{ad}$ , soit encore  $\vec{ab} \cdot (\vec{ac} \wedge \vec{ad}) / 6$ . Remarquez ici l'utilisation de la spécialisation partielle de la fonction `meas` : à pour chaque dimension  $d$ , représenté par le paramètre `D` du modèle de la fonction, correspond une implémentation différente. La fonction reste générique en ce qui concerne le type `T`, représentant la virgule flottante. La spécialisation partielle est une technique très utile pour écrire des algorithmes

génériques lorsqu'il s'agit de traiter les cas particuliers.

Afin de faciliter la lecture et la concision du code, l'implémentation précédente utilise des utilitaires suivants de la classe `point` :

`point.h` (suite)

```
template <typename T, size_t D>
point<T,D> operator+ (const point<T,D>& u, const point<T,D>& v) {
    point<T,D> w;
    for (size_t i = 0; i < D; ++i) w[i] = u[i] + v[i];
    return w;
}
template <typename T, size_t D>
point<T,D> operator- (const point<T,D>& u, const point<T,D>& v) {
    point<T,D> w;
    for (size_t i = 0; i < D; ++i) w[i] = u[i] - v[i];
    return w;
}
```

La fonction `dot` représente le produit scalaire dans  $\mathbb{R}^d$  et nous lui associons la norme et la distance.

`point.h` (suite)

```
#include <cmath>
template <typename T, size_t D>
T dot (const point<T,D>& u, const point<T,D>& v) {
    T sum = 0;
    for (size_t i = 0; i < D; ++i)
        sum += u[i]*v[i];
    return sum;
}
template <typename T, size_t D>
T norm (const point<T,D>& u) {
    return std::sqrt(dot(u,u));
}
template <typename T, size_t D>
T dist (const point<T,D>& a, const point<T,D>& b) {
    return norm(b-a);
}
```

Enfin, la fonction `wedge` le produit vectoriel de deux vecteurs.

`point.h` (suite)

```
template <typename T>
point<T,3> wedge (const point<T,3>& u, const point<T,3>& v) {
    point<T,3> w;
    w[0] = u[1]*v[2] - u[2]*v[1];
    w[1] = u[2]*v[0] - u[0]*v[2];
    w[2] = u[0]*v[1] - u[1]*v[0];
    return w;
}
```

### 3.3.4 Exercices

EXERCICE 24. (*Mesure d'un domaine maillé*)

Écrire une fonction membre `meas( $\mathcal{T}_h$ )` qui renvoie l'aire `meas( $\Omega$ )` du domaine  $\Omega$  associé au maillage.

EXERCICE 25. (*Longueurs extrêmes des arêtes*)

Par convention dans les méthodes d'éléments finis, la longueur de la plus grande arête d'un maillage est notée simplement  $h$ .

1) Dans le cas de la dimension  $d = 2$ , écrire la fonction

```
T hmax (const mesh<T,2>& Th, size_t e);
```

et qui calcule la longueur de la plus grande arête d'un élément du maillage, donné par son indice  $e$ . Indiquez également comment traiter les cas  $d = 1$  et  $d = 3$ .

2) Écrire la fonction suivante, et qui calcule  $h$  :

```
template <typename T, size_t D>
T hmax (const mesh<T,D>& Th);
```

EXERCICE 26. (*Qualité d'un maillage*)

Sauf mention contraire, nous supposons  $d = 2$  dans cet exercice. Considérons un triangle  $K$  de sommets  $(a, b, c)$ . Rappelons que diamètre  $r_K$  du cercle passant par les trois sommets de  $K$ , ainsi que le diamètre  $\rho_K$  du plus grand cercle inscrit à l'intérieur du triangle  $K$  sont données par les formules :

$$r_K = \frac{ab \, bc \, ca}{4 \, \text{meas}(K)} \quad \text{et} \quad \rho_K = \frac{2 \, \text{meas}(K)}{ab + bc + ca}.$$

où, par commodité,  $ab$  désigne simplement la distance entre deux points  $a$  et  $b$ .

1) Vérifier que le ratio  $Q_K = \rho_K/r_K$  est invariant par translation, rotation ou homothétie (dilatation) de l'élément.

2) Calculer  $Q_{K_0} = \rho_{K_0}/r_{K_0}$  lorsque  $K_0$  est un triangle équilatéral.

3) Dans le contexte de la méthode des éléments finis, la qualité d'un triangle est donnée par  $q_K = Q_K/Q_{K_0}$ . Cette quantité est entre zéro et un : elle est normalisée. Elle vaut un quand le triangle est isocèle et décroît d'autant plus que le triangle est aplati. Écrire la fonction `quality(Th,e)`, qui calcule  $q_K$  pour un élément  $K$  d'indice  $e$ .

4) La qualité d'un maillage est alors définie par  $q(\mathcal{T}_h) = \min\{q_K; K \in \mathcal{T}_h\}$ . Écrire une fonction `quality(Th)` qui calcule la qualité d'un maillage.

5) Écrire une fonction membre `histogram(Th)` qui synthétise un histogramme de qualité, c'est-à-dire le pourcentage de ses éléments qui sont dans les tranches de qualité  $[0, 0.1]$ ,  $]0.1, 0.2]$ ,  $\dots$ ,  $]0.9, 1]$ . Cet histogramme pourra facilement être visualisé, comme sur la Fig. 3.4.

6) La notion de qualité de maillage s'étend-elle à la dimension ? À la dimension trois ? Le cas échéant, décrire comment étendre la fonction `quality(Th,e)`.

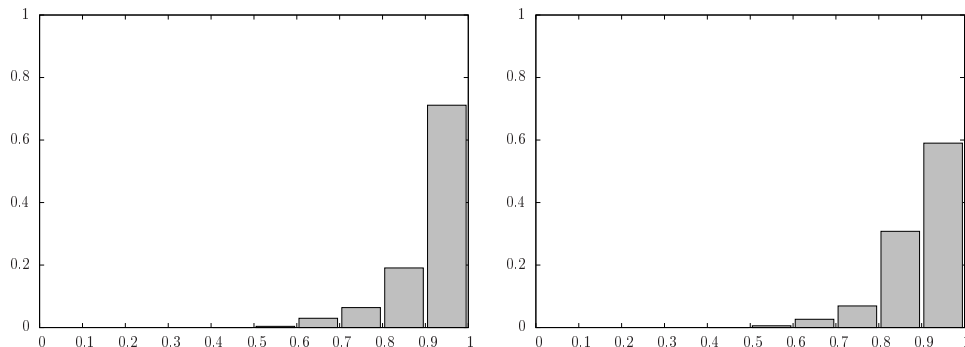


FIGURE 3.4 – Histogramme de qualité pour les maillages `square-40.mesh` et `L-40-adapt.mesh` de la Fig. 3.2.b, page. 3.2.

### 3.3.5 Notes

La notion de maillage, qui mêle des notions de géométrie algorithmique et des notions de théorie des graphes, a généré un très grand nombre de travaux algorithmiques. L'ouvrage [7] présente un éventail très large de ces travaux et la librairie `cgal` [1], écrite en langage C++ en propose une implémentation dans l'esprit de la librairie STL.

Le calcul par éléments finis, que nous allons aborder dans le chapitre suivant, impose à la génération de maillages certaines contraintes supplémentaires, telles qu'une certaine régularité dans la taille des éléments et le respect des frontières du domaine géométrique. Il existe un grand nombre de tels générateurs : les plus couramment utilisés sont `bamg` [27] pour les domaines bidimensionnels et `gmsh` [38] pour le cas tridimensionnel. Le livre de Georges [23] présente ces algorithmes ainsi que des applications à la méthode des éléments finis.

## 3.4 Méthode des éléments finis

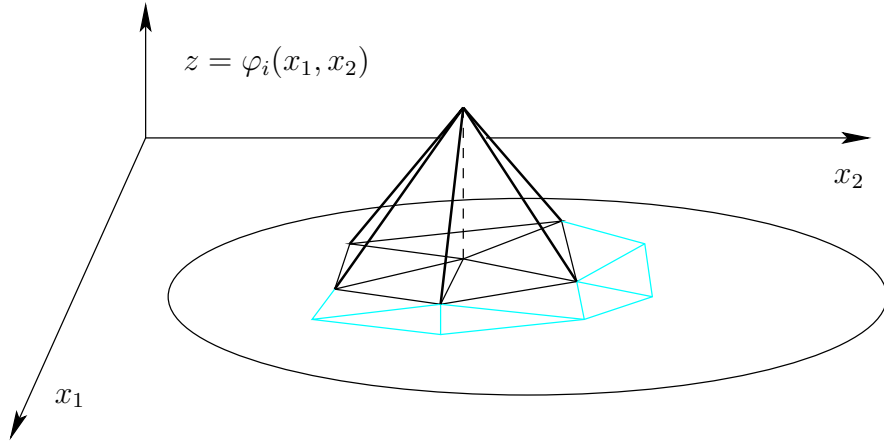
Le but de ce chapitre est de présenter une introduction à la programmation de la méthode des éléments finis en C++.

### 3.4.1 Une interpolation par éléments

Soit  $\Omega$  un ouvert borné de  $\mathbb{R}^d$ , où  $d = 1, 2$  ou  $3$ . Reprenons le problème de Poisson avec conditions aux limites homogènes, déjà abordé page 2.3.1 :

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



FIGURE 3.5 – Fonction de base  $\varphi_i$ .

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

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  $K$  d'un maillage donné. Le maillage est noté  $\mathcal{T}_h$ .

Nous considérons alors le problème suivant :

trouver  $u_h \in V_h$  telle que

$$\int_{\Omega} \nabla u_h(x, y) \cdot \nabla v_h(x, y) \, dx \, dy = \int_{\Omega} f(x, y) v_h(x, y) \, dx \, dy, \quad \forall v_h \in V_h \quad (3.4)$$

où  $V_h = X_h \cap H_0^1(\Omega)$  représente l'ensemble des fonctions continues de  $\Omega$  dans  $\mathbb{R}$  s'annulant sur la frontière  $\partial\Omega$  et affines dans chaque élément  $K$  du maillage.

L'espace  $X_h$  est de dimension finie. Soit  $n_v = \dim X_h$  : la dimension de  $X_h$  est égale au nombre de sommets de la triangulation  $\mathcal{T}_h$ . Nous pouvons choisir pour base de  $X_h$  les fonctions  $(\varphi_i)_{0 \leq i \leq n_v-1}$  valant 1 au  $i$ -ème sommet  $x_i$  de  $\mathcal{T}_h$  et zéro aux autres sommets (voir Fig. 3.5). Nous pouvons décomposer  $u_h$  sur cette base :

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

où  $u_i = u_h(x_i)$  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 <typename T, size_t D>
std::valarray<T>
interpolate (const mesh<T,D>& Th, T (*v)(const point<T,D>&)) {
    std::valarray<T> vh (Th.n_vertex());
    for (size_t i = 0; i < Th.n_vertex(); i++)
        vh[i] = v(Th.vertex (i));
    return vh;
}
```

En choisissant  $v_h = \varphi_i$  dans (3.4), le problème se ramène à

*trouver  $(u_i)_{0 \leq i \leq n_v - 1} \in \mathbb{R}^{n_v}$  telle que*

$$\sum_{j=0}^{n_v-1} \left( \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, dx \right) u_j = \sum_{j=0}^{n_v-1} \left( \int_{\Omega} \varphi_i \varphi_j \, dx \right) f(x_j, y_j), \quad \forall j \in \text{int}(\mathcal{T}_h), \quad (3.5)$$

$$u_j = 0, \quad \forall j \in \{0, \dots, n_v - 1\} \setminus \text{int}(\mathcal{T}_h). \quad (3.6)$$

où  $\text{int}(\mathcal{T}_h)$  désigne l'ensemble des indices des sommets de  $\mathcal{T}_h$  appartenant à l'intérieur du domaine  $\Omega$ . Son complémentaire noté  $\{0, \dots, n_v - 1\} \setminus \text{int}(\mathcal{T}_h)$  représente l'ensemble des sommets du maillage qui sont situés sur la frontière du domaine de calcul. Afin de bien imposer la condition aux bords  $u_h = 0$  sur  $\partial\Omega$ , exprimée par (3.6), nous allons séparer les degrés de liberté en deux catégories : (i) ceux associés à un sommet du maillage interne à  $\Omega$ , et qui sont inconnus ; (ii) ceux associés à un sommet sur la frontière de  $\Omega$ , et qui sont imposés à zéro. L'algorithme de renumérotation est :

```
renumbering.h
```

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

Dans la suite nous noterons simplement  $n$  le nombre de sommets internes du maillage :  $n$  est également la taille du système linéaire réduit, obtenu après élimination des degrés de liberté imposés sur la frontière.

### 3.4.2 Les matrices $A$ et $M$

Introduisons les matrices  $A$  et  $M$  dont les coefficients apparaissent dans (3.5) :

$$A_{i,j} = \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, dx, \quad \text{et} \quad M_{i,j} = \int_{\Omega} \varphi_i \varphi_j \, dx,$$

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

$$A = \sum_{e=0}^{n_e-1} A^{(e)} \quad \text{et} \quad M = \sum_{e=0}^{n_e-1} M^{(e)}$$

où  $A^{(e)}$  et  $M^{(e)}$  sont les matrices élémentaires relatives à l'élément  $K_e \in \mathcal{T}_h$ ,  $0 \leq e \leq n_e - 1$ , et  $n_e$  est le nombre d'éléments du maillage. Les matrices élémentaires sont données par :

$$A_{i,j}^{(e)} = \int_{K_e} \nabla \varphi_i \cdot \nabla \varphi_j \, dx, \quad \text{et} \quad M_{i,j}^{(e)} = \int_{K_e} \varphi_i \cdot \varphi_j \, dx,$$

L'intérêt de ces matrices élémentaires est que leurs coefficients sont relativement faciles à calculer : en effet, les fonctions de base  $\varphi_i$  sont polynomiales de degré un à l'intérieur de chaque élément  $K_e$ . Les matrices élémentaires  $A^{(e)}$  et  $M^{(e)}$  ne font intervenir que les  $d+1$  sommets du triangle  $K_e$  : elles n'ont donc au plus que  $(d+1)^2$  éléments non-nuls. Les matrices  $A$  et  $M$ , de taille  $n_v \times n_v$  ont donc au plus  $(d+1)^2 n_e$  éléments non-nuls. Or, pour un maillage classique,  $n_e = \Theta(n_v)$ , et ainsi  $A$  et  $M$  sont très creuses. Par exemple, en dimension  $d = 2$ , ces matrices n'ont au plus que  $9n_e$  éléments non-nuls, qui est très petit devant  $n_v^2$ .

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

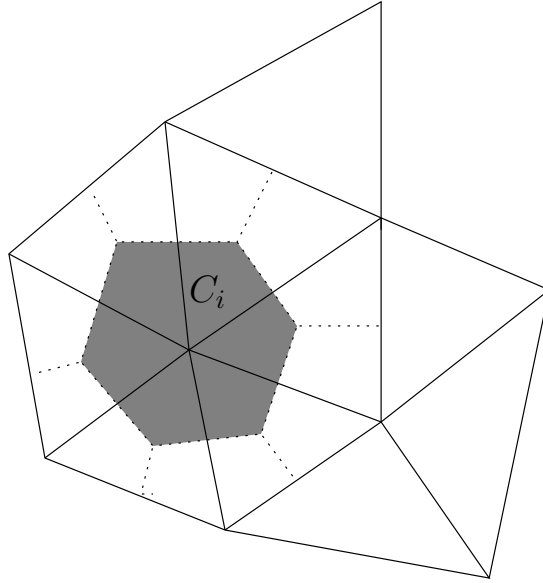
Utilisons la formule des trapèzes pour évaluer  $M_{i,j}^{(e)}$  :

$$M_{i,j}^{(e)} = \frac{\text{meas}(K_e)}{d+1} \sum_{k \in \text{sommet}(K_e)} \varphi_i(x_k) \varphi_j(x_k)$$

Si  $i \neq j$  ou bien, si  $i$  ou  $j$  ne sont pas des indices de sommets de  $K_e$ , alors  $M_{i,j}^{(e)} = 0$ . Autrement, si  $i = j \in \text{sommet}(K_e)$ , alors  $M_{i,i}^{(e)} = \text{meas}(K_e)/(d+1)$ . Ainsi, en utilisant la formule des trapèzes,  $M^{(e)}$  et  $M$  sont des matrices diagonales, et :

$$M_{i,i} = \sum_{e/i \in \text{sommet}(K_e)} \frac{\text{meas}(K_e)}{d+1}$$

La somme précédente s'effectue sur tous les indices  $e$ ,  $0 \leq e \leq n_e - 1$  pour lesquels l'élément  $K_e$  contient le  $i$ -ième sommet. 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. 3.6). Un algorithme pour assembler  $M$  est :

FIGURE 3.6 – Volume fini  $C_i$  centré autour du sommet  $i$ .

assembly\_mass.h

