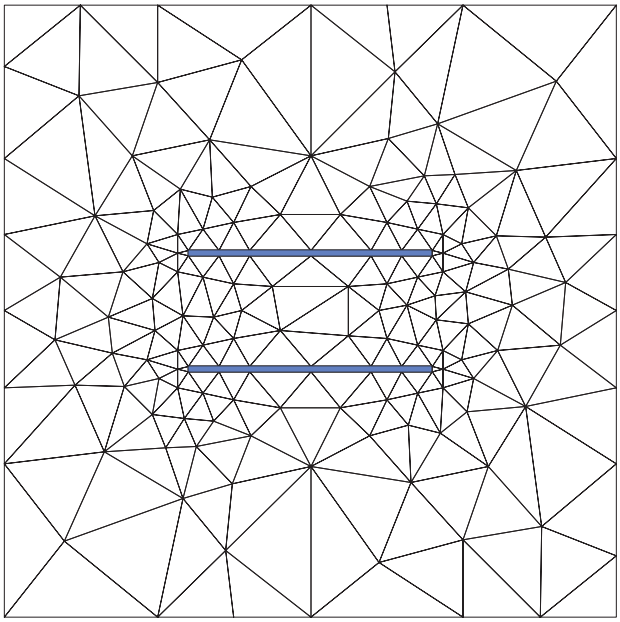
MÉTHODES NUMÉRIQUES et SIMULATIONS

PHQ 405



David Sénéchal

Département de physique
Faculté des sciences
Université de Sherbrooke

27 juillet 2011

Table des matières

Chapitre 1

Rappels de programmation scientifique

1.1 Approche numérique aux problème physiques

La description du monde physique repose sur plusieurs concepts représentés par des objets mathématiques dont la définition précise a demandé de longues réflexions aux mathématiciens des siècles passés : l'infini, les nombres réels et complexes, les fonctions continues, les distributions de probabilités, etc. Les principes de la physique sont généralement exprimés par des relations entre ces objets.

Dans plusieurs cas, ces objets mathématiques peuvent être manipulés symboliquement et les équations qui les gouvernent résolues analytiquement. Les modèles les plus simples de la physique se prêtent à ces calculs, et leur solution analytique permet de comprendre l'effet des divers paramètres impliqués. Notre compréhension de base de la physique doit donc énormément à notre capacité à résoudre exactement certains modèles simples.

Dans les cas plus réalistes les modèles ne peuvent être résolus analytiquement et tout un attirail de méthodes d'approximations analytiques a été développé, dans le but de conserver autant que possible les avantages d'une solution analytique, même approchée. La théorie des perturbations, que ce soit en mécanique classique ou quantique, est l'exemple le plus évident de méthode de calcul approchée.

Mais ces approches approximatives ont leur limites. Elles reposent généralement sur des hypothèses, telle la petitesse de certains paramètres, qui ne sont pas respectées en pratique. La vaste majorité des problèmes d'intérêt dans les sciences physiques se prêtent difficilement à une solution purement analytique, approximative ou non. On doit alors avoir recours à des méthodes numériques.

Une formation minimale sur les méthodes numériques est donc essentielle à toute personne s'intéressant à la modélisation du monde physique. En fait, l'expression «modélisation» est parfois utilisée pour signifier la description d'un système physique par un modèle qui ne peut être résolu que par des méthodes numériques.

Le premier problème rencontré en modélisation numérique est la représentation des objets

mathématiques courants (nombres, fonctions, etc.) sur un ordinateur. C'est donc par cet aspect que nous commencerons.

1.2 Représentation des nombres sur ordinateur

Un calculateur électronique représente les données (y compris les instructions visant à les manipuler) par un ensemble d'états électroniques, chacun ne pouvant prendre que deux valeurs, tel un commutateur qui est soit ouvert ou fermé. Chacun de ces systèmes abrite donc un atome d'information, ou bit, et l'état de chaque bit peut prendre soit la valeur 0, soit la valeur 1.

Le bit étant l'unité fondamentale d'information, l'*octet* (angl. *Byte*) est défini comme un ensemble de 8 bits et sert couramment d'unité pratique d'information. Traditionnellement, le Kilooctet (ko) désigne $2^{10} = 1024$ octets, le Mégaoctet (Mo) désigne 1024 ko, le Gigaoctet (Go) 1024 Mo, et ainsi de suite. [1] En 2011, la mémoire d'un ordinateur personnel typique se situe entre 10^9 et 10^{10} octets.

1.2.1 Nombres entiers

À partir des états binaires, un nombre entier naturel (l'un des concepts les plus simples, et probablement le plus ancien, des mathématiques) peut être représenté en base 2. Par exemple, on a la représentation binaire des nombres entiers suivants :

$$57 = 111001_2 \qquad 2532 = 100111100100_2 \tag{1.1}$$

Les entiers relatifs ($\mathbb{Z}$) sont représentés de la même manière, sauf qu'un bit supplémentaire est requis pour spécifier le signe ($\pm$). Bien évidemment, une quantité donnée d'information (un nombre donné de bits) ne peut représenter qu'un intervalle fini de nombre entiers. Un entier naturel de 4 octets (32 bits) peut donc prendre les valeurs comprises de 0 à $2^{32} - 1 = 4\,294\,967\,295$. Un entier relatif peut donc varier entre $-2\,147\,483\,648$ et $2\,147\,483\,647$.

Les opérations élémentaires sur les entiers (addition, multiplication, etc.) ne sont donc pas fermées sur ces entiers : ajouter 1 à l'entier naturel maximum redonne la valeur 0. Les opérations sont effectuées modulo la valeur maximale admissible. Il est donc important se s'assurer que les entiers manipulés dans un code soient toujours en-deça des bornes maximales permises si on désire qu'ils se comportent effectivement comme des entiers.

1. Le système international (SI) préconise plutôt de réserver ces préfixes aux puissances de 1000, et recommande d'utiliser le symbole ko pour 1000 octets, et plutôt le symbole kio pour 1024 octets, et pareillement pour les préfixes supérieurs.

1.2.2 Nombres à virgule flottante

La représentation binaire des nombres réels pose un problème plus complexe. On introduit le concept de *nombre à virgule flottante* (NVF) pour représenter de manière approximative un nombre réel. Un NVF comporte un *signe*, une *mantisse* et un *exposant*, comme suit :

$$x \to \pm b_p b_{p-1} \dots b_1 \times 2^{\pm e_q e_{q-1} \dots e_1} \tag{1.2}$$

où les b_i sont les bits de la mantisse et les e_i ceux de l'exposant binaire. Au total, $p+q+2$ bits sont requis pour encoder un tel nombre (2 bits pour les signes de la mantisse et de l'exposant).

Il a fallu plusieurs années avant qu'un standard soit développé pour les valeurs de p et q en 1987, fruit d'une collaboration entre l'*Institute of Electrical and Electronics Engineers* (IEEE) et l'*American National Standards Institute* (ANSI). Ce standard, appelé IEEE 754, prend la forme suivante :

$$x \to \pm 1.f \times 2^{e-\text{dcal.}} \tag{1.3}$$

où f est la partie fractionnaire de la mantisse et où un décalage est ajouté à l'exposant e. L'exposant e comporte 8 bits, de sorte que $0 \leq e \leq 255$, et le décalage, pour des nombres à 32 bits (précision simple) est 127. La mantisse comporte, elle, 23 bits, qui représentent la partie fractionnaire f de la mantisse. Par exemple, dans l'expression binaire à 32 bits

$$\underbrace{0}_{\text{signe}} : \underbrace{01111100}_{\text{exposant}} : \underbrace{01000000000000000000000}_{\text{mantisse}} \tag{1.4}$$

le signe est nul (donc $+$), l'exposant est $124-127=-3$, et la mantisse est $1,01_2 = 1,25$. Le nombre représenté est donc $+1.25 \times 2^{-3} = 0.15625$.

Signalons que des subtilités se produisent lorsque l'exposant est nul ($e=0$) ou maximum ($e=255$), mais nous n'entrerons pas dans ces détails ici. [2]

Un NVF de 32 bits peut effectivement représenter des nombres réels compris entre 1.4×10^{-45} et 3.4×10^{38}, avec 6 ou 7 chiffres significatifs ($23 \times \log_{10}(2) \approx 6.9$). Un tel nombre est qualifié de nombre à *précision simple*.

La précision simple est généralement insuffisante pour les applications scientifiques. On utilise plutôt la *précision double*, basée sur une représentation à 64 bits (8 octets) des NVF : l'exposant est codé en 11 bits et la mantisse en 52 bits, ce qui permet de représenter effectivement des nombres réels compris entre

$$4.9 \times 10^{-324} < \text{double précision} < 1.8 \times 10^{308} \tag{1.5}$$

avec 15 chiffres significatifs en base 10.

2. Voir par exemple les pages sur IEEE 754 dans Wikipedia.

Dépassements de capacité

Comme la représentation en virgule flottante ne représente qu'un intervalle de nombres possibles sur l'ensemble des réels, certaines opérations sur ces nombres produisent des nombres qui sont trop grands ou trop petits pour être représentés par un NVF. C'est ce qu'on appelle un *dépassement de capacité* (*overflow* ou *underflow*, en anglais). Un nombre trop grand pour être représenté par un NVF est plutôt représenté par le symbole `nan` (pour *not a number*). Ce symbole est aussi utilisé pour représenter le résultat d'opérations impossibles sur les réels, comme par exemple la racine carrée d'un nombre négatif. Les nombres trop petits sont généralement remplacés par zéro.

1.2.3 Erreurs d'arrondi

Les NVF représentent exactement un sous-ensemble des réels, en fait un sous-ensemble des nombres rationnels, ceux qui s'expriment exactement en base 2 par une mantisse de 52 bits et un exposant de 11 bits (pour une nombre à double précision). Cet ensemble n'est pas fermé sous les opérations arithmétiques habituelles (addition, multiplication, inversion). L'erreur ainsi générée dans la représentation des réels et dans les opérations arithmétique est qualifiée d'*erreur d'arrondi*.

Exemple 1.1: Erreur d'arrondi dans une addition simple

Voyons comment l'erreur d'arrondi se manifeste dans l'opération simple

$$7 + 1 \times 10^{-7} \tag{1.6}$$

en précision simple. Chacun des deux termes s'exprime, en binaire, comme suit :

$$\begin{aligned} 7 &= 0 : 10000001 : 11000000000000000000000 \\ 1 \times 10^{-7} &= 0 : 01100111 : 10101101011111100101001 \end{aligned} \tag{1.7}$$

Expliquons : l'exposant du nombre 7 est 129, moins le décalage de 127, ce qui donne 2. La mantisse est $1 + 2^{-1} + 2^{-2} = \frac{7}{4}$, et donc on trouve bien $\frac{7}{4} \times 2^2 = 7$. Pour le deuxième nombre, l'exposant est $103 - 127 = -24$ et la mantisse est $1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + \cdots = 1.6777216$, ce qui donne $1.6777216 \times 2^{-24} = 1.0 \times 10^{-7}$.

Additionnons maintenant ces deux nombres. Pour ce faire, on doit premièrement les mettre au même exposant, ce qui se fait en déplaçant les bits du deuxième nombre de 26 positions vers la droite, ce qui fait disparaître toute la mantisse. Le deuxième nombre devient donc effectivement nul, et on trouve

$$7 + 1.0 \times 10^{-7} = 7 \quad \text{(simple précision)} \tag{1.8}$$

On définit la *précision-machine* comme le nombre le plus grand qui, ajouté à 1, redonne 1. Pour un nombre à précision simple, précision-machine est de `5.96e-08`. Pour un nombre à double précision, elle est de `1.11e-16`.

1.3 Programmation par objets

Le contenu de ce cours est pour l'essentiel indépendant du langage de programmation utilisé pour réaliser les exemples et les travaux pratiques. Cependant, comme un choix doit être fait, nous utiliserons le C++, car il s'agit du langage de haut niveau général le plus utilisé. De plus, il permet d'écrire des programmes performants sans avoir recours à des modules codés dans un langage différent, à la différence d'un langage interprété, comme Python par exemple. Par contre, le C++ est plus complexe et plus strict. Nous supposerons que le lecteur est familier avec les bases du C++. Certaines de ses caractéristiques «intermédiaires» seront cependant discutées dans ce cours. [3]

Le C++ est une extension du langage C, développé entre 1969 et 1973 en même temps que le système d'exploitation UNIX, écrit justement dans ce langage. Le C++ a été développé à partir de 1979 ; sa raison d'être est la possibilité de construire des *objets*, c'est-à-dire des structures de données originales qui permettent de bien isoler certaines données du reste du programme. Cela facilite énormément la conception de programmes complexes et de librairies, et surtout la réutilisation et l'extension de ces programmes par d'autres auteurs.

Dans ce cours, nous appliquerons la programmation par objet au calcul scientifique. L'un des objectifs du cours est justement d'acquérir un style de programmation plus robuste et plus adapté à notre siècle.

1.3.1 Types fondamentaux en C++

Les types de variables fondamentaux en C++ sont indiqués dans le tableau 1.1. Chaque type entier peut être précédé du mot-clé `unsigned`, qui définit plutôt un entier naturel, dont le domaine s'étend de 0 à $2^n - 1$, n étant le nombre de bits contenu dans le type en question.

Un *pointeur* est un type général décrivant l'adresse en mémoire d'une donnée quelconque. Par exemple, pour une variable `X`, l'expression `p = &X` représente l'adresse en mémoire de la variable `X` ; l'opérateur `&` est appelé *opérateur d'adresse*. la valeur de `X` est alors représentée par l'expression `*p`. L'opérateur `*` placé immédiatement devant un nom de variable de pointeur effectue une *indirection*, c'est-à-dire représente les données situées à l'adresse contenue dans le pointeur. La définition et manipulation des pointeurs est très courante en C, mais moins fréquente en C++, où la tendance est de passer des objets par référence dans les fonctions ; sous le capot, cependant, passer des références ou des pointeurs revient au même.

Le langage C++ de base contient aussi la possibilité de définir des tableaux d'objets, en dimension 1, 2 ou plus. Par exemple, un tableau de n `double` se déclare (et la mémoire requise est allouée en même temps) par

```
double X[n];
```

3. Le site http://www.cplusplus.com est une mine extrêmement utile d'information et devrait figurer parmi les signets de tout programmeur.

Table 1.1 Types simples en C++, sur une machine à 64 bits

Nom	type	octets
`bool`	logique	1
`char`	caractère	1
`short`	entier	2
`int`	entier	4
`long`	entier	8
`float`	NVF	4
`double`	NVF	8
`long double`	NVF	16
	pointeur	8

Il est cependant conseillé de ne pas utiliser ces tableaux simples, mais plutôt des types complexes pour stocker un ensemble de données. L'avantage des types complexes est qu'on peut encapsuler dans le type la taille du tableau et surcharger des opérateurs pour effectuer des opérations sur ces structures (comme l'addition des vecteurs, par exemple). Cela allège le code et diminue les risques d'erreur.

1.3.2 Exemple d'objet : vecteurs en trois dimensions

Les objets en C++ sont généralement définis comme des classes (`class`). Ces notes de cours ne peuvent se substituer à un manuel complet de C++. Nous ne donnerons donc, dans ce qui suit, que quelques explications fragmentaires sur les classes.

Le code ci-dessous est un exemple de définition de classe décrivant des vecteurs en trois dimensions. Nous allons expliquer ce code en se référant aux numéros de lignes indiquées à gauche.

Important : *dans tous les exemples de code cités dans ces notes, les commentaires apparaissent en bleu, sans les symboles de commentaires requis en C++ : Dans un code réel, tous les commentaires doivent suivre le symbole ou être inscrits entre /* et */. Les codes cités dans ces notes sont en fait traités par un module LaTeX qui présentent les commentaires ainsi, afin d'en faciliter la lecture.*

Code 1.1 : Classe de vecteurs en 3D : `Vector3D.h`

```
1 #ifndef VECTOR3D_H
2 #define VECTOR3D_H
3 #include <iostream>
4 #include <cmath>
5
6 using namespace std;
```

```
7
8  class Vector3D
9    {
10   public:
11     double x; composante en x
12     double y; composante en y
13     double z; composante en z
14
15     constructeur
16     Vector3D() : x(0), y(0), z(0) {}
17
18     constructeur
19     Vector3D(const double &_x, const double &_y, const double &_z = 0)
20       : x(_x), y(_y), z(_z) {}
21
22     assignation
23     Vector3D& operator=(const Vector3D& q)
24     {x = q.x; y = q.y; z = q.z; return(*this);}
25
26     addition
27     Vector3D operator+(const Vector3D& q) {return(Vector3D(x+q.x,y+q.y,z+q.z));}
28
29     soustraction
30     Vector3D operator-(const Vector3D &q) {return(Vector3D(x-q.x,y-q.y,z-q.z));}
31
32     multiplication par une constante
33     Vector3D operator*(const double &c) {return(Vector3D(c*x,c*y,c*z));}
34
35     norme au carré
36     inline double norm2() {return(x*x+y*y+z*z);}
37
38     norme
39     inline double norm() {return(sqrt(norm2()));}
40
41     friend std::ostream & operator<<(std::ostream &flux, const Vector3D &x){
42       flux <<"(" <<x.x << ", " << x.y << ", " << x.z << ")";
43       return flux;
44     }
45
46     friend std::istream & operator>>(std::istream &flux, Vector3D &x){
47       flux >> x.x >> x.y >> x.z;
48       return flux;
49     }
50   };
51
52 produit scalaire
53 double operator*(const Vector3D &A, const Vector3D &B)
54 {
55   return(A.x*B.x + A.y*B.y + A.z*B.z);
```

```
56 }
57
58 Vector3D vector_product(const Vector3D &A, const Vector3D &B)
59 {
60   return Vector3D(A.y*B.z - A.z*B.y, A.z*B.x - A.x*B.z, A.x*B.y - A.y*B.x);
61 }
62
63 #endif
```

1. Une classe comporte des *membres*, qui peuvent être soit des données (des types simples ou d'autres classes ou structures préalablement définies) ou des fonctions (ou *méthodes*). Ces membres sont énumérés (déclarés ou définis) à l'intérieur d'une déclaration de type `class` (lignes 8–61 du code).
2. L'objet `Vector3D` comporte trois données : les trois composantes cartésiennes du vecteur, notées `x`, `y` et `z`. Chacune est un NVF à double précision (`double`).
3. Chaque classe comporte un ou plusieurs *constructeurs* qui, comme le nom l'indique, sont des méthodes visant à construire l'objet suite à une déclaration. La ligne 16 est le constructeur par défaut, qui ne prend aucun argument, et qui initialise les membres à zéro. La ligne 19 est un constructeur qui prend comme argument les trois composantes du vecteur. Le nom des constructeurs est celui de la classe elle-même. Ainsi, si on veut déclarer un objet noté `R` de type `Vector3D` et l'initialiser à la valeur $(1,0,0)$, on doit énoncer
   `Vector3D R(1.0,0,0);`
4. Les lignes 23-24 définissent une méthode d'assignation, qui permet d'assigner le contenu d'un objet à un autre. Par exemple :
   ```
   Vector3D R1, R2(1.0,0.0,0.0);
   R1 = R2;
   ```
   Notez que le mot-clé `this` représente un pointeur vers l'objet, et donc `*this` représente l'objet lui-même.
5. Les lignes 26-27 définissent une méthode d'addition qui permet de donner son sens naturel à l'expression `R1+R2`. L'un des avantages du C++ est la possibilité de surcharger les opérateurs existants. Ainsi, la définition du symbole + est ici étendue à une fonction effectuant l'addition des vecteurs. Notez que l'argument est passé en référence (`&q` et non `q`). Des méthodes semblables sont définies pour la soustraction et la multiplication par un scalaire. Le fait de définir ces surcharges dans la classe elle-même permet de considérer que l'une des cibles de l'opérateur surchargé est l'objet courant, l'autre cible étant donnée en argument. Ainsi, lorsque le compilateur rencontre l'expression `R1+R2`, il interprète l'opérateur + comme un méthode attachée à l'objet `R1`, qui prend comme argument l'objet `R2`. Il est aussi possible de définir la surcharge de l'opérateur + en dehors de la définition de classe, avec le même effet, de la manière suivante :

   ```
   1 Vector3D operator+(const Vector3D &p, const Vector3D &q){
   ```

```
2     return(Vector3D(p.x+q.x,p.y+q.y,p.z+q.z));
3  }
```

6. La fonction `norm2()` retourne la norme au carré du vecteur. Elle est déclarée `inline`, de sorte qu'elle ne sera pas vraiment compilée comme une sous-routine, mais intégré directement au code, ce qui améliore la performance dans fonctions qui sont rapides à évaluer, en éliminant le passage de contrôle entre deux routines. La fonction `norm()` retourne la norme elle-même. Comme elle invoque la fonction mathématique `sqrt()`, l'entête `<cmath>` doit être incluse (ligne 4).
7. Les opérateurs de flux << et >> sont surchargés (lignes 41–50). Ceci permet de lire ou d'écrire un vecteur comme si c'était un objet simple. Ces définitions nécessitent d'inclure l'entête `<iostream>` (ligne 3).
8. En dehors de la définition de la classe, des routines sont définies pour le produit scalaire de deux vecteurs (lignes 53–56) ainsi que pour le produit vectoriel (lignes 58–61).
9. Notez qu'une condition de pré-compilation est introduite à la ligne 1 pour s'assurer que le fichier d'entête `Vector3D.h` ne soit lu qu'une seule fois par compilation, même s'il est inclus dans plusieurs fichiers source du programme.

Nous utiliserons les classes à profusion dans ce cours. Il est donc important de bien comprendre ces concepts. Comme des exemples seront donnés pour chaque application, il n'est cependant pas nécessaire de tout comprendre à l'avance.

Test de l'utilisation de la classe Vector3D

Le fichier `Vector3D.h` ne constitue pas en soit un code utilisable, car il ne comporte pas de programme principal (`main()`). Le code ci-dessous, contenu dans un fichier séparé `Vector3D.cpp` contient le programme principal effectuant un test des fonctionnalités de la classe. Notez que le code de la classe est inclus par l'énoncé
`#include "Vector3D.h"`
et que les valeurs des vecteurs sont entrées via le terminal en utilisant la surcharge de l'opérateur de flux >>.

Code 1.2 : Programme principal utilisant la classe `Vector3D`

```
1  #include <iostream>
2  #include "Vector3D.h"
3  using namespace std;
4
5  int main() {
6
7    Vector3D A, B;
8
9    cout << "Test de la classe Vector3D" << endl;
10   cout << "Inscrire le vecteur A:";
11   cin >> A;
```

```
12    cout << "Inscrire le vecteur B:";
13    cin >> B;
14
15    cout << "Somme A+B = " << A+B << endl;
16    cout << "Multiplication par un scalaire: 2*A = " << A*2.0 << endl;
17    cout << "Produit scalaire: A . B = " << A*B << endl;
18    cout << "Produit vectoriel: A x B = " << vector_product(A,B) << endl;
19  }
```

1.3.3 Avantages de la programmation par objet

Résumons ici les principaux avantages de la programmation par objet. Ces avantages ne sont pas confinés au monde du génie logiciel, mais servent aussi le calcul scientifique.

1. Un objet bien défini suit la logique interne d'un problème et apparaît donc naturellement. Il permet de manipuler des concepts dans un code (par exemple, des vecteurs) sans être encombré des diverses données reliées à l'objet, qui sont dissimulées dans l'objet lui-même.
2. La surcharge des opérateurs permet de manipuler les types complexes (les objets) comme s'ils étaient des types simples.
3. Le *polymorphisme* est la capacité de définir plusieurs fonctions du même nom, mais comportant des types d'arguments différents. Le compilateur reconnaît les fonctions appropriées à leurs arguments. Cette caractéristique du C++ est étroitement associée aux objets, mais en est en fait indépendante. Elle permet de désigner par le même nom ou symbole des opérations ou fonctions analogues. La surcharge d'opérateur en est un cas particulier.

Nous allons adopter un style de programmation dans lequel les fichier d'entêtes (`*.h`) contiendront l'essentiel du code, alors que les fichiers `*.cpp` contiendront surtout des *pilotes* (angl. *drivers*) pour les différentes méthodes définies dans les fichiers d'entête.

1.3.4 Foncteurs

Un foncteur est une structure qui permet facilement de passer une fonction en argument à une routine, tout en contrôlant les paramètres en jeu dans la fonction. Par exemple, supposons qu'on souhaite introduire une fonction linéaire $f(x) = ax + b$ pour la passer en argument à une autre fonction qui, par exemple, calcule une intégrale définie. La façon classique de procéder est d'écrire

```
1 double f(double x, double *params){ return(params[0]*x+params[1]);}
2
3 double integrate(double (*F)(double x, double *params), double x1, double x2
      ){
```

```
4 intègre la fonction F de x=x1 à x=x2
5 ...
6 }
```

Dans cette façon de faire, on doit passer à la fonction `integrate()` les paramètres de la fonction `F`, ce qui nous oblige à écrire un fonction `f` qui prend ces paramètres comme argument, mais à l'intérieur d'un tableau de longueur indéfinie : donner une longueur définie à ce tableau nous forcerait à écrire une routine `integrate` différente pour chaque cas. En revanche, ne pas inclure les paramètres comme argument nous forcerait à recompiler la fonction `f` à chaque fois que les valeur des paramètres changent, ou à traiter les paramètres comme des variables globales, ce qui laisse à désirer du point de vue organisation. Notons qu'ici le tableau `params` contient les deux paramètres a et b en positions 0 et 1, respectivement.

Au lieu de procéder ainsi, on peut introduire une structure comme suit :

```
1  struct func{
2    double a;
3    double b;
4    func(double _a, double _b) : a(_a), b(_b) {} constructeur
5    double operator(double x){return(a*x+b);} surcharge de l'opérateur ()
6  }
7
8  template<class T>
9  double integrate(T &f, double x1, double x2){
10 intègre la fonction F de x=x1 à x=x2 en invoquant f(x)
11 ...
12 }
13
14 func F(0.1,3.0); instance du foncteur
15 double result = integrate(F,0,10); appel de la routine d'intégration
```

Expliquons :

1. la structure `func` a comme membres les paramètres de la fonction désirée, un constructeur qui permet d'initialiser ces paramètres, ainsi qu'un surcharge de l'opérateur `( )` qui permet d'évaluer la fonction à l'aide du nom de l'instance de la structure, comme par exemple `f(x)`. Généralement, les foncteurs sont à usage unique : une seule instance de la structure est requise dans le code.
2. Une instance `F` de la structure est déclarée et initialisée dans la partie principale du code.
3. La routine d'intégration est un *modèle de fonction* (angl. *function template*), qui est donc compilé séparément pour chaque type `T` utilisé dans le contexte du modèle.

L'avantage d'utiliser un foncteur plutôt que la méthode classique décrite plus haut est la localisation des données : les paramètres de la fonction n'ont pas besoin d'être introduits soit comme un tableau auxiliaire de longueur indéterminée, soit comme des variables globales. Notons cependant que les deux approches peuvent coexister et utiliser la même routine d'intégration

`integrate(...)` si elle est définie à partir d'un modèle comme ci-haut.

Exercice 1.1

A Mettez en place le programme `Vector3D.cpp` dans l'environnement de programmation de votre choix. Assurez-vous que le code compile correctement et exécutez-le.

B Écrivez une méthode de la classe `Vector3D`, intitulée `ortho(Vector3D &v)`, qui, étant donné un vecteur `v` dont la référence est donnée en argument, modifie l'objet pour lui enlever sa composante le long de `v`.

C Modifiez le code de la classe et celui du programme principal pour réaliser un modèle de classe qui puisse représenter des vecteurs dont les composantes sont d'un type quelconque (en fait entier, réel ou complexe). Apporter une attention particulière à la méthode qui calcule la norme. Vérifiez que le nouveau code fonctionne comme l'ancien dans le cas des réels.

1.4 Autres outils

1.4.1 Environnement de développement

Tout programmeur a avantage à tirer parti d'un gestionnaire de code, communément appelé *environnement de développement intégré*, ou IDE en anglais. Plusieurs solutions existent pour chaque plate-forme. [4] Aucune solution ne sera imposée dans ce cours, mais une seule sera supportée : Eclipse. Cet IDE existe sur Linux, Mac et Windows et est gratuit. Il est relativement complexe, mais stable, ce qui n'est pas encore le cas de `Code::Blocks` sur le Mac, par exemple. Ceci dit, le choix appartient à chacun. L'important est d'utiliser un environnement de développement efficace, ou aucun si on préfère éditer les fichiers de code avec un éditeur de texte sensible à la syntaxe et compiler les programmes à l'aide de `make`.

1.4.2 Serveurs de calcul

Sur un serveur de calcul standard, un code est compilé, exécuté, mais rarement développé. Il est donc courant qu'un code développé sur un ordinateur personnel, soit ensuite migré vers un serveur de calcul, où il est compilé et ensuite exécuté. Un serveur de calcul est mis à la disposition des étudiants inscrits à ce cours :
`phq405.physique.usherbrooke.ca`
Sur ce serveur, il est possible de compiler de vastes projets à l'aide d'un outil (make) qui permet de gérer plusieurs fichiers source. Cet outil est utile même pour des codes plus modestes, tels ceux qui seront réalisé dans le cadre de ce cours. Un modèle de fichier `makefile` est disponible sur le site du cours

4. Voir http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

1.4.3 Visualisation des données

La visualisation des données est un aspect crucial du calcul scientifique, mais dont l'ampleur varie beaucoup d'une discipline à l'autre. Elle prend plus de place en siences de la Terre et en génie qu'en physique théorique, par exemple. Dans ce cours, nous nous contenterons d'un outil très modeste, mais tout de même assez puissant pour nos besoins : gnuplot. [5] En plus d'appliquer gnuplot à des données pré-calculées, nous l'utiliserons aussi pour visualiser des données en cours de calcul, à l'aide d'une interface entre gnuplot et les codes que nous développerons : la librairie `gnuplot_i` [6]

5. http://www.gnuplot.info/
6. http://ndevilla.free.fr/gnuplot/

1.5 Annexe : code pour les vecteurs et matrices

1.5.1 Classe de vecteurs de longueur quelconque

Nous utiliserons dans ce cours une classe appelée `Vector` pour stocker des tableaux linéaire d'une longueur quelconque. Il y a plusieurs façon de définir des tableaux en C++ :

1. La plus élémentaire consiste à utiliser les éléments du langage lui-même. Par exemple, un tableau de 100 `double` intitulé `X` serait déclaré de la manière suivante :
   ```
   double X[100];
   ```
   La mémoire nécessaire au tableau sera allouée au début de la portée de l'énoncé (c'est-à-dire l'ensemble lignes de code contenues entre les délimiteurs { et }) et libérée à la fin de la portée. Le désavantage de cette méthode est que la longueur du tableau n'est pas contenue dans le tableau lui-même, et doit être stockée séparément : le tableau n'est pas un objet autonome. Une variante de cette méthode consiste à définir un pointeur et à en allouer la mémoire dynamiquement ainsi :
   ```
   double *X;
   X = new double[100];
   ```
   Dans ce cas, on ne doit pas oublier de libérer la mémoire lorsqu'elle n'est plus requise :
   ```
   delete[] X;
   ```
   Cette variante n'est utile que si l'existence du tableau est requise à l'extérieur d'une portée bien définie.

2. On peut utiliser la librairie des modèles standards (*Standard Template Library*, ou STL). Cette librairie définit un éventail assez complet de *conteneurs*, c'est-à-dire de structures souples qui permettent de stocker des objets selon plusieurs modèles : tableaux linéaires, listes chaînées, arbres, maps, etc. Pour un tableau simple, on utiliserait le modèle de classe `vector<double>` :
   ```
   #include <vector>
   vector<double> X(100);
   ```
   L'avantage de cette classe est que la taille du tableau est comprise dans l'objet – accessible par l'appel à la méthode `X.size()` – et qu'il est facile d'ajouter des éléments ou d'en retrancher. Bref, elle se prête bien aux tableaux dont la longueur est incertaine au moment de l'allocation de la mémoire, ou variable. Un désavantage de cette approche est que la classe modèle `vector` est tellement générale, qu'elle ne contient pas des méthodes élémentaires utiles pour les tableaux de nombres (produit scalaire, multiplication par un scalaire, etc) et qu'elle est légèrement moins efficace.

3. L'approche que nous suivrons est de définir notre propre classe de vecteurs, comme dans le code ci-dessous (voir les explications après le listage).

Code 1.3 : Classe `Vector.h`

```
#ifndef Vector_H
#define Vector_H

#include <cstdlib>
#include <cstring>
#include <cassert>
#include <iostream>

#define BOUND_CHECK

using namespace std;

inline double conj(double z) {return z;}

classe modèle pour un tableau
template<class T>
class Vector{
public:
  constructeur par défaut
  Vector(): n(0), v(0L){}

  constructeur d'un vecteur à un nombre donné d'éléments
  Vector(int the_size){Alloc(the_size);}

  constructeur par copie
  Vector(const Vector<T> &x){
    Alloc(x.n);
    memcpy(v,x.v,n*sizeof(*v));
  }

  destructeur
  ~Vector(){ Free();}

  taille
  inline int size() const {return n;}

  accès au tableau (utiliser uniquement si on ne peut faire autrement)
  inline T* array() const {return v;}

  alloue la mémoire
  void Alloc(int the_size){
    n = the_size;
    v = (T *)calloc(n,sizeof(T));
    assert(v);
  }

  opérateur d'assignation
  const Vector<T>& operator=(const Vector<T> &x){
    if(this==&x) return *this; évite les boucles infinies
```

```
50      if(n==0) Alloc(x.n);
51      int small = (x.n < n)? x.n : n;
52      memcpy(v,x.v,small*sizeof(*v)); copie de x à *this un maximum de n données
53      return *this;
54    }
55
56    met à zéro
57    void clear(){ memset(v,0,n*sizeof(*v));}
58
59    accès aux éléments (membre de droite)
60    T& operator[](int i){
61  #ifdef BOUND_CHECK
62      assert(i>=0 and i<n);
63  #endif
64      return v[i];
65    }
66
67    accès aux éléments (membre de gauche)
68    const T& operator[](int i) const {
69  #ifdef BOUND_CHECK
70      assert(i>=0 and i<n);
71  #endif
72      return v[i];
73    }
74
75    multiplication par un scalaire
76    Vector<T>& operator*=(const T &c)
77    {
78      for(int i=0; i<n; i++) v[i] *= c;
79      return *this;
80    }
81
82    ajoute un vecteur
83    Vector<T>& operator+=(const Vector<T>&x)
84    {
85  #ifdef BOUND_CHECK
86      assert(n == x.n);
87  #endif
88      for(int i=0; i<n; i++) v[i] += x.v[i];
89      return *this;
90    }
91
92    soustrait un vecteur
93    Vector<T>& operator-=(const Vector<T>&x)
94    {
95  #ifdef BOUND_CHECK
96      assert(n == x.n);
97  #endif
98      for(int i=0; i<n; i++) v[i] -= x.v[i];
```

```
99      return *this;
100   }
101
102   ajoute un nombre à toutes les composantes
103   Vector<T>& operator+=(const T x)
104   {
105     for(int i=0; i<n; i++) v[i] += x;
106     return *this;
107   }
108
109   soustrait un nombre de toutes les composantes
110   Vector<T>& operator-=(const T x)
111   {
112     for(int i=0; i<n; i++) v[i] -= x;
113     return *this;
114   }
115
116   ajoute un vecteur fois un nombre
117   Vector<T>& mult_plus(const Vector<T> &x, T a){
118     for(int i=0; i<n; i++) v[i] += a*x.v[i];
119     return *this;
120   }
121
122   échange avec un autre vecteur
123   void swap(Vector<T> &x)
124   {
125     assert(x.n == n);
126     assert(v and x.v);
127     T *tmp;
128     tmp = v;
129     v = x.v;
130     x.v = tmp;
131   }
132
133   retourne la norme au carré
134   double norm2(){
135     double z = 0.0;
136     for(int i=0; i<n; i++){
137       z += v[i]*v[i];
138     }
139     return z;
140   }
141
142   retourne la norme de la composante maximale du vecteur
143   double max(){
144     double z = 0.0;
145     for(int i=0; i<n; i++) if(abs(v[i]) > z) z = abs(v[i]);
146     return z;
147   }
```

```
148
149     insère un autre vecteur à la position offset
150     void insert(const Vector<T> &x, int o1, int o2){
151       int m1 = n-o1; place restante dans le vecteur courant
152       int m2 = x.n-o2; place restante dans le vecteur x
153       int m = (m1 < m2)? m1:m2;
154       memcpy(&v[o1],&x.v[o2],m*sizeof(*v));
155     }
156
157     Imprime dans un flux de sortie
158     friend std::ostream & operator<<(ostream& flux, const Vector<T>& x){
159       for(int i=0; i<x.size(); i++) flux << x[i] << "\t";
160       return flux;
161     }
162
163     Lit à partir d'un flux d'entrée
164     friend std::istream& operator >> (istream& flux, Vector<T>& x){
165       for(int i=0; i<x.size(); i++) flux >> x[i];
166       return flux;
167     }
168
169     déclarations seules
170     void ortho(Vector<T> &A);
171     void ortho(Vector<T> &V, Vector<T> &W);
172
173   private:
174
175     int n; nombre de composantes
176     T *v; pointeur vers le tableau des valeurs
177
178     libère la mémoire
179     void Free(){
180       free(v);
181       v = 0;
182       n = 0;
183     }
184
185   };
186
187   addition (opérateur binaire)
188   template <class T>
189   inline Vector<T> operator + (const Vector<T>& x, const Vector<T>& y){
190     Vector<T> tmp(x);
191     tmp += y;
192     return tmp;
193   }
194
195   multiplication par un scalaire (opérateur binaire)
196   template <class T>
```

```
197 inline Vector<T> operator * (const Vector<T>& x, const T &a){
198   Vector<T> tmp(x);
199   tmp *= a;
200   return tmp;
201 }
202
203 produit scalaire (opérateur binaire)
204 template <class T>
205 inline T operator * (const Vector<T>& x, const Vector<T>& y){
206   T z = 0;
207   for(int i=0; i<x.size(); i++) z += conj(x[i])*y[i];
208   return z;
209 }
210
211 soustrait la projection du vecteur courant le long d'un vecteur donné
212 template <class T>
213 void Vector<T>::ortho(Vector<T> &A){
214   T proj = (A*(*this))/(A*A);
215   mult_plus(A,-proj);
216 }
217
218 soustrait la projection du vecteur courant (X) sur V le long du vecteur W: X -= (X*V)W
219 template <class T>
220 void Vector<T>::ortho(Vector<T> &V, Vector<T> &W){
221   T proj = (V*(*this));
222   mult_plus(W,-proj);
223 }
224
225 #endif
```

Expliquons maintenant certaines caractéristiques de cette classe :

1. Il s'agit d'un modèle de classe (angl. *class template*). La ligne 17 signifie que la classe qui suit n'est pas précisément définie, mais constitue plutôt un modèle ou le caractère `T` est remplacé par un type quelconque au moment de la déclaration (ceci s'applique aussi à la STL). Pour déclarer un vecteur de 100 `double`, il faut alors écrire `Vector<double> X(100);`

2. Les données de la classe sont la longueur `n` du tableau et un pointeur `v` vers les éléments du tableau. Ces données sont déclarées `private`, ce qui entraîne qu'elles ne sont accessibles qu'aux méthodes définies dans la classe elle-même ; en particulier, écrire
   `Vector<double> X(100);`
   `cout << X.n;`
   causerait une erreur de compilation. Pour accéder à la taille du vecteur, il faut plutôt écrire `X.size()`. Cette précaution empêche les utilisateurs de la classe de modifier les données cruciales de la classe par accident. Ceci dit, nous ne suivrons généralement pas cette pratique dans les classes que nous définirons dans ce cours, même si elle est recom-

mandée pour des programmes plus complexes. Notons aussi qu'on peut avoir accès au pointeur `v` via la méthode `array()`.

3. Le tableau `v[]` lui-même est alloué dynamiquement, dans la méthode `Alloc()`. Cette méthode utilise la routine C standard `calloc()` déclarée dans l'entête `<cstdlib>`. Si l'allocation ne réussit pas, c'est-à-dire si la mémoire est insuffisante, le pointeur retourné par `calloc()` est nul. On peut alors vérifier que l'allocation a réussi en invoquant la fonction `assert(v)`, qui termine le programme si son argument est nul (ou logiquement faux). Nous utiliserons la fonction `assert()` à profusion dans ce cours, afin d'effectuer un traitement minimal des exceptions. Un traitement plus poussé des exceptions utiliserait plutôt les énoncés `try`, `catch` et `throw` de C++. Lorsque le constructeur `Vector<T>(_size )` est invoqué, l'allocation de la mémoire est effectuée.

4. L'accès aux éléments du tableau se fait comme dans un tableau simple : l'opérateur `[...]` a été surchargé pour pointer vers la composante désirée du tableau. On peut ainsi accéder à la composante `i` du tableau `X` par l'expression `X[i]`, à la fois pour la lire ou la modifier (c'est-à-dire à droite ou à gauche de l'opérateur d'assignation = .

5. La classe contient une précaution contre l'accès par erreur au-delà des limites du tableau. Si le mot-clé `BOUND_CHECK` est défini (comme dans le cas du listage), alors une vérification est effectuée avant chaque accès, ce qui certainement nuit à la performance, mais est très utile lors du développement.

6. La méthode `Free()` libère la mémoire associée au tableau et remet à zéro la taille du vecteur ainsi que le pointeur `v`. Elle est invoquée lors de la destruction de l'objet.

7. Pour déclarer un vecteur de nombres complexes à double précision dans un code, il faut s'assurer d'inclure l'entête `<complex>`, et ensuite écrire une déclaration du genre `Vector<complex<double> > X(n)` (`n` étant préalablement assigné à une valeur entière). Le modèle de classe `complex` s'applique à des entiers et des NVF quelconques. Les fonctions `real(z)`, `imag(z)`, `conj(z)` ont leur signification évidente. [7]

8. Notons la fonction `double conj(double z)` définie au début du fichier, qui constitue en fait la fonction identité, mais qui est nominalement supposée retourner le conjugué complexe d'un nombre réel. Cette fonction apparemment inutile a été définie afin de surcharger la fonction `complex<double> conj(complex<double> z)` afin qu'elle s'applique également aux nombre réels de type `double`, afin de pouvoir écrire un modèle de classe applicable à la fois aux nombres complexes et réels.

9. Les autres méthodes sont assez simples à comprendre.

1.5.2 Annexe : classe de matrices

7. Voir http://www.cplusplus.com/reference/std/complex/

Description

Le modèle de classe `Matrix` listé en annexe permet de manipuler des matrices pleines avec surcharge d'opérateurs pour l'accès aux éléments, l'addition des matrices, la multiplication de deux matrices ou la multiplication par une constante. L'application sur un vecteur est codée dans la méthode `mult_add()`. Les opérations d'inversion ou de solution d'un système linéaire sont effectuées en faisant appel à la librairie LAPACK (voir ci-dessous).

Les éléments de matrice sont stockés dans un tableau linéaire `v[]` de dimension `r*c`, `r` étant le nombre de rangées et `c` le nombre de colonnes. L'élément `(i,j)` (rangée `i` et colonne `j`) de la matrice est en position `i+r*j` de ce tableau. L'indice de rangée est celui qui change le plus rapidement dans le tableau : la matrice est parcourue de haut en bas, en ensuite de gauche à droite : on dit qu'elle est ordonnée par colonnes (angl. *column-major order*)). L'opérateur `()` a été surchargé pour donner accès aux éléments : on accède à l'élément `(i,j)` de la matrice `A` par l'appel `A(i,j)`.

La méthode `column(i)` produit un vecteur (décrit par le modèle de classe `Vector<>`) qui contient la colonne `i` de la matrice. De même, `row(i)` produit la rangée `i`.

Interface avec LAPACK

La méthode `Inverse()` de la classe remplace la matrice par son inverse, en invoquant la librairie LAPACK. Cette librairie est très largement utilisée pour un vaste éventail d'opérations matricielles ; il est fortement conseillé de l'utiliser plutôt que d'écrire soi-même le code nécessaire, ou même de compiler un code source de tierce partie, car la librairie est optimisée pour le processeur utilisé, parfois en insérant des portions de code écrites en assembleur au lieu d'un langage de haut niveau.

LAPACK a été initialement écrit en FORTRAN, et le nom des routines est un peu cryptique : le préfixe de chaque routine dépend du type de nombre à virgule flottante qui constitue la matrice : `s` pour les réels en simple précision, `d` pour les réels en double précision, `c` pour les complexes en simple précision et `z` pour les complexes en double précision. Une nomenclature est également suivie selon les types de matrices : `ge` pour les matrices générales, `sy` pour les matrices symétriques, `he` pour les matrices complexes hermitiques, etc. Ainsi, la routine effectuant la factorisation LU d'une matrice réelle générale constituée de réels en double précision s'intitule `dgetrf_()` : le `d` initial pour les réels en double précision, le `ge` suivant pour signifier une matrice générale et le `trf` suivant indique une factorisation (`f`) en matrices triangulaires (`tr`). Le souligné final est commun à toutes les routines initialement compilées en FORTRAN et invoquées à partir d'un code C++.

Voyons par exemple comment la routine `Inverse()` est construite pour une matrice de réels à double précision. On commence par effectuer une décomposition LU à l'aide de la routine LAPACK `dgetrf_()` (voir la section sec :matrices pour une explication de la décomposition LU) :

```
dgetrf_(&M, &N,(doublereal*)v, &LDA, IPIV,&INFO);
```

où

1. `M` et `N` sont les nombres de rangées et de colonnes (identiques ici).

2. `v` est le tableau des éléments de matrice ordonnée par colonnes.
3. `LDA` est `max(1,M)` (c'est-à-dire `M` dans notre cas).
4. `IPIV` est un tableau de la position des pivots, de dimension `min(M,n)`, qui doit être préalloué.
5. `INFO` est un entier contenant un code d'erreur à la sortie de la routine. Si ce code est zéro, c'est que le calcul s'est effectué sans erreur.
6. En sortie, le tableau `v[]` contient la factorisation LU, disposée comme une matrice. Les éléments diagonaux de L, qui valent 1, ne sont pas stockés.

Ensuite, on fait appel à la routine `dgetri_()`, qui procède à l'inversion proprement dite de la factorisation LU :

```
dgetri_(&N,(doublereal*)v,&LDA,IPIV,WORK,&N,&INFO);
```

où

1. `N=M` est l'ordre de la matrice.
2. `v[]` est le tableau calculé par `dgetrf_()`.
3. `LDA` est comme ci-haut.
4. `IPIV` est le tableau des pivots calculé par `dgetrf_()`.
5. `WORK` est un espace de travail préalloué, dont la taille est donnée par l'argument suivant (ici `N`).
6. `INFO` a le même rôle que dans la routine `dgetrf_()`, ou que dans toutes les routines LAPACK.

Code 1.4 : Classe de matrices `Matrix`

```
#ifndef Matrix_H
#define Matrix_H

#include <complex>
#include "Vector.h"

extern "C"{
#include "f2c.h"
#include "clapack.h"
}

using namespace std;

template<class T>
class Matrix
{
public:

   constructeur par défaut
   Matrix(): r(0), c(0), v(0L){}
```

```
21
22    constructeur d'une matrice carrée
23    Matrix(int _r){
24      r = c = _r;
25      alloc();
26    }
27
28    constructeur d'une matrice rectangulaire
29    Matrix(int _r, int _c){
30      r = _r;
31      c = _c;
32      alloc();
33    }
34
35    constructeur par copie
36    Matrix(const Matrix<T> &A){
37      r = A.r;
38      c = A.c;
39      alloc();
40      memcpy(v,A.v,r*c*sizeof(*v));
41    }
42
43    destructeur
44    ~Matrix(){
45      Free();
46    }
47
48    taille
49    inline int rows() const {return r;}
50    inline int columns() const {return c;}
51
52    accès au tableau (utiliser uniquement si on ne peut faire autrement)
53    inline T* array() const {return v;}
54
55    allocateur
56    void Alloc(int _r, int _c){
57      Free();
58      r = _r;
59      c = _c;
60      alloc();
61    }
62    void Alloc(int _r){Alloc(_r,_r);}
63
64    opérateur d'assignation (si les deux matrices sont de mêmes dimensions)
65    const Matrix<T>& operator=(const Matrix<T> &A){
66      if(this==&A) return *this;
67      else if(r == 0 or c == 0){
68        Alloc(A.r, A.c);
69        memcpy(v,A.v,r*c*sizeof(*v));
```

```
70      }
71      else if(r == A.r and c == A.c) memcpy(v,A.v,r*c*sizeof(*v));
72      else{
73        assert(false);
74      }
75      return *this;
76    }
77
78    accès aux éléments (membre de droite)
79    inline const T& operator()(const int &i, const int &j)const{
80    #ifdef BOUND_CHECK
81      assert(i>=0 and i<r);
82      assert(j>=0 and j<c);
83    #endif
84      return(v[i+r*j]);
85    }
86
87    accès aux éléments (membre de gauche)
88    T& operator()(const int &i, const int &j){
89    #ifdef BOUND_CHECK
90      assert(i>=0 and i<r);
91      assert(j>=0 and j<c);
92    #endif
93      return(v[i+r*j]);
94    }
95
96    met à zéro
97    void clear(){
98      memset(v,0,r*c*sizeof(*v));
99    }
100
101   Vérifie l'hermiticité
102   bool is_hermitian(){
103     for(int i=0; i<r; i++){
104       for(int j=0; j<=i; j++) if(v[i+r*j] != conj(v[j+r*i])) return false;
105     }
106     return true;
107   }
108
109   addition d'une matrice
110   template <class S>
111   Matrix<T>& operator+=(const Matrix<S> &A){
112   #ifdef BOUND_CHECK
113     assert(v);
114   #endif
115     int rc = r*c;
116     for(int i=0; i<rc; i++) v[i] += A.v[i];
117     return *this;
118   }
```

```
119
120   ajoute une constante à la diagonale
121   void add_to_diagonal(T a){
122     for(int i=0; i<r; i++) v[i+r*i] += a;
123   }
124
125   ajoute un vecteur à la diagonale
126   void add_to_diagonal(Vector<T> a){
127     for(int i=0; i<r; i++) v[i+r*i] += a[i];
128   }
129
130   pré−multiplie par une matrice diagonale contenue dans a
131   void mult_diagonal(Vector<T> a){
132     assert(a.size() == r);
133     for(int k=0; k<c; k++) for(int i=0; i<r; i++) v[i+r*k] *= a[i];
134   }
135
136   extrait une colonne
137   Vector<T> column(int col){
138     assert(col < c and col >= 0);
139     Vector<T> x(r);
140     for(int i=0; i<r; i++) x[i] = v[i+r*col];
141     return x;
142   }
143
144   extrait une rangée
145   Vector<T> row(int row){
146     assert(row < r and row >= 0);
147     Vector<T> x(c);
148     for(int i=0; i<c; i++) x[i] = v[row+r*i];
149     return x;
150   }
151
152   ajoute A.x au vecteur y: y += A.x
153   void mult_add(const Vector<T> &x, Vector<T> &y){
154     for(int j=0; j<c; j++){
155       for(int i=0; i<r; i++) y[i] += v[i+j*r]*x[j];
156     }
157   }
158
159   ajoute A.x au vecteur y: y += A.x
160   void mult_add(T *x, T *y){
161     for(int j=0; j<c; j++){
162       for(int i=0; i<r; i++) y[i] += v[i+j*r]*x[j];
163     }
164   }
165
166   ajoute la matrice B fois une constante c
167   void mult_add(const Matrix<T> &B, T cst){
```

```
168       assert(r==B.rows() and c==B.columns());
169       for(int j=0; j<c; j++){
170         for(int i=0; i<r; i++) v[i+j*r] += cst*B(i,j);
171       }
172     }
173
174     inverse la matrice
175     void Inverse();
176
177     solutionne un système linéaire général A.x = u pour x
178     void solve(Vector<T> &x, const Vector<T> &u);
179
180     multiplication par un scalaire
181     inline Matrix<T> operator *= (T a){
182       int rc = r*c;
183       for(int i=0; i<rc; i++) v[i] *= a;
184       return *this;
185     }
186
187     insère une matrice B à la position r1,c1 de A, à partir de la position r2,c2 de B
188     void insert(const Matrix<T> &B, int r1, int c1, int r2, int c2){
189       int r3 = (r-r1 < B.r-r2)? r-r1 : B.r-r2;
190       int c3 = (c-c1 < B.c-c2)? c-c1 : B.c-c2;
191       for(int i=0; i<r3; i++){
192         for(int j=0; j<c3; j++){
193           (*this)(i+r1,j+c1) = B(i+r2,j+c2);
194         }
195       }
196     }
197
198     void eigenvaluesSym(Vector<T> &d);
199     void eigensystemSym(Vector<T> &d, Matrix<T> &U);
200     void eigenvalues(Vector<complex<double> > &d);
201     void eigensystem(Vector<complex<double> > &d, Matrix<T> &U);
202     void generalized_eigensystem(Matrix<T> &B, Vector<T> &d, Matrix<T> &U);
203
204   private:
205
206     int r; !< nombre de rangées
207     int c; !< nombre de colonnes
208     T *v; !< tableau de valeurs (lues du haut vers le bas)
209
210     alloue la mémoire
211     void alloc(){
212       v = (T *)calloc(r*c,sizeof(T));
213       assert(v);
214     }
215
216     libère la mémoire
```

```
217   void Free(){
218     free(v);
219     v = 0;
220     r = 0;
221     c = 0;
222   }
223 };
224
225 addition (opérateur binaire)
226 template <class T>
227 inline Matrix<T> operator + (const Matrix<T>& x, const Matrix<T>& y){
228   Matrix<T> tmp(x);
229   tmp += y;
230   return tmp;
231 }
232
233 multiplication par un scalaire (opérateur binaire)
234 template <class T>
235 inline Matrix<T> operator * (const Matrix<T>& x, const T &a){
236   Matrix<T> tmp(x);
237   tmp *= a;
238   return tmp;
239 }
240
241 multiplication de deux matrices : this = A * B
242 template <class T>
243 inline Matrix<T> operator * (const Matrix<T>& A, const Matrix<T>& B){
244 #ifdef BOUND_CHECK
245   assert(A.columns() == B.rows());
246 #endif
247   Matrix<T> C(A.rows(),B.columns());
248   for(int j = 0; j < C.columns(); j ++){
249     for(int i = 0; i < C.rows(); i++){
250       T z = 0.0;
251       for(int k = 0; k < A.columns(); k++) z += A(i,k)*B(k,j);
252       C(i,j) = z;
253     }
254   }
255   return C;
256 }
257
258 Imprime dans un flux de sortie
259 template <class T>
260 inline ostream& operator << (ostream& out, const Matrix<T>& x){
261   out << '{';
262   for(int i=0; i<x.rows(); i++){
263     out << '{' << x(i,0);
264     for(int j=1; j<x.columns(); j++) out << ", " << x(i,j);
265     out << "}";
```

```
266       if(i+1 < x.rows()) out << ",\n";
267     }
268     out << "}\n";
269     return out;
270   }
271
272   Lit à partir d'un flux d'entrée
273   template <class T>
274   inline istream& operator >> (istream& in, Matrix<T>& x){
275     for(int i=0; i<x.rows(); i++){
276       for(int j=0; j<x.columns(); j++) in >> x(i,j);
277     }
278     return in;
279   }
280
281   template<> void Matrix<complex<double> >::Inverse()
282   {
283     assert(r==c);
284     integer INFO, M , N , LDA;
285     integer *IPIV;
286     doublecomplex *WORK;
287
288     IPIV = new integer[r];
289     WORK = new doublecomplex[r];
290
291     N = r;
292     M = r;
293     LDA = r;
294
295     zgetrf_(&M, &N,(doublecomplex*)v, &LDA, IPIV,&INFO);
296     assert((int)INFO==0);
297     zgetri_(&N,(doublecomplex*)v,&LDA,IPIV,WORK,&N,&INFO);
298     assert((int)INFO==0);
299
300     delete[] WORK;
301     delete[] IPIV;
302   }
303
304   template<> void Matrix<double>::Inverse()
305   {
306     assert(r==c);
307     integer INFO, M , N , LDA;
308     integer *IPIV;
309     doublereal *WORK;
310
311     IPIV = new integer[r];
312     WORK = new doublereal[r];
313
314     N = r;
```

```
315    M = r;
316    LDA = r;
317
318    dgetrf_(&M, &N,(doublereal*)v, &LDA, IPIV,&INFO);
319    assert((int)INFO==0);
320    dgetri_(&N,(doublereal*)v,&LDA,IPIV,WORK,&N,&INFO);
321    assert((int)INFO==0);
322
323    delete[] WORK;
324    delete[] IPIV;
325  }
326
327  solutionne le système linéaire A.x = u
328  template<> void Matrix<double>::solve(Vector<double> &x, const Vector<double> &u)
329  {
330    assert(r==c);
331
332    integer N = r;
333    integer NRHS = 1;
334    doublereal A[r*c];
335    integer LDA = r;
336    integer IPIV[r];
337    integer LDB = r;
338    integer INFO;
339
340    x = u;
341    memcpy((void *)A,v,r*c*sizeof(*v));
342    dgesv_(&N, &NRHS, A, &LDA, IPIV, (doublereal*)(x.array()), &LDB, &INFO);
343    assert((int)INFO==0);
344
345  }
346
347  template<> void Matrix<double>::eigenvaluesSym(Vector<double> &d)
348  {
349    assert(r==c);
350
351    char JOBZ = 'N'; calcule les vecteurs propres aussi
352    char UPLO = 'L'; le triangle inférieure est stocké
353    integer N = (integer)rows();
354
355    Matrix<double> U(*this);
356
357    integer INFO=0;
358    integer LDA = (integer)rows();
359    integer LWORK = N*(N+2);
360    doublereal *WORK = new doublereal[LWORK];
361
362    dsyev_(&JOBZ, &UPLO, &N, (doublereal *)U.array(), &LDA, (doublereal *)d.array(),
           WORK, &LWORK, &INFO);
```

```
363
364   delete [] WORK;
365   assert((int)INFO==0);
366 }
367
368 template<> void Matrix<double>::eigensystemSym(Vector<double> &d, Matrix<double> &U)
369 {
370   assert(r==c);
371
372   char JOBZ = 'V'; calcule les vecteurs propres aussi
373   char UPLO = 'L'; le triangle inférieure est stocké
374   integer N = (integer)rows();
375
376   U = *this;
377   integer INFO=0;
378   integer LDA = (integer)rows();
379   integer LWORK = N*(N+2);
380   doublereal *WORK = new doublereal[LWORK];
381
382   dsyev_(&JOBZ, &UPLO, &N, (doublereal *)U.array(), &LDA, (doublereal *)d.array(),
          WORK, &LWORK, &INFO);
383
384   delete [] WORK;
385   assert((int)INFO==0);
386 }
387
388 template<> void Matrix<double>::eigenvalues(Vector<complex<double> >&d)
389 {
390   assert(r==c);
391
392   char JOBVR = 'N'; calcule les vecteurs propres aussi
393   char JOBVL = 'N'; calcule les vecteurs propres aussi
394   integer N = (integer)rows();
395
396   integer INFO=0;
397   integer LDA = (integer)rows();
398   integer LWORK = 10*N;
399   doublereal *WORK = new doublereal[LWORK];
400
401   integer LDVL = (integer)rows();
402   integer LDVR = (integer)rows();
403   doublereal WR[N];
404   doublereal WI[N];
405   doublereal VL[1];
406   doublereal VR[1];
407   doublereal A[N*N];
408   memcpy(A,v,N*N*sizeof(*A));
409
410   dgeev_(&JOBVL, &JOBVR, &N, A, &LDA, WR, WI, VL, &LDVL, VR, &LDVR, WORK, &LWORK, &
```

```
        INFO);
411   for(int i=0; i<d.size(); i++) d[i] = complex<double>(WR[i],WI[i]);
412   delete [] WORK;
413   assert((int)INFO==0);
414 }
415
416 template<> void Matrix<double>::eigensystem(Vector<complex<double> >&d, Matrix<
      double> &U)
417 {
418   assert(r==c);
419
420   char JOBVR = 'V'; calcule les vecteurs propres aussi
421   char JOBVL = 'N'; calcule les vecteurs propres aussi
422   integer N = (integer)rows();
423
424   U = *this;
425   integer INFO=0;
426   integer LDA = (integer)rows();
427   integer LWORK = 10*N;
428   doublereal *WORK = new doublereal[LWORK];
429
430   integer LDVL = (integer)rows();
431   integer LDVR = (integer)rows();
432   doublereal WR[N];
433   doublereal WI[N];
434   doublereal VL[1];
435
436   dgeev_(&JOBVL, &JOBVR, &N, (doublereal *)v, &LDA, WR, WI, VL, &LDVL, (doublereal
        *)U.array(), &LDVR, WORK, &LWORK, &INFO);
437   for(int i=0; i<d.size(); i++) d[i] = complex<double>(WR[i],WI[i]);
438   delete [] WORK;
439   assert((int)INFO==0);
440 }
441
442 template<> void Matrix<double>::generalized_eigensystem(Matrix<double> &B, Vector<
      double> &d, Matrix<double> &U)
443 {
444   assert(r==c);
445
446   char JOBZ = 'V'; calcule les vecteurs propres aussi
447   char UPLO = 'L'; le triangle inférieure est stocké
448   integer N = (integer)rows();
449
450   U = *this;
451   integer ITYPE = 1;
452   integer INFO=0;
453   integer LDA = (integer)rows();
454   integer LDB = LDA;
455   integer LWORK = N*(N+2);
```

```
456     doublereal *WORK = new doublereal[LWORK];
457
458     dpotrf_(&UPLO, &N, (doublereal *)B.array(), &LDA, &INFO);
459     assert((int)INFO==0);
460     dsygst_(&ITYPE, &UPLO, &N, (doublereal *)U.array(), &LDA, (doublereal *)B.array()
            , &LDB, &INFO);
461     assert((int)INFO==0);
462     dsyev_(&JOBZ, &UPLO, &N, (doublereal *)U.array(), &LDA, (doublereal *)d.array(),
            WORK, &LWORK, &INFO);
463     assert((int)INFO==0);
464     delete [] WORK;
465 }
466
467 #endif
```

Chapitre 2

Équations différentielles ordinaires

Ce chapitre est consacré à la solution des systèmes d'équations différentielles ordinaires, c'est-à-dire aux systèmes d'équations de la forme

$$\frac{\partial \mathbf{x}}{\partial t} = \mathbf{f}(\mathbf{x}, t) \tag{2.1}$$

où $\mathbf{x}(t)$ est une collection de N fonctions dépendant d'une variable indépendante t, et $\mathbf{f}$ est un ensemble de N fonctions de $\mathbf{x}$ et de t.

Remarques:

1. La variable t joue typiquement le rôle du temps en mécanique, d'où la notation utilisée. Mais elle peut en général avoir une interprétation différente.
2. La forme (2.1) peut sembler restrictive, car il s'agit d'un système du premier ordre en dérivées seulement. Cependant tout système d'équations différentielles d'ordre plus élevé peut être ramené à un système du type (2.1) en ajoutant des variables au besoin pour tenir la place des dérivées d'ordre sous-dominant. Par exemple, considérons une équation différentielle du deuxième ordre pour une variable y :

   $$\ddot{y} = f(y, \dot{y}, t) \tag{2.2}$$

   En définissant les variables $x_1 = y$ et $x_2 = \dot{y}$, cette équation peut être ramenée au système suivant :

   $$\dot{x}_2 = f(x_1, x_2, t) \qquad \dot{x}_1 = x_2 \tag{2.3}$$

   qui est bien de la forme (2.1) où $f_1 = f$ et $f_2(\mathbf{x}, t) = x_2$.
3. Le passage d'un système d'équations du deuxième ordre à M variables vers un système du premier ordre à $N = 2M$ variables est justement ce qui est réalisé en passant de la mécanique de Lagrange à la mécanique de Hamilton. Les équations du mouvement de Hamilton sont de la forme (2.1), mais leur structure symplectique est spécifique, alors que la forme (2.1) est plus générale.

2.1 Méthode d'Euler

La méthode la plus élémentaire pour solutionner numériquement l'équation différentielle (2.1) est la *méthode d'Euler*. Sa simplicité est cependant contrebalancée par son manque de précision et de stabilité. Elle consiste à remplacer l'équation (2.1) par une équation aux différences finies : l'axe du temps est remplacé par une suite d'instants également espacés $t \to t_n = nh$, $(n = 0, 1, 2 \ldots)$, où h est le *pas temporel*. La fonction $\mathbf{x}(t)$ est alors remplacée par une suite $\{\mathbf{x}_n\}$ et la dérivée est estimée par l'expression [1]

$$\dot{\mathbf{x}}(t) \approx \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} \tag{2.4}$$

Le système (2.1) est alors modélisé par l'équation aux différences suivante :

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n, t_n)} \tag{2.5}$$

Cette expression définit une carte explicite $\mathbf{x}_n \mapsto \mathbf{x}_{n+1}$ qui permet, par récurrence, d'obtenir la suite complète à partir des valeures initiales $\mathbf{x}_0$.

2.1.1 Précision de la méthode d'Euler

La principale source d'erreur de la méthode d'Euler provient du pas temporel h, qui doit être pris suffisamment petit. Cette erreur est qualifiée d'*erreur de troncature*. Si nous avions accès aux dérivées d'ordre arbitraire de la fonction $\mathbf{x}(t)$, nous pourrions envisager le développement de Taylor suivant :

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \left.\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}\right|_{t_n} h + \frac{1}{2}\left.\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}\right|_{t_n} h^2 + \cdots \tag{2.6}$$

Le remplacement (2.5) revient à négliger les termes en h^2 (ou plus grand) dans ce développement. On dit alors que la méthode d'Euler est du *premier ordre* en h.

2.1.2 Stabilité de la méthode d'Euler

Au-delà des considérations de précision, la carte (2.5) est aussi sujette à des erreurs d'arrondi. Supposons à cet effet qu'une telle erreur produise une déviation $\delta\mathbf{x}_n$ de la solution numérique, par rapport à la solution exacte de l'équation aux différences (2.5). La méthode sera *stable* si cette déviation décroît avec le temps et *instable* dans le cas contraire. Appliquons un développement limité à la carte (2.5) :

$$\delta\mathbf{x}_{n+1} = \delta\mathbf{x}_n + h\frac{\partial}{\partial x_\alpha}\mathbf{f}(\mathbf{x}_n, t_n)\delta x_{\alpha,n} \quad \text{ou} \quad \delta\mathbf{x}_{n+1} = (1 + h\mathbf{M})\,\delta\mathbf{x}_n \tag{2.7}$$

1. Les éléments du vecteur $\mathbf{x}_n$ sont notés $x_{\alpha,n}$, où $\alpha = 1, 2, \ldots, N$.

où la matrice $\mathbf{M}$ possède les composantes suivantes :

$$M_{\alpha\beta} = \frac{\partial f_\alpha}{\partial x_\beta} \tag{2.8}$$

et est évaluée au temps t_n.

De toute évidence, la méthode d'Euler sera instable si la matrice $(1 + h\mathbf{M})$ possède au moins une valeur propre en dehors de l'intervalle $(-1, 1)$, de manière répétée en fonction du temps. Dans le cas d'une seule variable ($N = 1$), la condition de stabilité revient à

$$-1 < 1 + h\frac{\mathrm{d}f}{\mathrm{d}x} < 1 \implies \frac{\mathrm{d}f}{\mathrm{d}x} < 0 \quad \text{et} \quad \left|\frac{\mathrm{d}f}{\mathrm{d}x}\right| < \frac{2}{h} \tag{2.9}$$

Exemple 2.1: Équation linéaire à une variable

Considérons l'équation

$$\dot{x} = \lambda x \tag{2.10}$$

dont la solution analytique est $x(t) = x(0)\mathrm{e}^{\lambda t}$. La méthode d'Euler produit l'équation aux différences suivante :

$$x_{n+1} = x_n + h\lambda x_n = (1 + h\lambda)x_n \tag{2.11}$$

L'analyse de stabilité produit la relation suivante pour les déviations :

$$\delta x_{n+1} = (1 + h\lambda)\delta x_n \tag{2.12}$$

ce qui montre que la méthode d'Euler est stable seulement si $\lambda < 0$ (décroissance exponentielle) et instable dans le cas d'une croissance exponentielle ($\lambda > 0$) ou d'une oscillation (λ imaginaire).

