UNIVERSITE DU BURUNDI

FACULTE DES SCIENCES

SECTION POLYTECHNIQUE

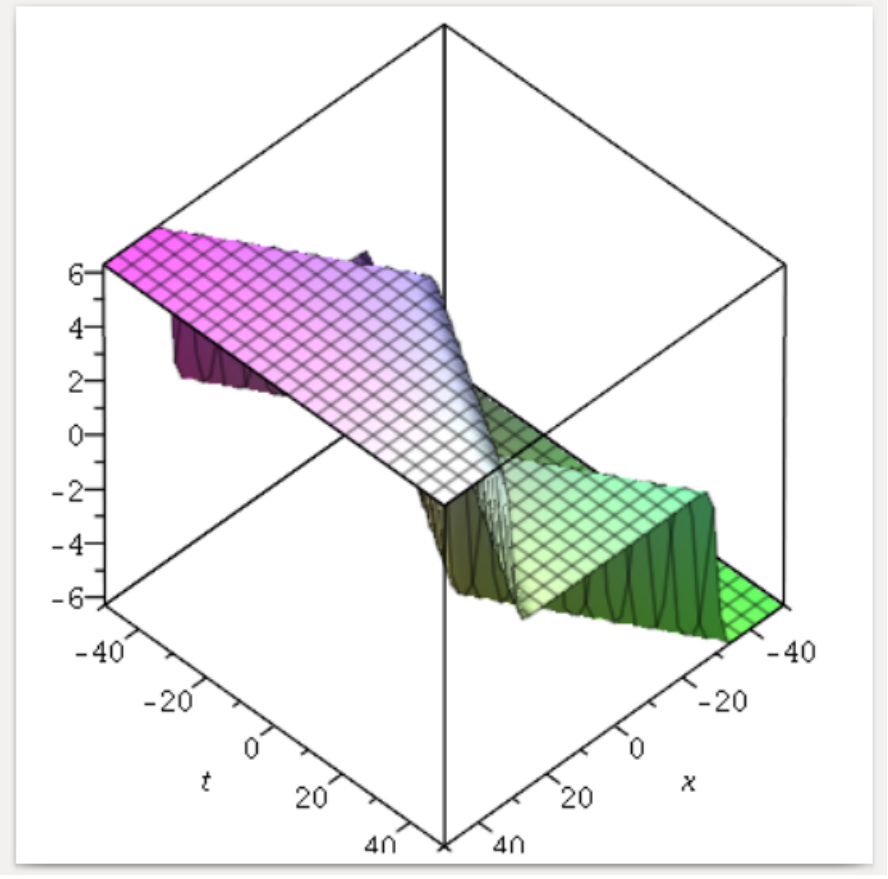
Bac I Polytechnique

Cours de Calcul Numérique et Programmation

(**Théorie** : 30 heures, **Travaux Dirigés** : 30 heures)

par

Prof. NYENGERI Hippolyte (Titulaire du cours) & **NDENZAKO Eugène**



Année académique 2020-2021

Table des matières

Descriptif du cours

Intitulé du cours : Calcul numérique et programmation

Titulaire : Prof. NYENGERI Hippolyte

Assistant : NDENZAKO Eugène

Objectifs du cours

Objectif général

L'objectif de ce cours est de présenter plusieurs méthodes numériques de base utilisées pour la résolution des systèmes linéaires et des équations non linéaires, ainsi que d'introduire aux étudiants les techniques d'analyse (théorique) de ces dernières, en abordant notamment les notions de convergence, de temps de calcul, de précision, d'erreurs et de stabilité. La présentation et l'analyse des méthodes sont suivies d'une implémentation et d'applications réalisées par les étudiants avec le langage de programmation **Python**.

Objectif spécifiques

A la fin de cet ECUE, les étudiants devraient être capables de :

- résoudre un problème numérique en faisant un choix raisonné de la méthode utilisée pour le résoudre,
- développer des algorithmes,
- étudier les algorithmes afin de sélectionner les bons algorithmes par une analyse poussée des erreurs de modélisation, des erreurs de représentation sur ordinateur, sans oublier les erreurs de troncature,
- implémenter les différentes méthodes de résolution des systèmes linéaires et d'équations non-linéaires avec le langage de programmation **Python** et dans l'*environnement* ***Linux*** (distribution ***UBUNTU***),
- faire preuve de rigueur,
- s'aider de représentations graphiques pour trouver la (les) solution(s) d'un problème.

Contenu du cours

- Histoire du calcul numérique.
- Objet de l'analyse numérique, théorie de la représentation des nombres, théorie des erreurs, conditionnement et sensibilité d'une méthode numérique.
- Initialisation à **Linux**.
- Introduction à la programmation en **Python**.
- Résolution des équations non linéaires : Position du problème ; localisation des zéros ; les méthodes d'encadrement comme celles de dichotomie et de la fausse position ; les méthodes des approximations successives ou méthodes du point fixe comme la méthode de la corde, celle de Newton-Raphson et celle de la sécante ; zéros de multiplicité m ; test d'arrêt des itérations. Cas particulier : recherche des racines d'un polynôme (Suite de Sturm : localisation des racines, approximation de la plus grande racine) ; normes matricielles et conditionnement.
- Résolution de systèmes linéaires : Position du problème, cas des matrices triangulaires ; méthode de Gauss, stratégie de pivot, coût de la méthode ; existence de la factorisation LU ; cas particulier des matrices symétriques définies positives.

Méthodologie du cours

L'enseignement magistral et des méthodes actives sont utilisés. Un syllabus servant de support de cours sera mis à la disposition des étudiants. La recherche individuelle et/ou en équipe est vivement conseillée. On pourra faire recours à l'apprentissage individuel. Il est prévu des Travaux Dirigés sur ordinateur.

Mode d'évaluation

Des travaux d'évaluation continue seront donnés (40%). Un examen final (60%) en deux parties $\big[$une partie écrite (25%) et une partie pratique : programmation sur ordinateur (75%)$\big]$ sera organisé aussi bien en première session qu'en deuxième session.

Prérequis

Analyse I, Algèbre I, Algorithmique I, Initiation à l'Informatique.

Conseils pour l'étude et l'examen

— Etudier au jour le jour et ne pas laisser la matière s'accumuler.

— Comprendre comment fonctionne la méthode utilisée pour résoudre un problème en s'aidant, quand c'est possible, d'une interprétation graphique de son mode opératoire.

— Décortiquer et exploiter pratiquement les programmes qui figurent dans les notes de cours.

— Résoudre effectivement les exercices proposés lors des séances en salle informatique et ne pas se contenter de lire les corrigés ou de faire fonctionner un programme écrit par une tierce personne.

— L'examen pratique (programmation) est à livre ouvert. Cette facilité peut se transformer en piège pour qui penserait que l'examen se réduirait à simplement appliquer des formules.

Références principales

1. Alfio Quarteroni, Riccardo Sacco and Fausto Saleri , **Méthodes Numériques. Algorithmes, analyse et applications** (Springer-Verlag Italia, Milano 2004).
2. Guillaume Legendre, **Méthodes numériques. Introduction à l'analyse numérique et au calcul scientifique**. Cours de Deuxième année de licence de Mathématiques et Informatique appliquées à l'Economie et à l'Entreprise (MI2E) à l'université de Paris-Dauphine, année académique 2009-2010.
3. Jean-Marc Huré et Didier Pelat, **Méthodes numériques. Eléments d'un premier parcours**. Cours destiné aux étudiants de DEA Astrophysique & Méthodes associées aux université Paris 7 et 11, année académique 2002-2003
4. Franck Jedrzejewski , **Introductions aux méthodes numériques,** 2^e **éd.** (Springer-Verlag France, Paris 2005).
5. Laurent Signac , **Introduction à l'algorithmique et à la programmation avec Python**, `https://deptinfo-ensip.univ-poitiers.fr`

Chapitre 0 Introduction générale

Ce document est un support au cours de *Calcul Numérique et Programmation* pour les étudiants de **Bac1** Polytechnique. Il aborde :

— la recherche des racines d'une fonction,

— la résolution numérique de systèmes d'équations linéaires.

Les applications se feront avec le langage de programmation **Python**, inventé par *Guido Van Rossum* en 1989 et utilisé par le monde scientifique depuis 20 ans environ. Le traçage des courbes sera fait avec le module **Matplotlib** en Python.

Le module **Matplotlib** peut être utilisé pour tracer des graphiques de toutes les formes. S'il est utilisé correctement, il peut fournir des résultats très professionnels, souvent bien supérieurs à ceux que l'on peut obtenir avec **Excel** par exemple.

Tous les **TP**, **TD** et **TPE** seront effectués dans un environnement **Linux** (distribution **UBUNTU**).

L'ordinateur est aujourd'hui un outil incontournable pour simuler et modéliser les systèmes, mais il faut encore savoir exprimer nos problèmes en langage formalisé des mathématiques pures. Nous sommes habitués à résoudre les problèmes de façon analytique, alors que l'ordinateur ne travaille que sur des suites de nombres. On verra dès lors qu'il existe souvent plusieurs approches pour résoudre un même problème, ce qui conduit à des algorithmes [1] différents. Un des objectifs de ce cours est de fournir des bases rigoureuses pour développer quelques algorithmes utiles dans la résolution de problèmes en physique.

Un algorithme, pour être utile, doit satisfaire un certain nombre de conditions. Il doit être :

- **rapide** : le nombre d'opérations de calcul pour arriver au résultat escompté doit être aussi réduit que possible.
- **précis** : l'algorithme doit savoir contenir les effets des erreurs qui sont inhérentes à tout calcul numérique. Ces erreurs peuvent être dues à la modélisation, à la représentation sur ordinateur ou encore à la troncature.
- **souple** : l'algorithme doit être facilement transposable à des problèmes différents.

Il importe de préciser que le principe d'un **modèle** est de remplacer un système complexe en un objet ou opérateur simple reproduisant les aspects ou comportements principaux de l'original (ex : modèle réduit, maquette, modèle mathématique ou numérique, modèle de pensée ou raisonnement).

Notons aussi que l'*aspect fini* des ordinateurs engendre nécessairement des erreurs. Considérons par exemple le cas d'un problème d'**Equation Différentielle Ordinaire** (EDO) ou d'**Equation aux Dérivées Partielles** (EDP). La solution exacte de ce problème est une fonction continue. Les ordinateurs, quant à eux, ne connaissent que le *fini* et le *discret*. En effectuant un calcul numérique, un ordinateur ne peut retenir qu'un nombre fini de chiffres pour représenter les opérandes et les résultats des calculs intermédiaires. Les solutions approchées seront calculées comme des ensembles de valeurs discrètes sous la forme de composantes d'un vecteur solution d'un problème matriciel. La représentation des nombres dans un ordinateur introduit la notion d'**erreur d'arrondi** ou de **troncature**. Ces erreurs peuvent se cumuler sur un calcul et la solution numérique finale pourra s'avérer très éloignée de la solution exacte.

1. Le mot algorithme vient du mathématicien arabe **Al-Khwarizmi** (VIIIè siècle) qui fut l'un des premiers à utiliser une séquence de calculs simples pour résoudre certaines équations quadratiques. Il est un des pionniers de l'al-jabr (algèbre).

Exemple d'erreur d'arrondi : Considérons un ordinateur utilisant 4 chiffres pour représenter un nombre. Calculons la somme $1.348 + 9.999$. Le résultat exact est 11.347 et comporte 5 chiffres. Le calculateur va le représenter de manière approchée : 11.35. Il commet une **erreur d'arrondi** égale à $(11.35 - 11.347) = 0.003$.

0.1 Un exemple : le calcul de $\sqrt{x}$

Sur ordinateur, l'addition de deux entiers peut se faire de façon exacte mais non le calcul d'une racine carrée. On procède alors par approximations successives jusqu'à converger vers la solution souhaitée. Il existe pour cela divers algorithmes. Le suivant est connu depuis l'antiquité (mais ce n'est pas celui que les ordinateurs utilisent).

Soit x un nombre réel positif dont on cherche la racine carrée. Désignons par a_0 la première estimation de cette racine, et par ϵ_0 l'erreur associée :

$$\sqrt{x} = a_0 + \epsilon_0 \,. \tag{1}$$

Cherchons une approximation de ϵ_0 . Nous avons :

$$x = (a_0 + \epsilon_0)^2 = a_0^2 + 2a_0\epsilon_0 + \epsilon_0^2 \,. \tag{2}$$

Supposons que l'erreur soit petite face à a_0, ce qui permet de négliger le terme en ϵ_0^2. Alors nous avons :

$$x \approx a_0^2 + 2a_0\epsilon_0 \,. \tag{3}$$

Remplaçons l'erreur ϵ_0 par un ϵ_0' , qui en est une approximation, de telle sorte que

$$x = a_0^2 + 2a_0\epsilon_0' \,. \tag{4}$$

On en déduit que

$$\epsilon_0' = (x/a_0 - a_0)\,/2 \tag{5}$$

Le terme

$$a_1 = a_0 + \epsilon_0' = (x/a_0 + a_0)\,/2 \tag{6}$$

constitue une meilleure approximation de la racine que a_0 , sous réserve que le développement soit convergent. Dans ce dernier cas, rien ne nous empêche de recommencer les calculs avec a_1 , puis a_2, etc., jusqu'à ce que la précision de la machine ne permette plus de distinguer le résultat final de la véritable solution. On peut donc définir une suite, qui à partir d'une estimation initiale a_0 devrait en principe converger vers la solution recherchée. Cette suite est :

$$a_{k+1} = (x/a_k + a_k)\,/2\,, \qquad a_0 > 0\,. \tag{7}$$

L'algorithme du calcul de la racine carrée devient donc

1. Démarrer avec une première approximation $a_0 > 0$ de $\sqrt{x}$.
2. A chaque itération k, calculer la nouvelle approximation $a_{k+1} = (x/a_k + a_k)/2$
3. Calculer l'erreur associée $\epsilon'_{k+1} = (x/a_{k+1} - a_{k+1})/2$
4. Tant que l'erreur est supérieure à un seuil fixé, recommencer en 2.