```

template <typename T, size_t D>
void assembly_mass (const mesh<T,D>& Th,
    const std::valarray<size_t>& num, std::valarray<T>& M) {
    M.resize (Th.n_internal_vertex());
    M = 0;
    for (size_t e = 0; e < Th.size(); e++) {
        T c = meas(Th,e)/(D+1);
        for (size_t r = 0; r < D+1; r++) {
            size_t i = Th[e][r];
            if (! Th.is_boundary(i)) M[num[i]] += c;
        }
    }
}

```

Rappelons que la fonction `meas(Th,1)` renvoie l'aire de l'élément d'indice  $e$  dans le maillage.

### 3.4.4 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 et le coefficient général de la matrice élémentaire s'écrit encore :

$$A_{i,j}^{(e)} = \nabla \varphi_i \cdot \nabla \varphi_j \text{ meas}(K_e)$$

Ce coefficient ne peut être non-nul que si  $i$  et  $j$  sont les indices de sommets appartenant à  $K_e$  : ces sommets sont alors soit égaux, soit adjacents par une arête de  $K$ . Ainsi

$A_{i,j} \neq 0$  dès que  $i$  et  $j$  sont des sommets d'un même élément.

Plaçons nous pour commencer dans le cas de la dimension un. Afin d'alléger les notations, posons  $a = x_i$  et  $b = x_j$ ,  $j \neq i$ . Sans perte de généralité, nous pouvons supposer que  $a < b$ , si bien que  $K_e = [a, b]$ . La fonction de base  $\varphi_i$  est caractérisée sur  $K_e$  comme étant l'unique fonction affine vérifiant  $\varphi_i(a) = 1$  et  $\varphi_i(b) = 0$ . Un rapide calcul donne les restrictions de  $\varphi_i$  et  $\varphi_j$  sur  $K_e$  :

$$\varphi_{i|K_e}(x) = \frac{b-x}{b-a} \quad \text{et} \quad \varphi_{j|K_e}(x) = \frac{x-a}{b-a} \quad \text{pour } j \neq i$$

si bien que

$$\varphi'_{i|K_e}(x) = -\frac{1}{\text{meas}(K_e)} \quad \text{et} \quad \varphi'_{j|K_e}(x) = \frac{1}{\text{meas}(K_e)} \quad \text{pour } j \neq i$$

Nous obtenons finalement

$$A_{i,i}^e = \frac{1}{\text{meas}(K_e)} \quad \text{et} \quad A_{i,j}^e = -\frac{1}{\text{meas}(K_e)} \quad \text{pour } j \neq i$$

En pré-multipliant le résultat par  $\text{meas}(K_e)$ , nous obtenons le coefficient de la matrice élémentaire :

coeff.h

```
template <typename T>
T coeff (const mesh<T,1>& Th, size_t e, size_t ie, size_t je) {
    return (ie == je) ? 1 : -1;
}
```

Considérons à présent le cas de la dimension deux. Notons  $a$  le sommet d'indice  $i$  et  $b$  et  $c$  les deux autres sommets de  $K_e$  tels que le trièdre  $(a, b, c)$  soit orienté dans le sens positif. La fonction de base  $\varphi_i$  est caractérisée sur  $K_e$  comme étant l'unique fonction affine vérifiant  $\varphi_i(a) = 1$  et  $\varphi_i(b) = \varphi_i(c) = 0$ . Un rapide calcul donne la restriction à  $K_e$  de  $\varphi_i$  :

$$\varphi_{i|K_e}(x) = \frac{(b_0 - x_0)(c_1 - b_1) - (c_0 - b_0)(b_1 - x_1)}{2 \text{meas}(K_e)}$$

si bien que

$$\frac{\partial \varphi_{i|K_e}}{\partial x_0} = -\frac{c_1 - b_1}{2 \text{meas}(K_e)} \quad \text{et} \quad \frac{\partial \varphi_{i|K_e}}{\partial x_1} = \frac{c_0 - b_0}{2 \text{meas}(K_e)}$$

Pour  $i = j$  nous obtenons :

$$A_{i,i}^{(e)} = \frac{(c_0 - b_0)^2 + (c_1 - b_1)^2}{4 \text{meas}(K_e)}$$

Pour  $i \neq j$ , notons  $a'$  le sommet d'indice  $j$  et  $b'$  et  $c'$  les deux autres sommets de  $K_e$  tels que le trièdre  $(a', b', c')$  soit orienté dans le sens positif. Nous obtenons :

$$A_{i,j}^{(e)} = \frac{(c_0 - b_0)(c'_0 - b'_0) + (c_1 - b_1)(c'_1 - b'_1)}{4 \text{meas}(K_e)}$$

En pré-multipliant le résultat par  $\text{meas}(K_e)$ , nous obtenons pour la dimension  $d = 2$  le coefficient de la matrice élémentaire :

`coeff.h` (suite)

```
#include "meas.h"
template <typename T>
T coeff (const mesh<T,2>& Th, size_t e, size_t ie, size_t je) {
    size_t ia1 = Th[e][ie], ia2 = Th[e][je],
           ib1 = Th[e][((ie+1)%3)], ib2 = Th[e][((je+1)%3)],
           ic1 = Th[e][((ie+2)%3)], ic2 = Th[e][((je+2)%3)];
    const point<T,2>& a1 = Th.vertex(ia1), a2 = Th.vertex(ia2),
                     b1 = Th.vertex(ib1), b2 = Th.vertex(ib2),
                     c1 = Th.vertex(ic1), c2 = Th.vertex(ic2);
    return ((c1[0]-b1[0])*(c2[0]-b2[0])
           + (c1[1]-b1[1])*(c2[1]-b2[1]))/4;
}
```

Les paramètres `ie` et `je` représentent les indices locaux, entre 0 et 2, des deux sommets du triangle à considérer.

Considérons enfin le cas de la dimension trois. Notons  $a$  le sommet d'indice  $i$  et  $b, c$  et  $d$  les trois autres sommets du tétraèdre  $K_e$  tels que  $(a, b, c, d)$  soit orienté dans le sens positif. La fonction de base  $\varphi_i$  est caractérisée sur  $K_e$  comme étant l'unique fonction affine vérifiant  $\varphi_i(a) = 1$  et  $\varphi_i(b) = \varphi_i(c) = \varphi_i(d) = 0$ . Recherchons la restriction à  $K_e$  de  $\varphi_i$  sous la forme :

$$\varphi_{i|K_e}(x) = \frac{\gamma \cdot \vec{ax}}{6 \text{ meas}(K_e)} + 1$$

avec  $\gamma \in \mathbb{R}^3$ . Un rapide calcul montre que  $\gamma = -\vec{ab} \wedge \vec{ac} - \vec{ac} \wedge \vec{ad} - \vec{ad} \wedge \vec{ab}$ . Rappelons l'identité  $\vec{ab} \cdot (\vec{ac} \wedge \vec{ad}) = 6 \text{ meas}(K_e)$ , déjà utilisée au paragraphe précédent. Nous pouvons vérifier que  $\gamma \cdot \vec{ab} = \gamma \cdot \vec{ac} = \gamma \cdot \vec{ad} = -6 \text{ meas}(K_e)$ . Remarquons l'expression équivalente  $\gamma = \vec{ab} \wedge \vec{cd} + \vec{ad} \wedge \vec{ac}$ , qui a l'avantage de faire intervenir un plus petit nombre d'opérations arithmétiques. Finalement

$$\nabla \varphi_{i|K_e}(x) = \frac{\gamma}{6 \text{ meas}(K_e)}$$

Pour  $i = j$  nous obtenons :

$$A_{i,i}^{(e)} = \frac{|\gamma|^2}{36 \text{ meas}(K_e)}$$

Pour  $j \neq i$ , notons  $a'$  le sommet d'indice  $j$ , et  $b', c'$  et  $d'$  les sommets suivants de  $K_e$ . Notons également  $\gamma'$  le vecteur obtenu en remplaçant  $(a, b, c, d)$  par  $(a', b', c', d')$  dans l'expression précédente de  $\gamma$ . Nous obtenons

$$A_{i,j}^{(e)} = \frac{\gamma \cdot \gamma'}{36 \text{ meas}(K_e)}$$

coeff.h (suite)

```

template <typename T>
T coeff (const mesh<T,3>& Th, size_t e, size_t ie, size_t je) {
    size_t ia1 = Th[e][ie], ia2 = Th[e][je],
    ib1 = Th[e][(ie+1)%4], ib2 = Th[e][(je+1)%4],
    ic1 = Th[e][(ie+2)%4], ic2 = Th[e][(je+2)%4],
    id1 = Th[e][(ie+3)%4], id2 = Th[e][(je+3)%4];
    const point<T,3>& a1 = Th.vertex(ia1), a2 = Th.vertex(ia2),
    b1 = Th.vertex(ib1), b2 = Th.vertex(ib2),
    c1 = Th.vertex(ic1), c2 = Th.vertex(ic2),
    d1 = Th.vertex(id1), d2 = Th.vertex(id2);
    point<T,3> g1 = wedge(b1-a1,d1-c1) + wedge(d1-a1,c1-a1),
    g2 = wedge(b2-a2,d2-c2) + wedge(d2-a2,c2-a2);
    T sgn = (((ie - je) % 2) == 0) ? 1 : -1;
    return sgn*dot(g1,g2)/36;
}

```

Notez la règle de signe : le signe est inversé lorsque  $i$  et  $j$  sont distant d'un nombre impair dans l'ordre local des sommets de  $K_e$ .

### 3.4.5 Assemblage dynamique avec la classe `list<T>`

La matrice  $A$  n'est donc pas diagonale. De plus, le nombre d'arêtes 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. Si le nombre de sommets internes du maillage nous donne la taille `nrow` et `ncol` de la matrice, nous ne connaissons pas *a priori* le nombre d'éléments non nuls `nnz` de la matrice.

L'idée est d'effectuer l'assemblage en deux passes. La première passe parcourt le maillage et range chaque indice de colonne  $j$  dans une *liste* associée à l'indice de ligne. Pour cela, nous allons utiliser la classe `list` de la librairie standard du C++. Elle présente un interface du type :

&lt;list&gt;

```

template <typename T>
class list {
    typedef iterator;
    typedef const_iterator;
    list();
    list(const list<T>&);
    list<T>& operator= (const list<T>&);
    ~list();
    iterator begin () const;
    iterator end () const;
    void insert (const_iterator iter, const T&);
    // ...
};

```

Il existe des itérateurs qui parcourent la liste : `iterator`, en lecture-écriture, et `const_iterator`, en lecture seule. Un parcours d'une liste `L` s'écrit :

```

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

```

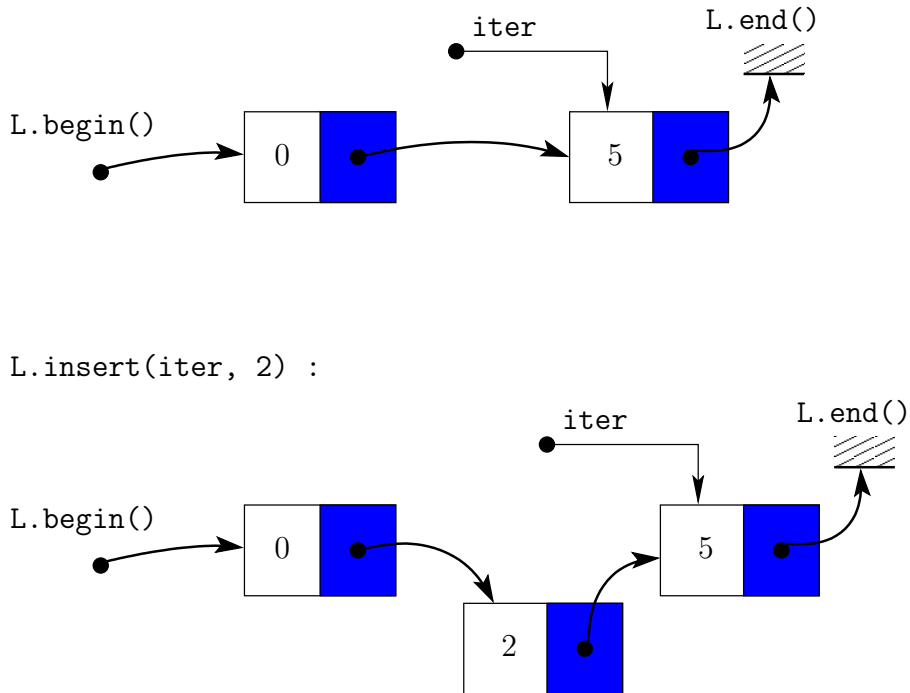


FIGURE 3.7 – Liste chaînée pour l’assemblage des matrices : opération d’insertion.

Le lecteur familier avec le langage C remarquera que les itérateurs reprennent la syntaxe des pointeurs dont ils sont la généralisation en C++. L’opération `L.insert(iter, j)` nous permettra d’ajouter un indice de colonne `j` dans la liste `L`, à la position indiquée par l’itérateur `iter`, comme représenté sur la Fig. 3.7. Supposons à présent la liste triée : voici une fonction qui insère un élément dans la liste s’il ne s’y trouvait pas auparavant :

#### unique\_insert.h

```
template <typename T>
void unique_insert (std::list<T>& L, const T& j) {
    typename std::list<T>::iterator iter = L.begin(), last = L.end();
    while (iter != last && *iter < j) ++iter;
    if (iter == last || *iter != j) L.insert(iter, j);
}
```

La classe `list` présente encore bien d’autres fonctionnalités qui ne sont pas présentées exhaustivement ici : notre propos est d’assembler la matrice.

Revenons donc à notre matrice creuse en construction : pour une ligne  $i$ , la liste, notée  $s_i$ , contient les indices de colonne  $j$  pour lesquelles il y a un coefficient dans la matrice.



## pass1.h

```

#include "unique_insert.h"
template <typename T, size_t D>
void pass1 (const mesh<T,D>& Th,
            const std::valarray<size_t>& num, matrix<T>& a)
{
    size_t n = Th.n_internal_vertex();
    std::vector<std::list<size_t> > s(n);
    for (size_t e = 0; e < Th.size(); e++)
        for (size_t ie = 0; ie < D+1; ie++) {
            size_t i = Th[e][ie];
            if (Th.is_boundary(i)) continue;
            unique_insert (s[num[i]], num[i]);
            for (size_t je = 0; je < D+1; je++) {
                size_t j = Th[e][je];
                if (je == ie || Th.is_boundary(j)) continue;
                unique_insert (s[num[i]], num[j]);
            }
        }
    a.resize (s, n);
}

```

Ce tableau de liste est ensuite converti en une structure de donnée `matrix` : ceci est effectué par la fonction membre `resize(s,n)` de la classe `matrix`, qui réorganise la matrice `a` à partir de la structure creuse donnée sous la forme de listes :

## matrix.h (suite)