2.1.3 Méthode prédicteur-correcteur

Une façon simple d'améliorer la méthode d'Euler est de la scinder en deux étapes :

1. On procède à une *prédiction* sur la valeur au temps t_{n+1} : $\mathbf{x}^{\text{pr.}}_{n+1} = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n, t_n)$.
2. On procède ensuite à une *correction* de cette prédiction, en remplaçant la dérivée évaluée à $\mathbf{x}_n$ par la moyenne de la dérivée évaluée à $(\mathbf{x}_n, t_n)$ et $(\mathbf{x}^{\text{pr.}}_n, t_{n+1})$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \tfrac{1}{2}h\left(\mathbf{f}(\mathbf{x}_n, t_n) + \mathbf{f}(\mathbf{x}^{\text{pr.}}_n, t_{n+1})\right) \tag{2.13}$$

On montre que cette méthode, qui requiert manifestement deux fois plus d'évaluations de $\mathbf{f}$ que la méthode d'Euler simple, est cependant du deuxième ordre en h, c'est-à-dire que l'erreur de troncature est d'ordre h^3. Voyons la démonstration : on applique le développement limité

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left.\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}\right|_{t_n} + \tfrac{1}{2}h^2\left.\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}\right|_{t_n} + \mathcal{O}(h^3) \tag{2.14}$$

La dérivée première au temps t_n est précisément $\mathbf{f}(\mathbf{x}_n, t_n)$. La dérivée seconde au temps t_n est, au premier ordre,

$$\left.\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}\right|_{t_n} = \frac{\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) - \mathbf{f}(\mathbf{x}_n, t_n)}{h} + \mathcal{O}(h^2) \tag{2.15}$$

où $\mathbf{x}_{n+1}$ peut être connu au premier ordre seulement. En substituant dans le développement (2.14), on retrouve effectivement l'équation (2.13), qui est donc d'ordre h^2.

2.2 Méthode de Runge-Kutta

La méthode de Runge-Kutta est une amélioration notable de la méthode d'Euler, et constitue en fait une généralisation de la méthode prédicteur-correcteur. Elle consiste à évaluer la dérivée $\mathbf{f}$ non pas au temps t_n, ni au temps t_{n+1}, mais entre les deux en général.

2.2.1 Méthode du deuxième ordre

Commençons par démontrer la méthode de Runge-Kutta d'ordre 2. Nous allons supposer qu'une étape de la méthode prend la forme suivante :

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + a\mathbf{k}_1 + b\mathbf{k}_2 \\ \mathbf{k}_1 &= h\mathbf{f}(\mathbf{x}_n, t_n) \\ \mathbf{k}_2 &= h\mathbf{f}(\mathbf{x}_n + \beta\mathbf{k}_1, t_n + \alpha h) \end{aligned} \tag{2.16}$$

et nous allons chercher à déterminer les constantes a, b, α et β de manière à maximiser la précision de la méthode.

Pour déterminer ces paramètres, appliquons encore une fois le développement limité (2.14), en tenant compte du fait que

$$\frac{\mathrm{d}\mathbf{f}}{\mathrm{d}t} = \frac{\partial \mathbf{f}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x_i}\frac{\mathrm{d}x_i}{\mathrm{d}t} \tag{2.17}$$

En substituant la règle (2.16) dans le développement limité, on trouve

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + ah\mathbf{f}(\mathbf{x}_n, t_n) + bh\mathbf{f}\left(\mathbf{x}_n + h\beta\mathbf{f}(\mathbf{x}_n, t_n), t_n + \alpha h\right) + \mathcal{O}(h^3) \\ &= \mathbf{x}_n + (a+b)h\mathbf{f} + b\beta h^2 \frac{\partial \mathbf{f}}{\partial x_i}\frac{\partial f_i}{\partial t} + b\alpha h^2 \frac{\partial \mathbf{f}}{\partial t} + \mathcal{O}(h^3) \end{aligned} \tag{2.18}$$

où toutes les expressions sont évaluées à $(\mathbf{x}_n, t_n)$ dans la dernière ligne. Pour que cette dernière expression coïncide avec le développement de Taylor correct de $\mathbf{x}_{n+1}$, les paramètres a, b, α et β doivent respecter les contraintes suivantes :

$$a + b = 1 \qquad b\alpha = \tfrac{1}{2} \qquad b\beta = \tfrac{1}{2} \tag{2.19}$$

La solution à ces contraintes n'est pas unique. En particulier, le choix $a = b = \frac{1}{2}$ et $\alpha = \beta = 1$ correpond à la méthode prédicteur-correcteur !

Un autre choix possible est $a = 0$, $b = 1$ et $\alpha = \beta = \frac{1}{2}$, qui correspond à ce qui est généralement appelé la méthode de Runge-Kutta du deuxième ordre :

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}(\mathbf{x}_n, t_n) \\ \mathbf{k}_2 &= h\mathbf{f}(\mathbf{x}_n + \mathbf{k}_1/2, t_n + h/2) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{k}_2 + \mathcal{O}(h^3) \end{aligned} \tag{2.20}$$

2.2.2 Méthode du quatrième ordre

La version la plus utilisée de la méthode de Runge-Kutta est celle du quatrième ordre, dans laquelle l'erreur commise est d'ordre h^5. La formule aux différences correspondante est la suivante :

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(\mathbf{x}_n, t_n) \\
\mathbf{k}_2 &= h\mathbf{f}(\mathbf{x}_n + \mathbf{k}_1/2, t_n + h/2) \\
\mathbf{k}_3 &= h\mathbf{f}(\mathbf{x}_n + \mathbf{k}_2/2, t_n + h/2) \\
\mathbf{k}_4 &= h\mathbf{f}(\mathbf{x}_n + \mathbf{k}_3, t_n + h) \\
\mathbf{x}_{n+1} &= \mathbf{x}_n + \tfrac{1}{6}\mathbf{k}_1 + \tfrac{1}{3}\mathbf{k}_2 + \tfrac{1}{3}\mathbf{k}_3 + \tfrac{1}{6}\mathbf{k}_4 + \mathcal{O}(h^5)
\end{aligned} \tag{2.21}$$

Voyons le sens de chacune de ces étapes :

1. Pour commencer, on évalue la dérivée $\mathbf{k}_1/h$ au point $(\mathbf{x}_n, t_n)$.
2. On utilise cette dérivée pour obtenir un premier point médian $\mathbf{x}_n + \mathbf{k}_1/2$.
3. On calcule ensuite la dérivée $\mathbf{k}_2/h$ à ce point médian.
4. On calcule une deuxième estimation $\mathbf{x}_n + \mathbf{k}_2/2$ de ce point médian et on y calcule encore une fois la dérivée $\mathbf{k}_3/h$.
5. On calcule ensuite la dérivée $\mathbf{k}_4/h$ à une première estimation du point final $\mathbf{x}_n + \mathbf{k}_3$.
6. Enfin, le point final estimé par la méthode est obtenu par une combinaison des quatre dérivées calculées aux étapes précédentes.

2.2.3 Contrôle du pas dans la méthode de Runge-Kutta

Un solutionneur d'équations différentielle qui procède en suivant un pas temporel h constant est condamné soit à l'inefficacité (h trop petit), soit à commettre des erreurs de troncature non contrôlées (h trop grand). Il est impératif que le pas temporel s'adapte à chaque instant à l'erreur de troncature commise à chaque étape. Cette dernière peut être estimée en procédant à deux étapes de Runge-Kutta du quatrième ordre, chacune de pas $h/2$, et en comparant le résultat $\mathbf{x}_{n+1}$ obtenu après la deuxième étape à celui obtenu en procédant directement à une étape de pas h. Si la différence est inférieure à une précision demandée à l'avance, on peut alors augmenter la valeur de h pour la prochaine itération, sinon on peut la réduire.

Des calculs supplémentaires sont bien sûr nécessaires pour évaluer l'erreur de troncature, mais l'algorithme au total demande moins de calculs car son erreur est *contrôlée* : le pas h peut augmenter si l'erreur est tolérable, et de cette manière alléger le coût des calculs. D'un autre côté, si h doit être diminué, c'est que les calculs auraient autrement été erronés à une degré jugé inacceptable.

Une façon élémentaire de procéder au contrôle du pas est la méthode du *doublement* :

1. On effectue un pas temporel h à l'aide de la méthode du quatrième ordre (`RK4`). On obtient une variable mise-à-jour $\mathbf{x}_{n+1}^{(1)}$.
2. On effectue deux pas temporels successifs $h/2$, à partir du même point de départ, pour arriver à une valeur $\mathbf{x}_{n+1}^{(2)}$.

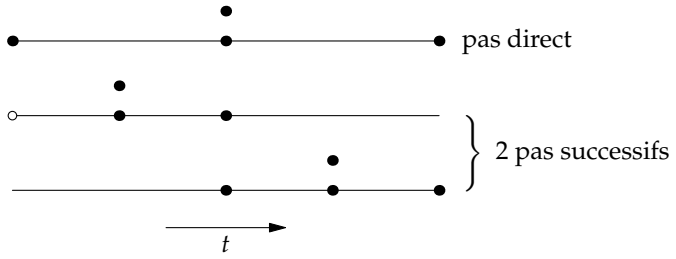


FIGURE 2.1 Nombre d'évaluation des dérivées dans une étape de la méthode de Runge-Kutta adaptative. Un cercle noir représente une évaluation des dérivées et un cercle blanc une dérivée qui a été gardée en mémoire.

3. La différence $|\mathbf{x}_{n+1}^{(2)} - \mathbf{x}_{n+1}^{(1)}|$ nous donne une estimation de l'erreur de troncature Δ_1. Nous pouvons par la suite modifier h (en l'augmentant ou le diminuant) pour viser une erreur de troncature constante Δ_0. Donc nous devons réajuster h ainsi :

$$h \to h \left(\frac{\Delta_0}{\Delta_1} \right)^{1/5} \tag{2.22}$$

   car nous savons que l'erreur doit se comporter comme h^5 dans une méthode du 4e ordre. On peut aussi multiplier par un facteur de sécurité (0.9 dans le code) afin de tempérer quelque peu l'augmentation de h, sans jamais faire plus que de quadrupler le pas.

4. On note que le nombre d'évaluations de la dérivée dans cette procédure est de 11 par étape, alors que la méthode utilisée avec un pas de $h/2$ (qui nous donne une précision semblable) nécessite 8 évaluations. Il y a donc un coût (surcharge) de $\frac{11}{8} = 1.375$ dans L'estimation de l'erreur. Cependant, ce coût est largement recouvré par le contrôle accru de la méthode.

Quoique la méthode du doublement soit simple à comprendre, elle est maintenant dépassée par la méthode dite de *Runge-Kutta-Fehlberg*, qui effectue une étape du 5e ordre en même temps qu'une étape du 4e ordre et compare les deux afin d'estimer l'erreur de troncature (la méthode est souvent désignée par son acronyme RKF45). Au total, le nombre d'évaluation nécessaire par étape est de 6, soit presque deux fois moins que la méthode de doublement. [2]

2.2.4 Code

La classe `ODE_solver` implante la méthode de Runge-Kutta au quatrième ordre (méthode `RK4`) et l'adaptation du pas temporel (méthode `RKadapt`). Notez que la fonction `deriv` qui calcule les dérivées est en fait un foncteur et constitue l'une des données de la classe.

Code 2.1 : Méthode de Runge-Kutta : `RK.h`

2. Voir *Numerical Recipes*, ou encore les articles de Wikipedia écrits sur le sujet.

```
#include <iostream>
#include <fstream>
#include <cmath>

#include "Vector.h"

using namespace std;

template <class T>
class ODE_solver{

public:
  int N; nombre de variables dépendantes
  T &deriv; référence à un foncteur
  double accur; précision
  double hnext; prochain pas
  double hdid; pas précédent
  static const double hmin = 1.0e-8; pas minimal

  ODE_solver(int _N, T &_deriv, double _accur, double _hnext) : N(_N), deriv(_deriv
      ), accur(_accur), hnext(_hnext) {}

  Effectue un pas adaptatif dans la méthode de Runge-Kutta
  t: variable indépendante
  x: variables dépendantes
  dx: changement dans les variables dépendantes suite au pas temporel
  bool RKadap(double t, Vector<double> &x, Vector<double> &dx){
    N = x.size(); nombre de variables
    double h,ts,h2,errmax;

    Vector<double> dxs(N); vecteurs temporaires
    Vector<double> xs(N);
    Vector<double> xt(N);

    deriv(t,x,dx); calcul des dérivées au temps t
    ts = t; copie du temps de départ t
    xs = x; copie de la valeur de départ de x_n
    dxs = dx; copie de la valeur de départ des dérivées
    h = hnext; valeur d'essai du pas
    while(true){
      h2 = h*0.5;
      RK4(ts,xs,dxs,xt,h2); demi-pas Runge-Kutta au temps t
      t = ts + h2;
      deriv(t,xt,dx); calcul des dérivées au temps t + h/2 et au point xt
      RK4(t,xt,dx,x,h2); demi-pas Runge-Kutta au temps t + h/2

       x est maintenant la valeur prédite par deux demi-pas
      t = ts + h; temps final
      if(h < hmin) return false; sort de la boucle si h est trop petit
```

```
49        RK4(ts,xs,dxs,xt,h); pas Runge−Kutta au temps t
50        xt *= -1.0; xt += x; xt = x − xt, différence des deux résultats
51        errmax = xt.max()/accur;
52        if(errmax <= 1.0){
53          hdid = h; valeur du pas effectué, suivie de l'augmentation du pas
54          hnext = (errmax > 0.0006 ? 0.9*h*exp(-0.20*log(errmax)) : 4.0*h);
55          break;
56        }
57        h = 0.9*h*exp(-0.2*log(errmax)); h → 0.9h(Δ0/Δ1)^1/5
58      }
59
60       correction de x (voir NR 2e edition): rend l'approximation d'ordre h^5 au lieu de h^4
61      x.mult_plus(xt,0.066666666); x += xt/15
62
63      return true;
64    }
65
66    Effectue un pas dans la méthode de Runge−Kutta au 2e ordre.
67    t: variable indépendante t [in]
68    x: x_n [in]
69    k1: k_1 [in]
70    xp: x_{n+1} [out]
71    h: pas temporel h [in]
72    void RK2(double t, Vector<double> &x, Vector<double> &k1, Vector<double> &xp,
          double h){
73
74      Vector<double> dxt(N); vecteurs temporaires
75
76      double h2 = h*0.5;
77      double th = t + h2;
78
79      xp = x + k1*h2; = x_n + (1/2)k_1
80      deriv(th,xp,dxt); dxt = f(x_n + (1/2)k_1, t_n + h/2) = k_2/h
81      xp = x + dxt*h; = x_n + (1/2)k_2
82    }
83
84    Effectue un pas dans la méthode de Runge−Kutta au 4e ordre (même arguments que RK2).
85    void RK4(double t, Vector<double> &x, Vector<double> &k1, Vector<double> &xp,
          double h){
86
87      Vector<double> xt(N); vecteurs temporaires
88      Vector<double> dxt(N);
89      Vector<double> dxm(N);
90
91      double h2 = h*0.5;
92      double th = t + h2;
93
94      xt = x + k1*h2; = x_n + (1/2)k_1
```

```
95      xp = xt*0.3333333333333333;
96      deriv(th,xt,dxt); dxt = f(x_n + 1/2 k_1, t_n + h/2) = k_2/h
97      xt = x + dxt*h2; = x_n + 1/2 k_2
98      xp += xt*0.6666666666666667;
99      deriv(th,xt,dxm); dxm = f(x_n + 1/2 k_2, t_n + h/2) = k_3/h
100     xt = x + dxm*h; = x_n + k_3
101     xp += dxm*h*0.3333333333333333;
102     deriv(t+h,xt,dxt); dxt = f(x_n + k_3, t_n + h) = k_4/h
103     xp += dxt*h*0.1666666666666667; x_{n+1} = x_n + 1/6 k_1 + 1/3 k_2 + 1/3 k_3 + 1/6 k_4
104   }
105
106   void solve_euler(double t0, Vector<double> &x, ostream &fout)
107   {
108     fout << 0.0 << '\t' << x << endl;
109     Vector<double> dx(N);
110     for(double t=0.0; t<t0; t += hnext){
111       deriv(t,x,dx);
112       x += dx*hnext;
113       fout << t << '\t' << x << endl;
114     }
115   }
116
117   void solve_RK2(double t0, Vector<double> &x, ostream &fout)
118   {
119     fout << 0.0 << '\t' << x << endl;
120     Vector<double> dx(N);
121     Vector<double> xp(N);
122     for(double t=0.0; t<t0; t += hnext){
123       deriv(t,x,dx);
124       RK2(t,x,dx,xp,hnext);
125       x = xp;
126       fout << t << '\t' << x << endl;
127     }
128   }
129
130   void solve_RK4(double t0, Vector<double> &x, ostream &fout)
131   {
132     fout << 0.0 << '\t' << x << endl;
133     Vector<double> dx(N);
134     Vector<double> xp(N);
135     for(double t=0.0; t<t0; t += hnext){
136       deriv(t,x,dx);
137       RK4(t,x,dx,xp,hnext);
138       x = xp;
139       fout << t << '\t' << x << endl;
140     }
141   }
142
```

```
143   void solve_RKadapt(double t0, Vector<double> &x, ostream &fout)
144   {
145     double t=0.0;
146     int nstep=0;
147     const int max_steps=10000;
148     Vector<double> dx(N);
149
150     deriv(t,x,dx);
151     while(RKadap(t,x,dx)){
152       t += hdid;
153       nstep++;
154       if(nstep > max_steps){
155         cout << "maximum number of steps exceeded!\n";
156         break;
157       }
158       if(t>t0) break;
159       fout << t << '\t' << x << endl;
160     }
161   }
162 };
```

Quelques explications sur ce code :

1. La classe est en fait un modèle de classe, dont l'argument est un *foncteur* qui permet de calculer les dérivées.
2. La classe comporte diverses méthodes-pilotes (commençant par le mot `solve`) qui mettent en oeuvre les méthodes écrites plus haut qui se chargent de chaque étape. Seule la méthode d'Euler, la plus simple, est codée entièrement dans `solve_euler`.
3. Les méthodes `RK2` et `RK4` ne contiennent pas le premier appel au foncteur qui calcule les dérivées. Ceci dans le but d'améliorer l'efficacité des codes adaptatifs qui devraient sinon recalculer les dérivées au temps t_n plusieurs fois. Il est donc nécessaire, dans les méthodes-pilotes, de faire appel au foncteur explicitement à chaque étape.

2.3 Exemple : Mouvement planétaire

2.3.1 Solution du problème de Kepler

Considérons premièrement un problème dont nous connaissons la solution analytique : celui du mouvement d'un corps de masse m dans le champ gravitationnel d'un objet de masse M, fixe à l'origine. Nous savons que le problème peut être limité à deux dimensions d'espace, en raison de la conservation du moment cinétique.

Adoptons les coordonnées cartésiennes (x, y), et soit (v_x, v_y) les vitesses correspondantes. Les équations du mouvement pour ce problème sont, en forme vectorielle,

$$\dot{\mathbf{r}} = \mathbf{v} \qquad \dot{\mathbf{v}} = -\frac{k}{r^3}\mathbf{r} \tag{2.23}$$

où nous avons introduit la constante $k = GM$. Nous pouvons toujours choisir un système d'unités pour le temps dans lequel cette constante vaut 1. Le système d'équations se ramène donc à l'ensemble suivant, si on numérote les variables dans l'ordre suivant : (x, y, v_x, v_y) :

$$\dot{x}_1 = x_3 \qquad \dot{x}_2 = x_4 \qquad \dot{x}_3 = -\frac{x_1}{r^3} \qquad \dot{x}_4 = -\frac{x_2}{r^3} \qquad r \stackrel{\text{def}}{=} (x_1^2 + x_2^2)^{1/2} \tag{2.24}$$

2.3.2 Code

Voici le code utilisé pour définir le problème de Kepler perturbé et l'appel aux méthodes de Runge-Kutta définies plus haut.

Code 2.2 : Méthode de Runge-Kutta : fichier maître pour la solution du problème de Kepler

```
#include <string>
#include "RK.h"
#include "read_parameter.h"

using namespace std;

Foncteur définissant le problème de Kepler perturbé
struct func_kepler{
  int count; incrément qui compte le nombre d'appels au foncteur
  func_kepler() {count=0;}
  void operator() (const double t, const Vector<double> &x, Vector<double> &dxdt){
    double r2 = x[0]*x[0]+x[1]*x[1]; = r^2
    double r = sqrt(r2);
    double z = 1.0/(r*r2); z = r^-3 + ar^-5
    dxdt[0] = x[2]; ẋ = v_x
    dxdt[1] = x[3]; ẏ = v_y
    dxdt[2] = -x[0]*z; v̇_x = -xz
    dxdt[3] = -x[1]*z; v̇_y = -yz
    count++;
  }
};

int main(){

  ifstream fin("in.dat"); ouverture du fichier d'entrée
  double accur; précision désirée
  double h; pas temporel initial
  double tmax; temps de simulation
  string method; méthode de calcul utilisée
```

```
30
31    fin >> "accur" >> accur; lecture des paramètres
32    fin >> "h" >> h;
33    fin >> "tmax" >> tmax;
34    fin >> "methode" >> method;
35
36    func_kepler deriv();
37    const int N=4;
38    ODE_solver<func_kepler> ode(N,deriv,accur,h);
39
40    Vector<double> x(N); valeurs initiales
41    fin >> "x0" >> x;
42    fin.close();
43
44    ofstream fout("out.dat"); ouverture du fichier de sortie
45    if(method=="euler") ode.solve_euler(tmax,x,fout); action selon la méthode choisie
46    else if(method=="RK2") ode.solve_RK2(tmax,x,fout);
47    else if(method=="RK4") ode.solve_RK4(tmax,x,fout);
48    else if(method=="RK_adapt") ode.solve_RKadapt(tmax,x,fout);
49    else{
50      cout << "Méthode inconnue!" << endl;
51      exit(1);
52    }
53    cout << method << ": " << deriv.count << " évaluations requises\n";
54    cout << "exécution terminée" << endl;
55  }
```

Explications :

1. On doit au début du code inclure le fichier d'entête `RK.h`, et ce dernier doit être placé dans un répertoire accessible (soit le même répertoire que le fichier maître, soit un des répertoire qui figure dans la liste des fichiers d'inclusion ; voir la documentation de l'IDE à cet effet).

2. Le code fait appel à une routine utilitaire définie dans `read_parameter.h` utilisée pour lire des mots-clés à partir d'un fichier d'entrée. L'énoncé
   `fin >> "nom_de_variable">>` *valeur*
   effectue la recherche de la chaîne de caractères `nom_de_variable` dans le fichier d'entrée (ici `in.dat`) et, une fois la chaîne trouvée, lit la variable qui suit immédiatement. On peut ainsi facilement récupérer la valeur de paramètres dans un fichier d'entrée en lisant des paires *nom*/*valeur*, et l'ordre dans lequel ces paires sont écrites dans le fichier d'entrée n'est pas important. Le fichier d'entrée peut aussi contenir d'autre chaînes de caractères qui ne sont jamais lues (des commentaires, etc.) sans dommage : c'est la première occurence de la chaîne recherchée qui compte.

2.4 Autres méthodes

La méthode de Runge-Kutta est couramment utilisée pour résoudre des systèmes d'équations différentielles. Peut-être est-elle même la plus souvent utilisée. Par contre, elle n'est pas nécessairement la plus précise ou la plus performante. Une façon ingénieuse d'augmenter la précision d'une méthode est de procéder, sur un intervalle h donné, à une subdivision en m sous-intervalles équidistants. On traite ensuite le problème pour chacun des sous-intervalles de dimension $\eta = h/m$, de manière à obtenir un prédicteur $\mathbf{x}(t+h)$ pour quelques valeurs de m (par exemple m =2, 4, 6) et on extrapole vers $m \to \infty$ (ou $\eta \to 0$). Cette méthode, dite de Richardson, suppose que le prédicteur $\mathbf{x}(t+h)$ est une fonction analytique de h autour de $h = 0$, et qu'une extrapolation, par exemple polynomiale, nous donne accès à cette limite.

On se trouve ici à travailler avec un pas h/m fixe dans un intervalle donné, mais un contrôle de précision peut aussi être appliqué subséquemment à la valeur de h elle-même, de manière à s'adapter à une précision requise.

Une façon simple de procéder avec m sous-intervalles est la méthode aux différences modifiées (angl. *modified midpoint method*), qui évalue la dérivée à partir de l'instant suivant et de l'instant précédent (sauf aux extrémités) :

$$\begin{aligned}
\mathbf{x}_{n,0} &= \mathbf{x}_n \\
\mathbf{x}_{n,1} &= \mathbf{x}_{n,0} + \eta \mathbf{f}(\mathbf{x}_0, t) \qquad\qquad \eta \stackrel{\text{def}}{=} \frac{h}{m} \\
\mathbf{x}_{n,2} &= \mathbf{x}_{n,0} + 2\eta \mathbf{f}(\mathbf{x}_{n,1}, t+\eta) \\
&\cdots \\
\mathbf{x}_{n,j+1} &= \mathbf{x}_{n,j-1} + 2\eta \mathbf{f}(\mathbf{x}_{n,j}, t+j\eta) \\
&\cdots \\
\mathbf{x}_{n+1} &= \tfrac{1}{2}\left(\mathbf{x}_{n,m} + \mathbf{x}_{n,m-1} + \eta \mathbf{f}(\mathbf{x}_{n,m}, t+h)\right)
\end{aligned} \tag{2.25}$$

Notons que nous avons introduit une sous-numérotation des intervalles, $\mathbf{x}_{n,j}$ désignant la j^{me} valeur dans l'intervalle no n. Cette méthode requiert de conserver en mémoire non seulement la valeur courante de $\mathbf{x}_{n,j}$, mais la valeur précédente $\mathbf{x}_{n,j-1}$, ce qui n'est pas un problème. Par contre, les permières et dernières valeurs ($j = 0$ et $j = m$) requièrent un traitement spécial.

Suite à ce calcul pour différentes valeurs de m, on pourrait procéder à une extrapolation de Richardson pour obtenir une estimation optimale de $\mathbf{x}_n$, et ensuite passer à l'intervalle h suivant. On obtient ainsi la méthode de Bulirsch-Stoer. Voir *Numerical Recipes* Pour plus de détails.

2.5 Simulation de particules : méthode de Verlet

L'une des applications les plus répandues du calcul scientifique est la simulation du mouvement d'un grand nombre de particules sous l'influence de forces mutuelles et externes. L'objectif de ces simulations est typiquement de comprendre le comportement statistique de la

matière, d'où le nom *dynamique moléculaire* donné à ce champ d'applications. Même s'il est en principe préférable d'utiliser la mécanique quantique pour décrire le mouvement des atomes et des molécules, il est acceptable d'utiliser la mécanique classique pour ce faire si les longueurs d'ondes impliquées sont suffisamment petites par rapport aux distances inter-moléculaires. D'autres applications de ce genre sont aussi très éloignées du domaine quantique, par exemple dans le domaine astronomique.

La dynamique moléculaire est un domaine vaste ; il est hors de question de lui rendre justice dans cette section. Nous allons uniquement décrire l'*algorithme de Verlet*, utilisé pour résoudre les équations du mouvement des particules impliquées. Au fond, il s'agit ici de résoudre un sytème d'équations différentielles, représentant les équations du mouvement de Newton, mais à un nombre plutôt grand de particules. Pourquoi ne pas simplement utiliser la méthode de Runge-Kutta avec pas contrôlé ? C'est une possibilité, mais l'algorithme que nous présenterons plus bas est plus simple, du deuxième ordre, et se compare à la méthode prédicteur-correcteur. La solution ne sera peut-être pas aussi précise que celle obtenue par Runge-Kutta à pas adapté, mais le calcul sera par contre plus rapide. Par contre, nous allons quand même utiliser un pas temporel variable : c'est une caractéristique essentielle à toute méthode alliant précision et rapidité. Notons que l'intérêt de la simulation n'est pas ici de suivre à la trace chaque particule, mais de dégager le comportement de l'ensemble.

Soit donc un ensemble de N particules, de positions $\mathbf{r}_i$ et de vitesses $\mathbf{v}_i$. Chaque particule ressent une force $\mathbf{F}_i$ qui dépend en principe le la position de toutes les particules, ainsi que de la vitesse $\mathbf{v}_i$, si on veut tenir compte de processus d'amortissement (par exemple le rayonnement ou la diffusion de chaleur). Nous avons donc à résoudre le système d'équations différentielles suivant pour les $2N$ vecteurs $\mathbf{r}_i$ et $\mathbf{v}_i$:

$$\frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i \qquad\qquad \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \frac{1}{m}\mathbf{F}_i(\mathbf{r}_j, \mathbf{v}_i, t) \tag{2.26}$$

La méthode de Verlet est basée sur une formule du deuxième ordre pour l'évaluation des dérivées :

$$\begin{aligned} \mathbf{r}_i(t+h) &= \mathbf{r}_i(t) + h\mathbf{v}_i(t) + \frac{h^2}{2m_i}\mathbf{F}_i(t) + \mathcal{O}(h^4) \\ \mathbf{v}_i(t+h) &= \mathbf{v}_i(t) + \frac{h}{2m_i}\Big[\mathbf{F}_i(t+h) + \mathbf{F}_i(t)\Big] + \mathcal{O}(h^2) \end{aligned} \tag{2.27}$$

L'important ici est que toutes les positions doivent être mises à jour avant que les vitesses le soient, car la force $\mathbf{F}_i(t+h)$ doit être calculée entretemps. Ceci suppose que la force ne dépend que des positions. Noter que nous avons supposé que la masse m_i peut être différente d'une particule à l'autre.

Si on ajoute à la force une composante qui dépend de la vitesse, dans le but de modéliser un amortissement par exemple, alors il est nécessaire de traiter cette partie au premier ordre seulement, comme dans la méthode d'Euler ; mais comme l'amortissement est généralement un effet petit, cela ne change pas de manière appréciable la précision de la méthode. On écrirait

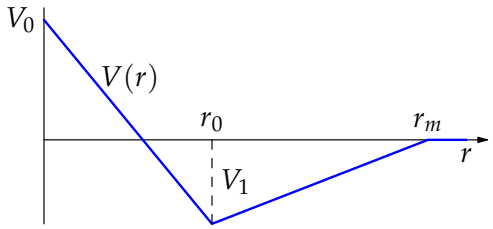


FIGURE 2.2

dans ce cas, par exemple,

$$\mathbf{v}_i(t+h) = (1-h\gamma)\mathbf{v}_i(t) + \frac{h}{2m_i}\left[\mathbf{F}_i(t+h) + \mathbf{F}_i(t)\right] \tag{2.28}$$

où γ est la force d'amortissement par unité de masse.

2.5.1 Exemple : impact d'un objet sur un solide

Nous allons étudier par la méthode de Verlet un ensemble de particules exerçant les unes sur les autres une force centrale constante en deça d'un certain domaine. Cette force sera dérivée du potentiel illustré à la figure 2.2, et aura la forme suivante :

$$F(r) = \begin{cases} \dfrac{V_0 - V_1}{r_0} & \text{si} \quad r < r_0 \\ \dfrac{V_1}{r_m - r_0} & \text{si} \quad r_0 < r < r_m \\ 0 & \text{si} \quad r > r_m \end{cases} \tag{2.29}$$

Cette loi de force est une caricature simple d'une attraction à distances modérées, et d'une répulsion à courte distance, avec une distance d'équilibre r_0. Un potentiel d'interaction plus réaliste serait le célèbre potentiel de Lennard-Jones :

$$V(r) = 4V_0\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right] \tag{2.30}$$

2.5.2 Code

Le code suivant contient la méthode de Verlet, ainsi que des appels permettant de porter la solution en graphique avec gnuplot, au fur et à mesure qu'elle est calculée.

Code 2.3 : Méthode de Verlet

```
#ifndef MD_H
#define MD_H

#include <cmath>
#include "Vector.h"
#include "Vector2D.h"

#define GNUPLOT

#ifdef GNUPLOT
extern "C"{
#include "gnuplot_i.h"
}
#endif

template <class T>
class MD_solver{
public:
  int N; nombre de particules
  T &force;
  Vector<Vector2D> r; positions des particules
  Vector<Vector2D> v; vitesses des particules
  Vector<double> m; masses des particules

  Vector<Vector2D> F; forces au temps t
  Vector<Vector2D> Fh; forces au temps t + h

  double gamma; amortissement
  double pot; énergie potentielle à un instant donné
  double K; énergie cinétique à un instant donné
  double J; moment cinétique (composante en z)
  Vector2D P; quantité de mouvement totale

  MD_solver(int _N, T &_force, double _gamma) : N(_N), force(_force), gamma(_gamma)
      {
    m.Alloc(N);
    r.Alloc(N);
    v.Alloc(N);
    F.Alloc(N);
    Fh.Alloc(N);
  }

  void step(double h, double &max){
    Vector2D f;

    calcul des positions au temps t+h
    for(int i=0; i<N; i++) r[i] += v[i]*h + F[i]*(0.5*h*h/m[i]);

    calcul des forces au temps t+h
```

```
49      max = 0.0;
50      double norm;
51      Fh.clear();
52      for(int i=0; i<N; i++){
53        for(int j=0; j<i; j++){
54          f = force(r[i],r[j]);
55          norm = f.norm();
56          if(norm > max) max = norm;
57          Fh[i] += f;
58          Fh[j] -= f;
59        }
60      }
61
62      calcul des vitesses au temps t+h
63      for(int i=0; i<N; i++) v[i] += (F[i]+Fh[i])*(0.5*h/m[i]) -v[i]*(h*gamma);
64      F.swap(Fh); remplacement des force au temps t par celles au temps t+h
65    }
66
67    void energie(){
68      K = 0.0;
69      P = Vector2D(0,0);
70      J = 0.0;
71      pot = 0.0;
72      for(int i=0; i<N; i++){
73        K += m[i]*v[i].norm2();
74        P += m[i]*v[i];
75        J += m[i]*vector_product(r[i],v[i]);
76      }
77      K *= 0.5;
78      for(int i=0; i<N; i++){
79        for(int j=0; j<i; j++){
80          pot += force.pot(r[i],r[j]);
81        }
82      }
83    }
84
85    conditions initiales
86    void set_initial(istream &fin){
87      assert(fin);
88      for(int i=0; i<N; i++) fin >> m[i] >> r[i] >> v[i];
89    }
90
91    écrit la configuration sur disque
92    void plot(ostream &fout){
93      assert(fout);
94      int N1 = N-1;
95      for(int i=0; i<N1; i++) fout << m[i] << '\t' << r[i] << '\t' << v[i] << '\n';
96      int i=N1; fout << "\n\n";
97      fout << m[i] << '\t' << r[i] << v[i] << '\n'; projectile
```

```
98    }
99
100   void solve(double tmax, double _h0, double L, double dt_display = 0.0){
101     double h0 = _h0;
102     double max_deltav; force maximale permise avant de diminuer h
103     double min_deltav; force minimale permise avant d'augmenter h
104     double max_deltar; vitesse maximale permise avant de diminuer h
105     double min_deltar; vitesse minimale permise avant d'augmenter h
106
107     max_deltar = 0.005;
108     max_deltav = 0.01;
109     min_deltar = 0.05*max_deltar;
110     min_deltav = 0.05*max_deltav;
111
112     if(dt_display == 0.0) dt_display = h0;
113
114     initialisation de la sortie des données
115 #ifdef GNUPLOT
116     ofstream gout;
117     gnuplot_ctrl * GP;
118     GP = gnuplot_init() ;
119     gnuplot_cmd(GP,(char *)"set term x11");
120     gnuplot_cmd(GP,(char *)"set xr [-%g:%g]",L,L);
121     gnuplot_cmd(GP,(char *)"set yr [-%g:%g]",L,L);
122     gnuplot_cmd(GP,(char *)"set size ratio 1");
123     gnuplot_cmd(GP,(char *)"set nokey");
124 #endif
125
126     int output_step=0; compteur des sorties sur fichier
127
128     double h = h0;
129     double max;
130     for(double t=0.0; t<tmax; ){
131
132       Vector<Vector2D> r_back(r);
133       Vector<Vector2D> v_back(v);
134
135       step(h,max);
136
137       double h1 = max_deltav/max;
138       double h2 = sqrt(2*max_deltar/max);
139       h1 = (h1<h2)? h1:h2;
140       if(h > h1){
141         r = r_back;
142         v = v_back;
143         h = h1;
144         continue;
145       }
146       t += h;
```

```
147
148         if(h>20*h0) h = 20*h0;
149         if(h<0.01*h0) h = 0.01*h0;
150
151         if(floor(t/dt_display) > output_step){
152           output_step++;
153           energie();
154 #ifdef GNUPLOT
155           gout.open("tmp.dat");
156           plot(gout);
157           gout.close();
158           gnuplot_cmd(GP,(char *)"set title 't = %2.3f, K=%2.2f, V=%2.2f, E=%2.2f'",t
                ,K/N,pot/N,(K+pot)/N);
159           gnuplot_cmd(GP,(char *)"plot 'tmp.dat' i 0 u 2:3 pt 6, '' i 1 u 2:3 pt 7 lc
                 3 ps 2");
160           usleep(2000);
161           cout << t << '\t' << K << '\t' << pot << '\t' << K+pot << endl;
162 #endif
163         }
164       }
165
166 #ifdef GNUPLOT
167       sleep(5);
168       gnuplot_close(GP) ;
169 #endif
170     }
171
172 };
173
174 #endif
```

Quelques explications sur ce code :

1. La classe `Vector2D` est requise. Il s'agit d'une version bi-dimensionnelle de la classe `Vector3D` déjà étudiée. En particulier, la routine `vector_product()` calcule la composante z (donc hors du plan) du produit vectoriel de deux vecteurs.
2. Ce code utilise une interface avec gnuplot pour afficher les résultats au fur et à mesure qu'ils sont calculés. En fait cette interface n'est utilisée que si le mot-clé `GNUPLOT` est défini. Le code doit être lié à la librairie `gnuplot_i` et le fichier d'entête `gnuplot_i.h` doit être inclus.
3. La classe `MD_solver` est un modèle de classe qui prend comme argument le type de foncteur utilisé pour calculer les forces.
4. La classe garde en mémoire 5 tableaux de taille N (N étant le nombre de particules) : les positions, vitesses et masses des particules, ainsi que les forces calculées au temps t et au temps $t+h$. Certaines quantités dynamiques sont aussi calculées au besoin : l'énergie cinétique, l'énergie potentielle, le moment cinétique et la quantité de mouvement.

5. La méthode `step` effectue un pas temporel h, c'est-à-dire calcule les nouvelles positions $\mathbf{r}_i(t+h)$, les forces au temps $t+h$ (stockées dans `Fh`) et ensuite les nouvelles vitesses $\mathbf{v}_i(t+h)$. La force maximale parmi les $N(N-1)/2$ forces mutuelles est aussi calculée, afin d'être retournée par référence. Si cette force est trop grande, le pas ne sera pas utilisé, mais repris avec une valeur plus petite de h. Par contre, si cette force est suffisamment petite, la valeur de h sera augmentée pour le prochain pas.
6. La méthode `energie()` calcule les quantités dynamiques, pour fins d'affichage ou de diagnostic. Ce calcul n'est pas requis pour la simulation elle-même.
7. La méthode `plot()` écrit l'ensemble des masses, positions et vitesses dans un flux de sortie, afin de les porter en graphique.
8. La solution du système d'équations est réalisée par la méthode `solve`, qui prend comme arguments le temps de simulation, le pas temporel initial, la taille du domaine spatial portée en graphique et l'intervalle de temps entre les affichages.

Ce code est ensuite appelée par le programme principal suivant, qui contient la définition du foncteur de force.

Code 2.4 : Programme principal associé

```
#include <fstream>
#include <cstdlib>
#include "MD_solver.h"
#include "read_parameter.h"

using namespace std;

génère un nombre aléatoire uniformément distribué dans l'intervalle [-1,1]
double random_double(){
  return 2.0*rand()/RAND_MAX - 1.0;
}

Foncteur définissant la force inter-particules
struct force{
  double r0; position du minimum
  double rm; portée
  double V0; potentiel à l'origine
  double V1; potentiel à r0
  double a0;
  double a1;
  double b0;
  double b1;
  force(double _r0, double _rm, double _V0, double _V1): r0(_r0), rm(_rm), V0(_V0),
      V1(_V1) {
    a0 = (V1-V0)/r0;
    b0 = V0;
    a1 = V1/(r0-rm);
```

```
27      b1 = -a1*rm;
28    }
29
30    double pot(Vector2D &r1, Vector2D &r2){
31      double d = (r1-r2).norm();
32      if(d < r0) return a0*d+b0;
33      else if(d < rm) return a1*d+b1;
34      else return 0.0;
35    }
36
37    Vector2D operator() (Vector2D &r1, Vector2D &r2){
38      Vector2D R = r1-r2;
39      double d = (r1-r2).norm();
40      if(d < 0.001*r0) Vector2D(0.0,0.0);
41      else if(d < r0) return R*(-a0/d);
42      else if(d < rm) return R*(-a1/d);
43      else return Vector2D(0.0,0.0);
44    }
45  };
46
47  int main(){
48
49    int n;
50    double V0, V1, r0, rm, v0, dt, dt_display, M, a, gamma, L;
51
52    ifstream para("para.dat");
53    para >> "n" >> n;
54    para >> "a" >> a;
55    para >> "gamma" >> gamma;
56    para >> "V0" >> V0;
57    para >> "V1" >> V1;
58    para >> "r0" >> r0;
59    para >> "rm" >> rm;
60    para >> "vitesse" >> v0;
61    para >> "pas_temporel" >> dt;
62    para >> "L" >> L;
63    para >> "intervalle_affichage" >> dt_display;
64    para >> "masse_projectile" >> M;
65    para.close();
66
67    ofstream fout("in.dat");
68    Vector2D e1(a,0);
69    Vector2D e2(a*cos(M_PI/3),a*sin(M_PI/3));
70    int N=0;
71    int nn = 2*n;
72    for(int i=-nn; i<nn; i++){
73      for(int j=-nn; j<nn; j++){
74        Vector2D r = e1*(i+0.01*random_double())+e2*(j+0.01*random_double());
75        if(r.norm() > n*a) continue;
```

```
76        fout << 1.0 << '\t' << r.x << '\t' << r.y << '\t' << 0 << '\t' << 0 << '\n';
77        N++;
78      }
79    }
80    fout << M << '\t' << -2*n*a << '\t' << 0.2*a << '\t' << v0 << '\t' << 0 << '\n';
81    N++;
82    fout.close();
83
84    force f(r0,rm,V0,V1);
85    MD_solver<force> MD(N,f,gamma);
86
87    ifstream fin("in.dat");
88    MD.set_initial(fin);
89    fin.close();
90
91    MD.solve(100, dt, L, dt_display);
92
93    cout << "exécution terminée normalement" << endl;
94  }
```

Quelques explications sur le programme principal :

1. Le constructueur du foncteur `force` prend comme arguments les paramètres illustrés sur la figure (2.2).
2. L'opérateur `()` est surchargé pour retourner la force entre deux particules situées aux positions r_1 et r_2. Cette structure suppose que la force ne dépend que de la position des particules. Dans le cas présent, nous avons ajouté la possibilité d'une particule de masse plus grande (la dernière du tableau) qui effectue une collision avec le reste des particules. En dépit de cette différence de masse, cette particule spéciale exerce la même force que les autres.
3. Le programme principal commence par lire les paramètres de la simulation dans le fichier d'entrée `para.dat`. Ensuite les positions initiales sont calculées et mises dans le fichier `in.dat`, qui est ensuite lu par la méthode `set_initial()` de la classe `MD_solver`.

2.5.3 Complexité algorithmique des simulations de particules

La méthode de Verlet que nous avons présentée a, dans sa version simple, un grave défaut : le temps de calcul des forces est proportionnel au nombre de paires de particules, soit $\frac{1}{2}N(N-1)$, qui se comporte comme $\mathcal{O}(N^2)$ quand N est grand. Si on désire simuler un très grand nombre de particules, l'évaluation des forces va simplement devenir trop onéreuse et la simulation impossible.

La solution, bien-sûr, est que les forces décroissent rapidement avec la distance. Donc il n'est pas nécessaire de calculer tous les détails des forces pour des particules qui sont éloignées l'une de l'autre. Deux possibilités générales se présentent :

1. Si la force est à courte portée, en particulier si elle décroît plus rapidement que $1/r^2$ en dimension trois, alors on peut négliger les particules qui sont trop éloignées. Supposons pour les fins de l'argument qu'on peut négliger toutes les forces mutuelles au-delà d'une certaine distance R. C'est effectivement ce qui est fait dans le code présenté ci-haut. Cependant, ce code doit tout-de-même effectuer une boucle sur toutes les paires de particules pour vérifier si les distances sont inférieures à R, ce qui ne règle pas le problème de la complexité algorithmique d'ordre N^2. Pour s'en sortir, il faut une représentation des données différentes, dans laquelle on peut avoir accès directement aux particules qui sont dans une région donnée, sans avoir à chercher le tableau contenant les positions des particules. Par exemple, on pourrait diviser l'espace en un réseau de cellules identiques de taille R, et construire une liste dynamique des particules qui résident à un instant donné dans chaque cellule. Dans le calcul des forces, nous n'aurions alors qu'à considérer les particules qui résident dans une cellule donnée et les cellules immédiatement voisines. La complexité algorithmique serait réduite à $\mathcal{O}(Mn^2) = \mathcal{O}(Nn)$, $M = N/n$ étant le nombre de cellules et n le nombre de particule par cellule. Un tel schéma demande bien sûr à garder trace des particules qui passent d'une cellule à l'autre lors d'un pas temporel. En pratique, on définirait des cellules qui se chevauchent sur une distance égale à la portée de la force, comportant chacune une région intérieure et une région périphérique : il ne serait alors pas requis de sortir de la cellule pour calculer les forces et seules les particules de la région intérieure seraient propagées : celles de la région périphérique étant en même temps dans la région intérieure d'une cellule voisine, elles ne servent dans la cellule courante qu'au calcul des forces.
2. Si les forces sont à longue portée, comme la force gravitationnelle, alors on ne peut pas simplement ignorer les particules éloignées. La raison étant que même si la force décroît comme $1/r^2$, le nombre de particules situées à une distance d'ordre r d'un point donné croît comme r^2 quand la densité est du même ordre partout ; donc on commetterait une erreur grave en ignorant les particules éloignées. Cependant, on peut appliquer dans ce cas le développement multipolaire et considérer l'influence de groupes de particules et non de particules individuelles. Une façon de procéder est de construire un réseau de cellules comme au cas précédent, mais en plus de les organiser en une hiérarchie de super-cellules à plusieurs niveaux : les cellules de taille R sont contenues dans des super-cellules de taille $3R$, elles-mêmes contenues dans des super-cellules de taille 3^2R, et ainsi de suite. Dans le calcul des forces, on calcule quelques multipôles produits par chaque cellule d'un niveau donné, qui ensuite servent à calculer les multipôles du niveau supérieur de cellules, etc. De cette manière, la complexité algorithmique du problème passe de $\mathcal{O}(N^2)$ à $\mathcal{O}(N \log N)$ (le nombre de niveaux de cellules étant d'ordre $\log(N)$. Ce qui présente est une caricature de ce qui est en fait accompli dans ces algorithmes dits de *multipôles rapides*, mais l'idée de base est la même.

2.5.4 Aspects quantiques et statistiques

Dans le domaine microscopique, on peut se demander si la mécanique classique est appropriée pour décrire le mouvement des molécules. En réalité, seul le mouvement des ions est déterminé

classiquement ; les forces inter-moléculaires dépendent cependant des configurations électroniques qui, elles, ne peuvent être déterminées que par la mécanique quantique. Les véritables simulations de dynamique moléculaire tiennent donc compte de la mécanique quantique, à un certain degré d'approximation, dans le calcul des forces. Une avancée considérable a été accomplie dans ce domaine par Car et Parinello en 1985 avec l'introduction d'une méthode rapide permettant d'avancer dans les temps les configurations électroniques en même temps que les positions des ions. Lorsqu'un calcul quantique – même approximatif – des forces interioniques est effectué, on parle de *dynamique moléculaire ab initio*. Sinon, il s'agit de *dynamique moléculaire classique*.

Les systèmes simples qui sont simulés par le code décrit plus haut sont décrits par l'ensemble micro-canonique : le nombre de particules et l'énergie du système sont constants (sauf si on inclut un amortissement). Il peut être cependant plus réaliste, en simulant un système complexe, de supposer que ce système est lui-même un sous-ensemble d'un système encore plus grand, avec lequel il échange de l'énergie et, à la rigueur, des particules. Autrement dit, on peut simuler le système en le mettant en contact avec un environnement caractérisé par une température T (ensemble canonique). La simulation doit alors comporter un procédure pour communiquer aux particules qui parviennent à la frontière du domaine étudié une signature de leur interaction avec l'environnement, par exemple une variation de l'énergie, caractéristique de l'ensemble canonique. On peut aussi ajouter une force aléatoire agissant sur chaque particule qui, combinée à un terme d'amortissement γ, permet de reproduire une distribution canonique (dynamique de Langevin).

Chapitre 3

Représentations des fonctions

L'utilisation en physique et en génie de fonctions de l'espace, ou *champs*, est omniprésente. Dans la plupart des cas, l'utilisation d'un continuum est elle-même l'approximation d'une réalité sous-jacente discontinue. Par exemple, la théorie de l'élasticité ou de la propagation des ondes dans les milieux repose sur l'approximation du continuum, alors qu'un traitement plus fondamental devrait tenir compte de la structure des milieux en question à l'échelle moléculaire. Cette approximation du continuum est cependant très utile car elle offre une description de ces systèmes indépendante des détails microscopiques et reposant sur certaines équations différentielles, telles l'équation d'onde, l'équation de Navier-Stokes en mécanique des fluides, etc.

Ces équations différentielles aux dérivées partielles, quand elles sont à leur tour traitées numériquement, doivent faire l'objet d'une *rediscrétisation*, car les calculateurs ne peuvent traiter que des problèmes finis. Il ne s'agit pas bien sûr de retourner, dans ces descriptions disrètes des sytèmes étudiés, à une description microscopique exacte du problème. On cherche simplement une formulation approchée, un problème différent, qui redonne cependant la même théorie continue dans une certaine limite.

Bref, un problème central du calcul scientifique est la représentation approximative de fonctions continues et des opérations qu'on effectue sur ces fonctions, comme la différentiation, l'intégration, la transformation de Fourier, etc. C'est l'objet de ce chapitre.

3.1 Différences finies

L'approche la plus simple, et malheureusement la plus limitée, à la représentation des fonctions est de considérer une grille uniforme de points et de s'en tenir là ; c'est-à-dire de ne rien supposer sur l'existence d'une fonction en dehors de ces points. Considérons, pour simplifier les choses, le cas d'une seule dimension, dans un intervalle fini $[0, \ell]$ de la coordonnée x. On suppose que la grille est constituée de N points x_i définis comme suit :

$$x_i = ia \qquad i = 0, 1, \ldots, N-1 \qquad a = \frac{\ell}{N-1} \tag{3.1}$$

Si l'intervalle est périodique, c'est-à-dire si les points $x = 0$ et $x = \ell$ sont les mêmes, alors il faut imposer cette condition périodique et retrancher l'une des extrémités. Une fonction $\psi(x)$ devient alors un tableau de N valeurs ψ_i.

Si on veut cependant définir des opérations comme la différentiation ou l'intégration de fonctions, il est nécessaire de supposer que la grille $\{x_i\}$ n'est qu'un échantillonage d'un intervalle continu et que le tableau ψ_i l'échantillonage correspondant d'une fonction $\psi(x)$ qui possède certaines propriétés de continuité de différentiabilité, etc.

L'opération de différentiation $D_x\psi$ pourrait être numériquement définie de diverses façon, mais celle qui converge le plus rapidement quand $a \to 0$ est l'expression du deuxième ordre

$$(D_x\psi)_i = \frac{\psi_{i+1} - \psi_{i-1}}{2a} \qquad i = 1, 2, \ldots, N-2 \tag{3.2}$$

Le cas des extrémités demande réflexion. Si les conditions aux limites sont périodiques, le problème se règle de lui-même de manière évidente. Sinon nous sommes limités à une expression du premier ordre aux extrémités :

$$(D_x\psi)_0 = \frac{\psi_1 - \psi_0}{a} \qquad (D_x\psi)_{N-1} = \frac{\psi_{N-1} - \psi_{N-2}}{a} \tag{3.3}$$

La dérivée seconde, elle, serait estimée de la manière suivante :

$$(D_x^2\psi)_i = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{a^2} \qquad i = 1, 2, \ldots, N-2 \tag{3.4}$$

les extrémités, encore une fois, nécéssitant un traitement spécial.

De même, l'intégration de la fonction ψ de 0 à ℓ pourrait à première vue se faire strictement par la sommation de ses valeurs :

$$\int_0^\ell \mathrm{d}x\, \psi(x) \to a \sum_{i=0}^{N-1} \psi_i \tag{3.5}$$

Cependant, dans le cas de conditions aux limites non périodiques, chaque point d'extrémité ne compte que pour la moitié d'un intervalle, de sorte que la formule d'intégration doit alors être

$$\int_0^\ell \mathrm{d}x\, \psi(x) \to \frac{a}{2}(\psi_0 + \psi_{N-1}) + a \sum_{i=1}^{N-2} \psi_i \tag{3.6}$$

Cette formule est la célèbre *formule des trapèzes* pour l'intégration d'une fonction, illustrée à la figure 3.1.

3.1.1 Interpolation

Sachant que la fonction $\psi(x)$ existe, on peut supposer qu'elle est analytique, c'est-à-dire qu'elle peut être représentée par un développement en série de Taylor

$$\psi(x + \epsilon) = \psi(x) + \epsilon\psi'(x) + \tfrac{1}{2}\epsilon^2\psi''(x) + \cdots \tag{3.7}$$

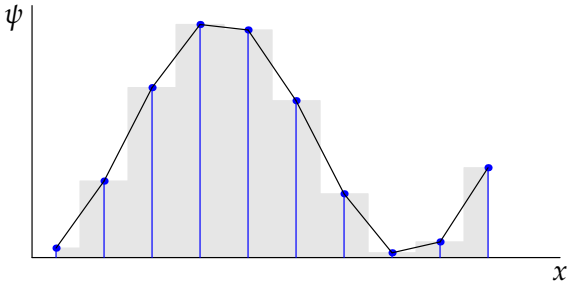


FIGURE 3.1 Illustration de la méthode des trapèzes. Notez que l'aire des rectangles ombragées est la même que l'aire limitée par les segments linéaires qui forment des trapèzes.

autour de chaque point de l'intervalle, sauf de possibles exceptions en nombre fini.

Cela entraîne que la fonction, dans un domaine limité autour de chaque point, peut être adéquatement représentée par un polynôme en tronquant cette série. Nous pourrions donc évaluer approximativement la fonction $\psi(x)$ en tout point, même si elle n'est connue que sur une grille discrète, en procédant à une interpolation polynomiale. La précision de cette approximation sera en proportion du degré du polynôme utilisé.

Les polynômes en question se trouvent par la *formule de Lagrange*. Si on connait la fonction $\psi_i = \psi(x_i)$ à M points notés $x_0, \ldots, x_{M-1}$ (attention : M n'est pas nécessairement égal à N, mais peut ne représenter qu'un petit sous-ensemble de points contigus) alors le polynôme de degré $M-1$ unique qui passe par tous ces points est

$$\begin{aligned} P(x) &= \frac{(x-x_1)(x-x_2)\cdots(x-x_{M-1})}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_{M-1})}\psi_0 + \frac{(x-x_0)(x-x_2)\cdots(x-x_{M-1})}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_{M-1})}\psi_1 + \cdots \\ &\qquad + \frac{(x-x_0)(x-x_1)\cdots(x-x_{M-2})}{(x_{M-1}-x_0)(x_{M-1}-x_1)\cdots(x_{M-1}-x_{M-2})}\psi_{M-1} \\ &= \sum_{j=0}^{M-1}\left[\prod_{\substack{i\\(i\neq j)}}^{M-1}\frac{x-x_i}{x_j-x_i}\psi_j\right] \end{aligned} \tag{3.8}$$

Il est manifeste que ce polynôme est de degré $M-1$, car chaque terme est le produit de $M-1$ facteurs impliquant x. D'autre part, le coefficient du terme de degré $M-1$ est

$$\sum_{j=0}^{M-1}\left[\prod_{i\ (i\neq j)}^{M-1}\frac{1}{x_j-x_i}\psi_j\right] \tag{3.9}$$

et ce terme est généralement non nul (il peut l'être accidentellement, bien sûr). Ensuite, ce polynôme passe manifestement par tous les points (x_i, ψ_i), car si $x = x_k$, seul le terme $j = k$ de la somme est non nul et la fraction correspondante est alors égale à l'unité, ne laissant que $P(x_k) = \psi_k$. La contrainte que le polynôme passe par $M-1$ points donnés fixe complètement les $M-1$ paramètres nécessaires pour spécifier uniquement un polynôme de degré $M-1$. La solution de Lagrange est donc unique.

Par exemple, la formule d'interpolation linéaire entre deux points x_0 et x_1 est

$$P(x) = \frac{x - x_1}{x_0 - x_1}\psi_0 + \frac{x - x_0}{x_1 - x_0}\psi_1 \tag{3.10}$$

et la formule d'interpolation quadratique entre trois points $x_{0,1,2}$ est

$$P(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}\psi_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}\psi_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}\psi_2 \tag{3.11}$$

Il est possible de coder directement la formule de Lagrange (3.8), mais un algorithme plus efficace existe pour cela, et donne accès à l'erreur commise entre les interpolations de degré $M-2$ et $M-1$.[1]

3.1.2 Cubiques raccordées

L'interpolation linéaire d'une fonction, d'un point à l'autre de la grille, produit une fonction approchée dont la dérivée seconde est nulle partout, sauf aux points de grille où elle est infinie. Pour pallier ce problème, on a fréquemment recours aux cubiques raccordées (angl. *cubic splines*) : il s'agit d'une approximation cubique qui a l'avantage de produire une fonction approchée dont les dérivées premières et secondes sont continues aux points de grilles.

Étant donnés deux points x_i et x_{i+1} de la grille, le polynôme cubique $P_i(x)$ à 4 paramètres doit alors obéir à 4 conditions :

$$P_i(x_i) = \psi(x_i) \qquad P_i(x_{i+1}) = \psi(x_{i+1}) \qquad P_i'(x_i) = P_{i-1}'(x_i) \qquad P_i''(x_i) = P_{i-1}''(x_i) \tag{3.12}$$

ce qui le détermine uniquement. Notez que les contraintes d'égalité des dérivées première et seconde au point x_{i+1} s'appliquent au polynôme $P_{i+1}(x)$ dans cette convention. Reste l'éternel problème des extrémités. Plusieurs possibilités existent ;

1. On peut imposer une valeur nulle des dérivées secondes aux deux extrémités, et obtenir ce qu'on appelles des cubiques raccordées *naturelles*.
2. On peut obéir à des conditions aux limites externes qui fixent les valeurs de la dérivée première aux deux extrémités, ce qui permet de déterminer les dérivées secondes.

Démontrons maintenant explicitement comment produire le polynôme interpolant. Rappelons que l'interpolation linéaire a la forme

$$P(x) = A\psi_i + B\psi_{i+1} \quad \text{où} \quad A = \frac{x - x_{i+1}}{x_i - x_{i+1}} \quad \text{et} \quad B = 1 - A = \frac{x - x_i}{x_{i+1} - x_i} \tag{3.13}$$

Une interpolation cubique peut être mise sous la forme suivante :

$$P_i(x) = a(x - x_i) + c(x - x_i)^3 + b(x - x_{i+1}) + d(x - x_{i+1})^3 \tag{3.14}$$

Cette forme contient 4 paramètres à déterminer, donc suffisamment générale pour un polynôme cubique. Pour déterminer ces 4 paramtères, nous allons imposer les valeurs de la fonction

1. Voir *Numerical Recipes*, section 3.2.

aux deux extrémités de l'intervalle : $P(x_i) = \psi_i$ et $P(x_{i+1}) = \psi_{i+1}$. Nous allons également supposer connues les deuxièmes dérivées au mêmes points : $P''(x_i) = \psi''_i$ et $P''(x_{i+1}) = \psi''_{i+1}$. Ceci mène aux équations suivantes :

$$\begin{aligned} a &= \frac{\psi_{i+1}}{x_{i+1} - x_i} - \frac{1}{6}\psi''_{i+1}(x_{i+1} - x_i) & c &= \frac{1}{6}\frac{\psi''_{i+1}}{x_{i+1} - x_i} \\ b &= \frac{\psi_i}{x_i - x_{i+1}} - \frac{1}{6}\psi''_i(x_i - x_{i+1}) & d &= \frac{1}{6}\frac{\psi''_i}{x_i - x_{i+1}} \end{aligned} \tag{3.15}$$

ce qui équivaut à la forme suivante pour le polynôme interpolant :

$$P_i(x) = A\psi_i + \frac{1}{6}(A^3 - A)\psi''_i(x_i - x_{i+1})^2 + B\psi_{i+1} + \frac{1}{6}(B^3 - B)\psi''_{i+1}(x_i - x_{i+1})^2 \tag{3.16}$$

A et B étant définis en (3.13). Attention à la notation cependant : les valeurs ψ''_i et ψ''_{i+1} ne sont pas nécessairement les vraies dérivées secondes de la fonction $\psi(x)$ sous-jacente, alors qu'on suppose que les valeurs ψ_i sont véritablement $\psi(x_i)$. Ce sont plutôt les dérivées secondes des polynômes interpolants, que nous allons raccorder à chaque point.

La condition qui nous manque pour déterminer ψ''_i est la continuité de la dérivée à chaque point. On calcule à cette fin que

$$P'_i(x) = \frac{\psi_{i+1} - \psi_i}{x_{i+1} - x_i} - \frac{1}{6}\left[(3A^2 - 1)\psi''_i - (3B^2 - 1)\psi''_{i+1}\right](x_{i+1} - x_i) \tag{3.17}$$

et donc que

$$\begin{aligned} P'_i(x_i) &= \frac{\psi_{i+1} - \psi_i}{x_{i+1} - x_i} - \frac{1}{6}(2\psi''_i + \psi''_{i+1})(x_{i+1} - x_i) \\ P'_i(x_{i+1}) &= \frac{\psi_{i+1} - \psi_i}{x_{i+1} - x_i} + \frac{1}{6}(\psi''_i + 2\psi''_{i+1})(x_{i+1} - x_i) \end{aligned} \tag{3.18}$$

La condition manquante est alors $P'_i(x_i) = P'_{i-1}(x_i)$, ou encore

$$\frac{\psi_i - \psi_{i+1}}{x_{i+1} - x_i} - \frac{1}{6}(2\psi''_i + \psi''_{i+1})(x_{i+1} - x_i) = \frac{\psi_{i-1} - \psi_i}{x_i - x_{i-1}} + \frac{1}{6}(\psi''_{i-1} + 2\psi''_i)(x_i - x_{i-1}) \tag{3.19}$$

ce qui s'exprime également comme

$$\frac{x_i - x_{i-1}}{6}\psi''_{i-1} + \frac{x_{i+1} - x_{i-1}}{3}\psi''_i + \frac{x_{i+1} - x_i}{6}\psi''_{i+1} = \frac{\psi_{i+1} - \psi_i}{x_{i+1} - x_i} - \frac{\psi_i - \psi_{i-1}}{x_i - x_{i-1}} \tag{3.20}$$

Il s'agit d'un système d'équations linéaires qui nous permet de déterminer les dérivées secondes ψ''_i, connaissant les points x_i et les valeurs ψ_i de la fonction. Ce système est tridiagonal, ce qui est un avantage calculatoire : les systèmes tridiagonaux à N variables étant résolus par un algorithme de complexité $\mathcal{O}(N)$.

3.1.3 Intégration suite à une interpolation

L'interpolation d'une fonction permet d'en définir les intégrales et les dérivées, en fonction de l'intégrale et des dérivées de l'interpolant. Par exemple, l'interpolation linéaire d'une fonction $\psi(x)$ nous mène à l'expression suivante pour son intégrale sur le domaine considéré :

$$\int_0^\ell \mathrm{d}x\,\psi(x) \to \sum_{i=0}^{N-2} \int_{x_i}^{x_{i+1}} \left\{ \frac{x-x_{i+1}}{x_i-x_{i+1}}\psi_i + \frac{x-x_i}{x_{i+1}-x_i}\psi_{i+1} \right\} = \sum_{i=0}^{N-2} \frac{1}{2}(\psi_i+\psi_{i+1})(x_{i+1}-x_i) \tag{3.21}$$

En considérant comment chaque terme de cette somme se combine avec les termes voisins, on trouve

$$\int_0^\ell \mathrm{d}x\,\psi(x) \to \tfrac{1}{2}\left[\psi_0(x_1-x_0)+\psi_1(x_2-x_0)+\psi_2(x_3-x_1)+\cdots+\psi_{N-1}(x_{N-1}-x_{N-2})\right] \tag{3.22}$$

Notez encore une fois que les extrémités sont particulières. Il s'agit manifestement de la méthode des trapèzes mentionnée plus haut. Pour une grille uniforme où $x_{i+1}-x_i = a$, ceci devient simplement

$$\int_0^\ell \mathrm{d}x\,\psi(x) \to \tfrac{1}{2}a(\psi_0+\psi_{N-1}) + a\sum_{i=1}^{N-2}\psi_i \tag{3.23}$$

Erreur de troncature dans la formule des trapèzes

La formule des trapèzes se base sur l'intégrale suivante dans un intervalle élémentaire de largeur a :

$$\int_0^a \mathrm{d}x\,\psi(x) \approx \frac{1}{2}a\big[\psi(0)+\psi(a)\big] \tag{3.24}$$

Nous allons montrer que cette expression s'accompagne d'une erreur de troncature d'ordre $\mathcal{O}(a^3)$. Pour ce faire, considérons les deux developpements limités suivants, l'un autour de $x=0$, l'autre autour de $x=a$:

$$\begin{aligned} \psi(x) &= \psi(0)+\psi'(0)x+\tfrac{1}{2}\psi''(0)x^2+\mathcal{O}(a^3) \\ \psi(x) &= \psi(a)+\psi'(a)(x-a)+\tfrac{1}{2}\psi''(a)(x-a)^2+\mathcal{O}(a^3) \end{aligned} \tag{3.25}$$

Les intégrales des ces deux développements sur $[0,a]$ donnent respectivement

$$\begin{aligned} \int_0^a \mathrm{d}x\,\psi(x) &= a\psi(0)+\tfrac{1}{2}a^2\psi'(0)+\tfrac{1}{6}a^3\psi''(0)+\mathcal{O}(a^4) \\ \int_0^a \mathrm{d}x\,\psi(x) &= a\psi(a)-\tfrac{1}{2}a^2\psi'(a)+\tfrac{1}{6}a^3\psi''(a)+\mathcal{O}(a^4) \end{aligned} \tag{3.26}$$

La moyenne de ces deux approximations est

$$\int_0^a \mathrm{d}x\,\psi(x) = \tfrac{1}{2}a\big[\psi(0)+\psi(a)\big] + \tfrac{1}{4}a^2\big[\psi'(0)-\psi'(a)\big] + \tfrac{1}{12}a^3\big[\psi''(0)+\psi''(a)\big] + \mathcal{O}(a^4) \tag{3.27}$$

Mais, d'après le théorème de la moyenne,

$$\big[\psi'(0)-\psi'(a)\big] = \psi''(\bar{x})a \tag{3.28}$$

où $\bar{x}$ est une valeur quelque part entre 0 et a. Donc finalement

$$\int_0^a \mathrm{d}x\, \psi(x) = \frac{1}{2}a\big[\psi(0) + \psi(a)\big] + \mathcal{O}(a^3) \tag{3.29}$$

En sommant les contributions des différents intervalles, on accumule les erreurs de troncature, commettant une erreur d'ordre $\mathcal{O}(Na^3) = \mathcal{O}(a^2) = \mathcal{O}(1/N^2)$ car $a = \ell/(N-1)$.

Formule de Simpson

Procédons maintenant à un ordre supérieur d'approximation et faisons passer une parabole entre trois points de la grille (disons x_1, x_2 et x_3 pour simplifier). Le polynôme unique passant pas ces points est donné à l'équation (3.11). Supposons pour simplifier que la grille soit uniforme, de sorte que $x_3 - x_2 = x_2 - x_1 = a$. L'expression (3.11) devient

$$P(x) = \psi_1 + \frac{x}{2a}(3\psi_1 - 4\psi_2 + \psi_3) + \frac{x^2}{2a^2}(\psi_1 - 2\psi_2 + \psi_3) \tag{3.30}$$

L'intégrale de cette expression entre x_1 et x_3 est

$$\int_{x_1}^{x_3} \mathrm{d}x\, P(x) = a\left(\frac{1}{3}\psi_1 + \frac{4}{3}\psi_2 + \frac{1}{3}\psi_3\right) \tag{3.31}$$

Il s'agit de la *règle de Simpson* pour un intervalle simple. Appliquons maintenant cette formule à une grille, en considérant les intervalles successifs $[x_0, x_2]$, $[x_2, x_4]$, etc. On trouve alors la *règle de Simpson étendue* :

$$\int_0^\ell \mathrm{d}x\, \psi(x) \approx a\left(\frac{1}{3}\psi_0 + \frac{4}{3}\psi_1 + \frac{2}{3}\psi_2 + \frac{4}{3}\psi_3 + \frac{2}{3}\psi_4 + \cdots + \frac{4}{3}\psi_{N-2} + \frac{1}{3}\psi_{N-1}\right) \tag{3.32}$$

(nous avons supposé que N est pair).