Le tableau ci-dessous illustre quelques itérations de cet algorithme pour le cas où $x = 4$ la valeur initiale $a_0 = 4$. Le programme suivant permet de générer les résultats du tableau 1. Si vous ne comprennez pas ces instructions pour le moment, le Chapitre 3 vous permettra d'en savoir plus. Durant la séance des TDs vous pourrez expérimenter l'effet du changement de la valeur initial a_0 et le nombre d'intérations sur la rapidité de la convergence.

```
x = 4
a0 = 4
i = 0
e0=(x/a0-a0)/2
print("%d\t\t%0.8f\t\t%0.9f"%(i,a0,e0))
for i in range(1,5):
    a1=(x/a0+a0)/2
    e1=(x/a1-a1)/2
    print("%d\t\t%0.8f\t\t%0.9f"%(i,a1,e1))
    a0=a1
```

i	a_i	ϵ'_i
0	4	-1.5
1	2.5	-0.45
2	2.05	-0.0494
3	2.00061	-0.000610
4	2.00000009	-0.000000093
etc		

TABLE 1 – Quelques itérations de l'algorithme du calcul de $\sqrt{x}$ dans le cas où $x = 4$ et $a_0 = 4$.

Nous voyons que l'algorithme converge très rapidement, et permet donc d'estimer la racine carrée d'un nombre moyennant un nombre limité d'opérations élémentaires (additions, soustractions, divisions, multiplications). Il reste encore à savoir si cet algorithme converge toujours et à déterminer la rapidité de sa convergence. L'analyse numérique est une discipline proche des mathématiques appliquées, qui a pour objectif de répondre à ces questions de façon rigoureuse.

0.2 Histoire du calcul numérique [Jean-Marc2002]

D'après les historiens, le calcul numérique remonte au moins au troisième millénaire avant notre ère. Il est à l'origine favorisé par le besoin d'effectuer des mesures dans différents domaines de la vie courante, notamment en agriculture, commerce, architecture, géographie et navigation ainsi qu'en astronomie. Il semble que les Babyloniens (qui peuplaient l'actuelle Syrie/Irak) sont parmi les premiers à réaliser des calculs algébriques et géométriques alliant complexité et haute précision. Surtout, ils donnent une importance et un sens au placement relatif des chiffres constituant un nombre, c'est-à-dire à introduire la notion de *base* de dénombrement, en l'occurrence, *la base sexagésimale* (base 60) que nous avons fini par adopter dans certains domaines. Ils se distinguent ainsi d'autres civilisations, même bien plus récentes, qui développent des méthodes plus lourdes, en introduisant une pléthore de symboles. Il y a environ 3500 ans, les populations de la vallée de l'Indus (région de l'Inde et du Pakistan) introduisent les notions de zéro et emploient les nombres négatifs. Ils adaptent également le système de comptage Babylonien au système *décimal* qui est le nôtre aujourd'hui. Ces premiers outils de calcul sont largement développés par la suite par les Grecs, puis transmis en Europe par l'intermédiaire des civilisations musulmanes peuplant le bassin méditéranéen.

Le calcul numérique tel que nous le concevons pratiquement aujourd'hui connaît son premier véritable essor à partir du XVII^{e} siècle avec les progrès fulgurants des Mathématiques et de la Physique, plus ou moins liés aux observations et aux calculs astronomiques. Plusieures machines de calcul sont en effet construites, comme la « Pascaline »inventée par B. Pascal en 1643, la DENO (« Defference

Engine Number One » ; voir la figure 1) de C. Babbage en 1834 mais qui fonctionnait mal, ou encore le tabulateur de H. Hollerith spécilament conçu pour recenser la population américaine, vers 1890. Il s'agit bien-entendu de machines mécaniques imposantes et d'utilisation assez limitée. Le manque de moyens de calcul performants limite en fait l'expansion et la validation de certaines théories du début du XX^{e} siècle. Ce fut le cas en particulier de la théorie de la Relativitié Générale due à A. Einstein.

La seconde Guerre Mondiale et les progrès technologiques qu'elle engendre va permettre au calcul numérique d'amorcer un second envol. Les anglais mettent au point le premier ordinateur en 1939, *COLOSSUS*, dont la mission est de décripter les messages codés envoyés par l'émetteur *ENIGMA* de l'Allemagne nazie. Cette machine introduit les concepts révolutionnaires émis par A. Turing dans les années 1936 concernant l'automatisation des calculs. Les calculateurs sont désormais entièrement électroniques. Autre machine qui fait date dans l'histoire, le *ENIAC* (« Electronic Numerical Integrator And Computer ») construit en 1946. Malheureusement, ce type de machine ne dispose pas de mémoire interne et doit être en permanence reprogrammée.

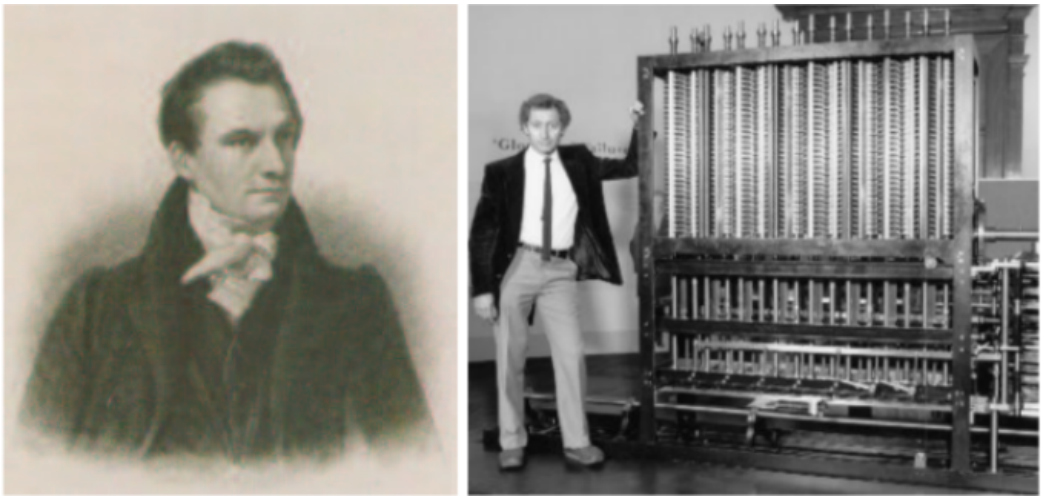


FIGURE 1 – Charles Babbage (*à gauche*), l'une des grandes figures de l'histoire de l'ordinateur. Il fut l'inventeur de la DENO (« Difference Engine Number One ») dont une copie est exposée dans un musée de Londres (*à droite*).

A la fin des années 1940, un certain J. von Neumann (voir la figure 3) repense l'architecture des ordinateurs et introduit, entre autres, les mémoires permettant de sauvegarder les programmes, et les concepts de *hardware* (matériel) et de *software* (logiciel). La première machine de calcul incluant les concepts de von Neumann (et ceux de Turing) est ainsi produite par la firme américaine **IBM** (International Business Machines) ; elle s'appelle **MARK I** et pèse 5 tonnes. Les premières applications concernent tous les domaines scientifiques et techniques. Le **FORTRAN I**, un language de programmation destiné aux scientifiques, est conçu dès 1954... mais il lui manque un vrai compilateur.

Vers la fin des années 1960, l'apparition progressive des transistors et de leur assemblage massif sur des surfaces de plus en plus réduites augmente considérablement les performances des machines et permet des simulations numériques de réalisme croissant. Cet effort de miniaturisation est d'ailleurs imposé par la course à la conquête de l'espace. Apparaissent ainsi en 1970 les fameux *microprocesseurs* mis au point par les firmes **INTEL** et **MOTOROLA** qui équipent la majeure partie des sondes spatiales de l'époque. Le calcul numérique devient rapidement une science à part entière. Les années 70 marquent aussi le tournant pour les languages de programmation : certains sont définitivement produits à des fins scientifiques, alors que d'autres seront pensés pour la gestion, comme le **COBOL**. Au début des années 1980, l'ordinateur le plus puissant du monde s'appelle CRAY I (voir la figure 3). Sa forme est spécialement choisie pour optimiser la rapidité des calculs. C'est aussi le début de l'informatique familiale avec la mise sur le marché des *PERSONAL COMPUTER* d'**IBM**.

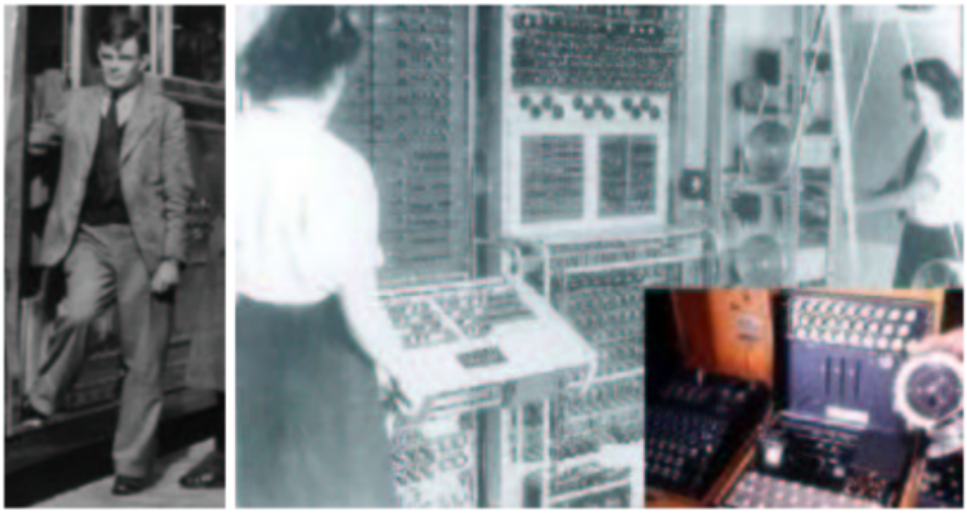


FIGURE 2 – Alan Turing (*à gauche*) donnera aux premiers ordinateurs les moyens de « penser »et de travailler de manière autonome. Sa théorie sera l'une des pièces maitresses de **COLOSSUS** (*à droite*), le premier calculateur mis au point par les anglais à l'aube de la Seconde Guerre Mondiale pour décripter les messages secrets émis par la machine ENIGMA (*en médaillon*) des Nazis.

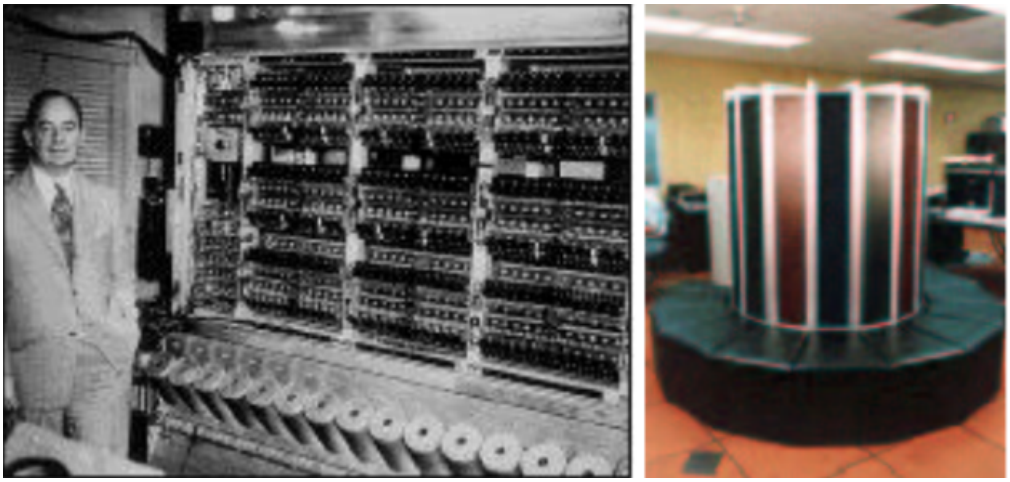


FIGURE 3 – John L. von Neumann (*à gauche*), enfant prodige de « l'informatique »de l'après guerre. Le CRAY I (*à droite*) construit dans les années 1980 dispose d'une puissance de calcul de 160 millions d'opérations à la seconde. Sa conception en forme de cylindre devait assurer une transmission optimale de l'information.

En une quinzaine d'années, la rapidité des calculateurs a été multipliée par plus de 10000. La vitesse d'exécussion des opérations élémentaires se compte maintenant en dizaines de millions de millions d'opérations à la seconde (ou dizaine de *tera-flops*, à comparer à la centaine de *méga-flops* du CRAY I). Les capacités de stockage ont gagné 7 ordres de grandeur au moins. Aujourd'hui, toutes ces performances doublent tous les ans. Pour le monde scientifique, celui de la Recherche Fondamentale et de l'Industrie, les calculateurs et le développement des techniques de programmation spécifiques (comme la *pagrammation parallèle*) sont devenus des outils incontournables à la connaissance et ouvrent de nouveaux horizons pour la modélisation et la compréhension des phénomèmes complexes et la mise au point de nouvelles technologies.

0.3 Vocabulaire

0.3.1 Algorithme

La description précise des opérations successives à effectuer pour obtenir un certain résultat s'appelle un algorithme. Un algorithme peut donc être assimilé à un protocole ou une recette. Le mot, d'origine arabe, vient du nom **Al Khuwarizmi** d'un mathématicien perse qui vécut au IXème siècle. Les premiers algorithmes remontent à Euclide et même aux Babyloniens.

0.3.2 Calcul numérique

On appellera calcul numérique tout calcul (évaluation d'un nombre, d'une fonction, d'une matrice,...) qui est effectué au moyen d'une machine et/ou de tables de valeurs numériques.

0.3.3 Analyse numérique

On peut définir l'analyse numérique comme l'étude théorique des méthodes constructives de l'algèbre et de l'analyse mathématique. Par méthode constructive il faut comprendre une méthode qui fournit le moyen d'obtenir la solution d'un problème, ou, à tout le moins, une approximation de celle-ci. Ainsi, un théorème qui établit l'existence et l'unicité de la solution d'un problème de Cauchy (cf. cours d'analyse) ne sera d'aucune utilité au numéricien s'il n'indique pas comment construire effectivement la solution du problème.

0.3.4 Programme

Ensemble d'instructions destinées à être exécutées par l'ordinateur. Un programme est conservé sur le disque dur sous forme de fichiers exécutables (qui font appel les uns aux autres) et de différents fichiers de données contenant les paramètres. Pour s'exécuter, un programme doit être chargé en mémoire vive (mémoire de travail de l'ordinateur).