```

template<typename T>
void matrix<T>::resize (const std::vector<std::list<size_t> >& s,
                        size_t ncol)
{
    using namespace std;
    size_t nrow = s.size();
    ptr.resize (nrow+1);
    ptr[0] = 0;
    for (size_t i = 0; i < nrow; ++i)
        ptr[i+1] = ptr[i] + s[i].size();
    size_t nnz = ptr[s.size()];
    idx.resize (nnz);
    for (size_t i = 0, p = 0; i < nrow; ++i)
        for (list<size_t>::const_iterator
             iter = s[i].begin(),
             last = s[i].end(); iter != last; ++iter, ++p)
            idx[p] = *iter;
    val.resize (nnz);
    val = 0;
    idxmax = ncol;
}

```

La deuxième passe parcourt le maillage et ajoute dans la matrice en construction la contribution de chaque élément. Nous pouvons à présent écrire l'algorithme d'assemblage de  $A$  :

## pass2.h

```
#include "coeff.h"
template <typename T, size_t D>
void pass2 (const mesh<T,D>& Th, const std::valarray<size_t>& num,
           matrix<T>& a) {
    for (size_t e = 0; e < Th.size(); e++) {
        T c = 1/meas(Th,e);
        for (size_t ie = 0; ie < D+1; ie++) {
            size_t i = Th[e][ie];
            if (Th.is_boundary(i)) continue;
            a.entry (num[i],num[i]) += c*coeff (Th, e, ie, ie);
            for (size_t je = 0; je < D+1; je++) {
                size_t j = Th[e][je];
                if (je == ie || Th.is_boundary(j)) continue;
                a.entry (num[i],num[j]) += c*coeff (Th, e, ie, je);
            }
        }
    }
}
```

La fonction finale enchaînera simplement les deux passes :

## assembly\_energy.h

```
#include "pass1.h"
#include "pass2.h"
template <typename T, size_t D>
void assembly_energy (const mesh<T,D>& Th,
                     const std::valarray<size_t>& num, matrix<T>& a)
{
    pass1 (Th, num, a);
    pass2 (Th, num, a);
}
```

Un programme de test est :

## assembly\_energy\_tst.cc

```
#include <valarray>
#include "mesh.h"
#include "matrix.h"
#include "renumbering.h"
#include "assembly_energy.h"
using namespace std;
int main() {
    mesh<double,2> 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 :

```
make assembly_energy_tst
./assembly_energy_tst < L-20-adapt.mesh > L-20-adapt.mtx
```

Le maillage ‘L-20-adapt.mesh’ a été obtenu avec le logiciel **bamg** [27]. La visualisation de la matrice creuse Fig. 3.8.b est obtenue à partir du programme **mtx2plot** présenté dans l’annexe A.1.2 et du logiciel **gnuplot** :

```
./mtx2plot < L-20-adapt.mtx > L-20-adapt.plot
gnuplot L-20-adapt.plot
```

où le fichier ‘L-20-adapt.mesh’ correspond au maillage de la figure 3.8.a.

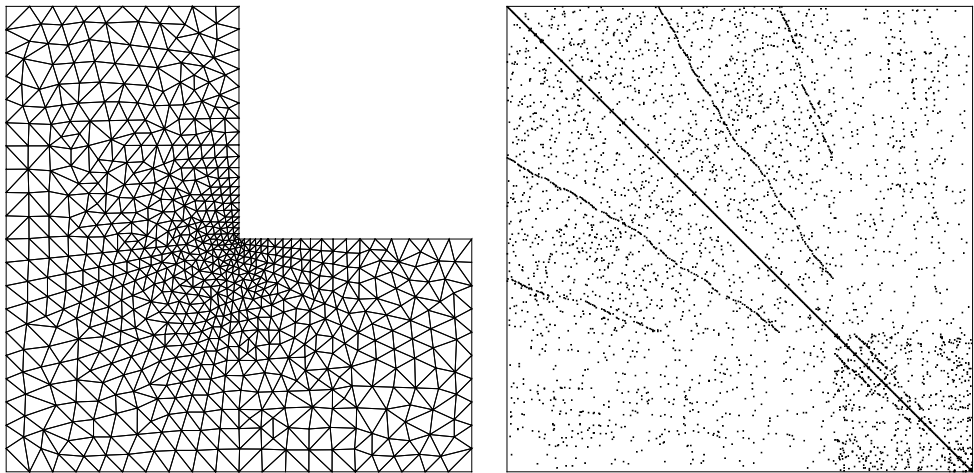


FIGURE 3.8 – Maillage et structure creuse de la matrice d’énergie associée.

### 3.4.6 Résolution du système linéaire

Passons à la résolution du système linéaire (3.5)-(3.6) : issu de la discrétisation éléments finis.

## 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 <typename T, size_t D>
std::valarray<T>
dirichlet (const mesh<T,D>& 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_vertex());
    for (size_t i = 0; i < Th.n_vertex(); i++)
        internal[i] = ! Th.is_boundary(i);
    std::valarray<T> b = M*fh[internal];
    std::valarray<T> x (0., b.size());
    cg (A, x, b, eye(), 100000, 1e-10);
    std::valarray<T> uh (0.0, Th.n_vertex());
    uh [internal] = x;
    return uh;
}

```

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

Nous allons à présent tester le code avec une solution connue explicitement : pour cela nous choisissons tout d'abord  $u$  de la façon suivante :

$$u(x) = \prod_{i=0}^{d-1} \sin(\pi x_i)$$

et nous ajustons le second membre  $f = -\Delta u$ .

## sinusprod.h

```

const double pi = acos(double(-1));
template <size_t D>
double u (const point<double,D>& x) {
    double value = 1;
    for (size_t i = 0; i < D; ++i)
        value *= sin(pi*x[i]);
    return value;
}
template <size_t D>
double f (const point<double,D>& x) {
    return D*pi*pi*u(x);
}

```

Ceci nous permet de calculer l'erreur entre la solution approchée  $u_h$  et l'interpolée  $\pi_h(u)$  de la solution exacte  $u$ . Voici le programme de test en dimension deux :

fem2d.cc

```

#include "dirichlet.h"
#include "sinusprod.h"
using namespace std;
int main() {
    mesh<double,2> 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 :

```

make fem2d
./fem2d < square-10.mesh

```

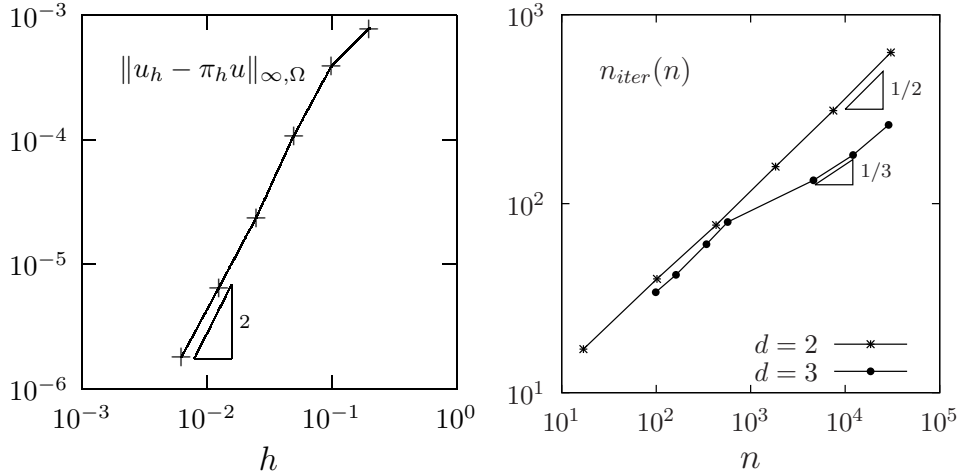


FIGURE 3.9 – Méthode des éléments finis : (a) convergence en fonction du pas  $h$  du maillage ( $d = 2$ ); (b) nombre d'itération du gradient conjugué en fonction de la taille  $n$  du problème ( $d = 2, 3$ ).

La figure 3.9.a trace l'erreur en norme  $L^\infty$ , en fonction du pas  $h$  du maillage. Par définition :

$$\|u_h - \pi_h u\|_{\infty, \Omega} = \max_{x_i \in \text{sommet}(\mathcal{T}_h)} |u_h(x_i) - u(x_i)|$$

Nous pouvons observer que  $\|u_h - \pi_h u\|_{\infty, \Omega} \approx C h^2$  avec une constante  $C > 0$ . Ce comportement est cohérent avec la théorie qui prévoit que  $\|u_h - \pi_h u\|_{\infty, \Omega} \leq C h^2 |\log h|^{3/2}$  pour une certaine constante  $C > 0$  (voir [13, p. 171]). La figure 3.9.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  $n$ . Pour  $d = 2$ , le nombre d'itération croît comme  $\Theta(n^{1/2})$ . Or le coût d'une itération du gradient conjugué

est celle d'un produit matrice-vecteur par  $A$ , soit  $\Theta(n)$ , si bien que le coût total de la résolution est  $\Theta(n^{3/2})$ . Pour  $d = 3$ , le nombre d'itération croît comme  $\Theta(n^{1/3})$ , et donc le coût total de la résolution est alors  $\Theta(n^{4/3})$ . Ceci correspond bien à l'estimation  $\Theta(n^{1+1/d})$ , avec ici  $d = 2$  ou  $3$ , prévue par la théorie développée à l'exercice 14, équation (3.3), page 66 du chapitre 3.1.

### 3.4.7 Problème de la membrane

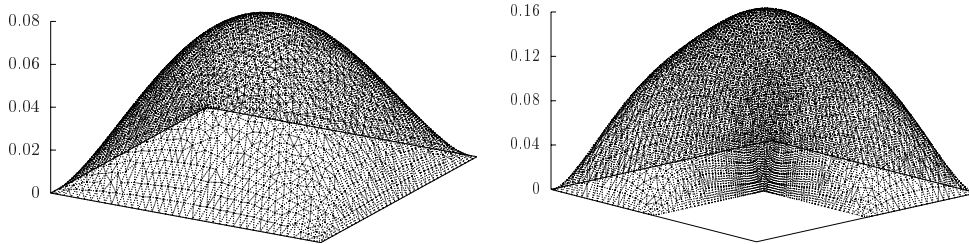


FIGURE 3.10 – Visualisation de la solution du problème de la membrane (a) dans le carré  $]0,1[^2$ ; (b) dans un domaine en forme de  $L$

Maintenant que notre calcul est validé et que nous connaissons les propriétés de notre algorithme numérique, nous pouvons résoudre un problème pour lequel la solution n'est plus connue de façon explicite par une formule simple. Considérons le cas d'un second membre constant  $f = 1$  et toujours avec des conditions de Dirichlet homogène : il s'agit du problème de la membrane, tenue par les bords et soumise à une poussée  $f$ . Le code suivant lit le maillage sur le standard d'entrée et écrit la solution sur le standard de sortie :

membrane.cc

```
#include "dirichlet.h"
#include "valarray_io.h"
using namespace std;
double f (const point<double,2>& x) { return 1; }
int main() {
    mesh<double,2> Th;
    cin >> Th;
    cout << dirichlet (Th, interpolate (Th, f));
}
```

Avec le maillage du domaine  $\Omega = ]0,1[^2$  et un pas  $h \approx 1/40$  :

```
make membrane
./membrane < square-40.mesh > square-40.val
```

Utilisons le programme de visualisation `fem_elevation` présenté en annexe A.3.1 :

```
./fem_elevation square-40.mesh square-40.val
```

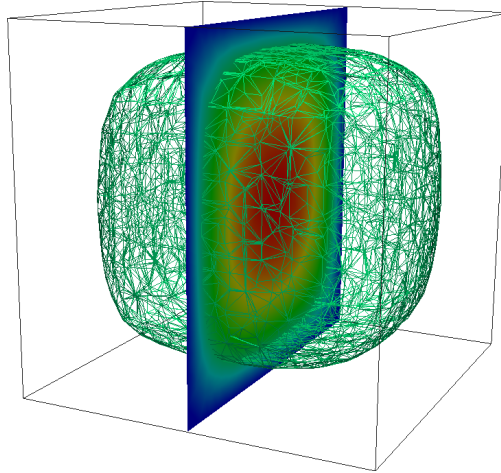


FIGURE 3.11 – Visualisation de la solution du problème de la membrane dans le cube  $]0, 1[^3$ .

Le cas tridimensionnel s'obtient en remplaçant les 2 par des trois dans le fichier `membrane.cc` : nous sauvegardons le résultat dans `membrane3d.cc` et effectuons les calculs :

```
make membrane3d
./membrane3d < cube.mesh > cube.val
```

En utilisant le programme de visualisation `fem3d.vtk` présenté en annexe A.3.2, nous obtenons la représentation de la Fig. 3.10.

### 3.4.8 Exercices

EXERCICE 27. (*Condition aux bords non-homogène*)

Considérons le problème de Poisson avec une condition aux bords de Dirichlet non-homogène :

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

Écrire la formulation variationnelle de ce problème. Modifier le code présenté dans ce chapitre, notamment la fonction `assembly_energy`, afin d'accepter cette condition aux bords. La fonction  $g$  donnée pourra être représentée par une fonction C++ ou une classe-fonction, comme présenté dans l'exemple `u.alpha.h`, page 42.

EXERCICE 28. (*Condition de type Neumann*)

On se propose de résoudre un problème avec une conditions de type Neumann :

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

Écrire la formulation variationnelle de ce problème. Proposer une modification du code présenté dans ce chapitre pour résoudre ce problème.

EXERCICE 29. (*Coefficients non-constants*)

On se propose de résoudre le problème suivant :

$$\begin{aligned} -\nabla \cdot (\eta \nabla u) &= f \text{ dans } \Omega \\ u &= 0 \text{ sur } \partial\Omega \end{aligned}$$

où  $\eta$  est une fonction donnée strictement positive. Ce type de problème apparaît dans la modélisation des milieux poreux (loi de Darcy) ou bien dans la description des fluides complexes, lorsque la viscosité n'est pas constante.

- 1) Montrer que lorsque  $\eta = 1$  nous retrouvons le problème de Poisson étudié dans ce chapitre.
- 2) Écrire la formulation variationnelle de ce problème.
- 3) Proposer une modification du code présenté dans ce chapitre pour résoudre ce problème. La fonction  $\eta$  donnée pourra être représentée par une fonction C++ ou une classe-fonction, comme présenté dans l'exemple `u_alpha.h`, page 42.

EXERCICE 30. (*Problème d'évolution*)

On se propose d'étudier problème d'évolution suivant, appelé équation de la chaleur :

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u &= f \text{ dans } ]0, T[ \times \Omega \\ u &= 0 \text{ sur } ]0, T[ \times \partial\Omega \\ u(t=0) &= u_0 \text{ dans } \Omega \end{aligned}$$

L'inconnue  $u(t, x)$  dépend à la fois du temps  $t \in ]0, T[$  et de l'espace  $x \in \Omega$ . La donnée  $u_0$  représente la condition initiale.