Exercice 3.1

Montrez que l'erreur de troncature de la formule de Simpson étendue (3.32) est $\mathcal{O}(1/N^4)$. Utilisez la même méthode que pour la méthode des trapèzes, en poussant le développement plus à un ordre plus élevé.

3.2 Polynômes orthogonaux

3.2.1 Polynômes orthogonaux

Considérons les fonctions définies en une dimension sur l'intervalle $[a,b]$. Définissons sur cet intervalle un produit scalaire de fonctions par l'expression suivante :

$$\langle \psi | \psi' \rangle = \int_a^b \mathrm{d}x\, w(x)\psi(x)\psi'(x) \tag{3.33}$$

où $w(x)$ est la *fonction poids* associée au produit scalaire. Cette fonction est par hypothèse partout *positive* dans l'intervalle $[a,b]$. Nous pouvons toujours construire une base de polynômes $p_j(x)$ qui sont orthogonaux par rapport à ce produit scalaire. En effet, il suffit de commencer par un polynôme de degré 0, et ensuite d'ajouter à cette base des polynômes de degrés croissants obtenus par la procédure d'orthogonalisation de Gram-Schmidt. Il n'est pas nécessaire de les normaliser : souvent, on préfère travailler avec des polynômes dont le coefficient du plus haut degré est l'unité, c'est-à-dire $p_j(x) = x^j + a_{j,j-1}x^{j-1} + \dots$ (on dit qu'ils sont *unitaires* (angl. *monic*)). Ces polynômes ont donc la propriété que

$$\langle p_i | p_j \rangle = \delta_{ij}\gamma_i \quad \text{ou} \quad \int_a^b w(x)p_i(x)p_j(x) = \delta_{ij}\gamma_i \qquad (i,j = 0,1,2,\dots) \tag{3.34}$$

Les polynômes orthogonaux unitaires s'obtiennent de la relation de récurrence suivante :

$$\begin{aligned} p_{-1}(x) &\stackrel{\text{def}}{=} 0 \\ p_0(x) &= 1 \\ p_{j+1}(x) &= (x - a_j)p_j(x) - b_j p_{j-1}(x) \end{aligned} \tag{3.35}$$

où

$$a_j = \frac{\langle x p_j | p_j \rangle}{\langle p_j | p_j \rangle} \quad (j \geq 0) \qquad\qquad b_j = \frac{\langle p_j | p_j \rangle}{\langle p_{j-1} | p_{j-1} \rangle} \quad (j \geq 1) \tag{3.36}$$

Exercice 3.2

Démontrez les relations (3.36), en supposant la récurrence (3.35) et l'orthogonalité. La solution trouvée démontre alors que la relation de récurrence est correcte, ainsi que l'orthogonalité.

Exemple 3.1: Polynômes de Legendre

Les polynômes de Legendre correspondent au cas $a = -1$, $b = 1$ et $w(x) = 1$. En appliquant la relation de récurrence, on trouve

$$\begin{array}{lll} p_0(x) = 1 & \langle p_0 | p_0 \rangle = 2 & \langle x p_0 | p_0 \rangle = 0 \\ p_1(x) = x & \langle p_1 | p_1 \rangle = \dfrac{2}{3} & \langle x p_1 | p_1 \rangle = 0 \\ p_2(x) = x^2 - \dfrac{1}{3} & \langle p_2 | p_2 \rangle = \dfrac{8}{45} & \langle x p_2 | p_2 \rangle = 0 \\ p_3(x) = x^3 - \dfrac{3}{5}x & \dots & \dots \end{array} \tag{3.37}$$

Les polynômes de Legendre proprement dits (notés $P_n(x)$) ne sont pas unitaires, mais sont normalisés par la condition $P_n(1) = 1$. En multipliant les polynômes unitaires ci-dessus par la constante appropriée, on trouve alors

$$P_0(x) = 1 \quad , \quad P_1(x) = x \quad , \quad P_2(x) = \frac{3}{2}x^2 - \frac{1}{2} \quad , \quad P_3(x) = \frac{5}{2}x^3 - \frac{3}{2} \quad , \quad \cdots \tag{3.38}$$

Théorème 3.1: Entrelacement des racines

Les racines de p_j sont réelles et celles de p_{j-1} s'intercalent entre celles de p_j.

Preuve:

Ce théorème se prouve par récurrence. Il est évident pour $j = 0$ (aucune racine). Supposons qu'il est vrai pour p_j et montrons qu'il doit alors être vrai pour $j+1$. Soit x_i ($i = 1, 2, \ldots, j$) les j racines de p_j dans l'intervalle $[a, b]$. Appliquons la relation de récurrence (3.35) au point x_i :

$$p_{j+1}(x_i) = -b_j p_{j-1}(x_i) \qquad\qquad p_j(x_i) \stackrel{\text{def}}{=} 0 \tag{3.39}$$

Comme les racines de p_{j-1} sont par hypothèse entrelacées avec celles de p_j, le signe de $p_j(x_i)$ doit être l'opposé de celui de $p_j(x_{i+1})$ (voir figure 3.3), car p_{j-1} change de signe exactement une fois entre x_i et $x_i + 1$. Comme b_j est toujours positif, cela entraînes que p_{j+1} aussi change de signe entre ces deux valeurs, et qu'il a au moins une racine entre x_i et $x_i + 1$. Il reste à montrer qu'en plus, p_{j+1} possède une racine dans l'intervalle $[x_j, b]$ et une autre dans $[a, x_1]$, c'est-à-dire à l'extérieur des racines de p_j. Comme p_{j+1} a au plus $j+1$ racine, cela montrerait aussi qu'il y a exactement une racine de p_{j+1} entre x_i et x_{i+1}. Or comme p_{j-1} est unitaire et que sa dernière racine est plus petite que x_j, il doit être positif à x_j. Donc $p_{j+1}(x_j) < 0$, alors que p_{j+1} est lui aussi unitaire ; donc il possède une racine plus élevée que x_j car $p_{j+1}(x) > 0$ pour x suffisamment grand. Le même argument s'applique pour $x < x_1$, car p_{j+1} tend vers $\pm\infty$ quand $x \to -\infty$, selon que j est impair ou pair.

Théorème 3.2: Existence des racines dans l'intervalle d'orthogonalité

Le polynôme $p_j(x)$ a exactement j racines dans l'intervalle $[a, b]$.

Preuve:

Encore une fois, nous procédons par récurrence, comme dans le théorème d'entrelacement. Par rapport à la preuve précédente, il nous reste à prouver que les racines extrêmes de p_j sont comprise dans l'intervalle $[a, b]$, si celles de p_{j-1} le sont. À cette fin, considérons la fonction

$$s(x) = \prod_{x_i \in [a,b]} (x - x_i) \tag{3.40}$$

où le produit est restreint aux racines x_i de p_j qui sont dans l'intervalle $[a, b]$. La fonction $s(x)$ est égale à $p_j(x)$ si le théorème est vrai, car il s'agit alors simplement de la factorisation de p_j en fonction de toutes ses racines. Supposons au contraire que le théorème soit faux et montrons qu'on arrive à une contradiction. La fonction $s(x)$ est alors un polynôme de degré $m < j$. Ce polynôme peut alors s'exprimer comme une combinaison linéaire des $p_k(x)$, ($k = 1, 2, \ldots, j-1$). Il est donc orthogonal à p_j et

$$\int_a^b w(x) s(x) p_j(x) = 0 \tag{3.41}$$

Mais comme $s(x)$ s'annule exactement aux même points que $p_j(x)$ dans l'intervalle, il possède toujours le même signe que $p_j(x)$ et donc l'intégrant est stricement positif dans l'intervalle et l'intégrale ne peut être nulle. Donc $s(x) = p_j(x)$.

3.2.2 Quadratures gaussiennes

Les polynômes orthogonaux sont à la base d'une technique d'intégration très répandue : la quadrature gaussienne. Supposons qu'on veuille intégrer une fonction $y(x) \stackrel{\text{def}}{=} w(x)\tilde{y}(x)$ dans l'intervalle $[a,b]$. En se basant sur le théorème ci-dessous, on voit que l'estimation de l'intégrale

$$\int_a^b \mathrm{d}x\, w(x)\tilde{y}(x) \quad \text{par la somme} \quad \sum_{i=1}^{N} w_i \tilde{y}(x_i) \tag{3.42}$$

est optimale si les n points de grille x_i sont choisis comme étant les racines du polynôme $p_N(x)$ dans $[a,b]$, et les poids w_i en solutionnant le système linéaire (3.43) ci-dessous. Dans ce cas, l'estimation de l'intégrale est exacte si $\tilde{y}$ est un polynôme de degré $2N-1$ ou moins.

Théorème 3.3: quadratures gaussiennes

Soit x_i les N racines du polynôme orthogonal $p_N(x)$, et w_i la solution du système linéaire suivant :

$$\sum_{i=1}^{N} w_i p_k(x_i) = \begin{cases} \langle p_0 | p_0 \rangle & \text{si} \quad k = 0 \\ 0 & \text{si} \quad k = 1, 2, \ldots, N-1 \end{cases} \tag{3.43}$$

Alors $w_i > 0$ et

$$\int_a^b w(x)p(x) = \sum_{i=1}^{N} w_i p(x_i) \tag{3.44}$$

pour tout polynôme $p(x)$ de degré $2N-1$ ou moins.

Preuve:

D'après la référence [SB02]. Considérons la matrice

$$A \stackrel{\text{def}}{=} \begin{pmatrix} p_0(x_1) & \cdots & p_0(x_N) \\ \vdots & \ddots & \vdots \\ p_{N-1}(x_1) & \cdots & p_{N-1}(x_N) \end{pmatrix} \tag{3.45}$$

Cette matrice est non singulière, car les polynômes sont linéairement indépendants. Autrement dit, si elle était singulière, il y aurait un vecteur $(c_0, \ldots, c_{N-1})$ tel que $\tilde{c}A = 0$ et donc le polynôme

$$q(x) \stackrel{\text{def}}{=} \sum_{i=0}^{N-1} c_i p_i(x) \tag{3.46}$$

qui est au plus de degré $N-1$, aurait N racines distinctes $x_1, x_2, \ldots, x_N$, ce qui est impossible, à moins qu'il soit identiquement nul, c'est-à-dire $c = 0$. Donc A est une matrice non singulière. Par conséquent, les coefficients w_i existent et leur valeur est unique.

Considérons ensuite un polynôme $p(x)$ de degré $2N-1$ au plus. Ce polynôme peut donc s'écrire comme $p(x) = p_N(x)q(x) + r(x)$, où $q(x)$ et $r(x)$ sont des polynômes de degré $N-1$

au plus. Ceci est vrai par construction de la division longue des polynômes (comme pour la division des nombres en représentation décimale, qui sont des polynômes en puissances de la base utilisée, à savoir 10). Le quotient $q(x)$ et le reste $r(x)$ peuvent donc être développés sur la base des p_j :

$$q(x) = \sum_{j=0}^{N-1} \alpha_j p_j(x) \qquad\qquad r(x) = \sum_{j=0}^{N-1} \beta_j p_j(x) \tag{3.47}$$

Comme $p_0(x) = 1$, il s'ensuit que

$$\int_a^b w(x)p(x) = \langle p_N|q\rangle + \langle r|p_0\rangle = \beta_0\langle p_0|p_0\rangle \tag{3.48}$$

D'un autre côté,

$$\sum_{i=1}^{N} w_i p(x_i) = \sum_{i=1}^{N} w_i r(x_i) = \sum_{i=1}^{N}\sum_{j=0}^{N-1} \beta_j w_i p_j(x_i) = \beta_0\langle p_0|p_0\rangle \tag{3.49}$$

Donc l'équation (3.44) est satisfaite. Il reste à démontrer que les poids w_i sont positifs. À cet effet, considérons le polynôme de degré $2N-2$

$$\bar{p}_j(x) \stackrel{\text{def}}{=} \prod_{k=1,k\neq j}^{N} (x - x_k)^2 \qquad j = 1,\ldots,N \tag{3.50}$$

Ce polynôme est strictement positif, et donc l'intégrale suivante aussi :

$$0 < \int_a^b \mathrm{d}x\, w(x)\bar{p}_j(x) = \sum_{i=1}^{N} w_i \bar{p}_j(x_i) = w_j \prod_{\substack{k=1\\k\neq j}}^{N} (x_j - x_k)^2 \tag{3.51}$$

(seul le terme $i = j$ de la somme survit). Comme le produit est nécessairement positif, w_j l'est aussi.

Le choix de la classe de polynômes orthogonaux utilisés dépend du type de fonction qu'on veut intégrer. Si on sait que les fonctions d'intérêt sont bien représentées par des polynômes, alors on choisit $w(x) = 1$ et les polynômes correspondants sont les polynômes de Legendre. Par contre, si la fonction d'intérêt $y(x)$ a des singularités ou un comportement tel que $\tilde{y}(x) = y(x)/w(x)$ est doux (ou bien représenté par un polynôme), alors on choisit les polynômes qui sont orthogonaux par le poids $w(x)$.

On démontre l'expression explicite suivante pour les poids w_j :

$$w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{3.52}$$

où $p'_N(x)$ est la dérivée de $p_N(x)$. Les poids w_i et les abscisses x_i sont tabulés pour les polynômes les plus communs, mais on peut également les calculer explicitement.

Le théorème de la quadrature gaussienne nous permet de définir le produit scalaire suivant :

$$\langle\psi|\psi'\rangle_g \stackrel{\text{def}}{=} \sum_{i=1}^{N} w_i \psi(x_i)\psi'(x_i) \tag{3.53}$$

TABLE 3.1 Propriétés des polynômes orthogonaux les plus courants

Nom	intervalle	$w(x)$	relation de récurrence	norme h_j
Legendre	$[-1,1]$	1	$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1}$	$2/(2j+1)$
Tchébychev	$[-1,1]$	$(1-x^2)^{-1/2}$	$T_{j+1} = 2xT_j - T_{j-1}$	$\frac{1}{2}\pi(1+\delta_{j0})$
Hermite	$[-\infty,\infty]$	e^{-x^2}	$H_{j+1} = 2xH_j - 2jH_{j-1}$	$\sqrt{\pi}2^N N!$
Laguerre	$[0,\infty]$	$x^\alpha e^{-x}$	$(j+1)L^\alpha_{j+1} = (-x+2j+\alpha+1)L^\alpha_j - (j+\alpha)L^\alpha_{j-1}$	$\frac{\Gamma(\alpha+j+1)}{N!}$

En autant que le produit des fonctions impliquées soit un polynôme de degré $2N-1$ ou moins, ce produit scalaire est identique au produit (3.33). On montre que la précision de la quadrature gaussienne augmente exponentiellement avec N. C'est là que réside en principe son avantage principal. Notre pratique sera donc d'utiliser la forme (3.53) du produit scalaire, en supposant qu'elle représente effectivement le produit (3.33), et nous ne ferons plus de distinction entre les deux du point de vue de la notation.

3.2.3 Polynômes orthogonaux classiques

Généralement, les polynômes orthogonaux ne sont pas définis comme étant unitaires, mais respectent plutôt une autre condition de normalisation. Par exemple, les polynômes de Legendre prennent la valeur $+1$ à $x = 1$. Cela ne fait pas de différence sur la position des racines, mais change les coefficients de la relation de récurrence (3.35). Ce changement de normalisation n'affecte en rien les propriétés générales que nous avons décrites plus haut, en particulier la relation (3.52). Les propriétés de base des polynômes les plus courants sont indiquées au tableau 3.1.

Polynômes de Legendre

Les polynômes de Legendre sont utiles dans le cas d'un intervalle fini (tout intervalle fini peut être rapporté par une transformation affine à l'intervalle $[-1,1]$). Les racines de $P_N(x)$ dans cet intervalle doivent être trouvées numériquement, par la méthode de Newton par exemple (voir. la section 8.1.4). Ceci est généralement vrai pour tous les polynômes orthogonaux (sauf les polynômes de Tchébychev pour lesquels une expression explicite est connue). Mais la propriété d'entrelacement des racines simplifie beaucoup cette recherche, car on peut procéder par récurrence et on connaît alors deux points entre lesquels chaque racine se trouve avec certitude.

Table 3.2 Racines et poids pour les premiers polynômes de Legendre

N	$\pm x_i$	w_i
2	0.5773502691896258	1.0
3	0	0.8888888888888889
	0.7745966692414834	0.5555555555555556
4	0.3399810435848563	0.6521451548625462
	0.8611363115940526	0.3478548451374539
5	0	0.5688888888888889
	0.5384693101056831	0.4786286704993665
	0.9061798459386640	0.2369268850561891
6	0.2386191860831969	0.4679139345726910
	0.6612093864662645	0.3607615730481386
	0.9324695142031520	0.1713244923791703

On utilise souvent les polynômes de Legendre dans des routines d'intégration numérique, où l'intervalle d'intégration est divisé en segments plus petits, de sorte qu'un petit nombre de points est requis pour calculer l'intégrale sur chaque segment. Les racines et poids des polynômes de Legendre pour de modestes valeurs de N sont alors requises ; elles sont données au tableau 3.2

Polynômes de Tchébychev

Les polynômes de Tchébychev sont en un sens plus simples que les polynômes de Legendre, en raison de leur relation avec les fonctions trigonométriques :

$$T_j(x) = \cos(j\theta) \quad \text{où} \quad x = \cos\theta \tag{3.54}$$

La relation d'orthogonalité se comprend alors facilement via les fonctions trigonométriques :

$$\int_{-1}^{1} \mathrm{d}x \, \frac{T_j(x)T_k(x)}{\sqrt{1-x^2}} = \int_0^{\pi} \mathrm{d}\theta \, \cos(j\theta)\cos(k\theta) = \delta_{jk}\frac{\pi}{2}(1+\delta_{j0}) \tag{3.55}$$

La représentation d'une fonction sur la base des polynômes de Tchébychev s'apparente donc à une série de Fourier ; cependant la fonction n'est pas périodique en x, mais en θ ! Les polynômes de Tchébychev sont encore plus utiles que les polynômes de Legendre pour représenter une fonction définie sur un intervalle fini. La relation (3.54) nous indique immédiatement où se trouvent les racines x_i de T_N :

$$x_i = \cos\left(\frac{\pi}{2}\frac{2i-1}{N}\right) \tag{3.56}$$

Les premiers polynômes de Tchébychev sont :

$$\begin{aligned} T_0(x) &= 1 & T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 & T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 & T_5(x) &= 16x^5 - 20x^3 + 5x \end{aligned} \tag{3.57}$$

Exercice 3.3

Montrez que si on définit les polynômes de Tchébychev par la relation (3.54), alors la relation de récurrence $T_{j+1} = 2xT_j - T_{j-1}$ s'ensuit.

3.3 Transformées de Fourier rapides

Une fonction complexe $\psi(x)$ définie sur un intervalle de longueur L peut être représentée par ses coefficients de Fourier $\tilde{\psi}_k$, définis comme suit :

$$\psi(x) = \sum_{k\in\mathbb{Z}} \mathrm{e}^{2\pi i k x/L}\tilde{\psi}_k \qquad \tilde{\psi}_k = \frac{1}{L}\int_{-L/2}^{L/2} \mathrm{d}x\, \mathrm{e}^{-2\pi i k x/L}\psi(x) \tag{3.58}$$

Cette représentation de la fonction $\psi(x)$ est explicitement périodique de période L. Dans la limite $L \to \infty$, on définit une variable continue $q = 2\pi k/L$ et les relations ci-dessus deviennent la transformation de Fourier (TdF) :

$$\psi(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{d}q\, \mathrm{e}^{iqx}\tilde{\psi}(q) \quad \text{et} \quad \tilde{\psi}(q) = \int_{-\infty}^{\infty} \mathrm{d}x\, \mathrm{e}^{-iqx}\psi(x) \quad \text{où} \quad \tilde{\psi}_n \to \frac{1}{L}\tilde{\psi}(2\pi k/L) \tag{3.59}$$

Bien que la notation utilisée fasse référence à une interprétation spatiale de la transformation, ce qui suit est bien sûr également valable dans le domaine temps-fréquence ; il suffit d'adapter la notation en conséquence.

3.3.1 Transformées de Fourier discrètes

Numériquement, une fonction $\psi(x)$ sera représentée par un ensemble de N valeurs ψ_j ($j = 0, \ldots, N-1$) associées à une grille régulière de pas a et d'étendue $L = Na$. L'équivalent dans ce cas de la relation (3.58) est la *transformée de Fourier discrète* :

$$\psi_j = \frac{1}{N}\sum_{k=0}^{N-1} \mathrm{e}^{2\pi i j k/N}\tilde{\psi}_k \qquad \tilde{\psi}_k = \sum_{j=0}^{N-1} \mathrm{e}^{-2\pi i j k/N}\psi_j \tag{3.60}$$

On peut également exprimer ces relations en fonction de la N^{me} racine complexe de l'unité $\omega \stackrel{\text{def}}{=} \mathrm{e}^{-2\pi i/N}$:

$$\psi_j = \frac{1}{N}\sum_{k=0}^{N-1} \bar{\omega}^{jk}\tilde{\psi}_k \qquad \tilde{\psi}_k = \sum_{j=0}^{N-1} \omega^{jk}\psi_j \qquad \bar{\omega} \stackrel{\text{def}}{=} \omega^* \tag{3.61}$$

Remarques :

1. La quantité ψ_j est périodique de période N : ψ_{j+N} doit être identifié à ψ_j.
2. Idem pour $\tilde{\psi}_k$, aussi périodique de période N.
3. La correspondance avec une fonction continue est la suivante :

$$x \to ja \qquad \psi_j \to \frac{L}{a}\psi(x) \tag{3.62}$$

La transformée de Fourier discrète peut être vue comme l'application d'une matrice $N \times N$ sur le vecteur ψ :

$$\tilde{\psi} = U\psi \quad \text{où} \quad U_{jk} = \omega^{jk} \tag{3.63}$$

La relation inverse, $\psi = \frac{1}{N}U^*\tilde{\psi}$, provient de l'identité suivante :

$$U^\dagger U = N \quad \text{ou encore} \quad \boxed{\sum_{k=0}^{N} \omega^{(j-j')k} = N\delta_{jj'}} \tag{3.64}$$

La preuve de cette relation est très simple, car en général, pour tout complexe z,

$$\sum_{k=0}^{N-1} z^k = \frac{1-z^N}{1-z} \quad \text{et donc} \quad \sum_{k=0}^{N-1} \omega^{(j-j')k} = \frac{1-\omega^{(j-j')N}}{1-\omega^{(j-j')}} \tag{3.65}$$

Mais le numérateur de cette expression est toujours nul, car $\omega^N = 1$. Le dénominateur n'est nul que si $j = j'$; dans ce dernier cas la fraction est indéterminée mais il est trivial d'évaluer directement la somme, qui donne manifestement N, d'où le résultat (3.64).

3.3.2 Algorithme de Danielson et Lanczos (ou Cooley-Tukey)

À première vue, il semble que le nombre d'opérations impliquées dans une TdF discrète soit d'ordre $\mathcal{O}(N^2)$, comme tout produit matrice-vecteur. Or il est possible d'effectuer la TdF discrète à l'aide d'un nombre d'opérations beaucoup plus petit, d'ordre $\mathcal{O}(N \log N)$.[2] Cette différence est considérable : si un signal est échantilloné à l'aide de 1 000 points, ce qui n'est pas énorme, le rapport entre N^2 et $N \log N$ est de l'ordre de 100. Le gain de performance est énorme. Les méthodes qui tirent parti de cette réduction dans le nombre d'opérations portent le nom de *transformées de Fourier rapides* (TdFR, en anglais *fast Fourier transforms*, ou FFT). L'algorithme de TdFR est l'un des plus utilisés dans la vie courante : traitement du son et de l'image, etc. Les téléphones mobiles partout dans le monde effectuent des TdFR constamment. La TdFR a été reconnue comme l'un des dix algorithmes les plus importants de l'histoire du calcul.[3]

La version la plus simple de l'algorithme de TdFR se présente lorsque le nombre de points est une puissance de deux : $N = 2^M$. C'est ce que nous allons supposer dans la suite. La clé de l'algorithme est la possibilité d'écrire la deuxième des relations (3.61) comme la somme

2. Ce fait a été découvert plusieurs fois depuis l'époque de Gauss, en particulier par Danielson et Lanczos en 1942. Ce sont cependant les noms de Cooley et Tukey qui sont habituellement associés à cette découverte (1965).
3. Par la revue *Computing in Science and Engineering*, Jan/Feb 2000.

de deux transformées de Fourier comportant $N/2$ points : l'une comportant les termes pairs, l'autre les termes impairs :

$$\tilde{\psi}_k = \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi 2ijk/N} \psi_{2j} + \omega^k \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi 2ijk/N} \psi_{2j+1} \tag{3.66}$$

Cela entraîne qu'une TdF à N points est la somme de deux TdF à $N/2$ points, si on arrange le vecteur ψ à N composantes en deux vecteurs $\psi^{(0)}$ et $\psi^{(1)}$ de $N/2$ composantes chacun, contenant respectivement les composantes paires et impaires de ψ. Dans chacune des deux TdF, la valeur de k doit être prise modulo $N/2$, sauf dans l'exposant ω^k.

Chacun des deux termes de (3.66) peut, à son tour, est séparé en deux TdF à $N/4$ termes, et ainsi de suite jusqu'à ce que le problème soit réduit à une some de N TdFs à 1 terme chacune. À l'étape 2, les quatre sous-vecteurs de longueur $N/4$ peuvent être notés $\psi^{(00)}$, $\psi^{(01)}$, $\psi^{(10)}$ et $\psi^{(11)}$. Le sous-vecteur $\psi^{(00)}$ contient les termes pairs de $\psi^{(0)}$, dont les indices originaux sont 0 modulo 4, alors que $\psi^{(01)}$ contient les termes impairs de $\psi^{(0)}$, dont les indices originaux sont 2 modulo 4, et ainsi de suite. À la dernière étape, chaque sous-vecteur ne contient qu'une seule composante et est indexé par une représentation binaire à M bits, par exemple $\psi^{(0011001011)} = \psi^J$ pour $M = 10$ et $N = 1024$. Cet élément est indexé par un entier J dont la représentation binaire (0011001011 dans notre exemple) est précisément l'inverse de celle de l'indice j du même élément dans le vecteur original ψ (c'est-à-dire $1101001100 = 844$). Si on dispose les éléments de ψ non pas dans leur ordre original (celui des indices j), mais dans l'ordre des indices binaires-inversés J, alors les TdF partielles qui doivent être réalisées à chaque étape sont plus facilement effectuées.

Cet algorithme est illustré dans le cas $N = 8$ sur la figure 3.7. La dernière colonne contient les 8 valeurs $\tilde{\psi}_k$, dans l'ordre des k croissants. Le passage de la dernière colonne à l'avant dernière se fait en vertu de l'équation (3.66). Les valeurs de k sont groupées en paires séparées de $N/2$, c'est-à-dire apparaissent comme ω^k et $-\omega^k$. La somme (3.66) apparaît donc dans le diagramme comme une somme de deux termes avec poid relatif ω^k ($k = 0, 1, 2, 3$), ou comme une différence des deux même termes. Cette double combinaison (somme et différence avec poids relatif ω^k) forme graphiquement ce qu'on appelle un «papillon» (voir fig. 3.8). Les éléments de la troisième colonne de la figure 3.7 forment deux groupes de TdF partielles, et une autre application de l'équation (3.66) sur ces deux séries nous amène à la deuxième colonne. À cette étape les papillons ne comportent que 4 valeurs de l'exposant, soit $\pm\omega^2 = \pm i$ et $\pm\omega^0 = \pm 1$. Il en résulte 4 TdF partielles. Enfin, une dernière application de l'équation (3.66) nous amène à a première colonne, qui ne comporte que des papillons appliquant les combinaisons $\pm\omega^0$. La première colonne est faite des éléments de la fonction directe dans l'ordre des indices binaires-inversés J.

Évidemment, l'algorithme de TdFR doit procéder dans l'ordre inverse, c'est-à-dire partir des ψ_j pour aboutir aux $\tilde{\psi}_k$. Ceci se fait en procédant aux étapes suivantes :

1. Placer les ψ_j dans l'ordre des indices binaires-inversés. Soit $I(j)$ l'indice obtenu en inversant l'ordre des bits de j. Comme le carré de l'inversion est l'opérateur identité, on a $I(I(j)) = j$ et le changement d'ordre se fait simplement en appliquant les transpositions

$$\psi_j \leftrightarrow \psi_{I(j)} \tag{3.67}$$

L'opération ne nécessite pratiquement pas de stockage supplémentaire et le tableau inversé remplace simplement le tableau original.

2. On effectue ensuite une boucle externe sur les niveau. Au premier niveau, on effectue tous les «papillons» au sein des paires successives d'éléments du tableau, autrement dit entre les paires d'indices qui ne diffèrent que par le première bit. Encore une fois le tableau est remplacé par un nouveau tableau sans besoin d'espace supplémentaire (sauf une valeur transitoire).
3. On continue dans cette boucle en passant à la colonne suivante. Les opérations papillons se font maintenant en combinant les indices qui ne diffèrent que par leur deuxième bit. Le tableau de l'itération précédente est remplacé par un nouveau tableau, et ainsi de suite.
4. Quand cette boucle interne est terminée, le tableau original a été remplacé par sa transformée de Fourier.
5. Chaque papillon comporte le même nombre d'opérations arithmétiques, et le nombre de papillons est $N/2$ à chaque étape de la boucle, qui comporte elle-même $\log_2 N$ étapes ; donc la complexité algorithmique de la TdFR est $\mathcal{O}(N \log_2 N)$.

3.3.3 Cas des dimensions supérieures

La TdF peut être appliquée à une fonction de plus d'une variable. Considérons par exemple le cas de deux dimensions et introduisons les variables discrètes x et y, chacune allant de 0 à $N-1$, ainsi que les indices réciproques correspondants k_x et k_y. La TdF prend alors la forme suivante :

$$\psi_{x,y} = \frac{1}{N^2} \sum_{k_x,k_y} \bar{\omega}^{xk_x+yk_y} \tilde{\psi}_{k_x,k_y} \qquad \tilde{\psi}_{k_x,k_y} = \sum_{x,y} \omega^{xk_x+yk_y} \psi_{x,y} \tag{3.68}$$

La façon la plus directe de procéder à la TdFR est de commencer par effectuer N TdFR sur la variable y, une pour chaque valeur de x. On obtient alors un objet intermédiaire Ψ_{x,k_y} qui réside dans l'espace réciproque en y, mais dans l'espace direct en x. Ensuite on procède à N TdFR sur la variable x, une pour chaque valeur de k_y. Il y a donc $2N$ TdFR à calculer, chacune de complexité $N \log_2 N$, ce qui donne une complexité totale $2N^2 \log_2 N = N^2 \log_2 N^2$, où N^2 est le nombre de points échantillonnés en deux dimensions.

3.3.4 Fonctions réelles

L'algorithme de TdFR peut être appliqué à des fonctions réelles tel quel, mais il entraîne un certain gaspillage d'espace et de temps de calcul, car une fonction réelle discrétisée ψ_j ne comporte que la moitié des degrés de liberté d'une fonction complexe. D'autre part, sa TdF $\tilde{\psi}_k$ est toujours complexe, mais jouit de la symétrie suivante :

$$\tilde{\psi}_k^* = \tilde{\psi}_{N-k} \tag{3.69}$$

(en particulier, $\tilde{\psi}_0$ et $\tilde{\psi}_{N/2}$ sont réels).

Deux procédures différentes peuvent être utilisées pour rendre le calcul plus efficace :

1. Calculer deux TdFR simultanément. Souvent les TdF doivent être faites en série. Si on combine deux foncions réelles ψ' et ψ'' en une seule fonction complexe $\psi = \psi' + i\psi''$, appliquer la TdFR sur ψ nous permet d'obtenir simultanément les TdF des parties réelle et imaginaire, en appliquant la formule simple

$$\tilde{\psi}'_k = \frac{1}{2}\left(\tilde{\psi}_k + \tilde{\psi}^*_{N-k}\right) \qquad \tilde{\psi}''_k = \frac{1}{2i}\left(\tilde{\psi}_k - \tilde{\psi}^*_{N-k}\right) \tag{3.70}$$

2. Combiner les partie paire et impaire de la fonction ψ en une seule fonction complexe Ψ de $N/2$ composantes : $\Psi = \psi^{(0)} + i\psi^{(1)}$. On montre que

$$\tilde{\psi}_k = \frac{1}{2}\left(\tilde{\Psi}_k + \tilde{\Psi}^*_{N/2-k}\right) - \frac{i}{2}\left(\tilde{\Psi}_k - \tilde{\Psi}^*_{N/2-k}\right) \mathrm{e}^{2\pi ik/N} \tag{3.71}$$

Exercice 3.4

Démontrez les relations (3.70) et (3.71).

3.3.5 Annexe : Code

Nous ne présenterons pas de code explicite de la transformée de Fourier rapide. Nous allons plutôt nous fier aux librairies existantes, en particulier GSL. Le code qui suit donne un exemple d'application d'une TdFR sur une fonction réelle : Il définit une fonction à travers un foncteur et applique un filtre passe-bas à cette fonction en appliquant premièrement une TdFR sur la fonction, ensuite en tronquant la transformée $\tilde{\psi}$, et enfin il calcule la transformée inverse. Les appels à GSL sont particuliers à des fonctions réelles et mettent en oeuvre la deuxième stratégie expliquée ci-dessus pour les fonctions réelles.

Code 3.1 : Filtre passe-bas (FFT.cpp)

```
1  #include <iostream>
2  #include <fstream>
3  #include <cmath>
4
5  #include <gsl/gsl_fft_real.h>
6  #include <gsl/gsl_fft_halfcomplex.h>
7  #include "read_parameter.h"
8  #include "Vector.h"
9
10 using namespace std;
11
12  Foncteur définissant la fonction à filtrer (fonction pas)
13 struct func{
14   double xc; valeur de x au-delà de laquelle la fonction est nulle
15   func(double _xc) : xc(_xc) {}
16   double operator() (double x){ return (x<xc)? 1.0: 0.0;}
17 };
18
```

```
19 int main(){
20
21   fstream fin("para.dat");
22
23   int log2n;  logarithme en base 2 du nombre de points
24   double min, max, L, xc, wc;
25   fin >> "log2n" >> log2n;
26   fin >> "min" >> min; borne inférieure du domaine de la fonction
27   fin >> "max" >> max; borne supérieure du domaine de la fonction
28   L = max - min; largeur du domaine
29
30   fin >> "xc" >> xc; spécifique à la fonction pas
31   fin >> "wc" >> wc;
32   fin.close();
33
34   int N = 1 << log2n; nombre de points = 2^log2N
35   double h = (L*1.0)/N; pas dans l'espace direct
36
37   ofstream fout("data.dat");
38
39   func f(xc);
40
41   Vector<double> data(N); initialisation du tableau contenant la fonction
42   for(int i=0; i<N; i++){
43     double x = min+i*h;
44     data[i] = f(x);
45     fout << x << '\t' << data[i] << endl;
46   }
47   fout << "\n\n";
48
49   gsl_fft_real_radix2_transform(data.array(), 1, N); transformée directe
50   int m = N/2;
51   fout << data[0] << '\t' << 0 << endl;
52   for(int i=1; i<m; i++) fout << data[i] << '\t' << data[N-i] << endl;
53   fout << data[m] << '\t' << 0 << "\n\n\n";
54
55    filtre  passe bas
56   int ic = (int)floor(L*wc/(2*M_PI));
57   cout << "Effacement des composantes d'indice > " << ic << endl;
58   if(ic<1) throw("valeur de wc trop basse!");
59   if(ic>=m) throw("valeur de wc trop élevée!");
60   for(int i=ic; i<m; i++) data[i] = data[N-i] = 0.0;
61   data[m] = 0.0;
62
63   gsl_fft_halfcomplex_radix2_inverse(data.array(),1,N); transformée inverse
64   for(int i=0; i<N; i++) fout << data[i] << endl;
65
66   fout.close();
67   cout << "Fin normale du programme\n";
```

```
68 }
```

Quelques remarques :

1. Les appels dans ce code sont spécifique à un nombre de points égal à une puissance de 2. Des appels différents sont nécessaires si cette condition n'est pas remplie.
2. La TdF est stockée dans le même tableau que la fonction d'entrée : après l'appel à la fonction
   `gsl_fft_real_radix2_transform()`,
   `data[i]` est la partie réelle de la TdF ($i = 0, \ldots, N/2$) alors que `data[N-i]` est sa partie imaginaire. Après l'appel à la transformée inverse
   `gsl_fft_halfcomplex_radix2_inverse()`,
   le tableau `data[]` contient la fonction filtrée.
3. Le code écrit sur un fichier de sortie la fonction de départ, sa transformée de Fourier et enfin la fonction filtrée.

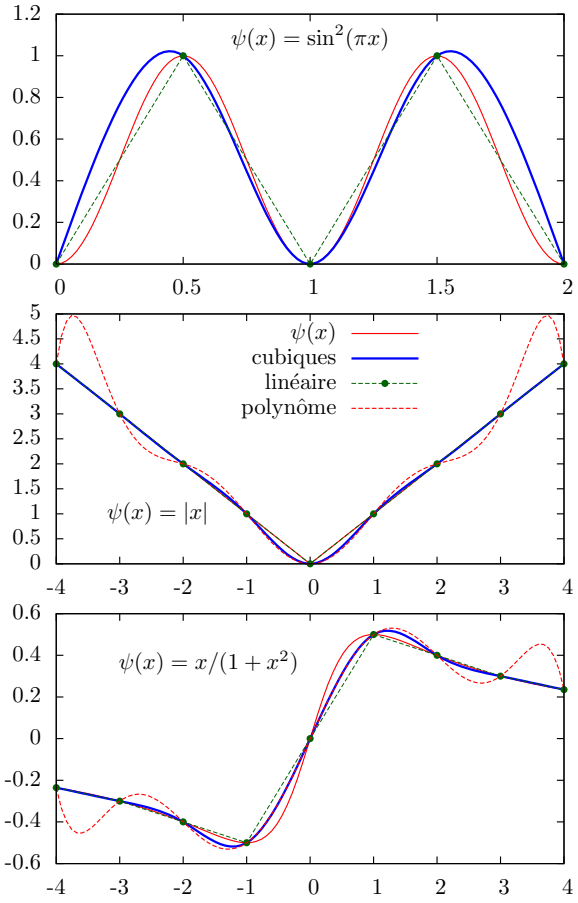


Figure 3.2 Exemples d'interpolation à l'aide de cubiques raccordées. Remarquons que les cubiques naturelles s'accordent mal aux extrémités lorsque la dérivée seconde est non négligeable à ces points (graphique du haut) et qu'elles représentent en général assez mal les points où la dérivée seconde est infinie (graphique du milieu). Les points de la grille sont indiqués, et sont peu nombreux. L'interpolant de Lagrange basé sur les 9 points de l'échantillon est aussi illustré ; son comportement est très oscillant par rapport aux cubiques raccordées.

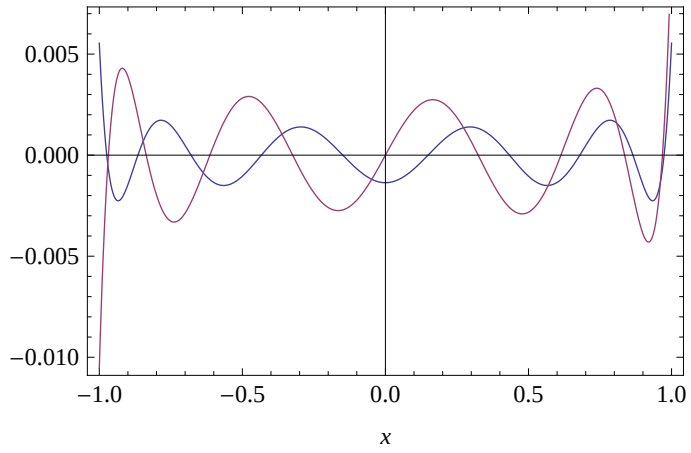


Figure 3.3 Polynômes unitaires de Legendre $p_{10}(x)$ (en bleu) et $p_9(x)$ (en rouge), avec leurs racines entrelacées.

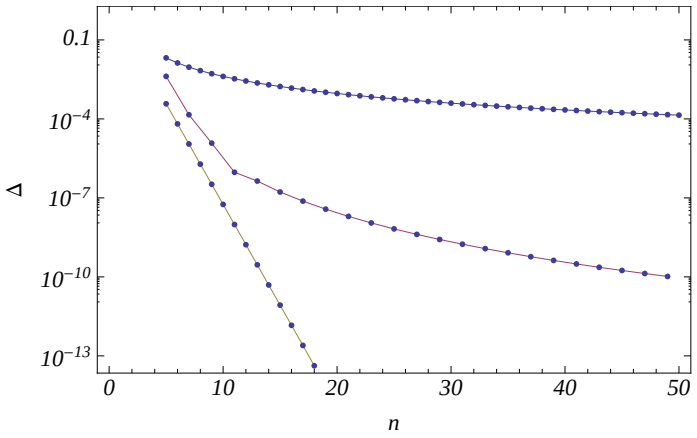


Figure 3.4 Graphique semi-logarithmique de l'erreur effectuée sur l'intégrale $\int_{-1}^{1} \mathrm{d}x\,(1+x)/(1+x^2)$ en fonction du nombre n d'évaluations de l'intégrant (le résultat exact est $\pi/2$ et donc l'erreur est connue exactement). La courbe supérieure provient de la méthode du trapèze ; la courbe médiane de la méthode de Simpson et la courbe du bas de l'intégrale gaussienne avec polynômes de Legendre. Notez que dans ce dernier cas, l'erreur diminue exponentiellement avec n.

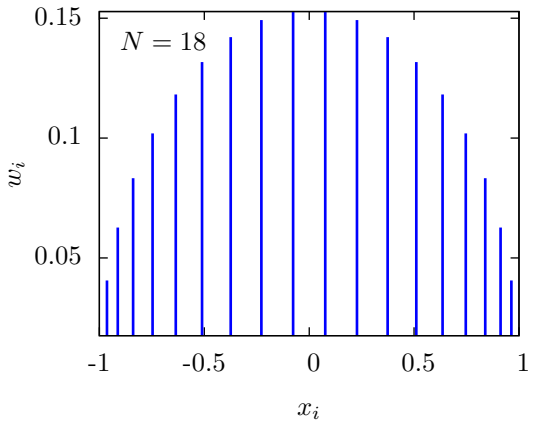


FIGURE 3.5 Les racines et poids du polynôme de Legendre $P_{18}(x)$.

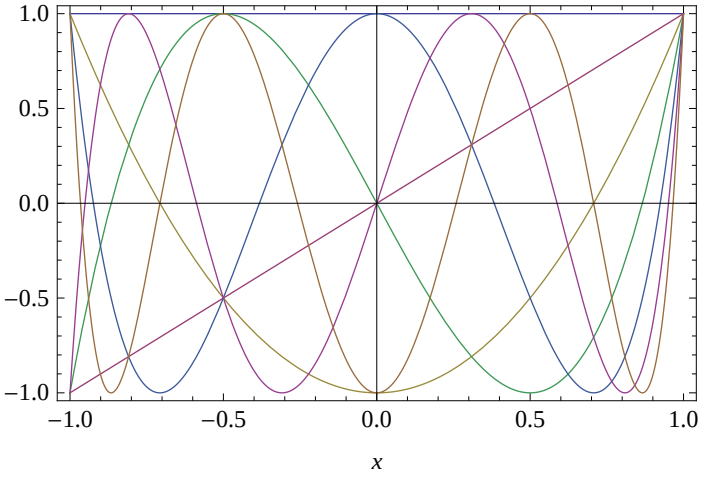


FIGURE 3.6 Les 7 premiers polynômes de Tchébychev

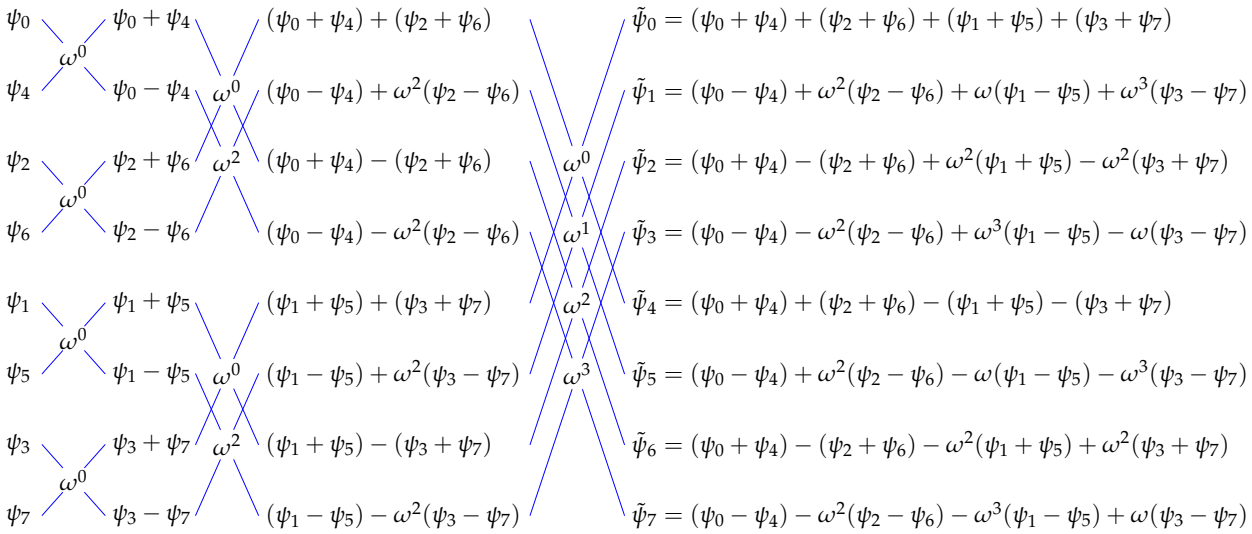


Figure 3.7

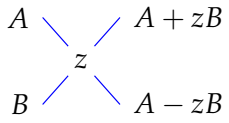


Figure 3.8 L'opérateur du «papillon» prend les deux données A et B et les remplace par les combinaisons $A \pm zB$ comme illustré.

Chapitre 4

Problèmes aux limites

Les problèmes de la physique classique et beaucoup de ses applications en génie peuvent être formulés via des équations différentielles. Souvent ces équations différentielles n'impliquent que des coordonnées spatiales, c'est-à-dire sont indépendantes du temps. Dans ce cas, leur solution est la plupart du temps contrainte par des conditions aux limites : c'est la valeur de la fonction recherchée, ou ses dérivées, sur la frontière du domaine qui détermine la solution. On parle alors de *problèmes aux limites*. Nous allons nous concentrer sur deux méthodes de solution des problèmes aux limites : la méthode des éléments finis, et les méthodes spectrales.

Les méthodes décrites dans ce chapitre pourront ensuite être appliquées au cas d'équations différentielles aux dérivées partielles qui dépendent du temps : la dépendance spatiale des fonctions impliquées étant alors représentée par éléments finis ou par représentation spectrale, et la dépendance temporelle faisant l'objet d'une traitement différent, purement séquentiel, comme ce qui a été fait au chapitre 2.

4.1 Éléments finis : dimension 1

La méthode des *éléments finis* repose sur le développement d'une fonction $\psi(x)$ sur une base de fonctions localisées. Elle est très utilisée dans la solution des équations aux dérivées partielles, et se combine habituellement à une discrétisation de l'espace en fonction de simplexes (ou de triangles en deux dimensions).

4.1.1 Base de fonctions tentes en dimension 1

Commençons par le cas unidimensionnel. Définissons, sur le segment $[0, \ell]$, en se basant sur une grille de points $\{x_i\}$ qui n'est pas nécessairement régulière, un ensemble de fonctions $u_i(x)$

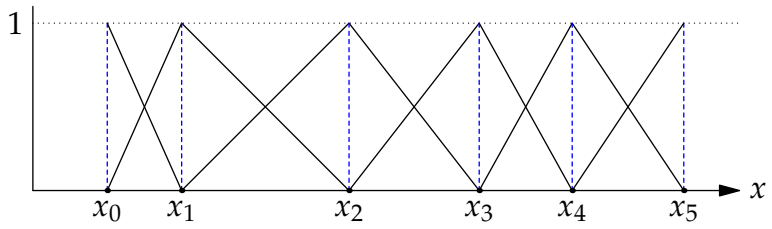


Figure 4.1 Fonctions-tentes en dimension 1, avec conditions aux limites ouvertes.

définies comme suit :

$$u_i(x) = \begin{cases} 0 & \text{si} \quad x > x_{i+1} \quad \text{ou} \quad x < x_{i-1} \\ \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & \text{si} \quad x_{i-1} < x < x_i \\ \dfrac{x - x_{i+1}}{x_i - x_{i+1}} & \text{si} \quad x_i < x < x_{i+1} \end{cases} \tag{4.1}$$

Ces fonctions linéaires par morceaux sont appelées *fonctions tentes* en raison de leur forme. Voir la figure 4.1 pour un exemple. Le cas des extrémités ($i = 0$ et $i = N - 1$) est particulier si des conditions aux limites ouvertes sont utilisées. Dans ce cas, la définition est la même, à la différence que la fonction s'annule en dehors de l'intervalle $[0, \ell]$

Nous allons fréquemment utiliser la notation de Dirac dans cette section et les suivantes. La fonction $u_i(x)$ correspond à alors au produit bilinéaire $\langle x|u_i\rangle$ du vecteur abstrait $|u_i\rangle$ représentant la fonction u_i, et de la fonction propre de la position au point x ; l'utilisation de cette notation sera tout-à-fait semblable à ce qui est pratiqué en mécanique quantique, à la différence que nous utiliserons un espace de fonctions sur les réels et non sur les complexes. Le produit bilinéaire sera, à moins d'avis contraire, défini par l'intégrale

$$\langle f|g\rangle = \int_0^\ell \mathrm{d}x\, f(x)g(x) \tag{4.2}$$

La propriété fondamentale des fonctions tentes est qu'elles sont nulles en dehors du domaine $[x_{i-1}, x_{i+1}]$ et égales à l'unité à $x = x_i$. Ces propriétés entraînent premièrement que

$$u_i(x_j) = \delta_{ij} \quad \text{ou, en notation de Dirac,} \quad \langle x_j|u_i\rangle = \delta_{ij} \tag{4.3}$$

et deuxièmement que

$$M_{ij} \stackrel{\text{def}}{=} \langle u_i|u_j\rangle = \int_0^\ell \mathrm{d}x\, u_i(x)u_j(x) \tag{4.4}$$

n'est non nul que pour les sites i et j qui sont des voisins immédiats. La matrice M_{ij} est appelée *matrice de masse*. Comme $M_{ij} \neq \delta_{ij}$, les fonctions tentes me forment pas une base orthonormée.

L'ensemble des fonctions $u_i(x)$ génère un sous-espace $\mathscr{V}_u$ de l'espace des fonctions définies dans l'intervalle $[0, \ell]$. Toute fonction appartenant à ce sous-espace admet par définition le développement suivant :

$$\psi(x) = \sum_i \psi_i u_i(x) \quad \text{ou, en notation de Dirac,} \quad |\psi\rangle = \sum_i \psi_i |u_i\rangle \tag{4.5}$$

Comme les fonctions tentes ont la propriété $u_i(x_j) = \delta_{ij}$, on constate immédiatement que

$$\psi_i = \psi(x_i) \tag{4.6}$$

L'approximation que nous ferons dans la résolution de problèmes aux limites ou d'équations différentielles aux dérivées partielles est que les solutions à ces équations appartiendront au sous-espace $\mathscr{V}_u$. Autrement dit, nous ne procéderons pas à une approximation de l'équation différentielle elle-même – par exemple en remplaçant les dérivées continues par des dérivées discrètes – mais nous allons restreindre l'espace de fonctions dans lequel nous chercherons des solutions.

Dans le sous-espace $\mathscr{V}_u$, le produit bilinéaire s'exprime comme suit :

$$\langle\psi|\psi'\rangle = \sum_{i,j}\langle u_i|u_j\rangle\psi_i\psi'_j = \widetilde{\psi}M\psi' \tag{4.7}$$

où $\widetilde{\psi}$ est le transposé du vecteur ψ. Ce produit scalaire étant défini positif, la matrice M est, elle-aussi, définie positive.

Considérons ensuite un opérateur différentiel $\mathscr{L}$, comme ceux qui apparaissent dans une équation différentielle. Par exemple, $\mathscr{L}$ pourrait être une combinaison de dérivées :

$$\mathscr{L} = u(x) + v(x)\partial_x + w(x)\partial_x^2 \tag{4.8}$$

L'action de l'opérateur $\mathscr{L}$ n'est pas fermée dans $\mathscr{V}_u$, c'est-à-dire que si $|\psi\rangle \in \mathscr{V}_u$, la fonction $\mathscr{L}|\psi\rangle$ n'appartient pas complètement à $\mathscr{V}_u$, mais possède un résidu $|\perp\rangle$ à l'extérieur de $\mathscr{V}_u$. Que veut-on dire exactement par là ? Cette affirmation n'a de sens, en fait, que via le produit scalaire : le résidu $|\perp\rangle$ doit être orthogonal à $\mathscr{V}_u$: $\langle u_i|\perp\rangle = 0$. On peut donc écrire, pour chaque fonction tente,

$$\mathscr{L}|u_i\rangle = \sum_j L^c_{ji}|u_j\rangle + |\perp\rangle \qquad (\langle u_k|\perp\rangle = 0) \tag{4.9}$$

où la matrice L^c définit l'action de $\mathscr{L}$ sur les fonctions tentes. En particulier, sur une fonction quelconque de $\mathscr{V}_u$, on a

$$\mathscr{L}|\psi\rangle = \sum_i \psi_i\mathscr{L}|u_i\rangle + |\perp\rangle = \sum_{i,j}\psi_i L^c_{ji}|u_j\rangle + |\perp\rangle \overset{\text{def}}{=} \sum_j \psi'_j|u_j\rangle + |\perp\rangle \tag{4.10}$$

(notons que $|\perp\rangle$ ne désigne pas une fonction en particulier, mais toute fonction orthogonale à $\mathscr{V}_u$). Donc l'image de ψ par $\mathscr{L}$ est $\psi' = L^c\psi$ (en notation vectorielle) lorsque projeté sur $\mathscr{V}_u$.

Définissons maintenant la matrice L ainsi :

$$L_{ki} \overset{\text{def}}{=} \langle u_k|\mathscr{L}|u_i\rangle = \sum_j L^c_{ji}\langle u_k|u_j\rangle = \sum_j M_{kj}L^c_{ji} = (ML^c)_{ki} \quad \text{ou} \quad \boxed{L = ML^c} \tag{4.11}$$

La distinction entre les matrices L et L^c provient du fait que les fonctions de base $|u_i\rangle$ ne forment pas une base orthonormée.

Exercice 4.1: matrice de masse (dimension 1)

Montrez que

$$\langle u_i|u_j\rangle = \begin{cases} \frac{1}{6}|x_i - x_j| & \text{si} \quad |i-j| = 1 \\ \frac{1}{3}|x_{i+1} - x_{i-1}| & \text{si} \quad i = j,\ i \neq 0,\ i \neq N-1 \end{cases} \quad \text{et} \quad \begin{aligned} \langle 0|0\rangle &= \tfrac{1}{3}|x_1 - x_0| \\ \langle N-1|N-1\rangle &= \tfrac{1}{3}|x_{N-1} - x_{N-2}| \end{aligned} \tag{4.12}$$

4.1.2 Solution d'un problème aux limites en dimension 1

Appliquons la représentation en éléments finis à la solution d'un problème aux limites, c'est-à-dire d'une équation différentielle pour la fonction $\psi(x)$, avec des conditions précises sur la valeur de $\psi(x)$ ou de ses dérivées à $x = 0$ et $x = \ell$.

Nous allons écrire l'équation différentielle sous la forme générale suivante :

$$\mathscr{L}|\psi\rangle = |\rho\rangle \tag{4.13}$$

où $\mathscr{L}$ est un opérateur différentiel, qu'on peut supposer linéaire pour le moment, et $|\rho\rangle$ une fonction déterminée, souvent appelé *fonction de charge* ou *vecteur de charge*. Par exemple, dans le cas de l'équation de Helmholtz avec des sources émettrices distribuées selon la densité $\rho(x)$,

$$\psi''(x) + k^2\psi(x) = \rho(x) \quad \text{et donc} \quad \mathscr{L} = \frac{\mathrm{d}^2}{\mathrm{d}x^2} + k^2 \tag{4.14}$$

Nous cherchons une solution approchée de l'équation (4.13) sous la forme de valeurs $\{\psi_i\}$ définies sur la grille. Nous allons en fait considérer une *forme faible* de l'équation différentielle :

$$\langle w_j|\mathscr{L}|\psi\rangle = \int_0^\ell \mathrm{d}x\, w_j(x)\mathscr{L}\psi = \langle w_j|\rho\rangle \tag{4.15}$$

où $w_j(x)$ est une fonction ou une collection de fonctions qu'il faut spécifier et qui définit précisément la forme faible utilisée. On qualifie cette forme de *faible*, parce que toute solution à l'équation (4.13) est nécessairement une solution à l'équation (4.15), alors que l'inverse n'est pas vrai.

On considère généralement les deux approches suivantes :

1. La méthode de **collocation** : On suppose alors que $|w_i\rangle = |x_i\rangle$, c'est-à-dire $w_i(x) = \delta(x - x_i)$. Ceci équivaut à demander que l'équation différentielle (4.13) soit respectée exactement sur la grille et mène à la relation suivante :

   $$\langle x_i|\mathscr{L}|\psi\rangle = \langle x_i|\rho\rangle = \rho_i \tag{4.16}$$

   où

   $$\langle x_i|\mathscr{L}|\psi\rangle = \langle x_i|\left(L^c_{kj}\psi_j|u_k\rangle + |\perp\rangle\right) = L^c_{ij}\psi_j + \langle x_i|\perp\rangle \tag{4.17}$$

   En négligeant le résidu $|\perp\rangle$, nous avons donc la relation matricielle

   $$\boxed{L^c\psi = \rho} \tag{4.18}$$

2. La méthode de **Galerkin** : On prend plutôt $|w_i\rangle = |u_i\rangle$, ce qui revient à demander que l'équation différentielle soit respectée en moyenne sur le domaine de chaque fonction tente. Il s'ensuit que
$$\langle u_i|\mathscr{L}|\psi\rangle = \langle u_i|\rho\rangle \tag{4.19}$$
où
$$\langle u_i|\mathscr{L}|\psi\rangle = \langle u_i|\left(L^c_{kj}\psi_j|u_k\rangle + |\perp\rangle\right) = M_{ik}L^c_{kj}\psi_j = L_{ij}\psi_j \tag{4.20}$$
On obtient donc la relation
$$(L\psi)_i = \langle u_i|\rho\rangle = \langle u_i|\left(\sum_j \rho_j|u_j\rangle + |\perp\rangle\right) = \sum_j M_{ij}\rho_j \quad \text{ou encore} \quad \boxed{L\psi = M\rho} \tag{4.21}$$
La subtilité ici est que ρ_i ne représente pas exactement la valeur $\rho(x_i)$, mais plutôt la valeur à x_i de la projection sur $\mathscr{V}_u$ de la fonction ρ.

Comme $L = ML^c$, les deux méthodes semblent superficiellement équivalentes. Cependant une approximation différente a été faite dans chacune avant d'arriver aux équations matricielles équivalentes $L^c\psi = \rho$ et $L\psi = M\rho$. Dans la méthode de collocation, nous avons traité $\rho(x)$ exactement mais négligé le résidu $|\perp\rangle$ résultant de l'application de $\mathscr{L}$ sur les fonctions tentes. Dans la méthode de Galerkin, ce résidu disparaît de lui-même, mais par contre seule la projection de $\rho(x)$ sur l'espace $\mathscr{V}_u$ est prise en compte.

Imposition des conditions aux limites de Dirichlet

La discussion qui précède a passé sous silence la question des conditions aux limites. Elle est certainement valable lorsqu'on impose des conditions aux limites périodiques, mais doit être raffinée dans les autres cas. Nous allons supposer ici qu'on impose à la fonction $\psi(x)$ des valeurs particulières à $x = 0$ et $x = \ell$ (conditions aux limites de type *Dirichlet*).

Supposons qu'on réordonne les indices vectoriels (et matriciels) de manière à ce que les indices associés à la frontière F apparaissent avant les indices associés à l'intérieur I. Un vecteur ψ est alors composé de deux parties :
$$\psi = \begin{pmatrix} \psi^F \\ \psi^I \end{pmatrix} \tag{4.22}$$
où ψ^F est un vecteur à 2 composantes (pour les deux points à la frontière en dimension 1) et ψ^I un vecteur à $N-2$ composantes, pour les points intérieurs de l'intervalle. Une matrice L serait de même décomposée comme suit :
$$L = \begin{pmatrix} L^F & L^{FI} \\ L^{IF} & L^I \end{pmatrix} \tag{4.23}$$
où L^I est une matrice d'ordre $N-2$ décrivant l'action de l'opérateur L sur les points intérieurs, L^{IF} est une matrice $(N-2)\times 2$ décrivant l'effet des points intérieurs via L sur les 2 extrémités, et ainsi de suite. L'équation différentielle ne devrait pas être imposée sur les points situés à la frontière du domaine : ce sont des conditions aux limites qui sont imposées en lieu et

place. Donc, en appliquant l'équation différentielle aux points intérieurs seulement, on trouve l'équation matricielle suivante :

$$L^I\psi^I + L^{IF}\psi^F = M^I\rho^I + M^{IF}\rho^F \tag{4.24}$$

La solution se trouve en inversant la matrice intérieure L^I, ce qui donne

$$\psi^I = -(L^I)^{-1}\Big(L^{IF}\psi^F + M^I\rho^I + M^{IF}\rho^F\Big) \tag{4.25}$$

Ceci suppose bien sûr que la matrice intérieure L^I est non singulière.

Si les conditions aux limites sont homogènes (c'est-à-dire si $\psi^F = 0$) et que l'équation différentielle aussi est homogène ($\rho = 0$), alors la seule possibilité de solution survient lorsque la matrice L^I est singulière. Cela correspond au problème des modes propres (voir ci-dessous).

Dans le cas de conditions aux limites périodiques, il n'existe pas de frontière ($F =$) et donc L et L^I sont identiques.

4.1.3 Calcul du laplacien en dimension 1

Calculons maintenant les éléments de matrice D^2_{ij} de l'opérateur de Laplace en dimension 1, c'est-à-dire la dérivée seconde. En appliquant la définition de l'élément de matrice, on trouve

$$D^2_{ij} = \langle u_i|\partial_x^2|u_j\rangle = \int_0^\ell \mathrm{d}x\, u_i(x)\frac{\mathrm{d}^2}{\mathrm{d}x^2}u_j(x) \tag{4.26}$$

Comme les fonction u_i sont linéaires, on pourrait naïvement conclure que leur dérivée seconde est nulle et donc que $D^2_{ij} = 0$. Cependant, il n'en est rien, car la dérivée seconde n'est pas définie partout sur la fonction tente : elle est nulle presque partout, et infinie sur les bords et au sommet de la tente. Il faut donc procéder par limite : supposer que la tente est très légèrement arrondie à ces points, de manière à ce que la dérivée seconde soit partout bien définie. On procède ensuite à une intégration par parties :

$$D^2_{ij} = \Big[u_i(x)u_j'(x)\Big]_0^\ell - \int_0^\ell \mathrm{d}x\, u_i'(x)u_j'(x) \tag{4.27}$$

On peut maintenant procéder à la limite, car cette expression est bien définie : les dérivées premières des fonctions tentes sont des constantes à l'intérieur du domaine de chaque fonction. Il est manifeste que D^2_{ij} s'annule si $|i-j| > 1$ car les fonctions tentes ne se recouvrent pas dans ce cas. Ignorons pour le moment le problème des extrémités ($i = 0$ et $i = N-1$). Commençons par évaluer l'élément diagonal D^2_{ii} : le premier terme est manifestement nul car la fonction u_i est nulle aux extrémités ; le deuxième terme donne, quant à lui,

$$D^2_{ii} = -\frac{1}{x_{i+1}-x_i} - \frac{1}{x_i - x_{i-1}} \tag{4.28}$$

Supposons ensuite que $j = i+1$. On trouve immédiatement que

$$D^2_{i,i+1} = \frac{1}{x_{i+1}-x_i} \tag{4.29}$$

Comme l'opérateur de Laplace est hermitique, c'est-à-dire symétrique, il s'ensuit que $D^2_{i+1,i} = D^2_{i,i+1}$.

4.1.4 Exemple : équation de Helmholtz

Considérons une corde vibrante de longueur ℓ, fixée à ses extrémités. Le déplacement transversal de la corde, noté ψ, obéit à l'équation d'onde :

$$\frac{\partial^2\psi}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2} = 0 \tag{4.30}$$

Supposons que la corde vibre à une fréquence ω. L'équation se réduit alors à l'équation de Helmholtz

$$\frac{\partial^2\psi}{\partial x^2} + k^2\psi^2 = 0 \qquad\qquad k \stackrel{\text{def}}{=} \frac{\omega}{c} \tag{4.31}$$

avec les conditions aux limites $\psi(0) = \psi(\ell) = 0$.

L'opérateur différentiel pertinent ici est donc $\mathscr{L} = \partial_x^2 + k^2$. Les éléments de matrice de ∂_x^2 ont été calculés ci-haut. Les éléments de matrice de la constante k^2 sont donnés par la matrice de masse $M_{ij} = \langle u_i | u_j \rangle$. Donc l'équation de Helmholtz sous forme matricielle se réduit à

$$(D^2 + k^2 M)\psi = 0 \tag{4.32}$$

Appliquons maintenant la condition aux limites $\psi_0 = \psi_{N-1} = 0$ du problème de la corde vibrante. L'équation (4.25) se réduit alors à

$$D^{2I}\psi = -k^2 M^I \psi \quad \text{ou encore} \quad (M^I)^{-1} D^{2I}\psi = -k^2\psi \tag{4.33}$$

L'équation (4.33) est une équation aux valeurs propres : seules les valeurs de k qui sont des valeurs propres de $(M^I)^{-1}D^{2I}$ sont admissibles. Ces valeurs propres déterminent alors les fréquences propres du problème, et les vecteurs propres correspondants sont les modes propres d'oscillation de la corde vibrante.

En pratique, il n'est pas courant d'inverser la matrice M^I, surtout en dimension supérieure à un. On tente alors de résoudre directement l'équation aux valeurs propres généralisée : $Ax = \lambda Bx$, où B est une matrice définie positive. Ce problème est plus difficile à résoudre que le problème aux valeurs propres ordinaire, et la recherche de méthodes efficaces pour ce faire, quand les matrices A et B sont énormes, est encore un champ actif de recherche.

Exercice 4.2: cas d'une grille uniforme

Considérons l'équation de Helmholtz en dimension 1 :

$$\frac{\partial^2\psi}{\partial x^2} + k^2\psi^2 = 0 \qquad\qquad \psi(0) = \psi(\ell) = 0 \tag{4.34}$$

A Donnez une expression exacte (analytique) des valeurs propres k^2 et des vecteurs propres correspondants dans ce problème. Portez une attention particulière aux valeurs propres dégénérées.

B Calculez les matrice M^I et D^{2I} dans le cas d'une grille uniforme, pour $\ell = 1$ et $N = 20$ (aidez-vous de Mathematica ou de l'équivalent ; ne codez pas en C++).

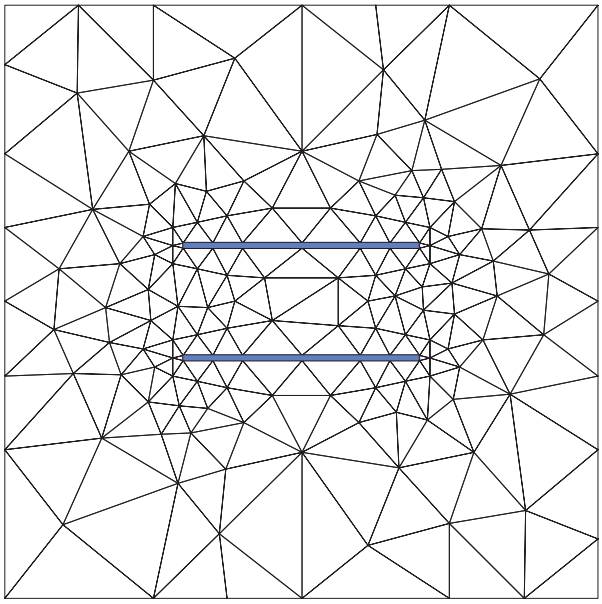


FIGURE 4.2 Triangulation de l'espace autour de deux plaques parallèles (en bleu-gris) d'un condensateur. Le problème est bi-dimensionnel, mais représente un condensateur 3D beaucoup plus long que large, coupé en son milieu.