0.3.5 Application

Aussi appelée *Programme* ou *Logiciel*, elle désigne une entité informatique du domaine du « software » (pas d'existence solide) exécutée à la demande de l'utilisateur par le système d'exploitation dans un but précis. Ainsi parle-t-on d'*application de traitement de texte, de traitement d'images*, etc. Notons que les termes *logiciel* (ensemble des éléments informatiques qui permettent d'assurer une tâche ou une fonction) et *application* sont souvent utilisés de manière équivalente.

0.3.6 Langages de programmation

Un langage de programmation est un code de communication, permettant à un être humain de dialoguer avec une machine en lui soumettant des instructions et en analysant les données matérielles fournies par le système, généralement un ordinateur. Il s'agit autrement dit d'*une notation conventionnelle destinée à formuler des algorithmes et produire des programmes informatiques qui les appliquent.* D'une manière similaire à une langue naturelle, un langage de programmation est composé d'un *alphabet*, d'un *vocabulaire*, de règles de *grammaire* et de *significations*. Le langage permet à la personne qui rédige un **programme** de faire abstraction de certains mécanismes internes, généralement des activations et désactivations de commutateurs électroniques, qui aboutissent au résultat désiré.

L'activité de rédaction du **code source** d'un programme est nommée **programmation**. Elle consiste en la mise en œuvre de techniques d'écriture et de résolution d'**algorithmes informatiques**, lesquelles sont fondées sur les **mathématiques**. À ce titre, un langage de programmation se distingue du langage mathématique par sa visée opérationnelle (une fonction et par extension, un programme, doit retourner une valeur), de sorte qu'un « langage de programmation est toujours un compromis entre la puissance d'expression et la possibilité d'exécution. »[Gilles1997]

Les langages utilisés pour programmer sont situés quelque part entre les séquences de 0 et 1 chères à la machine et le langage naturel cher à l'humain.

Pourquoi choisir un langage ?

L'informatique sur papier est possible, mais elle est moins distrayante que sur machine. De plus, la réalisation d'un programme est le moyen d'obtenir des réponses effectives aux problèmes qui nécessitent l'utilisation d'un ordinateur. Enfin, la programmation est une activité de création et de rigueur très formatrice, et elle aide à comprendre la manière dont fonctionnent les algorithmes.

Il existe des centaines de langages différents, qui peuvent être plus ou moins spécialisés pour certaines tâches, qui permettent d'utiliser différents paradigmes de programmation (impératif, objet, par contraintes, logique...). Les langages peuvent être plus ou moins verbeux, plus ou moins expressifs. Certains langages sont très répandus, ou au contraire ne sont utilisés que par une poignée de personnes. Les langages peuvent être jeunes ou vieillissant, et, lorsque le côté affectif s'en mêle, ils peuvent être agréables à utiliser ou au contraire agaçants... La tâche qui consiste à choisir un langage de programmation pour illustrer un cours d'algorithmique et d'informatique, à destination d'un public hétérogène est donc... difficile. Dans la suite, c'est le langage **Python** qui sera utilisé. Il est : généraliste, assez répandu (la pente étant dans le bon sens...), multi-paradigmes, assez jeune, expressif et agréable à utiliser. Il n'est cependant pas le seul à posséder toutes ces qualités. D'un point de vue plus technique, **Python** est aussi un langage interprété (par opposition à un langage compilé). En partie pour cette raison, il est d'apprentissage rapide, il est portable, mais il est en revanche lent à l'exécution. Enfin, il existe plusieurs interpréteurs Python, et l'interpréteur standard, nommé **CPython**, est libre, gratuit et multi plates-formes.

Voici un exemple de programme écrit en **Python** pour calculer les **nombres de Fibonacci**[2].

```
1  def fibonacci(n) :
2      f = [0, 1]
3      while len(f) < n + 1 :
4          f.append(f[-1] + f[-2])
5      return f
6  def main():
7      n = input("Entrez un nombre entier:")
8      n = int(n)
9      f = fibonacci(n)
10     print("suite de Fibonacci:U_{}={}\n".format(n,f))
11 main()
```

2. Les **nombres de Fibonacci** sont les termes de la **suite de Fibonacci** [Fibonacci]. La **suite de Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle est donc fondée sur la *formule de récurrence* suivante : $\mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n\,, \quad n \in \mathbb{N}$.

La suite de Fibonacci commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont : 0, 1, 1, 2, 3, 5, 8, 13, 21, etc. Elle doit son nom à **Leonardo Fibonacci** qui, dans un *problème récréatif* posé dans l'ouvrage **Liber abaci** publié en 1202, décrit la croissance d'une population de lapins : « Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ? » . Notons $\mathcal{F}_n$ le nombre de couples de lapins au début du mois n. Jusqu'à la fin du deuxième mois, la population se limite à un couple, ce qu'on note : $F_1 = F_2 = 1$). Dès le début du troisième mois, le couple de lapins a deux mois et il engendre un autre couple de lapins ; on note alors F_3=2. On choisit alors de poser $F_0 = 0$, de manière que cette équation soit encore vérifiée pour $n = 0$.

0.4 Exemples d'applications

Le calcul numérique est devenu indispensable dans de très nombreux domaines. La liste des disciplines qui ont recours au calcul et à l'analyse numériques est bien trop longue pour être dressée ici. Elle va de la physique à la démographie en passant par la chimie, la biologie, la médecine, le génie civil, l'archéologie, la gestion des stocks, la psychologie expérimentale, les jeux vidéos, etc.

Les sciences de l'ingénieur sont probablement celles qui font le plus appel au calcul numérique intensif. Les moyens informatiques modernes et les performances extraordinaires des microprocesseurs et des mémoires actuels ont permis un développement considérable des capacités de calcul des ordinateurs. A côté de ces matériels très performants, les chercheurs disposent de logiciels très élaborés (développés et mis au point par d'autres chercheurs). Il est devenu possible, par exemple, (ce n'était pas le cas il y a à peine dix ans) de simuler numériquement des « crash-tests » incluant à la fois les déformations des véhicules et les mouvements et chocs subis par les passagers (virtuels heureusement !). De telles simulations ont permis des avancées importantes en matière de sécurité automobile passive et active.

Un autre domaine qui a bénéficié des progrès du matériel et des logiciels informatiques est la météorologie. On peut, actuellement, lorsque les conditions de stabilité sont bonnes, prévoir l'évolution du temps sur une période de cinq à six jours. Ces prévisions sont effectuées dans des centres de calculs spécialisés (p.ex. Reading en Angleterre) et nécessitent de résoudre des millions de fois des systèmes de plusieurs millions d'équations à autant d'inconnues. Les ordinateurs utilisés dans ces centres peuvent effectuer plusieurs centaines de milliards d'opérations arithmétiques par seconde.

Une autre discipline dont le développement spectaculaire est lié au progrès des calculateurs numériques et de la modélisation mathématique est la bio-informatique. Certaines études, dans ce domaine, s'intéressent aux modifications conformationnelles de protéines qui sont à l'origine de maladies diverses telles la maladie d'Alzheimer et les encéphalopathies spongiformes. Ces études devraient permettre de mieux comprendre l'origine de ces maladies et par là, contribuer à les juguler.

Chapitre 1

Généralités

Dans ce chapitre, nous rappelons quelques notions générales concernant le calcul numérique. Pour les démonstrations et pour plus de détails, nous renvoyons le lecteur à [Jean-Marc2002, Alfio2004, Franck2005].

L'état d'esprit dans lequel travaille le « numéricien » est davantage celui que l'on connaît en Physique, avec ses multiples approximations (parfois difficilement justifiables) et hypothèses simplificatrices, plutôt que celui que l'on doit développer en Mathématiques. Une certaine rigueur est toutefois indispensable. Ainsi serons-nous le plus souvent amenés à changer assez significativement le problème initial, à le simplifier dans une certaine mesure, à le rendre linéaire, plus académique aussi, condition sine qua non au traitement numériquement. Une étape majeure sera alors de critiquer les solutions obtenues et de les interpréter par rapport au problème initialement posé. Une bonne illustration est le **pendule pesant** de longueur ℓ dont l'équation du mouvement dans un champ de pesanteur g est

$$\ell^2\ddot{\theta} - \ell g \sin\theta = 0. \tag{1.1}$$

L'harmonicité tant recherchée n'est pas réelle ; elle n'est que le pur produit d'une série d'approximations, notamment celle des « petits angles » qui permet de poser $\sin\theta \sim \theta$. L'équation précédente devient alors

$$\ell^2\ddot{\theta} - \ell g\theta = 0 \tag{1.2}$$

et l'on conçoit facilement que ses solutions soient différentes.

Numériquement par contre, il sera possible de décrire assez naturellement son mouvement anharmonique quelque soit l'amplitude initiale de son mouvement, soit par un développement limité de la fonction *sinus* à l'ordre 2 ou plus, soit par intégration directe de l'équation différentielle (1.1). Mais dans les deux cas, la solution obtenue, aussi précise soit-elle, ne sera qu'approchée, soit en raison de la troncature exercée sur le développement de Taylor, soit en raison des erreurs incontournables faites en intégrant l'équation différentielle ci-dessus.

1.1 Précision et temps de calcul

Deux aspects fondamentaux sous-tendent l'utilisation ou la mise en place de toute méthode numérique : la précision souhaitée (ou disponible) d'une part, et le temps de calcul d'autre part. Disposer de beaucoup de précision est confortable, voire indispensable, lorsque l'on étudie certains phénomènes, comme les systèmes chaotiques dont les propriétés affichent une forte sensibilité aux conditions initiales et à la précision. Mais lorsque l'on est trop « gourmand » sur cet aspect, le temps de calcul se trouve accru, parfois de manière prohibitive. Il n'est pas inutile de garder à l'esprit la *relation d'incertitude de Numerics*

$$\Delta t \times \varepsilon \gtrsim \hbar \tag{1.3}$$

où Δt exprime un temps de calcul (par exemple 13 min), ε la précision relative de la méthode (par exemple 10^{-10} ; voir ci-dessous) et $\hbar$ est une constante pour la méthode considérée. Cette relation rappelle l'impossibilité d'allier précision et rapidité.

1.1.1 Précision

La précision est limitée par les machines. Inévitablement, elle se trouve détériorée au fil des calculs. Sans entrer dans le détail, il faut savoir qu'un ordinateur fonctionne en effet avec un nombre limité de chiffres significatifs, au maximum 14, 15, voire 16 pour les plus performants, ces chiffres étant rangés en mémoire dans des boites virtuelles ou *bits*, après transcription en base 2. Certains bits sont réservés pour affecter un signe (+ ou -) au nombre et d'autres pour l'exposant. En ce sens, les machines classiques ne pourront généralement pas distinguer 3.1415926535897931 de 3.1415926535897932, à moins d'un traitement spécifique de la mantisse à la charge de l'utilisateur. Dans la mémoire, seuls les premiers chiffres d'un nombre seront enregistrés (voir le Tableau 1.1) ; les autres seront définitivement perdus. La plupart des langages permettent de choisir le nombre de décimales au niveau de la procédure déclarative, comme les anciennes instructions **REAL*4** ou **REAL*8** du langage *Fortran 77*, dont les équivalents sont respectivement **REAL(KIND=1)** et **REAL(KIND=2)** en *Fortran 90*. Le language Python quant à lui dispose de modules permettant d'augmenter la précision comme **mpmath** que nous utiliserons plus tard.

bits	Minimun (positif, non nul)	Maximum	Précision
32	$2.938736E-39$	$1.701412E+38$	$\sim 10^{-7}$
48	$2.9387358771E-39$	$1.7014118346E+38$	$\sim 10^{-12}$
64	$5.562684646268003E-309$	$8.988465674311580E+308$	$\sim 10^{-16}$

TABLE 1.1 – Minimum et maximum réels adressables et précision relative possible en fonction de bits de la machine.

1.1.2 Temps de calcul

Le temps de calcul est limité par la capacité des ordinateurs, par la durée de vie du matériel, par des facteurs extérieurs (comme les coupures d'électricité), et par l'utilisation que l'on souhaite classiquement faire des programmes de simulations. La plupart du temps, il s'agit d'une utilisation intensive où l'on balaye l'espace des paramètres d'un modèle. Ceci impose de rechercher toujours les méthodes les plus rapides. Il est de plus en plus fréquent de travailler sur des problèmes complexes dont la résolution nécessite des heures, des jours, parfois même des semaines de calculs sans interruption. Le choix s'orientera alors vers une méthode numérique rapide. Mais la rapidité se fera forcément au détriment de la précision (voir Eq.(1.3)). Quand les calculs sont intrinsèquement rapides et que le temps de calcul n'est pas un facteur déterminant, on aura tout loisir de (et peut-être intérêt à) choisir une méthode plus lente mais plus précise. Notons également un fait souvent oublié : les accès aux périphériques sont très couteux en temps, en particulier les accès aux disques durs et les impressions à l'écran.

Pour une méthode numérique donnée, on peut toujours estimer le temps de calcul en comptant le nombre total N d'opérations qu'effectue la machine. On peut ensuite éventuellement multiplier ce nombre par le temps de calcul τ_e relatif à une opération élémentaire, mais cela n'a de valeur qu'à titre comparatif (car ce temps τ_e varie d'une machine à l'autre et dépend aussi de la nature des opérations). Par exemple, pour calculer la force d'interaction gravitationnelle subie par une particule de masse m_i sous l'influence d'une particule de masse m_j située à la distance $\vec{r}_{ij} = x_{ij}\vec{u}_x + y_{ij}\vec{u}_y$, soit

$$\vec{F} = -\frac{m_i m_j}{r_{ij}^3}\,\vec{r}_i\,, \tag{1.4}$$

il faut effectuer $\mathcal{N} = 6 \times \frac{1}{2}(N-1)N$ opérations élémentaires s'il y a N particules au total, soit un temps de calcul égal à $3(N-1)N\tau_e$. En effet, le schéma associé à la relation (1.4) pourra être transcrits sous la forme

$$\begin{cases} a \leftarrow -m\,(i)\,/r\,(i,j) * m\,(j)\,/r\,(i,j)\,/r\,(i,j) \\ Fx\,(i,j) \leftarrow a * x\,(i,j) \\ Fy\,(i,j) \leftarrow a * y\,(i,j) \end{cases}$$

et met en jeu trois multiplications et trois divisions. On dira alors que *l'ordre de la méthode est* $\sim (N-1) \times N$, soit N^2 lorsque N est grand. Nous voyons que sur une machine parallèle qui pourra traiter simultanément l'interaction de chaque particule (soit une machine dotée de N processeurs indépendants), l'ordre est très inférieur : il varie comme $(N-1)$. C'est donc un gain de temps colossale pour N grand.