- 1) Discrétiser en temps cette équation par un schéma de différence finies rétrograde de pas  $\Delta t > 0$ . Indication : à chaque pas de temps, le schéma implicite conduit à la résolution d'un sous-problème linéaire continu en espace.
- 2) Écrire la formulation variationnelle du sous-problème.
- 3) Discrétiser en espace la formulation variationnelle précédente et présenter le système linéaire à résoudre.
- 4) Proposer une modification du code présenté dans ce chapitre pour résoudre ce problème.



### 3.4.9 Notes

La méthode des éléments finis a émergé dans les années 1940 avec l'apparition des premiers ordinateurs. L'idée mécanicienne d'origine, toujours d'actualité, était que seuls des éléments discrets, reliés par des forces, permettaient de faire des calculs là où le continu était nécessaire. Sa présentation a par contre considérablement évolué depuis ces débuts, avec des concepts théoriques et des notations associées qui permettent de passer du discret au continu. La formulation variationnelle s'est imposée comme l'outil clef, tant en pratique, pour calculer les coefficients des matrices que théoriquement, pour effectuer les analyses d'erreur et montrer que celle-ci diminuait en augmentant le nombre d'éléments. Parmi les ouvrages les plus classiques présentant la méthode, citons celui de Zienkiewicz et Taylor [57] pour une présentation à un public d'ingénieurs et celle, plus mathématique, de Ciarlet [12], présentant les propriétés de convergence. Mentionnons également l'excellent ouvrage pédagogique de Raviart et Thomas [37].

Un grand nombre de librairies se sont développées depuis les débuts. Parmi les précurseurs, nous mentionnons la librairie `modulef`, écrite en langage fortran et initié dans les années 1970 à l'INRIA, et qui met en avant l'idée de modularité du code. Dans les librairies récentes, citons la librairie [43], écrite en langage C++, et présentant une interface souple et très concis, s'appuyant sur les concepts de formulations variationnelles.

Le chapitre 2.3 de ce livre présente une résolution plus rapide dans le cadre de la méthode des différences finies : la méthode ne s'applique alors plus à une géométrie générale, décrite par un maillage comme ici, mais à une géométrie rectangulaire ou cubique. Dans ce cas particulier, nous saurons obtenir des performances supérieures en temps de résolution du système linéaire.

# Annexe A

## Pré- et post-traitements

### A.1 Ordonnancement et visualisation des matrices creuses

#### A.1.1 Ordonnancement des fichiers matrix market

Les fichiers de matrices creuses au format matrix market (suffixes `.mtx`), notamment ceux disponibles sur le site internet *matrix market* [8], ne sont pas toujours triés par indices de ligne croissants. Afin de pouvoir les lire avec les programmes C++ présentés ici, nous proposons le petit filtre suivant, écrit en langage shell du système unix :

`mtx_sort.sh`

```
#!/bin/sh
head -2 $1
tail -n +3 < $1 | \
  awk '{printf("%.4d %.4d %s\n", $1,$2,$3)}' | \
  sort -g | \
  awk '{printf("%d %d %s\n", $1,$2,$3)}'
```

Par exemple, pour réordonner le fichier `e05r0500.mtx` issu de ce site :

```
sh mtx_sort.sh e05r0500.mtx > e05r0500-new.mtx
mv e05r0500-new.mtx e05r0500.mtx
```

#### A.1.2 Visualisation des matrices creuses

Nous montrons ici comment visualiser les matrices creuses. Pour cela, on génère un fichier que on chargera à l'aide du logiciel `gnuplot`. Voici une fonction membre `put_gnuplot(ostream&)` de la classe `matrix<T>` :

## matrix.h (suite)

```

template <typename T>
void matrix<T>::put_gnuplot (std::ostream& plot) const {
    using namespace std;
    const double d = 0.05;
    size_t h = nrow(), w = ncol();
    plot << "set xrange [0:" << w << "]" << endl
    << "set yrange [0:" << h << "]" << endl
    << "set size ratio -1" << endl
    << "set arrow from 0,0 to " << w << ",0 nohead lc 0" << endl
    << "set arrow from "<<w<<","<<0 to "<<w<<","<<h<<" nohead lc 0"<<endl
    << "set arrow from "<<w<<","<<h<<" to 0,"<<h<<" nohead lc 0"<<endl
    << "set arrow from 0," << h << " to 0,0 nohead lc 0" << endl
    << "set noborder" << endl
    << "set notics" << endl
    << "set style line 1 lt rgb \"black\"" << endl;
    for (size_t i0 = 0; i0 < h; i0++) {
        size_t i = h - i0 - 1;
        for (size_t p = ptr[i0]; p < ptr[i0+1]; ++p) {
            size_t j = idx[p];
            plot << "set object " << p+1 << " rect from " << j+d << "," << i+d
                << " to " << j+1-d << "," << i+1-d << " fc ls 1" << endl;
        }
    }
    plot << "plot -1 notitle" << endl
    << "pause -1 \"<retour>\"" << endl;
}

```

Le programme `mtx2plot.cc` suivant lit une matrice sur le standard d'entrée et écrit les commandes pour `gnuplot` sur le standard de sortie.

## mtx2plot.cc

```

#include "matrix.h"
using namespace std;
int main() {
    matrix<double> a;
    cin >> a;
    a.put_gnuplot(cout);
}

```

```

make mtx2plot
./mtx2plot < e05r0500.mtx > e05r0500.plot
gnuplot e05r0500.plot

```

## A.2 Génération et visualisation de maillages

### A.2.1 Génération de maillages

Cette annexe montre comment générer un maillage avec le générateur `bamg` [27] et convertir les fichiers au format très simple `.mesh` utilisé par les codes C++ de ce livre. L'entrée suivante du programme `bamg` décrit la frontière du domaine  $\Omega = ]0, 1[^2$  :

## square.cad

```

MeshVersionFormatted 0
Dimension 2
Vertices 4
 0 0 1
 1 0 1
 1 1 1
 0 1 1
Edges 4
 1 2 1
 2 3 1
 3 4 1
 4 1 1
hVertices
0.1 0.1 0.1 0.1

```

Nous avons spécifié les quatre sommets, puis les quatres bords en tournant dans le sens positif. La quatrième colonne contient un numéro de domaine pour le bord. Nous renvoyons à la documentation du mailleur **bamg** pour plus de détails. La génération du maillage de triangle de  $\Omega$  se fait par :

```

bamg -g square.cad -o square.bamg
awk -f bamg2mesh.awk < square.bamg > square.mesh

```

Le fichier de script **awk** permet la conversion de formats de fichiers :

## bamg2mesh.awk

```

BEGIN { state = 0; }
($1 == "") { next; }
(state == 0 && $1 != "Vertices") { next; }
(state == 0 && $1 == "Vertices") { state = 1; next; }
(state == 1) { print; nv = $1; iv = 0; state = 2; next; }
(state == 2) { print; ++iv; if (iv == nv) state = 3; next; }
(state == 3 && $1 == "Triangles") { state = 4; next; }
(state == 4) { print; ne = $1; ie = 0; state = 5; next; }
(state == 5) { print; ++ie; if (ie == ne) state = 6; next; }

```

## A.2.2 Visualisation des maillages en dimension deux

Cette annexe a pour but de visualiser les maillages avec **gnuplot** [56]. La fonction **mesh\_gdat** suivante écrit les arêtes du maillage sous le format de données de **gnuplot** :

## mesh\_gdat.h

```

#include "mesh.h"
template <typename T>
void mesh_gdat (std::ostream& gdat, const mesh<T,2>& Th) {
    using namespace std;
    for (size_t e = 0; e < Th.size(); e++) {
        const point<T,2>& a = Th.vertex (Th[e][0]),
                        b = Th.vertex (Th[e][1]),
                        c = Th.vertex (Th[e][2]);

        gdat << a[0] << " " << a[1] << endl
              << b[0] << " " << b[1] << endl
              << c[0] << " " << c[1] << endl
              << c[0] << " " << c[1] << endl
              << a[0] << " " << a[1] << endl << endl;
    }
}

```

Le programme qui suit va lire un fichier de maillage donnée sur la ligne d'appel, écrire le fichier de données pour **gnuplot** et enfin lancer le programme **gnuplot** par une commande système :

## mesh.cc

```

#include "mesh_gdat.h"
#include <fstream>
using namespace std;
int main(int argc, char**argv) {
    if (argc <= 1) return 0;
    mesh<double,2> Th;
    ifstream in(argv[1]);
    in >> Th;
    ofstream gdat("tmp.gdat");
    mesh_gdat (gdat, Th);
    gdat.close ();
    ofstream plot ("tmp.plot");
    plot << "set size ratio 1" << endl
          << "plot 'tmp.gdat' notitle with lines" << endl
          << "pause -1 '<retour>'" << endl;
    plot.close();
    system("gnuplot tmp.plot");
    system("rm -f tmp.plot tmp.gdat");
}

```

Remarquez la création de fichiers temporaires, qui sont effacés à l'issue de la visualisation. La compilation et le lancement du programme sont :

```

make mesh
./mesh L-20-adapt.mesh
./mesh L-40-adapt.mesh

```

Le résultat est représenté sur la Fig. 3.2. page 82.

### A.2.3 Visualisation des maillages en dimension trois avec paraview

Cette annexe a pour but de visualiser les maillages en dimension trois avec paraview [46]. La fonction `mesh_vtk` suivante écrit le maillage sous le format de données `vtk` [44] utilisé par paraview :

`mesh_vtk.h`

```
#include "mesh.h"
template <typename T>
void mesh_vtk (std::ostream& vtk, const mesh<T,3>& Th) {
    using namespace std;
    vtk << "# vtk DataFile Version 1.0" << endl
        << "Unstructured Grid" << endl
        << "ASCII" << endl
        << "DATASET UNSTRUCTURED_GRID" << endl
        << "POINTS " << Th.n_vertex() << " float" << endl;
    for (size_t i = 0; i < Th.n_vertex(); i++)
        vtk << Th.vertex(i) << endl;
    vtk << "CELLS " << Th.size() << " " << 5*Th.size() << endl;
    for (size_t e = 0; e < Th.size(); e++) {
        vtk << "4";
        for (size_t ie = 0; ie < 4; ie++)
            vtk << " " << Th[e][ie];
        vtk << endl;
    }
    vtk << "CELL_TYPES " << Th.size() << endl;
    for (size_t e = 0; e < Th.size(); e++)
        vtk << "10" << endl;
}
```

Le programme qui suit va lire un fichier de maillage donnée sur le standard s'entrée et écrire le fichier de données au format `vtk` sur le standard de sortie.

`mesh3d.cc`

```
#include "mesh_vtk.h"
#include <fstream>
using namespace std;
int main(int argc, char**argv) {
    mesh<double,3> Th;
    cin >> Th;
    mesh_vtk (cout, Th);
}
```

La compilation et le lancement du programme sont :

```
make mesh3d
./mesh3d < cube.mesh > cube.vtk
paraview &
```

Le lancement de `paraview` fait apparaître une fenêtre graphique avec de nombreux menus. Ouvrez le menu `File->open` et chargez `cube.vtk`. Ensuite, dans l'onglet `Properties`, cliquez sur le bouton vert `apply` : le cube apparaît, avec ses surfaces pleines. Remplacez à présent `Surface` par `Wireframe` dans le menu du haut : les

arêtes des faces frontières apparaissent. Allons à présent explorer l'intérieur du maillage : Ouvrez le menu **Filters->Alphabetical->ExtractCellsByRegion** et, dans l'onglet **Properties**, cliquez sur le bouton vert `apply`. le cube apparait coupé en deux par un plan, et ses éléments internes apparaissent. Le résultat est représenté sur la Fig. 3.2, page 82. Il est possible de déplacer le plan de coupe en translation, par le cadre, et en rotation, par sa normale, indiquée par une flèche. Afin de mieux comprendre la structure des tétraèdres, nous pouvons les séparer : ouvrez le menu **Filters->Alphabetical->Shrink**, et, dans l'onglet **Properties**, ajustez le facteur de retrécissement. Il existe d'autres modes de représentation, comme la visualisation de toutes arêtes internes, avec le filtre **ExtractEdges**.

## A.3 Visualisation des solutions de type éléments finis

### A.3.1 Visualisation en dimension deux avec gnuplot

Cette annexe a pour but de visualiser les fonctions de type éléments finis en dimension deux avec **gnuplot** [56]. S'inspirant de l'annexe précédente, qui visualisait les maillages en dimension deux, écrivons la fonction **fem\_gdat** :

fem\_gdat.h

```
#include "mesh.h"
#include <valarray>
template <typename T>
void fem_gdat (std::ostream& gdat,
  const mesh<T,2>& Th, const std::valarray<T>& uh) {
  using namespace std;
  for (size_t e = 0; e < Th.size(); e++) {
    size_t i = Th[e][0], j = Th[e][1], k = Th[e][2];
    const point<T,2>& a = Th.vertex(i),
                    b = Th.vertex(j),
                    c = Th.vertex(k);

    gdat << a[0] << " " << a[1] << " " << uh[i] << "\n"
        << b[0] << " " << b[1] << " " << uh[j] << "\n\n"
        << c[0] << " " << c[1] << " " << uh[k] << "\n"
        << c[0] << " " << c[1] << " " << uh[k] << "\n\n\n";
  }
}
```

En utilisant la commande **splot** de **gnuplot** on obtient une représentation en élévation :

## fem.elevation.cc

```

#include "fem_gdat.h"
#include "valarray_io.h"
#include <fstream>
using namespace std;
int main(int argc, char**argv) {
    if (argc <= 2) return 0;
    mesh<double,2> Th;
    valarray<double> uh;
    ifstream m_in(argv[1]), s_in(argv[2]);
    m_in >> Th;
    s_in >> uh;
    ofstream gdat("tmp.gdat");
    fem_gdat(gdat, Th, uh);
    gdat.close();
    ofstream plot("tmp.plot");
    plot << "set size ratio 1" << endl
          << "splot 'tmp.gdat' notitle with lines" << endl
          << "pause -1 '<retour>'" << endl;
    plot.close();
    system("gnuplot tmp.plot");
    system("rm -f tmp.plot tmp.gdat");
}

```

```

make fem_elevation
./membrane < square-40.mesh > square-40.val
./fem_elevation square-40.mesh square-40.val

```

Le résultat est représenté sur la Fig. 3.10. page 101.

### A.3.2 Visualisation en dimension trois avec paraview

Cette annexe a pour but de visualiser les fonctions de type éléments finis en dimension trois avec **paraview** [46]. La fonction `fem_vtk` suivante écrit le maillage sous le format de données `vtk` [44] utilisé par **paraview** :

## fem.vtk.h

```

#include "mesh_vtk.h"
#include "valarray_io.h"
template <typename T, size_t D>
void fem_vtk(std::ostream& vtk,
             const mesh<T,D>& Th, const std::valarray<T>& uh) {
    using namespace std;
    mesh_vtk(vtk, Th);
    vtk << "POINT_DATA " << uh.size() << endl
        << "SCALARS fem float" << endl
        << "LOOKUP_TABLE default" << endl;
    for (size_t i = 0; i < uh.size(); ++i)
        vtk << uh[i] << endl;
}

```

Le programme qui suit va lire un fichier de maillage et un fichier de solution



élément finis, donnée sur la ligne de commande, et écrire le fichier de données au format `vtk` sur le standard de sortie.