C Comment l'application de la matrice D^{2I} sur le vecteur ψ se compare-t-elle à la formule (3.4) ? Est-ce différent ?

D Vérifiez numériquement que, dans ce cas d'une grille uniforme, les matrice D^{2I} et M^I commutent entre elles. Quelle relation s'ensuit-il entre les vecteurs propres de D^{2I} et ceux de $(M^I)^{-1}D^{2I}$?

E Comparez les 4 valeurs propres les plus petites (en valeur absolue) de la solution exacte, de D^{2I} et de $(M^I)^{-1}D^{2I}$. Dressez un tableau de ces valeurs propres et de l'écart relatif avec la valeur exacte.

F Portez les 4 premiers vecteurs propres en graphique, en les comparant aux fonctions propres exactes du problème (superposez un graphique continu et un graphique discret).

4.2 Éléments finis : dimension 2

La théorie générale des éléments finis en dimension supérieure à un n'est pas différente de ce qu'elle est en dimension 1. En particulier, la discussion de la section 4.1.2 se transpose sans modification en dimension supérieure, sauf pour le nombre de points à la frontière, qui est bien sûr plus grand que 2. La difficulté principale associée à une dimension supérieure vient de la forme plus complexe des fonctions tentes. La grille unidimensionnelle est typiquement remplacée, en dimension 2, par une *triangulation* (voir par exemple la figure 4.2).

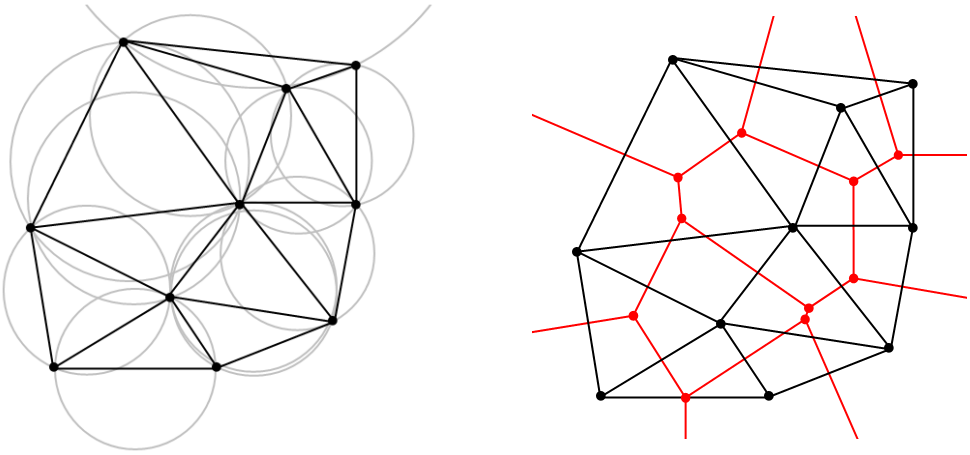


FIGURE 4.3 À gauche : triangulation de Delaunay. Le cercle circonscrit à chaque triangle ne contient aucun autre noeud. À gauche : diagramme de Voronoï (en rouge)(source: wikipedia)

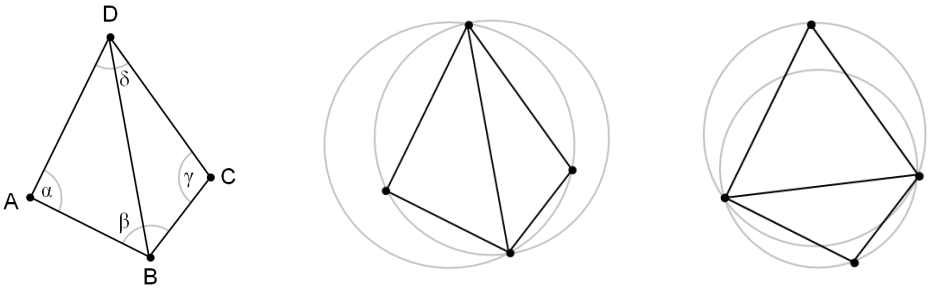


FIGURE 4.4 Illustration du basculement. Les deux triangles ABD et CBD ne respectent pas la condition de Delaunay. Mais en remplaçant le segment BD par le segment AC, on obtient deux nouveaux triangles (ABC et ADC) qui, eux, respectent la condition.(source: wikipedia)

4.2.1 Triangulations

La construction de triangulations est en soi un sujet vaste, qui s'insère dans une champ d'étude appelé *géométrie algorithmique* (angl. *computational geometry*). Partons d'un ensemble de points (les *noeuds*) qui forment la grille de points physique d'intérêt en dimension 2. Le problème est de construire un ensemble de triangles (la *triangulation*) à partir de ces points. Chaque triangle constitue alors une facette de l'espace et aucun noeud ne doit se trouver à l'intérieur d'un triangle. La solution à ce problème n'est pas unique.

Par contre, si on demande que le cercle circonscrit à chaque triangle ne contienne aucun autre noeud (les noeuds sur la circonférence étant permis), alors la solution devient unique. Les triangulations qui respectent cette condition sont appelées *triangulations de Delaunay*.[1] La

1. En fait, Делоне en russe, d'après le nom du mathématicien russe auteur du concept. Curieusement, l'article original étant publié en français, la translitération française est celle qui a été retenue, même en anglais.

figure 4.3 illustre une triangulation de Delaunay avec les cercles circonscrits à chaque triangle. La partie droite de la figure illustre le *diagramme de Voronoï* correspondant. L'intérieur de chaque polygone du diagramme de Voronoï est l'ensemble des points qui sont plus proches de chaque noeud que de tout autre point.

L'avantage des triangulations de Delaunay est que les triangles sont les plus compacts possibles et qu'un algorithme existe pour les construire. Cet algorithme est basé sur le *basculement* (voir figure 4.4) : les deux triangles ABD et CBD ne respectent pas la condition de Delaunay, comme on peut le voir à l'image du centre. On montre que la condition est violée à chaque fois que la somme des angles opposés (ici α et γ) est supérieure à π, ce qui est le cas ici. Par contre, la somme $\beta + \delta$ est alors forcément inférieure à π, et il suffit donc de remplacer le segment BD par le segment AC pour obtenir deux nouveaux triangles qui respectent la condition de Delaunay. De cette manière, on peut progressivement arriver à une triangulation entièrement conforme à la condition de Delaunay. Chaque paire de triangles opposés (c'est-à-dire partageant une arête) détermine au total 6 angles intérieurs. L'angle le plus petit, ou angle minimum, est plus grand si les deux triangles respectent la condition de Delaunay, que dans le cas contraire. En ce sens, les triangulations de Delaunay évitent plus que toutes les autres les angles petits, c'est-à-dire les triangles effilés. Notons cependant que la triangulation n'est définie qu'une fois l'ensemble des noeuds spécifié, et que le choix des noeuds peut entraîner l'existence de triangles effilés, même dans une triangulation de Delaunay.

Une triangulation de Delaunay est dite *contrainte* si elle se base non seulement sur un ensemble de noeuds, mais aussi sur un ensemble de segments, par exemple délimitant une région. Ces segments forment la frontière de la région physique d'intérêt, comme par exemple le bord de la région illustrée sur la figure 4.2, ainsi que le périmètre de chacune des deux plaques. La triangulation contrainte doit contenir les segments d'origine parmi les arêtes des triangles, en plus des noeuds.

En pratique, dans la solution d'un problème aux limites, on cherchera à raffiner la triangulation, c'est-à-dire ajouter des noeuds, jusqu'à ce qu'une certaine précision soit atteinte. Le raffinement des triangulations de Delaunay est un sujet de recherche actif. Nous utiliserons dans les travaux pratiques un programme utilisant l'algorithme de Ruppert. [2] Le raffinement procède par ajout de noeuds :

1. Chaque triangle est caractérisé par un cercle circonscrit. Si le rayon de ce cercle dépasse une certain maximum prescrit, un noeud supplémentaire est ajouté en son centre.
2. Des noeuds sont ajoutés sur le périmètre de la région au point milieu de segments, de manière à conserver un angle minimum prescrit dans tous les triangles. Ces noeuds additionnels sont appelés *points de Steiner*. L'ajout de ces noeuds permet d'augmenter la qualité locale des triangles, c'est-à-dire d'augmenter l'angle minimum.

Il y a naturellement un compromis à atteindre entre la qualité des triangles et le nombre de points : on doit augmenter le nombre de noeuds afin d'améliorer la qualité minimale des triangles.

2. Voir http://en.wikipedia.org/wiki/Ruppert's_algorithm

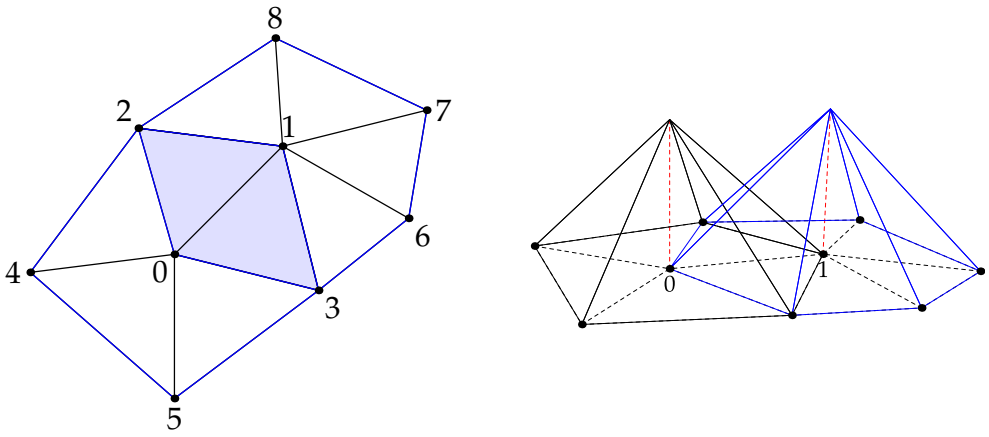


FIGURE 4.5 Fonctions tentes associées à une triangulation. À gauche : les triangles associés à deux fonctions tentes qui se recouvrent, centrées aux sites 0 et 1 respectivement. La zone de recouvrement est indiquée en bleu. À droite : vue 3D des deux fonctions tentes, pour les mêmes triangles.

Dimension 3

En trois dimensions, les triangles sont remplacés par des simplexes, c'est-à-dire des solides à 4 faces triangulaires. La notion de cercle circonscrit est remplacée par celle de sphère circonscrite, etc. On parle encore de triangulation cependant, par abus de langage.

4.2.2 Fonctions tentes

En deux dimensions, les fonctions tentes ressemblent plus à de véritables tentes, comme illustré à la figure 4.5. Elle conservent les caractéristiques suivantes :

1. Leur valeur est 1 sur le noeud correspondant.
2. Elles n'ont de recouvrement qu'avec les fonctions des noeuds voisins, c'est-à-dire les noeuds reliés par un seul segment de la triangulation.

Nous allons utiliser la notation suivante dans ce qui suit :

▷ Les noeuds seront indexés un indice latin : leurs positions seront notées $\mathbf{r}_i$, $\mathbf{r}_j$, etc.
▷ Les triangles (ou faces) seront notés T_a ($a = 0, 1, \ldots, N_T - 1$), N_T étant le nombre de faces.
▷ La surface de la face T_a sera notée A_a.
▷ Les trois noeuds de chaque face seront notés $\mathbf{r}_{a1}$, $\mathbf{r}_{a2}$ et $\mathbf{r}_{a3}$; chacun de ces noeuds étant bien sûr partagé par plusieurs faces.
▷ Inversement, le triangle formé par les trois noeuds i, j et k (dans le sens antihoraire) sera noté T_{ijk}.

L'expression analytique de chaque fonction tente dépend bien sûr de la face considrée. Considérons à cet effet trois noeuds $\mathbf{r}_{1,2,3}$ délimitant une face, et trouvons l'expression de la fonction tente centrée à $\mathbf{r}_1$ sur cette face. Définissons d'abord la fonction

$$\begin{aligned} \gamma(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3) &= \mathbf{z} \cdot [(\mathbf{r}_2 - \mathbf{r}_1) \wedge (\mathbf{r}_3 - \mathbf{r}_1)] \\ &= x_2 y_3 - x_3 y_2 - x_1 y_3 + x_3 y_1 + x_1 y_2 - x_2 y_1 \end{aligned} \tag{4.35}$$

D'après les propriétés du produit vectoriel, cette fonction est 2 fois l'aire orientée de la face formée des trois points en question, si ces points sont pris dans le sens anti-horaire. Elle possède les propriétés suivantes :

1. Elle est antisymétrique lors de l'échange de deux arguments et inchangée lors d'une permutation cyclique de ses trois arguments.
2. Elle est linéaire dans chacun de ses arguments

Exercice 4.3

Démontrez ces propriétés.

La fonction tente centrée à $\mathbf{r}_1$ est linéaire en (x, y), s'annule à $\mathbf{r} = \mathbf{r}_2$ et $\mathbf{r} = \mathbf{r}_3$, et est égale à 1 si $\mathbf{r} = \mathbf{r}_1$. la seule possibilité est

$$u_{1,2,3}(\mathbf{r}) = \frac{\gamma(\mathbf{r}, \mathbf{r}_2, \mathbf{r}_3)}{\gamma(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)} \tag{4.36}$$

Cette expression répond manifestement aux critères, et de plus est unique, car une seule fonction linéaire (un seul plan) passe par les trois points $(x_1, y_1, 1)$, $(x_2, y_2, 0)$ et $(x_3, y_3, 0)$. Nous utiliserons la notation abrégée

$$\gamma_{ijk} = \gamma(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) \tag{4.37}$$

où il est sous-entendu que les trois points forment une face de la triangulation.

La fonction tente complète centrée en $\mathbf{r}_0$ sur la figure 4.5 sera donc

$$u_0(\mathbf{r}) = \begin{cases} u_{012}(\mathbf{r}) & \text{si} \quad \mathbf{r} \in T_{012} \\ u_{024}(\mathbf{r}) & \text{si} \quad \mathbf{r} \in T_{024} \\ \dots & \dots \\ u_{031}(\mathbf{r}) & \text{si} \quad \mathbf{r} \in T_{031} \end{cases} \tag{4.38}$$

Les fonctions tente de deux noeuds voisins partagent deux faces situées de part et d'autre du lien qui relie les deux noeuds, comme illustré à la figure 4.5.

Exercice 4.4: matrice de masse

L'objectif de ce problème est de calculer la matrice de masse $M_{ij} = \langle u_i | u_j \rangle$ des fonctions tentes en dimension 2.

A Montrez que l'élément de matrice diagonal est

$$\langle u_i | u_i \rangle = \frac{1}{6} \sum_{a, i \in T_a} A_a \tag{4.39}$$

où la somme est effectuée sur les faces a qui touchent le site i.

B Montrez que l'élément de matrice entre sites voisins est

$$\langle u_i | u_j \rangle = \frac{1}{12}(A_a + A_b) \tag{4.40}$$

où a et b indexent les triangles situés de part et d'autre du lien ij.

Indice : procéder à un changement de variable pour effectuer les intégrales. Par exemple, pour trois noeuds 1, 2 et 3, il faut paramétrer les points à l'intérieur du triangle par les variables s et t telles que

$$\mathbf{r} = \mathbf{r}_1 + s(\mathbf{r}_2 - \mathbf{r}_1) + t(\mathbf{r}_3 - \mathbf{r}_1) \tag{4.41}$$

et intégrer dans le domaine approprié du plan (s,t), après avoir calculé le jacobien associé à ce changement de variables. Les propriétés de $\gamma(\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)$ (linéarité, antisymétrie) contribuent à simplifier considérablement le calcul.

4.2.3 Évaluation du laplacien

Supposons que nous ayons à notre disposition une triangulation de l'ensemble des noeuds $\mathbf{r}_i$ en deux dimensions. L'opérateur de Laplace ∇^2 intervient dans un grand nombre d'équations différentielles et il est donc nécessaire dans ces cas d'en calculer une représentation dans la base des fonctions tente. Le problème est donc de calculer les éléments de matrice

$$D^2_{ij} = \langle u_i|\nabla^2|u_j\rangle = \int \mathrm{d}^2r\, u_i(\mathbf{r})\nabla^2 u_j(\mathbf{r}) \tag{4.42}$$

L'évaluation se fait encore une fois par intégration par parties, en utilisant l'identité de Green :

$$\int_R \mathrm{d}^2r\, f\nabla^2 g = \oint_{\partial R} \mathrm{d}\mathbf{a}\cdot\nabla g f - \int_R \mathrm{d}^2r\,\nabla f\cdot\nabla g \tag{4.43}$$

où la première intégrale est menée sur le périmètre ∂R de la région R considérée.

Les fonctions $u_i(\mathbf{r})$ étant linéaires, leur gradient est constant et les intégrales ne présentent donc pas de difficulté, sinon dans l'organisation des différentes faces impliquées. De plus, les fonctions s'annulant sur le périmètre de leur domaine, le terme de périmètre ne contribue jamais.

Chaque fonction u_i est une combinaison des fonctions u_{ijk} définies en (4.36). On calcule sans peine que

$$\nabla u_{ijk}(\mathbf{r}) = \frac{1}{\gamma_{ijk}}\mathbf{z}\wedge(\mathbf{r}_k - \mathbf{r}_j) \tag{4.44}$$

et ensuite que

$$\int_{T(ijk)} \mathrm{d}^2r\,(\nabla u_{ijk})^2 = \frac{\ell^2_{jk}}{4A_{ijk}} \tag{4.45}$$

où ℓ_{jk} est la longueur du segment reliant j à k. D'autre part,

$$\int_{T(ijk)} \mathrm{d}^2r\,\nabla u_{ijk}\cdot\nabla u_{jik} = -\frac{\mathbf{r}_{ik}\cdot\mathbf{r}_{jk}}{4A_{ijk}} \tag{4.46}$$

où $\mathbf{r}_{ij} \overset{\text{def}}{=} \mathbf{r}_i - \mathbf{r}_j$. Ces relations permettent de calculer les éléments de matrice du laplacien simplement en sommant les contributions des différentes faces :

$$\begin{aligned} D^2_{ii} &= -\frac{1}{4}\sum_{a,i\in T_a}\frac{\ell^2_{ia}}{A_a} \\ D^2_{ij} &= -\frac{\ell^2_{ij}-\ell^2_{ik}-\ell^2_{jk}}{8A_{ijk}} - \frac{\ell^2_{ij}-\ell^2_{ik'}-\ell^2_{jk'}}{8A_{jik'}} \end{aligned} \tag{4.47}$$

La notation utilisée est la suivante : ℓ_{ij} est la longueur du lien reliant les noeuds i et j ; ℓ_{ia}, a étant un indice de face, est la longueur du lien de la face a opposé au site i. Dans la deuxième équation, k et k' sont les noeuds qui complètent les faces (ijk) et (jik') situées de part et d'autres du lien (ij). Ainsi, en se référant à la figure 4.5, on a par exemple

$$\begin{aligned} D^2_{00} &= -\frac{1}{4}\left(\frac{\ell^2_{12}}{A_{012}} + \frac{\ell^2_{24}}{A_{024}} + \frac{\ell^2_{45}}{A_{045}} + \frac{\ell^2_{53}}{A_{053}} + \frac{\ell^2_{31}}{A_{031}}\right) \\ D^2_{01} &= -\frac{\ell^2_{01} - \ell^2_{02} - \ell^2_{12}}{8A_{012}} - \frac{\ell^2_{01} - \ell^2_{03} - \ell^2_{13}}{8A_{103}} \end{aligned} \tag{4.48}$$

Exercice 4.5

Démontrez les relations (4.47).

Exercice 4.6: Modes de l'équations de Helmholtz dans un enclos carré

Considérons l'équation de Helmholtz

$$\nabla^2\psi + k^2\psi = 0 \tag{4.49}$$

à l'intérieur d'un enclos carré de côté ℓ. On suppose que la fonction ψ s'annule sur le périmètre.

A Donnez une expression exacte (analytique) des valeurs propres k^2 et des vecteurs propres correspondants dans ce problème. Portez une attention particulière aux valeurs propres dégénérées.

B Utilisez le programme `triangle` de J.R. Shewchuk (disponible sur la page Moodle du cours) pour effectuer une triangulation à l'intérieur de ce carré. Le fichier d'entrée nécessaire (`carre.poly`) est fourni en annexe ; utilisez la commande en ligne
`triangle -q -a0.01 carre.poly`
pour générer les fichiers `carre.1.node` et `carre.1.ele` nécessaires au calcul.

C Installez le programme `Helmholtz2D.cpp` fourni en annexe et appliquez-le aux fichiers produits à l'étape précédente. Le code permet de résoudre l'équation aux valeurs propres soit à l'aide d'une méthode de Lanczos ou d'une méthode appliquée aux matrices denses. Les valeurs et vecteurs propres sont écrits dans le fichier de sortie `vp.dat`. Comment les valeurs propres se comparent-elles aux valeurs exactes ? Appliquez et comparez les deux approches de calculs des modes propres.

D Portez en graphique les fonctions propres associées aux 4 valeurs propres les plus petites (en valeur absolue). Produisez des graphiques densité à l'aide de gnuplot (un script gnuplot est fourni en annexe).

Exercice 4.7: Condensateur à plaques parallèles finies

Considérons un condensateur à plaques parallèles (ou plutôt une coupe en deux dimensions d'un condensateur dont les plaques ont une extension très grande selon l'axe des z). Ce système est décrit à la figure 4.2. On supposera que la tension est $\psi = 1$ sur une plaque et $\psi = -1$ sur l'autre plaque, alors qu'elle s'annule au loin (c'est-à-dire sur la frontière du système). Le potentiel électrique autour des plaques obéit à l'équation de Laplace : $\nabla^2\psi = 0$.

A Utilisez le programme `triangle` pour effectuer une triangulation dans cet espace. Le fichier d'entrée nécessaire (`cond.poly`) est fourni en annexe ; utilisez la commande en ligne
`triangle -q32 -a0.1 cond.poly`
pour générer les fichiers `cond.1.node` et `cond.1.ele` nécessaires au calcul.

B Installez le programme `Laplace2D.cpp` fourni en annexe et appliquez-le aux fichiers produits à l'étape précédente. Le code permet de résoudre le problème aux limites avec les conditions aux limites définies dans le fichier d'entrée `para.dat`.

C Portez la solution en graphique à l'aide de gnuplot. Faites un graphique densité et un graphique 3D (un script gnuplot est fourni en annexe).

4.3 Méthodes spectrales

La méthode des éléments finis est caractérisée par l'utilisation de fonctions de base locales (les fonctions tentes des sections précédentes). Ces fonctions sont bien adaptées aux géométries compliquées, en raison de la possibilité de faire des triangulations, mais ne constituent pas la meilleure solution dans le cas des géométries simples. Dans cette section, nous allons expliquer comment utiliser une base de polynômes afin de représenter les fonctions plus efficacement (c'est-à-dire avec moins de fonctions de base).

4.3.1 Bases de polynômes orthogonaux et fonctions cardinales

Les méthodes spectrales utilisent les polynômes orthogonaux comme fonctions de base, ou encore des fonctions trigonométriques dans le cas de conditions aux limites périodiques. La discussion qui suit supposera qu'un interval ouvert (i.e. non périodique) est considéré. Le cas périodique sera traité à la section 4.3.4 ci-dessous.

Fixons un entier N. Les N racines x_i de $p_N(x)$ sont utilisées comme points de grille. Définissons aussi les polynômes orthonormés

$$\phi_k(x) = \frac{1}{\sqrt{\gamma_k}} p_k(x) \qquad \langle \phi_k | \phi_m \rangle = \delta_{km} \qquad k, m = 0, \ldots, N-1 \tag{4.50}$$

Une fonction $\psi(x)$ admet alors le développement limité suivant :

$$\psi(x) \approx \sum_{k=0}^{N-1} \bar{\psi}_k \phi_k(x) \qquad \bar{\psi}_k = \langle \phi_k | \psi \rangle = \sum_{i=1}^{N} w_i \psi_i \phi_k(x_i) \qquad \psi_i \stackrel{\text{def}}{=} \psi(x_i) \tag{4.51}$$

Les polynômes orthogonaux sont des fonctions oscillantes étendues sur tout l'intervalle $[a, b]$. En ce sens ils sont l'analogue des ondes planes (fonctions trigonométriques). Il est généralement plus utile d'avoir recours à des fonctions localisées, comme dans la méthode des éléments

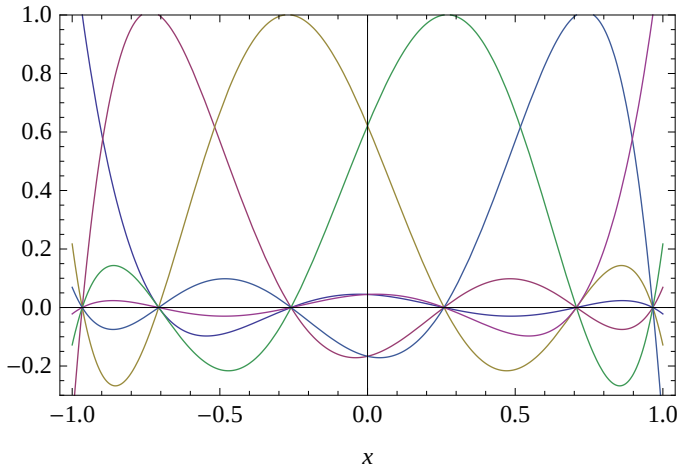


FIGURE 4.6 Les fonctions cardinales associées aux 6 racines du polynôme de Tchébychev $T_6(x)$.

finis, mais qui nous permettent de profiter des excellentes propriétés de convergence des séries définies sur des bases de polynômes orthogonaux. C'est pour cela qu'on définit les *fonctions cardinales* :

$$C_i(x) \stackrel{\text{def}}{=} \prod_{\substack{j=1 \\ j\neq i}}^{N} \frac{x - x_j}{x_i - x_j} \tag{4.52}$$

La fonction $C_i(x)$ est un polynôme de degré $N-1$ qui s'annule à tous les points x_j de la grille, sauf au point x_i où elle est égale à l'unité : $C_j(x_i) = \delta_{ij}$.

Exercice 4.8

Montrez que

$$C_i(x) = \frac{p_N(x)}{(x - x_i)p'_N(x_i)} \tag{4.53}$$

où p'_N est la dérivée du polynôme p_N. Indice : les racines du polynômes C_i sont presque les mêmes que celles de p_N.

Nous avons donc à notre disposion deux bases de fonctions pour l'intervalle $[a,b]$: les polynômes orthogonaux normalisés $\phi_j(x)$ (ou $|\phi_j\rangle$) et les fonctions cardinales $C_j(x)$ (ou $|C_j\rangle$). Ces deux bases sont toutes les deux polynomiales de même degré, et sont reliées par une transformation de similitude, qu'on trouve facilement en exprimant les fonctions cardinales sur la base des polynômes orthogonaux :

$$|C_j\rangle = \sum_i M_{ij}|\phi_i\rangle \quad \text{où} \quad M_{ij} \stackrel{\text{def}}{=} \langle\phi_i|C_j\rangle = \sum_k w_k\phi_i(x_k)C_j(x_k) = w_j\phi_i(x_j) \tag{4.54}$$

car $C_j(x_k) = \delta_{jk}$.

Le développement d'une fonction $\psi(x)$ sur la base des fonctions cardinales est très simple :

$$\psi(x) \approx \sum_{i=0}^{N-1} \psi(x_i)C_i(x) = \sum_{i=0}^{N-1} \psi_i C_i(x) \quad \text{ou encore} \quad |\psi\rangle \approx \sum_{i=0}^{N-1} \psi_i|C_i\rangle \tag{4.55}$$

On démontre ce développement en substituant $x = x_j$ ($j = 0, 1, \cdots, N-1$). Le signe $\approx$ devient une égalité si la fonction ψ est un polynôme de degré $N-1$ ou moins.

La relation existant entre les coefficients $\bar{\psi}_i$ et ψ_i est

$$\bar{\psi}_i = \langle \phi_i | \psi \rangle = \sum_j \langle \phi_i | C_j \rangle \psi_j = \sum_j M_{ij} \psi_j \tag{4.56}$$

ou, en notation matricielle : $\bar{\psi} = M\psi$. On vérifie d'ailleurs, d'après l'expression ci-dessus de M_{ij} et de $\bar{\psi}_i$, que

$$\sum_j M_{kj} \psi_j = \sum_j w_j \phi_k(x_j) \psi_j = \bar{\psi}_k \tag{4.57}$$

comme démontré plus haut.

Notons que la base des fonctions cardinales est orthogonale, mais pas orthonormée :

$$\langle C_i | C_j \rangle = \sum_k w_k C_i(x_k) C_j(x_k) = \sum_k w_k \delta_{ik} \delta_{jk} = w_i \delta_{ij} \tag{4.58}$$

Opérateur différentiel

Considérons maintenant un opérateur différentiel linéaire $\mathscr{L}$, par exemple le laplacien. Sur la base des fonctions cardinales, cet opérateur a une forme matricielle L telle que

$$\mathscr{L} | C_i \rangle = \sum_j L_{ji} | C_j \rangle \tag{4.59}$$

(notez l'ordre des indices de la matrice L). L'équation différentielle $\mathscr{L}\psi(x) = f(x)$, où $f(x)$ est le vecteur de charge (connu) et $\psi(x)$ la fonction recherchée, s'écrit abstraitement de la manière suivante : $\mathscr{L}|\psi\rangle = |f\rangle$. L'utilisation des fonctions cardinales est étroitement liée à la méthode de collocation, selon laquelle l'équation différentielle est imposée aux points de grille, à savoir

$$\langle x_i | \mathscr{L} | \psi \rangle = \langle x_i | f \rangle = f(x_i) = f_i \tag{4.60}$$

D'après la définition de L_{ij} ci-haut, le membre de gauche de cette équation devient

$$\begin{aligned} \langle x_i | \mathscr{L} | \psi \rangle &= \sum_j \psi_j \langle x_i | \mathscr{L} | C_j \rangle \\ &= \sum_{j,k} \psi_j L_{kj} \langle x_i | C_k \rangle \\ &= \sum_{j,k} \psi_j L_{kj} \delta_{ik} \\ &= \sum_j L_{ij} \psi_j \end{aligned} \tag{4.61}$$

ce qui s'exprime comme $L\psi$ en notation matricielle, ψ étant le vecteur-colonne des coefficients ψ_i. L'équation différentielle devient donc l'équation matricielle

$$L\psi = f \tag{4.62}$$

Les matrices associées aux dérivées d'ordre quelconque $\mathscr{L} = \partial_x^{(n)}$ peuvent être calculées relativement facilement à l'aide de l'expression explicite (4.52) des fonctions cardinales. L'opérateur de différentiation ∂_x a la représentation générale

$$\partial_x C_i(x) = \sum_k D_{ki}^{(1)} C_k(x) \tag{4.63}$$

et la matrice $D_{ji}^{(1)}$ n'est rien d'autre que la valeur de la dérivée de $C_i(x)$ évaluée à $x = x_j$, comme on peut le voir en posant $x = x_j$ dans l'équation ci-dessus :

$$C_i'(x_j) = \sum_k D_{ki}^{(1)} C_k(x_j) = \sum_k D_{ki}^{(1)} \delta_{kj} = D_{ji}^{(1)} \tag{4.64}$$

Les matrices associées aux dérivées d'ordre supérieur peuvent être calculées simplement en prenant les puissances appropriées de la matrice $D^{(1)}$. Une routine à cet effet fait partie des codes annexes à ce cours : c'est la méthode `poids()` du fichier `collocation.h`.

4.3.2 Quadratures de Lobatto

La quadrature gaussienne se base sur des abscisses qui sont strictement contenues à l'intérieur du domaine $[a, b]$. Cela pose un problème si on veut représenter des fonctions qui ont des valeurs précises aux frontières de l'intervalle. On peut remédier à ce problème en définissant une approche légèrement différente basée sur la formule de quadrature de Lobatto qui, elle, inclut les points extrêmes. Cette formule existe en général pour une fonction poids $w(x)$ quelconque et s'obtient en imposant que les extrémités de l'intervalle (a et b) fassent partie de la grille d'intégration, qui comporte alors $N-2$ points intérieurs et 2 points sur la frontière. Les poids w_i et les noeuds intérieurs forment un ensemble de $2N-2$ paramètres ajustables qui sont déterminés afin de rendre l'intégrale exacte pour des polynômes du plus haut degré possible, c'est-à-dire $2N-3$.

La formule de Lobatto pour les polynômes de Legendre est

$$\int_{-1}^{1} \mathrm{d}x\, f(x) \approx \sum_{i=1}^{N} w_i f(x_i) \tag{4.65}$$

où

$$\begin{aligned} &x_1 = -1 \qquad x_N = 1 \qquad w_1 = w_N = \frac{2}{N(N-1)} \\ &w_i = \frac{2}{N(N-1)[P_{N-1}(x_i)]^2} \qquad P_{N-1}'(x_i) = 0 \qquad i = 2, \ldots, N-1 \end{aligned} \tag{4.66}$$

Autrement dit, les points intérieurs sont les $N-2$ racines de la dérivée de P_{N-1}, qui est un polynôme d'ordre $N-2$.

La formule de Lobatto pour les polynôme de Tchébychev est

$$\int_{-1}^{1} \mathrm{d}x\, \frac{f(x)}{\sqrt{1-x^2}} \approx \sum_{i=1}^{N} w_i f(x_i) \qquad w_i = \frac{\pi}{N} \qquad x_i = \cos\frac{(i-1)\pi}{N} \tag{4.67}$$

Notons que la position des abscisses et les poids sont particulièrement simples avec ces polynômes.

Nous ne démontrerons pas les formules de Lobatto ici, comme nous l'avons fait pour la formule d'intégration gaussienne en général. Ces formules fournissent des approximations différentes au produit scalaire (3.53), qui sont d'ordre plus bas : exactes pour les polynômes de degré $2N-3$ au lieu de $2N-1$. Par contre, leur avantage est que les fonctions cardinales associées $C_i(x)$ permettent de représenter les valeurs aux frontières de l'intervalle. En ce sens, ces fonctions sont une alternative aux fonctions tentes utilisées dans la méthode des éléments finis, et le traitement des conditions aux limites peut se faire exactement comme dans le section 4.1.2, c'est-à-dire en définissant une représentation L^I de l'opérateur différentiel pour les points intérieurs

4.3.3 Exemple : problème aux limites en dimension 1

Revisitons le problème des valeurs propres de l'équation de Helmholtz étudié à la section 4.1.4. Il s'agit ici de calculer les valeurs propres du laplacien $\mathscr{L} = \partial_x^2$, donc de résoudre l'équation aux valeurs propres

$$\mathscr{L}|\psi\rangle = \lambda|\psi\rangle \tag{4.68}$$

En projetant sur les fonctions $\langle x_i|$, cette équation devient

$$\langle x_i|\mathscr{L}|\psi\rangle = L_{ij}\psi_j = \lambda\psi_i \quad \text{ou encore} \quad L\psi = \lambda\psi \tag{4.69}$$

en notation matricielle. Il s'agit donc d'une équation aux valeurs propres en fonction de la matrice L.

Il faut cependant tenir compte des conditions aux limites, c'est-à-dire imposer l'annulation de la fonction aux extrémités de l'intervalle. Dans l'intervalle $[-1,1]$ associé aux polynômes de Tchébychev, cela revient à demander $\psi(-1) = \psi(1) = 0$. Pour appliquer cette contrainte, nous avons besoin de points de grille aux extrémités, donc de la grille de Gauss-Lobatto. L'équation aux valeurs propres elle-même n'est valable que pour les points intérieurs de la grille.

Nous devons donc partitionner la matrice L comme expliqué à la fin de la section 4.1.2. L'équation aux valeurs propres prend alors la forme

$$\begin{pmatrix} L^F & L^{FI} \\ L^{IF} & L^I \end{pmatrix} \begin{pmatrix} 0 \\ \psi^I \end{pmatrix} = \lambda \begin{pmatrix} 0 \\ \psi^I \end{pmatrix} \tag{4.70}$$

ce qui revient à demander

$$L^I\psi^I = \lambda\psi^I \tag{4.71}$$

Autrement dit, nous devons chercher les valeurs propres de la matrice intérieure seulement.

Le tableau 4.1 illustre les résultats obtenus de cette manière et les compare aux résultats exacts et à ceux obtenus par la méthode des éléments finis. Le fait remarquable est que la méthode spectrale, avec seulement 12 points (donc 10 points intérieurs) est beaucoup plus précise que la méthode des éléments finis avec 101 points, du moins pour les valeurs propres

TABLE 4.1 Les 5 premières valeurs propres de l'opérateur ∂_x^2 dans l'intervalle $[-1,1]$. À droite : valeurs exactes $\lambda_n = (n\pi/2)^2$. Au milieu, valeurs obtenues à l'aide de la méthode des éléments finis et $N = 101$ points de grille. À gauche, valeurs obtenues à l'aide d'une méthode spectrale et d'une grille de Gauss-Lobatto de $N = 12$ points.

n	exact	éléments finis	méthode spectrale
1	2.4674	2.4676	2.4674
2	9.8696	9.8728	9.8696
3	22.206	22.223	22.206
4	39.478	39.530	39.472
5	61.685	61.812	61.900

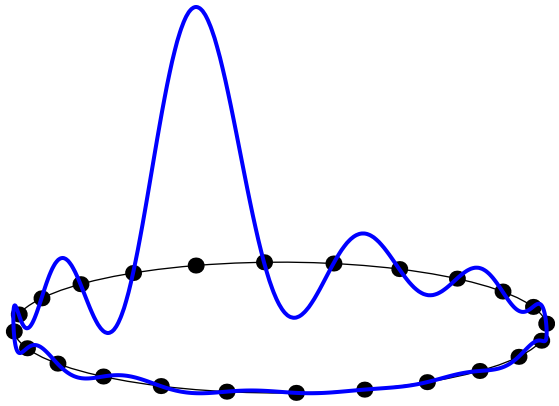


FIGURE 4.7 Fonction cardinale sur une grille périodique de 24 points. Les 24 fonctions sont identiques à celle illustrée ici, sauf pour une translation. L'espace périodique est ici représenté comme un cercle.

les plus basses. En fait, les valeurs obtenues avec la méthode spectrale cessent d'être fiable après $\sim n/2$ valeurs propres, n étant le nombre de points utilisés. Mais souvent ce sont les valeurs propres les plus basses qui sont recherchées.

4.3.4 Conditions aux limites périodiques

Lorsqu'on a affaire à des conditions aux limites périodiques, c'est-à-dire lorsque le problème est défini sur une cercle au lieu d'un segment ouvert, les fonctions spectrales les plus utiles ne sont pas les polynômes de Legendre ou de Tchébychev, mais plutôt les fonctions trigonométriques qui sont d'emblée périodiques. Dans cette section nous allons décrire les fonctions cardinales utilisées dans les problèmes périodiques.

Il est toujours possible, par un changement d'échelle approprié, de fixer la longueur de

l'intervalle périodique à 2π. Nous allons adopter une grille de N points également espacés (N étant pair)[3] :

$$x_j = \frac{2\pi j}{N} \qquad j = 0, 2, \ldots, N-1 \tag{4.72}$$

Les fonctions cardinales appropriées à cette grille sont définies comme suit :

$$C_i(x) = \frac{1}{N} \frac{\sin\left(\frac{1}{2}N(x_i - x)\right)}{\sin\left(\frac{1}{2}(x_i - x)\right)} \cos\left(\tfrac{1}{2}(x_i - x)\right) \tag{4.73}$$

Ces fonctions ne sont pas des polynômes en x (les polynômes ne sont pas périodiques de toute manière). Par contre, elles ont les propriétés suivantes :

1. Elles sont périodiques, de période 2π, à condition que N soit pair. Voir à cet effet la figure 4.7.
2. $C_i(x_j) = \delta_{ij}$. Cela se vérifie immédiatement si $i \neq j$. Pour $i = j$, il s'agit d'un processus de limite standard pour le rapport des sinus. Donc le développement d'une fonction périodique $\psi(x)$ sur la base des fonctions cardinales s'effectue comme auparavant :

$$\psi(x) = \sum_{i=0}^{N-1} \psi_i C_i(x) \quad \text{ou} \quad \psi_i \stackrel{\text{def}}{=} \psi(x_i) \tag{4.74}$$

3. En fonction d'exponentielles complexes, la fonction $C_0(x)$ s'exprime comme

$$\begin{aligned} C_0(x) &= \frac{1}{2N} \frac{e^{iNx/2} - e^{-iNx/2}}{e^{ix/2} - e^{-ix/2}} \left(e^{ix/2} + e^{-ix/2}\right) \\ &= \frac{1}{2N} \frac{z^{N/2} - z^{-N/2}}{z^{1/2} - z^{-1/2}} (z^{1/2} + z^{-1/2}) \qquad z \stackrel{\text{def}}{=} e^{ix} \\ &= \frac{z^{-N/2}}{2N} \frac{z^N - 1}{z - 1} (z + 1) \\ &= \frac{z^{-N/2}}{2N} (1 + z + z^2 + \cdots + z^{N-1})(z + 1) \\ &= \frac{1}{N} \left(\tfrac{1}{2} z^{-N/2} + z^{-N/2+1} + z^{-N/2+2} + \cdots + z^{N/2-1} + \tfrac{1}{2} z^{N/2}\right) \end{aligned} \tag{4.75}$$

   Autrement dit, $C_0(x)$ est un *polynôme trigonométrique* de degré $N/2$, c'est-à-dire une combinaison des puissances entières de $z = e^{ix}$, de $-N/2$ at $N/2$.

4. La fonction $C_j(x)$ s'obtient en remplaçant x par $x - x_j$, ou encore z par $w \stackrel{\text{def}}{=} \omega^j z$, où $\omega \stackrel{\text{def}}{=} e^{-2\pi i/N}$:

$$C_j(x) = \frac{1}{N} \left[\tfrac{1}{2} w^{-N/2} + w^{-N/2+1} + w^{-N/2+2} + \cdots + w^{N/2-1} + \tfrac{1}{2} w^{N/2}\right] \tag{4.76}$$

   Les N fonctions cardinales sont toutes des polynômes trigonométriques de degré $N/2$. Elles sont linéairement indépendantes. Elles sont aussi réelles, alors que w est complexe, en raison de leur invariance lors du remplacement $w \to w^{-1}$. Le nombre de degrés de liberté de l'ensemble des polynômes respectant cette condition est $N + 1$. Comme il n'y a

3. il n'y a aucune raison que les points ne soient pas également espacés, étant donnée l'invariance par translation

que N fonctions cardinales, l'un de ces polynômes ne peut pas être exprimé comme une combinaison de fonctions cardinales : il s'agit de

$$i\left(z^{-N/2}-z^{N/2}\right)=2\sin\frac{Nx}{2} \tag{4.77}$$

comme on peut le vérifier aisément en vérifiant que les coefficients de développement (4.74) sont tous nuls.

5. On montre facilement que l'action de la dérivée première sur les fonctions cardinales est la suivante :

$$C_j'(x_i)=D_{ij}^{(1)}=\begin{cases}0 & (i=j)\\ \frac{1}{2}(-1)^{i+j}\cot\left(\frac{x_i-x_j}{2}\right) & (i\neq j)\end{cases} \tag{4.78}$$

La matrice $D^{(2)}$ représentant la dérivée seconde est simplement le carré de $D^{(1)}$.

Exercice 4.9

Montrez que les N fonctions cardinales (4.76) sont linéairement indépendantes. Indice : exprimer dans la base des puissances z^n, et calculez le déterminant formé des N fonctions cardinales. C'est un déterminant de Vandermonde.

4.4 Annexe : code

4.4.1 Fonctions tentes en dimension 1

Ce code et le suivant utilisent une représentation des matrices creuses définie dans `SparseMatrix.h`.

Code 4.1 : Éléments finis en dimension 1 : `maillage1D.h`

```
#ifndef MAILLAGE1D_H_
#define MAILLAGE1D_H_

#include <iostream>
#include <fstream>
#include <string>
#include <cassert>

#include "SparseMatrix.h" classe représentant des matrices creuses
#include "Gauss_Legendre.h" quadrature gaussienne (7 points)
#define MAX_NODES 10000 nombre maximum de noeuds

class maillage1D{
public:
  Vector<double> x; coordonnées des noeuds

  allocation d'une grille uniforme de n points
  void init(int n, double L){
```

```
19      assert(n>1);
20      x.Alloc(n);
21      double h = L/(n-1);
22      for(int i=0; i<n; i++) x[i] = i*h;
23    }
24
25    lecture d'une grille à partir d'un fichier
26    void init(string &name){
27      ifstream fin(name.c_str());
28      assert(fin);
29      int n;
30      fin >> n;
31      assert(n>1 and n<MAX_NODES); précaution contre les erreurs d'entrée
32      int tmp;
33      for(int i=0; i<n; i++) fin >> tmp >> x[i];
34      fin.close();
35    }
36
37    friend std::ostream & operator<<(std::ostream &flux, const maillage1D &x){
38      flux << "Liste des noeuds :\n";
39      flux << x.x.size() << endl;
40      for(int i=0; i<x.x.size(); i++) flux << i << '\t' << x.x[i] << endl;
41      return flux;
42    }
43
44     méthode de calcul de la matrice de masse
45     Mi: partie intérieure
46     Mf: partie qui couple les noeuds de frontière aux noeuds intérieurs
47    void mass_matrix(SparseMatrix &Mi, SparseMatrix &Mf){
48      const int n = x.size();
49      const int ni = n-2;
50      const double i6 = 1.0/6;
51      const double i3 = 1.0/3;
52
53      assert(Mi.rows() == ni);
54      assert(Mf.rows() == ni and Mf.columns() == 2);
55      Mf.Insert(0,0,i6*(x[1]-x[0]));
56      Mf.Insert(n-3,1,i6*(x[n-1]-x[n-2]));
57      Mi.Insert(0,0,i3*(x[2]-x[0]));
58      for(int i=1; i<ni; i++){
59        Mi.Insert(i-1,i,i6*(x[i+1]-x[i]));
60        Mi.Insert(i,i-1,i6*(x[i+1]-x[i]));
61        Mi.Insert(i,i,i3*(x[i+2]-x[i]));
62      }
63    }
64
65     méthode de calcul du laplacien
66     Li: partie intérieure
67     Lf: partie qui couple les noeuds de frontière aux noeuds intérieurs
```

```
68      z : multiplicateur
69    void Laplacian(SparseMatrix &Li, SparseMatrix &Lf, double z = 1.0){
70      const int n = x.size();
71      const int ni = n-2;
72
73      assert(Li.rows() == ni);
74      assert(Lf.rows() == ni and Lf.columns() == 2);
75      Lf.Insert(0,0,z/(x[1]-x[0]));
76      Lf.Insert(n-3,1,z/(x[n-1]-x[n-2]));
77      Li.Insert(0,0,-z/(x[2]-x[1])-z/(x[1]-x[0]));
78      for(int i=1; i<ni; i++){
79        Li.Insert(i-1,i,z/(x[i+1]-x[i]));
80        Li.Insert(i,i-1,z/(x[i+1]-x[i]));
81        Li.Insert(i,i,-z/(x[i+2]-x[i+1])-z/(x[i+1]-x[i]));
82      }
83    }
84
85     évaluation des fonctions tente
86    double tente(int i, double z){
87      double u = 0.0;
88      int n1 = x.size()-1;
89      if(i==0){
90        if(z>x[0] and z<x[1]) u = (z-x[1])/(x[0]-x[1]);
91      }
92      else if(i==n1){
93        if(z>x[n1-1] and z<x[n1]) u = (z-x[n1-1])/(x[n1]-x[n1-1]);
94      }
95      else{
96        if(z>x[i-1] and z<x[i]) u = (z-x[i-1])/(x[i]-x[i-1]);
97        else if(z>x[i] and z<x[i+1]) u = (z-x[i+1])/(x[i]-x[i+1]);
98      }
99      return u;
100   }
101
102 };
103 #endif
104
105  calcul des éléments de matrice d'une fonction f(x)
106  f: foncteur représentant la fonction
107  Mi: partie intérieure
108  Mf: partie qui couple les noeuds de frontière aux noeuds intérieurs
109  l'intégration est réalisée par quadrature gaussienne
110 template<class T>
111 void potentiel(maillage1D &grille, T &f, SparseMatrix &Mi, SparseMatrix &Mf){
112   const int n = grille.x.size();
113   const int ni = n-2;
114   double z;
115
116   const int gln = 7;
```

```
117     Vector<double> glx(gln);
118     Vector<double> glf(gln);
119
120      éléments hors diagonale mixtes
121     Gauss_Legendre_x(glx,grille.x[0],grille.x[1]);
122     for(int j=0; j<gln; j++) glf[j] = grille.tente(0,glx[j])*grille.tente(1,glx[j])*f
            (glx[j]);
123     z = Gauss_Legendre(glf);
124     Mf.Insert(0,0,z);
125
126     Gauss_Legendre_x(glx,grille.x[ni],grille.x[ni+1]);
127     for(int j=0; j<gln; j++) glf[j] = grille.tente(ni,glx[j])*grille.tente(ni+1,glx[j
            ])*f(glx[j]);
128     z = Gauss_Legendre(glf);
129     Mf.Insert(ni-1,1,z);
130
131      éléments hors diagonale internes
132     for(int i=2; i<=ni; i++){
133       Gauss_Legendre_x(glx,grille.x[i-1],grille.x[i]);
134       for(int j=0; j<gln; j++) glf[j] = grille.tente(i-1,glx[j])*grille.tente(i,glx[j
              ])*f(glx[j]);
135       z = Gauss_Legendre(glf);
136       Mi.Insert(i-2,i-1,z);
137       Mi.Insert(i-1,i-2,z);
138     }
139
140      éléments diagonaux internes
141     for(int i=1; i<=ni; i++){
142       Gauss_Legendre_x(glx,grille.x[i-1],grille.x[i]);
143       for(int j=0; j<gln; j++) glf[j] = grille.tente(i,glx[j])*grille.tente(i,glx[j])
              *f(glx[j]);
144       z = Gauss_Legendre(glf);
145       Gauss_Legendre_x(glx,grille.x[i],grille.x[i+1]);
146       for(int j=0; j<gln; j++) glf[j] = grille.tente(i,glx[j])*grille.tente(i,glx[j])
              *f(glx[j]);
147       z += Gauss_Legendre(glf);
148       Mi.Insert(i-1,i-1,z);
149     }
150   }
```

Code 4.2 : Programme principal pour le calcul des premiers modes propres : `Helmholtz1D.cpp`

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
```

```
5  #include "maillage1D.h"
6  #include "read_parameter.h"
7
8   calcul des modes propres de l'équation de Helmholtz en 1D, conditions aux limites de Dirichlet
9  int main() {
10
11   int n_points;
12   maillage1D grille;
13   string filename;
14   ofstream fout;
15   ifstream fin("para.dat");
16
17   const bool dense = true; 'true' si on utilise des matrices denses
18
19    initialisation de la grille
20   if(fin == "grille"){ lecture d'une grille calculée au préalable (non uniforme)
21     fin >> filename;
22     grille.init(filename);
23   }
24   else{ construction d'une grille uniforme
25     double L;
26     fin >> "points" >> n_points;
27     fin >> "L" >> L;
28     grille.init(n_points,L);
29   }
30
31   int modes=0; nombre de modes calculés
32   fin >> "modes" >> modes;
33   fin.close();
34   int ni = grille.x.size()-2; nombre de points intérieurs
35   if(modes>ni) modes = ni;
36
37    matrices creuses représentant le laplacien
38   SparseMatrix Li(ni); pour les noeuds intérieurs
39   SparseMatrix Lb(ni,2); couplage intérieur−frontière
40   cout << "calcul du laplacien...\n";
41   grille.Laplacian(Li,Lb);
42
43   matrices creuses représentant la matrice de masse
44   SparseMatrix Mi(ni); pour les noeuds intérieurs
45   SparseMatrix Mb(ni,2); couplage intérieur−frontière
46   cout << "calcul de la matrice de masse...\n";
47   grille.mass_matrix(Mi,Mb);
48
49   cout << "calcul des vecteurs et valeurs propres...\n";
50
51   if(dense){
52     Matrix<double> Li_dense(ni);
53     Li.make_dense(Li_dense);
```

```
54      Matrix<double> Mi_dense(ni);
55      Mi.make_dense(Mi_dense);
56      Matrix<double> U(ni);
57      Vector<double> eigenvalue(ni);
58      Li_dense.generalized_eigensystem(Mi_dense,eigenvalue,U);
59      Li_dense.eigensystem(eigenvalue,U);
60
61       impression des valeurs propres et vecteurs propres
62      fout.open("vp.dat");
63      fout << "#x\t";
64      for(int j=0; j<modes; j++){
65        fout << eigenvalue[ni-j-1] << '\t';
66      }
67      fout << endl;
68      fout << grille.x[0] << '\t';
69      for(int j=0; j<modes; j++) fout << 0 << '\t';
70      fout << endl;
71      for(int i=0; i< ni; i++){
72        fout << grille.x[i+1] << '\t';
73        for(int j=0; j<modes; j++) fout << U(i,ni-j-1) << '\t';
74        fout << endl;
75      }
76      fout << grille.x[ni+1] << '\t';
77      for(int j=0; j<modes; j++) fout << 0 << '\t';
78      fout << endl;
79      fout.close();
80    }
81    else{
82      allocation de l'espace pour les valeurs propres et les vecteurs propres
83      Vector<double> eigenvalue(modes);
84      vector<Vector<double> > eigenvector(modes);
85      for(int i=0; i<modes; i++) eigenvector[i].Alloc(Li.rows());
86      Li.Lanczos_generalized(modes, eigenvalue, eigenvector, true, &Mi);
87      Li.Lanczos(modes, eigenvalue, eigenvector, true);
88
89       impression des valeurs propres et vecteurs propres
90      fout.open("vp.dat");
91      fout << "#x\t";
92      for(int j=0; j<modes; j++){
93        fout << eigenvalue[j] << '\t';
94      }
95      fout << endl;
96      fout << grille.x[0] << '\t';
97      for(int j=0; j<modes; j++) fout << 0 << '\t';
98      fout << endl;
99      for(int i=0; i< ni; i++){
100       fout << grille.x[i+1] << '\t';
101       for(int j=0; j<modes; j++) fout << eigenvector[j][i] << '\t';
102       fout << endl;
```

```
103       }
104       fout << grille.x[ni+1] << '\t';
105       for(int j=0; j<modes; j++) fout << 0 << '\t';
106       fout << endl;
107       fout.close();
108     }
109
110     cout << "Programme terminé normalement\n";
111   }
```

4.4.2 Fonctions tentes en dimension 2

Code 4.3 : Éléments finis en dimension 2 : `maillage2D.h`

```
1  #ifndef MAILLAGE2D_H_
2  #define MAILLAGE2D_H_
3
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  #include <cassert>
8
9  #include "SparseMatrix.h" classe représentant des matrices creuses
10
11 typedef struct{
12   double x; coordonnée x
13   double y; coordonnée y
14   int b; frontière à laquelle appartient le noeud (0=aucune)
15 } Node;
16
17 typedef struct{
18   int a; indice du 1er noeud
19   int b; indice du 2e noeud
20   int c; indice du 3e noeud
21   double ab; longueur au carré du lien ab
22   double bc; longueur au carré du lien bc
23   double ca; longueur au carré du lien ac
24   double A; aire de la face
25 } Face;
26
27 class maillage2D{
28 public:
29   int n; nombre de noeuds
30   int ni; nombre de noeuds intérieurs
31   int nb; nombre de noeuds à la frontière
32   int nf; nombre de faces
33   int na; nombre d'attributs par noeud
```

```
34
35    Node *node; tableau des noeuds
36    Face *face; tableau des faces
37    double **att; attributs de chaque noeud
38    string name; radical du nom du fichier contenant la description de la triangulation (pour
          lecture)
39
40    maillage2D(string &_name) : name(_name){
41
42        ouverture et lecture du fichier .node
43      string filename;
44      filename = name + ".node";
45      ifstream elem_file(filename.c_str());
46      assert(elem_file);
47
48      int dummy;
49      elem_file >> n >> dummy >> na >> dummy;
50
51       allocation des attributs (si présents)
52      double att_tmp[na];
53      if(na>0){
54        att = new double*[na];
55      }
56      for(int i=0; i<na; i++) att[i] = new double[n];
57
58       il faut réindexer les noeuds, pour séparer les noeuds intérieurs de la frontière
59       donc on introduit un index
60      int index[n+1];
61
62      node = new Node[n];
63      Node node_tmp;
64      int k=0;
65      for(int i=0; i<n; i++){
66        elem_file >> dummy >> node_tmp.x >> node_tmp.y;
67        assert(dummy == i+1);
68        for(int a=0; a<na; a++) elem_file >> att_tmp[a];
69        elem_file >> node_tmp.b;
70        if(node_tmp.b) continue;
71        index[i+1] = k;
72        if(na) memcpy(att[k],att_tmp,na*sizeof(*att_tmp));
73        node[k] = node_tmp;
74        k++;
75      }
76      ni = k; nombre de noeuds intérieurs
77      nb = n - ni;
78      elem_file.seekg(0);
79      elem_file >> n >> dummy >> na >> dummy;
80      for(int i=0; i<n; i++){
81        elem_file >> dummy >> node_tmp.x >> node_tmp.y;
```

```
82        for(int a=0; a<na; a++) elem_file >> att_tmp[a];
83        elem_file >> node_tmp.b;
84        if(node_tmp.b==0) continue;
85        index[i+1] = k;
86        if(na) memcpy(att[k],att_tmp,na*sizeof(*att_tmp));
87        node[k] = node_tmp;
88        k++;
89      }
90      elem_file.close();
91
92       ouverture et lecture du fichier .ele
93      filename = name + ".ele";
94      elem_file.open(filename.c_str());
95      assert(elem_file);
96
97      elem_file >> nf >> dummy >> dummy;
98      assert(dummy==0); on suppose aucun attribut pour les faces
99      face = new Face[nf];
100
101      lecture des faces et calcul de leurs propriétés
102     for(int i=0; i<nf; i++){
103       int a,b,c;
104       elem_file >> dummy >> a >> b >> c;
105       assert(dummy == i+1);
106       face[i].a = index[a];
107       face[i].b = index[b];
108       face[i].c = index[c];
109
110       Node na = node[face[i].a];
111       Node nb = node[face[i].b];
112       Node nc = node[face[i].c];
113       face[i].ab = (na.x-nb.x)*(na.x-nb.x)+(na.y-nb.y)*(na.y-nb.y);
114       face[i].bc = (nc.x-nb.x)*(nc.x-nb.x)+(nc.y-nb.y)*(nc.y-nb.y);
115       face[i].ca = (na.x-nc.x)*(na.x-nc.x)+(na.y-nc.y)*(na.y-nc.y);
116       face[i].A = 0.5*abs((nc.x*(na.y - nb.y) + na.x*(nb.y - nc.y) + nb.x*(nc.y -
              na.y)));
117     }
118     elem_file.close();
119   }
120
121    assignation des valeurs aux frontières
122   void set_BC(Vector<double> &rho, int border, int value){
123     for(int i=ni; i<n; i++) if(node[i].b == border) rho[i-ni] = value;
124   }
125
126    surcharge de l'opérateur de flux pour l'impression de la triangulation
127   friend std::ostream & operator<<(std::ostream &flux, const maillage2D &x){
128     flux << "Noeuds intérieurs :\n";
129     for(int i=0; i<x.ni; i++){
```

```
130         flux << i+1 << " :\t" << x.node[i].x << '\t' << x.node[i].y << endl;
131       }
132       flux << "Noeuds à la frontière :\n";
133       for(int i=x.ni; i<x.n; i++){
134         flux << i+1 << " :\t" << x.node[i].x << '\t' << x.node[i].y << '\t' << x.node
              [i].b << endl;
135       }
136       flux << "Triangles :\n";
137       for(int i=0; i<x.nf; i++){
138         flux << i+1 << " :\t" << x.face[i].a+1 << '\t' << x.face[i].b+1 << '\t' << x.
              face[i].c+1 << endl;
139       }
140
141       return flux;
142     }
143
144     méthode de calcul de la matrice de masse
145     Mi: partie intérieure
146     Mf: partie qui couple les noeuds de frontière aux noeuds intérieurs
147     void mass_matrix(SparseMatrix &Mi, SparseMatrix &Mf){
148       assert(Mi.rows() == ni);
149       assert(Mf.rows() == ni and Mf.columns() == nb);
150
151       on boucle sur les faces: chaque face ayant trois liens
152       on doit tenir compte des faces qui ont déjà été couvertes, pour ne pas répéter les liens
153       boucle sur les face
154       for(int i=0; i<nf; i++){
155         int na = face[i].a;
156         int nb = face[i].b;
157         int nc = face[i].c;
158
159         double aa = face[i].A/6;
160         if(na<ni) Mi.Insert(na,na,aa);
161         if(nb<ni) Mi.Insert(nb,nb,aa);
162         if(nc<ni) Mi.Insert(nc,nc,aa);
163         aa *= 0.5;
164
165         if(na < ni and nb < ni) {Mi.Insert(na,nb,aa); Mi.Insert(nb,na,aa);}
166         else if(na < ni and nb >= ni) Mf.Insert(na,nb-ni,aa);
167         else if(nb < ni and na >= ni) Mf.Insert(nb,na-ni,aa);
168
169         if(na < ni and nc < ni) {Mi.Insert(na,nc,aa); Mi.Insert(nc,na,aa);}
170         else if(na < ni and nc >= ni) Mf.Insert(na,nc-ni,aa);
171         else if(nc < ni and na >= ni) Mf.Insert(nc,na-ni,aa);
172
173         if(nb < ni and nc < ni) {Mi.Insert(nb,nc,aa); Mi.Insert(nc,nb,aa);}
174         else if(nb < ni and nc >= ni) Mf.Insert(nb,nc-ni,aa);
175         else if(nc < ni and nb >= ni) Mf.Insert(nc,nb-ni,aa);
176       }
```

```
177   }
178
179    méthode de calcul du laplacien
180    Li: partie intérieure
181    Lf: partie qui couple les noeuds de frontière aux noeuds intérieurs
182   void Laplacian(SparseMatrix &Li, SparseMatrix &Lf){
183     assert(Li.rows() == ni);
184     assert(Lf.rows() == ni and Lf.columns() == nb);
185
186      on boucle sur les faces: chaque face ayant trois liens
187      on doit tenir compte des faces qui ont déjà été couvertes, pour ne pas répéter les liens
188      boucle sur les face
189     for(int i=0; i<nf; i++){
190       int na = face[i].a;
191       int nb = face[i].b;
192       int nc = face[i].c;
193
194       double z = -0.25/face[i].A;
195       if(na<ni) Li.Insert(na, na, z*face[i].bc);
196       if(nb<ni) Li.Insert(nb, nb, z*face[i].ca);
197       if(nc<ni) Li.Insert(nc, nc, z*face[i].ab);
198
199       z *= 0.5;
200
201       double zz = z*(face[i].ab - face[i].bc - face[i].ca);
202       if(na < ni and nb < ni) {Li.Insert(na,nb,zz); Li.Insert(nb,na,zz);}
203       else if(na < ni and nb >= ni) Lf.Insert(na,nb-ni,zz);
204       else if(nb < ni and na >= ni) Lf.Insert(nb,na-ni,zz);
205
206       zz = z*(face[i].ca - face[i].ab - face[i].bc);
207       if(na < ni and nc < ni) {Li.Insert(na,nc,zz); Li.Insert(nc,na,zz);}
208       else if(na < ni and nc >= ni) Lf.Insert(na,nc-ni,zz);
209       else if(nc < ni and na >= ni) Lf.Insert(nc,na-ni,zz);
210
211       zz = z*(face[i].bc - face[i].ca - face[i].ab);
212       if(nb < ni and nc < ni) {Li.Insert(nb,nc,zz); Li.Insert(nc,nb,zz);}
213       else if(nb < ni and nc >= ni) Lf.Insert(nb,nc-ni,zz);
214       else if(nc < ni and nb >= ni) Lf.Insert(nc,nb-ni,zz);
215     }
216   }
217
218 #define MIN_AREA 0.0001
219    impression des triangles compatibles avec le programme "triangle", pour un éventuel raffinement
         de la triangulation
220    psi: solution à une étape donnée
221    rho: vecteur de charge
222    prec: précision requise
223    void print_area_file(Vector<double>&psi, Vector<double>&rho, double prec){
224      string filename;
```

```
225       filename = name + ".area";
226       ofstream fout(filename.c_str());
227       filename = name + ".err";
228       ofstream fout2(filename.c_str());
229       fout << nf << endl;
230       for(int i=0; i<nf; i++){ boucle sur les faces
231         double psi_a, psi_b, psi_c; valeurs du champs aux trois sommets de la face
232         Face F = face[i];
233         int k = F.a;
234         if(k < ni) psi_a = psi[k]; else psi_a = rho[k-ni];
235         k = F.b;
236         if(k < ni) psi_b = psi[k]; else psi_b = rho[k-ni];
237         k = F.c;
238         if(k < ni) psi_c = psi[k]; else psi_b = rho[k-ni];
239
240         double area; aire visée à la prochaine étape
241         area = psi_a*psi_a*F.bc + psi_b*psi_b*F.ca + psi_c*psi_c*F.ab + psi_a*psi_b*(
              F.ab-F.bc-F.ca) + psi_b*psi_c*(F.bc-F.ab-F.ca)+psi_c*psi_a*(F.ca-F.bc-F.
              ab);
242         area /= (4*F.A);
243         area = prec/area;
244         fout2 << 0.333333*(node[F.a].x+node[F.b].x+node[F.c].x) << '\t' << 0.333333*(
              node[F.a].y+node[F.b].y+node[F.c].y) << '\t' << area << endl;
245         if(area < MIN_AREA) area = MIN_AREA;
246         fout << i << '\t' << area << '\n';
247
248       }
249       fout.close();
250     }
251
252     void print(ostream &out, int M, double *psi, double *psif=0L){
253       for(int j=0; j<M; j++){
254         int offset = j*ni;
255         for(int i=0; i<ni; i++) out << node[i].x << '\t' << node[i].y << '\t' << psi[
              i+offset] << endl;
256         if(psif==0L){
257           for(int i=ni; i<n; i++) out << node[i].x << '\t' << node[i].y << '\t' << 0
                << endl;
258         }
259         else{
260           offset = j*(n-ni);
261           for(int i=ni; i<n; i++) out << node[i].x << '\t' << node[i].y << '\t' <<
                psif[i-ni+offset] << endl;
262         }
263         out << "\n\n";
264       }
265     }
266
267     ~maillage2D(){
```

```
268       for(int i=0; i<na; i++) delete[] att[i];
269       delete[] att;
270       delete[] node;
271       delete[] face;
272     }
273   };
274   #endif
```

Code 4.4 : Programme principal pour la solution de l'équation de Laplace : `Laplace2D.cpp`

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   #include "maillage2D.h"
6   #include "read_parameter.h"
7
8    résolution de l'équation de Laplace en 2D, conditions aux limites de Dirichlet
9   int main() {
10
11    string filename;
12    ofstream fout;
13    ifstream fin("para.dat");
14
15    fin >> "grille" >> filename;
16
17    maillage2D grille(filename);
18
19     matrices creuses représentant le laplacien
20    SparseMatrix Li(grille.ni); pour les noeuds intérieurs (L^I)
21    SparseMatrix Lf(grille.ni,grille.nb); couplage intérieur−frontière (L^F)
22    cout << "calcul du laplacien...\n";
23    grille.Laplacian(Li,Lf);
24
25    cout << "Solution du problème aux limites...\n";
26
27    Vector<double> psi(grille.ni); allocation de ψ^I
28
29    allocation et définition des conditions aux limites
30    Vector<double> psif(grille.nb); vecteur de charge ψ^F
31    int n_cond=0; nombre de conditions aux limites
32    double prec; facteur de précision requis dans le raffinement de la grille
33    fin >> "précision" >> prec;
34    fin >> "conditions_limites" >> n_cond;
35    for(int i=0; i<n_cond; i++){
36      int frontiere;
37      double val;
38      fin >> frontiere >> val;
```

```
39      grille.set_BC(psif,frontiere,val); assigne les valeurs de ψ^F à la frontière
40    }
41    fin.close();
42
43    on calcule D^{2F} ψ^F
44    Vector<double> b(grille.ni);
45    Lf.mult(b,psif);
46    b *= -1.0;
47    Li.ConjugateGradient(b, psi, 1.0e-6); méthode du gradient conjugué
48    fout.open("out.dat");
49    grille.print(fout,1,psi.array(),psif.array());
50    fout.close();
51
52    grille.print_area_file(psi, psif, prec); imprime un fichier en vue de raffiner la grille
53
54    cout << "Programme terminé normalement\n";
55  }
```

4.4.3 Représentation spectrale par collocation (domaine ouvert)

Code 4.5 : Fonctions cardinales `collocation.h`

```
1  #ifndef COLLOCATION_H
2  #define COLLOCATION_H
3
4  #include <cassert>
5  #include <iostream>
6  #include <string>
7  #include "Vector.h"
8  #include "Matrix.h"
9  using namespace std;
10
11 représentation d'une fonction à l'aide des fonctions cardinales
12 class collocation{
13 public:
14   int N; nombre de points sur la grille
15   double x1; début de l'intervalle
16   double L; largeur de l'intervalle
17   int max_order; ordre maximum des dérivées dans l'opérateur différentiel
18   int plot_points; nombre de points pour les graphiques
19   Vector<double> x; abcisses
20   Matrix<double> C; matrice des poids
21   Vector<Matrix<double> > D; matrice des dérivées
22
23    constructeur
24   collocation(string type, int _N, int _max_order, int _np) : N(_N), max_order(
         _max_order), plot_points(_np) {
```

```
25      x.Alloc(N);
26
27      if(type=="gauss_lobatto"){
28        x1 = -1.0;
29        L = 2.0;
30        for(int i=0; i<N; i++){
31          x[i] = cos(((N-1-i)*M_PI)/(N-1)); points de la grille de Gauss-Lobatto
32        }
33      }
34      else if(type=="tchebychev"){
35        x1 = -1.0;
36        L = 2.0;
37        for(int i=0; i<N; i++){
38          x[i] = cos(((N-i-0.5)*M_PI)/N); points de la grille de Tchébychev (points intérieurs)
39        }
40      }
41      else{
42        cout << "type de grille non reconnu (" << type << ")\n";
43        exit(1);
44      }
45      C.Alloc(N,max_order+1);
46      D.Alloc(max_order);
47
48       calcule les matrices de dérivées jusqu'à l'ordre M inclus
49      for(int k=0; k<max_order; k++) D[k].Alloc(N);
50      for(int i=0; i<N; i++){
51        poids(x[i],max_order);
52        for(int k=0; k<max_order; k++){
53          for(int j=0; j<N; j++) D[k](i,j) = C(j,k+1);
54        }
55      }
56    }
57
58     fonctions cardinales et leurs dérivées
59     routine "weights" de Numerical Recipes
60     D'arpès Fornberg. Voir http://www.scholarpedia.org/article/Finite_difference_method
61     z: position
62     m: ordre de la dérivée (0 pour les fonctions cardinales elles-mêmes)
63    void poids(const double z, int m)
64    {
65      assert(m<=max_order);
66      double c1=1.0;
67      double c4=x[0]-z;
68      C.clear();
69      C(0,0)=1.0;
70      for (int i=1; i<N; i++){
71        int mn = (i<m)? i:m;
72        double c2=1.0;
73        double c5=c4;
```

```
74          c4=x[i]-z;
75          for (int j=0; j<i; j++){
76            double c3=x[i]-x[j];
77            c2 *= c3;
78            if (j == (i-1)){
79              for(int k=mn; k>0; k--) C(i,k)=c1*(k*C(i-1,k-1)-c5*C(i-1,k))/c2;
80              C(i,0) = -c1*c5*C(i-1,0)/c2;
81            }
82            for (int k=mn; k>0; k--) C(j,k)=( c4*C(j,k) - k*C(j,k-1) )/c3;
83            C(j,0) *= c4/c3;
84          }
85          c1 = c2;
86        }
87      }
88
89       sortie de la fonction psi
90       sous forme d'un tableau linéaire contenant les M composantes de la fonction, chaque
           composante
91       étant inscrite sur une colonne différente du fichier
92      void plot(ostream &fout, double *psi, int M = 1){
93        double dz = L/(plot_points-1);
94        double z = x1;
95        for(int i=0; i<plot_points; i++, z += dz){
96          fout << z;
97          poids(z,0);
98          for(int k=0; k<M; k++){
99            double y = 0.0;
100           for(int j=0; j<N; j++) y += psi[k*N+j]*C(j,0);
101           fout << '\t' << y;
102         }
103         fout << endl;
104       }
105     }
106
107      sortie des fonctions cardinales (pour fins d'illustration seulement)
108     void plot_cardinal(ostream &fout){
109       double dz = L/(plot_points-1);
110       double z = x1;
111       for(int i=0; i<plot_points; i++, z += dz){
112         poids(z,0);
113         fout << z;
114         for(int j=0; j<N; j++) fout << '\t' << C(j,0);
115         fout << endl;
116       }
117     }
118
119 };
120
121 #endif
```

Chapitre 5

Équations aux dérivées partielles dépendant du temps

Nous allons traiter dans ce chapitre de la solution numérique des équations aux dérivées partielles qui impliquent une évolution temporelle, et donc qui requièrent des conditions initiales. Ce type de problème est différent des problèmes aux limites statiques, comme par exemple la solution de l'équation de Laplace, qui sont déterminés par des conditions aux frontières uniquement. Dans les problèmes dynamiques qui seront étudiés ici, nous aurons à la fois à tenir compte de conditions aux limites et de conditions initiales.