1.2 Les erreurs

Outre les erreurs de programmation (celles que le compilateur détectera et que vous corrigerez, et celles que vous ne trouverez jamais), il existe trois sources d'erreur qu'il convient d'avoir présent à l'esprit : l'*erreur de schéma* (*erreur de troncature*), l'*erreur de représentation* (*erreur d'arrondi*) et l'*erreur par perte d'information* (*erreur de méthode*).

1.2.1 L'erreur de schéma

L'*erreur de schéma* est générée lorsque l'on remplace une relation « exacte » par une autre, plus simple ou plus facilement manipulable. C'est par exemple le cas d'une série de Taylor tronquée (on parle ici plus précisément d' **erreur de troncature**, comme

$$\mathrm{e}^x \approx 1 + x + \frac{1}{2}x^2, \tag{1.5}$$

ou le cas de l'approximation d'une dérivée par une *différence finie* comme

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{1.6}$$

ou encore de l'estimation d'une *quadrature*

$$\int_a^b f(x)\mathrm{d}x \approx f(a)(b-a)\cdot \tag{1.7}$$

L'*erreur de schéma* est généralement l'erreur dominante. Elle est en principe incontournable car toutes les méthodes numériques sont basées sur des schémas. Il est important qu'elle soit connue ou appréciable. Notez que tous les schémas ne génèrent pas les mêmes erreurs. On choisira donc, quand c'est possible, les schémas les plus précis.

En résumé, les *erreurs de schéma* (ou *de troncature*) sont liées à la précision de l'algorithme utilisé. Elles peuvent être contrôlées par l'algorithme lui-même. Si une fonction est approchée par son développement de Taylor, l'*erreur de troncature* sera obtenue par une évaluation du reste du développement. Son contrôle sera obtenu par une majoration de ce reste. Au voisinage d'un point a, si une fonction f admet un développement de Taylor de la forme

$$f(x) = f(a) + \frac{x-a}{1!}f'(a) + \cdots + \frac{(x-a)^{n-1}}{(n-1)!}f^{(n-1)}(a) + \int_a^x \frac{(x-t)^{n-1}}{(n-1)!}f^{(n)}(t)\mathrm{d}t\,, \tag{1.8}$$

et si la dérivée n-ième de f est majorée par une constante M, le reste sera majoré par

$$\left| \int_a^x \frac{(x-t)^{n-1}}{(n-1)!}f^{(n)}(t)\mathrm{d}t \right| \leq M\frac{|x-a|^n}{n!}\cdot \tag{1.9}$$

1.2.2 La représentation machine (erreur d'arrondi)

L'*erreur de représentation* est liée aux modes de représentation, de stockage et de calcul des ordinateurs. Par exemple, supposons que l'on veuille effectuer l'opération $\frac{1}{3} - \left(\frac{2}{3} - \frac{1}{3}\right)$ sur une machine fonctionnant avec 4 chiffres significatifs. En interne, la machine effectuera d'abord $\frac{2}{3} - \frac{1}{3}$, ce qui pourra donner

$$0.6667 - 0.3333 = 0.3334.$$

Ici, $\frac{2}{3}$ a été remplacé par 0.6667 ; c'est une erreur de représentation (erreur d'arrondi). Reste à effectuer $\frac{1}{3} - 0.3334$, d'où le résultat faux (ou vrai à 10^{-4} près) : -0.0001. Certaines machines peuvent d'ailleurs fournir 0.6666 pour $\frac{2}{3}$ et finalement donner, par coïncidence, un résultat exact.

L'*erreur de représentation* est généralement négligeable devant les autres, sauf dans certains cas particuliers où, sans précaution, l'on manipule des nombres très différents ou très proches.

1.2.3 La perte d'information (erreur de méthode)

La perte d'information est une conséquence de la représentation machine. Elle se produit lorsque l'on soustrait deux nombres très proches ; le résultat est alors un nombre comportant peu de chiffres significatifs. Soit à effectuer $\frac{71}{500} - \frac{1}{7}$, toujours sur une machine travaillant avec 4 chiffres significatifs en mantisse. En pratique, la machine donne

$$0.1420 - 0.1429 = -0.0009$$

contre $-\frac{3}{3500} \simeq -8.5714 \times 10^{-4}$. Ce résultat, compte-tenu de l'arrondi, est correct. Toutefois, il ne comporte plus qu'un seul chiffre significatif alors que les deux nombres initiaux en possédaient quatre.

L'*erreur par perte d'information* peut être dévastatrice (par exemple lorsque l'on veut calculer numériquement une dérivée). Souvent, on ne la soupçonne pas. Elle survient aussi lorsque l'on effectue beaucoup d'additions et soustractions à la queue leu leu en mettant en jeu des termes de même amplitude (c'est le cas par exemple du calcul de séries alternées).

Les *erreurs de méthode* se produisent aussi lorsqu'une expression est mal équilibrée et mélange des valeurs dont la différence est importante. C'est un problème de calibration numérique qui est sensible aux erreurs d'arrondi. Dans la plupart des cas, l'algorithme doit être modifié. Considérons l'équation du second degré

$$10^{-8}x^2 - 0.8x + 10^{-8} = 0\cdot \tag{1.10}$$

Cette équation admet deux racines, $r_1 \simeq 0.8 \times 10^8$ et $r_2 \simeq 1.25 \times 10^{-8}$. Si on ne s'intéresse qu'à la plus petite racine, certains calculateurs et en particulier les calculatrices de poche donnent des valeurs erronées. Cela provient du fait que lors du calcul du discriminant, la soustraction $\Delta = 0.64 - 4 \times 10^{-16}$ n'est pas toujours correctement effectuée car le terme 4×10^{-16} est négigé devant 0.64. Pour obtenir une valeur exacte, on doit modifier l'algorithme en proposant par exemple de calculer la racine r_2 par la relation donnant le produit des racines $r_2 = \frac{1}{r_1}$. Remarquons que si l'on multiplie l'équation par 10^8 le problème reste entier.

Dans les processus récurrents ou itératifs, les erreurs s'ajoutent, ce qui a pour effet d'amplifier l'erreur globale et de diminuer la précision du calcul.

La propagation des erreurs dans diverses parties du calcul a pour conséquence d'ajouter de l'imprécision là où elle n'était pas nécessairement attendue. Dans les calculs itératifs, l'erreur se propage d'une étape à l'autre.

Exemple. Dans le calcul numérique des termes de la suite définie par la relation de récurrence

$$x_{n+1} = \frac{1}{n} + a x_n\,, \tag{1.11}$$

l'erreur donnée par

$$\Delta x_{n+1} \simeq a \Delta x_n \tag{1.12}$$

évolue exponentiellement. A l'étape n, l'erreur est multipliée par a^n. D'une étape à l'autre, l'erreur se propage et peut conduire à l'explosion de l'algorithme.

1.3 Conditionnement et sensibilité

1.3.1 Propagation des erreurs

Dans toute méthode numérique, il y a au moins un paramètre permettant de régler (c'est-à-dire d'imposer) ou de contrôler le bon déroulement du processus et la précision du résultat. Il s'agit généralement d'un critère faisant appel à l'*erreur absolue* ou à l'*erreur relative* sur une quantité y par rapport à une quantité de référence $y^\star$. Ces erreurs sont définies respectivement par

$$\Delta(y) = y - y^\star \tag{1.13}$$

et par

$$\epsilon(y) = \frac{y - y^\star}{y^\star}, \qquad y^\star \neq 0\cdot \tag{1.14}$$

Un fait est incontournable : les erreurs, quelle que soit leur origine, se propagent et s'amplifient. On peut le comprendre sur un exemple simple. Si une quantité x est connue avec une précision relative $\epsilon(x) \ll 1$, alors son image par une fonction f est aussi entachée d'une erreur

$$f(x + \epsilon(x)x) \approx f(x) + \epsilon(x) x f'(x) \tag{1.15}$$

et l'erreur relative sur f vaut

$$\epsilon(f) = \frac{f(x + \epsilon(x)x) - f(x)}{f(x)} \approx \epsilon(x) x \frac{f'(x)}{f(x)}\cdot \tag{1.16}$$

On voit donc que si $f'(x)$ est important et $f(x)$ faible, à la fois $\Delta(f)$ et $\epsilon(f)$ peuvent être grands. Notez que l'on peut aussi écrire la relation précédente sous la forme

$$\epsilon(f) = \epsilon(x)\eta \tag{1.17}$$

où η est le *nombre de conditionnement* (voir ci-dessous).

Ajoutons qu'il faudrait en toute rigueur être capable de mettre une barre d'erreur sur les résultats issus d'une méthode ou plus généralement d'une simulation , en tenant compte de toutes les sources d'erreur possibles. Il est navrant de constater que la notion de barre d'erreur ne semble concerner que les physiciens expérimentateurs. C'est un tort, car une simulation numérique est une expérience comme une autre...

1.3.2 Nombre de conditionnement

Une méthode numérique travaille généralement avec un certain nombre de paramètres, les *paramètres d'entrée*, et renvoie des résultats, que l'on peut assimiler à des *paramètres de sortie*. Un objectif (souvent difficile à atteindre) que recherchera le Physicien est de produire des résultats qui ne dépendent pas (ou peu) de la méthode choisie. L'aspect qui guidera son choix sur une méthode plutôt qu'une autre est la *stabilité*, c'est-à-dire une certaine garantie que la procédure n'aura pas la facheuse propriété d'amplifier et/ou de générer sans limites les erreurs.

La sensibilité (ou la stabilité) peut être quantifiée grâce au *nombre de conditionnement* (ou *nombre de condition*). Prenons le cas d'une méthode qui, pour une quantité x donnée, fournit une quantité y. On définit ce nombre pour deux couples (x,y) et (x', y') par

$$\eta = \frac{\dfrac{y' - y}{y}}{\dfrac{x' - x}{x}} \equiv \eta(x) \tag{1.18}$$

Ce nombre rend compte du *pouvoir amplificateur* d'une méthode. Ainsi, une méthode numérique est très sensible (ou instable) si $\eta \gg 1$, peu sensible (stable) si $\eta \lesssim 1$ et est insensible si $\eta \ll 1$.

Notez que si x et x' sont très proches, alors

$$\eta \sim \frac{x f'(x)}{f(x)} \tag{1.19}$$

La sensibilité peut donc être importante lorsque $f \to 0$ et/ou $f' \to \infty$ et/ou $x \to \infty$.

1.3.3 Normes et conditionnement d'une matrice

Le conditionnement mesure l'influence des erreurs d'arrondi sur la solution d'un problème donné. Il est mis en évidence par une légère perturbation des données initiales. C'est une notion générale qui s'applique aussi bien aux racines d'un polynôme vis-à-vis de la variation de ses coefficients qu'aux valeurs propres ou vecteurs propres d'une matrice vis-à-vis de la perturbation de ses éléments. Considérons le système linéaire $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ suivant :

$$\begin{pmatrix} 23 & 9 & 12 \\ 12 & 10 & 1 \\ 14 & -12 & 25 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 44 \\ 23 \\ 27 \end{pmatrix} \tag{1.20}$$

qui admet la solution $(1, 1, 1)$. Remarquons que la matrice $\mathbf{A}$ est invisible et admet trois valeurs propres $\lambda_1 \simeq 36,16 \quad \lambda_2 \simeq 0,056$ et $\lambda_3 \simeq 21,79$. Considérons le problème suivant dans lequel le vecteur $\mathbf{b}$ est légèrement perturbé

$$\begin{pmatrix} 23 & 9 & 12 \\ 12 & 10 & 1 \\ 14 & -12 & 25 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 44,44 \\ 22,77 \\ 26,73 \end{pmatrix} \tag{1.21}$$

La solution du système perturbé est $(6,23 \qquad 4,73 \qquad 4,69)$. Remarquons qu'une erreur de $\frac{1}{100}$ sur les données entraîne une erreur relative de l'ordre de 5 sur la solution : les composantes du vecteur solution sont multipliées par 5. De même, si on perturbe les éléments de la matrice en considérant le système

$$\begin{pmatrix} 23,23 & 9,09 & 12,12 \\ 12,12 & 9,9 & 1,01 \\ 14,14 & -11,88 & 25,25 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 44 \\ 23 \\ 27 \end{pmatrix}, \tag{1.22}$$

on constate qu'une erreur de $\frac{1}{100}$ sur les données provoque une erreur de l'ordre de 6. L'amplification des erreurs relatives est d'environ 600. En effet, la solution du système est $(6,89 \qquad 5,56 \qquad 5,40)$.

Envisageons ce problème du point de vue algébrique. Soit $\mathbf{A}$ une matrice inversible et $\mathbf{Ax} = \mathbf{b}$ un système linéaire. Étudions la perturbation

$$(\mathbf{A} + \delta\mathbf{A})(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{1.23}$$

où $\delta\mathbf{A}$ et $\delta\mathbf{b}$ sont les perturbations sur $\mathbf{A}$ et $\mathbf{b}$ dues aux erreurs d'arrondi et $\delta\mathbf{x}$ l'erreur commise sur la solution du système linéaire. Comme $\mathbf{Ax} = \mathbf{b}$, il vient

$$(\mathbf{A} + \delta\mathbf{A})(\delta\mathbf{x}) = \delta\mathbf{b} - \delta\mathbf{A} \cdot \mathbf{A}^{-1} \cdot \mathbf{b}. \tag{1.24}$$

Si la matrice $\mathbf{I} + \mathbf{A}^{-1}.\delta\mathbf{A}$ est inversible, alors :

$$\delta\mathbf{x} = \left(\mathbf{I} + \mathbf{A}^{-1} \cdot \delta\mathbf{A}\right)^{-1} \cdot \mathbf{A}^{-1} \cdot \left(\delta\mathbf{b} - \delta\mathbf{A}.\mathbf{A}^{-1}.\mathbf{b}\right) \tag{1.25}$$

D'où la majoration

$$\|\delta\mathbf{x}\| \leq \frac{\|\mathbf{A}^{-1}\|.\left(\|\delta\mathbf{b}\| + \|\delta\mathbf{A}\|.\|\mathbf{A}^{-1}.\mathbf{b}\|\right)}{1 - \|\mathbf{A}^{-1}\|.\|\delta\mathbf{A}\|}. \tag{1.26}$$

Comme $\|\mathbf{A}^{-1}.\mathbf{b}\| = \|\mathbf{x}\| \geq \frac{\|\mathbf{b}\|}{\|\mathbf{A}\|}$, on a une majoration de l'erreur relative

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1 - \kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}} \left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}\right) \tag{1.27}$$

si la constante $\kappa(\mathbf{A}) = \|\mathbf{A}\|.\|\mathbf{A}^{-1}\|$ vérifie $\kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} < 1$.