`fem3d_vtk.cc`

```
#include "fem_vtk.h"
#include <fstream>
using namespace std;
int main(int argc, char**argv) {
    if (argc <= 2) return 0;
    mesh<double,3> Th;
    valarray<double> uh;
    ifstream m_in(argv[1]), s_in(argv[2]);
    m_in >> Th;
    s_in >> uh;
    fem_vtk (cout, Th, uh);
}
```

```
make femd_vtk
./membrane3d < cube.mesh > cube.val
./fem3d_vtk cube.mesh cube.val > cube-val.vtk
paraview &
```

Dans la fenêtre `paraview`, ouvrez le menu `File->open` et chargez `cube-val.vtk`. Ensuite, dans l'onglet `Properties`, cliquez sur le bouton vert `apply` : le cube apparaît, avec ses surfaces pleines. La couleur est unie, car la solution vaut zéro aux bords. Allons à présent explorer l'intérieur du maillage : Ouvrez le menu `Filters->Alphabetical->Slice` et, dans l'onglet `Properties`, cliquez sur le bouton vert `apply` : un plan de coupe apparaît. Dans la fenêtre `Pipeline browser`, sélectionnez `cube-val.vtk` en cliquant dessus, puis ouvrez le menu `Filters->Alphabetical->Contours`. Remplacez à présent `Surface` par `Wireframe` dans le menu du haut : la surface isovaleur et à présent en mode grillage. Enfin, ouvrez le menu `Sources->Outline` pour faire apparaître le cadre du cube. Le résultat est représenté sur la Fig. 3.11. page 102.

# Annexe B

## Corrigé des exercices

CORRIGÉ DE L'EXERCICE 1, page 11. (*Application aux rotations*)

L'exercice ne pose pas de problèmes, sinon quelques calculs avec des matrices  $4 \times 4$ .

CORRIGÉ DE L'EXERCICE 2, page 12. (*Promotion de type*)

1) L'opération d'addition `operator+` n'est définie qu'entre deux `quaternion<T>` utilisant la même précision en virgule flottante `T`. Or ici, pour calculer `h3`, les précisions des opérandes `h1` et `h2` sont `float` et `double`. Le résultat aurait un sens, et il n'y aurait pas de perte de précision, car le résultat est attendu en `double`.

2) Le tableau des promotions de type est :

+	float	double	long double
float	float	double	long double
double	double	double	long double
long double	long double	long double	long double

3) La classe `promote` s'écrit :

`promote.h`

```
struct undefined {};  
template<typename T1, typename T2>  
struct promote { typedef undefined type; };  
template<typename T> struct promote<T,T> { typedef T type; };  
template<> struct promote<float, double> { typedef double type; };  
template<> struct promote<double, float> { typedef double type; };
```

Nous n'avons mentionné que la promotion (float,double). L'extension à long double se fait en ajoutant les promotions (float,long double) et (double,long double), soit quatre nouvelles lignes.

4) L'addition entre deux complexes est :

promote.h (suite)

```
#include <complex>
template <typename T1, typename T2>
std::complex<typename promote<T1,T2>::type>
operator+ (const std::complex<T1>& a, const std::complex<T2>& b) {
    return std::complex<typename promote<T1,T2>::type>(
        a.real()+b.real(), a.imag()+b.imag());
}
```

5) L'addition entre deux quaternions est :

promote.h (suite)

```
#include "quaternion.h"
template <typename T1, typename T2>
quaternion<typename promote<T1,T2>::type>
operator+ (const quaternion<T1>& a, quaternion<T2>& b) {
    return quaternion<typename promote<T1,T2>::type>(
        a.z()+b.z(), a.w()+b.w());
}
```

Un programme de test est :

promote.cc

```
#include "promote.h"
#include <iostream>
using namespace std;
int main () {
    complex<float> z1 (0.0, 1.0);
    complex<double> z2 (1.0, 2.0);
    complex<double> z3 = z1 + z2;
    cout << "z3 = " << z3 << endl;
    quaternion<float> h1 (0.0, 1.0, 0.0, 0.0);
    quaternion<double> h2 (1.0, 2.0, 0.0, 0.0);
    quaternion<double> h3 = h1 + h2;
    cout << "h3 = " << h3 << endl;
}
```

6) Insérer une nouvelle classe de virgule flottante, par exemple la classe `qd_real`, revient à ajouter une ligne et une colonne supplémentaire au tableau de la question 2. Ceci nécessite d'ajouter à la classe `promote` des spécialisations avec les trois types à virgule flottante standard, soit six combinaisons. Ensuite, l'addition entre complexes, ou entre quaternions, et combinant ces types flottants, fonctionnera sans autre modification supplémentaire.

CORRIGÉ DE L'EXERCICE 3, page 22. (*Factorisation  $LDL^T$* )

L'application de l'algorithme de factorisation  $LDL^T$  avec cette matrice conduit à un pivot nul. La matrice est inversible, mais n'est pas à valeurs propres positives : ses valeurs propres sont  $1 \pm \sqrt{2}$ . Ainsi, une des valeurs propres est négative, l'autre positive. La matrice n'est donc ni symétrique définie positive, ni symétrique définie négative : on dit qu'elle est symétrique indéfinie. Il est cependant possible, par une permutation des lignes et colonnes, d'appliquer l'algorithme  $LDL^T$  sans rencontrer

de pivot nul :

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \tilde{A} = PAP = \tilde{L}\tilde{D}\tilde{L}^T = \begin{pmatrix} 1 & 0 \\ 0.5 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & -0.5 \end{pmatrix} \begin{pmatrix} 1 & 0.5 \\ 0 & 1 \end{pmatrix}$$

si bien que  $A = P\tilde{L}\tilde{D}(\tilde{P}\tilde{L})^T$ . La matrice  $P\tilde{L}$  n'est pas triangulaire inférieure, mais  $\tilde{L}$  l'est. Il ne s'agit donc pas d'une factorisation  $LDL^T$  à proprement parler, mais à une permutation près.

CORRIGÉ DE L'EXERCICE 4, page 22. (*Factorisation  $LDL^T$  d'une matrice bande*)

1) Le résultat s'obtient par inspection des boucles de l'algorithme de factorisation  $LDL^T$  de la page 19.

2)

**algorithme** : factorisation  $A = LDL^T$  bande  
**entrée** :  $A$  matrice  $n \times n$  symétrique définie positive de bande  $b$   
**sortie** :  $L$  triangulaire inférieure unitaire de bande  $b$  et  $D$  diagonale  
**début**  
  **pour**  $i = 1 : n$   
     $L(i, i) = 1$   
    **pour**  $j = \max(1, i - b) : i - 1$   
       $L(i, j) = A(i, j)$   
      **pour**  $k = \max(1, i - b, j - b) : j - 1$   
         $L(i, j) = L(i, j) - L(i, k) * D(k, k) * L(j, k)$   
       $L(i, j) = L(i, j) / D(j, j)$   
       $D(i, i) = A(i, i)$   
      **pour**  $j = \max(1, i - b) : i - 1$   
         $D(i, i) = D(i, i) - L(i, j)^2 * D(j, j)$   
  **fin**

La boucle la plus externe a une longueur  $n$  et les deux boucles internes ont une longueur de l'ordre de  $b$ , ce qui donne un coût de la factorisation  $\Theta(nb^2)$ . La résolution des systèmes triangulaires conserve également une boucle externe de longueur  $n$  tandis que la boucle interne est de l'ordre de  $b$ , ce qui donne un coût de la résolution  $\Theta(nb)$ .

CORRIGÉ DE L'EXERCICE 5, page 22. (*Algorithme de Strassen*)

L'exercice ne pose pas de problèmes : l'algorithme se déroule sans difficultés.

CORRIGÉ DE L'EXERCICE 6, page 23. (*Multiplication de polynômes*)

Posons  $c_1 = (a + b)(c + d)$ ,  $c_2 = ac$  et  $c_3 = bd$ . Alors  $p_3(x) = p_1(x)p_2(x) = c_2x^2 + (c_1 - c_2 - c_3)x + c_3$  et nous n'avons effectué que trois multiplications.

CORRIGÉ DE L'EXERCICE 7, page 23. (*Multiplication de nombres complexes*)

La solution est similaire à celle de l'exercice précédent.

CORRIGÉ DE L'EXERCICE 8, page 23. (*Matrices denses*)

1) Le rangement peut s'écrire ligne par ligne ou colonne par colonne :

dmatrix.h (suite)

```

template <class T>
dmatrix<T>::dmatrix (size_t nr1, size_t nc1)
: nr(nr1), nc(nc1), v(nr1*nc1) {}
template <class T>
void dmatrix<T>::resize (size_t nr1, size_t nc1)
{ nr = nr1; nc = nc1; v.resize(nr1*nc1); }
template <class T>
size_t dmatrix<T>::nrow () const { return nr; }
template <class T>
size_t dmatrix<T>::ncol () const { return nc; }
template <class T>
T& dmatrix<T>::operator() (size_t i, size_t j) {
    return v[i+j*nr]; }
template <class T>
T dmatrix<T>::operator() (size_t i, size_t j) const {
    return v[i+j*nr]; }

```

2) Le code du produit matrice-vecteur est :

dmatrix.h (suite)

```

template <typename T>
std::valarray<T>
operator* (const dmatrix<T>& a, const std::valarray<T>& x){
    std::valarray<T> y (a.ncol());
    for (size_t i = 0; i < a.nrow(); i++) {
        y[i] = 0;
        for (size_t j = 0; j < a.ncol(); j++)
            y[i] += a(i,j)*x[j];
    }
    return y;
}

```

3) L'algorithme du produit de deux matrices sur la base de trois boucles imbriquées ne pose aucune difficulté particulière, et son coût est  $\Theta(n^3)$ .

CORRIGÉ DE L'EXERCICE 9, page 31. (*Où pincer une corde de guitare ?*)

1)  $c_0 = 1/4$ .

2)  $c_1 = -1/\pi^2$ .

3)  $c_3 = -1/(9\pi^2)$ .

4)  $c_5 = -1/(25\pi^2)$ .

5)  $c_p = -1/(p^2\pi^2)$  quand  $p$  est impair. Ainsi les amplitudes  $|c_p|$  des harmoniques successives décroissent comme  $1/p^2$ .

6) Nous avons  $c_1 = -1/\pi^2 + i(x_0 - 1/2)/\pi$  et plus généralement :

$$c_p = \begin{cases} -\frac{1}{p^2\pi^2} + \frac{i\pi}{p}(x_0 - 1/2) & \text{si } p \text{ est impair} \\ -\frac{i\pi}{p}(x_0 - 1/2) & \text{si } p \text{ est pair} \end{cases}$$

Ainsi les amplitudes  $|c_p|$  des harmoniques successives décroissent comme  $1/p$  dès que  $x_0 \neq 0$  et comme  $1/p^2$  pour  $x_0 = 1/2$  : il est préférable de pincer la corde au milieu pour entendre la note fondamentale la plus pure possible.

CORRIGÉ DE L'EXERCICE 10, page 45. (*Convergence de la méthode des différences finies*)

. 1,2) Le codage nécessite une simple modification du fichier `fish1d.cc`, en incluant `interpolate_1d.h`.

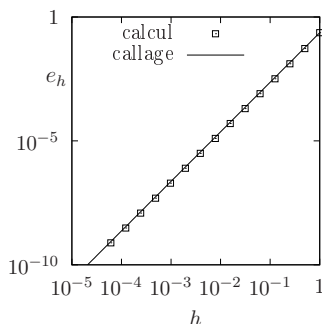


FIGURE B.1 – Convergence en  $\mathcal{O}(h^2)$  de la méthode des différences finies.

3) Rangons les valeurs  $(h, e_h)$  sur deux colonnes dans un fichier de données `err.gdat` puis executons sous `gnuplot` les commandes suivantes :

```
e(h)=c*h**a
a = 1
c = 1
fit e(x) 'err.gdat' via c
set logscale
plot 'fish1d_exo.gdat' w p, e(x)
```

Nous obtenons le graphique représenté sur la Fig. B.1 ainsi que  $a \approx 2$  et  $c \approx 0.23$  : l'erreur asymptotique est  $\mathcal{O}(h^2)$ .

CORRIGÉ DE L'EXERCICE 11, page 46. (*Factorisation  $LDL^T$  avec structure bande*)

1) Le temps de factorisation est  $\Theta(nk^2)$  et celui de la résolution est  $\Theta(nk)$ . Le paramètre  $k$  est le degré des polynômes : il est en pratique de l'ordre de 1 à 5 ou 10, et reste relativement petit devant  $n$  qui peut être arbitrairement grand. Aussi, la méthode reste optimale en dimension un d'espace : le coût total, en considérant  $k$  fixé, est  $\Theta(n)$ .

2) Nous avons  $b = \Theta(m^{d-1}) = \Theta\left(n^{\frac{d-1}{d}}\right)$  si bien que le coût de la factorisation est  $\Theta(n^{3-2/d})$ . en utilisant le résultat de l'exercice 4, page 4. Nous retrouvons un coût optimal en dimension un, et  $\Theta(n^2)$  et  $\Theta(n^{7/3})$  en dimension deux et trois,

respectivement. Dans ces deux derniers cas, la méthode n'est pas du tout intéressante, et il est bien mieux d'utiliser la méthode utilisant la transformée de Fourier qui conduit à un coût  $\Theta(n \log n)$  en dimension deux et trois.

CORRIGÉ DE L'EXERCICE 12, page 58. (*Problème de Poisson par Fourier en dimension trois*)

1) Le schéma et les conditions aux limites s'écrivent :

$$\begin{aligned} -u_{i,j,k+1} - u_{i,j+1,k} - u_{i+1,j,k} + 8u_{i,j,k} \\ - u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} &= h^2 f(x_{i,j,k}), \quad 1 \leq i, j, k \leq m-1 \\ u_{0,j,k} = u_{m,j,k} &= 0, \quad 0 \leq j, k \leq m \\ u_{i,0,k} = u_{i,m,k} &= 0, \quad 1 \leq i \leq m-1, \quad 0 \leq k \leq m \\ u_{i,j,0} = u_{i,j,m} &= 0, \quad 1 \leq i, j \leq m-1 \end{aligned}$$

2) Posons  $u_{j,k} = (u_{i,j,k})_{1 \leq j,k \leq m-1}$  et  $f_{j,k} = (f(x_{i,j,k}))_{1 \leq j,k \leq m-1}$ . En regroupant par blocs :

$$\left\{ \begin{array}{l} -u_{1,k-1} + \tilde{C}u_{1,k} - u_{1,k+1} - u_{2,k} = h^2 f_{1,k}, \quad 2 \leq k \leq m-2 \\ -u_{j-1,1} + \tilde{C}u_{j,1} - u_{j,2} - u_{j+1,1} = h^2 f_{j,1}, \quad 2 \leq j \leq m-2 \\ -u_{j-1,k} - u_{j,k-1} + \tilde{C}u_{j,k} - u_{j,k+1} - u_{j+1,k} = h^2 f_{j,k}, \quad 2 \leq j, k \leq m-2 \\ -u_{j-1,k} - u_{j,k-1} + \tilde{C}u_{j,k} - u_{j,k+1} - u_{j+1,k} = h^2 f_{j,k}, \quad 2 \leq j, k \leq m-2 \\ -u_{m-2,k} - u_{m-1,k-1} + \tilde{C}u_{m-1,k} - u_{m-1,k+1} = h^2 f_{m-1,k}, \quad 2 \leq k \leq m-2 \\ -u_{j-1,m-1} - u_{j,m-2} + \tilde{C}u_{j,m-1} - u_{j+1,m-1} = h^2 f_{j,m-1}, \quad 2 \leq j \leq m-2 \end{array} \right.$$