L'approche sera donc de considérer la représentation disrète d'un champ $\psi(\mathbf{r},t)$ à un instant donné et de faire évoluer dans le temps cette représentation discrète. Celle-ci pourra être basée sur l'une des méthodes décrites au chapitre précédent : soit une grille régulière, une représentation par éléments finis ou par une méthode spectrale.

5.1 L'équation de diffusion

5.1.1 Introduction

L'équation de diffusion, ou équation de la chaleur, prend la forme suivante :

$$\frac{\partial \psi}{\partial t} = \kappa \nabla^2 \psi \tag{5.1}$$

où κ est le coefficient de diffusion. Le champ ψ peut représenter la température locale à l'intérieur d'un matériau, ou la densité locale d'un soluté dans un solvant, etc. On peut justifier l'équation (5.1) de manière assez générale en supposant que la quantité ψ représente une quantité conservée (c'est-à-dire qui n'est pas localement créée) dont la diffusion d'un endroit à l'autre est caractérisée par une densité de courant $\mathbf{j}$ proportionnelle au gradient de ψ :

$$\mathbf{j} = -\kappa \nabla \psi \tag{5.2}$$

où le signe $-$ signidie que la quantité en question a tendance à se répartir là où elle est la plus faible. En combinant cette relation à l'équation de continuité

$$\frac{\partial \psi}{\partial t} + \nabla \cdot \mathbf{j} = 0 \tag{5.3}$$

qui ne fait que représenter la conservation de la quantité en question, on retrouve bien l'équation de diffusion.

Dans le cas de la chaleur, l'équation (5.2) est l'expression de la loi de conduction linéaire de la chaleur (ou loi de Fourier) qui stipule que le courant de chaleur $\mathbf{q}$ (en W/m^2) et proportionnel au gradient de température :

$$\mathbf{q} = -k\nabla T \qquad\qquad k : \text{conductivité thermique, en W/cm.K} \tag{5.4}$$

La variation d'énergie interne par unité de volume (ΔQ) est proportionnelle à la variation de température :

$$\Delta Q = c_p \rho \Delta T \tag{5.5}$$

où c_p est la chaleur spécifique à pression constante et ρ la densité du milieu. Le courant de chaleur $\mathbf{q}$ étant associé à Q, le courant de température $\mathbf{j}$ sera donc proportionnel à $\mathbf{q}$: $\mathbf{j} = \mathbf{q}/(c_p\rho)$. La loi de Fourier se réduit donc à la relation (5.2), avec $\kappa = k/(c_p\rho)$.

5.1.2 Évolution directe en dimension un

Considérons l'équation de diffusion en une dimension d'espace, entre les frontières $x = 0$ et $x = \ell$. La méthode la plus simple pour résoudre l'équation numériquement est d'adopter une discrétisation uniforme de l'intervalle $[0, \ell]$, et une discrétisation également uniforme du temps, de sorte qu'on doit traiter l'évolution d'un vecteur ψ_r, en fait une suite $\psi_{r,n}$, où r est l'indice spatial et n l'indice temporel. En utilisant la valeur de la dérivée temporelle calculée au temps t_n et la méthode d'Euler pour l'évolution dans le temps, on obtient le système d'équations suivant :

$$\psi_{r,n+1} = \psi_{r,n} + \eta\left[\psi_{r+1,n} - 2\psi_{r,n} + \psi_{r-1,n}\right] \qquad\qquad \eta = \frac{h\kappa}{a^2} \tag{5.6}$$

où $h = \Delta t$ est le pas temporel et $a = \Delta x$ le pas spatial. On doit appliquer cette relation aux valeurs intérieures de ψ, et imposer les conditions aux limites en tout temps aux frontières :

$$\psi_{0,n} = \psi_{\text{gauche}} \qquad\qquad \psi_{N,n} = \psi_{\text{droite}} \tag{5.7}$$

Il faut également spécifier les conditions initiales $\psi_{r,0}$. Une fois cela fait, la solution est immédiate par récurrence : les valeurs $\psi_{r,n+1}$ sont données de manière explicite en fonction des $\psi_{r,n}$. Pour cette raison, cette méthode simple est qualifiée d'*explicite*.

Analyse de stabilité de von Neumann

La méthode explicite décrite ci-dessus cours cependant le danger d'être instable. Von Neumann a étudié ce problème en supposant que la solution du problème discrétisé, qui est encore un problème linéaire, serait une combinaison de modes propres. Il a supposé une forme exponentielle pour les modes propres, qui seraient indexés par un nombre d'onde k, en fonction de la position et du temps $t = nh$:

$$\psi_{r,n} = \xi(k)^n \mathrm{e}^{ikar} \tag{5.8}$$

où ξ est un nombre complexe qui peut dépendre de k. En substituant cette forme dans l'équation discrète (5.6), on trouve

$$\xi(k) = 1 + 2\eta(\cos(ka) - 1) = 1 - 4\eta \sin^2(ka/2) \tag{5.9}$$

La solution numérique sera stable seulement si toutes les valeurs possible de ξ respectent la condition $|\xi(k)| < 1$. Ceci n'est vrai que si la condition suivante est respectée :

$$\eta < \frac{1}{2} \quad \text{ou} \quad \kappa h < \frac{a^2}{2} \tag{5.10}$$

Autrement dit, le pas temporel doit être suffisamment petit en comparaison du pas spatial. Dans le cas de l'équation de diffusion, le pas temporel doit même varier en raison quadratique du pas spatial.

Exercice 5.1: Test de la condition de von Neumann

Écrivez un programme très simple qui permet de tester la condition de stabilité de von Neumann pour l'équation de la diffusion en une dimension. Posez $\ell = 1$ et les conditions aux limites $\psi(0) = 0$ et $\psi(\ell) = 1$.

5.1.3 Méthode implicite de Crank-Nicholson

Le défaut principal de la méthode simple décrite à la section précédente est que la dérivée temporelle utilisée pour passer du temps t au temps $t + h$ est évaluée au temps t, c'est-à-dire au début de l'intervalle, comme dans la méthode d'Euler. Cette méthode est du premier ordre en h. La précision et la stabilité de la méthode est grandement améliorée si la dérivée est estimée au milieu de l'intervalle.

Supposons que l'évolution temporelle ait la forme suivante :

$$\frac{\partial \psi}{\partial t} = \mathscr{L}\psi \tag{5.11}$$

où $\mathscr{L}$ est un opérateur différentiel linéaire mais qui n'agit que sur la dépendance spatiale de ψ. La méthode simple décrite plus haut équivaut à la récurrence suivante :

$$\psi(t + h) = \psi(t) + h\mathscr{L}\psi(t) \tag{5.12}$$

Nous allons améliorer cette façon de faire en partant d'une valeur inconnue $\psi(t + h/2)$ évaluée au milieu de l'intervalle et en la propageant de $\Delta t = h/2$ vers $\psi(t + h)$ et de $\Delta t = -h/2$ vers $\psi(t)$:

$$\psi(t) = \left[1 - \tfrac{1}{2}h\mathscr{L}\right]\psi(t + h/2) \qquad \psi(t + h) = \left[1 + \tfrac{1}{2}h\mathscr{L}\right]\psi(t + h/2) \tag{5.13}$$

La première de ces équations est un système linéaire qui peut être résolu pour $\psi(t+h/2)$, connaissant $\psi(t)$. En pratique, nous aurons une version discrète du champ ψ et de l'opérateur $\mathscr{L}$, suite à une représentation du champ ψ selon l'une des méthodes décrites au chapitre précédent. L'équation ci-haut devient donc

$$\psi(t) = \left[1 - \tfrac{1}{2}hL\right]\psi(t+h/2) \qquad\qquad \psi(t+h) = \left[1 + \tfrac{1}{2}hL\right]\psi(t+h/2) \tag{5.14}$$

où ψ est maintenant un vecteur fini et L une matrice carrée. Une fois $\psi(t+h/2)$ connu par solution de la première équation, on applique la deuxième équation de manière directe pour obtenir $\psi(t+h)$.

La résolution de la première des équations (5.14) pour $\psi(t+h/2)$ est particulièrement simple en dimension 1 si on utilise une représentation par grille simple ou par éléments finis de la fonction ψ, car la matrice L est alors tridiagonale si $\mathscr{L}$ contient des dérivées du deuxième ordre ou moins. Un tel système s'inverse en un temps d'ordre $\mathcal{O}(N)$, N étant le nombre de points sur la grille. Par contre, une représentation spectrale demanderait d'inverser une matrice pleine. Si l'équation différentielle est linéaire et homogène dans le temps, et que le pas h est constant, alors le calcul d'une seule matrice inverse est suffisant pour tout le calcul. Si l'équation est non linéaire, alors l'opérateur L dépend de ψ et l'inversion doit être faite à chaque étape $t \to t+h$.

Analyse de stabilité

Supposons pour simplifier les choses que l'équation différentielle soit linéaire et homogène dans le temps, ce qui revient à dire que la matrice L ci-haut est constante. Dans ce cas, l'évolution du système du temps t au temps $t+h$ se fait par application d'une matrice d'évolution constante U :

$$\psi(t+h) = U\psi(t) \quad \text{où} \quad U \stackrel{\text{def}}{=} \frac{1 + \frac{1}{2}hL}{1 - \frac{1}{2}hL} \tag{5.15}$$

La stabilité de cette évolution est régie par les valeurs propres de l'opérateur U : si au moins une valeur u de U est telle que $|u| > 1$, alors la méthode est instable, car la moindre composante de ψ le long du vecteur-propre correspondant va être amplifiée par l'évolution temporelle. Si λ désigne une valeur propre de l'opérateur L, alors la valeur propre correspondante de U est simplement

$$u = \frac{1 + \frac{1}{2}h\lambda}{1 - \frac{1}{2}h\lambda} \tag{5.16}$$

car U est une fonction directe de la matrice L. La condition de stabilité $|u| \leq 1$ se traduit donc par

$$-1 < \frac{1 + \frac{1}{2}h\lambda}{1 - \frac{1}{2}h\lambda} < -1 \implies \tfrac{1}{2}h\lambda < 1 \tag{5.17}$$

Dans le cas particulier de l'équation de la chaleur, les valeurs propres λ du laplacien D^2 étant toujours négatives, la stabilité de la méthode est assurée.

5.1.4 Méthode du saute-mouton

Une façon alternative de procéder à l'évolution temporelle sans avoir à résoudre le système implicite (5.14) tout en conservant une précision du deuxième ordre en h est la méthode dite du «saute-mouton» (angl. *leapfrog*). Il s'agit simplement de calculer la dérivée au temps t à l'aide de la fonction (inconnue) au temps $t+h$ et de la fonction (conservée en mémoire) au temps $t-h$. Autrement dit :

$$\mathscr{L}\psi(t) = \frac{\partial\psi}{\partial t} \approx \frac{1}{2h}\left[\psi(t+h) - \psi(t-h)\right] \tag{5.18}$$

ce qui mène à une expression *explicite* pour $\psi(t+h)$:

$$\psi(t+h) = \psi(t-h) + 2h\mathscr{L}\psi(t) \tag{5.19}$$

Cette approche requiert de garder en mémoire, à chaque étape, la valeur du champ ψ à trois temps différents : elle est plus coûteuse en mémoire, mais potentiellement plus rapide qu'une méthode implicite dans les cas où le système (5.14) doit être résolu à chaque étape, surtout en dimensions supérieures quand les matrices en jeu ne sont pas tridiagonales.

Appliquons à cette méthode la même analyse de von Neumann que nous avons appliquée à la méthode directe, c'est-à-dire en nous basant sur les différences finies dans l'espace. On trouve alors la relation

$$\xi(k) = \xi^{-1}(k) + 2\eta(\cos(ka) - 1) \implies \xi^2 - 1 = 2\eta\xi(\cos(ka) - 1) \tag{5.20}$$

La condition de stabilité $\xi^2 < 1$ revient donc à demander

$$\eta\xi(\cos(ka) - 1) < 0 \tag{5.21}$$

ce qui n'est respecté que si $\xi > 0$. Or, dans les conditions où la méthode directe (5.6) devient instable, ξ est proche de -1, donc déjà négatif. La méthode du saute-mouton, en dépit d'être du deuxième ordre en temps, est donc instable, même pour des h très petits !

Donc malgré ses avantages potentiels, la méthode du saute-mouton est pleine de dangers ! Elle n'est pas toujours à conseiller : cela dépend de l'équation différentielle considérée.

5.1.5 Application basée sur une représentation spectrale

Le code déployé ci-dessous utilise la représentation spectrale d'une fonction en dimension 1, dans l'intervalle $[-1,1]$, et résoud l'équation de diffusion en fonction du temps, en affichant la solution de manière interactive à l'aide de gnuplot. Le schéma de Crank-Nicholson est utilisé pour propager la solution d'un temps à l'autre. Un opérateur d'évolution `U`, indépendant du temps, est construit au début du calcul et est ensuite appliqué à la solution à chaque instant t pour la propager vers l'instant suivant $(t+h)$.

La grille de Gauss-Lobatto est utilisée dans le calcul, mais seuls les points intérieurs doivent être propagés. Élaborons sur ce point : rappelons que dans la méthode de collocation, l'équation

$\dot{\psi} = \mathscr{L}\psi$ doit être imposée aux points intérieurs seulement. En fonction des sous-vecteurs ψ^F et ψ^I représentant respectivement les valeurs aux extrémités et à l'intérieur du domaine, la version discrétisée de l'équation prend la forme

$$\frac{\partial}{\partial t}\begin{pmatrix}\psi^F\\ \psi^I\end{pmatrix} = \begin{pmatrix}L^F & L^{FI}\\ L^{IF} & L^I\end{pmatrix}\begin{pmatrix}\psi^F\\ \psi^I\end{pmatrix} \quad \text{ou encore} \quad \frac{\partial \psi^I}{\partial t} = L^I\psi^I + L^{IF}\psi^F \tag{5.22}$$

où seule la deuxième composante de l'équation matricielle est appliquée, ψ^F étant fixé par les conditions aux limites, indépendamment du temps.

Supposons maintenant que les conditions aux limites soient homogènes, c'est-à-dire $\psi^F = 0$. On retrouve alors la même forme que l'équation différentielle originale obtenue sans tenir compte des conditions aux limites, mais pour les points intérieurs seulement. La méthode de Crank-Nicholson implique donc l'opérateur d'évolution suivant :

$$U = \frac{1 + \frac{1}{2}hL^I}{1 - \frac{1}{2}hL^I} \tag{5.23}$$

qui soit être construit et appliqué sur le vecteur ψ^I de manière répétée.

Si les conditions aux limites ne sont pas homogènes, alors on peut sans trop de peine ramener le problème à celui de conditions aux limites homogènes de la manière suivante : on pose

$$\psi(x,t) = \phi(x,t) + \psi_s(x) \tag{5.24}$$

où par définition $\psi_s(x)$ est une solution statique (indépendante du temps) à l'équation différentielle, et qui en plus respecte les bonnes conditions aux limites. Comme l'équation est linéaire, on a donc

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= \mathscr{L}\phi \quad \text{où} \quad \phi(-1) = \phi(1) = 0 \\ \mathscr{L}\psi_s &= 0 \quad \text{où} \quad \psi_s(-1) = \psi(-1) \quad \text{et} \quad \psi_s(1) = \psi(1) \end{aligned} \tag{5.25}$$

La solution au problème indépendant du temps $\mathscr{L}\psi_s = 0$ est généralement simple. Dans le cas de l'équation de diffusion, $\partial_x^2\psi_s = 0$ et donc $\psi_s(x)$ est une fonction linéaire qui interpole entre les valeurs aux extrémités de l'intervalle. Cette fonction est en fait la valeur asymptotique de $\psi(x,t)$ dans la limite $t \to \infty$. Il nous reste alors à déterminer numériquement $\phi(x,t)$, c'est-à-dire à résoudre un problème aux limites homogènes.

Code 5.1 : Résolution de l'équation de diffusion 1D

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 #define GNUPLOT
6
7 #ifdef GNUPLOT
8 extern "C"{
9 #include "gnuplot_i.h"
```

```
10 }
11 #endif
12
13 #include "collocation.h"
14 #include "read_parameter.h"
15
16 int main(){
17   double kappa; coefficient de diffusion
18   int np=50; nombre de points sur le graphique
19   double h; pas temporel
20   double dt_display = 10*h; intervalle d'affichage
21   double tmax; temps de simulation
22   int N; nombre de points de collocation
23   double x1 = -1.0; extrémité gauche
24   double x2 = 1.0; extrémité droite
25
26   ifstream fin("para.dat");
27   fin >> "N" >> N;
28   fin >> "kappa" >> kappa;
29   fin >> "h" >> h;
30   fin >> "intervalle_affichage" >> dt_display;
31   fin >> "tmax" >> tmax;
32
33   int Ni = N-2; nombre de points intérieurs
34
35   Vector<double> psi(N); champ
36   Matrix<double> U(Ni); opérateur d'évolution
37   collocation grille("gauss_lobatto", N, 2, np); Grille de fonctions cardinales
38
39   Matrix<double> Li(N-2); construction de la partie intérieure du laplacien
40   int N1 = N-1;
41   for(int i=1; i<N1; i++) for(int j=1; j<N1; j++) Li(i-1,j-1) = grille.D[1](i,j);
42
43    construction de l'opérateur d'évolution (Crank−Nicholson)
44   Matrix<double> A(Li); A *= 0.5*kappa*h; A.add_to_diagonal(1.0); A = 1 + 1/2 κhL^I
45   Matrix<double> B(Li); B *= -0.5*kappa*h; B.add_to_diagonal(1.0); B = 1 − 1/2 κhL^I
46   B.Inverse();
47   U = B*A; U = (1 − 1/2 κhL^I)^{-1}.(1 + 1/2 κhL^I)
48
49    conditions initiales
50   for(int i=0; i<grille.N; i++) psi[i] = 1.0;
51   psi[0] = 0.0;
52   psi[grille.N-1] = 0.0;
53
54    initialisation de la sortie des données
55 #ifdef GNUPLOT
56   int temps_mort; temps mort entre affichages, pour gnuplot_i
57   fin >> "temps_mort" >> temps_mort;
```

```
58    double ymin = 0;
59    double ymax = 1;
60    ofstream gout;
61    gnuplot_ctrl * GP;
62    GP = gnuplot_init() ;
63    gnuplot_cmd(GP,(char *)"set term x11");
64    gnuplot_cmd(GP,(char *)"set xr [%g:%g]",x1,x2);
65    gnuplot_cmd(GP,(char *)"set yr [%g:%g]",ymin,ymax);
66  #endif
67
68    fin.close();
69    int output_step=0; compteur des sorties sur fichier
70    for(double t=0.0; t<tmax; t+= h){
71      if(floor(t/dt_display) >= output_step){
72        output_step++;
73  #ifdef GNUPLOT
74        gout.open("tmp.dat");
75        grille.plot(gout,psi.array());
76        gout.close();
77        gnuplot_cmd(GP,(char *)"set title 't = %g'",t);
78        gnuplot_cmd(GP,(char *)"plot 'tmp.dat' u 1:2 w l t ''");
79        usleep(temps_mort); temps mort entre les affichages, en microsecondes
80  #endif
81      }
82       évolution sur un intervalle de temps h
83      Vector<double> tmp(Ni);
84      U.mult_add(&psi[1],&tmp[0]);
85      memcpy(&psi[1],&tmp[0],Ni*sizeof(double));
86    }
87
88  #ifdef GNUPLOT
89    sleep(3); temps mort après le dernier affichage, en secondes
90    gnuplot_close(GP) ;
91  #endif
92
93    cout << "Programme terminé normalement\n";
94  }
```

Exercice 5.2: Résolution de l'équation de diffusion par représentation spectrale

A Installez le programme `diffusion1D.cpp` et testez-le avec les paramètres suivants figurant dans le fichier `para.dat` :

```
1  N 10
2  kappa 1
3  h 0.001
4  intervalle_affichage 0.01
```

```
5 tmax   2
6 temps_mort 100000
```

B Portez en graphique la solution à $t = 0.1$ pour les quatre valeurs de N étudiées. Vous pouvez à cette fin stipuler que le temps de simulation est `tmax=0.5`.

5.2 Propagation d'une onde et solitons

5.2.1 Équation d'advection

L'équation la plus simple décrivant la propagation d'une onde est l'*équation d'advection* en une dimension d'espace :

$$\frac{\partial \psi}{\partial t} + v\frac{\partial \psi}{\partial x} = 0 \tag{5.26}$$

Cette équation est un cas particulier de l'équation de continuité

$$\frac{\partial \psi}{\partial t} + \nabla\cdot\mathbf{j} = 0 \quad \text{où} \quad \mathbf{j} = (v\psi, 0, 0) \tag{5.27}$$

Le champ ψ représente donc la densité d'une quantité conservée, mais qui ne peut se propager que vers la droite (si $v > 0$).

La solution analytique de l'équation d'advection est très simple : on procède au changement de variables

$$\xi = x + vt \qquad \eta = x - vt \tag{5.28}$$

de sorte que

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial \xi} + \frac{\partial}{\partial \eta} \qquad \frac{\partial}{\partial t} = v\frac{\partial}{\partial \xi} - v\frac{\partial}{\partial \eta} \tag{5.29}$$

L'équation d'advection s'écrit comme suit en fonction des variables (ξ, η) :

$$\frac{\partial \psi}{\partial \xi} = 0 \quad \text{et donc} \quad \psi(x,t) = \psi_0 u(\eta) = \psi_0 u(x - vt) \tag{5.30}$$

où $u(\eta)$ est une fonction différentiable quelconque, représentant la forme du paquet d'ondes qui se propage vers la droite.

5.2.2 Équation de Korteweg-de Vries

L'étude de la propagation des vagues – l'archétype des ondes dans l'histoire des sciences – est un sujet relativement complexe. La vague elle-même est une interface entre deux milieux (le liquide en bas, l'air ou le vide en haut) dont la forme varie en fonction du temps. La manière dont cette interface se déplace repose sur une description de l'écoulement du fluide sous-jacent, par les équations standards de l'hydrodynamique (éq. de Navier-Stokes et de continuité).

L'étude théorique et expérimentale de la propagation des vagues a été motivée en bonne partie par l'observation en 1834, par l'ingénieur naval John Scott Russell, de la propagation d'une vague singulière dans un canal en Écosse :

> I was observing the motion of a boat which was rapidly drawn along a narrow channel by a pair of horses, when the boat suddenly stopped—not so the mass of water in the channel which it had put in motion ; it accumulated round the prow of the vessel in a state of violent agitation, then suddenly leaving it behind, rolled forward with great velocity, assuming the form of a large solitary elevation, a rounded, smooth and well-defined heap of water, which continued its course along the channel apparently without change of form or diminution of speed. I followed it on horseback, and overtook it still rolling on at a rate of some eight or nine miles an hour, preserving its original figure some thirty feet long and a foot to a foot and a half in height. Its height gradually diminished, and after a chase of one or two miles I lost it in the windings of the channel. Such, in the month of August 1834, was my first chance interview with that singular and beautiful phenomenon which I have called the Wave of Translation.

Russell procéda par la suite à une étude contrôlée de la propagation des vagues dans un petit bassin artificiel qu'il fit creuser chez lui. L'une des découvertes qualitatives qu'il fit est que la vitesse de propagation augmente avec la profondeur du bassin (en autant qu'il ne soit pas trop profond). Donc, si ψ représente la hauteur de la vague, la vitesse est proportionnelle à ψ.

Les calculs théoriques du Français Boussinesq, et ensuite des Hollandais Korteweg et de Vries, on mené à la forme simplifiée suivante pour l'équation décrivant la propagation d'une onde en une dimension, dans un canal étroit et plat :

$$\frac{\partial \psi}{\partial t} + \epsilon \psi \frac{\partial \psi}{\partial x} + \mu \frac{\partial^3 \psi}{\partial x^3} = 0 \tag{5.31}$$

C'est l'*équation de Korteweg-de Vries* (ou KdeV). Elle diffère de l'équation d'advection de deux manières :

1. La vitesse de propagation dépend de l'amplitude de l'onde : v a été remplacé par $\epsilon\psi$.
2. Un terme de dispersion a été ajouté, proportionnel à la dérivée troisième de ψ. Ce terme provoque la dispersion – l'étalement d'un paquet d'onde initial – car la vitesse de propagation dépend alors du nombre d'onde, et donc n'est pas constante pour une onde non sinusoïdale.

L'équation d'advection est retrouvée si on néglige la dispersion, et si on linéarise le terme non linéaire : on pose $\psi = \psi_0 + \delta\psi$, ψ_0 étant l'élévation moyenne de la surface, et on néglige les termes quadratiques en $\delta\psi$. On retrouve alors l'équation d'advection pour la déviation $\delta\psi$ par rapport à l'élévation moyenne.

Il est pratique de conserver les paramètres ϵ et μ dans l'étude de cette équation, car on peut alors facilement retrouver diverses limites. Par exemple, en posant $\mu = 0$, on trouve l'*équation de Burgers*, qui constitue un modèle simple des ondes de choc. En supposant cependant que ϵ et μ sont non nuls, il est toujours possible de procéder à un changement d'échelle de x pour fixer μ à l'unité, et à un changement d'échelle de ψ (l'équation étant non linéaire) pour fixer $\epsilon = 6$. On obtient ainsi une forme normalisée de l'équation de KdeV :

$$\frac{\partial \psi}{\partial t} + \frac{\partial^3 \psi}{\partial x^3} + 6\psi \frac{\partial \psi}{\partial x} = 0 \tag{5.32}$$

5.2.3 Solitons

Nous allons maintenant démontrer que l'équation de KdeV (5.31) admet comme solution particulière des ondes d'étendue finie qui se propagent sans déformation, ou *solitons* (parfois aussi appelée *ondes solitaires*). Celles-ci correspondent à l'onde de translation (*wave of translation*) observée par Scott Russell.

Commençons par supposer une solution de la forme $\psi(x,t) = u(x - vt)$ et substituons dans l'équation de KdeV ; on obtient une équation différentielle ordinaire en fonction de $\eta = x - vt$, ou en fonction de x si on pose $t = 0$:

$$-vu' + u''' + 6uu' = 0 \quad \text{ou} \quad -vu' + u''' + 3(u^2)' = 0 \tag{5.33}$$

ce qui peut s'intégrer une fois par rapport à x pour donner

$$-vu + u'' + 3u^2 = A \tag{5.34}$$

où A est une constante d'intégration. Si on suppose que la solution u tend vers zéro en même temps que ses dérivées (loin du maximum du paquet d'onde), on doit poser $A = 0$. En multipliant par u', qui sert de facteur intégrant, on trouve

$$-vuu' + u'u'' + 3u'u^2 = 0 = \frac{\mathrm{d}}{\mathrm{d}x}\left[-\frac{1}{2}vu^2 + \frac{1}{2}(u')^2 + u^3\right] \tag{5.35}$$

Donc l'expression entre crochets est indépendante de x, et doit être nulle quand $x \to \pm\infty$. On peut alors isoler u' :

$$-\frac{1}{2}vu^2 + \frac{1}{2}(u')^2 + u^3 = 0 \implies \frac{\mathrm{d}u}{\mathrm{d}x} = u\sqrt{v - 2u} \tag{5.36}$$

ce qui nous permet d'intégrer :

$$x = \int \frac{\mathrm{d}u}{u\sqrt{v - 2u}} = \frac{2}{\sqrt{v}} \operatorname{arctanh} \sqrt{1 - \frac{2u}{v}} \tag{5.37}$$

En isolant u, on trouve

$$u(x) = \frac{v}{2}\left[1 - \tanh^2\left(\frac{\sqrt{v}x}{2}\right)\right] = \frac{v/2}{\cosh^2\left(\frac{\sqrt{v}x}{2}\right)} \tag{5.38}$$

Nous avons donc trouvé une solution à l'équation de KdeV de la forme suivante :

$$\psi(x,t) = \frac{v/2}{\cosh^2\left[\frac{\sqrt{v}}{2}(x-vt)\right]} \tag{5.39}$$

Notons que l'amplitude du soliton est proportionnelle à sa vitesse.

Exercice 5.3: Mise à l'échelle des solitons

On peut écrire la solution solitonique comme suit :

$$\psi(x,t) = \frac{A}{\cosh^2\left[(x-vt)/\sigma\right]} \tag{5.40}$$

où A est l'amplitude maximale du soliton et σ sa largeur caractéristique. En partant de la solution (5.39) à l'équation (5.32), écrivez la solution correspondante à l'équation (5.31) sous la forme ci-haut en procédant aux transformations d'échelle nécessaires. Exprimez A, σ en fonction de ϵ, μ et de la vitesse du soliton.

5.2.4 Solution numérique de l'équation de Korteweg-de Vries

Nous allons utiliser des conditions aux limites périodiques et une méthode spectrale pour étudier la propagation des solitons. À cette fin, la classe `collocation_periodic.h` a été construite afin de représenter des fonctions cardinales associées aux séries de Fourier.

Code 5.2 : Classe de fonctions cardinales

```
1  #ifndef COLLOCATION_PERIODIC_H
2  #define COLLOCATION_PERIODIC_H
3
4  #include <cassert>
5  #include <iostream>
6  #include <string>
7  #include "Vector.h"
8  #include "Matrix.h"
9  using namespace std;
10
11 représentation d'une fonction à l'aide des fonctions cardinales associées aux séries de Fourier
12 class collocation_periodic{
13 public:
14   int N; nombre de points sur la grille
15   double L; largeur de l'intervalle
16   int max_order; ordre maximum des dérivées dans l'opérateur différentiel
17   int plot_points; nombre de points pour les graphiques
18   Vector<double> x; abcisses
19   Vector<Matrix<double> > D; matrice des dérivées
20
```

```
21    constructeur
22    collocation_periodic(int _N, double _L, int _max_order, int _np) : N(_N), L(_L),
          max_order(_max_order), plot_points(_np) {
23      x.Alloc(N);
24      for(int i=0; i<N; i++) x[i] = i*2*M_PI/N;
25      D.Alloc(max_order);
26
27      calcule les matrices de dérivées jusqu'à l'ordre M inclus
28      D[0].Alloc(N);
29      for(int i=0; i<N; i++){
30        for(int j=0; j<i; j++){
31          D[0](i,j) = 0.5/tan(0.5*(x[i]-x[j]));
32          if((i-j)%2) D[0](i,j) = -D[0](i,j);
33          D[0](j,i) = -D[0](i,j);
34        }
35      }
36
37      if(max_order>1){
38        D[1].Alloc(N);
39        for(int i=0; i<N; i++){
40          D[1](i,i) = -(1.0+0.5*N*N)/6;
41          for(int j=0; j<i; j++){
42            double z = sin(0.5*(x[i]-x[j]));
43            D[1](i,j) = 0.5/(z*z);
44            if((i-j+1)%2) D[1](i,j) = -D[1](i,j);
45            D[1](j,i) = D[1](i,j);
46          }
47        }
48      }
49      for(int j=2; j<max_order; j++) D[j] = D[j-1]*D[0];
50
51    }
52
53    fonctions cardinales
54    inline double cardinal(int i, double z){
55      return sin(0.5*N*(z-x[i]))/(N*tan(0.5*(z-x[i])));
56    }
57
58    sortie de la fonction psi
59    void plot(ostream &fout, double *psi, int M = 1){
60      double dz = L/(plot_points-1);
61      double z = 0;
62      for(int i=0; i<plot_points; i++, z += dz){
63        fout << z;
64        for(int k=0; k<M; k++){
65          double y = 0.0;
66          for(int j=0; j<N; j++) y += psi[k*N+j]*cardinal(j,z);
67          fout << '\t' << y;
68        }
```

```
69          fout << endl;
70        }
71      }
72
73      sortie des fonctions cardinales
74      void plot_cardinal(ostream &fout){
75        double dz = L/(plot_points-1);
76        double z = 0;
77        for(int i=0; i<plot_points; i++, z += dz){
78          fout << z;
79          for(int j=0; j<N; j++) fout << '\t' << cardinal(j,z);
80          fout << endl;
81        }
82      }
83
84  };
85
86  #endif
```

Code 5.3 : Résolution de l'équation de KdeV

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   #define GNUPLOT
6
7   #ifdef GNUPLOT
8   extern "C"{
9   #include "gnuplot_i.h"
10  }
11  #endif
12
13  #include "collocation_periodic.h"
14  #include "read_parameter.h"
15
16  routine qui construit un soliton de large sigma et d'amplitude A centré à A
17  x: grille de points
18  psi: champ auquel on ajoute le soliton
19  void soliton(Vector<double> &x, Vector<double> &psi, double x0, double A, double
        sigma){
20    for(int i=0; i<x.size(); i++){
21      double v = (x[i]-x0)/sigma;
22      v = 1.0/cosh(v);
23      v *= v;
24      psi[i] += A*v;
25    }
26  }
```

```
27 int main(){
28   Vector<double> psi; champ
29   int N; nombre de points de collocation
30   int np; nombre de points sur le graphique
31   double h; pas temporel
32   double dt_display; intervalle d'affichage
33   double tmax; temps de simulation
34   bool deux_solitons = false; vrai si la valeur initiale comporte deux solitons
35   bool lineaire = false; vrai si on modifie l'équation pour la rendre linéaire
36
37   ifstream para("para.dat");
38   para >> "N" >> N;
39   para >> "plot_points" >> np;
40   para >> "h" >> h;
41   para >> "temps_de_simulation" >> tmax;
42   para >> "intervalle_affichage" >> dt_display;
43
44   double mu; paramètre de dispersion dans l'équation KdV
45   double epsilon; paramètre non linéaire dans l'équation KdV
46   para >> "mu" >> mu;
47   para >> "epsilon" >> epsilon;
48
49   conditions initiales
50   double sigma; largeur du paquet d'ondes (espace réel)
51   double x0; centre du paquet d'ondes
52   double psi_max; valeur maximale
53   double ymax;
54   double ymin;
55   para >> "sigma" >> sigma;
56   para >> "psi_max" >> psi_max;
57   para >> "x0" >> x0;
58   para >> "ymax" >> ymax;
59   para >> "ymin" >> ymin;
60
61   if(para=="deux_solitons") deux_solitons = true;
62   if(para=="lineaire") lineaire = true;
63
64   Matrix<double> evol(N);
65   psi.Alloc(N);
66   evol.Alloc(N);
67   collocation_periodic grille(N, 2*M_PI, 3, np); Grille de fonctions cardinales
68
69   soliton(grille.x, psi, x0, psi_max,sigma); construction du premier soliton
70   ajout du deuxième soliton
71   if(deux_solitons) soliton(grille.x,psi, x0+2.0, 0.5*psi_max,sigma*1.4142);
72
73    initialisation de la sortie des données
74 #ifdef GNUPLOT
75   int temps_mort;
```

```
76    para >> "temps_mort" >> temps_mort;
77    ofstream gout;
78    gnuplot_ctrl * GP;
79    GP = gnuplot_init() ;
80    gnuplot_cmd(GP,(char *)"set term x11");
81    gnuplot_cmd(GP,(char *)"set xr [0:2*pi]");
82    gnuplot_cmd(GP,(char *)"set yr [%g:%g]",ymin,ymax);
83  #endif
84
85    para.close();
86    int output_step=0; compteur des sorties sur fichier
87    for(double t=0.0; t<tmax; t+= h){
88      if(floor(t/dt_display) > output_step){
89        output_step++;
90  #ifdef GNUPLOT
91        gout.open("tmp.dat");
92        grille.plot(gout,psi.array());
93        gout.close();
94        gnuplot_cmd(GP,(char *)"set title 't = %g'",t);
95        gnuplot_cmd(GP,(char *)"plot 'tmp.dat' u 1:2 w l t ''");
96        usleep(temps_mort); temps mort entre les affichages, en microsecondes
97  #endif
98      }
99      évolution sur un intervalle de temps h (Crank−Nicholson)
100     Vector<double> xi(N);
101     evol.clear(); U = 0
102     evol += grille.D[0]; U = ∂_x
103     if(lineaire == false) evol.mult_diagonal(psi); U = u∂_x.
104     evol *= epsilon; U = εu∂_x
105     evol.mult_add(grille.D[2],mu); U = εu∂_x + μ∂_x^3
106     evol *= -h*0.5; U = −(1/2)dt(εu∂_x + μ∂_x^3)
107     evol.add_to_diagonal(1.0); U = 1 − (1/2)dt(εu∂_x + μ∂_x^3)
108
109     evol.mult_add(psi,xi); ξ = (1 + (1/2)dtL)ψ: évolution vers l'avant de t_n vers t_n + h/2
110     evol.add_to_diagonal(-2.0);
111     evol *= -1.0; maintenant U = (1 − (1/2)dtL)
112     evol.solve(psi,xi); trouve ψ tel que ξ = (1 − (1/2)dtL)ψ
113   }
114
115 #ifdef GNUPLOT
116   sleep(-1); temps mort après le dernier affichage, en secondes
117   gnuplot_close(GP) ;
118 #endif
119
120   cout << "Programme terminé normalement\n";
121 }
```

Les options d'entrée du programme principal sont les suivantes :

1. `N` : le nombre de points de collocation
2. `plot_points` : le nombre de points utilisés pour le graphique
3. `h` : le pas temporel
4. `temps_de_simulation` : le temps de simulation
5. `intervalle_affichage` : l'intervalle entre les affichages successifs
6. `mu` : le coefficient μ de l'équation (5.31)
7. `epsilon` : le coefficient ϵ de l'équation (5.31)
8. `sigma` : la largeur σ du soliton initial
9. `x0` : la position x_0 du soliton initial
10. `ymin,ymax` : le domaine d'affichage selon l'axe vertical
11. `temps_mort` : le temps d'attente entre deux images (graphiques)
12. `deux_solitons` : Si ce mot-clé apparaît, deux solitons sont mis en condition initiale. Le deuxième est à $x_0 + 2$, est deux fois plus petit et $\sqrt{2}$ fois plus large que le premier.
13. `lineaire` : Si ce mot-clé apparaît, le problème est modifier pour simuler une équation linéaire :

$$\frac{\partial \psi}{\partial t} + \epsilon \frac{\partial \psi}{\partial x} + \mu \frac{\partial^3 \psi}{\partial x^3} = 0 \qquad \text{(linéaire)} \tag{5.41}$$

Exercice 5.4: Résolution de l'équation de Korteweg de Vries

A Installez le programme `KdeV.cpp`. Vérifiez que l'équation d'advection, simulée en posant $\mu = 0$ et en spécifiant le mot-clé `linéaire` dans le fichier d'entrée, mène à des paquets d'onde qui se propagent sans dispersion.

B Posez maintenant $\mu = 0.001$, toujours dans l'équation linéaire, et observez la dispersion.

C Restaurez la non linéarité dans l'équation, c'est-à-dire retournez à l'équation de KdeV, et posez $\mu = 0$ (équation de Burgers). Le comportement observé est-il crédible ?

D Observez maintenant la propagation d'un soliton en posant $\mu = 0.001$, $\sigma = 0.25$ et la valeur appropriée de l'amplitude afin que l'expression (5.40) soit une solution exacte de l'équation. Que se passe-t-il si l'amplitude est 20% trop élevée ou trop basse par rapport à la bonne valeur ?

E Spécifiez maintenant deux solitons dans les conditions initiales, avec les paramètres originaux de la partie précédente. Qu'observez-vous ? Quelle différence y a-t-il dans le processus de collision des solitons et celui de la superposition de deux paquets d'ondes linéaires ?

F En-deça de quelle valeur de N (le nombre de points de collocation) la solution commence-t-elle à se détériorer de manière perceptible ? Qu'arrive-t-il éventuellement quand N est encore plus bas ?

Chapitre 6

Méthodes stochastiques

6.1 Nombres aléatoires

Les méthodes stochastiques reposent toutes sur la possibilité de générer des nombres de manière aléatoire, c'est-à-dire des séquences de nombres entiers pris au hasard dans un intervalle donné. Que veut-on dire par là précisément ? Premièrement, remarquons qu'un nombre n'est pas aléatoire en soi : le qualificatif s'applique à une suite infinie de nombres, chacun étant compris entre 0 et M (M peut être très grand, comme par exemple 2^{64}, ou simple $M = 2$ pour une production de bits aléatoires). Le caractère aléatoire est fondamentalement lié à l'impossibilité de prédire quel sera le nombre suivant de la suite, ou de déceler des corrélations significatives entre les membres différents de la suite. Par exemple, une suite de nombres aléatoires sera *incompressible*, au sens informatique du terme : aucun algorithme de compression ne pourrait y être appliqué avec un gain supérieur à un dans la limite d'une suite infinie.

La façon idéale de générer une séquence aélatoire est d'avoir recours à un processus physique fondamentalement aléatoire, gouverné par les lois de la mécanique quantique ou statistique. Des dispositifs qui produisent des bits aléatoires liés à des processus optiques, comme le passage ou non d'un photon au travers d'un miroir semi-transparent, existent sur le marché [1]. Cependant, ces générateurs physiques de séquences aléatoires peuvent être chers, et aussi trop lents pour les besoins du calcul. En fait, il est plus économique et simple de les remplacer par des générateurs de nombres *pseudo-aléatoires*.

Un générateur pseudo-aléatoire est une contradiction dans les termes : il s'agit d'une méthode *déterministe* pour générer une suite de nombres qui se comporte à toutes fins pratiques comme un suite réellement aléatoire. En particulier, chaque nombre de la suite est déterminé de manière unique par un ou quelques uns des nombres qui le précèdent dans la suite, mais la loi déterministe est «non naturelle» et n'a pratiquement aucune chance d'avoir un effet sur le calcul, si on la compare à un processus réellement aléatoire.

1. Voir par exemple http://www.idquantique.com/

Générateur à congruence linéaire

La manière classique de générer des entiers aléatoires est la relation de récurrence suivante, dite à *congruence linéaire* :

$$x_{n+1} = (ax_n + c) \mod M \tag{6.1}$$

où les entiers a et c doivent être choisis judicieusement, ainsi qu'un premier entier non nul dans la séquence. Par exemple, $a = 16807$, $c = 0$ et $M = 2^{31} - 1$ sont un choix courant. La séquence ainsi définie est périodique de période M : elle se répète exactement après M nombres, car le nombre suivant est déterminé par le nombre courant, et au plus M possibilités existent. Les générateurs simples de ce type ont longtemps été la règle. Ils sont à proscrire absolument dans toute application ou la qualité de la séquence aléatoire est importante. En particulier, sur une machine moderne, la séquence complète peut être générée en quelques secondes, ce qui démontre clairement son insuffisance pour les calculs sérieux.

Générateurs de Fibonacci

Cette méthode est basée sur la récurrence suivante :

$$x_{n+1} = x_{n-p} + x_{n-q} \mod M \tag{6.2}$$

On doit spécifier les entiers p, q et M et fournir les premiers éléments de la séquence par une autre méthode (par exemple une congruence linéaire). Parmi les choix acceptables de (p, q), signalons

$$(607, 273) \qquad (2281, 1252) \qquad (9689, 5502) \qquad (44497, 23463) \tag{6.3}$$

Des générateurs plus avancés encore sont obtenus en se basant sur la méthode de Fibonacci et en ajoutant un mélange des bits du nombre à chaque étape.

Générateur de base conseillé

Le code suivant montre comment utiliser deux générateurs de nombres aléatoires. Le premier (`Random`) est simplement un appel à GSL, alors que le deuxième (`RandomNR`) est proposé dans *Numerical Recipes*. La méthode par défaut utilisée par GSL est une variante de la méthode de Fibonacci appelée *Mersenne twister* et proposée par Makoto Matsumoto and Takuji Nishimura en 1997 [2].

Code 6.1 : Générateur de nombres aléatoire : `Random.h`

```
1 #ifndef RANDOM_H_
2 #define RANDOM_H_
3
4 #include <gsl/gsl_rng.h>
5
6 class Random{
7 public:
```

2. M. Matsumoto et T. Nishimura, ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**, 3–30 (1998).

```
8     Random(){ constructeur par défaut
9       gsl_rng_env_setup();
10      T = gsl_rng_default;
11      r = gsl_rng_alloc(T);
12    }
13
14    ~Random(){gsl_rng_free (r);}
15
16    inline double uniform(){ return gsl_rng_uniform(r);}
17
18    inline unsigned long int integer(){return gsl_rng_get(r);}
19
20  private:
21    const gsl_rng_type *T;
22    gsl_rng *r;
23  };
24
25  class RandomNR{
26  public:
27    unsigned long int u,v,w;
28
29    RandomNR(unsigned long int j) : v(4101842887655102017LL), w(1) {
30      u = j ^ v; int64();
31      v = u; int64();
32      w = v; int64();
33    }
34    inline unsigned long int int64() {
35      u = u * 2862933555777941757LL + 7046029254386353087LL;
36      v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
37      w = 4294957665U*(w & 0xffffffff) + (w >> 32);
38      unsigned long int x = u ^ (u << 21); x ^= x >> 35; x ^= x << 4;
39      return (x + v) ^ w;
40    }
41    inline double uniform() { return 5.42101086242752217E-20 * int64(); }
42    inline unsigned int int32() { return (unsigned int)int64(); }
43  };
44
45  #endif
```

6.1.1 Distribution uniforme

À partir d'un générateur de nombres aléatoires entiers, compris entre 0 et $M-1$, on peut générer des nombres aléatoires à virgule flottante compris entre 0 et 1 simplement en divisant par M (au sens des NVF). C'est ce qui est fait dans les routines `Random::uniform()` et `RandomNR::uniform()` listées ci-dessus. Ces nombres aléatoires suivent une distribution de probabilité uniforme : $p(x) = 1$ dans l'intervalle $x \in [0,1]$. Le fait que la distribution soit uniforme est précisément l'une des conditions qui caractérisent un bon générateur de nombres aléatoires.

Plusieurs tests statistiques de la qualité des générateurs pseudo-aléatoires peuvent être appliqués pour s'en assurer.

À partir d'une distribution uniforme dans l'intervalle $[0,1]$, on peut générer une distribution uniforme dans tout intervalle fini par transformation affine $x' = ax + b$. Plus important : on peut générer des distributions de probabilité plus complexes, par les méthodes expliquées ci-dessous.

6.1.2 Méthode de transformation

Une méthode simple pour simuler une variable aléatoire y qui suit une distribution de probabilité non-uniforme est d'appliquer une transformation $x \mapsto y = f(x)$ à la variable uniforme x. En général, si une variable aléatoire x suit une distribution $p_1(x)$, les aléatoires[3] situés dans l'intervalle $\mathrm{d}x$ autour de x correspondent alors à des valeurs aléatoires de y situéees dans un intervalle $\mathrm{d}y$ autour de y, distribués selon une fonction $p_2(y)$ telle que

$$p_1(x)|\mathrm{d}x| = p_2(y)|\mathrm{d}y| \quad \text{ou encore} \quad p_2(y) = p_1(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| \tag{6.4}$$

Par exemple, si $y = -\ln(x)$ et $p_1(x) = 1$ (distribution uniforme), alors

$$p_2(y) = \left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| = x = \mathrm{e}^{-y} \tag{6.5}$$

On parvient alors à la distribution exponentielle, définie dans l'intervalle $y \in [0,\infty]$.

Plus généralement, produire une distribution p_2 quelconque à partir d'une distribution uniforme requiert la solution analytique de l'équation différentielle suivante :

$$p_2(y)\mathrm{d}y = \mathrm{d}x \implies \int_0^y \mathrm{d}z\, p_2(z) = F_2(y) = x \tag{6.6}$$

(on a supposé $\mathrm{d}y/\mathrm{d}x > 0$), où $F_2(y)$ est l'intégrale de la distribution p_2. En pratique, tirer un aléatoire selon la distribution p_2 revient à tirer un aléatoire x selon une distribution uniforme, puis à effectuer la correspondance $y = F_2^{-1}(x)$ pour obtenir la valeur de y distribuée selon p_2. La méthode de transformation est utile et efficace uniquement si F_2 peut être calculé et inversé analytiquement.

6.1.3 Méthode du rejet

Une méthode plus générale pour générer une distribution $p(x)$ est la méthode du rejet, illustrée à la figure 6.1. Supposons que nous désirions générer un aléatoire x qui suive la distribution $p(x)$ (en rouge). Sur la figure le domaine de x est fini (de a à b), mais cela n'est pas nécessaire. Supposons en outre que nous connaissions une distribution $f(x)$ (en noir sur la figure) qui

3. Nous désignerons couramment une variable aléatoire par le nom *aléatoire*, correspondant à l'anglais *deviate*.

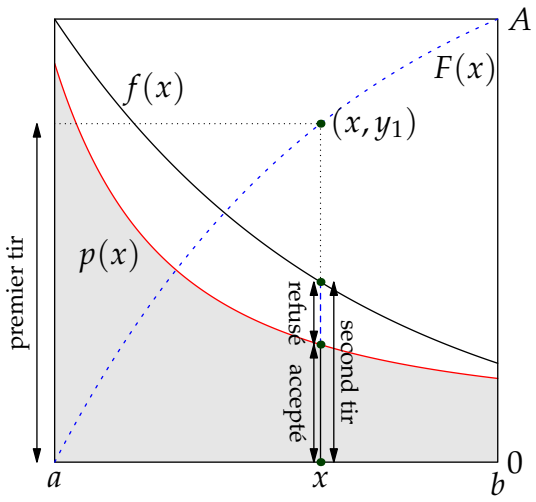


Figure 6.1 Méthode du rejet.

soit partout plus grande que $p(x)$, et que nous connaissions analytiquement l'intégrale $F(x)$ de cette distribution et sa réciproque F^{-1} :

$$F(x) \stackrel{\text{def}}{=} \int_0^x \mathrm{d}y\, f(y) \tag{6.7}$$

Cela suppose bien sûr que la distribution f n'est pas normalisée (sinon elle ne pourrait pas être partout plus grande que $p(x)$) et donc que $F(b) = A > 1$. On procède ensuite comme suit :

1. On tire une valeur au hasard $y_1 \in [0, A]$, distribuée uniformément.
2. On en déduit une valeur $x = F^{-1}(y_1)$. Jusqu'ici la procédure est la même que pour la méthode de transformation, sauf pour le fait que f n'est pas normalisée.
3. On tire un deuxième nombre y_2 uniformément distribué entre 0 et $f(x)$.
4. Si $y_2 < p(x)$, on accepte la valeur de x ainsi produite. Sinon, on la rejette et on recommence jusqu'à ce que la valeur de x soit acceptée.
5. L'ensemble des valeurs de x ainsi générées suit la distribution $p(x)$.

Plus la fonction $f(x)$ est proche de $p(x)$, plus la méthode est efficace, c'est-à-dire plus les rejets sont rares. Le cas particulier d'une distribution $f(x)$ uniforme peut toujours servir en pratique, mais risque d'être très inefficace si la fonction $p(x)$ est piquée autour d'une valeur en particulier.

6.1.4 Méthode du rapport des aléatoires uniformes

Une méthode puissante pour générer un grand nombre de distributions est la méthode du rapport des aléatoires uniformes (Kinderman et Monahan), que nous allons maintenant expliquer. La méthode du rejet peut se représenter ainsi dans le plan (x, p) :

$$p(x)\mathrm{d}x = \int_0^{p(x)} \mathrm{d}p'\, \mathrm{d}x \tag{6.8}$$

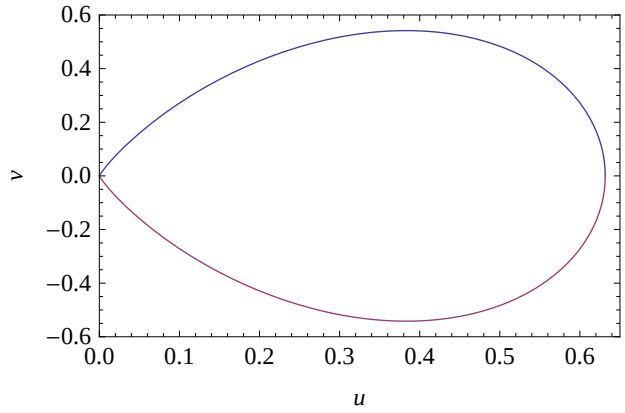


Figure 6.2 Domaine du plan (u,v) permettant d'échantillonner une distribution gaussienne pour le rapport $x = v/u$ (on a choisi $\sigma = 1$ et $\mu = 0$).

Ce qui revient à dire que si on échantillonne uniformément la région bornée par l'axe des x et par la courbe $p(x)$ dans le plan (x,p), on se trouve à échantillonner x selon la distribution $p(x)$. Cette affirmation, toute évidente qu'elle soit, repose sur le fait que l'intégrant de l'intégrale ci-dessus est l'unité.

Procédons maintenant à un changement de variables :

$$x = \frac{v}{u} \qquad\qquad p = u^2 \tag{6.9}$$

Le jacobien associé est une constante :

$$\frac{\partial(p,x)}{\partial(u,v)} = \begin{vmatrix} 2u & -v/u^2 \\ 0 & 1/u \end{vmatrix} = 2 \tag{6.10}$$

et donc on peut écrire, sur le plan (u,v),

$$p(x)\mathrm{d}x = 2\int_0^{\sqrt{p(x)}} \mathrm{d}u\,\mathrm{d}v = 2\int_0^{\sqrt{p(v/u)}} \mathrm{d}u\,\mathrm{d}v \tag{6.11}$$

Le domaine d'intégration est défini par la courbe représentant la distribution $p(x)$, d'une part, et par la droite $p = 0$, d'autre part, exprimées en fonction de u et v. Comme l'intégrant est encore une fois constant, un échantillonage uniforme à l'intérieur de cette courbe dans le plan (u,v) revient à échantilloner x selon la distribution $p(x)$, la valeur de x étant simplement donnée par le rapport v/u. Nous avons ainsi généré x par un rapport de variables aléatoires (u et v) distribuées uniformément, avec cependant des conditions de rejet.

Considérons par exemple la distribution gaussienne :

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp -\frac{(x-\mu)^2}{2\sigma^2} \tag{6.12}$$

où σ est l'écart-type et μ la moyenne. Posons $\mu = 0$, ce que nous pouvons faire sans perte de généralité. En fonction de u et v, l'équation qui définit la distribution devient

$$u^2 = \frac{1}{\sqrt{2\pi}\sigma} \exp -\frac{v^2}{2u^2\sigma^2} \quad \text{ou} \quad v = \pm u\sigma\sqrt{2\ln\left(\sqrt{2\pi}\sigma u^2\right)} \tag{6.13}$$

Cette équation est représentée par la courbe illustrée à la figure 6.2.

Pour échantillonner efficacement une distribution gaussienne, on doit alors échantilloner de manière uniforme u et v dans le domaine du graphique, rejeter les points qui tombent à l'extérieur de la courbe, et ensuite retourner la valeur $x = v/u$. Pour augmenter l'efficacité de la procédure, on définit des bornes externe et interne (angl. *squeeze*) à la courbe, bornes qui sont rapidement calculables en comparaison de la formule (6.13). On accepte alors les points à l'intérieur de la borne interne et on les rejette à l'extérieur de la borne externe. Ce n'est que dans les cas plus rares où le point tombe entre les deux bornes qu'on doit tester la frontière exacte (6.13). On peut ainsi proposer un code qui produit un aléatoire gaussien au coût moyen de 2.74 aléatoires uniformes.

6.2 Méthode de Monte-Carlo

L'épithète «Monte-Carlo» est appliquée généralement à des méthodes de calcul qui reposent sur l'échantillonnage aélatoire d'un ensemble très vaste. Elle fait bien sûr référence à la célèbre maison de jeu de Monaco, et a été proposée par J. von Neumann et Stanislaw Ulam lors du développement des premières applications de ce genre à Los Alamos dans les années 1950. [4] L'idée générale la suivante : plusieurs problèmes d'intérêt requièrent de sommer (ou de faire une moyenne) sur un très grand nombre d'états. Ceux-ci peuvent être les états possibles d'un gaz ou d'un liquide, d'un système magnétique, ou les trajectoires d'une particule dans un milieu. Le mot «état» est ici employé au sens de «configuration», mais peut désigner quelque chose d'aussi simple qu'un point dans un domaine d'intégration en plusieurs dimensions. Au lieu de sommer systématiquement sur tous les états, ce qui est pratiquement impossible en raison de leur nombre astronomique, on échantillonne ces états à l'aide de nombre aléatoires. La difficulté consiste généralement à (1) échantillonner de manière efficace et (2) bien estimer les erreurs statistiques ainsi commises.

6.2.1 Intégration par Monte-Carlo : exemple simple

Considérons, comme exemple simple, le problème du calcul de l'aire d'un cercle de rayon 1, soit π. Au lieu de faire l'intégrale analytiquement, évaluons-la en tirant deux aléatoires uniformes $x \in [-1,1]$ et $y \in [-1,1]$, et en rejetant le couple (x,y) s'il est en dehors du cercle, c'est-à-dire si la condition $x^2 + y^2 < 1$ n'est pas satisfaite. La valeur de π ainsi calculée, dans un exemple de simulation, est illustrée à la figure 6.3 en fonction du nombre de points.

Comment estimer l'erreur commise lors d'un tel calcul ? On a recourt au *théorème de la limite centrale*, qui stipule essentiellement ce qui suit : Si on somme N variables aléatoires X_i indépendantes qui suivent toutes la même distribution de probabilité avec variance $\operatorname{Var} X$, alors

4. L'oncle de Ulam était, semble-t-il, un joueur compulsif qui empruntait de l'argent pour jouer à la roulette à Monte-Carlo.

N	$\bar{X}$	σ	$\Delta X/\sigma$
10	2.4	0.52	1.4
10^2	3.12	0.16	0.13
10^3	3.176	0.052	−0.66
10^4	3.1148	0.016	1.63
10^5	3.1498	0.052	−1.6
10^6	3.1417	0.0016	−0.068
10^7	3.14177	0.00052	−0.35

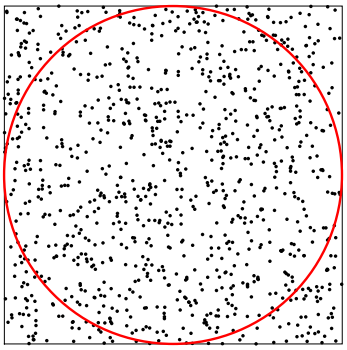


Figure 6.3 Calcul de l'aire d'un cercle par la méthode de Monte-Carlo. À gauche : valeur estimée de π en fonction du nombre de points, écart-type de l'estimation et erreur commise en rapport avec l'écart-type.

l'écart-type de l'espérance mathématique

$$\bar{X} \stackrel{\text{def}}{=} \frac{1}{N}\sum_i X_i \quad \text{est} \quad \sqrt{\frac{\operatorname{Var} X}{N}} \tag{6.14}$$

et, dans la limite où N est grand, cette espérance mathématique suit une loi normale dont la moyenne est $\mu = \langle X \rangle$.

Par exemple, si on tire N aléatoires uniformément distribués dans $[-1,1]$, la moyenne de chaque aléatoire est 0, et l'écart-type est $1/\sqrt{3}$. Chacun des tirs suit la même loi de probabilité, et donc la moyenne des résultats, $\bar{X}$, suivra une loi normale de moyenne 0 et de variance $1/3N$. En clair, cela signifie que la moyenne véritable de la somme, qu'on cherche à établir, ne sera pas donnée précisément par la moyenne mesurée, mais en différera d'une quantité inférieure à $1/\sqrt{3N}$ la plupart du temps. Plus précisément, en suivant la loi normale, on tombe à l'intérieur d'un écart-type 68% du temps, et à l'intérieur de deux écarts-types 95% du temps.

Dans le cas de l'aire du cercle, on définit une variable aléatoire à deux composantes (x,y) dans le carré circonscrit au cercle, et une autre aléatoire X qui prend deux valeurs, définie en fonction de (x,y) comme

$$X = \begin{cases} 1 & \text{si} \quad (x^2+y^2) < 1 \\ 0 & \text{sinon} \end{cases} \tag{6.15}$$

X suit une distribution binomiale avec $P(X=1) = p = \pi/4$, $P(X=0) = q = 1-p$. La valeur moyenne de X est $\langle X \rangle = p$ et sa variance est $\operatorname{Var} X = pq = \frac{\pi}{4}(1-\frac{\pi}{4})$. Le fait de tirer N points aléatoires indépendants dans le carré revient à mesurer l'espérance mathématique $\bar{X}$ définie en (6.14), dont l'écart-type est

$$\sigma = \sqrt{\frac{1}{N}\frac{\pi}{4}\left(1-\frac{\pi}{4}\right)} \tag{6.16}$$

6.2.2 Intégrales multi-dimensionnelles

Supposons qu'on doivent intégrer une fonction $f(x)$ de d variables, ou plutôt la moyenne de f dans son domaine de définition :

$$\langle f \rangle = \frac{\int_\Omega \mathrm{d}^d x \, f(x)}{\int_\Omega \mathrm{d}^d x} \tag{6.17}$$

Le principe de l'intégration Monte-Carlo est d'échantilloner l'espace Ω avec N points x_i et d'estimer la moyenne ainsi :

$$\langle f \rangle \approx \frac{1}{N} \sum_i f(x_i) \tag{6.18}$$

On commet alors une erreur

$$\Delta = \sqrt{\frac{\mathrm{Var}\, f}{N}} \qquad \mathrm{Var}\, f \stackrel{\mathrm{def}}{=} \langle f^2 \rangle - \langle f \rangle^2 \tag{6.19}$$

Si, au contraire, on intègre la même fonction en utilisant une grille fixe de points, avec par exemple la méthode de Simpson, l'erreur commise sera de l'ordre de $1/n^4$, où n est le nombre de points par direction, menant à un nombre total de points $N = n^d$ (on suppose un nombre de points égal dans chaque direction de l'espace). L'erreur commise dans la méthode de Simpson est donc

$$\Delta_{\mathrm{Simpson}} \sim \frac{1}{N^{4/d}} \tag{6.20}$$

On voit que la méthode Monte-Carlo converge plus rapidement que la méthode de Simpson pour des dimensions $d > 8$. Autrement dit, pour les intégrales dans des domaines de grande dimension, le Monte-Carlo est la meilleure méthode ! Le fait de remplacer la méthode de Simpson par une autre méthode convergeant plus rapidement ne change pas fondamentalement ce résultat, mais ne fait que repousser son impact à des dimensions plus grandes. Or, en mécanique statistique classique, on doit calculer les valeurs moyennes en intégrant sur l'espace des phases Ω de M particules. En trois dimensions d'espace, la dimension de l'espace Ω est $d = 6M$ (ou $d = 4M$ en deux dimensions d'espace). Donc quelques particules seulement suffisent à rendre l'intégration Monte-Carlo incontournable.

Dans plusieurs modèles de physique statistique, les configurations du système sont discrètes et non continues. Par exemple, le modèle d'Ising défini sur M sites comporte 2^M états possibles, un nombre fini, mais qui croît exponentiellement avec M. Là aussi l'échantillonnage Monte-Carlo est la méthode de choix, car la somme exacte sur toutes les configurations est impossible à réaliser si M est grand, alors que l'erreur Monte-Carlo, elle, est contrôlable.

6.2.3 L'algorithme de Metropolis

Soit Ω l'espace sur lequel vit une variable aléatoire multidimensionnelle x suivant une distribution $\mu(x)$. Le problème est de calculer la valeur moyenne d'une fonction $f(x)$ dans cet espace :

$$\langle f \rangle = \frac{\int_\Omega \mathrm{d}x \, \mu(x) f(x)}{\int_\Omega \mathrm{d}x \, \mu(x)} \tag{6.21}$$

En mécanique statistique, x représente un état dynamique du système (un point dans l'espace des phases, par exemple) ; $f(x)$ représente une quantité physique (une fonction dans l'espace des états), comme l'énergie ou l'aimantation. Enfin la distribution $\mu(x)$ est la probabilité relative d'occupation de l'état x, comme par exemple la loi de Boltzmann $\mu(x) = \exp -E(x)/T$, T étant la température absolue en unités de l'énergie E.

Pour calculer la moyenne (6.21), on doit en pratique générer un ensemble de valeurs aléatoires x_i, ce qui n'est pas toujours facile et économique dans un espace de grande dimension. En particulier, la méthode du rejet simple peut dans ces cas devenir très inefficace. L'*algorithme de Metropolis* vise à échantilloner Ω non pas en produisant des valeurs successives qui sont absolument indépendantes, mais plutôt à produire une «marche aléatoire» dans Ω, qui puisse traverser Ω en passant plus de temps dans les régions où $\mu(x)$ est plus grand, c'est-à-dire en générant une séquence de points x_i qui finit par être distribuée selon $\mu(x)$ lorsqu'elle est suffisamment longue.