Soit $\mathbf{A}$ une matrice inversible, on appelle *conditionnement* de $\mathbf{A}$ le nombre

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|.\|\mathbf{A}^{-1}\|. \tag{1.28}$$

Ce nombre depend du choix de la norme : il y a autant de définitions du conditionnement que de normes matricielles :

- Pour la norme standard, appelée la *1-norme*, notée $\|\cdot\|_1$ et définie par

$$\|\mathbf{A}\|_1 = \sup_j \sum_i |a_{ij}|, \tag{1.29}$$

  c'est-à-dire le maximum de la somme des modules des éléments d'une colonne, le nombre de conditionnement est

$$\mathrm{cond}_1(\mathbf{A}) = \|\mathbf{A}\|_1 \cdot \|A^{-1}\|_1. \tag{1.30}$$

- Pour la *2-norme* notée $\|\cdot\|_2$ et définie par

$$\|\mathbf{A}\|_2 = \|\mathbf{A}^*\|_2 = \sqrt{\rho(\mathbf{A}\mathbf{A}^*)} = \sqrt{\rho(\mathbf{A}^*\mathbf{A})} \tag{1.31}$$

  où $\rho(\mathbf{A})$ est le rayon spectral de $\mathbf{A}$, c'est-à-dire le plus grand des modules des valeurs propres de $\mathbf{A}$, le conditionnement est

$$\mathrm{cond}_2(\mathbf{A}) = \|\mathbf{A}\|_2 \cdot \|\mathbf{A}^{-1}\|_2. \tag{1.32}$$

  La *2-norme* vérifie les inégalités

$$\|\mathbf{A}\|_2 \leq \|\mathbf{A}\|_e \leq \sqrt{n}\|\mathbf{A}\|_1. \tag{1.33}$$

- Pour la *norme eucludienne* ou *norme de Frobenius*

$$\|\mathbf{A}\|_e = \sqrt{\sum_{i,j} |a_{ij}|^2}\ , \tag{1.34}$$

  le nombre de conditionnement est de la forme

$$\mathrm{cond}_e(\mathbf{A}) = \|\mathbf{A}\|_e \cdot \|\mathbf{A}^{-1}\|_e. \tag{1.35}$$

- Pour la norme $\|\cdot\|_\infty$ définie par
$$\|\mathbf{A}\|_\infty = \sup_i \sum_j |a_{ij}|, \tag{1.36}$$
le nombre de conditionnement est
$$\mathrm{cond}_\infty(\mathbf{A}) = \|\mathbf{A}\|_\infty \cdot \|A^{-1}\|_\infty. \tag{1.37}$$

Pour une matrice carrée $\mathbf{A}$ d'ordre n, le conditionnement pour la 2-norme a les propriétés suivantes :

(1) Le conditionnement est un nombre positif :
$$cond_2(\mathbf{A}) \geq 0 \tag{1.38}$$

(2) Le conditionnement de la matrice ne varie pas lorsqu'on multiplie la matrice par un scalaire :
$$\forall \alpha \in \mathbb{C},\ cond_2(\alpha\mathbf{A}) = cond_2(\mathbf{A}) \tag{1.39}$$

(3) Le conditionnement est invariant par transformation unitaire. Pour toute matrice unitaire $\mathbf{U}$, on a
$$cond_2(\mathbf{AU}) = cond_2(\mathbf{UA}) = cond(\mathbf{A}) \tag{1.40}$$

(4) Soit $\sigma^2_{\max}$ la plus grande et $\sigma^2_{\min}$ la plus petite des valeurs propres de la matrice $\mathbf{A}^*\mathbf{A}$; on a alors
$$cond_2(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}. \tag{1.41}$$

(5) Si $\mathbf{A}$ est une matrice hermitienne et si $\lambda_{\max}$ et $\lambda_{\min}$ désignent respectivement la plus grande et la plus petite des valeurs propres de $\mathbf{A}$ en valeur absolue, on a :
$$cond_2(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{1.42}$$

La vérification de ces propriétés est immédiate. La propriété (1) s'établit en considérant le produit $\mathbf{I} = \mathbf{A}.\mathbf{A}^{-1}$.
$$\|\mathbf{A}\|.\|\mathbf{A}^{-1}\| \geq \|\mathbf{A}\mathbf{A}^{-\mathbf{1}}\| = \|\mathbf{I}\| = 1. \tag{1.43}$$

La propriété (2) est une conséquence directe des axiomes de définition d'une norme. La propriété (3) résulte de
$$cond_2(\mathbf{U}.\mathbf{A}) = \|\mathbf{U}.\mathbf{A}\|.\|\mathbf{A}^{-1}\mathbf{U}^*\| = \|\mathbf{A}\|.\|\mathbf{A}^{-1}\| = cond_2(\mathbf{A}). \tag{1.44}$$
Pour démontrer la propriété (4), remarquons que
$$\|\mathbf{A}\|^2 = \rho(\mathbf{A}^*\mathbf{A}) = \sup_i \lambda_i(\mathbf{A}^*\mathbf{A}) = \sigma^2_{\max}. \tag{1.45}$$

Les matrices $\mathbf{A}^*\mathbf{A}$ et $\mathbf{A}\mathbf{A}^*$ étant semblables, on a
$$\begin{aligned} \|\mathbf{A}^{-1}\|^2 &= \rho\left((\mathbf{A}\mathbf{A}^*)^{-1}\right) = \rho\left((\mathbf{A}^*\mathbf{A})^{-1}\right) \\ &= \sup_i \lambda_i(\mathbf{A}^*\mathbf{A})^{-1} = \frac{1}{\inf_i \lambda_i(\mathbf{A}^*\mathbf{A})} = \frac{1}{\sigma^2_{\min}}. \end{aligned} \tag{1.46}$$

La propriété (5) résulte de l'égalité $\|\mathbf{A}\| = \rho(\mathbf{A})$ vérifié pour les matrices normales.

Le conditionnement mesure l'*éparpillement relatifs* ($\lambda_{\max}/\lambda_{\min}$) des valeurs propres. Dans l'exemple qui précède, les valeurs numériques donnent un conditionnement égal à $cond_2(\mathbf{A}) \simeq 645$. Si on pose
$$\mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad \delta\mathbf{x} = \begin{pmatrix} 5.23 \\ -5.73 \\ -5.69 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 44 \\ 23 \\ 27 \end{pmatrix} \text{ et } \quad \delta\mathbf{b} = \begin{pmatrix} 0.44 \\ -0.23 \\ -0.27 \end{pmatrix}, \tag{1.47}$$

les calculs montrent que

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \simeq \frac{9.62}{1.73} \simeq 5.55 \,\text{et}\, \kappa(\mathbf{A})\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \simeq 645 * 0.01 \simeq 6.45. \tag{1.48}$$

L'égalité est presque satisfaite.

Une matrice est dite *bien équilibrée* si ses vecteurs lignes et ses vecteurs colonnes ont une norme de l'ordre de grandeur de l'unité. Une matrice est dite *bien conditionnée* si son conditionnement est de l'ordre de grandeur de l'unité. Remarquons qu'une matrice équilibrée peut être mal conditionnée. La matrice suivante, matrice carrée d'ordre 100, écrite sous sa forme de Jordan

$$\mathbf{A} = \begin{pmatrix} 1/2 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1/2 & 1 & 0 & \cdots & \cdots & 0 \\ \vdots & & \ddots & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & & \vdots \\ 0 & & & 0 & 1/2 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 1/2 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1/2 \end{pmatrix} \tag{1.49}$$

est equilibrée et mal conditionnée. En effet, sa plus petite valeur propre est $\frac{1}{2}$. Son inverse est formé d'éléments b_{ij}. L'élément $b_{1,100} = 2^{100} \geq 10^{30}$ montre que $\|\mathbf{A}^{-1}\| > 10^{30}$ et comme $\|\mathbf{A}\| \geq \frac{1}{\sqrt{n}}\|\mathbf{A}\|_e > 1,1,$ on en déduit que le conditionnement de $\mathbf{A}$ excède 10^{30}.

On appelle *matrice de Hilbert* une matrice symétrique d'ordre n dont les éléments sont donnés par :

$$h_{ij} = \frac{1}{i+j-1}. \tag{1.50}$$

Pour les ordres 2 et 3, les matrices de Hilbert s'écrivent

$$\mathbf{H}_2 = \begin{pmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{pmatrix} \qquad \mathbf{H}_3 = \begin{pmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{pmatrix} \tag{1.51}$$

Les matrices de Hilbert sont des matrices mal conditionnées. Selon l'ordre de la matrice et le choix de la norme, on a les valeurs suivantes du conditionnement.

n	$cond_1$	$cond_2$	$cond_e$
2	27	19.281	19.3
3	748	524.06	526.2
4	28375	$0.6.10^{-4}$	15613.8
5	943656	$0.21.10^{-5}$	480849.1
6	29070279	$0.66.10^{-7}$	15118987.1

TABLE 1.2 – Différents nombres de conditionnement pour la matrice de Hilbert

Exercices

1. Calculer le conditionnement de la matrice

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

2. Calculer manuellement $\mathrm{cond}_1(\mathbf{H}_1)$, $\mathrm{cond}_e(\mathbf{H}_2)$, $\mathrm{cond}_2(\mathbf{H}_2)$ et $\mathrm{cond}_\infty(\mathbf{H}_2)$.
3. Montrer que le conditionnement pour la 2-norme de la matrice $(a_{i\,j})$ qui est nul partout sauf pour $a_{i\,i} = 1$ et $a_{i,\,i+1} = 2$ est toujours supérieur ou égal à 2^n.

4. On considère la matrice

$$\mathbf{A} = \begin{pmatrix} 1 + \varepsilon\cos(2/\varepsilon) & -\varepsilon\sin(2/\varepsilon) \\ -\varepsilon\sin(2/\varepsilon) & 1 - \varepsilon\cos(2/\varepsilon) \end{pmatrix}$$

où ε est un réel compris entre 0 et 1. Calculer le conditionnement de $\mathbf{A}$. Étudier le cas où ε tend vers 0. Soit $\mathbf{P}$ la matrice de passage formée des vecteurs propres de $\mathbf{A}$. Montrer que si ε tend vers 0, la matrice $\mathbf{P}$ n'a pas de limite. Calculer le conditionnement de $\mathbf{P}$. Commenter.

Chapitre 2

Initiation à linux

Le présent chapitre vise à se familiariser avec les commandes usuelles pour les opérations de routines sous l'environnement Linux. Ces commandes permettent de réaliser des tâches comme la navigation à travers les répertoires, la création de nouveaux répertoires et sous-répertoires, la création de fichiers, la copie ou la suppression de fichiers et de répertoires, la visualisation du contenu d'un répertoire, etc. A part ces opérations habituelles, beaucoup d'autres tâches sont exécutées à l'aide de commandes dans un terminale Linux. Comme on le verra au Chapitre 3, l'exécution d'un script Python se fait avec une commande où on précise le nom du fichier contenant les instructions à exécuter.

2.1 Quelques commandes Linux

- La plupart de commandes Linux nécessitent un terminal
- Pour ouvrir un terminal, on clique successivement sur

  Applications, **Accessoires**, **Terminal**

  ou on appuie simultanément sur CTRL ALT T

- Mode Edition et mode Commande

  Lorsque l'on ouvre un fichier avec la commande **vi**, on entre directement en mode commande. Pour passer en mode Edition, il faut appuyer sur I ou A. Inversement, on passe de **mode Edition** à **mode commande** en appuyant sur ÉCHAP ou ESC.

2.1.1 Ouverture et Edition de fichiers

L'ouverture ou la création d'un nouveau fichier se fait en invoquant le programme **vim** avec la commande **vi** en suivant la synthaxe ci-desssous.

■ Ouverture d'un fichier : la commande **vi**

Taper, dans un terminal, **vi nom-du-fichier.ext** où **ext** peut être remplacée l'extension du fichier. ↩

Si le fichier existe déjà, son contenu apparaît sur l'écran ; si le fichier est nouveau, un document vide apparaît sur l'écran.

■ Edition d'un fichier

— Après avoir ouvert un fichier, on peut le modifier, l'enregistrer et même le fermer

— Il faut être en mode commande pour pouvoir utiliser les *touches d'orientation*. Les touches de direction peuvent être activées avec le programme **vim-latexsuite** que vous installerez à la section 2.3.

— Il faut être en mode Edition pour pouvoir éditer quelque chose dans un fichier

— **Quelques commandes d'édition** (en mode commande SVP)

X : supprimer le caractère courant

DD : supprimer la ligne courante

:w : Enregistrer le fichier courant

:wq : Enregistrer le fichier courant puis le quitter (le fermer)

:q : Quitter (fermer) le fichier courant après l'avoir enregistré

:q ! : Quitter (fermer) le fichier courant sans l'enregistrer,

r : Pour remplacer le caractère courant,
pour éviter l'enregistrement des modifications désagréables par exemple

:5 : Accéder à la **ligne numéro 5** du fichier courant,
5 pouvant être n'importe quel autre chiffre ou nombre

/expression : Rechercher *expression* dans le fichier courant à partir de l'endroit où se trouve le curseur. Pour continuer la recherche, il faut appuyer sur [N]

2.1.2 Quelques autres commandes (dans un terminal SVP)

■ La commande **ls** sert à lister tous les fichiers et répertoires se trouvant dans le répertoire courant. Elle possède plusieurs options :

ls -a : permet de lister tous les fichiers, même ceux cachés

ls -m : quand on veut séparer les répertoires et les fichiers par des virgules

ls -t : permet de voir les dates de création des fichiers

ls -lu : permet de voir les dates et heures de création des fichiers

ls -F : peut afficher les fichiers par type (extension)

ls -s : Les fichiers sont classés par ordre de tailles décroissants

ls -x : Les fichiers sont classés par ordre d'extensions

ls -lh : Les tailles des fichiers sont données en octets, ko et Mo.