Nous avons noté la matrice  $\tilde{C} = 8I - C_0$ , de taille  $m-1$ , où  $C_0$  est la matrice introduite dans le cours. Les valeurs propres de  $\tilde{C} = 8I - C_0$  sont  $\lambda_l = 8 - 2\cos(l\pi/m)$ ,  $1 \leq l \leq m-1$ . Soit  $V$  la matrice introduite en cours, dont les colonnes sont les vecteurs propres de  $C_0$ . On note  $\Lambda$  la matrice diagonale des  $\lambda_l$ , si bien que  $\tilde{C} = V\Lambda V^T$ . En décomposant  $u_{j,k}$  et  $f_{j,k}$  dans la base des vecteurs propres :  $u_{j,k} = V^T \tilde{u}_{j,k}$  et  $f_{j,k} = V^T \tilde{f}_{j,k}$ , il vient :

$$\left\{ \begin{array}{l} -\tilde{u}_{1,k-1} + \Lambda \tilde{u}_{1,k} - \tilde{u}_{1,k+1} - \tilde{u}_{2,k} = h^2 \tilde{f}_{1,k}, \quad 2 \leq k \leq m-2 \\ -\tilde{u}_{j-1,1} + \Lambda \tilde{u}_{j,1} - \tilde{u}_{j,2} - \tilde{u}_{j+1,1} = h^2 \tilde{f}_{j,1}, \quad 2 \leq j \leq m-2 \\ -\tilde{u}_{j-1,k} - \tilde{u}_{j,k-1} + \Lambda \tilde{u}_{j,k} - \tilde{u}_{j,k+1} - \tilde{u}_{j+1,k} = h^2 \tilde{f}_{j,k}, \quad 2 \leq j, k \leq m-2 \\ -\tilde{u}_{j-1,k} - \tilde{u}_{j,k-1} + \Lambda \tilde{u}_{j,k} - \tilde{u}_{j,k+1} - \tilde{u}_{j+1,k} = h^2 \tilde{f}_{j,k}, \quad 2 \leq j, k \leq m-2 \\ -\tilde{u}_{m-2,k} - \tilde{u}_{m-1,k-1} + \Lambda \tilde{u}_{m-1,k} - \tilde{u}_{m-1,k+1} = h^2 \tilde{f}_{m-1,k}, \quad 2 \leq k \leq m-2 \\ -\tilde{u}_{j-1,m-1} - \tilde{u}_{j,m-2} + \Lambda \tilde{u}_{j,m-1} - \tilde{u}_{j+1,m-1} = h^2 \tilde{f}_{j,m-1}, \quad 2 \leq j \leq m-2 \end{array} \right.$$

Redéveloppons à présent par blocs. Nous avons, pour tout  $l \in \{1, \dots, m-1\}$  :

$$\left\{ \begin{array}{l} -\tilde{u}_{l,1,k-1} + \lambda_l \tilde{u}_{l,1,k} - \tilde{u}_{l,1,k+1} - \tilde{u}_{l,2,k} = h^2 \tilde{f}_{l,1,k}, \quad 2 \leq k \leq m-2 \\ -\tilde{u}_{l,j-1,1} + \lambda_l \tilde{u}_{l,j,1} - \tilde{u}_{l,j,2} - \tilde{u}_{l,j+1,1} = h^2 \tilde{f}_{l,j,1}, \quad 2 \leq j \leq m-2 \\ -\tilde{u}_{l,j-1,k} - \tilde{u}_{l,j,k-1} + \lambda_l \tilde{u}_{l,j,k} - \tilde{u}_{l,j,k+1} - \tilde{u}_{l,j+1,k} = h^2 \tilde{f}_{l,j,k}, \quad 2 \leq j, k \leq m-2 \\ -\tilde{u}_{l,j-1,k} - \tilde{u}_{l,j,k-1} + \lambda_l \tilde{u}_{l,j,k} - \tilde{u}_{l,j,k+1} - \tilde{u}_{l,j+1,k} = h^2 \tilde{f}_{l,j,k}, \quad 2 \leq j, k \leq m-2 \\ -\tilde{u}_{l,m-2,k} - \tilde{u}_{l,m-1,k-1} + \lambda_l \tilde{u}_{l,m-1,k} - \tilde{u}_{l,m-1,k+1} = h^2 \tilde{f}_{l,m-1,k}, \quad 2 \leq k \leq m-2 \\ -\tilde{u}_{l,j-1,m-1} - \tilde{u}_{l,j,m-2} + \lambda_l \tilde{u}_{l,j,m-1} - \tilde{u}_{l,j+1,m-1} = h^2 \tilde{f}_{l,j,m-1}, \quad 2 \leq j \leq m-2 \end{array} \right.$$

Ainsi, ces  $m - 1$  systèmes linéaires, chacun de taille  $(m - 1)^2$ , sont complètement découplés, et peuvent être résolus séparément, pour chaque  $l$  fixé. Fixons donc un indice  $l$ , et notons  $A_l$  la matrice du système correspondant :

$$A_l = \begin{pmatrix} \hat{C}_l & -I & & & \\ -I & \hat{C}_l & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & \hat{C}_l & -I \\ & & & -I & \hat{C}_l \end{pmatrix}$$

avec  $\hat{C}_l = \lambda_l I - C_0$ . Ce système est analogue à celui que nous obtenons par une méthode de différence finie en dimension deux : seuls les coefficients diagonaux, qui valaient 4 valent ici  $\lambda_l$ . Nous pouvons donc recommencer une seconde fois l'opération. Nous connaissons les valeurs propres de  $\hat{C}_l$ , qui sont

$$\hat{\lambda}_{l,j} = \lambda_l - 2 \cos(j\pi/m) = 8 - 2 \cos(l\pi/m) - 2 \cos(j\pi/m)$$

En notant  $\hat{\Lambda}_l$  la matrice diagonale des  $(\hat{\lambda}_{l,j})_{1 \leq j \leq m-1}$  et en décomposant  $\tilde{u}_{l,k} = (\tilde{u}_{l,j,k})_{1 \leq j \leq m-1}$  et  $\tilde{f}_{l,k} = (\tilde{f}_{l,j,k})_{1 \leq j \leq m-1}$  dans la base des vecteurs propres :  $\tilde{u}_{l,k} = V^T \hat{u}_{l,k}$  et  $\tilde{f}_{l,k} = V^T \hat{f}_{l,k}$ , il vient

$$\begin{cases} \Lambda_l \hat{u}_{l,1} & - & \hat{u}_{l,2} & = & \hat{f}_{l,1}, \\ -\hat{u}_{l,k-1} & + & \Lambda_l \hat{u}_{l,k} & - & \hat{u}_{l,k+1} & = & \hat{f}_{l,k}, \\ -\hat{u}_{l,m-2} & + & \Lambda_l \hat{u}_{l,m-1} & - & & = & \hat{f}_{l,m-1}. \end{cases} \quad 2 \leq k \leq m-2,$$

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

$$\begin{cases} \hat{\lambda}_{l,j} \hat{u}_{l,j,1} & - & \hat{u}_{l,j,2} & = & \hat{f}_{l,j,1}, \\ -\hat{u}_{l,j,k-1} & + & \hat{\lambda}_{l,j} \hat{u}_{l,j,k} & - & \hat{u}_{l,j,k+1} & = & \hat{f}_{l,j,k}, \\ -\hat{u}_{l,j,m-2} & + & \hat{\lambda}_{l,j} \hat{u}_{l,j,m-1} & - & & = & \hat{f}_{l,j,m-1}. \end{cases} \quad 2 \leq k \leq m-2,$$

Ainsi, ces  $(m-1)^2$  systèmes linéaires sont complètement découplés : pour chaque  $(l, j)$  fixés, nous sommes en présence d'un système tridiagonal de matrice  $A_{l,j} = \hat{\lambda}_{l,j} I - C_0$  que nous savons résoudre par méthode directe en un temps de calcul optimal.

CORRIGÉ DE L'EXERCICE 13, page 65. (*Majoration du nombre d'itérations*)

Partons du développement limité suivant, au voisinage de  $\mu = 0$ , avec  $\mu \in [0, 1]$  :

$$\log \left( \frac{1+\mu}{1-\mu} \right) = 2 \left( \mu + \frac{\mu^3}{3} + \dots + \frac{\mu^{2k+1}}{2k+1} + \dots \right) \geq 2\mu$$

Notez que ce développement se déduit directement de celui de  $\log(1+\mu)$ . En prenant l'exponentielle de l'inégalité précédente, croissante, puis l'inverse, qui est décroissante, et change le sens de l'inégalité, et enfin en posant  $x = 1/\mu$ , nous obtenons  $(x-1)/(x+1) \leq \exp(-2/x)$ , ceci pour tout  $x \geq 1$ . Le résultat des deux questions suivantes se déduit directement de la majoration précédente.



CORRIGÉ DE L'EXERCICE 14, page 65. (*Problème de Poisson multi-dimensionnel*)

1) En dimension un, le schéma est donné par (2.4). Ce schéma se généralise en dimensions suivant :

$$\begin{aligned} -u_{i,j+1} - u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j-1} &= h^2 f(x_{i,j}), \quad 1 \leq i, j \leq m-1 \\ u_{0,j} &= u_{m,j} = 0, \quad 0 \leq j \leq m \\ u_{i,0} &= u_{i,m} = 0, \quad 1 \leq i \leq m-1 \end{aligned}$$

et en dimension trois suivant

$$\begin{aligned} -u_{i,j,k+1} - u_{i,j+1,k} - u_{i+1,j,k} + 8u_{i,j,k} \\ -u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} &= h^2 f(x_{i,j,k}), \quad 1 \leq i, j, k \leq m-1 \\ u_{0,j,k} = u_{m,j,k} &= 0, \quad 0 \leq j, k \leq m \\ u_{i,0,k} = u_{i,m,k} &= 0, \quad 1 \leq i \leq m-1, \quad 0 \leq k \leq m \\ u_{i,j,0} = u_{i,j,m} &= 0, \quad 1 \leq i, j \leq m-1 \end{aligned}$$

2) La taille du système linéaire est  $n = (m-1)^d$ .

3) Partant des schémas précédents, on vérifie que le nombre de coefficients par ligne pour  $d = 1$  est majoré par 3. Dans le cas  $d = 2$ , il est majoré par 5 et pour  $d = 3$  par 7. D'une façon générale, le nombre de coefficients non-nuls est majoré par  $2^d + 1$ . Lorsque  $h = 1/m$  tend vers zéro, la matrice devient grande,  $n$  tend vers l'infini, mais le nombre de coefficients non-nuls  $\text{nnz}(A)$  de la matrice est majoré par  $(2^d + 1)n$ , avec  $n = (m-1)^d$ . Si bien que le coût du produit matrice vecteur est  $\Theta(n)$ .

4) De  $n = (m-1)^d$  il vient  $m = n^{1/d} + 1$  et  $h = 1/(n^{1/d} + 1)$  et enfin  $\text{cond}(A) = \mathcal{O}(n^{2/d})$ .

5) L'expression (3.3) découle directement de (3.2) et de la question précédente

6) Nous avons une multiplication matrice-vecteur par itération : le coût de la résolution est  $T(n) = \mathcal{O}(n^{1+1/d} \log(\varepsilon^{-1}))$ .

7) Il vient  $\text{cond}(M^{-1}A) = \mathcal{O}(n^{1/d})$  et  $T(n) = \mathcal{O}(n^{1+1/(2d)} \log(\varepsilon^{-1}))$ .

8) Pour  $d = 1$ , la méthode directe sur matrice tridiagonale est optimale, avec un coût  $\Theta(n)$  alors que le gradient conjugué préconditionné conduit à un coût  $T(n) = \mathcal{O}(n^{3/2} \log(\varepsilon^{-1}))$ . Pour  $d = 2$  ou 3, la méthode directe utilisant la transformation de Fourier conduit à au coût  $\mathcal{O}(n \log n)$ , ce qui est également meilleur que le gradient conjugué préconditionné. De plus, ces méthodes directes ne font pas intervenir la précision  $\varepsilon$ . Ainsi, pour résoudre le problème issu de la discrétisation en différences finie avec une grille uniforme, il vaut mieux utiliser une méthode directe plutôt que l'algorithme du gradient conjugué. L'algorithme du gradient conjugué a l'intérêt de s'appliquer à des problèmes plus généraux, pour des grilles non uniformes ou des problèmes issus de la méthode des éléments finis.

CORRIGÉ DE L'EXERCICE 15, page 74. (*Intégrité d'une matrice creuse*)

Après avoir vérifié les dimensions des tableaux, il suffit de vérifier que les indices de colonne sont dans l'intervalle  $[0, \text{ncol}[$  :

## matrix.h (suite)

```

template<typename T>
void matrix<T>::check() const {
    assert (ptr[0] == 0);
    assert (ptr[nrow()] == idx.size() && ptr[nrow()] == val.size());
    for (size_t i = 0; i < nrow(); ++i) {
        assert (ptr[i] <= ptr[i+1]);
        for (size_t p = ptr[i]; p < ptr[i+1]; ++p)
            assert (idx[p] < ncol());
    }
}

```

CORRIGÉ DE L'EXERCICE 16, page 74. (*Information sur une matrice*)

1) Pour avoir la norme de Frobenius, il suffit d'effectuer la somme des carrés des coefficients non-nuls, puis d'en prendre la racine carrée :

## matrix.h (suite)

```

template<typename T>
T matrix<T>::frobenius_norm() const {
    T sum = 0;
    for (size_t p = 0; p < nnz(); ++p)
        sum += val[p]*val[p];
    return sqrt(sum);
}

```

2) Cette boucle nécessite le numéro de ligne, aussi nous écrivons deux boucles emboîtées :

## matrix.h (suite)

```

template<typename T>
size_t matrix<T>::average_dist_diag() const {
    size_t sum = 0;
    for (size_t i = 0; i < nrow(); ++i)
        for (size_t p = ptr[i]; p < ptr[i+1]; ++p)
            sum += abs(idx[p] - i);
    return sum/nnz();
}

```

3) Cette boucle a une structure similaire à la précédente :

matrix.h (suite)

```
template<typename T>
T matrix<T>::diagonal_dominant() const {
    T dd = 0;
    for (size_t i = 0; i < nrow(); ++i) {
        T sum = 0, aii = 0;
        for (size_t p = ptr[i]; p < ptr[i+1]; ++p)
            if (idx[p] == i) aii = abs(val[p]);
            else sum += abs(val[p]);
        dd = std::max(dd, aii-sum);
    }
    return dd;
}
```

CORRIGÉ DE L'EXERCICE 17, page 74. (*Multiplications par une matrice diagonale*)

1) La première fonction est très simple :

matrix.h (suite)

```
template<typename T>
matrix<T>&
matrix<T>::inplace_left_mult(const std::valarray<T>& d) {
    for (size_t p = 0; p < nnz(); ++p)
        val[p] *= d[idx[p]];
    return *this;
}
```

2) Nous avons besoin ici de l'indice de ligne, aussi nous écrivons deux boucles imbriquées :

matrix.h (suite)

```
template<typename T>
matrix<T>&
matrix<T>::operator*= (const std::valarray<T>& d) {
    for (size_t i = 0; i < nrow(); ++i)
        for (size_t p = ptr[i]; p < ptr[i+1]; ++p)
            val[p] *= d[i];
    return *this;
}
```

3) Il suffit de sauvegarder  $A$  dans  $B$  et  $C$ , puis d'appeler les fonctions précédentes sur  $B$  et  $C$  :

## matrix\_diag\_mult.cc

```
#include "matrix.h"
using namespace std;
int main() {
    matrix<double> a;
    cin >> a;
    valarray<double> d1(2.0,a.nrow());
    matrix<double> b = a;
    b.inplace_left_mult(d1);
    valarray<double> d2(2.0,a.ncol());
    matrix<double> c = a;
    c *= d2;
}
```

CORRIGÉ DE L'EXERCICE 18, page 74. (*Extraction de la diagonale*)

## matrix.h (suite)

```
template<typename T>
std::valarray<T> diag (const matrix<T>& a) {
    std::valarray<T> d (T(0), std::min(a.nrow(), a.ncol()));
    for (size_t i = 0; i < d.size(); ++i)
        for (size_t p = a.ptr[i]; p < a.ptr[i+1]; ++p)
            if (i == a.idx[p]) d[i] = a.val[p];
    return d;
}
```

CORRIGÉ DE L'EXERCICE 19, page 75. (*Ajout d'une matrice diagonale*)

La structure du code est similaire à celle proposée par les exercices précédents :

## matrix.h (suite)

```
template<typename T>
matrix<T>&
matrix<T>::operator+= (const std::valarray<T>& d) {
    assert (d.size() == std::min(nrow(), ncol()));
    for (size_t i = 0; i < d.size(); ++i) {
        bool diag_founded = false;
        for (size_t p = ptr[i]; p < ptr[i+1]; ++p)
            if (i == idx[p]) { val[p] += d[i]; diag_founded = true; }
        assert(diag_founded);
    }
    return *this;
}
```

CORRIGÉ DE L'EXERCICE 20, page 75. (*Produit transposé*)

Il faut bien prendre garde à initialiser le tableau du résultat à zéro avant la boucle, car le résultat y est accumulé ensuite.