L'algorithme vise en fait à produire une *chaîne de Markov*, c'est-à-dire une successsion de valeurs aléatoires dont chacune provient de la précédente selon une loi de probabilité bien précise :

$$x_1 \to x_2 \to x_3 \to \cdots \to x_i \to x_{i+1} \to \cdots \tag{6.22}$$

Le passage d'une valeur x à un valeur y à chaque étape de la chaîne est effectué avec une probabilité $W(x \to y)$, appelée *probabilité de transition*. Dans tout calcul numérique, le nombre de valeurs possibles de x est fini (disons M), et chaque valeur possible x_α de x pourrait alors être étiquetée à l'aide d'un indice α variant de 1 à M. Dans ce cas, les probabilités de transition W forment une matrice finie d'ordre M notée $W_{\alpha\beta} = W(x_\beta \to x_\alpha)$ (noter l'ordre inverse des indices). Un vecteur $X = (x_\alpha)$ de dimension M représente alors une certaine distribution de la variable aléatoire x, qui n'est pas nécessairement la distribution recherchée $(\mu_\alpha) = (\mu(x_\alpha))$. Par exemple, le vecteur X pourrait n'avoir qu'une composante non nulle au départ. Par contre, en appliquant de manière répétée la matrice W sur ce vecteur, la distribution évolue, et peut tendre vers une limite stable, ce qui correpond à un vecteur propre de W de valeur propre unité :

$$WX = X \tag{6.23}$$

Le vecteur propre X correspond alors à une distribution de probabilité qui n'est pas affectée par la marche, idéalement celle qu'on désirait générer au départ : $X = (\mu_\alpha)$.

Pour ce faire, les probabilités de transition $W(x \to y)$ (alias $W_{\alpha\beta}$) doivent respecter les contraintes suivantes :

1. Normalisation : $\int_y W(x \to y) = 1$, ou $\sum_\alpha W_{\alpha\beta} = 1$, en langage discret. Cette propriété est nécessaire afin que W soit effectivement une matrice de probabilités. Elle impose M contraintes sur les M^2 composantes de W.
2. Ergodicité : toute valeur de y doit être éventuellement accessible à partir de toute valeur de x si on applique la matrice W un nombre suffisant de fois : il existe un entier n tel que $W^n(x \to y) \neq 0$, ou encore $(W^n)_{\alpha\beta} \neq 0 \ \forall \alpha, \beta$. Cette propriété nous assure que toutes les régions de l'espace Ω seront visitées, quel que soit le point de départ de la chaîne.

3. Le bilan détaillé :

$$\mu(x)W(x \to y) = \mu(y)W(y \to x) \quad \text{ou encore} \quad \mu_\alpha W_{\beta\alpha} = \mu_\beta W_{\alpha\beta} \tag{6.24}$$

où on ne somme pas sur l'indice répété. Cette relation impose $\frac{1}{2}M(M-1)$ contraintes à la matrice W. Elle suffit cependant à s'assurer que le vecteur μ_α est un vecteur propre de W de valeur propre unité. En effet,

$$\sum_\beta W_{\alpha\beta}\mu_\beta = \sum_\beta \frac{\mu_\alpha}{\mu_\beta} W_{\beta\alpha}\mu_\beta = \sum_\beta W_{\beta\alpha}\mu_\alpha = \mu_\alpha \tag{6.25}$$

où on a utilisé la condition de normalisation et la condition du bilan détaillé. Ces deux conditions ensemble ne fixent pas W de manière unique, mais seulement $\frac{1}{2}M(M+1)$ composantes.

L'algorithme de Metropolis utilise la forme suivante de la matrice W :

$$W(x \to y) = \min\left(1, \frac{\mu(y)}{\mu(x)}\right) \quad \text{ou encore} \quad W_{\alpha\beta} = \min\left(1, \frac{\mu_\alpha}{\mu_\beta}\right) \tag{6.26}$$

ce qui peut également se représenter par le tableau suivant :

	$W(x \to y)$	$W(y \to x)$
$\mu(x) > \mu(y)$	$\mu(y)/\mu(x)$	1
$\mu(x) < \mu(y)$	1	$\mu(x)/\mu(y)$

On constate que cette prescription respecte la condition du bilan détaillé. Par contre, elle ne respecte pas la condition de normalisation, mais il suffit de multiplier W par une constante de normalisation inconnue Z^{-1} pour régler ce problème. L'avantage de l'algorithme de Metropolis est que la connaissance préalable de cette constante de normalisation n'est pas requise ; seules les probabilités relatives $\mu(x)/\mu(y)$ le sont.

6.3 Analyse d'erreur

L'aspect le plus subtil de la méthode de Monte-Carlo est l'estimation des résultats et de leur incertitude à partir des données statistiques recueillies. Considérons un ensemble de valeurs mesurées A_i $(i = 1, \ldots, N)$ d'une observable A, lors d'un processus markovien, par exemple basé sur l'algorithme de Metropolis. On suppose que la variable aléatoire A possède une moyenne exacte $\langle A \rangle$ et une variance exacte $\operatorname{Var} A$ définie par

$$\operatorname{Var} A = \langle A^2 \rangle - \langle A \rangle^2 \tag{6.27}$$

L'objectif principale de la simulation Monte-Carlo est d'obtenir une estimation de la valeur moyenne $\langle A \rangle$, ainsi que de l'incertitude Δ_A sur cette valeur.

6.3.1 Théorème de la limite centrale

Il est assez évident que la meilleure estimation de la moyenne $\langle A \rangle$ est précisément la moyenne statistique des valeurs mesurées :

$$\bar{A} = \frac{1}{N} \sum_i A_i \tag{6.28}$$

Il est cependant intéressant de le démontrer, en considérant l'expression ci-haut de $\bar{A}$ comme une variable aléatoire dont on doit calculer la valeur moyenne :

$$\langle \bar{A} \rangle = \frac{1}{N} \sum_i \langle A_i \rangle = \frac{1}{N} \sum_i \langle A \rangle = \langle A \rangle \tag{6.29}$$

où on a supposé que chaque mesure A_i obéissait à la même loi de probabilité qui régit la variable A, de sorte que $\langle A_i \rangle = \langle A \rangle$.

L'estimation de l'erreur s'obtient ensuite en calculant la variance de $\bar{A}$:

$$\operatorname{Var} \bar{A} = \langle \bar{A}^2 \rangle - \langle \bar{A} \rangle^2 = \langle \bar{A}^2 \rangle - \langle A \rangle^2 \tag{6.30}$$

Or

$$\langle \bar{A}^2 \rangle = \frac{1}{N^2} \sum_{i,j} \langle A_i A_j \rangle = \frac{1}{N^2} \sum_i \langle A_i^2 \rangle + \frac{1}{N^2} \sum_{i,j \ (i \neq j)} \langle A_i A_j \rangle \tag{6.31}$$

Dans l'hypothèse où les mesures successives de A ne sont pas corrélées, alors $\langle A_i A_j \rangle = \langle A_i \rangle \langle A_j \rangle = \langle A \rangle^2$ si $i \neq j$ (il y a $N(N-1)$ termes comme celui-là). On trouve alors

$$\langle \bar{A}^2 \rangle = \frac{1}{N} \langle A^2 \rangle + \frac{N-1}{N} \langle A \rangle^2 \tag{6.32}$$

Au total, la variance de $\bar{A}$ est

$$\operatorname{Var} \bar{A} = \frac{1}{N} \langle A^2 \rangle - \frac{1}{N} \langle A \rangle^2 = \frac{1}{N} \operatorname{Var} A \tag{6.33}$$

On trouve donc le résultat du théorème de la limite centrale, que nous venons en fait de démontrer : l'erreur Δ_A commise sur l'estimation de $\langle A \rangle$ est l'écart-type $\sqrt{\operatorname{Var} A}$, divisé par $\sqrt{N}$.

Exercice 6.1

Montrez que la variance $\operatorname{Var} A$ peut être estimée comme suit à partir des valeurs mesurées :

$$\operatorname{Var} A = \frac{N}{N-1} \left(\overline{A^2} - \bar{A}^2 \right) \quad \text{où} \quad \overline{A^2} \overset{\text{def}}{=} \frac{1}{N} \sum_i A_i^2 \tag{6.34}$$

6.3.2 Analyse logarithmique des corrélations

Le problème avec l'analyse ci-dessus est que les mesures successives A_i dans un processus markovien non idéal sont corrélées. En pratique, les mesures successives sont corrélées sur une certain «temps» caractéristique noté τ_A et appelé *temps d'autocorrélation* :

$$\tau_A \overset{\text{def}}{=} \frac{1}{\operatorname{Var} A} \sum_{t=1}^{\infty} (\langle A_{1+t} A_1 \rangle - \langle A \rangle^2) \tag{6.35}$$

Nous employons le mot «temps» dans le sens markovien, c'est-à-dire désignant la position dans la chaîne de Markov. On peut supposer généralement que la corrélation chute exponentiellement avec le temps, de sorte que seuls quelques termes de la somme sur t contribuent de manière appréciable, et donc la borne supérieure infinie de la somme sur les temps n'est pas réellement importante.

Retournons à l'éq. (6.31), sans toutefois supposer que les mesures sont non corrélées :

$$\langle \bar{A}^2 \rangle = \frac{1}{N} \langle A^2 \rangle + \frac{N-1}{N} \langle A \rangle^2 + \frac{1}{N^2} \sum_{i,j \ (i \neq j)} \left(\langle A_i A_j \rangle - \langle A \rangle^2 \right) \tag{6.36}$$

et donc

$$\begin{aligned} \operatorname{Var} \bar{A} &= \frac{1}{N} \operatorname{Var} A + \frac{1}{N^2} \sum_{i,j \ (i \neq j)} \left(\langle A_i A_j \rangle - \langle A \rangle^2 \right) \\ &= \frac{1}{N} \operatorname{Var} A + \frac{2}{N^2} \sum_{i}^{N} \sum_{t=1}^{N-i} \left(\langle A_i A_{i+t} \rangle - \langle A \rangle^2 \right) \\ &\approx \frac{1}{N} \operatorname{Var} A + \frac{2}{N} \sum_{t=1}^{\infty} \left(\langle A_1 A_{1+t} \rangle - \langle A \rangle^2 \right) \\ &= \frac{1}{N} \operatorname{Var} A (1 + 2\tau_A) \end{aligned} \tag{6.37}$$

Nous avons donc une estimation corrigée de l'erreur commise sur l'estimation de la valeur moyenne d'une observable. Il nous faut cependant une méthode efficace pour calculer le temps d'autocorrélation τ_A, car la définition directe de cette quantité ne se prête pas à un calcul pratique.

À cette fin, nous allons procéder à une *analyse logarithmique* des erreurs (angl. *binning*), dont le principe est illustré à la figure 6.4. Au fur et à mesure que les mesures A_i sont prises, on construit des séries accessoires de moyennes binaires

$$A^{(l)} = \tfrac{1}{2} \left(A_{2i-1}^{(l-1)} + A_{2i}^{(l-1)} \right) \tag{6.38}$$

où l est l'indice du niveau de la série accessoire. Chaque série accessoire contient la moitié du nombre de termes de la série précédente, jusqu'à n'en contenir qu'un seule (voir la figure). Si on procède à 2^M mesures, alors $M+1$ séries sont ainsi formée, y compris la série principale ($l = 0$). Pour chaque série on peut calculer l'estimation de la moyenne et de la variance :

$$\operatorname{Var} A^{(l)} = \frac{1}{N_l (N_l - 1)} \sum_{i=1}^{N_l} \left(A_i^{(l)} - \overline{A^{(l)}} \right)^2 \tag{6.39}$$

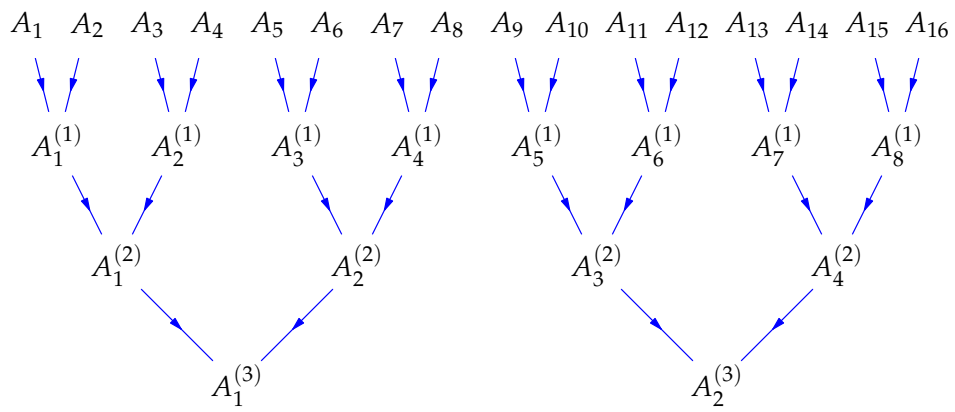


Figure 6.4 Schéma de l'analyse logarithmique d'erreurs.

où $N_l = 2^{M-l}$ est le nombre de terme dans la série de niveau l. Les erreurs associées à chaque série, $\Delta_A^{(l)} = \sqrt{\mathrm{Var}\, A^{(l)}}$, convergent vers l'erreur véritable dans la limite $l \to \infty$, car les termes consécutifs des séries accessoires deviennent non corrélés dans cette limite. En pratique, on ne construit pas toutes les séries accessoires jusqu'au niveau maximum possible, car l'estimation de l'erreur requiert un nombre M_l de données qui n'est pas trop petit ; par exemple on peut arrêter de construire les séries accessoires en-deça de $M_l = 2^5 = 32$. On doit être en mesure d'observer que l'erreur estimée $\Delta_A^{(l)}$ sature en fonction de l, et on peut adopter cette valeur limite comme estimation de l'erreur véritable Δ_A, et en même temps estimer le temps d'autocorrélation en fonction de cette valeur :

$$\tau_A = \frac{1}{2}\left[\left(\frac{\Delta_A}{\Delta_A^{(0)}}\right)^2 - 1\right] \tag{6.40}$$

En pratique, cette méthode ne requiert pas de stocker toutes les séries accessoires, ni la série principale. Le nombre de termes serait beaucoup trop grand. Comme on ne requiert que le calcul des valeurs moyennes et des variances, il suffit de garder en mémoire la somme courante des valeurs et la somme courante des valeurs au carré, pour chaque niveau l :

$$\Sigma^{(l)} = \sum_i A_i^{(l)} \qquad T^{(l)} = \sum_i \left(A_i^{(l)}\right)^2 \tag{6.41}$$

1. À chaque fois qu'une mesure est effectuée au niveau 0, on met à jour $\Sigma^{(0)}$ et $T^{(0)}$.
2. Si le nombre de mesures courant i est impair, on garde en mémoire A_i.
3. S'il est pair, alors on forme la valeur courante de $A^{(1)}$ selon l'équation (6.38) et on reprend à l'étape 1 ci-dessus, cette fois au niveau $l = 1$, et ainsi de suite.

Cette méthode est mise en oeuvre dans les routines `collecte` et `calcul_erreur` de la classe `observable` listée plus bas.

6.4 Modèle d'Ising

6.4.1 Définition

Wilhelm Lenz a proposé en 1920 un modèle simple pour décrire le changement de phase magnétique dans un ferroaimant. Ce modèle fut étudié par son étudiant Ernst Ising qui l'a solutionné en 1925 dans le cas d'une dimension d'espace, et qui lui a laissé son nom. On suppose dans ce modèle que les atomes du matériau sont disposés régulièrement, par exemple sur un réseau carré ou cubique. Chaque atome porte un spin, et un moment magnétique associé, représenté par une variable s_i pouvant prendre deux valeurs : $+1$ et -1 (i étant l'indice de l'atome, ou du site cristallin). On peut se figurer que ces deux valeurs représentent deux orientations opposées de l'aimantation (ou du spin) de chaque atome. Le modèle est ensuite défini par l'expression de l'énergie totale du système :

$$H[s] = -J \sum_{\langle ij \rangle} s_i s_j \tag{6.42}$$

où la somme est effectuée sur les paires de sites qui sont des voisins immédiats sur le réseau, ce qu'on appelle couramment les *premiers voisins* (ces paires sont notées $\langle ij \rangle$). Deux spins voisins contribuent une énergie $-J$ s'ils sont parallèles, et $+J$ s'ils sont antiparallèles. La tendance énergétique est donc de favoriser les configurations ferromagnétiques (spins alignés), mais celles-ci sont rares en comparaison des configurations d'aimantation nulle : l'entropie favorise l'état paramagnétique.

Exercice 6.2: Dénombrement des configurations

A Pour un système comportant N sites, combien y a-t-il de configurations d'aimantation M ?

B Tracez le logarithme du nombre de configurations d'aimantation M en fonction de M, pour $N = 36$.

Un système comportant N sites supporte donc 2^N configurations de spins différentes (notées s). Selon la mécanique statistique, les quantités observables (énergie ou aimantation) sont obtenues en moyennant leur valeur sur toutes les configurations, chacune étant prise avec un poids

$$W[s] = \frac{1}{Z} \exp -\frac{H[s]}{T} = \frac{1}{Z} \exp -\beta H[s] \qquad \beta \stackrel{\text{def}}{=} \frac{1}{T} \quad , \quad Z \stackrel{\text{def}}{=} \sum_s \mathrm{e}^{-\beta H[s]} \tag{6.44}$$

où T est la température absolue, dans des unités telles que $k_B = 1$. L'énergie E et l'aimantation M moyennes sont alors données par

$$\begin{aligned} \langle E \rangle &= \frac{1}{Z} \sum_s H[s] \mathrm{e}^{-\beta H[s]} \\ \langle M \rangle &= \frac{1}{Z} \sum_s M[s] \mathrm{e}^{-\beta H[s]} \quad \text{où} \quad M[s] = \sum_i s_i \end{aligned} \tag{6.45}$$

L'objectif de Lenz et d'Ising était de voir si un modèle aussi simple que celui-ci pouvait expliquer l'existence d'un changement de phase entre un état ferromagnétique à basse température ($T < T_c$), dans lequel $\langle M \rangle \neq 0$, et un état paramagnétique $\langle M \rangle = 0$ à haute température ($T > T_c$). La température critique T_c correspondrait alors à la température de Curie d'un matériau ferromagnétique.

Ising réussit à résoudre exactement ce modèle en dimension 1 seulement, et observa qu'il n'y avait pas de changement de phase à température non nulle, autrement dit que la température de Curie T_c était nulle. [5] Par contre, en 1944, Lars Onsager réussit à résoudre analytiquement le modèle d'Ising en deux dimensions d'espace, et trouva une température critique non nulle, donnée par

$$\frac{T_c}{J} = \frac{2}{\ln(1+\sqrt{2})} = 2.2691853142130221\ldots \tag{6.46}$$

Ce résultat confirma la pertinence du modèle et stimula l'étude de modèles plus réalistes.

Un modèle plus réaliste du magnétisme doit tenir compte plus exactement de la nature vectorielle et quantique du spin. Par exemple, le modèle de Heisenberg pour le magnétisme est défini à l'aide des opérateurs du spin :

$$H = -J \sum_{\langle ij \rangle} \mathbf{S}_i \cdot \mathbf{S}_j \tag{6.47}$$

Ce modèle peut être traité de manière classique (les vecteurs $\mathbf{S}_i$ sont alors de norme constante) ou de manière quantique (les vecteurs $\mathbf{S}_i$ sont alors des opérateurs quantiques). Le traitement quantique est bien sûr plus réaliste. Le couplage spin-spin du modèle de Heisenberg est isotrope, c'est-à-dire invariant par rotation. Par contre, dans certains matériaux, l'interaction spin-orbite peut mener à un couplage spin-spin qui n'est pas invariant par rotation dans l'espace des spins :

$$H = -J_{xy} \sum_{\langle ij \rangle} \left(S_i^x \cdot S_j^x + S_i^y \cdot S_j^y \right) - -J_z \sum_{\langle ij \rangle} S_i^z \cdot S_j^z \tag{6.48}$$

Dans les cas où $J_z > J_{xy}$, on peut montrer que le modèle se comporte effectivement comme un modèle d'Ising proche de la transition ferromagnétique (les composantes S_i^z ne pouvant prendre que deux valeurs opposées $\pm\frac{1}{2}$). Tout cela pour dire que le modèle d'Ising, malgré sa simplicité extrême, n'est pas tout à fait irréaliste.

6.4.2 Simulation du modèle d'Ising avec l'algorithme de Metropolis

Le modèle d'Ising est véritablement le plus simple qu'on puisse imaginer : il comporte un nombre fini de configurations pour un nombre fini N de sites (ou degrés de liberté), contrairement à un modèle de gaz basé sur les positions et vitesses continues des particules. Et pourtant le calcul direct des valeurs moyennes (6.45) est peu pratique, le nombre de configuration étant trop élevé (2^N) pour toute valeur intéressante de N. En fait, à basse température, la très vaste

5. Ising quitta ensuite la recherche académique et fit une carrière de professeur dans un *High School* américain, alors que son nom fut ensuite associé au modèle le plus célèbre de la mécanique statistique.

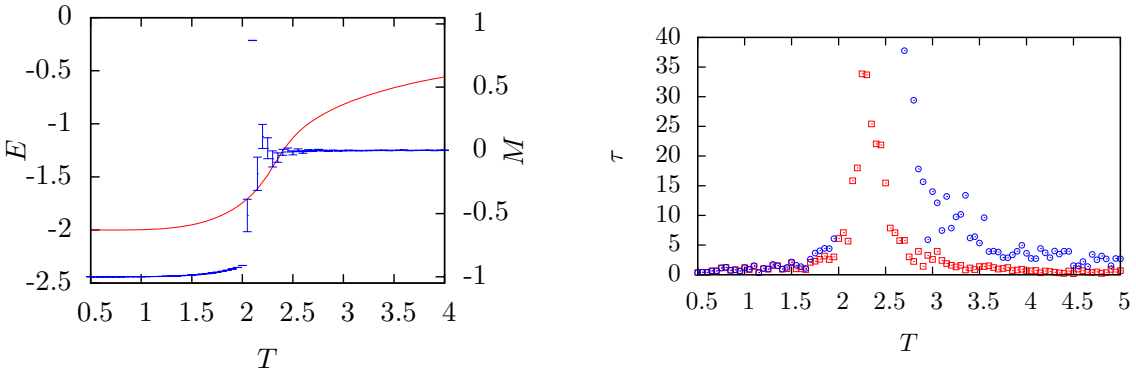


Figure 6.5 À gauche : valeurs moyennes de l'énergie E (rouge) et de l'aimantation M (bleu) en fonction de la température dans le modèle d'Ising 2D, sur un amas 16×16. Les barres d'erreur sont affichées pour M. À droite : temps d'autocorrélation pour les mêmes quantités, avec le même code de couleurs. Le temps τ_M pour l'aimantation dépasse les bornes acceptables proche de la transition ; les valeurs de M ne sont donc pas fiables dans cette région.

majorité des configurations ont une énergie trop élevée pour contribuer de manière significative aux valeurs moyennes. En revanche, à très haute température, les configurations sont toutes à peu près équivalentes, et il n'est pas plus sage alors de les compter toutes.

La solution est de mettre sur pied un processus markovien, une marche aléatoire dans l'espace des configurations. Cette marche doit nous faire résider sur une configuration donnée avec un probabilité proportionnelle à l'expression (6.44). Les étapes principales sont les suivantes :

1. Choisir une configuration initiale des spins. On peut la choisir de manière aléatoire, ce qui est alors typique d'une température élevée et demandera un plus long temps d'équilibration (ou thermalisation) si $T < T_c$.
2. À l'intérieur d'une boucle :
   (a) Effectuer une mise à jour de la configuration, en choisissant un site au hasard et en renversant le spin sur ce site.
   (b) Calculer la différence d'énergie ΔE entre cette configuration et la précédente.
   (c) Si $\Delta E < 0$ (énergie plus basse), accepter le changement.
   (d) Si $\Delta E > 0$ (énergie plus élevée), accepter ce changement avec probabilité $\mathrm{e}^{-\Delta E/T}$, sinon conserver l'ancienne configuration.
   (e) Mesurer les observables et collecter les statistiques périodiquement (c'est-à-dire à toutes les R mises à jour), pourvu que le nombre de mises à jour déjà effectuées soit suffisant pour que le processus markovien ait convergé vers la distribution de Boltzmann – ce qu'on appelle le temps de thermalisation.
3. Arrêter lorsque L niveaux de séries accessoires ont été complétés. Calculer l'erreur et le temps d'autocorrélation τ_A pour chaque observable. S'assurer que τ_A est considérablement plus petit que 2^L (le nombre de mesures effectuées au niveau 0 pour une valeur donnée au niveau L).

On reconnaît dans l'étape 2 ci-dessus l'algorithme de Metropolis.

Les résultats d'une simulation typique réalisée avec le code présenté ci-après sont illustrés à la figure 6.5.

6.4.3 Changements de phase

L'algorithme de Metropolis fonctionne généralement bien, mais des problèmes peuvent survenir à proximité des changements de phase. Malheureusement, ce sont généralement les changements de phase qui sont intéressants !

Longueur de corrélation

L'un des concepts fondamentaux de la physique statistique est celui de *longueur de corrélation*. Il s'agit, *grosso modo*, de la distance caractéristique ξ en-deça de laquelle les spins de sites différents sont corrélés. On définit la *fonction de corrélation* χ_{ij} entre les sites i et j comme suit :

$$\chi_{ij} = \langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle \tag{6.49}$$

Cette fonction est nulle lorsque les sites i et j sont suffisamment éloignés l'un de l'autre, mais décroît généralement de manière exponentielle en fonction de la distance :

$$\chi_{ij} \propto \exp - \frac{|\mathbf{r}_i - \mathbf{r}_j|}{\xi} \tag{6.50}$$

$\mathbf{r}_i$ étant la position du site no i. Cette forme exponentielle définit plus rigoureusement la longueur de corrélation ξ.

Types de changements de phase

On distingue deux types généraux de changements de phase :

1. Les changements de phase *continus*, souvent appelés *changements de phase du deuxième ordre*. Ceux-ci sont caractérisés par une divergence de la longueur de corrélation ξ à l'approche de la transition. Les différentes quantités thermodynamiques, comme l'énergie ou l'aimantation, sont cependant continues à la transition. La transition magnétique du modèle d'Ising est de ce type.
2. Les changements de phase du *premier ordre*. Ceux-là séparent deux phases dont les énergies libres sont identiques exactement à la transition ; mais comme dans ce cas la longueur de corrélation ne diverge pas, différentes régions de l'espace peuvent se retrouver dans des phases différentes : on dit qu'il y a coexistence de phase à la transition. Sous certaines conditions, il peut aussi y avoir hystérésis : une phase de haute température peut se prolonger en-deça de la température de transition pendant un certain temps et se trouver dans un état métastable ; à l'inverse, une phase de basse température peut se retrouver en équilibre métastable au-delà de la température de transition. L'exemple type d'une telle transition est l'ébullition de l'eau : les phase liquide et gazeuse sont en coexistence à la température de transition.

Ralentissement critique

En principe, un véritable changement de phase ne peut exister que dans la limite thermodynamique, c'est-à-dire la limite de taille infinie du système. En pratique, il suffit que le système soit suffisamment grand pour que toutes les caractéristiques d'un véritable changement de phase se manifestent, physiquement ou numériquement. L'un des phénomènes physiques qui se manifeste à proximité d'un changement de phase continu est le *ralentissement critique*, c'est-à-dire des fluctuations de plus en plus lentes qui se manifestent par un temps d'autocorrélation qui diverge. Dans le modèle d'Ising, on observe en fait que, à la transition elle-même ($T = T_c$), le temps d'autocorrélation croît comme le carré de la taille L du système :

$$\tau \propto L^2 \tag{6.51}$$

On observe aussi que, lors d'une transition du premier ordre qu'on franchit de manière progressive dans une simulation, le système reste dans la phase métastable, et le temps nécessaire pour plonger dans la phase de plus basse énergie se comporte comme

$$\tau \sim \exp L^{d-1} \tag{6.52}$$

où L est la taille du système et d la dimension de l'espace.

6.4.4 Code

Code 6.2 : Classe d'observables statistiques : observable.h

```
#ifndef OBSERVABLE_H_
#define OBSERVABLE_H_

#include <iostream>
#include <cassert>
#include <cmath>
#include <vector>

using namespace std;

#define LMAX 256
#define MAXFLOOR 32
#define MINFLOOR 8
#define FLOOR_OFFSET 5

extern bool observable_verbose; vrai pour imprimer plus de détails sur l'analyse d'erreur

class observable{
public:
  vector<double> sum; sommes des moyennes partielles
  vector<double> sum2; sommes des moyennes partielles au carré
  vector<double> prec; valeurs précédentes de la moyenne partielle
  vector<double> delta; variance partielle
```

```
24    vector<int> n; nombre de moyennes partielles
25    double tau; temps d'autocorrélation
26
27    double value; valeur de l'observable pour la configuration courante
28    double moyenne; moyenne estimée de l'observable
29    double erreur; erreur estimée de l'observable
30
31     Constructeur
32    observable() {
33      sum.reserve(MAXFLOOR);
34      sum2.reserve(MAXFLOOR);
35      prec.reserve(MAXFLOOR);
36      delta.reserve(MAXFLOOR);
37      n.reserve(MAXFLOOR);
38    }
39
40     Accumule les statistiques de l'observable par l'analyse logarithmique d'erreur (binning)
41    void collecte(unsigned int &level, double x){
42      if(level < sum.size()){
43        sum[level] += x;
44        sum2[level] += x*x;
45        n[level]++;
46      }
47      else{
48        sum.push_back(x);
49        sum2.push_back(x*x);
50        n.push_back(1);
51      }
52      if(n[level]%2 == 0){
53        double ave = 0.5*(prec[level]+x);
54        collecte(++level,ave);
55      }
56      else prec[level] = x;
57    }
58
59    Calcule la moyenne et l'erreur sur l'observable
60    void calcul_erreur(){
61      delta.clear();
62      int nn = sum.size()-FLOOR_OFFSET;
63      for(int i=0; i<nn; i++){
64        double tmp = sum2[i] - sum[i]*(sum[i]/n[i]);
65        tmp /= n[i];
66        tmp /= (n[i]-1);
67        delta.push_back(sqrt(tmp));
68      }
69      moyenne = sum[sum.size()-1]/n[sum.size()-1];
70      int j = sum.size()-FLOOR_OFFSET-1;
71      erreur = delta[j];
72      tau = delta[j]/delta[0];
```

```
73      tau -= 1.0;
74      tau *= 0.5;
75
76      if(observable_verbose){
77        cout << endl;
78        for(int i=0; i<j; i++) cout << i << '\t' << n[i] << '\t' << sum[i]/n[i] << '\
              t' << sum2[i]/n[i] << '\t' << delta[i] << endl;
79      }
80    }
81
82    Surcharge l'opérateur de flux
83    friend std::ostream & operator<<(std::ostream &flux, const observable &x){
84      flux << x.moyenne << '\t' << x.erreur << '\t' << x.tau;
85      return flux;
86    }
87
88    remet à zéro les sommes partielles
89    void reset(){
90      sum.clear();
91      sum2.clear();
92      prec.clear();
93      delta.clear();
94      n.clear();
95    }
96  };
97
98  #endif
```

Code 6.3 : Classe de modèle statistique : Ising.h

```
1  #ifndef ISING_H_
2  #define ISING_H_
3
4  #include <iostream>
5  #include <cassert>
6  #include <cmath>
7  #include <vector>
8  #include "Random.h"
9  #include "observable.h"
10
11 using namespace std;
12
13 #define GNUPLOT
14
15 #ifdef GNUPLOT
16 extern "C"{
17 #include "gnuplot_i.h"
18 }
```

```
19 #endif
20
21 class Ising{
22 public:
23   int L; nombre de sites sur un côté du carré
24   int LL; = L*L;
25   observable E; énergie
26   observable M; aimantation
27   Random R; générateur de nombre aléatoire
28   char **s; variables d'état (tableau 2D)
29   double T; température (Tc = 2/(ln(1+\sqrt2)) = 2.2691853142130221)
30
31    constructeur et initialisation
32   Ising(int _L) : L(_L){
33     assert(L > 0 and L <= LMAX);
34     s = new char*[L];
35     for(int i=0; i<L; i++) s[i] = new char[L];
36
37     LL = L*L;
38
39      configuration initiale aléatoire
40     for(int x=0; x<L; x++){
41       for(int y=0; y<L; y++) s[x][y] = R.integer()%2;
42     }
43     energie();
44
45   }
46
47   Calcule l'énergie et l'aimantation de la configuration
48   void energie(){
49     int e = 0;
50     int m = -LL;
51     for(int x=0; x<L; x++){
52       for(int y=0; y<L; y++){
53         if((s[x][y]^s[(x+1)%L][y])&1) e--;
54         else e++;
55
56         if((s[x][y]^s[x][(y+1)%L])&1) e--;
57         else e++;
58
59         if(s[x][y]) m += 2;
60       }
61     }
62     M.value = (double)m;
63     E.value = -(double)e;
64   }
65
66   Changement d'énergie obtenu en inversant le spin à (x,y)
67   double delta(int x, int y){
```

```
68      int D=0;
69
70      if((s[x][y]^s[(x+1)%L][y])&1) D--;
71      else D++;
72
73      if((s[x][y]^s[x][(y+1)%L])&1) D--;
74      else D++;
75
76      if((s[x][y]^s[(x+L-1)%L][y])&1) D--;
77      else D++;
78
79      if((s[x][y]^s[x][(y+L-1)%L])&1) D--;
80      else D++;
81
82      return(2.0*D);
83    }
84
85      Met à jour la configuration par l'algorithme de Metropolis
86    void update(){
87       choisir le spin à retourner
88      int xy = R.integer()%LL;
89      int x = xy/L;
90      int y = xy%L;
91      double del = delta(x,y);
92
93      bool flip = false;
94
95      if(del < 0) flip = true; le renversement est accepté
96      else{
97        if(R.uniform() < exp(-del/T)) flip = true; le renversement est accepté
98      }
99
100     if(flip){
101       s[x][y]^=1; renverse le spin
102       E.value += del;
103       if(s[x][y]&1) M.value += 2.0;
104       else M.value -= 2.0;
105     }
106   }
107
108   Effectue la simulation, incluant le réchauffement.
109   nmesure: nombre de mesures
110   intervalle: intervalle entre les mesures
111   rechauffement: période de réchauffement
112   T: température
113   void simulate(unsigned int log2N, int intervalle, int rechauffement, double _T){
114     T = _T;
115     if(log2N < MINFLOOR) log2N = MINFLOOR;
116
```

```
117 #ifdef GNUPLOT
118     ofstream gout;
119     gnuplot_ctrl * GP;
120     GP = gnuplot_init() ;
121     gnuplot_cmd(GP,(char *)"set term x11");
122     gnuplot_cmd(GP,(char *)"set xr [0:%d]",L-1);
123     gnuplot_cmd(GP,(char *)"set yr [0:%d]",L-1);
124     gnuplot_cmd(GP,(char *)"unset tics");
125     gnuplot_cmd(GP,(char *)"set size ratio 1");
126     gnuplot_cmd(GP,(char *)"set nokey");
127 #endif
128
129     for(int i=0; i<rechauffement; i++) update();
130
131     E.reset();
132     M.reset();
133     while(1){
134       for(int j=0; j<intervalle; j++) update();
135       unsigned int level=0;
136       E.collecte(level,E.value);
137       level=0; M.collecte(level,M.value);
138
139       if(level==log2N) break;
140 #ifdef GNUPLOT
141         gout.open("tmp.dat");
142         plot(gout);
143         gout.close();
144         gnuplot_cmd(GP,(char *)"set title 'T = %2.3f, M = %1.3f'",T,M.value/LL);
145         gnuplot_cmd(GP,(char *)"plot 'tmp.dat' i 0 pt 5 ps 2 lc 0");
146         usleep(5000);
147 #endif
148     }
149     E.calcul_erreur();
150     M.calcul_erreur();
151
152     int nn = E.sum.size()-FLOOR_OFFSET-1;
153
154     ofstream fout("out.dat",ios::app);
155     cout << T << '\t' << (1<<nn) << '\t' << E << '\t' << M << endl;
156     fout << "# T\tN\tE\tdE\ttau(E)\tM\tdM\ttau(M)\n";
157     fout << T << '\t' << (1<<nn) << '\t' << E << '\t' << M << endl;
158     fout.close();
159   }
160
161    imprime la configuration courante à l'écran
162   void plot(ostream &fout){
163     fout <<"\n\n";
164     for(int x=0; x<L; x++){
165       for(int y=0; y<L; y++){
```

```
166           if(s[x][y]&1) fout << x << '\t' << y << '\n';
167         }
168       }
169       fout <<"\n\n";
170       for(int x=0; x<L; x++){
171         for(int y=0; y<L; y++){
172           if(!s[x][y]) fout << x << '\t' << y << '\n';
173         }
174       }
175       fout <<"\n\n";
176     }
177
178     destructeur
179     ~Ising(){
180       for(int i=0; i<L; i++) delete[] s[i];
181       delete[] s;
182     }
183   };
184
185   #endif
```

Code 6.4 : Programme principal : Ising.cpp

```
1   #include <iostream>
2   #include <fstream>
3
4   #include "read_parameter.h"
5   #include "Ising.h"
6   using namespace std;
7
8   bool observable_verbose;
9
10  int main() {
11
12    double T1,T2,delta_T,h;
13    int L;
14    unsigned long int log2N;
15    int intervalle;
16    int rechauffement;
17
18    fstream fin("para.dat");
19
20    fin >> "L" >> L;
21    fin >> "log2N" >> log2N;
22    fin >> "intervalle" >> intervalle;
23    fin >> "rechauffement" >> rechauffement;
```

```
24    fin >> "temperature_initiale" >> T1;
25    fin >> "temperature_finale" >> T2;
26    fin >> "delta_T" >> delta_T;
27    if(fin=="verbose") observable_verbose = true;
28
29    assert(T1 > 0);
30    assert(T2 > 0);
31
32    fin.close();
33
34    cout << "Simulation du modèle d'Ising 2D sur un réseau " << L << " x " << L <<
          endl;
35    cout << "Analyse d'erreur logarithmique à " << log2N << " niveaux." << endl;
36
37    Ising systeme(L);
38    if(T1<T2) for(double T=T1; T <= T2; T += delta_T) systeme.simulate(log2N,
          intervalle, rechauffement, T);
39    else for(double T=T1; T >= T2; T -= delta_T) systeme.simulate(log2N, intervalle,
          rechauffement, T);
40
41    cout << "Fin normale du programme\n";
42  }
```

Chapitre 7

Dynamique des fluides

L'étude du mouvement des fluides est l'une des applications les plus courantes du calcul scientifique. Elle est particulièrement répandue en génie mécanique, dans la conception de véhicules (automobile, aéronautique) et de réacteurs. En physique, elle constitue le problème de base de la prévision météorologique et climatique, et de la dynamique stellaire. Bref, il s'agit d'un problème extrêmement important à la fois en sciences fondamentales et, surtout, en sciences appliquées.

Le problème du mouvement des fluides va bien au-delà de l'étude de l'écoulement d'un fluide unique dans une géométrie statique. Dans les problèmes concrets, le fluide comporte plusieurs composantes, c'est-à-dire plusieurs substances formant un mélange ; ces substances peuvent se transformer l'une dans l'autre par réactions (chimiques, nucléaires, etc.). Il peut s'agir de phases différentes (liquide et gaz, ou même liquide et solide). La géométrie dans laquelle le ou les fluides s'écoulent peut aussi changer dans le temps, les parois solides peuvent être élastiques (par ex. en aéronautique) ou impliquer une croissance (ex. propagation d'une frontière de phase liquide-solide). Bref, la richesse et la complexité des problèmes impliquant l'écoulement des fluides peut être considérable, et nous ne ferons qu'effleurer le sujet dans ce chapitre, en nous concentrant sur une méthode de calcul : la méthode de Boltzmann sur réseau.

7.1 Équations fondamentales

7.1.1 Équation de Navier-Stokes

Pendant très longtemps, la mécanique des fluides s'est réduite à une tentative de résolution de l'équation de Navier-Stokes. Cette équation gouverne l'évolution dans le temps d'un champ de vitesse $\mathbf{u}(\mathbf{r},t)$, qui décrit la vitesse au point $\mathbf{r}$ d'un fluide à une composante. La densité du fluide est décrite par un champ scalaire $\rho(\mathbf{r})$, et le produit $\rho\mathbf{u}$ définit une densité de courant qui obéit à l'équation de continuité, reflétant la conservation de la masse :

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0 \tag{7.1}$$

L'équation de Navier-Stokes elle-même est

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla P + \rho\nu\nabla^2\mathbf{u} + \rho\mathbf{f} \tag{7.2}$$

où :

1. P est la pression à la position $\mathbf{r}$.
2. ν le coefficient de viscosité cinématique.
3. $\mathbf{f}$ est une force externe (par unité de masse) agissant au point $\mathbf{r}$

L'origine des différents termes de l'équation de Navier-Stokes est la suivante :

1. l'opérateur
$$\frac{\partial}{\partial t} + \mathbf{u}\cdot\nabla \tag{7.3}$$
est une *dérivée en co-mouvement*, qui calcule la dérivée par rapport au temps d'une quantité liée à une élément de masse qui se déplace à une vitesse $\mathbf{u}$. Si on suit un élément de fluide pendant un cours instant, la dérivée temporelle d'une quantité relative à cet élément doit tenir compte du déplacement de cet élément pendant cet instant. La dérivée totale par rapport au temps est alors
$$\frac{\mathrm{d}}{\mathrm{d}t} = \frac{\partial}{\partial t} + \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t}\cdot\frac{\partial}{\partial \mathbf{r}} = \frac{\partial}{\partial t} + \mathbf{u}\cdot\nabla \tag{7.4}$$
Appliquer une simple dérivée partielle par rapport au temps, à position fixe, serait donc erroné. Le membre de gauche de l'équation (7.2) représente donc la dérivée temporelle de la quantité de mouvement par unité de volume.
2. Le membre de droite de l'équation (7.2) contient alors les forces par unité de volume qui agissent sur l'élément de fluide : la première est le gradient de pression ; la deuxième une force qui s'oppose aux variations spatiales de la vitesse (la viscosité) et la dernière une force externe quelconque, comme par exemple la force de gravité.

L'équation (7.2) par elle-même est incomplète : on doit lui ajouter l'équation de continuité (7.1), qui représente la contrainte de conservation de la masse. On doit également disposer d'une équation d'état qui détermine la pression $P(\rho, T)$ en fonction de la densité et d'un paramètre externe comme la température (en supposant que celle-ci soit uniforme et constante). Si la température n'est pas uniforme, alors il faut ajouter un champ de température $T(\mathbf{r}, t)$ qui sera gouverné par la conservation de l'énergie, sous la forme d'une équation de continuité pour l'énergie interne, avec un courant de chaleur associé.

7.1.2 Équation de Boltzmann

L'équation de Navier-Stokes est valable dans la limite macroscopique, ou *hydrodynamique*, dans laquelle les longueurs caractéristiques du problème (les dimensions des conduits, par exemple) sont très grandes par rapport au libre parcours moyen des molécules qui forment le fluide, afin que la notion d'élément de fluide ait un sens physique.

L'équation de Navier-Stokes peut être dérivée d'une équation plus fondamentale, c'est-à-dire plus microscopique : l'équation de Boltzmann. Cette dernière régit l'évolution dans le temps d'une distribution de probabilité de positions et de vitesses dans l'espace des phases et est l'équation fondamentale de la mécanique statistique hors équilibre.

Considérons à cet effet un fluide comportant un très grand nombre N de particules. Si nous avions la prétention d'étudier la dynamique classique exacte de ce système complexe, nous devrions considérer l'espace des phases de ce système, qui comporte $6N$ dimensions, pour les $3N$ positions et $3N$ impulsions du système. L'état classique du système serait alors défini par un point dans cet espace et l'évolution temporelle du système serait une trajectoire suivie par ce point dans l'espace des phases. Comme il est irréaliste de procéder ainsi, nous allons plutôt raisonner sur la base de l'espace des phase pour une seule molécule, en supposant que les N molécules du fluide sont distribuées dans cet espace des phases. Une telle distribution est définie par une fonction $f(\mathbf{r}, \mathbf{p}, t)$ de la position et de l'impulsion ; la probabilité de trouver une molécule dans un élément de volume $\mathrm{d}^3 r$ à la position $\mathbf{r}$, possédant une impulsion contenue dans un élément de volume $\mathrm{d}^3 p$ autour de $\mathbf{p}$, est alors proportionnel à $f(\mathbf{r}, \mathbf{p})\mathrm{d}^3 r\, \mathrm{d}^3 p$. Nous allons normaliser la distribution f par le nombre total N de particules :

$$\int \mathrm{d}^3 r\, \mathrm{d}^3 p\; f(\mathbf{r}, \mathbf{p}) = N \tag{7.5}$$

Moments de la distribution

Si on intègre la distribution f sur toutes les valeurs de l'impulsion $\mathbf{p}$ à une position donnée $\mathbf{r}$, on génère des *moments partiels* de la distribution de probabilité. Les trois premiers moments correspondent à la densité, la vitesse moyenne et l'énergie cinétique moyenne par unité de volume :

$$\begin{aligned} \rho(\mathbf{r}) &= m \int \mathrm{d}^3 p\; f(\mathbf{r}, \mathbf{p}) \\ \rho(\mathbf{r})\mathbf{u}(\mathbf{r}) &= \int \mathrm{d}^3 p\; \mathbf{p} f(\mathbf{r}, \mathbf{p}) \\ \rho(\mathbf{r})\epsilon &= \frac{1}{2} \int \mathrm{d}^3 p\; (\mathbf{p} - m\mathbf{u})^2 f(\mathbf{r}, \mathbf{p}) \end{aligned} \tag{7.6}$$

En effet, la première intégrale nous donne naturellement le nombre de particules par unité de volume, ou encore la densité massique ρ si on multiplie par la masse m de chaque particule. La deuxième expression est l'impulsion par unité de volume à une position $\mathbf{r}$, soit précisément la densité ρ multipliée par la vitesse $\mathbf{u}$ du fluide à cet endroit. Enfin, la dernière expression est proportionnelle à la variance de l'impulsion à la position $\mathbf{r}$, ce qui n'est autre que l'énergie cinétique par unité de volume, telle que définie dans le référentiel qui se déplace à la vitesse moyenne des particules à cet endroit ; c'est donc l'énergie cinétique interne par unité de volume, ϵ étant l'énergie moyenne d'une particule de fluide dans ce référentiel.

Équation de Boltzmann

Si les N molécules sont indépendantes les unes des autres et ne répondent qu'à des forces externes, alors l'évolution dans le temps de la distribution f suit les règles de la mécanique

de Hamilton à une particule : au temps $t + \mathrm{d}t$, la molécule à $(\mathbf{r}, \mathbf{p})$ s'est déplacée vers $(\mathbf{r} + \mathbf{p}\mathrm{d}t/m, \mathbf{p} + \mathbf{f}\mathrm{d}t)$, où $\mathbf{f}$ est une force externe conservative agissant sur chaque particule. Donc

$$f(\mathbf{r} + \mathbf{p}\mathrm{d}t/m, \mathbf{p} + \mathbf{f}\mathrm{d}t, t + \mathrm{d}t) = f(\mathbf{r}, \mathbf{p}, t) \tag{7.7}$$

ou encore, en développant,

$$\frac{\partial f}{\partial t} + \frac{1}{m}\mathbf{p} \cdot \frac{\partial f}{\partial \mathbf{r}} + \mathbf{f} \cdot \frac{\partial f}{\partial \mathbf{p}} = 0 \tag{7.8}$$

Cette équation décrit des particules indépendantes qui se déplacent sans interagir. On peut se figurer la distribution f comme un nuage de probabilité qui se déplace dans l'espace des phases. Selon le théorème de Liouville, [1] une portion donnée de l'espace des phases évolue dans le temps en se déformant mais en conservant son volume. Le nuage de probabilité associé à f s'écoule donc dans l'espace des phases en se déformant, mais de manière incompressible.

Ceci n'est plus vrai si on tient compte des collisions entre les particules élémentaires du fluide. À cette fin, on modifie la relation (7.8) comme suit :

$$\frac{\partial f}{\partial t} + \frac{1}{m}\mathbf{p} \cdot \frac{\partial f}{\partial \mathbf{r}} + \mathbf{f} \cdot \frac{\partial f}{\partial \mathbf{p}} = \left.\frac{\delta f}{\delta t}\right|_{\text{coll.}} \tag{7.9}$$

où le membre de droite représente tout changement dans f (à un point donné de l'espace des phases) causé par les interactions entre les particules (ou molécules).

Approximation des collisions moléculaires

Le terme de collision est souvent représenté dans *l'approximation des collisions moléculaires*, qui suppose que les molécules collisionnent par paires et que deux molécules incidentes d'impulsions $\mathbf{p}$ et $\mathbf{p}'$ se retrouvent après collision avec des impulsions $\mathbf{p} + \mathbf{q}$ et $\mathbf{p}' - \mathbf{q}$: $\mathbf{q}$ est le transfert d'impulsion entre les deux particules et la quantité de mouvement totale des deux particules est conservée lors de la collision. Cette collisions se produit avec une densité de probabilité par unité de temps $g(\mathbf{p} - \mathbf{p}', \mathbf{q})$ qui ne dépend que de deux variables en raison de l'invariance galiléenne : l'une est la vitesse relative des deux particules, $(\mathbf{p} - \mathbf{p}')/m$, l'autre le paramètre d'impact de la collision, qui se traduit par un angle de diffusion θ, ou de manière équivalente par un transfert d'impulsion $\mathbf{q}$.

La variation de la fonction f à $\mathbf{p}$ due aux collisions avec des molécules d'impulsion $\mathbf{p}'$ est alors

$$\left.\frac{\delta f}{\delta t}\right|^{(1)}_{\text{coll.}} = -\int \mathrm{d}^3p' \, \mathrm{d}^3q \, g(\mathbf{p} - \mathbf{p}', \mathbf{q}) f(\mathbf{r}, \mathbf{p}, t) f(\mathbf{r}, \mathbf{p}', t) \tag{7.10}$$

Il s'agit bien sûr d'une diminution du nombre de particules d'impulsion $\mathbf{p}$ due aux collisions qui diffusent ces particules vers d'autres impulsions. Par contre, les collisions peuvent aussi augmenter f à $\mathbf{p}$ en raison des particules qui diffusent *vers* l'impulsion $\mathbf{p}$. Cette augmentation s'écrit naturellement comme

$$\left.\frac{\delta f}{\delta t}\right|^{(2)}_{\text{coll.}} = \int \mathrm{d}^3p' \, \mathrm{d}^3q \, g(\mathbf{p} - \mathbf{p}', \mathbf{q}) f(\mathbf{r}, \mathbf{p} + \mathbf{q}, t) f(\mathbf{r}, \mathbf{p}' - \mathbf{q}, t) \tag{7.11}$$

1. Voir le cours de Mécanique II – PHQ310

qu'il faut lire comme suit : les particules d'impulsion $\mathbf{p}+\mathbf{q}$ diffusent sur les particules d'impulsion $\mathbf{p}'-\mathbf{q}$, ce qui résulte, après un transfert d'impulsion $\mathbf{q}$, vers les impulsions $\mathbf{p}$ et $\mathbf{p}'$. On intègre sur toutes les possibilités $\mathbf{p}'$ et $\mathbf{q}$ pour obtenir la variation ci-dessus. Au total, l'équation de Boltzmann dans l'approximation des collision moléculaires (*Stosszahl Ansatz*) prend la forme suivante :

$$\frac{\partial f}{\partial t}+\frac{1}{m}\mathbf{p}\cdot\frac{\partial f}{\partial \mathbf{r}}+\mathbf{f}\cdot\frac{\partial f}{\partial \mathbf{p}}=\int \mathrm{d}^3p'\,\mathrm{d}^3q\,g(\mathbf{p}-\mathbf{p}',\mathbf{q})\Big[f(\mathbf{r},\mathbf{p}+\mathbf{q},t)f(\mathbf{r},\mathbf{p}'-\mathbf{q},t)-f(\mathbf{r},\mathbf{p},t)f(\mathbf{r},\mathbf{p}',t)\Big] \tag{7.12}$$

L'équation (7.12) est difficile à résoudre : c'est une équation intégro-différentielle, qui implique une intégrale double sur les impulsions à chaque évaluation de la dérivée temporelle de f. On peut montrer (mais nous ne le ferons pas ici, car c'est une proposition assez complexe) que l'équation de Navier-Stokes est une conséquence de l'équation (7.12).

Remarquons qu'en l'absence de forces externes, l'équation de Boltzmann sans le terme de collisions entraîne que les distributions partielles $f(\mathbf{r},\mathbf{p},t)$ associées à des valeurs différentes de $\mathbf{p}$ sont indépendantes, c'est-à-dire qu'elles ne s'influencent nullement en fonction du temps. Le terme de collision se trouve à coupler entre elles des valeurs différentes de $\mathbf{p}$.

Approximation du temps de relaxation

On s'attend naturellement à ce que la distribution f tende vers une distribution d'équilibre $f^{\mathrm{eq.}}$ en raison des collisions. Cette convergence vers $f^{\mathrm{eq.}}$ va s'effectuer en général avec un certain temps caractéristique τ si la distribution initiale n'est pas trop éloignée de $f^{\mathrm{eq.}}$. Une façon courante de simplifier considérablement l'équation de Boltzmann est de remplacer le terme de collision par une simple relaxation vers la distribution d'équilibre :

$$\frac{\partial f}{\partial t}+\frac{1}{m}\mathbf{p}\cdot\frac{\partial f}{\partial \mathbf{r}}+\mathbf{f}\cdot\frac{\partial f}{\partial \mathbf{p}}=\frac{1}{\tau}\Big[f^{\mathrm{eq.}}(\mathbf{r},\mathbf{p})-f(\mathbf{r},\mathbf{p},t)\Big] \tag{7.13}$$

La distribution $f^{\mathrm{eq.}}$ se calcule, bien sûr, en mécanique statistique des systèmes à l'équilibre. Si la distribution décrit un fluide qui se déplace à une vitesse $\mathbf{u}(\mathbf{r})$, ce sont les déviations par rapport à cette vitesse qui seront l'objet d'une distribution de Maxwell-Boltzmann :

$$f^{\mathrm{eq.}}(\mathbf{r},\mathbf{p})=\frac{\rho}{m(2\pi mT)^{d/2}}\exp-\frac{[\mathbf{p}-m\mathbf{u}(\mathbf{r})]^2}{2mT} \tag{7.14}$$

où d est la dimension de l'espace et T la température absolue (on pose $k_B=1$). Notez que, dans cette dernière équation, la vitesse $\mathbf{u}$ dépend en général de la position. Cette distribution des impulsions à l'équilibre respecte les règles de somme décrites à l'éq. (7.6), sauf que dans ce cas la densité d'énergie cinétique ϵ est donnée par le théorème d'équipartition : $\epsilon=Td/2$.

Notons que l'approximation du temps de relaxation couple effectivement les différentes valeurs de $\mathbf{p}$: la distribution à l'équilibre $f^{\mathrm{eq.}}$ dépend de la vitesse du fluide $\mathbf{u}$ à un point donné, et celle-ci à son tour se calcule par une intégrale sur les différentes impulsions.

7.2 Méthode de Boltzmann sur réseau

7.2.1 Généralités

Depuis les années 1990, l'hydrodynamique numérique se tourne de plus en plus vers l'équation de Boltzmann, au lieu de s'attaquer à l'équation de Navier-Stokes. L'avantage de l'équation de Boltzmann est qu'elle permet plus facilement de considérer des géométries complexes (des milieux poreux, par exemple) et des fluides à plusieurs composantes, en tenant compte des réactions entre ces composantes : le terme de collision peut en effet convertir des espèces de molécules en d'autres espèces, c'est-à-dire décrire la cinétique des réactions chimiques. Le traitement numérique de l'équation de Boltzmann dans le but de décrire le mouvement des fluides constitue ce qu'on appelle la *méthode de Boltzmann sur réseau* (angl. *lattice Boltzmann method* ou LBM). Beaucoup de logiciels commerciaux ou ouverts utilisés en génie ou ailleurs sont maintenant basés sur cette méthode.

Cependant, l'équation (7.12) est encore trop complexe. La méthode LBM est basée plutôt sur la version simplifiée (7.13) de l'équation de Boltzmann. C'est ce qu'on appelle dans ce contexte le modèle de Bhatnagar-Gross-Krook (BGK).

La numérisation de l'équation (7.13) passe nécessairement par une discrétisation de l'espace des phases. En pratique, la partie spatiale a besoin d'une discrétisation fine, car souvent on s'intéresse au passage des fluides à travers des géométries compliquées. Par contre, la discrétisation de l'espace des vitesses (ou des impulsions) est extrêmement simplifiée : on considère généralement 9 valeurs de la vitesse en deux dimensions (schéma D2Q9), 15 ou 27 en dimension 3 (schémas D3Q15 et D3Q27). Il peut sembler surprenant à première vue qu'un aussi petit nombre de vitesses puisse bien représenter le mouvement des fluides, mais il faut garder à l'esprit qu'on décrit ici une distribution de probabilité f, et que la vitesse du fluide qui sera produite par cette approche est la moyenne pondérée des probabilités associées aux 9 (ou 15, ou 27) valeurs formant la grille des vitesses, et donc qu'elle pourra prendre un continuum de valeurs dans l'espace borné par cette grille.

7.2.2 Le schéma D2Q9 : dimension 2, 9 vitesses

Voyons plus précisément comment se formule la méthode dans le schéma D2Q9 (voir fig. 7.1). Les sites forment un réseau carré de pas a. Définissons les 9 vecteurs sans unités $\mathbf{e}_i$ ($i = 0, \ldots, 8$) :

$$\begin{array}{lll} & \mathbf{e}_1 = (1,0) & \mathbf{e}_5 = (1,1) \\ & \mathbf{e}_2 = (0,1) & \mathbf{e}_6 = (-1,1) \\ \mathbf{e}_0 = (0,0) & \mathbf{e}_3 = (-1,0) & \mathbf{e}_7 = (-1,-1) \\ & \mathbf{e}_4 = (0,-1) & \mathbf{e}_8 = (1,-1) \end{array} \tag{7.15}$$

Les 9 déplacements possibles sont $a\mathbf{e}_i$ et sont illustrés par les flèches sur la figure. Les impulsions correspondantes sont définies de telle manière qu'en un intervalle de temps h (le pas

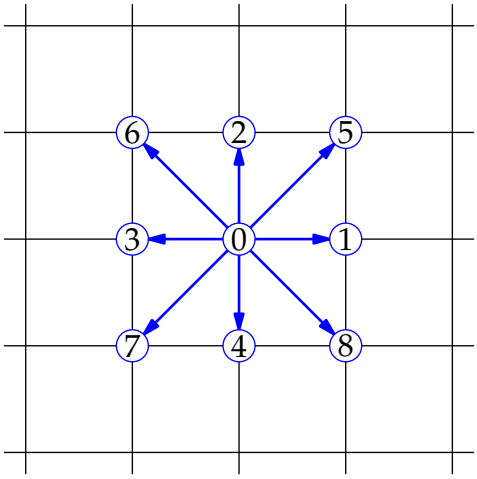


FIGURE 7.1 Les 9 vitesses possibles dans le schéma D2Q9 de la méthode de Boltzmann sur réseau, en relation avec les sites de la grille.

temporel de la méthode), le déplacement $a\mathbf{e}_i$ soit la différence de deux positions appartenant au réseau : $\mathbf{p}_i = (ma/h)\mathbf{e}_i$.

La distribution $f(\mathbf{r},\mathbf{v})$ devient dans ce cas un ensemble de 9 distributions $f_i(\mathbf{r})$, où les $\mathbf{r}$ appartiennent au réseau. La densité du fluide au point $\mathbf{r}$ est alors

$$\rho(\mathbf{r}) = m\sum_i f_i(\mathbf{r}) \tag{7.16}$$

alors que la vitesse $\mathbf{u}$ du fluide est telle que

$$\rho(\mathbf{r})\mathbf{u}(\mathbf{r}) = \frac{ma}{h}\sum_i f_i(\mathbf{r})\mathbf{e}_i \tag{7.17}$$

La version discrète de l'équation de Boltzmann (7.13) est alors

$$f_i(\mathbf{r}+\mathbf{e}_i h, t+h) = f_i(\mathbf{r},t) - \frac{h}{\tau}\left[f_i(\mathbf{r},t) - f_i^{\text{eq.}}(\mathbf{r},t)\right] \tag{7.18}$$

Le premier terme du membre de droite effectue l'*écoulement* du fluide, alors que le deuxième terme représente l'effet des collisions. Nous avons supposé ici qu'aucune force externe $\mathbf{f}$ n'est à l'oeuvre. Notons qu'en l'absence de terme de collision, cette équation revient simplement à déplacer rigidement la distribution f_i sur une distance $a\mathbf{e}_i$ lors d'un pas temporel h : encore une fois il n'y a aucun mélange des vitesses et les différentes distributions sont indépendantes. Le terme de relaxation va modifier ce comportement en couplant les distributions partielles f_i via le calcul de la vitesse du fluide $\mathbf{u}$.

Nous allons déterminer la forme de la distribution à l'équilibre $f_i^{\text{eq.}}$ en nous inspirant de la distribution de Maxwell-Boltzmann (7.14) et en imposant les règles de somme (7.6), qui de-

viennent dans ce contexte

$$\begin{aligned}\rho(\mathbf{r}) &= m\sum_i f_i^{\text{eq.}}(\mathbf{r})\\ \rho(\mathbf{r})\mathbf{u}(\mathbf{r}) &= \sum_i \mathbf{p}_i f_i^{\text{eq.}}\\ \rho(\mathbf{r})T &= \frac{1}{2}\sum_i (\mathbf{p}_i - m\mathbf{u})^2 f_i^{\text{eq.}}\end{aligned} \tag{7.19}$$

En principe, ces règles de somme devraient s'appliquer pour toute valeur de $\mathbf{u}$. Malheureusement, le nombre limité de vitesses (neuf), ne nous permet pas d'y arriver : nous ne pourrons respecter ces contraintes que pour les termes constants, linéaires et quadratiques en u. Autrement dit, il ne sera possible d'imposer ces contraintes que dans l'approximation des petites vitesses (petits nombres de Mach). En développant la distribution (7.14) à cet ordre en u, nous obtenons la forme suivante :

$$f_i^{\text{eq.}} = w_i\rho(\mathbf{r})\left\{1 + \frac{ma}{hT}\mathbf{e}_i\cdot\mathbf{u} - \frac{m}{2T}\mathbf{u}^2 + \frac{1}{2}\left(\frac{ma}{hT}\right)^2(\mathbf{e}_i\cdot\mathbf{u})^2\right\} \tag{7.20}$$

où les constantes w_i sont considérées comme ajustables. En raison de la symétrie des 9 vecteurs $\mathbf{e}_i$, il est clair que trois seulement de ces constantes sont indépendantes : w_0, $w_1 = w_2 = w_3 = w_4$ et $w_5 = w_6 = w_7 = w_8$. L'imposition des trois règles de somme (7.19) à l'ordre u^2 fixe de manière unique ces constantes. Pour simplifier les choses, il est de coutume de fixer le pas temporel h de manière à ce que

$$\frac{ma^2}{h^2T} = 3 \tag{7.21}$$

On démontre alors que

$$w_0 = \frac{4}{9} \qquad w_{1,2,3,4} = \frac{1}{9} \qquad w_{5,6,7,8} = \frac{1}{36} \tag{7.22}$$

Exercice 7.1

Obtenez explicitement les valeurs (7.22) à partir des conditions (7.19) et de la distribution à l'équilibre (7.20), en négligeant les termes d'ordre supérieur à u^2.

On montre que ce modèle spécifique nous ramène, dans la limite continue, à l'équation de Navier-Stokes pour une fluide incompressible :

$$\nabla\cdot\mathbf{u} = 0 \qquad \rho\frac{\partial\mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla P + \rho\nu\nabla^2\mathbf{u} \tag{7.23}$$

où la pression P est reliée à la densité par

$$P = c_s^2\rho \qquad c_s = \quad \text{vitesse du son} \quad = \frac{1}{\sqrt{3}} \tag{7.24}$$

et le coefficient de viscosité est relié au temps de relaxation par

$$\nu = c_s^2\left(\frac{\tau}{h} - \frac{1}{2}\right) \tag{7.25}$$

Conditions aux limites

Comment traiter les conditions aux limites dans le méthode de Boltzmann sur réseau ? Autrement dit, quel doit être le comportement du fluide au contact d'une surface ? La façon la plus simple et la plus courante de traiter une paroi, en fait l'une des forces de la méthode, est la *condition de rebond* (angl. *no slip condition*), qui stipule que la particule qui collisionne avec la paroi est rétrodiffusée ($\mathbf{e}_i \rightarrow -\mathbf{e}_i$). En pratique, cela signifie que l'équation (7.18) ne s'applique pas aux sites du réseau situés sur la paroi. Au contraire, sur une tel site, la distribution $f_i(\mathbf{r}, t+h)$ est simplement donnée par $f_j(\mathbf{r}, t)$, où $\mathbf{e}_j = -\mathbf{e}_i$. Cette condition de rebond revient à dire que le fluide ne peut pas glisser le long de la paroi : celle-ci exerce sur le fluide un frottement statique parfait.

Algorithme

Résumons ici l'algorithme de la méthode LBM :

1. On commence par mettre en place une distribution initiale $f_i(\mathbf{r})$ associée à des valeurs initiales de la densité ρ et de la vitesse $\mathbf{u}$ (gardons en tête que $\mathbf{r}$ est maintenant un indice discret défini sur une grille régulière). À cette fin, on initialise $f_i = f_i^{\text{eq.}}(\mathbf{u}_0)$ en fonction d'une vitesse initiale spécifiée à l'avance $\mathbf{u}_0(\mathbf{r})$.
2. À chaque pas temporel, on propage les distributions selon l'équation (7.18) sur les noeuds intérieurs de la grille, et on applique les conditions de rebond sur les frontières.
3. On calcule ensuite les nouvelles densités et vitesses selon les expressions (7.16) et (7.17), ce qui nous permet de calculer les nouvelles distributions à l'équilibre.
4. On retourne à l'étape 2 jusqu'à ce que le temps de simulation soit écoulé.

7.2.3 Exemple

Au lieu d'illustrer la méthode de Boltzmann sur réseau à l'aide d'un code maison, nous allons utiliser l'un des codes libres disponibles : `openLB`.[2] L'un des exemples inclus dans la distribution illustre le passage d'un fluide en deux dimensions entre deux parois horizontales, entre lesquelles figure un obstacle cylindrique – donc à section circulaire. En trois dimensions, cette situation correspondrait au passage d'un fluide dans un conduit cylindrique au milieu duquel un obstacle sphérique a été inséré.

Un instantanné de la simulation est illustré à la figure 7.2. La grandeur $|\mathbf{u}|$ de la vitesse d'écoulement est représentée en fonction de la position par un code de couleur. La partie inférieure de la figure représente plutôt le tourbillon (angl. *vorticity*) $\nabla \wedge \mathbf{u}$ associé à la vitesse $\mathbf{u}$. Les deux sections de gauche correspondent à un écoulement caractérisé par un petit nombre de Reynolds (Re=1), alors que les sections de droite correspondent à un écoulement turbulent, caractérisé par Re=800. Le nombre de Reynolds, en hydrodynamique, est le rapport des forces d'inertie aux forces visqueuses. Sa définition pratique est la suivante :

$$\text{Re} = \frac{uL}{\nu} \tag{7.26}$$

2. Voir http://www.numhpc.org/openlb/

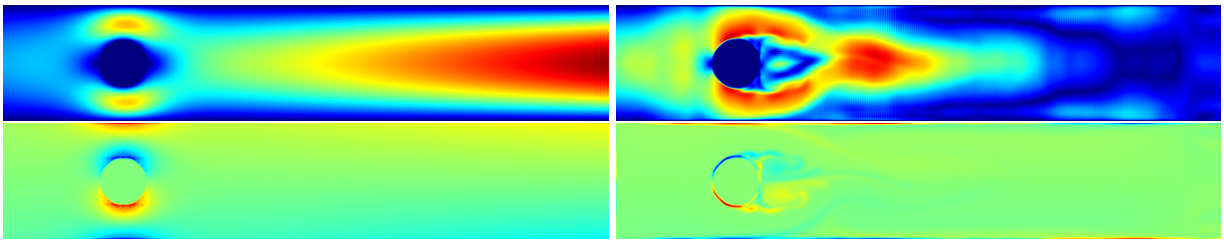


FIGURE 7.2 Simulation du passage d'un fluide 2D entre deux parois, avec un obstacle cylindrique, réalisée à l'aide du logiciel libre `openLB`. En haut : profil de la grandeur de la vitesse $|\mathbf{u}|$ en fonction de la position. Les vitesses plus grandes sont en rouge, les plus faibles en bleu. L'écoulement s'effectue de la gauche vers la droite, et l'obstacle est visible au cinquième de la longueur, à partir de la gauche. En bas, profil du tourbillon $\nabla\wedge\mathbf{u}$ associé au même écoulement. Le tourbillon n'a ici qu'une seule composante (en z) et le code de couleur va des valeurs négatives (bleues) à positives (rouge) en passant par les valeurs nulles (vert pâle). À gauche : écoulement à petit nombre de Reynolds (Re=1), à droite, écoulement turbulent (Re=800).

où

1. u est la vitesse moyenne du fluide, ou la vitesse d'un objet dans un fluide au repos.
2. L est la longueur caractéristique du système (la distance entre les deux parois dans notre exemple).
3. ν est le coefficient de viscosité cinématique (le même qui figure dans l'équation de Navier-Stokes (7.2)).