■ Création de répertoires : la commande **mkdir**

mkdir NOM-DU-REPERTOIRE [↩]

■ Copie de fichiers et de répertoires : la commande **cp**

cp fichier1 fichier2 [↩] quand on veut copier le contenu de *fichier1* (qui déjà existe) dans *fichier2* (nouveau)

cp nom-du-fichier Repertoire-de-destination [↩]

cp -r nom-du-REPERTOIRE Repertoire-de-destination [↩]

- Renommer un fichier : la commande **mv**

  mv fichier1 fichier2 ↩ quand on veut changer *fichier1* (qui déjà existe) en *fichier2* (nouveau).

- Suppression de fichiers et de répertoires : les commandes **rm** et **rmdir**

  rm nom-du-fichier ↩

  rm -r nom-du-REPERTOIRE ↩

  rmdir nom-du-REPERTOIRE ↩ quand on veut supprimer un répertoire vide.

- Les commandes **clear**, **history** et **pwd**

  clear ↩ efface l'écran

  history ↩ affiche l'historique des commandes

  pwd ↩ (Print Working Directory) affiche le chemin absolu du répertoire courant.

- La commande **cd** (Change Directory)

  cd .. ↩ Répertoire parent
  cd ~ ↩ Répertoire de base
  cd - ↩ Répertoire précédent
  **cd ** ↩ Répertoire racine
  cd rep1 ↩ fait entrer dans le répertoire **rep1** contenu dans le répertoire courant

2.2 Les commandes init, exit, logout, reboot, halt et shutdown

init 0 ↩ arrête le système
init 6 ↩ reboot le système
exit ↩ clore la session
logout ↩ ferme la session
reboot ↩ redémarre l'ordinateur
halt ↩ éteint la machine péremptoirement
shutdown ↩ éteint l'ordinateur proprement

shutdown +3 ↩ ferme l'ordinateur dans trois minutes
shutdown now ↩ ferme l'ordinateur maintenant
shutdown -r now ↩ « reboot »(réinitialise) le système
shutdown -h 10 ↩ **:** arrêt dans 10 minutes
shutdown -h 18 :30 ↩ **:** arrêt à 18h30

2.3 Commande Linux pour installer un package

La commande suivante une fois exécutée dans un terminal installera un ensemble de packages qui nous seront utiles dans ce cours. Pour l'exécuter, il faut être connecté en tant qu'utilisateur identifié avec un mot de passe (**Remarque :** Cette commande doit être écrite sur une même ligne !-).

```
1 sudo apt install build-essential python-dev python3-numpy python3-scipy
2 ipython3 python-pandas python3-sympy python3-nose python3-mpmath
3 python3-matplotlib gfortran vim-latexsuite
```

Chapitre 3

Introduction à la programmation en Python

Comme indiqué dans le paragraphe 0.3.6, programmer signifie **développer un logiciel qui est également appelé un programme**. Un programme est fait d'instructions qui indiquent à l'ordinateur ou tout autre système informatique ce qu'il faut faire. Prénons comme exemple un programme, qui à l'exécution demande à l'utilisateur de fournir son nom, et affiche le message de salutation suivant : **Bonjour Monsieur VOTRE_NOM**, où **VOTRE_NOM** sera remplacé par le nom tapé au clavier. Un tel programme peut être rédigé de la manière suivante en Python :

```
1 VOTRE_NOM = input('entre votre nom : ')
2 print("Bonjour Monsieur {}".format(VOTRE_NOM))
```

Dans ce chapitre, nous passons en revue les éléments de base du *langage Python* (variables, types, structures de contrôle, procédures et fonctions, etc.). Il importe de préciser que pour établir le présent chapitre, nous nous sommes principalement référés à [Laurent2015].

3.1 Types simples, variables, expressions et instructions Python

3.1.1 Instructions et blocs d'instructions

Dans le langage Python, chaque instruction occupe une ligne : il n'y a pas de symbole de fin, nous passons simplement à la ligne après chaque instruction.

Les blocs d'instructions, eux, ne sont pas délimités par un symbole particulier mais repérés par l'indentation des instructions :

- des instructions qui sont dans le même bloc ont le même nombre d'espaces à leur gauche,
- pour marquer le début d'un nouveau bloc, on va rajouter quelques espaces par rapport à l'instruction précédente.

Idéalement, nous aimerions utiliser la touche tabulation du clavier et que notre éditeur amène le curseur à l'endroit adéquat en produisant le nombre d'espaces nécessaires. Dans le cadre de ce cours, nous utiliserons l'éditeur **vim** de la distribution Linux, et **le nombre d'espaces pour séparer les intructions du même block sera fixé à quatre**.

3.1.2 Types et variables

Python autorise, sans déclaration aucune, la manipulation de types classiques : booléens, entiers, réels, caractères, chaîne de caractères. Une variable est décrarée par un nom à laquelle on peut affecter une valeur. Le nom d'une variable doit être différente de mots réservés à Python pour certaines significations. Pour éviter toute confusion, il faudra éviter d'utiliser par exemple **print**, **def**, **float**, **and**, etc. Dans certains cas, l'usage de ce genre de mots peut conduire à des erreurs de translation par l'interpréteur Python.

```
1 pi = 3.14 # declaration et affectation d'un nombre
2 z = 'toto' # affectation d'une chaine de caracteres
```

La ligne 1 déclare une variable « pi » et lui affecte une valeur 3.14 de type **float**, la ligne 2 déclare une variable « z » et lui affecte une valeur « toto » de type **chaîne de caractère**.

Notons la syntaxe permettant d'écrire des **commentaires** dans le code Python : le signe # indique que la suite de la ligne n'est pas destinée à Python mais à un lecteur humain.

Comme suggéré ci-dessus, le symbole = est réservé à **l'affectation d'une valeur** à une variable. Le symbole ==, lui, permet d'exprimer un **test d'égalité** qui ne modifie en rien les variables.

Concernant les caractères, il est possible de repérer un caractère par **un numéro** (son code ASCII,voir la Figre 3.1 [1]). Par exemple, les lettres de a à z en minuscules correspondent aux codes de 97 à 122. La fonction Python **chr** permet de passer de l'entier au caractère correspondant, la fonction **ord** réalise la conversion inverse.

```
1 print(chr(100)) # affichage du caractere 100, c'est un 'd'
2 print(ord('m')) # affiche le code ASCII du 'm' : 109
3 print([chr(i) for i in range(97,123)]) # affiche les codes
4                                        # ASCII de 'a' a 'z'
```

Lorsque nous souhaiterons des conversions explicites d'un type à l'autre, nous utiliserons **str** pour obtenir une chaîne de caractères (à partir d'un nombre) et **int** pour obtenir un entier (à partir d'un texte par exemple).

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

FIGURE 3.1 – Le code ascii de centains caractères

```
1 n = '6'                    # une chaine de caracteres
2 r = int(n) * 7             # convertie en entier pour calcul
3 print('reponse: '+str(r)) # et retour en chaine pour affichage
```

1. https://www.fil.univ-lille1.fr/~wegrzyno/portail/Info/Doc/HTML/seq7$_$codage$_$caracteres.html

3.1.3 Opérateurs

Nous disposons en Python des opérateurs arithmétiques classiques : +, -, *, /, // pour la division entière et % pour l'opération modulo (reste de la division entière).

Sur les chaînes de caractères, nous trouvons **l'opérateur de concaténation** noté par un plus (symbole +, comme pour l'addition entre entiers) et **un opérateur de répétition** des chaînes avec un fois (symbole *, comme pour la multiplication entre entiers).

```
1 i = 10
2 i = i + 8
3 texte = 'La variable i vaut '+str(i)
4 texte = texte+"\n"
5 print(texte)
6 print('coucou'*3) # repeter une chaine
```

Nous retrouvons également les habituels opérateurs logiques **and**, **or** et **not**.

3.1.4 Instruction de sortie et chaînes de caractères

Les affichages en Python sont réalisés à l'aide de l'instruction **print**(text,end=« final ») : elle affiche le text et termine cet affichage avec final. Si la partie end=« final » n'est pas précisée, Python utilise end=« \n », ce qui signifie que l'on passe à la ligne après l'affichage de text. Nous remarquons que cette synthaxe fonctionne en Python3 et les versions supérieures.

Enfin, notons que nous pourrons confier en une fois plusieurs éléments à afficher à l'instruction print. Voici quelques exemples qui illustrent différentes utilisations possibles de print.

```
1 print("coucou he !")  # affiche et passe a la ligne
2 print("toto","titi")  # titi suit toto, passage a la ligne
3 print("toto",end="+") # termine par un + et pas par un passage
4                       # a la ligne
5 print("titi")         # titi s'ajoute a toto et passage a la ligne
6 print()               # simple passage a la ligne
7 #formatage specialisee de chaines de caracteres
8 print("Bonjour {} le visiteur.".format("Monsieur"))
```

Le langage Python permet de délimiter les chaînes de caractères soit par des apostrophes, soit par des guillemets. Il est pratique d'utiliser les guillemets lorsque le texte contient une apostrophe, et vice-versa. Il est aussi possible d'utiliser un backslash (caractère \) pour neutraliser la signification Python d'un caractère.

```
1 prenom = 'Toto'
2 print("salut",prenom)
3 print("ca va aujourd'hui ?")
4 print('oui, ca va aujourd\'hui !')
```

3.2 Fonctions et procédures en Python

Rappelons que procédures et fonctions ont en commun d'utiliser des paramètres et de rassembler des instructions quelconques. Par contre, **une fonction doit toujours se terminer par le renvoi d'un résultat** (à l'aide du mot-clef return en Python), ce qui n'est jamais le cas pour une procédure.

Cela signifie en particulier qu'un appel à une fonction se trouvera souvent dans la partie droite d'une affectation, par contre un appel à une procédure ne pourra pas être à cette position.

3.2.1 Définition et utilisation d'une procédure

La définition d'une procédure en Python suit la sythaxe « **def nom_procedure(arg1,arg2,...)** » où **def** est un mot-clé réservé, **nom_procédure** le nom de la procédure à définir, arg1, arg2, ... les arguments ou paramètres éventuels de la procédure. Il n'est pas obligatoire qu'une procédure ait des arguments.

```
1  # forme generale d'une procedure
2  def nom_de_la_procedure(parametres):
3      # ...
4      # des instructions ici
5      # ...
6  # procedure qui affiche entre etoiles un texte donne
7  def affiche_joliment (msg):
8      print('***',msg,'***')
9  # on appelle la procedure et on renseigne son parametre
10 affiche_joliment('coucou')
```

3.2.2 Définition et utilisation d'une fonction

Il faut distinguer la définition d'une fonction et son appel comme on le voit ci-dessous. Une fonction est un moyen de regrouper les instructions afin de réaliser une tâche.

```
1  # forme generale d'une fonction
2  def nom_de_la_fonction(parametres):
3      # ...
4      # des instructions ici
5      # ...
6      return(resultat)
7  # fonction qui calcule la somme de deux entiers et la renvoie
8  def addition (x,y):
9      s = x+y
10     return(s)
11 # on appelle la fonction et on renseigne ses parametres
12 somme = addition(2,2)
13 print('le resultat est',somme)
```

Remarquons que l'appel d'une fonction se trouve à droite de l'instruction. La valeur retournée par la fonction est affectée à la variable **somme** sur la partie gauche.

3.3 Structures de contrôle en Python

Modulo les notations propres à Python, nous retrouvons les structures de contrôle classiques, en particulier celles vues en cours d'algorithmique.

3.3.1 Structure conditionnelle : le si alors sinon

L'intruction de contrôle **si alors sinon** ou **if** en anglais suis la synthaxe suivante :

```
1  # forme generale
2  if(une condition ici):
3      # ...
```

```
    # des instructions ici
    # ...
else:
    # ...
    # des instructions ici
    # ...
# affichage si un test est vrai
temperature = 28
if temperature>25:
    print('il fait chaud !')
# affichages distincts selon un test
temperature = 28
if temperature>25:
    print('il fait chaud !')
else:
    print('il fait frais...')
# enchainement de tests
temperature = 28
if temperature>25:
    print('il fait chaud !')
elif temperature>20:
    print('il fait bon.')
else:
    print('il fait frais...')
```

Remarque : Il faut noter l'espace qui commence chaque instruction faisant partie du block de **if**.

3.3.2 Structure itérative : la boucle tant que

La boucle **tant que** ou **while** en anglais est nécessaire si l'on veut exécuter un block d'instructions tant qu'une certaine condition n'est pas satisfaite.

```
# forme generale
while(une condition ici):
    # ...
    # des instructions ici
    # ...
# on compte de 1 a 10
i = 1
while (i<=10):
    print(i)
    i = i + 1
```

3.3.3 Structure itérative : la boucle pour

La fonction **range** permet de définir un intervalle qui peut être parcouru à l'aide de la *boucle for ... in ....*

```
# forme generale
for i in range(min,max):
    # ...
    # des instructions ici
    # ...
# affichage de 10 etoiles
for i in range(1,11):
```

```
8       print('*')
9   # on compte de 1 a 10
10  for i in range(1,11):
11      print(i)
```

Nous verrons plus loin une utilisation plus générale de cette boucle pour parcourir les éléments d'une séquence quelconque.