## matrix.h (suite)

```

template<typename T>
std::valarray<T>
matrix<T>::trans_mult (const std::valarray<T>& x) const {
    std::valarray<T> y (T(0), ncol());
    for (size_t i = 0; i < nrow(); ++i)
        for (size_t p = ptr[i]; i < ptr[i+1]; ++p)
            y[idx[p]] += val[p]*x[i];
    return y;
}

```

CORRIGÉ DE L'EXERCICE 21, page 75. (*Extraction d'une partie triangulaire*)

Un premier passage sur la structure creuse de  $A$  permet de calculer  $\text{nnz}(\text{tril}(A))$  et de dimensionner la structure creuse de la partie inférieure. Un second passage effectue la copie. Le code pour la fonction  $\text{triu}(A)$  est similaire.

## matrix.h (suite)

```

template <typename T>
matrix<T> tril (const matrix<T>& a) {
    size_t l_nnz = 0;
    for (size_t i = 0; i < a.nrow(); ++i)
        for (size_t p = a.ptr[i]; p < a.ptr[i+1]; ++p)
            if (a.idx[p] < i) l_nnz++;
    matrix<T> l;
    l.ptr.resize(a.nrow()+1);
    l.idx.resize(l_nnz);
    l.val.resize(l_nnz);
    l.idxmax = a.idxmax;
    l.ptr[0] = 0;
    for (size_t q = 0, i = 0; i < a.nrow(); ++i) {
        for (size_t p = a.ptr[i]; p < a.ptr[i+1]; ++p) {
            if (a.idx[p] >= i) continue;
            l.idx[q] = a.idx[p];
            l.val[q] = a.val[p];
            ++q;
        }
        l.ptr[i+1] = q;
    }
    return l;
}

```

CORRIGÉ DE L'EXERCICE 22, page 75. (*Vecteurs creux*)

1) Pour les deux vecteurs  $x$  et  $y$ , nous avons :

x.val    

1	2	-7
---	---	----

y.val    

3	5	7	11
---	---	---	----

x.idx    

0	1	3
---	---	---

y.idx    

1	2	3	4
---	---	---	---

2) Le constructeur est :

## spvalarray.h (suite)

```
template <class T>
spvalarray<T>::spvalarray (size_t n)
: idx(), val(), idxmax(n) {}
```

3) Le constructeur de copie et l'opérateur d'affectation sont :

## spvalarray.h (suite)

```
template <class T>
spvalarray<T>::spvalarray (const spvalarray<T>& x)
: idx(x.idx), val(x.val), idxmax(x.idxmax) {}
template <class T>
spvalarray<T>& spvalarray<T>::operator= (const spvalarray<T>& x) {
    idx = x.idx;
    val = x.val;
    idxmax = x.idxmax;
    return *this;
}
```

4) Le constructeur à partir d'un tableau dense s'effectue en deux boucles. La première compte le nombre d'éléments non-nuls et permet de dimensionner les tableaux. La seconde boucle copie les valeurs.

## spvalarray.h (suite)

```
template <class T>
spvalarray<T>::spvalarray (const std::valarray<T>& x)
: idx(), val(), idxmax(x.size()) {
    size_t nnz = 0;
    for (size_t i = 0; i < x.size(); ++i)
        if (x[i] != 0) nnz++;
    idx.resize(nnz);
    val.resize(nnz);
    for (size_t i = 0, p = 0; i < x.size(); ++i) {
        if (x[i] == 0) continue;
        idx[p] = i;
        val[p] = x[i];
        ++p;
    }
}
```

tandis que la conversion en tableau dense s'écrit :

## spvalarray.h (suite)

```
template <class T>
spvalarray<T>::operator std::valarray<T>() const {
    std::valarray<T> x (T(0), idxmax);
    for (size_t p = 0; p < nnz(); ++p)
        x[idx[p]] = val[p];
    return x;
}
```

5) Les fonctions d'accès ne présentent pas de difficulté :

## spvalarray.h (suite)

```
template <class T>
size_t spvalarray<T>::size() const { return idxmax; }
template <class T>
size_t spvalarray<T>::nnz() const { return idx.size(); }
```

6) En utilisant les fonctionnalités de la classe valarray :

## spvalarray.h (suite)

```
template <class T>
spvalarray<T> operator* (const T& c, const spvalarray<T>& x) {
    spvalarray<T> y = x;
    y.val *= c;
    return y;
}
```

7) Par un algorithme de dichotomie :

## spvalarray.h (suite)

```
template <class T>
T spvalarray<T>::operator[] (const size_t i) const {
    long pmin = 0, pmax = nnz()-1;
    while (pmin <= pmax) {
        long p = (pmin + pmax)/2;
        if (idx[p] == i) return val[p];
        if (idx[p] < i) pmin = p+1; else pmax = p-1;
    }
    return 0;
}
```

8) La somme des deux vecteurs est :

$$z = x+y = (1, 5, 5, 9, 11, 0)$$

et  $\text{nnz}(z) = 5$  avec la représentation creuse :

z.val	1	5	5	9	11
z.idx	0	1	2	3	4

9) Le comptage du nombre d'éléments non-nuls d'une somme s'écrit :

## spvalarray.h (suite)

```

#include <cassert>
template <class T>
size_t spvalarray<T>::add_nnz (const spvalarray<T>& y) const {
    assert (idxmax == y.idxmax);
    size_t nnz_z = 0;
    for (size_t p = 0, q = 0; p < nnz() || q < y.nnz(); ++nnz_z) {
        size_t ix = (p < nnz()) ? idx[p] : idxmax;
        size_t iy = (q < y.nnz()) ? y.idx[q] : idxmax;
        if (ix == iy) { ++p; ++q; }
        else if (ix < iy) ++p;
        else ++q;
    }
    return nnz_z;
}

```

10) Ceci permet de dimensionner le tableau contenant le résultat puis d'y effectuer le calcul :

## spvalarray.h (suite)

```

template <class T>
spvalarray<T>
spvalarray<T>::operator+ (const spvalarray<T>& y) const {
    size_t nnz_z = add_nnz(y);
    spvalarray<T> z (idxmax);
    z.idx.resize(nnz_z);
    z.val.resize(nnz_z);
    for (size_t p = 0, q = 0, r = 0; p < nnz() || q < y.nnz(); ++r) {
        size_t ix = (p < nnz()) ? idx[p] : idxmax;
        size_t iy = (q < y.nnz()) ? y.idx[q] : idxmax;
        if (ix == iy) { z.idx[r] = ix;
            z.val[r] = val[p++] + y.val[q++]; }
        else if (ix < iy) { z.idx[r] = ix;
            z.val[r] = val[p++]; }
        else { z.idx[r] = iy;
            z.val[r] = y.val[q++]; }
    }
    return z;
}

```

11) Un format de fichier pour les vecteurs creux et similaire à celui des matrices creuses contiendrait le nombre d'éléments non-nuls suivi des couples (indice,valeur).



Le profil final de la classe est :

spvalarray.h (suite)

```
#include <valarray>
template <class T>
class spvalarray {
public:
    spvalarray (size_t n = 0);
    spvalarray (const spvalarray<T>&);
    spvalarray (const std::valarray<T>&);
    spvalarray<T>& operator= (const spvalarray<T>&);
    operator std::valarray<T>() const;
    size_t size() const;
    size_t nnz() const;
    T operator[] (const size_t i) const;
    template <class U> friend
    spvalarray<U> operator* (const U& c, const spvalarray<U>& x);
    spvalarray<T> operator+ (const spvalarray<T>& y) const;
protected:
    std::valarray<size_t> idx;
    std::valarray<T> val;
    size_t idxmax;
    size_t add_nnz (const spvalarray<T>& y) const;
};
```

CORRIGÉ DE L'EXERCICE 23, page 76. (*Somme de deux matrices creuses*)

Cet exercice est très similaire aux questions 9 et 10 de l'exercice précédent, concernant les vecteurs creux : il y a juste à ajouter une boucle sur les lignes creuses de la matrice.

—

CORRIGÉ DE L'EXERCICE 24, page 86. (*Mesure d'un domaine maillé*)

Il s'agit d'une simple boucle faisant la somme des mesures des éléments :

mesh\_meas.h

```
#include "mesh.h"
#include "meas.h"
template<typename T, size_t D>
T meas (const mesh<T,D>& Th) {
    T sum = 0;
    for (size_t e = 0; e < Th.size(); ++e)
        sum += meas(Th,e);
    return sum;
}
```

CORRIGÉ DE L'EXERCICE 25, page 86. (*Longueurs extrêmes des arêtes*)

1) Le cas de la dimension deux correspond à une spécialisation :

## mesh\_hmax.h (suite)

```
template<typename T>
T hmax (const mesh<T,2>& Th, size_t e) {
    const element<2>& K = Th[e];
    size_t ia = K[0], ib = K[1], ic = K[2];
    const point<T,2>& a = Th.vertex(ia), b = Th.vertex(ib),
                    c = Th.vertex(ic);
    return std::max(dist(a,b), std::max(dist(b,c), dist(c,a)));
}
```

La spécialisation pour  $d = 1$  s'obtient en mesurant la longueur de l'élément. Pour  $d = 3$ , il faudra prendre le max des six arêtes du tétraèdre. 2) La fonction générale sur un maillage est :

## mesh\_hmax.h (suite)

```
template<typename T, size_t D>
T hmax (const mesh<T,D>& Th) {
    T h = 0;
    for (size_t e = 0; e < Th.size(); ++e)
        h = std::max (h, hmax(Th,e));
    return h;
}
```

CORRIGÉ DE L'EXERCICE 26, page 86. (*Qualité d'un maillage*)

- 1) La vérification est immédiate.
- 2) L'aire d'un triangle équilatéral  $K_0$  de longueur de coté  $L$  est  $\text{meas}(K_0) = \sqrt{3}L^2/4$ , si bien que  $r_{K_0} = L/\sqrt{3}$  et  $\rho_{K_0} = L/(2\sqrt{3})$  et finalement  $Q_{K_0} = 1/2$ .
- 3) L'écriture des fonctions à partir des formules données pour  $r_K$  et  $\rho_K$  est similaire à celle de la fonction  $\text{meas}(K)$  de ce chapitre ou du calcul de la plus longue arête, comme dans l'exercice précédent.

## mesh\_quality.h (suite)

```
#include "mesh.h"
#include "meas.h"
template<typename T>
T quality (const mesh<T,2>& Th, size_t e) {
    const element<2>& K = Th[e];
    size_t ia = K[0], ib = K[1], ic = K[2];
    const point<T,2>& a = Th.vertex(ia), b = Th.vertex(ib),
                    c = Th.vertex(ic);
    T ab = dist(a,b), bc = dist(b,c), ca = dist(c,a);
    T area = meas(Th,e);
    T r_K = ab*bc*ca/(4*area);
    T rho_K = 2*area/(ab + bc + ca);
    return 2*rho_K/r_K;
}
```

- 4) La qualité du maillage s'obtient par itération sur les éléments :

mesh\_quality.h (suite)

```
template<typename T, size_t D>
T quality (const mesh<T,D>& Th) {
    T q = 1;
    for (size_t e = 0; e < Th.size(); ++e)
        q = std::min (q, quality(Th,e));
    return q;
}
```

5) Il suffit de compter le nombre d'éléments par tranche :

mesh\_quality.h

```
#include "mesh_quality2d.icc"
#include "mesh_quality.icc"
```

6) Dans le cas de la dimension un, il n'y a pas de notion de qualité d'un élément. Dans le cas de la dimension trois,  $r_k$  désigne le rayon de la sphère passant par les quatres sommets d'un tétraèdre, et  $\rho_K$  le rayon de la plu grande sphère inscrite : l'extension de la notion de qualité est alors immédiate.

# Liste des exercices

1	Application aux rotations	11
2	Promotion de type	12
3	Factorisation $LDL^T$	22
4	Factorisation $LDL^T$ d'une matrice bande	22
5	Algorithme de Strassen	22
6	Multiplication de polynômes	23
7	Multiplication de nombres complexes	23
8	Matrices denses	23
9	Où pincer une corde de guitare ?	31
10	Convergence de la méthode des différences finies	45
11	Factorisation $LDL^T$ avec structure bande	46
12	Problème de Poisson par Fourier en dimension trois	58
13	Majoration du nombre d'itérations	65
14	Problème de Poisson multi-dimensionnel	65
15	Intégrité d'une matrice creuse	74
16	Information sur une matrice	74
17	Multiplications par une matrice diagonale	74
18	Extraction de la diagonale	74
19	Ajout d'une matrice diagonale	75
20	Produit transposé	75
21	Extraction d'une partie triangulaire	75
22	Vecteurs creux	75
23	Somme de deux matrices creuses	76
24	Mesure d'un domaine maillé	86
25	Longueurs extrêmes des arêtes	86
26	Qualité d'un maillage	86
27	Condition aux bords non-homogène	102
28	Condition de type Neumann	102
29	Coefficients non-constants	103
30	Problème d'évolution	103



# Annexe C

## GNU free documentation license

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

`<http://fsf.org/>`

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF

designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties : any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.



### 3. Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

### 4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer

- than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
  - D. Preserve all the copyright notices of the Document.
  - E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
  - F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
  - G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
  - H. Include an unaltered copy of this License.
  - I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
  - J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
  - K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
  - L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
  - M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
  - N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
  - O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

## 6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of

that document.

## 7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright

holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

## 10. Future revision of this licence

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

## 11. Relicensing

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## Addendum : how to use this license for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page :

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this :

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.