Cette définition, comme on le voit, n'est pas universelle : elle dépend du problème étudié. Plus le nombre de Reynolds est petit, plus la viscosité est importante ; plus il est grand, plus la turbulence a de chances de s'établir.

On remarque que le tourbillon $\nabla\wedge\mathbf{u}$ est important autour de l'obstacle dans les deux cas (Re=1 et Re=800), ce qui reflète l'existence de la couche limite, c'est-à-dire une couche mince autour des parois où la vitesse varie de zéro sur la paroi à une valeur non nulle dans un espace assez étroit. Le fait que le tourbillon $\nabla\wedge\mathbf{u}$ soit non nul n'entraîne pas nécessairement l'existence d'un écoulement tourbillonnaire : il faut pour cela déterminer les lignes d'écoulement, ce qui n'est pas visible sur les graphiques de la figure 7.2. Cependant, l'écoulement illustré sur la partie droite de la figure est clairement turbulent.

7.3 Simulation de l'écoulement d'un plasma

Nous allons dans cette section procéder à une simulation d'un autre genre, qui se rapproche plus de la dynamique moléculaire : celle de l'écoulement d'un plasma en dimension 1. Cette simulation va révéler une instabilité qui se manifeste dans l'interaction de deux faisceaux opposés.

Un plasma est un gaz d'électrons en coexistence avec des ions, de sorte que le système est neutre au total. Par contre, comme les électrons sont beaucoup plus légers que les ions, les temps caractéristiques associés au mouvement des deux composantes du plasma sont très différents. Nous n'allons étudier ici que le mouvement des électrons, et supposer que le rôle des ions n'est que de neutraliser le système dans son entier.

7.3.1 Description de la méthode

Considérons donc un ensemble de N électrons de charge e, positions r_i et de vitesses v_i en dimension 1. Si nous procédions à une simulation de dynamique moléculaire classique, nous devrions calculer, à chaque instant de la simulation, la force électrique agissant sur chaque particule, ce qui est un processus de complexité $\mathcal{O}(N^2)$. Nous allons réduire la complexité du calcul à $\mathcal{O}(N)$ à l'aide de l'astuce suivante :

1. Les forces seront représentées par un champ électrique $E(x)$, dérivant du potentiel électrique $\phi(x)$ et de la densité de charge $e\rho(x)$ ($\rho(x)$ étant la densité volumique de particules). Ces différents champs seront représentés sur une grille de M points, où $M \ll N$.
2. Les positions r_i des particules seront utilisées pour calculer la densité ρ.
3. Le potentiel $\phi(x)$ sera déterminé par la solution de l'équation de Poisson : $\mathrm{d}^2\phi/\mathrm{d}x^2 = -e\rho(x)/\varepsilon_0$, à l'aide de transformées de Fourier rapides.
4. Le champ électrique $E(x)$ sera dérivé du potentiel électrique : $E(x) = -\mathrm{d}\phi/\mathrm{d}x$.

C'est l'utilisation de transformées de Fourier rapides (TdFR) qui rend cette méthode plus rapide : sa complexité sera la moindre de $\mathcal{O}(N)$ et de $\mathcal{O}(M \log M)$.

Mise à l'échelle du problème

Il est important lors d'une simulation numérique de rapporter les différentes quantités physiques en fonction de quantités normalisées dont les grandeurs caractéristiques sont typiques au problème. En définissant un potentiel normalisé Φ tel que

$$\phi \stackrel{\text{def}}{=} \frac{e\bar{\rho}}{\varepsilon_0}\Phi \tag{7.27}$$

où $\bar{\rho}$ est la densité moyenne d'électrons, l'équation du mouvement de la particule i est

$$\dot{v}_i = \frac{e}{m}E(x_i) = -\frac{e^2\bar{\rho}}{m\varepsilon_0}\Phi'(x_i) \stackrel{\text{def}}{=} -\omega_p\Phi'(x_i) \tag{7.28}$$

où ω_p est la *fréquence plasma*, soit la fréquence à laquelle un plasma uniforme qui serait déplacé latéralement par rapport à son fond neutralisant se mettrait à osciller. Parallèlement, l'équation de Poisson devient alors

$$\Phi'' = -\frac{\rho}{\bar{\rho}} = -n \tag{7.29}$$

où n est la densité normalisée d'électrons. Les équations sont donc particulièrement simples si on les exprime en fonction de $n(x)$ et du potentiel normalisé $\Phi(x)$. La fréquence plasma inverse ω_p^{-1} servira ici d'unité naturelle de temps.

Revoyons donc en détail chaque étape de la méthode :

1. La densité normalisée n, le potentiel électrique normalisé Φ et le champ électrique normalisé $F = -\Phi'$ seront définis sur une grille uniforme périodique de M points et de pas $a = L/M$. La présence d'un électron à la position r modifie la densité n uniquement sur les points x_j et x_{j+1} situés de part et d'autre de r.

$$n_j \to n_j + \frac{L}{Na^2}(x_{j+1} - r) \qquad n_{j+1} \to n_{j+1} + \frac{L}{Na^2}(r - x_j) \qquad x_j < r < x_{j+1} \tag{7.30}$$

2. En fonction des transformées de Fourier $\tilde{\Phi}(q)$ et $\tilde{n}(q)$, l'équation de Poisson prend la forme suivante :

$$q^2\tilde{\Phi}(q) = \tilde{n}(q) \quad \text{et donc} \quad \tilde{\Phi}(q) = \frac{1}{q^2}\tilde{n}(q) \tag{7.31}$$

   En pratique, on procède à une TdFR de n et on calcule $\tilde{\Phi}$ en divisant par q^2 comme ci-dessus ($q \neq 0$). On met également à zéro la composante $q = 0$ de $\tilde{n}$, ce qui entraîne que la charge totale est nulle. Cette étape impose la condition de neutralité et est le seul endroit du calcul où la présence des ions positifs est prise en compte. Enfin, on procède à une TdFR inverse pour retrouver $\Phi_j = \Phi(x_j)$.

3. Le champ électrique normalisé est l'opposée de la dérivée du potentiel électrique normalisé :

$$F_j = -\frac{\Phi_{j+1} + \Phi_{j-1}}{2a} \tag{7.32}$$

4. La méthode de Verlet sera employée pour évoluer dans le temps les positions et les vitesses, selon les équations suivantes :

$$r_i(t+h) = r_i(t) + v_i h + \frac{h^2 f_i}{2} \qquad v_i(t+h) = v_i(t)(1-\gamma h) + \frac{f_i(t) + f_i(t+h)}{2} \tag{7.33}$$

   où γ est un amortissement dû à des causes diverses (rayonnement, etc.). La force f_i à un instant donné sera calculée en interpolant le champ électrique défini sur la grille :

$$f_i = \frac{r_i - x_j}{a} F_j + e\frac{x_{j+1} - r_i}{a} F_{j+1} \tag{7.34}$$

   où x_j est le point no j de la grille périodique (il est implicite que $j + M \to j$, c'est-à-dire que j est défini modulo M) et F_j est la valeur du champ électrique à ce point. On retourne ensuite à l'étape 1 (calcul mis à jour de la densité), jusqu'à ce que le temps de simulation soit écoulé.

Exercice 7.2

Décrivez comment on pourrait appliquer la méthode décrite dans cette section à une force inter-particule générale qui dérive d'un potentiel central $U(r)$ en dimension 3, de sorte que l'énergie potentielle d'une particule à la position $\mathbf{r}_i$ en présence des autres particules est

$$V(\mathbf{r}_i) = \sum_{j \neq i} U(|\mathbf{r}_i - \mathbf{r}_j|) \tag{7.35}$$

A Formulez la méthode en fonction des transformées de Fourier $\tilde{U}(\mathbf{q})$ de $U(\mathbf{r}) = U(r)$ et $\tilde{\rho}$ de la densité volumique des particules $\rho(\mathbf{r})$.

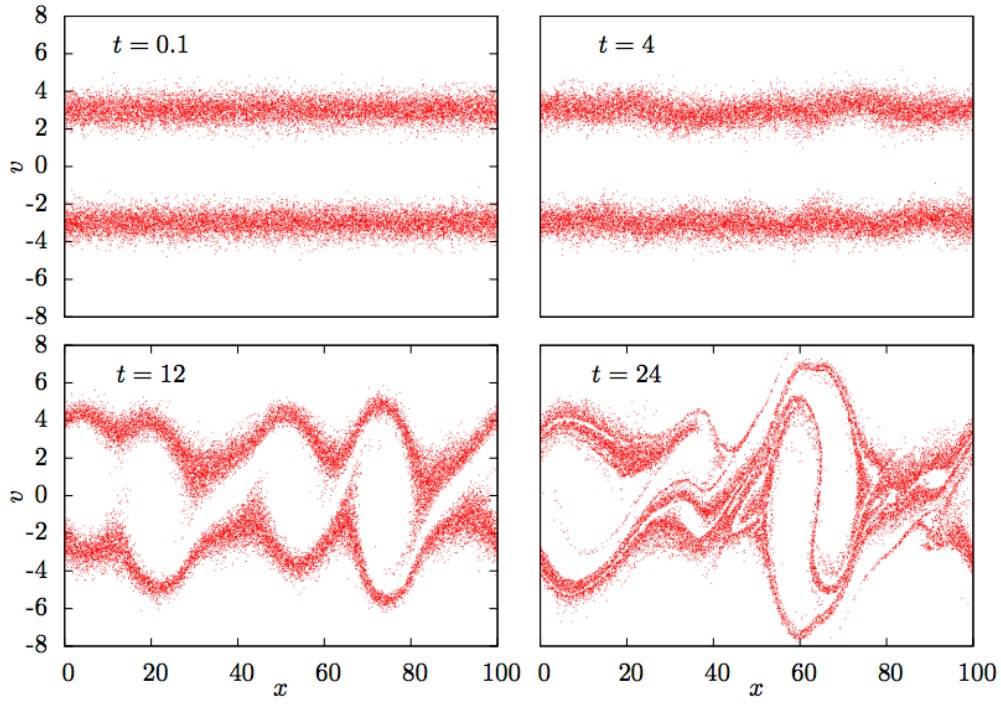


Figure 7.3 Distribution des positions et des vitesses des électrons du plasma dans l'espace des phases, pour 4 valeurs du temps t. L'instabilité est visible dès $t = 4$. Les paramètres utilisés sont $v_0 = 1$, $v_T = 0.5$, $h = 0.001$, $\gamma = 0$, $L = 100$, $M = 1024$ et $N = 20\,000$.

B Quelle condition le potentiel $U(r)$ doit-il respecter pour que $\tilde{U}$ soit bien défini ? Comment peut-on modifier $U(r)$ au besoin, sans conséquences physiques, afin que cette cette condition soit respectée ? Considérez le potentiel de Lennard-Jones comme exemple.

Les conditions initiales seront les suivantes :

1. Une distribution initiale aléatoire des positions r_i dans l'intervalle $[0, L]$.
2. Une distribution gaussienne des vitesses v_i autour de deux valeurs $\pm v_0$, ce qui représente deux faisceaux contraires de particules. L'écart-type de cette distribution des vitesses est $v_T = \sqrt{T/m}$, où T est la température absolue. Cette distribution est donc maxwellienne.

Le problème est véritablement unidimensionnel, ce qui signifie qu'on néglige le rayon R du faisceau et les effets transversaux en comparaison de la longueur L : $R \ll L$.

Outre la fréquence plasma ω_p, un échelle caractéristique du plasma est la longueur de Debye $\lambda_D = v_T/\omega_p$, au-delà de laquelle les particules du plasma démontrent un comportement collectif au lieu d'un comportement strictement individuel.

Le phénomène observé lors de cette simulation est illustré à la figure 7.3 : une instabilité du mouvement apparaît progressivement. Le mouvement uniforme des particules du plasma se transforme en une combinaison de mouvement uniforme et d'oscillation, menant à une inhomogénéité de la densité.

7.3.2 Annexe : code de simulation du plasma

Code 7.1 : Simulation du plasma 1D (`plasma.cpp`)

```
#include <iostream>
#include <fstream>
#include <cmath>

#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include "read_parameter.h"
#include "Vector.h"
#include "Random.h"

using namespace std;

#define GNUPLOT

#ifdef GNUPLOT
extern "C"{
#include "gnuplot_i.h"
}
#endif

int main(){

  fstream fin("para.dat");

  int log2M;  fin >> "log2M" >> log2M; logarithme en base 2 du nombre de points de grille
  int M = 1 << log2M; nombre de points = 2^log2M
  int N;  fin >> "N" >> N; nombre de particules
  double L;  fin >> "L" >> L; longueur de l'intervalle périodique
  double h;  fin >> "h" >> h; pas temporel
  double T;  fin >> "T" >> T; temps de simulation
  double v0;  fin >> "v0" >> v0; vitesse caractéristique des faisceaux opposés
  double dv0;  fin >> "dv0" >> dv0; écart-type de la distribution des vitesses
  double gamma;  fin >> "gamma" >> gamma; amortissement
  double dt_display; fin >> "intervalle_affichage" >> dt_display; intervalle entre les
      affichages

  double a = L/M; pas de réseau
  double ia = 1.0/a; pas de réseau inverse
  Vector<double> rho(M); tableau contenant la densité normalisée
  Vector<double> phi(M); tableau contenant le potentiel électrique normalisé
  Vector<double> E(M); tableau contenant le champ électrique normalisé
  Vector<double> x(N); tableau contenant les positions
  Vector<double> v(N); tableau contenant les vitesses
```

```
45    Vector<double> F(N); tableau contenant les accélérations
46
47    initialisation des positions et des vitesses des particules
48    gsl_rng *R;
49    R = gsl_rng_alloc(gsl_rng_default);
50    for(int i=0; i<N; i++){
51      x[i] = gsl_rng_uniform(R)*L;
52      v[i] = (1-2*(i%2))*v0 + gsl_ran_gaussian(R,dv0);
53    }
54
55    ofstream gout;
56  #ifdef GNUPLOT
57      gnuplot_ctrl * GP;
58      GP = gnuplot_init() ;
59      gnuplot_cmd(GP,(char *)"set term x11");
60      gnuplot_cmd(GP,(char *)"set xr [%g:%g]",0,L);
61      gnuplot_cmd(GP,(char *)"set yr [%g:%g]",-2.5*v0,2.5*v0);
62      gnuplot_cmd(GP,(char *)"set nokey");
63  #endif
64
65    boucle sur les temps
66    int output_step=0; compteur des sorties sur fichier
67    for(double t=0.0; t<=T; t+=h){
68
69      calcul de la densité d'électrons
70      rho.clear();
71      double LNaa = L/(N*a*a);
72      for(int i=0; i<N; i++){
73        int j = x[i]*ia; conversion implicite d'un réel en entier
74        assert(j<M and j>= 0);
75        int jn = ((j+1)+M)%M;
76        rho[j] += ((j+1)*a-x[i])*LNaa;
77        rho[jn] += (x[i]-j*a)*LNaa;
78      }
79
80      calcul du potentiel électrique par TdFR
81      phi = rho;
82      gsl_fft_real_radix2_transform(phi.array(), 1, M); transformée directe
83      phi[0] = 0.0; condition de neutralité
84      int m = M/2;
85      double k0 = 2.0*M_PI/L;
86      for(int i=1; i<=m; i++){ solution de l'équation de Poisson
87        double z = i*k0;
88        z = 1.0/(z*z);
89        phi[i] *= z;
90        phi[M-i] *= z;
91      }
92      gsl_fft_halfcomplex_radix2_inverse(phi.array(),1,M); transformée inverse
93
```

```
94      calcul du champ électrique
95      int M1 = M-1;
96      for(int i=1; i<M1; i++) E[i] = -0.5*ia*(phi[i+1]-phi[i-1]);
97      E[0] = -0.5*ia*(phi[1]-phi[M1]);
98      E[M1] = -0.5*ia*(phi[0]-phi[M1-1]);
99
100     propagation des particules (méthode de Verlet)
101     double z = 0.5*h*h;
102     double z1 = 0.5*h;
103     for(int i=0; i<N; i++){
104       x[i] += v[i]*h + F[i]*z; avancement des positions
105       if(x[i] > L) x[i] -= L; imposition des conditions aux limites périodiques
106       if(x[i] < 0.0) x[i] += L;
107
108       double X = x[i]; calcul de la force
109       int j = X*ia;
110       assert(j<M and j>= 0);
111       int jn = ((j+1)+M)%M;
112       double F0 = F[i];
113       F[i] = ia*(((j+1)*a-X)*E[j] + (X-j*a)*E[jn]); interpolation linéaire
114       v[i] += (F[i]+F0)*z1 -v[i]*(h*gamma); calcul de la vitesse au temps t+h
115     }
116
117     affichage périodique des résultats
118     if(floor(t/dt_display) > output_step){
119       output_step++;
120 #ifdef GNUPLOT
121       gout.open("out.dat");
122       for(int i=0; i<N; i++) gout << x[i] << '\t' << v[i] << endl;
123       gout << "\n\n";
124       gout.close();
125       gnuplot_cmd(GP,(char *)"set title 't = %2.3f'",t);
126       gnuplot_cmd(GP,(char *)"plot 'out.dat' w d",t);
127       usleep(50000);
128 #endif
129     }
130
131   }
132   gout.open("out.dat");
133   for(int i=0; i<N; i++) gout << x[i] << '\t' << v[i] << endl;
134   gout.close();
135
136   cout << "Fin normale du programme\n";
137 }
```

Chapitre 8

Équations non linéaires et optimisation

8.1 Équations non linéaires à une variable

Le problème traité dans cette section consiste à résoudre une équation non linéaire, ce qui revient à chercher la ou les racines d'une fonction $y(x)$:

$$y(x) = 0 \tag{8.1}$$

La fonction $y(x)$ peut avoir une forme analytique explicite ou être le résultat d'un autre calcul numérique ne correspondant pas à une forme connue.

8.1.1 Cadrage et bissection

Une méthode simple et robuste pour trouver une racine de f consiste premièrement à *cadrer* la racine, c'est-à-dire à trouver deux points x_1 et x_2 entre lesquels au moins une racine existe. Cela ne peut se faire que si on fait l'hypothèse que la fonction est continue, de sorte que si $y_1 = y(x_1) < 0$ et $y_2 = y(x_2) > 0$, on a l'assurance qu'une racine $x^\star$ existe dans l'intervalle $[x_1, x_2]$. Nous n'expliquerons pas ici d'algorithme particulier pour trouver les valeurs x_1 et x_2 ; nous supposons plutôt que, selon la fonction désirée, ces valeurs peuvent se trouver sans trop de difficulté.

La *méthode de bissection* consiste à diviser l'intervalle de recherche en deux parties égales, et ce de manière répétée, jusqu'à ce que la largeur de l'intervalle soit comparable à la précision recherchée ϵ sur la position de la racine. Plus précisément :

1. On calcule $y(\bar{x})$, où $\bar{x} = (x_1 + x_2)/2$.
2. Si $y(\bar{x})$ est du même signe que x_1, alors on met à jour $x_1 \leftarrow \bar{x}$, sinon on fait $x_2 \leftarrow \bar{x}$ et on recommence à l'étape 1.
3. On arrête la procédure lorsque $x_2 - x_1 < \epsilon$.

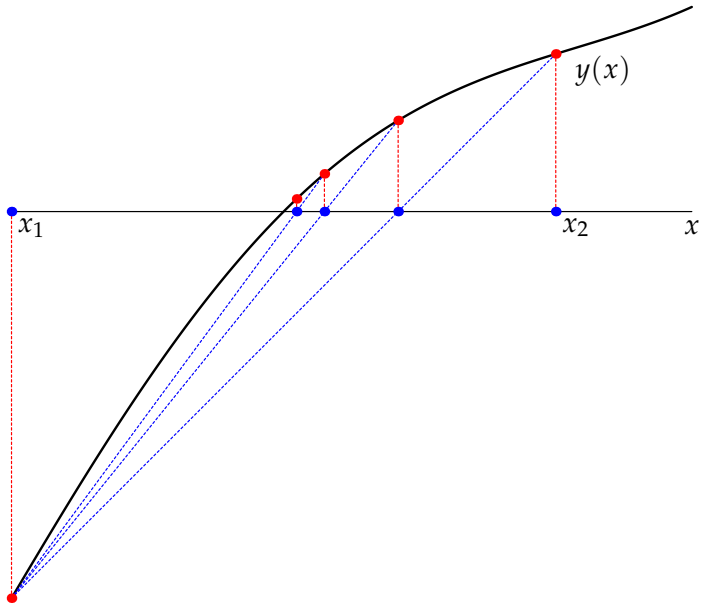


Figure 8.1 Méthode de la fausse position.

La méthode de bissection est sûre, mais inefficace : elle converge linéairement. Par là on veut dire que la différence $\delta_n \stackrel{\text{def}}{=} \bar{x} - x^\star$ entre la solution véritable et la meilleure estimation de la solution à l'étape n de la procédure se comporte comme

$$\delta_{n+1} \sim \frac{1}{2}\delta_n \tag{8.2}$$

En général, le *degré de convergence* η d'une méthode itérative est défini par

$$\delta_{n+1} \sim A\delta_n^\eta \tag{8.3}$$

où A est une constante ; plus l'exposant η est élevé, plus la convergence est rapide.

8.1.2 Méthode de la fausse position

Une amélioration sensible par rapport à la méthode de bissection est la méthode de la *fausse position*, qui consiste non pas à adopter la moyenne $\bar{x} = (x_1 + x_2)/2$ comme estimation de la racine à chaque étape, mais plutôt l'intersection avec l'axe des y de l'interpolant linéaire entre les deux points x_1 et x_2 :

$$\frac{\bar{x} - x_2}{x_1 - x_2} y_1 + \frac{\bar{x} - x_1}{x_2 - x_1} y_2 = 0 \implies \bar{x} = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1} \tag{8.4}$$

On met ensuite là jour les valeurs de x_1 et x_2 comme dans la méthode de bissection (voir la figure 8.1). La méthode de la fausse position est robuste, en ce sens que la racine est toujours cadrée et donc on ne peut pas la rater. Son degré de convergence n'est pas bien défini, mais la méthode converge en général plus rapidement que la méthode de bissection, cette dernière n'étant plus rapide que dans quelques cas pathologiques.

Exercice 8.1

Illustrez schématiquement une fonction dont la racine serait trouvée plus rapidement par la méthode de bissections que par la méthode de la fausse position.

8.1.3 Méthode de la sécante

La méthode de la sécante est une variation de la méthode de la fausse position, dans laquelle on procède à une interpolation ou extrapolation linéaire pour trouver une estimation de la position de la racine, même si cette nouvelle estimation x_{n+1} est en dehors de l'intervalle formé par les deux points précédents :

$$x_{n+1} = \frac{x_{n-1}y_n - x_n y_{n-1}}{y_n - y_{n-1}} \tag{8.5}$$

À la différence de la méthode de la fausse position, on base chaque nouvelle estimation x_{n+1} de la racine sur les deux estimations précédentes (x_n et x_{n-1}), au lieu de rejeter l'une des deux frontières de l'intervalle, qui une fois sur deux en moyenne correspond à x_{n-1}.

On montre que le degré de convergence de la méthode de la sécante est le nombre d'or :

$$\eta = \frac{1+\sqrt{5}}{2} = 1.618034\ldots \tag{8.6}$$

Par contre, sa convergence n'est pas garantie car la racine n'est pas systématiquement cadrée.

8.1.4 Méthode de Newton-Raphson

L'une des méthodes numériques les plus anciennes a été proposée indépendamment par I. Newton et J. Raphson à la fin du XVII^e siècle pour trouver les racines d'une fonction. Elle se base sur le calcul différentiel et non sur un cadrage de la racine (voir la figure 8.2) :

1. On choisit une estimation initiale x_1 de la racine.
2. On calcule la valeur de la fonction $y_1 = y(x_1)$ et de sa dérivée $y'_1 = y'(x_1)$.
3. On trace la droite de pente y'_1 passant par (x_1, y_1) et on trouve l'intersection de cette droite avec l'axe des x. Il s'agit de la prochaine valeur x_2 de l'estimateur :

$$x_2 = x_1 - \frac{y_1}{y'_1} \quad \text{ou, à l'étape } n, \quad x_{n+1} = x_n - \frac{y_n}{y'_n} \tag{8.7}$$

4. On répète jusqu'à convergence.

On montre que le degré de convergence de la méthode de Newton-Raphson est $\eta = 2$. Cela signifie qu'à l'approche de la solution, le nombre de chiffres significatifs de l'estimateur double à chaque résolution ! Par contre, la méthode n'est pas robuste : si on la démarre trop loin de

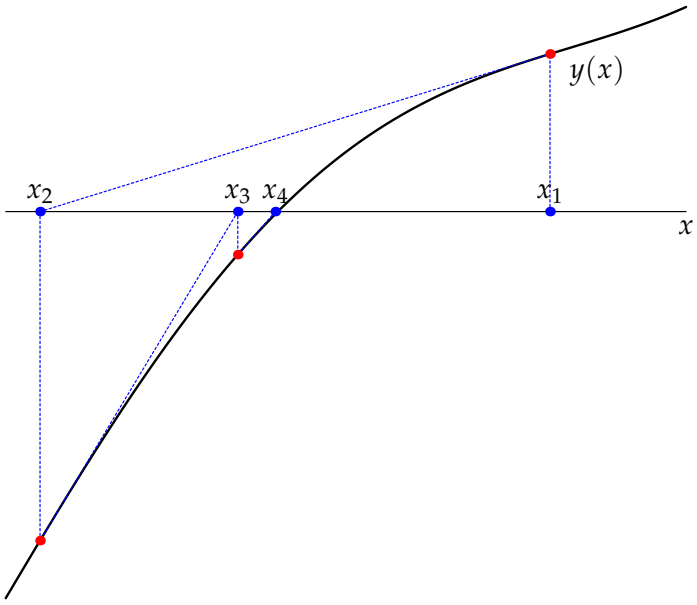


Figure 8.2 Méthode de Newton-Raphson pour la recherche des racines.

la solution, elle peut facilement diverger ou rester sur un cycle limite. La dynamique discrète définie par l'équation (8.7) peut être complexe et intéressante en soi.

Exercice 8.2

Montrez que le degré de convergenve de la méthode de Newton-Raphson est $\eta = 2$.

8.2 Équations non linéaires à plusieurs variable

La résolution d'un système d'équations non linéaires couplées est un problème pour lequel il n'existe pas de méthode sûre et générale. Considérons par exemple le cas de deux variables x et y obéissant à deux équations non linéaires simultanées qu'on peut toujours mettre sous la forme

$$f(x,y) = 0 \qquad g(x,y) = 0 \tag{8.8}$$

où f et g sont des fonctions qui définissent le problème.

L'équivalent de ces équations en dimension 1 est une équation unique $y(x) = 0$; si cette fonction est continue et que deux valeurs $y(x_1)$ et $y(x_2)$, l'une positive et l'autre négative, sont connues, alors une racine existe certainement. On ne peut affirmer l'existence d'une solution avec autant de généralité en dimension 2 : chacune des équations (8.8) définit une certaine courbe de niveau (une pour f, l'autre pour g). Une solution existe seulement si ces deux courbes de niveaux se croisent. Il est également possible que l'une ou l'autre des équations (8.8) n'ait pas de solution, c'est-à-dire que la courbe de niveau associée à la valeur 0 n'existe pas.

Une façon quelque peu brutale de tenter la solution des équations (8.8) est de les traiter comme une succession de problèmes unidimensionnels : On résoud la première équation en fonction de

x pour une valeur donnée de y, qui est alors traité comme un paramètre. L'un des algorithmes à une variable décrits ci-dessus est mis à contribution pour cela. On obtient ainsi une fonction $x(y)$, qu'on injecte ensuite dans la deuxième équation : $g(x(y), y) = 0$. Cette dernière équation est alors résolue, encore une fois par un algorithme à une variable. Cette approche a le mérite de la simplicité, mais est inefficace car elle ne traite pas les deux variables sur un pied d'égalité, et accordera trop de précision à des valeurs intermédiaires de y.

8.2.1 Méthode de Newton-Raphson

Nous allons décrire la méthode de Newton-Raphson appliquée à N équations à N variables, qu'on écrira comme

$$\mathbf{f}(\mathbf{x}) = 0 \quad \text{ou} \quad f_i(\mathbf{x}) = 0 \qquad i = 1, \ldots, N \tag{8.9}$$

Commençons par un point de départ $\mathbf{x}_0$. Autour de ce point les fonctions f_i admettent un développement de Taylor :

$$f_i(\mathbf{x}) = f_i(\mathbf{x}_0) + \sum_j \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}_0} \delta x_j + \mathcal{O}(\delta \mathbf{x}^2) \qquad\qquad \delta \mathbf{x} \stackrel{\text{def}}{=} \mathbf{x} - \mathbf{x}_0 \tag{8.10}$$

On note habituellement la matrice des dérivées premières (le jacobien) comme

$$J_{ij} \stackrel{\text{def}}{=} \frac{\partial f_i}{\partial x_j} \tag{8.11}$$

et donc l'équation non linéaire peut s'écrire, à cet ordre d'approximation, comme

$$0 = f_i(\mathbf{x}_0) + J_{ij}\delta x_j \quad \text{ou} \quad \mathbf{J}(\mathbf{x}_0)\delta \mathbf{x} = -\mathbf{f}(\mathbf{x}_0) \tag{8.12}$$

ce qui constitue un système linéaire simple qu'on résoud dans le but d'obtenir une nouvelle estimation de la racine cherchée :

$$\delta \mathbf{x} = -\mathbf{J}^{-1}(\mathbf{x}_0)\mathbf{f}(\mathbf{x}_0) \tag{8.13}$$

Répéter cette prodécure revient à poser la relation de récurrence suivante :

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n)\mathbf{f}(\mathbf{x}_n) \tag{8.14}$$

Cette relation est la généralisation à plusieurs variables de l'équation unidimensionnelle (8.7).

8.2.2 Méthode itérative directe

Il arrive dans plusieurs applications que les équations non linéaires aient la forme suivante :

$$\mathbf{x} = \mathbf{K}(\mathbf{x}|\alpha) \tag{8.15}$$

où α désigne un paramètre ou un ensemble de paramètres. Ceci n'est évidemment pas restrictif, car tout système non linéaire peut être mis sous cette forme. C'est le cas notamment dans l'approximation du champ moyen en physique statistique ou dans le problème à N corps en mécanique quantique. Les paramètres α dans ce cas pourraient être la température, ou encore la force d'une interaction entre particules. Ces équations pourraient bien sûr être traitées par la méthode de Newton-Raphson, avec la fonction $\mathbf{f}(\mathbf{x}) = \mathbf{K}(\mathbf{x}|\alpha) - \mathbf{x}$. Il est cependant fréquent de leur appliquer une méthode de solution beaucoup plus simple, qui suppose qu'on connaît à l'avance une solution dans un cas limite (par exemple pour une valeur précise α_0 des paramètres). On suppose alors que, si α n'est pas trop différent de α_0, on peut adopter comme première approximation la solution correspondante $\mathbf{x} = \mathbf{x}_0$ et appliquer la relation de récurrence suivante :

$$\mathbf{x}_{n+1} = \mathbf{K}(\mathbf{x}_n|\alpha) \tag{8.16}$$

L'application répétée de cette relation, si elle converge, mène effectivement à la solution recherchée. En pratique, si la solution est requise pour plusieurs valeurs de α, il est avantageux de procéder par *proximité*, en mettant sur pied une boucle sur α qui recycle la solution $\mathbf{x}(\alpha)$ de l'étape précédente comme point de départ de la nouvelle recherche pour la valeur suivante de α.

Cette méthode itérative directe est simple, mais converge moins rapidement que la méthode de Newton-Raphson.

Exercice 8.3

Appliquez cette méthode à la solution de l'équation transcendante $x = \frac{1}{4}e^x$, en adoptant comme point de départ $x = 0$ (utilisez une calculatrice ou Mathematica). Quel est le degré de convergence de cette méthode ?

Exercice 8.4

Supposons qu'on veuille appliquer cette méthode à la solution de l'équation $x = \lambda x(1-x)$, en adoptant comme point de départ $x = 0$ en en faisant progresser λ de 0 jusqu'à 4. Quel problème rencontrerions-nous ? L'application $x \mapsto \lambda x(1-x)$ porte le nom de *carte logistique*.

8.3 Optimisation d'une fonction

Les problèmes d'optimisation sont parmi les plus fréquents rencontrés en calcul scientifique. Dans plusieurs cas, il s'agit de trouver le minimum d'une fonction différentiable $E(\mathbf{x})$ à N variables. On peut alors utiliser notre connaissance des dérivées de cette fonction – ou la calculer numériquement – dans la recherche du minimum. D'autres problèmes, qui seront traités

dans la section suivante, ne peuvent pas être formulés en fonction d'une ou plusieurs variables continues, mais sont plutôt de nature discrète.

Dans tous les cas, la difficulté principale de l'optimisation est la recherche d'un minimum global, par opposition à un minimum local. Il n'y a pas de solution générale et certaine à ce problème.

8.3.1 Méthode de Newton-Raphson

Le minimum d'une fonction $E(\mathbf{x})$ doit respecter la condition de dérivée nulle :

$$\frac{\partial E}{\partial x_i} = 0 \qquad (i = 1, \ldots, N) \tag{8.17}$$

Le problème de minimisation se ramène dans ce cas à celui de la solution de N équations non linéaires couplées et peut être traité à l'aide de la méthode de Newton-Raphson. Il suffit de poser $\mathbf{f} = \nabla E$ (voir section 8.2.1). Il faut cependant remarquer que cette méthode trouvera aussi bien les maximums et les points d'inflexion que les minimums. Cela n'est pas trop mal en soi, car souvent c'est précisément ce qu'on cherche : plusieurs problèmes soi-disant de *minimisation* sont en réalité des problèmes où un principe physique nous demande de trouver des solutions à dérivée nulle plutôt que des minimums véritables. Cependant, si ce ne sont que les minimums qui nous intéressent, il faut alors vérifier, par calcul de la dérivée seconde, que c'est bien ce que nous avons trouvé.

Lorsque la forme des dérivées de la fonction E n'est pas connue explicitement, la méthode peut quand même être appliquée à condition de procéder à un calcul numérique des dérivées. Comme la méthode de Newton-Raphson requiert les dérivées premières de $\mathbf{f}$, les dérivées secondes de E doivent être estimées numériquement. Ainsi, une fonction de N variables à un point donné comporte 1 valeur, N dérivées premières et $\frac{1}{2}N(N+1)$ dérivées secondes, soit au total $1 + N(N+3)/2$ quantités qui doivent être estimées à chaque itération de la méthode. Le calcul numérique de ces quantité requiert donc un nombre égal de valeurs de E calculées au point $\mathbf{x}$ et dans son voisinage immédiat : il s'agit en pratique de lisser une forme quadratique sur ces valeurs de la fonction E.

8.3.2 Méthode de Powell

Lorsqu'elle converge, la méthode de Newton-Raphson le fait rapidement. Cependant, elle n'est pas robuste et la solution trouvée n'est pas nécessairement un minimum. La méthode de Powell, que nous allons maintenant décrire, corrige ces deux travers, au prix d'un plus grand nombre d'évaluations de la fonction E.

Approximation quadratique en dimension 1

La méthode de Powell repose sur la capacité de trouver le minimum d'une fonction $E(\mathbf{x})$ dans une direction donnée de l'espace, c'est-à-dire de résoudre un problème de minimisation à une variable. Si on désire minimiser une fonction à une variable $y(x)$, la stratégie générale est la suivante :

1. On doit premièrement *cadrer* le minimum, c'est-à-dire trouver trois points $x_1 < x_2 < x_3$ tels que $y(x_1) > y(x_2)$ et $y(x_3) > y(x_2)$.
2. Ensuite on lisse une parabole sur ces trois points (la solution est unique). Le minimum $x_{\text{min.}}$ de cette parabole est alors l'estimation suivante du minimum $x^\star$ de la fonction.
3. On doit conserver deux autres points, afin d'itérer la procédure. Si $x_{\text{min.}} < x_2$, alors x_1 et x_2 servent de cadre au minimum, sinon ce sont x_2 et x_3 : on procède aux substitutions suivantes :

$$x_{\text{min.}} < x_2 \ : \begin{cases} x_1 \leftarrow x_1 \\ x_2 \leftarrow x_{\text{min.}} \\ x_3 \leftarrow x_2 \end{cases} \qquad x_{\text{min.}} > x_2 \ : \begin{cases} x_1 \leftarrow x_2 \\ x_2 \leftarrow x_{\text{min.}} \\ x_3 \leftarrow x_3 \end{cases} \tag{8.18}$$

   On calcule ensuite $y(x_{\text{min.}})$ et on retourne à l'étape 2, et ainsi de suite jusqu'à convergence.
4. On arrête la procédure lorsque $x_{\text{min.}} \approx x^\star$ varie par une valeur inférieure à une précision ϵ.

Directions conjuguées

La méthode de Powell proprement dite comporte les étapes suivantes :

1. Choisir un point de départ $\mathbf{x}_0$ et une direction de départ $\mathbf{e}_0$.
2. Minimiser $E(\mathbf{x}_0 + \lambda \mathbf{e}_0)$ en fonction de λ, en utilisant la méthode décrite ci-dessus. On atteint ainsi un deuxième point $\mathbf{x}_1$.
3. Autour de ce point $\mathbf{x}_1$, on calcule les dérivées secondes, comme dans la méthode de Newton-Raphson. Ces dérivées secondes nous permettent de construire une approximation quadratique à la fonction E dans le voisinage du point $\mathbf{x}_1$:

$$E(\mathbf{x}) \approx E(\mathbf{x}_1) + \sum_j \frac{\partial E}{\partial x_j}(x_j - x_{1j}) + \frac{1}{2}\sum_{ij} \frac{\partial^2 E}{\partial x_i \partial x_j}(x_i - x_{1i})(x_j - x_{1j}) \tag{8.19}$$

   On traite ensuite la minimisation de cette fonction quadratique comme dans la méthode du gradient conjugué : on définit une direction conjuguée $\mathbf{e}_1$ à la direction qu'on vient de parcourir, au sens de la matrice hessienne $\partial^2 E / \partial x_i \partial x_j$.
4. On retourne à l'étape 2, cette fois avec la nouvelle direction $\mathbf{e}_1$, et on itère jusqu'à convergence sur la position (ou sur la valeur de la fonction, ou les deux).

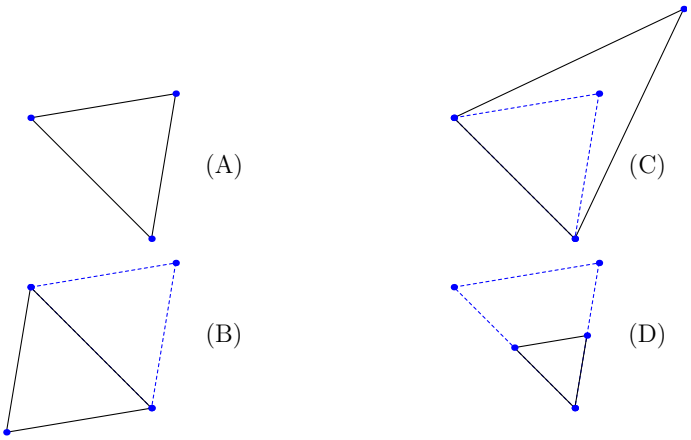


FIGURE 8.3 Différentes transformations d'un simplexe en deux dimensions : (A) le simplexe original ; (B) réflexion par rapport à une face ; (C) élongation d'un sommet par rapport à la face opposée ; (D) contraction du simplexe vers un sommet.

8.3.3 Méthode du simplexe descendant

S'il n'est pas possible d'utiliser l'expression des dérivées de la fonction E, ou si la forme de la fonction ne laisse pas espérer qu'une méthode basée sur des dérivées puisse avoir du succès, alors on peut se rabattre sur la méthode du *simplexe descendant* (angl. *downhill simplex*), proposée par Nelder et Mead.

Un *simplexe* en d dimensions est une figure géométrique de dimension d bornée par $d+1$ simplexes de dimension $d-1$. Par exemple :

1. Un simplexe de dimension 0 est un point.
2. Un simplexe de dimension 1 est un segment, borné par deux points.
3. Un simplexe de dimension 2 est un triangle, c'est-à-dire une portion de $\mathbb{R}^2$ bornée par trois segments.
4. Un simplexe de dimension 3 est un tétrahèdre irrégulier, c'est-à-dire une portion de $\mathbb{R}^3$ bornée par 4 triangles, et ainsi de suite.

L'idée générale de la méthode du simplexe descendant est de faire évoluer un simplexe dans l'espace $\mathbb{R}^d$ sur lequel est défini la fonction $E(\mathbf{x})$. Cette évolution se fait via différentes transformations du simplexe, notamment des réflexions et des contractions, qui amène progressivement le simplexe vers la région où la fonction $E(\mathbf{x})$ comporte un minimum local. Certaines de ces transformations sont illustrées à la figure 8.3. Il y a plusieurs variantes de la méthode du simplexe. L'une d'entre elles est décrite ci-dessous :

1. Choisir un simplexe de départ, comportant $n = d+1$ sommets en dimension d. Les positions sont notées $\mathbf{x}_1, \ldots, \mathbf{x}_n$.
2. Trier les sommets dans l'ordre croissant de la fonction f à minimiser : $f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \cdots < f(\mathbf{x}_n)$.
3. Calculer le centre de gravité $\mathbf{x}_0$ de tous les points sauf $\mathbf{x}_n$.

4. Calculer le point réfléchi $\mathbf{x}_r = \mathbf{x}_0 + \alpha(\mathbf{x}_0 - \mathbf{x}_n)$. Ici α est le coefficient de réflexion, habituellement égal à 1. Si $f(\mathbf{x}_1) < f(\mathbf{x}_r) < f(\mathbf{x}_{n-1})$, alors remplacer $\mathbf{x}_n \to \mathbf{x}_r$ et retourner à l'étape 2.
5. Si, au contraire, le point réfléchi est le meilleur à date, c'est-à-dire si $f(\mathbf{x}_r) < f(\mathbf{x}_1)$, alors calculer le point étiré $\mathbf{x}_e = \mathbf{x}_0 + \gamma(\mathbf{x}_0 - \mathbf{x}_n)$, où γ est le coefficient d'élongation, habituellement égal à 2. Si $f(\mathbf{x}_e) < f(\mathbf{x}_r)$, alors remplacer $\mathbf{x}_n \to \mathbf{x}_e$ et retourner à l'étape 2. Sinon remplacer $\mathbf{x}_n \to \mathbf{x}_r$ et retourner à l'étape 2.
6. Si le point réfléchi n'est pas meilleur que $\mathbf{x}_n$ (c'est-à-dire si $f(\mathbf{x}_r) > f(\mathbf{x}_{n-1})$, alors calculer le point contracté $\mathbf{x}_e = \mathbf{x}_0 + \rho(\mathbf{x}_0 - \mathbf{x}_n)$, où ρ est le coefficient de contraction, habituellement 0.5. Si $f(\mathbf{x}_c) < f(\mathbf{x}_n)$, alors remplacer $\mathbf{x}_n \to \mathbf{x}_c$ et retourner à l'étape 2.
7. Sinon, alors contracter le simplexe en remplaçant $\mathbf{x}_i \to \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1)$ pour tous les points sauf le meilleur ($\mathbf{x}_1$). Ensuite retourner à l'étape 2. Le coefficient σ (deuxième coefficient de contraction) est typiquement égal à 0.5.
8. Une condition de convergence est requise. Par exemple, la valeur

$$r = \frac{|f(\mathbf{x}_n) - f(\mathbf{x}_1)|}{|f(\mathbf{x}_n) + f(\mathbf{x}_1)| + \epsilon} \tag{8.20}$$

peut être calculée à l'étape 2 et le programme terminé si cette valeur est inférieure à ϵ, une précison choisie à l'avance.

8.4 Lissage d'une fonction

8.4.1 Méthode des moindres carrés et maximum de vraisemblance

Supposons que nous disposions d'une modèle pour décrire une certaine quantité y, qui prend la forme d'une fonction $y(x|a)$, où x représente une variable sur laquelle nous avons le contrôle et a est un ensemble de paramètres du modèle qu'on cherche à déterminer. On suppose en outre qu'un ensemble de mesures entachées d'erreurs a produit N observations (x_i, y_i), avec une erreur σ_i sur la valeur de y_i ($i = 1, \ldots, N$). Le problème décrit dans cette section consiste à estimer les meilleures valeurs possibles des paramètres a_j ($j = 1, \ldots, M$) qui découlent de ces observations.

Le principe de base que nous allons suivre consiste à *maximiser la vraisemblance*, c'est-à-dire à trouver les valeurs a_j les plus probables. Pour cela, nous devons définir une probabilité d'observer les y_i, étant données des valeurs de a, c'est-à-dire étant donné le modèle. Nous allons supposer que cette probabilité suit une loi gaussienne, c'est-à-dire qu'elle prend la forme suivante :

$$P(\text{observations}|a) = \prod_{i=1}^{N} \exp\left[-\frac{1}{2}\left(\frac{y_i - y(x_i|a)}{\sigma_i}\right)^2\right] \Delta y \tag{8.21}$$

où $\sigma_i = \sigma(x_i)$ est un écart-type qui dépend de x et qui est lié au processus de mesure lui-même ; Δy est un intervalle conventionnel de y nécessaire ici parce que nous avons affaire à une densité de probabilité (cet intervalle doit être petit en comparaison de σ_i). L'emploi d'une

loi gaussienne se justifie en supposant que le processus de mesure est perturbé par une suite d'événements non corrélés et de variances identiques, dont la résultante, en vertu de la loi des grands nombres, suit une distribution gaussienne.

La probabilité (8.21) est une *probabilité conditionnelle*, c'est-à-dire qu'elle exprime la probabilité d'un événement A (les observations) étant donné la certitude sur l'événement B (le modèle). Or ce qui nous intéresse ici est l'inverse : quelle est la probabilité du modèle B étant donnée une certitude sur les observations A : $P(B|A)$. Ces deux probabilités sont reliées par le théorème de Bayes :

$$P(A \cap B) = P(A|B)P(B) = P(B|A)P(A) \quad \text{ou encore} \quad P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{8.22}$$

Autrement dit : la probabilité que A et B se produisent est la probabilité de A étant donné B, fois la probabilité de B. Ceci est manifestement identique à la probabilité de B étant donné A, fois la probabilité de A. Ce théorème est donc extrêmement simple, en dépit des apparences.

Donc, appliqué à notre problème, ce théorème stipule que :

$$P(a|\text{observations}) = P(\text{observations}|a)P(a)/P(\text{observations}) \tag{8.23}$$

Nous allons supposer ici que la probabilité absolue du modèle est indépendante de a, c'est-à-dire que toutes les valeurs de a sont a priori équiprobables, en l'absence d'observation. Donc seul le premier facteur du membre de droite dépend de a. Nous devons chercher les valeurs de a_j qui maximisent cette probabilité, ou qui minimisent l'opposé de son logarithme, ce qui veut dire minimiser l'expression suivante :

$$\chi^2 = \sum_i \left(\frac{y_i - y(x_i|a)}{\sigma_i} \right)^2 \tag{8.24}$$

En conclusion : la méthode du maximum de vraisemblance, alliée à quelques hypothèses générales, nous dicte la manière d'inférer les paramètres a_j : il faut minimiser la somme des écarts au carré entre les observations et le modèle, somme pondérée par l'erreur σ. Ceci est accompli en annulant les dérivées premières par rapport à a_j, c'est-à-dire en solutionnant les M équations suivantes :

$$\sum_i \frac{y_i - y(x_i|a)}{\sigma_i^2} \frac{\partial y(x_i|a)}{\partial a_j} = 0 \qquad (j = 1, \ldots, M) \tag{8.25}$$

8.4.2 Combinaisons linéaires de fonctions de lissage

Les équations (8.25) sont en général difficiles à résoudre, car la fonction (8.24) est en général non linéaire en fonction des paramètres a_j. Par contre, le cas linéaire est assez fréquent et simple : le modèle est une superpositions de fonctions $Y_j(x)$, dont les coefficients sont les paramètres a_j :

$$y(x) = \sum_j a_j Y_j(x) \tag{8.26}$$

Les équations (8.25) deviennent alors

$$\sum_i \frac{1}{\sigma_i^2}\left(y_i - \sum_k a_k Y_k(x_i)\right) Y_j(x_i) = 0 \qquad (j = 1, \ldots, M) \tag{8.27}$$

ce qui peut également s'écrire sous la forme concise suivante :

$$\sum_j \alpha_{jk} a_k = \beta_j \quad \text{où} \quad \alpha_{jk} = \sum_i \frac{1}{\sigma_i^2} Y_j(x_i) Y_k(x_i) \quad \text{et} \quad \beta_j = \sum_i \frac{y_i}{\sigma_i^2} Y_j(x_i) \tag{8.28}$$

L'équation de gauche est un système linéaire qui se résoud par les méthodes matricielles usuelles. On l'appelle le *système des équations normales* pour le problème du lissage de la fonction $y(x|a)$. [1]

Si on définit la matrice $N \times M$

$$A_{ij} = \frac{Y_j(x_i)}{\sigma_i} \tag{8.29}$$

et le vecteur colonne à N composantes $b_i = y_i/\sigma_i$, alors les quantités α et β s'expriment comme suit :

$$\alpha = \widetilde{A}A \qquad \beta = \widetilde{A}b \tag{8.30}$$

La matrice inverse $C = \alpha^{-1}$ est appelée *matrice de covariance* et donne accès aux incertitudes sur les valeurs des paramètres inférés a_j. En effet, ces paramètres sont

$$a_j = \sum_k C_{jk}\beta_k = \sum_k \sum_i C_{jk} \frac{y_i Y_k(x_i)}{\sigma_i^2} \tag{8.31}$$

or la variance $\sigma^2(a_j)$ se trouve par la propagation normale des erreurs de la variable y_i vers la variable a_j (notons que la matrice α est indépendante des y_i) :

$$\begin{aligned}
\sigma^2(a_j) &= \sum_i \sigma_i^2 \left(\frac{\partial a_j}{\partial y_i}\right)^2 \\
&= \sum_i \sigma_i^2 \left(\sum_k \frac{C_{jk} Y_k(x_i)}{\sigma_i^2}\right)^2 \\
&= \sum_{l,k} \sum_i \frac{1}{\sigma_i^2} C_{jk} C_{jl} Y_k(x_i) Y_l(x_i) \\
&= \sum_{l,k} C_{jk} C_{jl} \alpha_{kl} \\
&= \sum_l \delta_{jl} C_{jl} = C_{jj}
\end{aligned} \tag{8.32}$$

Nous avons utilisé la relation $\alpha^{-1} = C$ dans la dernière équation. En somme, l'élément diagonal no j de la matrice de covariance est la variance associée au paramètre a_j, suite à la propagation des erreurs σ_j.

1. La solution du système linéaire peut être délicate, cependant, car le problème est souvent proche d'un système singulier et une méthode du genre SVD (décomposition en valeurs singulières) est recommandée. Voir *Numerical Recipes* à cet effet.

Qu'arrive-t-il si on ne connaît pas les erreurs σ_i ? Dans ce cas on peut formellement prétendre que toutes les erreurs σ_i sont égales à une constante σ. Les valeurs des paramètres tirées de l'équation (8.31) seront alors manifestement indépendantes de σ : la matrice α comporte un facteur $1/\sigma^2$ et la matrice C comporte donc le facteur inverse, qui annule le $1/\sigma^2$ apparaissant dans l'équation.

8.4.3 Lissages non linéaires

Si la fonction à lisser comporte un ou plusieurs paramètres qui apparaissent de manière non linéaire, le problème est plus difficile, et doit être traité par une méthode de minimisation du χ^2 comme celles décrites plus haut, comme par exemple la méthode de Newton-Raphson. Cependant il est fréquent dans ce contexte d'utiliser la méthode de *Levenberg-Marquardt*, que nous expliquons sommairement dans ce qui suit. Cette méthode requiert la connaissance des premières et deuxièmes dérivées de la fonction à minimiser (le χ^2 de l'éq. (8.24)). Proche du minimum de la fonction χ^2, on peut faire l'approximation quadratique et écrire

$$\chi^2(a) = \text{cte} - \widetilde{d}a + \frac{1}{2}\widetilde{a}Da \tag{8.33}$$

où D est la matrice hessienne de χ^2 (la matrice des deuxièmes dérivées). Dans cette approximation, la solution est immédiate :

$$a \to a + D^{-1}\left(-\nabla\chi^2(a)\right) \tag{8.34}$$

Par contre, cette approche n'est bonne que si on n'est pas trop loin de la solution. Si on est plus éloigné, alors il est plus efficace de faire le pas suivant, le long du gradient de la fonction :

$$a \to a - \gamma\nabla\chi^2(a) \tag{8.35}$$

où la constante γ est suffisamment petite. Ceci équivaut à adopter un hessien diagonal. La méthode de Levenberg-Marquardt combine ces deux approches, en utilisant la deuxième initialement, et en se rapprochant de la première au fur et à mesure qu'on approche de la solution : on définit la matrice

$$\alpha_{ij} = \frac{1}{2}\frac{\partial^2\chi^2}{\partial a_i \partial a_j} \quad \text{ainsi que} \quad \beta_k = -\frac{1}{2}\frac{\partial\chi^2}{\partial a_k} \tag{8.36}$$

On définit ensuite un matrice modifiée

$$\alpha'_{ij} = \alpha_{ij} + \lambda\delta_{ij} \tag{8.37}$$

où λ est une constante qui est initialement petite ($\lambda \sim 0.001$). La méthode consiste alors à solutionner le système linéaire

$$\alpha'\delta a = \beta \implies \delta a = (\alpha')^{-1}\beta \tag{8.38}$$

à répétition, en suivant la prescription suivante :

1. Choisir une valeur initiale a des paramètres et calculer $\chi^2(a)$.

2. Choisir une petite valeur de λ (ex. 0.001) et calculer α'.
3. Calculer δa selon l'éq. (8.38). Sortir de la boucle si δa est suffisamment petit.
4. Si $\chi^2(a+\delta a) \geq \chi^2(a)$, augmenter λ d'un facteur important (ex. 10) et retourner à l'étape 3. Sinon, diminuer λ par un facteur important (ex. 10), mettre à jour $a \to a+\delta a$ et retourner à l'étape 3.

La méthode de Levenberg-Marquardt est utilisée par la plupart des programmes de lissage, incluant par la commande `fit` de gnuplot. Dans ce cas aussi, les éléments diagonaux de la matrice de covariance $C = \alpha^{-1}$ sont les variances des paramètres obtenus, compte tenu des erreurs σ_i.

8.5 La méthode du recuit simulé

Une grande partie des problèmes d'optimisation vise à minimiser une fonction $E(x)$ définie sur un espace Ω de configurations discret, sur lequel la notion de continuité n'existe pas. Le problème le plus connu de ce genre est celui du *commis-voyageur* : un voyageur de commerce doit visiter N villes dont les positions sont connues, tout en minimisant le chemin parcouru au total. On peut supposer pour simplifier l'argument qu'il peut parcourir la distance entre chaque paire de villes selon une ligne droite, mais cela n'est pas essentiel à la formulation du problème. Le voyageur doit donc choisir un itinéraire, qui prend la forme d'une permutation p_i de la liste des villes ($i = 1, \ldots, N$), qu'il devra parcourir dans cet ordre, en commençant et en terminant par la même ville (la trajectoire est fermée). La figure 8.4 illustre un exemple de chemin minimal reliant $N = 36$ points.

Ce problème ne peut pas être résolu par les méthode classiques d'optimisation, car la variable sur laquelle nous devons minimiser est une permutation p, et non une variable continue. D'un autre côté, la simple énumération de toutes les possibilités est hors de question, car le nombre d'itinéraires possibles est $N!$, un nombre qui défie l'imagination même pour une valeur modérée de N, comme 36. [2]

La méthode de choix pour résoudre ce type de problème est l'algorithme du *recuit simulé* (angl. *simulated annealing*). [3] L'idée générale est d'explorer différentes configurations par l'algorithme de Métropolis, et de diminuer progressivement la température T jusqu'à un minimum proche de zéro. Autrement dit, on procède à une marche aléatoire dans l'espace des configurations Ω. Cette marche favorise les configurations de basse «énergie» E, mais permet tout de même de passer pendant un certain temps à des configurations d'énergie plus grande, quand la température est suffisamment élevée. Cet aspect est crucial, car il permet de surmonter les barrières de potentiel qui entourent souvent les minimums locaux (l'adjectif «local»

2. $36! = 371\,993\,326\,789\,901\,217\,467\,999\,448\,150\,835\,200\,000\,000 \sim 3.10^{41}$.
3. En métallurgie, le *recuit* est un procédé par lequel un alliage est porté à haute température et ensuite refroidi lentement, ce qui permet aux défauts cristallins de se propager et d'être évacués, par opposition à la *trempe*, qui est un refroidissement soudain du matériau, qui gèle sur place les différents défauts cristallins et augmente la dureté de l'alliage.

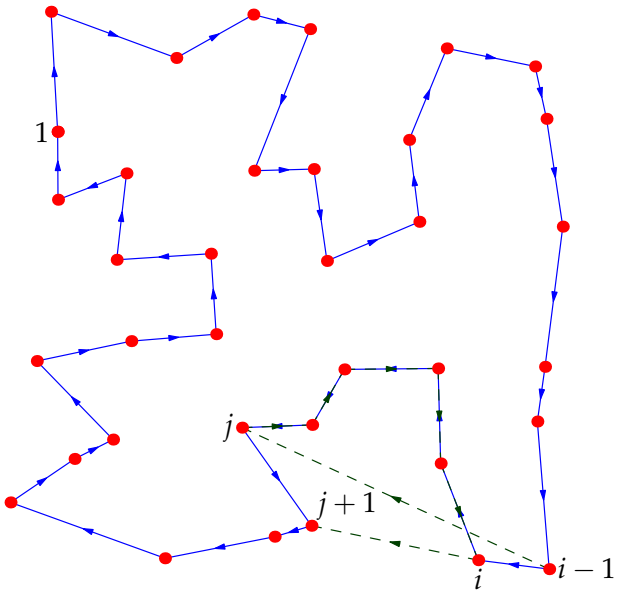


FIGURE 8.4 Le problème du commis-voyageur : comment relier N points (ou «villes») par un chemin qui passe au moins une fois par chaque ville et minimise la distance totale parcourue. La portion pointillée illustre un changement local qu'il est possible d'apporter à l'itinéraire dans le recherche d'un minimum de la distance totale.

doit être correctement interprété dans le contexte d'un espace Ω où la notion de distance n'est pas définie de manière évidente).

La méthode du recuit simulé requiert donc les ingrédients suivants :

1. Une définition claire de ce que constitue une configuration. Dans le cas du problème du commis-voyageur, une configuration est une permutation p de la liste des villes à visiter.
2. Une fonction de type «énergie» $E(p)$, qu'on désire minimiser. Dans le problème du voyageur, c'est la longueur de l'itinéaire :
$$E(p) = \sum_{i=0}^{N-1} d(p_i, p_{i+1}) \tag{8.39}$$
où $d(j,k)$ est la distance entre les villes j et k, et p_i est l'indice no i de la permutation p. Il est sous-entendu que les indices sont traités de manière périodique, c'est-à-dire modulo N (l'indice $N+i$ étant considéré synonyme de i).
3. Une procédure de changement local, l'équivalent du renversement local du spin dans le modèle d'Ising. Dans le problème du voyageur, une procédure possible est de sélectionner un indice i au hasard, ainsi qu'un deuxième indice j également au hasard, mais suivant une distribution exponentielle en fonction de la séparation $|j-i|$, de sorte que les paires (i,j) d'indices proches sont plus probables que les paires éloignées. Ensuite, on sectionne le chemin entre les indices $i-1$ et i, et entre les indices j et $j+1$ et on recolle la portion du chemin comprise entre i et j dans l'autre sens, c'est-à-dire de $i-1$ à j, ensuite $j-1$, etc. jusqu'à i et ensuite $j+1$. Cette procédure est illustrée par les traits pointillés sur la figure 8.4.

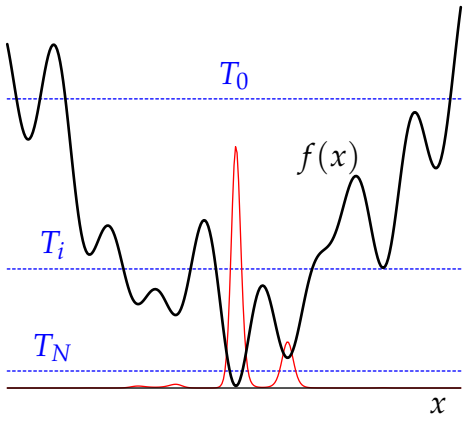


Figure 8.5 Application de la méthode du recuit simulé à la minimisation d'une fonction à une variable $f(x)$ possédant plusieurs minimums locaux. En rouge : probabilité $\mathrm{e}^{-f(x)/T}$ d'une configuration x à la température finale T_N.

4. Un mode de refroidissement, c'est-à-dire une séquence de températures décroissantes, entrecoupées d'une série de changements locaux à chaque température. Le mode choisi dans le code décrit ci-dessous est caractérisé par une température initiale T_1, une température finale T_2, un facteur $1-\epsilon$ par lequel la température est multipliée à chaque étape, et un nombre m de mises-à-jour de la configuration effectué à chaque valeur de T.

Au total, un code de recuit simulé ressemble beaucoup, mais en plus simple, à un code de simulation de physique statistique dans l'ensemble canonique. Il n'y a pas lieu, cependant, de procéder à une analyse d'erreur, car c'est l'état fondamental du système qui nous intéresse et non une moyenne statistique. Ceci dit, il n'y a aucune garantie que l'algorithme va converger vers le minimum absolu et non vers un minimum local. Il est cependant très peu probable que ce minimum local soit mauvais, au sens que son «énergie» E soit élevée. Donc, même si le minimum absolu n'est pas garanti, l'algorithme est tout de même extrêmement utile car le résultat qu'il produit est toujours intéressant d'un point de vue pratique. Dans tous les cas, l'algorithme est rapide, mais demande une certaine exploration des paramètres afin d'optimiser sa convergence vers un minimum de qualité ; plusieurs simulations répétées (avec des configurations initiales aléatoires différentes) sont à conseiller, afin de valider le point d'arrivée.

Application à une fonction ordinaire

Rien n'empêche d'utiliser la méthode du recuit simulé lors de la minimisation d'une fonction définie sur les réels. Dans ce cas, la mise à jour de la configuration consiste simplement en un déplacement δx dans une direction aléatoire dont la grandeur est, par exemple, distribuée uniformément dans un certain invervalle. Le mérite de la méthode dans ce cas est sa capacité à trouver souvent le minimum global de la fonction et non un minimum local. Voir la figure 8.5 pour un exemple unidimensionnel : la température initiale T_0 est suffisamment élevée pour permettre une exploration de l'espace dans un domaine large où plusieurs minimums relatifs existent. À mesure que la température diminue, la distribution de probabilité régissant l'algorithme de Metropolis devient de plus en plus concentrée autour du minimum global, de

sorte que ce minimum est celui qui est atteint le plus souvent à la fin de la simulation.

8.5.1 Annexe : code du recuit simulé pour le problème du commis-voyageur

Code 8.1 : Recuit simulé (`recuit.cpp`)

```
#include <iostream>
#include <fstream>

#include "read_parameter.h"
#include "Random.h"
#include "Matrix.h"
#include "Vector2D.h"

#define GNUPLOT

#ifdef GNUPLOT
extern "C"{
#include "gnuplot_i.h"
}
#endif

Random R;

using namespace std;

classe décrivant la configuration dans le problème du commis-voyageur
class voyageur{
public:
  int n; nombre de villes
  double E; longueur du chemin courant
  Vector<Vector2D>x; position des villes
  Vector<int> I; ordre de position des villes (permutation de la séquence 0..n-1)
  Vector<int> I0; ordre de position des villes (permutation de la séquence 0..n-1)
  Matrix<double> dist;

  positions distribuées sur une grille régulière, avec bruit
  void init(int sn, double noise){
    n = sn*sn;
    x.Alloc(n);
    int k=0;
    for(int i=0; i<sn; i++){
      for(int j=0; j<sn; j++){
        x[k++] = Vector2D(i+noise*(2*R.uniform()-1),j+noise*(2*R.uniform()-1));
      }
    }
    consol();
```

```
42    }
43
44    lecture des positions à partir d'un fichier
45    void init(istream &fin){
46      fin >> n;
47      x.Alloc(n);
48      for(int i=0; i<n; i++) fin >> x[i];
49      consol();
50    }
51
52    conclusion de l'initialisation
53    void consol(){
54      I.Alloc(n);
55      for(int i=0; i<n; i++) I[i] = i; initialisation du chemin
56      int np = 5*n;
57      for(int i=0; i<np; i++){ mélange aléatoire
58        int i1 = R.integer()%n;
59        int i2 = R.integer()%n;
60        int tmp = I[i1];
61        I[i1] = I[i2];
62        I[i2] = tmp;
63      }
64
65      dist.Alloc(n);
66      for(int i=0; i<n; i++){
67        for(int j=0; j<i; j++){
68          dist(i,j) = dist(j,i) = (x[i]-x[j]).norm();
69        }
70      }
71      E = energie();
72      I0 = I;
73    }
74
75    calcul de la fonction "énergie" E(x)
76    double energie(){
77      double e=0.0;
78      for(int i=1; i<n; i++) e += dist(I[i],I[i-1]);
79      e += dist(I[0],I[n-1]);
80      return e;
81    }
82
83    mise à jour de la configuration à une température T
84    void update(double T){
85      int i0,i1,i2,i3;
86      i1 = R.integer()%n; choix aléatoire d'une première ville (indice i)
87      int diff = floor(-(n/5)*log(R.uniform())); choix d'une distance
88      i2 = (i1 + diff + n)%n; position de la deuxième ville (indice j)
89      if(i2==i1) return;
90      else if(i2<i1){
```

```
91          i0 = i1; i1 = i2; i2 = i0;
92        }
93        if(i2-i1 >= n-2) return;
94        i0 = (i1+n-1)%n; position de la ville d'incide i − 1
95        i3 = (i2+1)%n; position de la ville d'incide j + 1
96        différence d'énergie après la section et le recollage inversé
97        double deltaE = dist(I[i0],I[i2])+dist(I[i1],I[i3])-dist(I[i0],I[i1])-dist(I[i2
              ],I[i3]);
98
99        bool flip = false;
100
101       if(deltaE < 0) flip = true; le changement est accepté
102       else if(R.uniform() < exp(-deltaE/T)) flip = true; le renversement est accepté
103
104       if(flip){
105         E += deltaE;
106         recollage du segment sectionné
107         if(i1<i2) for(int i=i1; i<=i2; i++) I[i] = I0[i2+i1-i];
108         else for(int i=i2; i<=i1; i++) I[i] = I0[i2+i1-i];
109         I0 = I;
110       }
111     }
112
113     friend std::ostream & operator<<(ostream& flux, const voyageur& X){
114       for(int i=0; i<X.n; i++) flux << X.x[X.I[i]].x << '\t' << X.x[X.I[i]].y << '\n'
              ;
115        flux << X.x[X.I[0]].x << '\t' << X.x[X.I[0]].y << '\n';
116       return flux;
117     }
118
119     identifie les bornes d'un graphique des villes (positions maximales et minimales)
120     void bornes(Vector2D &x1, Vector2D &x2){
121       x2 = Vector2D(-1e6,-1e6);
122       x1 = Vector2D(1e6,1e6);
123       for(int i=0; i<n; i++){
124         if(x[i].x > x2.x) x2.x = x[i].x;
125         if(x[i].y > x2.y) x2.y = x[i].y;
126         if(x[i].x < x1.x) x1.x = x[i].x;
127         if(x[i].y < x1.y) x1.y = x[i].y;
128       }
129     }
130
131 };
132
133 int main() {
134
135   double T1,T2,epsilon,noise;
136   int n,m;
137   ofstream fout;
```

```
138   bool carre = false;
139
140   fstream fin("para.dat"); lecture des paramètres
141   fin >> "n" >> n;
142   fin >> "m" >> m;
143   fin >> "temperature_initiale" >> T1;
144   fin >> "temperature_finale" >> T2;
145   fin >> "epsilon" >> epsilon;
146   fin >> "noise" >> noise;
147   if(fin == "carre") carre = true;
148   fin.close();
149
150   assert(T1 > 0);
151   assert(T2 > 0 and T2 < T1);
152   assert(epsilon > 0.001 and epsilon < 0.5);
153   assert(m>0);
154
155   cout << "Recuit simulé" << endl;
156
157   voyageur X; création d'une instance du commis−voyageur
158   if(carre) X.init(n,noise);
159   else{
160     fin.open("villes.dat");
161     X.init(fin);
162     fin.close();
163   }
164
165   cout << "longueur initiale: " << X.E << endl;
166
167 #ifdef GNUPLOT
168   ofstream gout;
169   Vector2D x1,x2;
170   X.bornes(x1,x2);
171   gnuplot_ctrl * GP;
172   GP = gnuplot_init() ;
173   gnuplot_cmd(GP,(char *)"set term x11");
174   gnuplot_cmd(GP,(char *)"set xr [%f:%f]",x1.x-0.5,x2.x+0.5);
175   gnuplot_cmd(GP,(char *)"set yr [%f:%f]",x1.y-0.5,x2.y+0.5);
176   gnuplot_cmd(GP,(char *)"unset tics");
177   gnuplot_cmd(GP,(char *)"set size ratio 1");
178   gnuplot_cmd(GP,(char *)"set nokey");
179 #endif
180
181   boucle de refroidissement
182   for(double T=T1; T > T2; T *= (1-epsilon)){
183     for(int i=0; i<m; i++) X.update(T); m mises à jour par valeur de T
184 #ifdef GNUPLOT
185     fout.open("tmp.dat");
186     fout << X << endl;
```

```
187       fout.close();
188       gnuplot_cmd(GP,(char *)"set title 'T = %2.3f, L = %1.3f'",T,X.E);
189       gnuplot_cmd(GP,(char *)"plot 'tmp.dat' w lp pt 5 ps 2 lw 2");
190       usleep(5000);
191   #endif
192     }
193   #ifdef GNUPLOT
194     sleep(5);
195   #endif
196
197     cout << "longueur finale: " << X.E << " = " << X.energie() << endl;
198
199     fout.open("out.dat");
200     fout << X << endl;
201     fout.close();
202
203     cout << "Fin normale du programme\n";
204   }
```

Annexe A

Opérations matricielles

L'algèbre linéaire est au coeur des méthodes numériques en science, même dans l'étude des phénomènes non linéaires. Si on pouvait recenser les cycles de calcul sur tous les ordinateurs de la planète, il est probable qu'une fraction proche de l'unité serait dédiée à des calculs impliquant des matrices. Il est donc important de survoler les méthodes utilisées pour procéder aux deux opérations les plus importantes de l'algèbre linéaire : la résolution des systèmes d'équations linéaires, et le calcul des valeurs et vecteurs propres.