3.4 Structures de données Python : listes et dictionnaires

3.4.1 Les listes Python

Pas de contrainte sur le contenu des cases, celles-ci sont numérotées à partir de 0. Tout d'abord, nous voyons les différentes manières de créer une liste et de lui affecter des valeurs.

```
1   # creation d'une liste vide et definition de trois cases
2   l = []
3   l.append(2)
4   l.append(3.14)
5   l.append('coucou')
6   # liste fournie par range (entiers de 1 a 100)
7   premiers = range(1,101)
8   # une liste de notes
9   notes = [5,12,8,20,10] # on definit une liste
10  notes[2] = 9           # on modifie la 2eme case
```

Ensuite, Python propose plusieurs syntaxes pour parcourir une liste. Notons l'expression **len**(l) qui fournit la taille d'une liste l et la boucle **for in** dédiée au parcours de listes.

```
1   # affichage des elements de la liste...
2   # ... en utilisant les indices et la longueur de la liste
3   for i in range(0,len(notes)) :
4       print(notes[i])
5   # ... et plus simplement :
6   for note in notes:
7       print(note)
8   # ... et pourquoi pas :
9   print(notes)
```

Opérations Python dédiées aux listes :

- **list** permet de copier une liste existante pour en créer une nouvelle,
- **remove** efface la première occurrence d'un élément dans une liste,
- **del** supprime une case,
- **insert** ajoute un élément à une position donnée d'une liste,
- **append** ajoute un élément en fin de liste,
- **pop** supprime et fournit le dernier élément d'une liste.

```
1   notes = [5,12,9,20,12,10] # definition d'une liste
2   l = list(notes)           # copier la liste
3   l.append(12)              # ajout de 12 en fin de liste
```

```
l.insert(3,14)              # ajout de 14 en position 3
del(l[2])                   # suppression de la case 2
l.remove(12)                # suppression du premier 12
l.pop()                     # suppression du dernier element
print notes                 # donne [5, 12, 9, 20, 12, 10]
print l                     # donne [5, 14, 20, 12, 10]
```

Il est possible de parcourir simultanément deux listes de même longueur.

```
liste1 = [5,8,9,2,2,4] # definition de la premiere liste
liste2 = [1,7,9,10,2,11] # definition de la deuxieme liste
for elem1, elem2 in zip(liste1,liste2):
    print('{}\t\t{}'.format(elem1,elem2))
```

Retour sur les chaînes de caractères : il est parfois commode de voir les chaînes de caractères comme des listes de caractères, et effectivement Python permet d'appliquer aux chaînes les opérations dédiées aux listes.

```
# definition d'une chaine de caracteres
phrase = "coucou"
# parcours d'une chaine caractere par caractere
for l in phrase:
    print(l)
print()
# autre parcours caractere par caractere
for i in range(0,len(phrase)):
    print(phrase[i],end="-")
print()
```

3.4.2 Les dictionnaires Python

Les *dictionnaires* sont des listes dont la particularité est que les cases ne sont plus indicées par des numéros mais par des objets plus complexes, comme des textes. Ces textes qui servent d'indices sont appelés les clefs du dictionnaire. A chaque clef est associé une valeur.

Premiers exemples avec la définition d'un carnet d'adresses mails, les clefs du dictionnaire sont les noms des personnes :

```
# definition d'un dictionnaire vide
mails = {}
# definition du mail de Titi
mails['Titi'] = 'titi@univ.fr'
# definition du mail de Toto
mails['Toto']  = 'toto.machin@free.fr'
# affichage du mail de Toto
print('Mail de toto :',mails['Toto']);
```

Autre exemple, l'implémentation d'un type abstrait «produit», ici un type «livre» :

```
# definition d'un premier livre
livre1 = {}
livre1['titre'] = 'Apprendre Python'
livre1['annee'] = 2010
print(livre1['titre'])
# autre syntaxe pour un second livre
livre2 = { 'titre': 'Algorithmique' , 'annee': 1970 }
# procedure dediee a ce type
def affiche_livre (l):
    print('titre :',l['titre'])
    print('annee :',l['annee'])
affiche_livre(livre1)
```

Opérations Python sur les dictionnaires :

- **keys** donne les clefs d'un dictionnaire,
- **values** donne les valeurs d'un dictionnaire,
- **items** fournit clefs et valeurs du dictionnaire sous forme de couples,
- **len** donne le nombre de couples dans le dictionnaire,
- **get** extrait la valeur associée à une clef, comme on le fait avec la notation crochets mais permet de prévoir une valeur par défaut pour le cas où la clef n'existe pas dans le dictionnaire,
- **in** dit si la clef existe ou non dans le dictionnaire,
- **del** permet de supprimer une entrée du dictionnaire.

```
# definition d'un dictionnaire
d = { 'Titi':'titi@univ.fr' , 'Toto':'toto.truc@free.fr' }
print(d.keys())   # ['Toto', 'Titi']
print(d.values()) # ['toto.truc@free.fr', 'titi@univ.fr']
print(d.items())
# [('Toto','toto.truc@free.fr') , ('Titi','titi@univ.fr')]
# parcours de clefs et affichage des couples nom/mail
for nom in d:
    print(nom,':',d[nom])
print(len(d)) # 2
print('Toto' in d) # True
print('Lulu' in d) # False
print(d['Toto']) # toto.truc@free.fr
print(d['Lulu']) # KeyError: 'Lulu'
print(d.get("Toto","contact inconnu")) # toto.truc@free.fr
print(d.get("Lulu","contact inconnu")) # contact inconnu
del(d['Toto']) # suppression des infos sur Toto
print(d.get("Toto","contact inconnu")) # contact inconnu
```

3.5 Quelques librairies Python

3.5.1 Accès aux éléments d'une librairie

A la section section 2.3, vous avez pu installer un certain nombre de librairies qui sont couramment utilisées en Python. Ces librairies ne sont pas disponibles par défaut. Si l'on veut exploiter leurs possibilités l'utilisateur doit procéder à leur installation. Pour notre cas, il s'agit entre

autre de la librairie **numpy**, **scipy**, **matplotlib**, **sympy**. Pour utiliser une librairie, plusieurs syntaxes sont possibles pour avoir accès aux fonctions qu'elle fournit.

```
# on va utiliser des fonctions de la librairie
import librairie
...
# on precise a nouveau le nom de la librairie
# avec chaque utilisation d'une fonction de cette librairie
librairie.fonction()
...
```

```
# nous precisons la fonction qui nous interesse
from librairie import fonction
...
# il n'est pas necessaire de rappeler le nom de la librairie
fonction()
...
# plusieurs fonctions nous interessent...
from librairie import f1,f2
...
f1()
...
f2()
...
# toutes les fonctions de la librairie nous interessent...
from librairie import*
...
f1()
...
f2()
...
f3()
...
```

3.5.2 Gestion de l'aléatoire : la librairie random

- **random.randint** choisit aléatoirement un entier dans un intervalle donné,
- **random.randrange** choisit aléatoirement entre 0 et un entier donné (exclu), permet aussi de préciser une borne inférieure autre que 0 et un pas de progressions,
- **random.choice** fournit aléatoirement un élément parmi ceux d'une liste,
- **random.shuffle** mélange sur place les éléments d'une liste.

```
import random
print(random.randint(1,10))        # un entier entre 1 et 10
print(random.randrange(10))        # un entier entre 0 et 9
print(random.randrange(10,100,2)) # entier pair entre 10 et 98
notes = [5,12,8,20,10]             # on definit une liste
print(random.choice(notes))        # une note au hasard
print(notes)
random.shuffle(notes)              # melange aleatoire
print(notes)
```

3.5.3 Time

En Python, il est possible de mesurer le temps d'exécution d'un block d'instructions en utilisant les fonctions de la librairie « **Time** ».

```
1 import time
2 t1 = time.process_time()
3 ...
4 # ici les instructions a chronometrer
5 ...
6 t2 = time.process_time()
7 print((t2 - t1),'sec.')
```

3.5.4 Gestion du système : sys

Si nous avons un algorithme récursif qui nécessite d'augmenter le nombre de récursions autorisées, cela se fait de cette manière :

import sys
sys.setrecursionlimit(2000)

3.6 Les fichiers en Python

Voici les instructions Python de base pour créer un fichier texte :

```
1 # ouverture du fichier en ecriture
2 f = open('monfichier.dat','w')
3 # ecritures dans le fichier
4 f.write("coucou ! ") # on ecrit coucou dans le fichier
5 f.write(str(12))     # on convertit 12 en texte et on l'ecrit
6 f.write("\n")        # on passe a la ligne dans le fichier
7 f.close() # fermeture du fichier
```

write ne traite que des chaînes de caractères, nous devons donc convertir tout autre type de données en chaîne à l'aide de la fonction **str**.

La lecture de données à partir d'un fichier sera traitée à la Section 3.9 qui est dédiée à la manipulmation de données pour faire une représentation graphique.

3.7 Les matrices avec Python

3.7.1 Utilisation du module numpy

Il existe un module additionnel à Python nommé **numpy** permettant de créer et d'effectuer des opérations sur les matrices. Ce module est librement téléchargeable sur internet à l'adresse suivante : `http://numpy.scipy.org/`. Les instructions décrites ci-dessous ne représentent qu'une partie des possibilités de ce module. Pour utiliser ce module il faut en tête de votre programme saisir :
from numpy import*.

La synthaxe précédente charge la totalité du module **numpy**. Si on n'a besoin que de quelques fonctions, il est possible de ne charger que ce dont on a besoin. Par exemple, la fonction **sqrt**

de numpy pour calculer la racine carrée d'un nombre réel peut être chargée comme suit : **from numpy import sqrt**.

3.7.1.1 Créer des matrices

a. Créer une matrice en saisissant une à une les valeurs de chacun des termes. L'instruction est « array ».

$$array([\underbrace{(\ldots,\ldots,\ldots)}_{1 \text{ ère ligne}},\underbrace{(\ldots,\ldots,\ldots)}_{2 \text{ ème ligne}},\ldots,\underbrace{(\ldots,\ldots,\ldots)}_{\text{dernière ligne}}])$$

Le séparateur pour les différentes valeurs ou pour les différentes lignes est « , ». Exemple :

$array([(1,2,3),(4,5,6)])$ crée la matrice 2×3 : $\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$

résultat à l'affichage :

$$[[1,2,3]$$
$$[4,5,6]]$$

b. Créer une matrice ne contenant que des 0, ou bien que des 1. Les instructions sont « zeros » et « ones ».
Pour une matrice à une seule ligne :

$$\text{zeros}\,(\textit{nombre de colonnes}) \quad \text{et} \quad \text{ones}\,(\textit{nombre de colonnes})$$

Pour les autres matrices :

$$\text{zeros}\,((\textit{nombre de lignes}, \textit{nombre de colonnes}))$$

$$\text{et} \quad \text{ones}\,((\textit{nombre de lignes}, \textit{nombre de colonnes}))$$

Exemples : zeros (5) crée la matrice 1×5 : $(0,0,0,0,0)$

ones $((2,3))$ crée la matrice 2×3 : $\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix}$.

c. Créer une matrice avec des valeurs séparées d'un pas régulier. L'instruction est « arange ».
Pour les matrices avec les entiers consécutifs de 0 à $n-1$:
les matrices à une seule ligne et contenant les entiers de 0 à $n-1$: arange (n)
les autres matrices $a\times b$ et contenant les entiers 0 à $n-1$:
arange (n) $.reshape(a,b)$

Pour les matrices avec les réels débutant à d, finissant à f et séparées d'un pas p :
les matrices à une seule ligne : arange (d,f,p)
pour les autres matrices $a\times b$: arange (d,f,p) $.reshape(a,b)$

exemples : arange (4) crée la matrice 1×4 : (0 1 2 3)

arange (6) $.reshape\,(2,3)$ crée la matrice 2×3 : $\begin{pmatrix}0 & 1 & 2\\ 3 & 4 & 5\end{pmatrix}$,

arange $(1,16,4.2)$ crée la matrice 1×4 : (1 5.2 9.4 13.6),

arange $(1,16,4.2)$ $.reshape(2,2)$ crée la matrice 2×2 : $\begin{pmatrix}1 & 5.2\\ 9.4 & 13.6\end{pmatrix}$,

d. Créer une matrice $a \times b$ diagonale avec des 1. L'instruction est « eye » avec la synthaxe : $eye\left(a, b\right)$.

exemple : eye $(2,3)$ crée $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, eye $(3,3)$ crée $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

eye $(4,3)$ crée $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$,

3.7.1.2 Accéder aux termes d'une matrice

Il faut retenir que les lignes et les colonnes sont numérotées à partir de 0.

a. Accéder à un terme d'une matrice A. Synthaxe : A [numéro de ligne, numéro de colonne]
exemple : Si $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ alors $A[1,2]$ correspond au terme 6.

b. Accéder à une ligne d'une matrice A. Synthaxe : A [numéro de ligne]
exemple : Si $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ alors $A[1]$ correspond à la ligne numéro 1
Résultat : [4 5 6]

c. Accéder à une colonne d'une matrice A. Synthaxe : A [:, numéro de colonne]
exemple : Si $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ alors $A[:,1]$ correspond à la colonne numéro 1
Résultat : $[2,5]$

3.7.1.3 Opérations sur les matrices

a. Effectuer une même opération sur tous les termes d'une matrice. Avec $A = \begin{pmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{pmatrix}$.

Multilier tous les termes $2 * A$ donne $\begin{pmatrix} 2. & 4. & 6. \\ 8. & 10. & 12. \end{pmatrix}$,

Diviser tous les éléments $A/2$ donne $\begin{pmatrix} 0.5 & 1. & 1.5 \\ 2. & 2.5. & 3. \end{pmatrix}$,

Additionner ou soustraire à tous les termes : $A - 1$ donne $\begin{pmatrix} 0. & 1. & 2. \\ 3. & 4. & 5. \end{pmatrix}$,

Mettre tous les termes au carré : $A * *2$ donne $\begin{pmatrix} 1. & 4. & 9. \\ 16. & 25. & 36. \end{pmatrix}$,

Calculer le sinus de tous les termes :

$\sin(A)$ donne $\begin{pmatrix} 0.84147098 & 0.90929743 & 0.14112001 \\ -0.7568025 & -0.95892427 & -0,2794155 \end{pmatrix}$.

b. Effectuer les opération avec les matrices

Avec $A = \begin{pmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{pmatrix}$, $B = \begin{pmatrix} 3. & 2. & 4. \\ 1. & 5. & 6. \end{pmatrix}$, $C = \begin{pmatrix} 1. & 3. \\ 2. & 4. \\ 2. & 5. \end{pmatrix}$ et $D = \begin{pmatrix} 3. & 2. \\ 1. & 2. \end{pmatrix}$.

Effectuer le produit, terme à terme, de deux matrices :	$A * B$	donne $\begin{pmatrix} 3. & 4. & 12. \\ 4. & 25. & 36. \end{pmatrix}$
Multiplier deux matrices :	dot (A, C)	donne $\begin{pmatrix} 11. & 26. \\ 26. & 62 \end{pmatrix}$
Additionner ou soustraire deux matrices :	$A + B$	donne $\begin{pmatrix} 4. & 4. & 7. \\ 5. & 10. & 12. \end{pmatrix}$
Inverser une matrice :	linalg.inv (D)	donne $\begin{pmatrix} 0.5 & -0.5 \\ -0.25 & 0.75 \end{pmatrix}$
Transposer une matrice :	D.transpose()	donne $\begin{pmatrix} 3. & 1. \\ 2. & 2. \end{pmatrix}$

Pour calculer une puissance d'une matrice, voici une fonction que l'on peut définir en tête de programme :

```
1 def puissance(mat,exp):
2     m = mat
3     for i in range(1,exp):
4         mat = dot(mat,m)
5     return mat
```

ainsi, puissance(D,2) donne $\begin{pmatrix} 11. & 10. \\ 5. & 6. \end{pmatrix}$.