# Bibliographie

- [1] P. Alliez, A. Fabri, and E. Fogel. Computational geometry algorithm library, 2008. slides. [87](#)
- [2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *Lapack user's guide*. SIAM, Philadelphia, USA, third edition, 1999. [25](#)
- [3] O. Axelsson and V. A. Barker. *Solution of boundary value problems. Theory and computation*. SIAM, Philadelphia, PA, USA, second edition, 2001. [49](#), [64](#), [66](#), [67](#)
- [4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozow, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems*. SIAM, Philadelphia, PA, USA, second edition, 1994. <http://www.netlib.org/templates>. [67](#)
- [5] F. Di Benedetto, G. Fiorentino, and S. Serra. CG preconditioning for Toeplitz matrices. *Computers & Mathematics with Applications*, 25(6) :35–45, 1993. [59](#)
- [6] M. Benzi. Preconditioning techniques for large linear systems : a survey. *J. Comput. Phys.*, 182(2) :418–477, 2002. [67](#)
- [7] J.-D. Boissonnat and M. Yvinec. *Géométrie algorithmique*. Ediscience international, Paris, 1995. [14](#), [24](#), [87](#)
- [8] R. Boisvert, R. Pozo, and K. Remington. The matrix market exchange formats : initial design. Technical Report NISTIR-5935, National Institute of Standards and Technology, 1996. <http://math.nist.gov/MatrixMarket>. [70](#), [71](#), [72](#), [105](#)
- [9] The Boost-Community. *The boost C++ library*. [www.boost.org](http://www.boost.org), 2004. [13](#), [63](#)
- [10] B. J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *PETSc : portable, extensible toolkit for scientific computation*. Argonne national laboratory, 2011. [77](#)
- [11] E. Chu, A. George, J. Liu, and E. Ng. SPARSEPAK : Waterloo sparse matrix package. users' guide for SPARSEPAK-A. Technical report, Univ. Waterloo, Waterloo, Ontario, USA, 1984. [http://people.sc.fsu.edu/~jburkardt/f\\_src/sparsepak/sparsepak.html](http://people.sc.fsu.edu/~jburkardt/f_src/sparsepak/sparsepak.html). [77](#)
- [12] P. Ciarlet. *The finite element method for elliptic problems*. SIAM, Philadelphia, USA, second edition, 2002. [104](#)



- [13] P. G. Ciarlet. Basic error estimates for elliptic problems. In P. G. Ciarlet and J.-L. Lions, editors, *Handbook of numerical analysis. Volume 2. Finite element methods (part 1)*, chapter 2, pages 18–351. Elsevier, 1991. 100
- [14] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique. Cours, exercices et problèmes*. Dunod, Paris, third edition, 2010. 17, 24, 33
- [15] P. D’Alberto and A. Nicolau. Adaptive Strassen’s matrix multiplication. In *Proceedings of the 21st annual international conference on supercomputing*, pages 284–292, 2007. 24
- [16] T. A. Davis and I. S. Duff. An unsymmetgric-pattern multifrontal method for sparse LU factorization. Technical Report TR-94-038, University of Florida, 1994. 77
- [17] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the second object oriented numerics conference*, pages 214–218, 1992. <http://math.nist.gov/iml++>. 67
- [18] J. W. Eaton, D. Bateman, and S. Hauberg. *Octave : A high-level interactive language for numerical computations*. Free software foundation, 2011. <http://www.gnu.org/software/octave>. 19, 21
- [19] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package. I. the symmetric codes. Technical report, Yale Univ., 1977. 77
- [20] A. B. Fontaine. *La bibliothèque standard STL du C++*. Masson, Paris, 1997. 80
- [21] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, 2005. <http://www.fftw.org>. 33
- [22] A. George and J. W. Liu. *Computer solution of large sparse positive systems*. Prentice Hall, Englewood, New-Jersey, USA, 1981. 77
- [23] P. L. George and H. Borouchaki. *Triangulation de Delaunay et maillage. Applications aux éléments finis*. Hermès, Paris, 1997. 87
- [24] G. H. Golub and C. F. van Loan. *Matrix computations*. The John Hopkins University Press, Baltimore, MD, USA, third edition, 1996. 66
- [25] G. H. Golub and G. A. Meurant. *Résolution numérique des grands systèmes linéaires*. Eyrolles, Paris, 1983. 59
- [26] T. Granlund. *GMP : the GNU multiple precision arithmetic library*. Free software foundation, 2012. <http://gmplib.org>. 7
- [27] F. Hecht. *BAMG : bidimensional anisotropic mesh generator*, 2006. <http://www.rocq.inria.fr/gamma/cdrom/www/bamg/eng.htm>. 81, 87, 98, 106
- [28] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4) :14–21, 1985. 33
- [29] Y. Hida, X. S. Li, and D. H. Bailey. Library for double-double and quad-double arithmetic, 2008. <http://crd.lbl.gov/~dhbailey/mpdist>. 7, 13
- [30] D. E. Knuth. *The art of computer programming. Volume 1. Fundamental algorithms*. Addison Wesley, Reading, 1969. 24

- 
- [31] D. E. Knuth. *The art of computer programming. Volume 2. Seminumerical algorithms*. Addison Wesley, Reading, second edition, 1981. 24
  - [32] P. Lascaux and R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur. Tome 1*. Masson, Paris, 1986. 19, 34
  - [33] X. S. Li, J. W. Demmel, J. R. Gilbert, and M. Shao. *SuperLU user's guide*. Univ. of California, Berkeley, USA, 2010. 77
  - [34] G. Meurant. *Computer solution of large linear systems*. Elsevier, 1999. 59
  - [35] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide. C++ programming with the standard template library*. Addison Wesley, Reading, second edition, 2001. 47
  - [36] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C. The art of scientific computing*. Cambridge University Press, UK, 1994. <http://www.nrbook.com/a/bookcpdf.php>. 24
  - [37] P. A. Raviart and J. M. Thomas. *Introduction à l'analyse numérique des équations aux dérivées partielles*. Masson, Paris, 1983. 46, 104
  - [38] J.-F. Remacle and C. Geuzaine. Gmsh : a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, 2007. 87
  - [39] S. Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM news*, 38(9) :1–5, 2005. 24
  - [40] Y. Saad. *Sparskit : a basic tool-kit for sparse matrix computation*. Y. Saad, Minneapolis, USA, 1994. <http://www-users.cs.umn.edu/~saad>. 67, 77
  - [41] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, USA, second edition, 2000. 67, 77
  - [42] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, 2005. <http://trilinos.sandia.gov/packages/ifpack>. 67
  - [43] P. Saramito. *Efficient C++ finite element computing with Rheolef*. CNRS and LJK, 2011. <http://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef>. 46, 67, 77, 104
  - [44] W. Schroeder, K. Martin, and B. Lorensen. *The visualization toolkit : an object-oriented approach to 3D graphics*. Kitware, USA, third edition, 2002. 109, 111
  - [45] K. Shoemake. Animating rotation with quaternion curves. *SIGGRAPH*, 19(3) :245–254, 1985. 5, 13
  - [46] A. H. Squillacote. *Paraview user's guide*. Kitware, USA, 2011. <http://www.paraview.org>. 109, 111
  - [47] G. W. Stewart. Building an old-fashioned sparse solver. Technical report, University of Maryland, USA, 2003. 77
  - [48] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4) :354–356, 1969. 17, 24
  - [49] B. Stroustrup. *Le language C++*. Addison Wesley, Reading, special edition, 2001. 21, 80

- [50] P. N. Swarztrauber. *Vectorizing the FFTs*, pages 51–83. Academic Press, New York, 1982. [=http://www.netlib.org/fftpack](http://www.netlib.org/fftpack). 33
- [51] P. N. Swarztrauber, R. A. Sweet, and J. C. Adams. **fishpack** : efficient fortran subprograms for the solution of elliptic partial differential equations. Technical report, Univ. Corp. for Atmospheric research (UCAR), 1999. <http://www2.cisl.ucar.edu/resources/legacy/fishpack>. 59
- [52] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *proceedings of the IEEE*, 55(11) :1801, 1967. 77
- [53] T. veldhuisen. Blitz++ : a c++ class library for scientific computing, 2006. <http://blitz.sourceforge.net>. 13, 63
- [54] T. Veldhuizen. Techniques for scientific c++. Technical Report 542, Indiana University, Computer Science, 2000. 13
- [55] A.F. Ware. Fast approximate Fourier transforms for irregularly spaced data. *SIAM review*, 40(4) :838–856, 1998. 33
- [56] T. Williams and C. Keley. **gnuplot** : an interactive program, 2010. <http://www.gnuplot.info>. 57, 107, 110
- [57] O. C. Zienkiewicz and R. L. Taylor. *The finite element method. Volume 1. The basis*. Butterworth-Heinemann, Newton, MA, USA, fifth edition, 2000. 104

# Liste des fichiers d'exemples

<iostream>, 9  
<list>, 94  
<valarray>, 21, 40  
<vector>, 80  
<array>, 78  
<complex>, 6  
<list>, 94  
<valarray>, 21  
Makefile, 11  
a.mtx, 71  
as.mtx, 73  
assembly\_energy.h, 97  
assembly\_energy\_tst.cc, 97  
assembly\_mass.h, 91  
bang2mesh.awk, 107  
chi\_beta.h, 43  
coeff.h, 92–94  
data2gdat.cc, 58  
dirichlet.h, 99  
dmatrix.h, 23, 116  
element.h, 78, 79  
energy.h, 39  
eye.h, 63  
f\_alpha.h, 44  
fem2d.cc, 100  
fem3d\_vtk.cc, 112  
fem\_elevation.cc, 111  
fem\_gdat.h, 110  
fem\_vtk.h, 111  
fft.cc, 30  
fft.h, 29  
fft\_sin.h, 55  
fish1d.cc, 35  
fish1d\_adapt.cc, 44  
fish2d\_fft.cc, 57  
fish2d\_fft.h, 56  
guitar.cc, 30  
index.h, 55  
inplace\_factorize.h, 34  
inplace\_solve.h, 35  
interpolate.h, 89  
interpolate\_1d.h, 37  
interpolate\_2d.h, 48  
mass.h, 38  
matrix.h, 68–70, 72, 73, 96, 106, 121–124  
matrix\_cg\_tst.cc, 72  
matrix\_diag\_mult.cc, 123  
matrix\_tst.cc, 70  
meas.h, 83, 84  
membrane.cc, 101  
mesh.cc, 108  
mesh.h, 80, 81  
mesh3d.cc, 109  
mesh\_gdat.h, 108  
mesh\_hmax.h, 129  
mesh\_meas.h, 128  
mesh\_quality.h, 129, 130  
mesh\_tst.cc, 83  
mesh\_vtk.h, 109  
mtx2plot.cc, 106  
mtx\_sort.sh, 105  
pass1.h, 96  
pass2.h, 97  
point.h, 79, 85  
pow\_linear.cc, 15  
pow\_recursive.cc, 16  
promote.cc, 114  
promote.h, 113, 114  
quaternion.h, 6–10

quaternion.tst.cc, [10](#)  
range.h, [22](#)  
renumbering.h, [89](#)  
row\_col.h, [55](#)  
sinusprod.h, [99](#)  
spvalarray.h, [125–128](#)  
square.cad, [107](#)  
square.mesh, [83](#)  
tridiag.h, [40](#), [41](#)  
tridiag\_ldlt.h, [41](#), [42](#)  
u\_alpha.h, [42](#)  
unique\_insert.h, [95](#)  
valarray\_util.h, [21](#)  
cg.h, [62](#)  
fem\_elevation.cc, [101](#)  
fish1d.cc, [46](#), [117](#)  
interpolate\_1d.h, [46](#), [117](#)  
mesh.cc, [83](#)  
mtx2plot.cc, [71](#), [98](#)

# Index

- adaptation de maillage, 46
- algorithme de Strassen pour la multi-  
plication matricielle, 17
- algorithme du gradient conjugué, 61, 100
- awk, langage de script, 107
  
- c++ standard 2011, 11, 78
- calcul distribué, 59
- callage de paramètres avec `gnuplot`, 117
- classe défine dans le livre
  - `element`, 78
  - `eye`, 63
  - `matrix`, 68, 96
  - `mesh`, 79
  - `point`, 79
  - `quaternion`, 6
  - `tridiag`, 40
- classe de la librairie standard
  - `array`, 78, 79
  - `complex`, 6
  - `istream`, 10
  - `list`, 94
  - `ostream`, 9
  - `valarray`, 20, 21, 40
  - `vector`, 80
- classe dérivée, 80
- classe-fonction, 42, 46, 102
- complexe, nombre, 5
- condition aux bord de type Dirichlet
  - non homogène, 102
- condition aux bord de type Neumann, 103
- constructeur de copie, 6
- constructeur et conversion implicite de type, 6
- constructeur par défaut, 6
- convergence, 45, 100
- conversion de type défini dans une classe, 76
- conversion implicite de type, 6
- coût en temps de calcul, 100
  
- degrés de liberté, 88, 89
  
- espace de nom `std::`, 7
  
- factorisation  $LDL^T$ , 34
- factorisation de Choleski, 19
- fonction
  - `assembly_energy`, 97
  - `assembly_mass`, 90
  - `cg`, 62
  - `dirichlet`, 98
  - `interpolate`, 37, 47, 88
  - `range`, 21
  - `renumbering`, 89
- formule de quadrature
  - formule des trapèzes, 37, 90
  
- généricité, 34, 62
  
- héritage, 80
  
- implémentation sans matrice, 63
  
- logiciel
  - `atlas`, 24
  - `awk`, 107
  - `bamg`, 81, 87, 98, 106
  - `blitz++`, 13, 63

- boost/promote, 13
  - boost/ublas, 63
  - boost, 23
  - cgal, 87
  - fftpack, 33
  - fftw, 33
  - gmp, 7
  - gmsh, 87
  - gnuplot
    - fit, 117
  - gnuplot, 57, 98, 105, 107, 110
  - iml++, 67
  - lapack, 23, 24
  - make, 11
  - modulef, 104
  - octave, 21
  - opengl, 13
  - paraview, 109, 111
  - qd, 7, 13
  - rheolef, 67, 77, 104
  - sh, 105
  - sparsepak, 77
  - sparskit, 67, 77
  - stl, 47
  - superlu, 77
  - trilinos/ifpack, 67
  - umfpack, 77
  - yale, 77
- maillage, 78, 88, 98
- adapté, 43
- matlab, langage : voir octave, 21
- matrice
- conditionnement, 64
  - creuse, 67, 90, 98
    - classe `matrix`, 68
  - de Toeplitz, 53
  - format de fichier *matrix-market*
    - '`.mtx`', 70, 105
  - identité, classe `eye`, 63
  - préconditionnement, 63
    - classe `eye`, 63
  - structure bande, 22, 46
  - triangulaire inférieure, 34
  - tridiagonale, 34, 53
    - factorisation  $LDL^T$ , 34
    - valeurs et vecteurs propres, 50
    - valeur propre, 64
    - à diagonale dominante, 74
    - élémentaire, 90
- matrice dense, 23
- matrice tridiagonale, 49
- méthode
- des différences finies, 33, 39
  - des volumes finis, 90
  - des éléments finis, 87
- méthode des éléments finis, 36
- méthode directe
- Choleski, 64
  - Gauss, 64
- méthode itérative de résolution, 63
- nom célèbre
- Choleski, 64
  - Dirichlet, 33, 102
  - Fourier, 24, 27, 52, 54
  - Frobenius, 74
  - Gauss, 64
  - Hamilton, 5
  - Neumann, 103
  - Poisson, 33, 47, 65, 87
  - Tchebychev, 51
  - éléments finis, 65
- norme de Frobenius, 74
- parallélisme : voir calcul distribué, 59
- polynôme
- de Tchebychev, 51
- problème d'évolution, 103
- promotion de type, 12
- préconditionnement, méthode itérative
- de résolution, 63
- préconditionneur, 67
- qualité d'un maillage, 86
- quaternion, nombre, 5
- rotation, 11
- shell unix, langage de script, 105
- sommet interne, 81, 89
- spécialisation de classe, 13
- spécialisation partielle de fonction, 84

sur place, opération qui modifie les  
données, [19](#), [34](#), [74](#), [75](#)

transformée

de Fourier rapide, [54](#)

en sinus, [54](#)

transformée de Fourier, [52](#)

transformée de Fourier rapide, [24](#), [27](#)

triangle

cercle circonscrit, [86](#)

cercle inscrit, [86](#)

vecteur creux, [75](#)

équation de la chaleur, [103](#)