A.1 Systèmes d'équations linéaires

A.1.1 Système général et types de matrices

Le problème de base de l'algèbre linéaire est la solution d'un système d'équations linéaires :

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \quad \text{ou encore} \quad Ax = b \tag{A.1}$$

Nous avons supposé que le nombre d'inconnues est égal au nombre d'équations, et donc que la matrice est carrée. Si ce n'est pas le cas, le système est soit sur déterminé (aucune solution exacte n'existe), ou sous-déterminé (certaines variables doivent être déterminées autrement).

La méthode indiquée pour résoudre un tel système dépend de la configuration de la matrice A. Par exemple, on distingue les types suivantes :

matrice pleine La matrice ne comporte aucune catégorie d'éléments ou disposition particulière : ses éléments sont en général non nuls.

matrice triangulaire supérieure Les éléments situés au dessous de la diagonale sont nuls.

matrice triangulaire inférieure Les éléments situés au dessus de la diagonale sont nuls.

matrice tridiagonale Les éléments non nuls sont situés sur la diagonale et sur les deux diagonales voisines (la matrice possède donc trois diagonales).

matrice de bande Les éléments non nuls sont situés sur la diagonale ainsi que sur $\ell - 1$ diagonales de chaque côté de la diagonale. On dit alors que la matrice de bande est de demi-largeur ℓ.

matrice creuse La vaste majorité des élments sont nuls, mais les éléments non nuls sont dispersés un peu partout. Une telle matrice doit être stockée de manière à que seuls les éléments non nuls soient repérés (plusieurs schémas de stockage sont possibles).

Système triangulaire

Le système (A.1) est très simple à résoudre si la matrice A est triangulaire. Par exemple, si elle est triangulaire supérieure, le système se résoud trivialement à partir du bas, par *rétro-substitution* (ou *substitution vers l'arrière*) :

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}} \\ x_{n-1} &= \frac{1}{a_{n-1,n-1}} \left(b_{n-1} - a_{n-1,n} x_n\right) \\ x_{n-2} &= \frac{1}{a_{n-2,n-2}} \left(b_{n-2} - a_{n-2,n-1} x_{n-1} - a_{n-2,n} x_n\right) \\ &\text{etc}\ldots \end{aligned} \tag{A.2}$$

La solution n'existe manifestement que si tous les éléments diagonaux a_{ii} sont non nuls.

Si la matrice est triangulaire inférieure, la solution est également simple, par substitution vers l'avant (c'est-à-dire en commençant par x_1), comme on le voit aisément.

A.1.2 Élimination gaussienne

Supposons maintenant que nous ayons affaire à une matrice pleine. L'algorithme de base pour résoudre le système (A.1) est l'*élimination gaussienne*. Il consiste à soustraire de chaque équation (rangée i) une combinaison des équations précédentes (rangées $< i$) de manière à ramener le système à une forme triangulaire supérieure. Une fois ceci accompli, la solution est immédiate, comme montré ci-dessus.

Illustrons l'algorithme d'élimination gaussienne à l'aide d'un exemple. Considérons le système suivant :

$$\begin{pmatrix} 4 & 8 & 12 \\ 3 & 8 & 13 \\ 2 & 9 & 18 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \\ 11 \end{pmatrix} \tag{A.3}$$

La première étape consiste à diviser la première rangée du système par 4 afin de ramener la première valeur diagonale à l'unité :

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 8 & 13 \\ 2 & 9 & 18 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 11 \end{pmatrix} \tag{A.4}$$

Ensuite, on soustrait les multiples appropriés de la première rangée des autres rangées, afin d'annuler le reste de la première colonne :

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & 4 \\ 0 & 5 & 12 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 9 \end{pmatrix} \tag{A.5}$$

Ensuite, on recommence avec le deuxième élément de la diagonale : on divise par 2 cette fois et il reste

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 5 & 12 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 9 \end{pmatrix} \tag{A.6}$$

On soustrait de la dernière équation 5 fois la seconde, pour trouver enfin une forme triangulaire supérieure :

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 4 \end{pmatrix} \tag{A.7}$$

Ce système se résoud alors comme expliqué plus haut. Dans ce cas-ci la solution est $(x_1, x_2, x_3) = (2, -3, 1)$.

On montre que le nombre d'opérations arithmétiques nécessaires à cet algorithme est

$$\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \tag{A.8}$$

ce qui se comporte comme $\frac{1}{3}n^3$ quand n est grand. Autrement dit, inverser un système 2 fois plus grand prend 8 fois plus de temps.

Pivotage

L'algorithme de Gauss simple illustré ci-dessus ne fonctionne que si les éléments diagonaux sont non nuls à chaque étape. Si ce n'est pas le cas, on doit avoir recours au *pivotage*, c'est-à-dire à une permutation des rangées de manière à repousser au bas de la matrice la rangée qui pose problème. Ceci produira toujours une solution, en autant que la matrice est non singulière.

A.1.3 Décomposition LU

La décomposition LU est la représentation d'une matrice A comme produit d'une matrice triangulaire inférieure L par une matrice triangulaire supérieure U :

$$A = LU \quad \text{ou} \quad \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix} \tag{A.9}$$

On peut supposer que les éléments diagonaux de la matrice L sont l'unité.

La décomposition LU s'effectue d'une manière analogue à l'élimination de Gauss, par l'*algorithme de Crout*, que nous n'expliquerons pas ici (voir les références standards, comme [PTVF07]). Notons cependant que cet algorithme a en général recours au pivotage des rangées, et qu'il est toujours stable numériquement (en supposant bien sûr que la matrice A est non singulière). La complexité algorithmique de la décomposition LU est la même que celle de la multiplication de deux matrices, soit $\mathcal{O}(n^3)$.

Une fois la décomposition effectuée, la solution du système linéaire (A.1) s'effectue en deux étapes : on solutionne premièrement le système triangulaire $Ly = b$ pour y (par substitution vers l'avant), et ensuite le système triangulaire $Ux = y$ (par substitution vers l'arrière). De plus, la décomposition a de multiples avantages collatéraux :

1. Une fois effectuée pour une matrice A, elle permet de résoudre le système avec plusieurs vecteurs b différents, même si les vecteurs b ne sont pas connus au moment d'effectuer la décomposition.
2. Elle permet de calculer simplement l'inverse de la matrice, l'inverse d'une matrice se calcule en solutionnant le système avec n vecteurs b tels que $b_i = \delta_{ij}$ ($j = 1, 2, \ldots, n$).
3. Elle permet de calculer facilement le déterminant de A, comme le produit des éléments diagonaux de la matrice U.

A.1.4 Système tridiagonal

Supposons maintenant que la matrice A est tridiagonale. La résolution du système (A.1) ou la décomposition LU est alors beaucoup simplifiée, et s'effectue par une algorithme de complexité $\mathcal{O}(n)$. Nous n'irons pas dans les détails ici, mais nous contenterons d'insister sur le fait qu'utiliser une routine standard conçue pour une matrice pleine sur une matrice tridiagonale est extrêmement inefficace (temps de calcul n^3 au lieu de n).

Il existe aussi des routines spéciales pour la solution de systèmes de bandes qui sont avantageux si la largeur de bande ℓ est beaucoup plus petite que l'ordre de la matrice.

A.1.5 Matrices creuses et méthode du gradient conjugué

Supposons maintenant que le système (A.1) est très grand (par exemple $n = 10^5$) et que la matrice A est creuse, de sorte qu'il est hors de question d'utiliser l'élimination gaussienne de complexité $\mathcal{O}(n^3)$, d'autant plus que la matrice ne peut pas être stockée comme une matrice pleine. En général, un tel problème ne pourra être résolu. Cependant, si la matrice A est *définie positive*, c'est-à-dire si le produit $\widetilde{x}Ax$ est positif pour tout vecteur x, on peut utiliser la méthode du *gradient conjugué*, qui ne demande accès qu'à une procédure permettant de multiplier la matrice A par un vecteur quelconque. On sait généralement a priori, selon la nature du problème étudié, si la matrice A est définie positive ou non.

Stockage d'une matrice creuse

Notons premièrement que si la matrice est creuse, une façon simple de la garder en mémoire est de n'emmagasiner que les éléments non nuls et leurs positions. Par exemple, une structure couramment employée consiste à stocker trois tableaux :

1. Un tableau `v[N]` qui stocke les valeurs des `N` éléments de matrices non nuls, en progressant de gauche à droite, en ensuite de haut en bas.
2. Un tableau `J[N]` qui stocke les indices de colonne des éléments du tableau précédent.
3. Un tableau `I[n]`, qui stocke les indices des deux tableaux précédents où une novuelle rangée commence. Ainsi, `I[i]` est l'indice du premier élément de `v[]` ou de `J[]` appartenant à la rangée `i`.

Une routine qui calcule un vecteur `y[]` en multipliant la matrice A par un autre vecteur `x[]` contiendrait la boucle suivante :

```
1 for(i=0; i<n; i++){
2   y[i] = 0.0;
3   for(j=I[i]; i<I[i+1]; j++){
4     y[i] += v[j]*x[J[i]];
5   }
6 }
```

Directions conjuguées

On dit que deux vecteurs u et v sont *conjugués* selon A si $\widetilde{u}Av = 0$. En fait, comme A est défini positif, on peut définir un produit scalaire $\langle u|v\rangle_A$ ainsi :

$$\langle u|v\rangle_A \stackrel{\text{def}}{=} \widetilde{u}Av \tag{A.10}$$

La conjugaison par rapport à A signifie simplement l'orthogonalité par rapport à ce produit. On peut donc supposer qu'il existe une base de vecteurs p_k ($k = 1, \ldots, n$) qui sont tous mutuellement conjugués. La solution recherchée $x^\star$ à l'équation $Ax = b$ peut donc en principe s'exprimer sur cette base :

$$x^\star = \sum_{k=1}^{n} \alpha_k p_k \tag{A.11}$$

et les coefficients α_k de ce développement se trouvent par projection :

$$\alpha_k = \frac{\langle p_k|x^\star\rangle_A}{\langle p_k|p_k\rangle_A} = \frac{\widetilde{p}_k A x^\star}{\widetilde{p}_k A p_k} = \frac{\widetilde{p}_k b}{\widetilde{p}_k A p_k} \tag{A.12}$$

L'utilité des vecteurs conjugués vient précisément de la possibilité de remplacer $Ax^\star$ par b dans l'équation ci-dessus. La stratégie de la méthode sera donc de trouver une séquence de m vecteurs p_k qui nous rapproche le plus possible de la solution $x^\star$, tout en maintenant $m \ll n$.

Algorithme du gradient

Commençons par formuler le problème linéaire en tant que problème de minimisation. Étant donné le système (A.1), on forme la fonction à n variables suivante :

$$f(x) = \frac{1}{2}\widetilde{x}Ax - \widetilde{b}x \tag{A.13}$$

La solution recherchée est précisément le point $x^\star$ qui minimise la fonction f, car la condition de dérivée nulle donne précisément

$$\nabla f = Ax - b = 0 \tag{A.14}$$

Afin de converger vers la solution $x^\star$ à partir d'un point initial $x_0 = 0$, une méthode simple consiste à trouver la position x_1 du minimum de f dans la direction du gradient ∇f à x_0. Ensuite, on calcule de nouveau le gradient $r_1 = \nabla f|_{x_1}$ au point x_1 et on trouve la position x_2 du minimum dans cette nouvelle direction, et ainsi de suite jusqu'à convergence. Le gradient r_0 au point x_0 est simplement $(Ax_0 - b) = -b$. Le deuxième point x_1 sera au minimum de la fonction f le long de la direction r_0. Comme

$$f(\alpha r_0) = \tfrac{1}{2}\alpha^2\widetilde{r}_0 A r_0 - \alpha\widetilde{b}r_0 \quad \text{alors} \quad \frac{\partial f}{\partial \alpha} = \alpha\widetilde{r}_0 A r_0 - \widetilde{b}r_0 \tag{A.15}$$

et la condition de dérivée nulle donne la valeur

$$\alpha = \alpha_1 = \frac{\widetilde{b}r_0}{\widetilde{r}_0 A r_0} \quad \text{et donc} \quad x_1 = \alpha_1 r_0 \tag{A.16}$$

ce qui correspond précisément à la formule (A.12), avec $p_1 = r_0$.

Comme deuxième direction de minimisation, nous devons minimiser f le long de la direction $r_1 = Ax_1 - b$, ce qui nous mène au point

$$x_1 + \frac{\widetilde{r}_1 r_1}{\widetilde{r}_1 A r_1} r_1 \tag{A.17}$$

et ainsi de suite. Nous calculons de cette manière une suite de directions r_k et de positions x_k telles que

$$r_k = b - Ax_k \qquad x_{k+1} = x_k + \frac{\widetilde{r}_k r_k}{\widetilde{r}_k A r_k} r_k \tag{A.18}$$

Cette méthode itérative est appelée *l'algorithme du gradient* (angl. *steepest descent method*). Typiquement, même en dimension 2, cette méthode requiert un nombre infini d'itération avant de converger (voir la figure A.1). Ce n'est donc pas la route à suivre.

directions conjuguées

En fait, la direction r_k du gradient à un point donné n'est pas conjuguée aux directions précédentes. Pour que l'algorithme converge rapidement, les directions successives doivent être conjuguées selon A, ce qui permet d'épuiser systématiquement, sans répétition, les directions disponibles. L'algorithme du gradient conjugué construit plutôt des directions p_k qui sont les

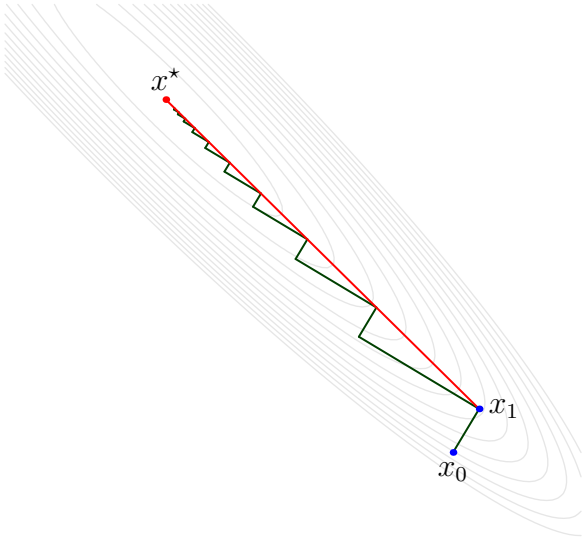


Figure A.1 Comparaison entre l'algorithme du gradient (en vert) et la méthode du gradient conjugué (en rouge) dans le cas $n = 2$. Les courbes de niveau de la fonction f sont indiquées en gris. La première nécessite une infinité d'étapes avant de converger, alors que la seconde converge en deux étapes.

plus proches possibles des gradients r_k, mais mutuellement conjuguées, ce qui se calcule par récurrence comme suit :

$$\begin{aligned} r_k &= Ax_k - b \\ p_{k+1} &= r_k - \sum_{i=1}^{k} \frac{\widetilde{p}_i A r_k}{\widetilde{p}_i A p_i} p_i \\ x_{k+1} &= x_k + \frac{\widetilde{p}_{k+1} b}{\widetilde{p}_{k+1} A p_{k+1}} p_{k+1} \end{aligned} \tag{A.19}$$

La première équation est simplement la direction du gradient au point x_k. La deuxième décrit une direction p_{k+1} obtenue de r_k en soustrayant les composantes de r_k qui sont parallèles aux directions p_i antérieures (au sens du produit $\langle \, | \, \rangle_A$ défini plus haut). Enfin, la nouvelle position x_{k+1} correspond au minimum de la fonction f le long de cette nouvelle direction, et coincide avec l'expression (A.12) pour les approximants successifs de $x^\star$. L'algorithme est interrompu lorsque le résidu r_k est suffisamment petit. La méthode du gradient conjugué converge vers la solution exacte en au plus n itérations. En particulier, la solution en deux dimensions (voir figure A.1) est atteinte après deux itérations seulement.

Le mérite principal de la méthode du gradient conjugué est qu'elle permet de résoudre un système linéaire d'ordre n en m étapes, où $m \ll n$, avec un nombre d'opérations de l'ordre $\mathcal{O}(nm^2) \sim \mathcal{O}(n \log n)$. Bien sûr, si on choisissait les directions p_k au hasard, le nombre d'itérations nécessaires pour converger serait de l'ordre de n et le gain serait nul. C'est le choix judicieux des directions p_k en rapport avec le gradient de f qui permet cette convergence accélérée.

A.2 Valeurs et vecteurs propres

A.2.1 Généralités

L'un des problèmes les plus fréquents de l'algèbre linéaire est la recherche des valeurs propres et vecteurs propres d'une matrice. En physique, ce problème surgit dans plusieurs contextes, le plus souvent dans la détermination des modes d'oscillations en mécanique, en électromagnétisme ou en mécanique quantique. Le calcul des niveaux d'énergie d'un système décrit par la mécanique quantique (atome, molécule, structure de bandes d'un solide, etc.) entre dans cette dernière catégorie.

L'équation aux valeurs propres d'une matrice carrée A est

$$Ax = \lambda x \tag{A.20}$$

Étant donné A, le problème est de trouver les valeurs λ pour lesquelles cette équation a une solution non nulle. Le vecteur propre x correspondant définit en fait un sous-espace propre et peut être multiplié par une constante quelconque. La dimension du sous-espace propre est le *degré de dégénérescence* de la valeur propre λ.

Si une matrice d'ordre n possède n vecteurs propres linéairement indépendants, on dit qu'elle est *diagonalisable*. Cela signifie qu'on peut construire une base de vecteurs propres de A, et que par conséquent la matrice A est diagonale dans cette base. Spécifiquement, si les n vecteurs propres sont normalisés et forment les colonnes d'une matrice U, alors

$$AU = UD \implies U^{-1}AU = D \tag{A.21}$$

où D est une matrice diagonale dont les éléments sont les valeurs propres de A (dans le même ordre que les vecteurs propres correspondants).

On montre qu'une condition suffisante pour que les vecteurs propres d'une matrice soient orthogonaux et qu'ils forment une base complète de dimension n est que la matrice A soit *normale*, c'est-à-dire qu'elle commutte avec son conjugué hermitique :

$$AA^\dagger = A^\dagger A \qquad \text{(matrice normale)} \tag{A.22}$$

Dans ce cas, les vecteurs propres étant orthogonaux, la matrice U est unitaire : $U^\dagger U = 1$. Une matrice hermitique ($A^\dagger = A$) ou symétrique ($\widetilde{A} = A$) est nécessairement normale. Si, au contraire, la matrice n'est pas normale, alors elle peut soit avoir un ensemble complet de n vecteurs propres qui ne sont pas orthogonaux, ou encore un ensemble incomplet de vecteurs propres.

Une littérature très vaste décrit les méthodes de diagonalisation des matrices. Ces méthodes sont typiquement itératives et visent à rendre la matrice A progressivement diagonale en appliquant une suite de transformation de similitude. La méthode de Jacobi est la plus ancienne. Une autre méthode consiste à réduire une matrice symétrique à une matrice tridiagonale (méthode de Householder) et ensuite à diagonaliser cette matrice tridiagonale, par l'une des méthodes suivantes :

1. On trouve les valeurs propres en trouvant les racines numériques du polynôme caractéristique, ce dernier s'évaluant rapidement dans le cas d'une matrice tridiagonale. Une fois les valeurs propres connues, calculer le vecteur propre revient à résoudre le système linéaire $(A - \lambda 1)x = 0$, où l'une des composantes de x est arbitraire.
2. Une alternative est de procéder à une décomposition QR, une procédure par laquelle la matrice tridiagonale T est exprimée sous la forme $T = QR$, où Q est une matrice orthogonale ($\widetilde{O} = O$) et R une matrice triangulaire supérieure. Cette transformation peut être itérée jusqu'à ce que la matrice tridiagonale devienne en fait diagonale (voir [PTVF07]).

En vérité, les méthodes efficaces de diagonalisation complète, par lesquelles toutes les valeurs propres et tous les vecteurs propres sont calculés, sont relativement complexes. Nous ferons appel aux méthodes pré-programmées de la librairie LAPACK plutôt que de les programmer.

Par contre, il est souvent nécessaire de trouver non pas toutes les couples (valeur et vecteur) propres, mais seulement les couples propres associées à la valeur propre la plus basse (ou la plus élevée), ou du moins à un petit nombre de valeurs propres extrêmes. Pour ce faire, la méthode de *Lanczos* est souvent utilisée, ou encore la méthode du gradient conjuguée, comme nous allons maintenant l'expliquer.

A.2.2 Méthode de Lanczos

La méthode de Lanczos permet de calculer les valeurs et vecteurs propres extrêmes – c'est-à-dire les plus élevées et les plus basses – d'une matrice de très grande taille. Elle se base sur une application itérative de la matrice sur des vecteurs : on doit fournir une façon d'appliquer la matrice A sur un vecteur quelconque, indépendamment de la manière dont cette matrice est emmagasinée, ce qui en fait une méthode particulièrement adaptée aux matrices creuses de très grande dimension. Certaines applications calculent même les éléments de matrice au fur et à mesure qu'ils sont requis, sans stocker la matrice d'aucune façon. [1]

L'idée de base derrière la méthode de Lanczos est de construire une projection de la matrice H (de dimension N) dont nous voulons les valeurs propres sur un sous-espace de petite dimension, mais qui contient les vecteurs propres extrêmes avec une assez bonne précision. Ce sous-espace de dimension $M \ll N$, appelé *sous-espace de Krylov*, est obtenu en appliquant à répétition $M-1$ fois la matrice H sur un vecteur de départ $|\phi_0\rangle$ qui peut être choisi au hasard :

$$\mathscr{K}(\phi_0, H, M) = \text{span}\left\{|\phi_0\rangle, H|\phi_0\rangle, H^2|\phi_0\rangle, \cdots, H^{M-1}|\phi_0\rangle\right\} \tag{A.23}$$

Les vecteurs $H^j|\phi_0\rangle$ ne sont pas orthogonaux, mais une base de vecteurs orthogonaux du même sous-espace est obtenue en appliquant la relation de récurrence suivante :

$$|\phi_{n+1}\rangle = H|\phi_n\rangle - a_n|\phi_n\rangle - b_n^2|\phi_{n-1}\rangle \tag{A.24}$$

1. Une référence importante sur les méthodes de solution des problèmes aux valeurs propres, en particulier pour les matrices de grande taille, est disponible en ligne : http://web.eecs.utk.edu/~dongarra/etemplates/book.html

avec les coefficients suivants :

$$a_n = \frac{\langle \phi_n | H | \phi_n \rangle}{\langle \phi_n | \phi_n \rangle} \qquad b_n^2 = \frac{\langle \phi_n | \phi_n \rangle}{\langle \phi_{n-1} | \phi_{n-1} \rangle} \tag{A.25}$$

et les conditions initiales $b_0 = 0$, $|\phi_{-1}\rangle = 0$.

Exercice 1.1

Démontrez que la relation de récurrence (A.24), qui peut être considérée comme une définition des vecteurs $|\phi_n\rangle$, avec les coefficients donnés en (A.25), entraîne que ces vecteurs sont mutuellement orthogonaux .

À chaque étape du calcul, trois vecteurs sont gardés en mémoire (ϕ_{n+1}, ϕ_n and ϕ_{n-1}). Ces vecteurs ne sont pas normalisés, mais on peut définir les vecteurs normalisés suivants :

$$|n\rangle = \frac{|\phi_n\rangle}{\sqrt{\langle \phi_n | \phi_n \rangle}} \tag{A.26}$$

En fonction de ces vecteurs orthonormés, la relation de récurrence (A.24) prend la forme suivante :

$$b_{n+1}|n+1\rangle = H|n\rangle - a_n|n\rangle - b_n|n-1\rangle \quad \text{ou encore} \quad H|n\rangle = b_n|n-1\rangle + a_n|n\rangle + b_{n+1}|n+1\rangle \tag{A.27}$$

Si on tronque l'espace de Krylov à l'étape M, l'action de la matrice H sur les vecteurs de base (A.26) peut être représentée par la matrice d'ordre M suivante :

$$T = \begin{pmatrix} a_0 & b_1 & 0 & 0 & \cdots & 0 \\ b_1 & a_1 & b_2 & 0 & \cdots & 0 \\ 0 & b_2 & a_2 & b_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{M-1} \end{pmatrix} \tag{A.28}$$

Il est alors très simple de diagonaliser cette matrice triadiagonale, dont l'ordre est petit par rapport à N, et dont les valeurs propres extrêmes convergent rapidement vers les valeurs propres extrêmes de H.

Convergence vers les valeurs propres extrêmes

Comment peut-on démontrer heuristiquement que les valeurs propres de la matrice T convergent vers les valeurs propres extrêmes de la matrice H ? Ceci est en fait une propriété de l'espace de Krylov. On constate sans peine les deux propriétés suivantes du sous-espace de Krylov :

1. $\mathscr{K}(\sigma\phi_0, \tau H, M) = \mathscr{K}(\phi_0, H, M)$. Autrement dit, multiplier le vecteur de départ ou la matrice par une constante ne change pas le sous-espace. C'est une conséquence de la définition d'un sous-espace vectoriel.
2. $\mathscr{K}(\phi_0, H + \sigma, M) = \mathscr{K}(\phi_0, H, M)$. Autrement dit, ajouter une constante à la matrice H ne change pas l'espace de Krylov. Ceci est une conséquence immédiate de la définition

de l'espace de Krylov : la puissance $(H+\sigma)^j$ est, après développement, une combinaison linéaire des puissances de H qui sont égales ou inférieures à j, et donc qui font déjà partie de l'espace de Krylov.

En vertu de ces deux propriétés, il n'y a aucune perte de généralité à supposer que les deux valeurs propres extrêmes de H sont -1 et 1, toutes les autres valeurs propres étant contenues entre ces deux valeurs. En effet, on peut toujours trouver les constantes σ et τ appropriées afin que les valeurs propres extrêmes de $\tau H+\sigma$ soient ± 1, et l'espace de Krylov est le même pour $\tau H+\sigma$ que pour H. Soit maintenant les vecteurs et valeurs propres exacts de la matrice H :

$$H|e_i\rangle = e_i|e_i\rangle \tag{A.29}$$

Le vecteur de départ $|\phi_0\rangle$ peut être dévelopé sur une base des vecteurs propres de H :

$$|\phi_0\rangle = \sum_i \gamma_i |e_i\rangle \tag{A.30}$$

et l'application de la puissance H^m sur $|\phi_0\rangle$ donne

$$H^m|\phi_0\rangle = \sum_i \gamma_i e_i^m |e_i\rangle \tag{A.31}$$

Quand m est suffisamment grand, on constate que $H^m|\phi_0\rangle$ est dominé par les deux vecteurs propres $|1\rangle$ et $|-1\rangle$. Donc ces deux vecteurs propres seront bien représentés dans l'espace de Krylov : leur projection sur ce sous-espace convergera rapidement vers l'unité en fonction du nombre d'itérations M.

Calcul des vecteurs propres

À chaque itération de la procédure de Lanczos, on doit calculer les valeurs propres et vecteurs propres de la matrice tridiagonale (A.28). Une procédure efficace de complexité $\mathcal{O}(M)$ est disponible au sein de LAPACK pour ce type de diagonalisation. On doit ensuite utiliser un critère de convergence pour arrêter la procédure. Une critère heuristique est que le changement relatif de la valeur propre la plus basse (ou la plus élevée) d'une itération à l'autre est inférieur à une limite donnée. Un meilleur critère, que nous ne démontrerons pas ici, est de cesser les itérations quand le résidu de Ritz

$$|R\rangle \stackrel{\text{def}}{=} H|e_0\rangle - e_0|e_0\rangle \tag{A.32}$$

est suffisamment petit. Dans cette expression e_0 est la valeur propre la plus petite estimée à l'étape M, et $|e_0\rangle$ le vecteur propre correspondant. On montre qu'une estimation de la norme de ce résidu est donnée par la dernière composante du vecteur propre de T associé à la valeur propre la plus basse, multipliée par b_{M-1}. Cette estimation est calculée à la l'étape suivante de la méthode `Lanczos` :

```
Ritz = abs(evector_tmp(j-1,0)*beta[j]);
```

Une fois la convergence atteinte, on peut obtenir quelques vecteurs propres estimés de H (ceux associés aux valeurs propres extrêmes et quelques autres qui leur sont proches) à l'aide des

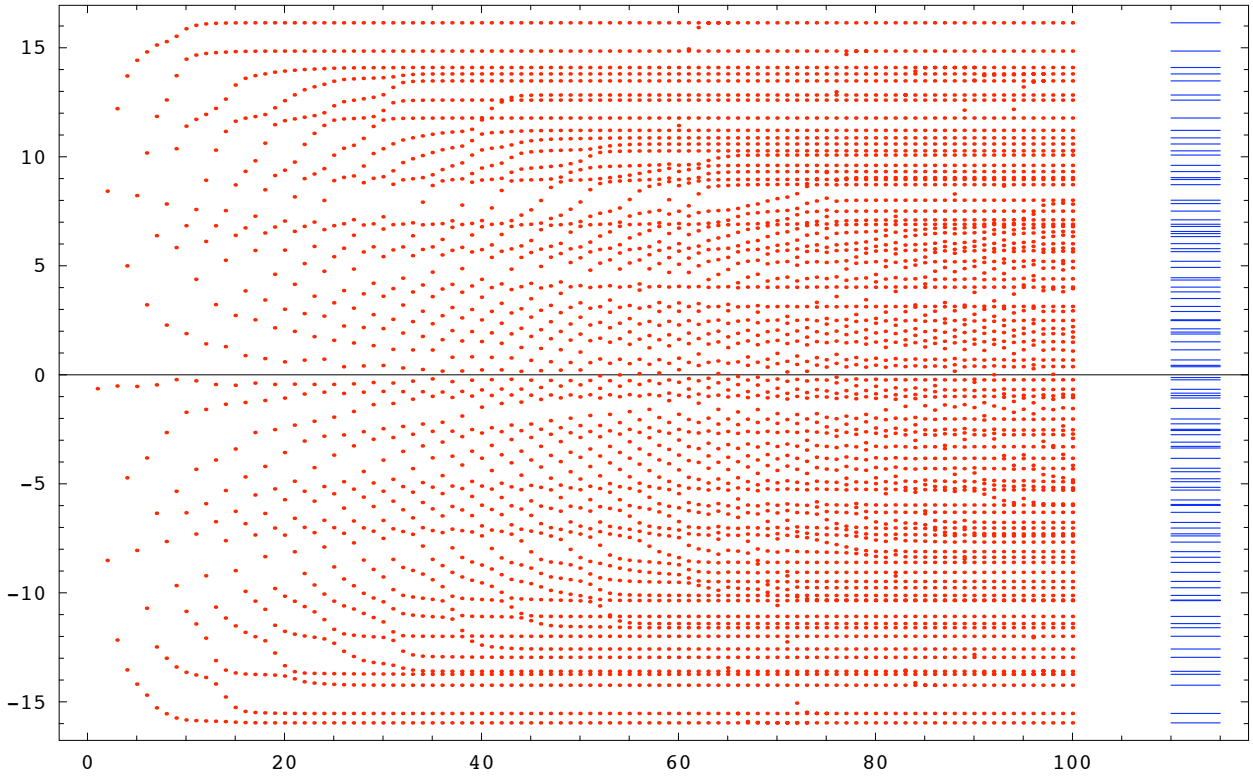


Figure A.2 Illustration de la convergence des valeurs propres dans la procédure de Lanczos, en fonction du nombre d'itérations effectuées, jusqu'à $M = 100$. La matrice H a une dimension $N = 600$. On constate que les valeurs propres extrêmes convergent rapidement. À chaque itération, une valeur propre additionnelle s'ajoute à l'ensemble. On note aussi qu'à l'itération 72, un valeur propre superflue apparaît, qui se fond avec la valeur propre la plus basse un peu après ; ceci est la manifestation d'une fuite d'orthogonalité.

vecteurs propres de la matrice T (qui sont connus dans la base de Lanczos (A.26)) et des vecteurs de cette base. Dans le code en annexe, ces derniers sont écrits sur disque au fur et à mesure qu'ils sont calculés, et ensuite relus afin de calculer les vecteurs propres dans la base originale du problème.

Remarques

1. Même si la relation de récurrence (A.24) garantit en principe l'orthogonalité des vecteurs de Lanczos, les erreurs d'arrondi vont éventuellement briser cette orthogonalité : il y aura des «fuites d'orthogonalité» et les M vecteurs résultants ne seront pas tous orthogonaux. Il est en fait assez courant que les états propres extrêmes soient représentés plus d'une fois dans la base de Lanczos. Ceci n'est pas grave si on ne cherche que les vecteurs propres extrêmes.

2. La méthode de Lanczos ne peut pas déterminer la dégénérescence des valeurs propres : si les valeurs propres extrêmes sont dégénérées, alors un seul vecteur par sous-espace

propre sera généralement trouvé.

3. La convergence vers les vecteurs propres extrêmes est d'autant plus rapide que la différence entre ces valeurs propres et les suivantes est grande.
4. La méthode de Lanczos est également valable pour estimer quelques valeurs et vecteurs propres sous-dominants (c'est-à-dire la deuxième plus grande ou petite, la troisième, etc.), mais la précision se détériore rapidement quand on s'éloigne des valeurs extrêmes.

A.3 Annexe : code

A.3.1 Classe de matrices creuses

Ce code et le suivant utilisent une représentation des matrices creuses définie dans `SparseMatrix.h`. on y trouve en particulier :

1. La méthode du gradient conjugué : `ConjugateGradient`
2. La méthode de Lanczos pour le problème aux valeurs propres $Ax = \lambda x$: `Lanczos`
3. La méthode de Lanczos pour le problème aux valeurs propres généralisé $Ax = \lambda Mx$: `Lanczos_generalized`

Code A.1 : Classe de matrices creuses : `SparseMatrix.h`

```
#ifndef SPARSEMATRIX_H_
#define SPARSEMATRIX_H_

#include <complex>
#include <vector>
#include <list>
#include <cassert>
#include <cmath>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <unistd.h>

#include "Matrix.h"

extern "C"{
#include "f2c.h"
#include "clapack.h"
}

#define LANCZOS_ITERMAX 200
#define LANCZOS_ACCURACY 1.0e-14

void EigensystemTridiagonal(int M, const double *alpha, const double *beta, Vector<
    double> &evalue, Matrix<double> &evector, bool evector_flag);
```

```
25 inline double real(double z) {return z;}
26
27 class matrix_element{
28 public:
29   int c;  indice de colonne
30   double v;  élément de matrice
31   matrix_element(int _c, double _v) : c(_c), v(_v){}
32 };
33 int matrix_element_comparator(const void *m1, const void *m2){ return(((
       matrix_element*)m1)->c - ((matrix_element*)m2)->c);}
34
35 Classe décrivant une rangée d'une matrice creuse
36 struct compressed_row{
37 public:
38   vector<matrix_element> elem;
39
40   void insert(int c, double v){
41     unsigned int i;
42     for(i=0; i<elem.size(); i++){
43       if(c == elem[i].c){
44         elem[i].v += v;
45         break;
46       }
47     }
48     if(i==elem.size()) elem.push_back(matrix_element(c,v));
49   }
50
51   effectue le tri des éléments de matrice dans la rangée
52   void consol(){
53     qsort((void *)&elem[1], elem.size(), sizeof(matrix_element),
           matrix_element_comparator);
54   }
55 };
56
57 Matrice creuse
58 class SparseMatrix{
59 public:
60   int n_rows; nombre de rangées
61   int n_cols; nombre de colonnes
62   Vector<compressed_row> row; tableau des rangées compressées.
63
64   SparseMatrix(int _n_rows, int _n_cols) : n_rows(_n_rows), n_cols(_n_cols) {
65     row.Alloc(n_rows);
66   }
67
68   SparseMatrix(int _n_rows) : n_rows(_n_rows), n_cols(_n_rows) {
69     row.Alloc(n_rows);
70   }
71
```

```
72    insère un nouvel élément de matrice dans la liste
73    inline void Insert(int r, int c, double v){
74      if(abs(v)<1.0e-12) return;
75      assert(r >= 0 and r < n_rows);
76      assert(c >= 0 and c < n_cols);
77      row[r].insert(c,v);
78    }
79
80    consolide la structure de donnée
81    void consol(){
82      for(int i=0; i<n_rows; i++) row[i].consol();
83    }
84
85    retourne le nombre de rangées
86    inline int rows(){return n_rows;}
87
88    retourne le nombre de colonnes
89    inline int columns(){return n_cols;}
90
91    multiplie la matrice par b, et produit le vecteur x
92    void mult(Vector<double> &x, Vector<double> &b){
93      x.clear();
94      mult_plus(x,b);
95    }
96
97    multiplie la matrice par b et ajoute au vecteur x
98    void mult_plus(Vector<double> &x, Vector<double> &b){
99      assert(x.size() == n_rows and b.size() == n_cols);
100     for(int i=0; i<n_rows; i++){
101       for(unsigned int j=0; j< row[i].elem.size(); j++){
102         x[i] += b[row[i].elem[j].c]*row[i].elem[j].v;
103       }
104     }
105   }
106
107   multiplie la matrice par b*z et ajoute au vecteur x
108   void mult_plus(Vector<double> &x, Vector<double> &b, double z){
109     assert(x.size() == n_rows and b.size() == n_cols);
110     for(int i=0; i<n_rows; i++){
111       for(unsigned int j=0; j< row[i].elem.size(); j++){
112         x[i] += b[row[i].elem[j].c]*row[i].elem[j].v*z;
113       }
114     }
115   }
116
117   surcharge de l'opérateur de flux d'impression
118   friend std::ostream & operator<<(std::ostream &flux, const SparseMatrix &x){
119     if(x.n_rows + x.n_cols > 128) flux << "Matrice trop grande pour être imprimée\n
        ";
```

```
120       else{
121         for(int i=0; i< x.n_rows; i++){
122           for(unsigned int j=0; j< x.row[i].elem.size(); j++){
123             cout << i << '\t' << x.row[i].elem[j].c << '\t' << x.row[i].elem[j].v <<
                    endl;
124           }
125         }
126       }
127       return flux;
128     }
129
130     construit une matrice dense M à partir de la représentation creuse
131     void make_dense(Matrix<double> &M){
132       M.clear();
133       assert(M.rows() == n_rows and M.columns() == n_cols);
134       for(int i=0; i< n_rows; i++){
135         for(unsigned int j=0; j< row[i].elem.size(); j++){
136           M(i,row[i].elem[j].c) = row[i].elem[j].v;
137         }
138       }
139     }
140
141     méthode du gradient conjugué
142     void ConjugateGradient(Vector<double> &b, Vector<double> &x, const double tol){
143
144       Vector<double> r(b); initialisation de r
145       mult_plus(r, x, -1.0);
146       Vector<double> p(r);
147       double res_old = r*r;
148       for(int i=0; i<x.size(); i++){
149         Vector<double> Ap(x.size());
150         mult_plus(Ap, p);
151         double z = p*Ap;
152         double alpha = res_old/z;
153         x.mult_plus(p,alpha);
154         r.mult_plus(Ap,-alpha);
155         double res_new = r*r;
156         if(sqrt(res_new) < tol) break;
157         p *= (res_new/res_old);
158         p += r;
159         res_old=res_new;
160       }
161     }
162
163     Méthode de Lanczos pour le problème aux valeurs propres généralisé
164     voir http://web.eecs.utk.edu/~dongarra/etemplates/node103.html
165     nvalue: nombre de vecteurs propres requis
166     eigenvalues: Tableau des valeurs propres
167     eigenvector: Tableau des vecteurs propres
```

```
168   highest: vrai si on sélectionne les plus hautes valeurs propres (au lieu des plus basses)
169   void Lanczos(int nvalue, Vector<double> &eigenvalues, vector<Vector<double> > &
          eigenvector, bool highest){
170
171     vérifier que les vecteurs propres sont alloués à la bonne taille
172     for(int i=0; i<nvalue; i++) assert(eigenvector[i].size() == rows());
173
174     vecteur de départ aléatoire
175     for(int i = 0; i < rows(); i++) eigenvector[0][i] = 2.0*rand()/RAND_MAX-1.0;
176     eigenvector[0] *= 1.0/sqrt(eigenvector[0].norm2());
177
178     int niter = (rows() > LANCZOS_ITERMAX)? LANCZOS_ITERMAX:rows();
179
180     FILE *V_file; fichier pour le stockage temporaire des vecteurs v_i
181     V_file = tmpfile();
182     assert(V_file);
183
184     assert(nvalue<=n_rows);
185
186     double Ritz=1.0;
187     Vector<double> beta(niter+1);
188     Vector<double> alpha(niter+1);
189     Vector<double> evalue(niter);
190     Matrix<double> evector;
191
192     Vector<double> r(eigenvector[0]); vecteur de départ
193     Vector<double> v(n_rows);
194     Vector<double> q(n_rows);
195     beta[0] = sqrt(r*r); β_0 = q̃r
196
197     int j;
198     for(j=1; j<=niter; j++){
199       q = v; stockage temporaire pour v_{j-1}
200       v = r; v *= 1.0/beta[j-1]; v_j = r/β_{j-1}
201       fwrite(v.array(),v.size(),sizeof(double),V_file); écriture de v_j sur disque
202       r.clear(); mult_plus(r,v); r = Av_j
203       r.mult_plus(q,-beta[j-1]); r = r − β_{j-1}v_{j-1}
204       alpha[j] = v*r; α_j = ṽ_j r
205       r.mult_plus(v,-alpha[j]);
206       beta[j] = sqrt(r*r); β_j = √(r̃r)
207
208       Matrix<double> evector_tmp(j);
209       EigensystemTridiagonal(j, &alpha[1], &beta[1], evalue, evector_tmp, true);
210       Ritz = abs(evector_tmp(j-1,0)*beta[j]);
211
212       cout << "itération " << j << "\trésidu = " << Ritz << endl;
213
214       if(Ritz < LANCZOS_ACCURACY or j== niter){
```

```
215           evector.Alloc(j,j);
216           evector = evector_tmp;
217           break;
218         }
219       }
220       niter = j;
221
222       if(highest) for(int k=0; k<nvalue; k++) eigenvalues[k] = evalue[niter-k-1];
223       else for(int k=0; k<nvalue; k++) eigenvalues[k] = evalue[k];
224
225       Calcul des vecteurs propres
226       rewind(V_file);
227       for(int k=0; k<nvalue; k++) eigenvector[k].clear();
228       for(j=0; j<niter; j++){
229         fread(v.array(),v.size(),sizeof(double),V_file); lecture de v_j depuis le disque
230         for(int k=0; k<nvalue; k++){
231           if(highest) eigenvector[k].mult_plus(v,evector(j,niter-k-1));
232           else eigenvector[k].mult_plus(v,evector(j,k));
233         }
234       }
235       fclose(V_file);
236     }
237
238     Méthode de Lanczos pour le problème aux valeurs propres généralisé
239     voir http://web.eecs.utk.edu/~dongarra/etemplates/node170.html
240     nvalue: nombre de vecteurs propres requis
241     eigenvalues: Tableau des valeurs propres
242     eigenvector: Tableau des vecteurs propres
243     highest: vrai si on sélectionne les plus hautes valeurs propres (au lieu des plus basses)
244     M: matrice de masse (problème aux valeurs propres généralisé)
245     void Lanczos_generalized(int nvalue, Vector<double> &eigenvalues, vector<Vector<
            double> > &eigenvector, bool highest, SparseMatrix *M){
246
247       vérifier que les vecteurs propres sont alloués à la bonne taille
248       for(int i=0; i<nvalue; i++) assert(eigenvector[i].size() == rows());
249
250       vecteur de départ aléatoire
251       for(int i = 0; i < rows(); i++) eigenvector[0][i] = 2.0*rand()/RAND_MAX-1.0;
252       eigenvector[0] *= 1.0/sqrt(eigenvector[0].norm2());
253
254       int niter = (rows() > LANCZOS_ITERMAX)? LANCZOS_ITERMAX:rows();
255
256       assert(nvalue<=n_rows);
257
258       double Ritz=1.0;
259       Vector<double> beta(niter+1);
260       Vector<double> alpha(niter+1);
261       Vector<double> evalue(niter);
262       Matrix<double> evector;
```

```
263
264       Vector<double> r(n_rows);
265       vector<Vector<double> > v;
266       vector<Vector<double> > w;
267
268       Vector<double> q(eigenvector[0]); vecteur de départ
269       M->mult_plus(r,q); r = Mq
270       beta[0] = sqrt(q*r); β_0 = √(q̃r)
271
272       int j;
273       for(j=1; j<=niter; j++){
274         v.push_back(q); v[j-1] *= 1.0/beta[j-1]; v_j = q/β_{j-1}
275         w.push_back(r); w[j-1] *= 1.0/beta[j-1]; w_j = r/β_{j-1}
276         r.clear(); mult_plus(r,v[j-1]); r = Av_j
277         if(j>1) r.mult_plus(w[j-2],-beta[j-1]); r = r − β_{j-1}w_{j-1}
278         alpha[j] = v[j-1]*r; α_j = ṽ_j r
279         r.mult_plus(w[j-1],-alpha[j]);
280
281         reorthogonalization
282         for(int k=0; k< j-1; k++) r.ortho(v[k],w[k]);
283
284         M->ConjugateGradient(r, q, 1e-7); résoudre Mq = r
285         beta[j] = sqrt(q*r); β_j = √(q̃r)
286
287         Matrix<double> evector_tmp(j);
288         EigensystemTridiagonal(j, &alpha[1], &beta[1], evalue, evector_tmp, true);
289         Ritz = abs(evector_tmp(j-1,0)*beta[j]);
290
291         cout << "itération " << j << "\trésidu = " << Ritz << endl;
292
293         if(Ritz < LANCZOS_ACCURACY or j== niter){
294           evector.Alloc(j,j);
295           evector = evector_tmp;
296           break;
297         }
298       }
299       niter = j;
300
301       if(highest) for(int k=0; k<nvalue; k++) eigenvalues[k] = evalue[niter-k-1];
302       else for(int k=0; k<nvalue; k++) eigenvalues[k] = evalue[k];
303
304       Calcul des vecteurs propres
305       for(int k=0; k<nvalue; k++) eigenvector[k].clear();
306       for(j=0; j<niter; j++){
307         for(int k=0; k<nvalue; k++){
308           if(highest) eigenvector[k].mult_plus(v[j],evector(j,niter-k-1));
309           else eigenvector[k].mult_plus(v[j],evector(j,k));
310         }
```

```
311       }
312     }
313 };
314
315 Calcule les valeurs propres et (optionnellement) les vecteurs propres d'une matrice tridiagonale
        symétrique.
316 M: taille de la matrice
317 alpha: première diagonale (commence à l'indice 0)
318 beta: deuxième diagonale (commence à l'indice 0)
319 evector: vecteurs propres
320 evector_flag: vrai si les vecteurs propres doivent être calculés aussi
321 void EigensystemTridiagonal(int M, const double *alpha, const double *beta, Vector<
        double> &evalue, Matrix<double> &evector, bool evector_flag)
322 {
323   doublereal *d, *e, *work, *z;
324
325   if(M==1){
326     if(evector_flag) evector(0,0) = 1.0;
327     evalue[0] = alpha[1];
328     return;
329   }
330
331   work = new doublereal[2*M]; assert(work!=0);
332   d = new doublereal[M]; assert(d!=0);
333   e = new doublereal[M-1]; assert(e!=0);
334   z = (doublereal *)evector.array();
335   memcpy(d,alpha,M*sizeof(*d));
336   memcpy(e,beta,(M-1)*sizeof(*e));
337
338   char jobz;
339
340   if(evector_flag) jobz='V'; else jobz='N';
341   integer info, ldz, nn;
342   nn = M;
343   ldz = M;
344
345   dstev_(&jobz, &nn, d, e, z, &ldz, work, &info); appel à LAPACK
346
347   assert((int)info==0);
348
349   for(int i=0; i<M; i++) evalue[i] = (double)d[i];
350
351   delete[] work;
352   delete[] d;
353   delete[] e;
354 }
355
356 #endif
```

Annexe B

Calcul Parallèle

La puissance des ordinateurs basés sur les technologies du silicium et des circuits intégrés progresse de manière exponentielle depuis 1970. Cette progression est généralement exprimée par la *loi de Moore* [1], qui stipule que le nombre de transistors à effet de champ qu'on peut placer sur un circuit intégré unique double environ à tous les deux ans. Cette loi industrielle s'applique de manière remarquable depuis 40 ans (voir fig. B.1).

Cette progression dans la densité des circuits a entraîné une augmentation concommitante de la fréquence des horloges internes (cadence), une complexification des processeurs et au total une augmentation de la puissance des microprocesseurs, elle aussi exponentielle. Cependant, cette augmentation de la cadence s'est à peu près interrompue au milieu des années 2000, en raison de la difficulté à évacuer la chaleur produite à l'intérieur du microprocesseur, chaleur proportionnelle à sa cadence. Comme la puissance de calcul des microprocesseurs ne peut plus être augmentée de manière directe en augmentant la cadence, il reste deux façons de l'augmenter :

1. Optimiser l'architecture interne de ces processeurs, par exemple en améliorant l'efficacité du *pipeline* des instructions. Cela est relativement difficile.
2. Multiplier les processeurs sur un même circuit en espérant qu'ils puissent travailler de concert. C'est l'introduction des *coeurs multiples*.

C'est évidemment la deuxième voie qui est a été suivie de la manière la plus spectaculaire par les manufacturiers. Pour le programmeur, elle pose le défi suivant : *Comment écrire un programme destiné à être exécuté simultanément par plusieurs processeurs distincts travaillant de concert* ? C'est le problème fondamental du *calcul parallèle*.

Précisons tout de suite que le calcul parallèle n'est pas né avec les processeurs multi-coeurs. Il se développe depuis les années 1980, car une façon évidente de multiplier la puissance des ordinateurs a toujours été le parallélisme. Cependant, le problème est différent depuis quelques années car il s'agit dorénavant de la seule façon d'y arriver, les cadences des processeurs étant maintenant saturées. Le calcul scientifique repose dans une large mesure sur la programmation parallèle, et les progrès futurs dans ce domaine – ainsi que dans les applications "non

1. D'après Gordon Moore, l'un des fondateurs de la compagnie Intel.

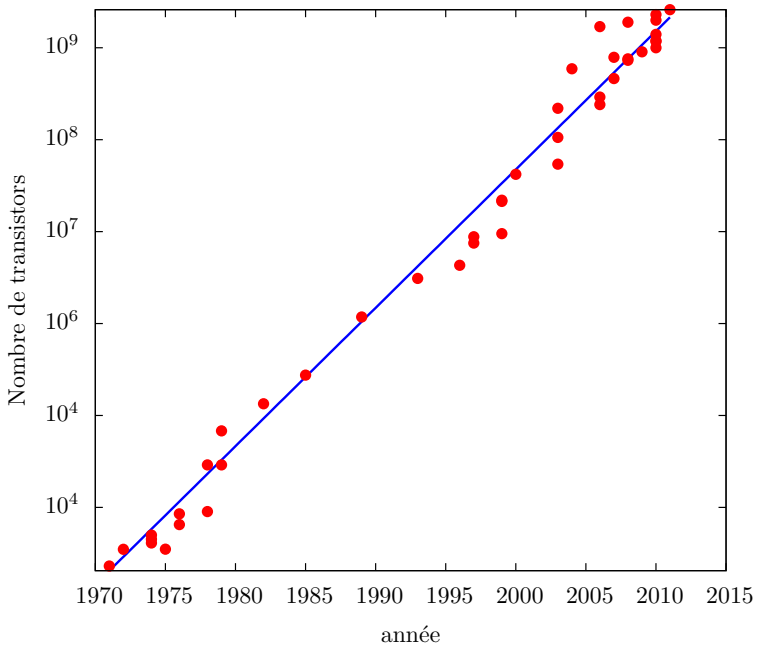


Figure B.1 La loi de Moore. Chaque point représente un processeur mis sur le marché. La droite représente une exponentielle doublant à tous les deux ans (en fait $2^{(y-1949)/2}$, où y est l'année. Notons que le transistor date de 1947).

scientifiques" qui reposent sur un calcul intensif – dépendent de manière cruciale de la capacité des programmeurs à tirer parti du calcul parallèle.

B.1 Généralités

On distingue généralement trois schémas d'organisation du traitement des données dans un système de calcul :

1. SISD (*single instruction, single data*). C'est le schéma séquentiel. Un seul processus existe traite une série d'instructions agissant sur une série unique de données.
2. SIMD (*single instruction, multiple data*). Dans ce schéma, une seule série d'instructions est exécutée, mais en parallèle sur plusieurs séries de données.
3. MIMD (*multiple instruction, multiple data*). Enfin, dans ce schéma, des instructions potentiellement différentes sont appliquées à des données différentes. C'est le modèle le plus général.

Jusqu'à la fin des années 1990, les supercalculateurs les plus puissants étaient souvent des ordinateurs *vectoriels*, qui fonctionnent selon le schéma SIMD. Les processeurs des ordinateurs vectoriels peuvent agir simultanément avec la même instruction sur un vecteur de données d'une certaine longueur (par exemple 128). Ainsi, l'addition de deux vecteurs de taille inférieure à la taille maximale dictée par le processeur se fait dans le même nombre de cycles d'horloge que l'addition de deux scalaires. Le produit scalaire se fait alors en deux opérations,

au lieu de $2N$ opérations, et ainsi de suite. Ces ordinateurs ont à toute fin pratique disparu du marché depuis l'arrivée de processeurs génériques plus puissants, car ils étaient très chers et produits en nombre relativement faible, alors que les processeurs génériques bénéficient de la production de masse. De nos jours, les processeurs graphiques (GPU) jouent ce rôle jusqu'à un certain point, quoiqu'ils soient beaucoup plus flexibles. Un processeur Intel ou AMD générique contient aussi une série d'instructions (désignées par l'acronyme SSEx, où x va de 1 à 5) qui mettent en oeuvre un schéma SIMD partiel pour des vecteurs relativement courts. Ces instructions ont été ajoutées dans le but d'accélérer des fonctions fréquemment utilisées dans le domaine du multimédia, mais peuvent être également utiles dans une programmation générique. Elles sont mises à contribution par l'utilisation d'options de compilation particulières.

Dans cette annexe, nous nous intéresserons plutôt à une mise en oeuvre du schéma MIMD, qui requiert la mise en place de plusieurs processus indépendants qui gèrent chacun leur propre ensemble de données. Nous devons distinguer deux types principaux de systèmes :

1. Les systèmes à *mémoire partagée*. Il s'agit d'une collection de processeurs qui ont accès à une mémoire commune et qui sont coordonnés par un système d'opération unique. Par exemple, un ordinateur doté d'un processeur multi-coeur appartient à cette catégorie. La parallélisation se fait à travers la création de processus concurrents (ou tâches, en anglais *threads*), qui ont accès au même espace-mémoire.
2. Les systèmes à *mémoire distribuée*. Il s'agit d'un ensemble d'ordinateurs indépendants ou *noeuds*, chacun pouvant être un petit système à mémoire partagée. Les noeuds communiquent entre eux par l'intermédiaire d'un réseau de communication rapide et la parallélisation se fait par échange de messages entre les différents noeuds.

B.1.1 Loi d'Amdahl

La loi d'Amdahl est une relation élémentaire qui donne l'accélération A (ou gain parallèle) d'un programme exécuté en parallèle en fonction de la fraction f du programme est qui parallélisable et du nombre N de processus concurrents :

$$A = \frac{1}{1 - f + f/N} \tag{B.1}$$

Expliquons. Soit t_1 le temps d'exécution du programme sur un seul processeur. La partie parallèle du programme demande un temps d'exécution ft_1 et la partie séquentielle un temps $(1-f)t_1$. Si on répartit le même programme sur N processeurs, chacun devra séparément exécuter la même partie séquentielle, en un temps $(1-f)t_1$, mais la partie parallèle, en négligeant les inefficacités liées à la communication entre les processus, sera exécuté en un temps ft_1/N. Le temps total d'exécution en parallèle, t_N, sera donc

$$t_N = (1-f)t_1 + \frac{ft_1}{N} \tag{B.2}$$

et l'accélération (angl. *speedup*) sera

$$A \stackrel{\text{def}}{=} \frac{t_1}{t_N} = \frac{1}{1 - f + f/N} \tag{B.3}$$

Cette relation simple est bien sûr une caricature de la réalité, car souvent les tâches concurrentes n'ont pas exactement la même durée, les problèmes de communication entre tâches se posent, la taille en mémoire de chaque tâche dépend de N et cela affecte le temps de calcul, etc. Cependant, la loi d'Amdahl nous permet de comprendre un aspect fondamental du calcul parallèle : l'accélération n'est appréciable que si la fraction parallèle f est importante. La règle d'or est qu'il ne faut pas utiliser plus de processeurs qu'il n'en faut pour diminuer le temps parallèle $\frac{ft_1}{N}$ à un niveau inférieur au temps séquentiel $(1-f)t_1$; autrement dit, le nombre raisonnable de processus à utiliser est

$$N \sim \frac{f}{1-f} \tag{B.4}$$

(évidemment, d'autres considérations sont en jeu ici, comme le nombre de processeurs disponibles, etc.). Le défi du parallélisme massif est donc de diminuer la partie séquentielle du programme, de manière à pouvoir efficacement augmenter N vers des tailles de plus en plus grandes.

B.2 Parallélisation avec openMP

L'un des outils de programmation parallèle les plus simples est le standard openMP. Il s'agit d'un ensemble de directives de compilation (qu'on appelle pragma en C++) et de fonctions de librairies qui permettent de paralléliser un code dans un modèle de *mémoire partagée*. Cette façon de faire est adaptée aux processeurs multicoeurs, mais ne permet pas de paralléliser vers un système à mémoire distribuée.

Nous n'expliquerons pas toutes les directives openMP ici ; ce n'est pas l'endroit. Plusieurs références et tutoriels sont disponibles sur le web. Nous allons simplement commenter un exemple extrêmement simple de parallélisation effectuée à l'aide de cette technique :

Code B.1 : Exemple élémentaire de parallélisation avec openMP

```
1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4  #include <omp.h>
5
6  using namespace std;
7
8  int main(){
9    const int n = 200000000;
10   double z=0.0;
11   double h = M_PI/n;
12   int i;
13
14 #pragma omp parallel for private(i) reduction(+: z)
15   for(i=0; i<n; i++) z += h*sin(i*h);
```

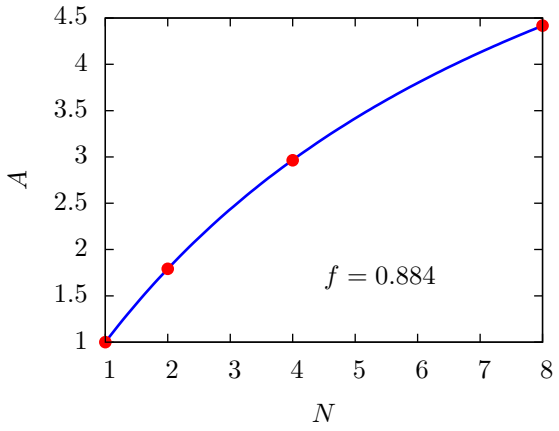


FIGURE B.2 Illustration de la loi d'Amdahl suite à l'exécution du programme openMP listé ci-dessus sur 1, 2, 4 et 8 coeurs. La courbe est un lissage de la loi d'Amdahl sur les données, résultant en une fraction parallèle $f = 0.884$.

```
16    cout << "integrale = " << setprecision(14) << z << endl;
17  }
```

Explications :

1. Le code est rudimentaire et calcule l'intégrale $\int_0^\pi \mathrm{d}x\ \sin(x) = 2$ par la méthode des trapèzes (sans tenir compte des extrémités) avec un nombre astronomique `n` de points et un pas d'intégration `h`$= \pi/$`n`. Le but du code est d'illustrer la parallélisation, avec la possibilité de mesurer le temps d'exécution en fonction du nombre de processus.
2. La ligne 4 inclut l'entête nécessaire à l'utilisation de openMP.
3. La ligne 14 est la directive nécessaire à la parallélisation openMP. Elle stipule que la boucle qui suit sera parallélisée, que la variable `i` sera privée à chaque tâche et que les variables `z` associées aux différentes tâches seront additionnées à la fin (ce qu'on appelle la réduction).
4. Le nombre de tâches pourrait être spécifié dans le code, mais il est généralement préférable de fixer la variable d'environnement `OMP_NUM_THREADS` à la valeur désirée, par exemple par la commande
   `export OMP_NUM_THREADS=8`
   avant l'exécution du programme.
5. La compilation du code avec openMP requiert une option, par exemple `-openmp` dans le compilateur `icpc`, ou `-fopenmp` dans le compilateur `g++`.

La figure B.2 illustre la loi d'Amdahl à l'aide des résultats obtenus par ce code. Le temps d'exécution a été estimé à l'aide de la commande `time`, par exemple en invoquant
`time ./a.out`

B.3 Parallélisation avec MPI

La programmation parallèle en mémoire distribuée ne peut pas se faire à l'aide de openMP. L'outil généralement utilisé dans ce cas est la librairie MPI (pour *Message Passing Interface*). La programmation MPI est plus complexe que la programmation openMP. Par contre, c'est essentiellement la seule méthode utilisée de nos jours en mémoire distribuée. On peut aussi l'utiliser sur les systèmes de mémoire partagée.

Encore une fois, le but de cette section n'est pas d'expliquer toutes les fonctionalités de MPI, mais simplement de donner un exemple très simple. Il sera basé sur le même code trivial utilisé plus haut pour openMP :

Code B.2 : Exemple élémentaire de parallélisation avec MPI

```
1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4  #include <mpi.h>
5
6  using namespace std;
7
8  int main(int argc, char *argv[]){
9
10   int nproc, rank;
11   MPI_Init(&argc, &argv);
12   MPI_Comm_size(MPI_COMM_WORLD,&nproc); nombre de processus
13   MPI_Comm_rank(MPI_COMM_WORLD,&rank); rang de chaque processus
14
15   const int n = 200000000;
16   double z=0.0;
17   double h = M_PI/n;
18   int i;
19
20   for(i=rank; i<n; i+=nproc) z += h*sin(i*h); boucle d'intégration
21   double ztot = 0.0;
22   MPI_Reduce(&z,&ztot,1,MPI_DOUBLE ,MPI_SUM,0,MPI_COMM_WORLD);
23   cout << "rang " << rank << "\tintegrale = " << setprecision(14) << ztot << endl;
24 }
```

Explications :

1. La ligne 4 inclut l'entête nécessaire
2. La ligne 13 initialise MPI.
3. La ligne 14 lance un appel de fonction afin de connaître le nombre de processus `nproc` utilisé.

4. La ligne 15 lance un appel de fonction afin de connaître le numéro (ou *rang*) `rank` du processus courant. Lire un code MPI correctement demande de se mettre à la place de chaque processus (tous les processus partagent le même code). La seule différence entre les différents processus est la valeur de la variable `rank`, et le code agira différemment selon cette valeur. Le processus associé à `rank=0` est appelé le *processus racine* (en anglais *root process*) ; mais cette distinction n'est pas toujours importante – du moins elle ne l'est pas dans cet exemple.
5. La boucle exécutée à la ligne 20 démarre à une valeur différente de `i` pour chaque processus, et saute de `nproc`, afin que chaque valeur de `i` soit couverte une seule fois, par le processus de rang `i%rank`.
6. Une fois à la ligne 22, chaque processus possède une partie de l'intégrale dans sa copie de la variable `z`. Pour obtenir la somme de ces variables, soit la réponse désirée, il faut que chaque processus envoie sa valeur de `z` au processus racine, qui les additionnera dans un nouvelle variable `ztot`. Cette étape est la *réduction* des données et est effectuée par la fonction `MPI_Reduce()`. L'avant-dernier argument de la fonction stipule que les données sont réduites vers le rang 0.
7. À la ligne 23, chaque processus imprime sa valeur de `ztot`. Elles sont toutes nulles, sauf celle associée au rang 0. Si on ne veut que voir les sortie émanant du rang 0, on n'a qu'à stipuler la condition
   `if(rank==0)···`

Bibliographie

[AS64] M. Abramowitz and I.A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. Dover publications, 1964.

[AT10] Vinay Ambegaokar and Matthias Troyer. Estimating errors reliably in monte carlo simulations of the ehrenfest model. *Am. J. Phys.*, 78 :150, 2010.

[BB01] J.P. Boyd and J.P. Boyd. *Chebyshev and Fourier spectral methods*. Dover Pubns, 2001.

[CD98] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annu. Rev. Fluid Mech.*, 30 :329–364, 1998.

[Cha] L. Champaney. Méthodes numériques pour la mécanique. Notes de cours, Université de Versailles, St-Quentin en Yvelines.

[DD01] H.M. Deitel and P.J. Deitel. *Comment programmer en C+*. Éditions R. Goulet, 2001.

[Fit] Richard Fitzpatrick. Computational physics. Lecture notes, University of Texas at Austin.

[For98] B. Fornberg. Calculation of weights in finite difference formulas. *SIAM review*, 40(3) :685–691, 1998.

[For08] A. Fortin. *Analyse numérique pour ingénieurs*. Presses inter Polytechnique, 2008.

[GO89] Gene H. Golub and Dianne P. O'Leary. Some history of the conjugate gradient and lanczos algorithms : 1948-1976. *SIAM Review*, 31(1) :pp. 50–102, 1989.

[GT88] H. Gould and J. Tobochnik. *An introduction to computer simulation methods : applications to physical systems*. Addison-Wesley Reading (MA), 1988.

[HL97] Xiaoyi He and Li-Shi Luo. Theory of the lattice boltzmann method : From the boltzmann equation to the lattice boltzmann equation. *Phys. Rev. E*, 56(6) :6811–6817, Dec 1997.

[LPB08] Rubin H. Landau, Manuel José Páez, and Cristian C. Bordeianu. *A survey of computational physics*. Princeton University Press, 2008.

[Mac] Angus MacKinnon. Computational physics — 3rd/4th year option. Lecture notes, Imperial College, London.

[Pai72] C.C. Paige. Computational variants of the Lanczos method for the eigenproblem. *J. Inst. Maths Applics*, 10 :373–381, 1972.

[Pai80] C.C. Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Linear algebra and its applications*, 34 :235–258, 1980.

[PTVF07] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes, the art of scientific computing*. Cambridge, 2007.

[SB02] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*. Springer, 2002.