3.8 Créer une matrice avec les listes

On peut créer une matrice avec deux boucles « for » imbriquées. On obtient alors des listes de listes. Etant donné l'existence du module **« numpy »**, nous ne proposerons pas ici la programmation pour réaliser les différentes opérations sur ces matrices. Cela peut éventuellement faire l'objet d'exercices.

Exemple : On veut créer la matrice 3 lignes par 4 colonnes : $\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$

```
1 M = [[0 for j in range(0,4)] for i in range(0,3)]
2
3 M[0][0] = 4
4
5 M[1][2] = 8
6
7 print(M)
```

Résultat : $[4, 0, 0, 0]$, $[0, 0, 8, 0]$, $[0, 0, 0, 0]$

Commentaires :

— « for j in range(0,4) » crée les colonnes numérotées de 0 à 3, et « for i in range(0,3) » crée les lignes numérotées de 0 à 2.
— L'élément de la ligne numéro 1 et de la colonne numéro 3 est identifié par : $M\,[1]\,[3]$
On peut alors ainsi fabriquer une fonction pour créer des matrices d'une taille définie.
Exemple. Pour créer des matrices 2×3 :

```
1 def matrice2x3():
2     return [[0 for j in range(0,2)] for i in range(0,3)]
3 A = matrice2x3()
4 print("A = ", A)
```

Résultat : $[[0,0,0],[0,0,0]]$
On peut alors aussi fabriquer une fonction pour créer des matrices d'une taille non définie.
Exemple. Pour créer des matrices $i \times j$:

```
1 def matrice(i,j):
2     return [[0 for q in range(0,j)] for p in range(0,i)]
3 A = matrice(3,4)
4 print("A = ", A)
```

Résultat : $[[0,0,0,0],[0,0,0,0],[0,0,0,0]]$

3.9 Représentation graphique de données en Python

Matplotlib est un package Python dédié à produire des graphiques en deux dimension. Dans son module pyplot, on y trouve un ensemble de fonctionalités utiles pour produire des graphiques de qualité. Dans cette section, nous allons explorer quelques fontions de base faire une représenattaion graphique de données. Les données peuvent être générées à partir d'une fonction mathématique ou lues à partir d'un fichier, par exemple les données obtenues par mesure d'une grandeur physique au laboratoire. Aux point 3.9.1 nous générerons les données à partir d'une fonction mathéematique alors que au point 3.9.3, nous verons command s'aquérir de données à partir d'un fichier et faire une représetation graphique.

3.9.1 Représentation graphique d'une fonction mathématique

Le conde suivant permet de faire une représenation graphique de la distribution de Gauss $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ avec les paramètres $\mu = 0$ et $\sigma = 0.5$.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x = np.arange(-2,2,0.01)
4 sigma =0.5
5 pi = np.pi
6 mu = 0
7 y = (1/(sigma*np.sqrt(2*pi)))*np.exp(-(x-mu)**2/(2*sigma**2))
8 plt.plot(x,y)
9 plt.show()
```

On peut ajouter un title à la figure, annoter les axes et une legende en complétant le code précédent par les instructions suivantes :

```
1
2  import matplotlib.pyplot as plt
3  import numpy as np
4  x = np.arange(-2,2,0.01)
5  sigma =0.5
6  pi = np.pi
7  mu = 0
8  y = (1/(np.sqrt(2*pi)*sigma**2))*np.exp(-(x-mu)**2/sigma**2)
9  plt.plot(x,y,label='Gauss') #Ajout d'une chaine de
10                             #caractere pour la legende
11 #Annoter une figure
12 plt.xlabel(r'Les valeurs de $x$ [-2,2]')
13 ylabel = r'$f\left(x\right)$'
14 plt.ylabel(ylabel)
15 plt.title('Courbe en deux dimensions')
16 plt.legend() #Positionnement de la legende
17 plt.grid(True)
```

3.9.2 Représentation de plusieurs graphiques dans une même figure

Dans cette section, nous explorons la possibilité de faire une représentation graphique dans une même figure en y ajoutant d'autres propriétés comme la couleur et le type du trait de la courbe, les marqueurs, l'annotation des axes, l'ajout du titre sur la figure, etc.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  x = np.arange(-2,2,0.1)
4  sigma =0.5
5  pi = np.pi
6  mu = 0
7  y = (1/(np.sqrt(2*pi)*sigma**2))*np.exp(-(x-mu)**2/sigma**2)
8  #Creer 4 axes pour les figures
9  fig,((ax1,ax2),(ax3,ax4)) = plt.subplots(2,2)
10 ax1.plot(x,y,'r--',linewidth=6,label='Gauss') #Ajout d'une chaine
11                                              #de caractere pour la legende
12 ax1.set_xlabel(r'Les valeurs de $x$ [-2,2]',
13 fontsize=14,fontdict={'color':'red'})
14 ax1.set_ylabel(r'$f\left(x\right)$',
15                fontsize=14,fontdict={'color':'red'})
16 ax1.set_title('Courbe 1')
17 ax1.legend() #Positionnement de la legende
18 ax1.grid(True)
19 ax2.plot(x,y,'g.',linewidth=4,label='Gauss')
20 ax2.set_xlabel(r'Les valeurs de $x$ [-2,2]',fontsize=18)
21 ax2.set_ylabel(r'$f\left(x\right)$',
22                fontsize=18,fontdict={'color':'green'})
23 ax2.set_title('Courbe 2')
24 ax2.legend() #Positionnement de la legende
```

```
25 ax2.grid(True)
26 ax3.plot(x,y,'m*',linewidth=5,label='Gauss')
27 ax3.set_xlabel(r'Les valeurs de $x$ [-2,2]',fontsize=12)
28 ax3.set_ylabel(r'$f\left(x\right)$',
29                 fontsize=12,fontdict={'color':'magenta'})
30 ax3.set_title('Courbe 3')
31 ax3.legend() #Positionnement de la legende
32 ax3.grid(True)
33 ax4.plot(x,y,linewidth=2,marker='o',markersize=8,label='Gauss')
34 ax4.set_xlabel(r'Les valeurs de $x$ [-2,2]',fontsize=16,
35 fontdict={'color':'blue'})
36 ax4.set_ylabel(r'$f\left(x\right)$',fontsize=16)
37 ax4.set_title('Courbe 4')
38 ax4.legend() #Positionnement de la legende
39 ax4.grid(True)
40 plt.tight_layout() #Pour le redimensionnement
41                   #automatique de la figure
```

La ligne 9 du code précédent permet de créer une figure contenant quatre systèmes d'axes sous forme de grille (2 lignes , 2 colonnes). L'accès à chaque figure se fait avec la vrariable d'axe correspondante.

3.9.3 Représentation graphique de données à partir d'un fichier

La lecture de données dans un fichier peut se faire de différentes façons en fonction de la structure du fichier. Par exemple si les données sont enregistrées dans un **fichier texte** sous forme de tableau comme celui de la Figure 3.2, la lecture peut se faire à l'aide de la fonction **loadtxt** du module **numpy**. Cette fonction retourne une variable de type **nump.ndarray** dont les lignes sont les colonnes lues dans le fichier. Dans notre cas, les données sont sous forme de tableau de 4 colonnes et un certain nombre de lignes. Dans le code suivant, nous verons comment on peut lire un tel fichier. Signalons qu'il est possible de lire des colonnes au choix ou toutes les données. Nous supposons que le fichier de données est appelé **data_file.dat**.

```
0.003888356     0.994148000     0.993479000     0.995117000
0.043283802     0.943050000     0.939429000     0.951840000
0.093239886     0.888487000     0.884088000     0.904719000
0.123213537     0.859263000     0.855003000     0.879123000
0.323037874     0.705683000     0.705356000     0.740395000
0.522862211     0.595969000     0.599629000     0.636540000
0.722686548     0.512289000     0.518668000     0.554280000
0.922510885     0.446101000     0.454057000     0.487120000
1.122335220     0.392439000     0.401112000     0.431179000
1.322159560     0.348124000     0.356900000     0.383898000
......          ......          .....           ......
```

FIGURE 3.2 – Données sous forme d'un tableau

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 data = np.loadtxt('./data/data_file.dat',
4                    unpack=True,usecols=[0,1]) #Lecture de la
5                  #premiere et de la deuxieme colonne du tableau
6 x,y = data
```

```
7  #Repesentation graphique de donnees extraites dans
8  # les variables x et y
9  fig,ax = plt.subplots()
10 ax.plot(x,y)
11 ax.set_ylabel('y')
12 ax.set_xlabel('x')
13 ax.set_title('Representation graphique de donnes lues dans un fichier')
```

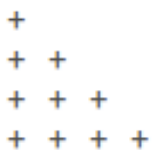
La fonction **loadtxt** reçoit en argument le nom ou le chemin d'accès au fichier et quelques options. Si l'option **unpack** est fixées à la valeur **True**, le tableau retourné est transposé et par conséquent, les colonnes deviennent les lignes du tableau. L'option **usecols** permet de spécifier les numéros de colonnes à lire. Notons que la première colonne porte le numéro zéro. On peut récupérer les données dans d'autres variables comme à la ligne 5 du code précédent. De façon générale si on a lu un nombre n de colonnes, l'accès au résultat de la lecture peut se faire de la manière suivante x1,x2,x3,... = data. Après extraction de données on peut procéder comme à la Section 3.9.2 pour faire une représentation graphique.

Notons, pour terminer cette section, que le module **matplotlib** peut être utilisé pour faire une représentation graphique en trois dimensions. Pour en savoir plus sur toutes ses fonctionnalités, vous pouvez consulter la page dédié à ce module [2]. Les autres possibilités offertes par le module **numpy** pour la lecture des fichiers peuvent être explorées en consultant la page suivante [3].

2. https://matplotlib.org/
3. https://numpy.org/

3.10 Exercices

1. Modifier l'expression suivante pour avoir comme résultat 16 : *print(5 + 3*2)*.
2. Quel sera le résultat de *print(type(4.5))* ?
3. Afficher tous les nombres pairs jusqu'à 30. Terminer l'affichage par * si le nombre est divible par 6. (Hint : Utiliser l'opérateur modulo %).
4. Ecrire un code Python pour enlever les deux dernières caractères de « Je suis une longue chaîne de caractères » sans compter le nombre de caractères. (Hint : Utiliser l'indexation négative).
5. s = '012345' . (a) Utiliser l'indexation pour enlever les deux derniers éléments (b) Pour enlever les deux premiers éléments.
6. $a = [1, 2, 3, 4, 5]$. Utiliser l'indexation et la multiplication pour générer $[2, 3, 4, 2, 3, 4]$.
7. Compare les résultats suivants : $5//2,\ 5.0/2$ et $2.0/3$.
8. Afficher les résultat suivants en utilisant la boucle **while**



9. Ecrire un programme pour lire en entrée par exemple $8A$, $10A$ et affiche la partie entière et littérale séparément.
10. Ecrire un programme pour afficher un nombre en binaire. (par exemple 5 sera affiché comme 101).
11. Ecrire un programme pour afficher tous les cubes parfaits jusqu'à 2000.
12. Ecrire un programme Python pour afficher la table de multiplication de 5.
13. Ecrire un programme pour calculer le volume d'une boîte de côtés 3, 4 et 4 inches en cm^3. (1 inch = 2.54 cm)
14. Ecrire un programme pour calculer le pourcentage de volume occupé par une sphère de diamètre r incluse dans un cube de côté r. Fournir en entrée r.
15. Ecrire un programme Python pour calculer la surface d'un cercle.
16. Ecrire un programme Python pour calculer la division d'un entier sans utiliser l'opérateur / (Hint : utiliser l'opérateur $-$).
17. Ecrire un programme pour compter le nombre de fois que la lettre « a » apparaît dans une chaîne de caractères entrée au clavier. Le programme continu à demander à l'utilisateur d'entrer une chaîre tant que la chaîne réçue en entrée ne contient pas la lettre « a ».
18. Ecrire un programme informatique pour la division entière qui demande à l'utilisateur deux entiers et affiche le résultat. Il pourra donner le résultat en deux parties : le quotient de la division entière et le reste de la division. Par exemple si l'utilisateur entre 11/4, l'ordinateur pourra afficher le résultat 2 et le reste 3.
19. Modifier le programme précédent pour éviter la division par zéro.
20. Ecrire un programme pour calculer la somme des nombres entrés au clavier par l'utilisateur. Le programme pourra continuer à demander à l'utilisateur d'entrer des nombres et afficher le résultat de la sommation après chaque étape. Il pourra se terminer si l'utilisateur frappe zéro.

21. Modifier le programme précédent pour utiliser **raw_input()** pour gérer les erreurs tel que l'entrée par l'utilisateur d'un mauvais caractère.
22. Ecrire un script pour convertir les degrés Celcius en degrés Fahrenheit. Le programme pourra demander à l'utilisateur d'entrer la température en Celcius et afficher le résultat en Fahrenheit. Utiliser la formule de conversion suivante : $f = \frac{9}{5}c + 32$.
23. Ecrire un script pour convertir les Fahrenheit en Celcius.
24. Ecrire un script qui utilise une variable et affiche 20 fois la phrase « **I will not talk in class** » sur chaque ligne.
25. Définir $2 + 5j$ et $2 - 5j$ comme des nombres complexes et calculer leur produit. Vérifier le résultat en définissant séparement la partie réelle et la partie imaginaire et utiliser la formule de multiplication.
26. Afficher la table de multiplication de 12 en utilisant la boucle **while**.
27. Afficher la table de multiplication d'un nombre réçu au clavier en utilisant la boucle **for**.
28. Afficher la puissance de 2 jusqu'à 1024 (Utiliser 2 lignes uniquement).
29. Définire la liste $a = [123, 12.4, \text{'haha'}, 3.4]$.
    a. Afficher tous les éléments de a en utilisant la boucle **for**,
    b. Afficher les éléments de type **float** (utiliser la fonction **type()**),
    c. Insérer un élément après l'élément 12.4,
    d. Ajouter plus d'éléments à la fin de la liste.
30. Construire une liste contenant 10 éléments en utilisant la boucle **for**.
31. Générer une table de multiplication de 5 en utilisant un code Python de deux ligne.
32. Ecrire un programme pour trouver la somme de 5 nombres lus au clavier.
33. Ecrire un programme pour lire les nombres au clavier jusqu'à ce que leur somme dépasse 200. Modifier le programme pour ignorer les nombres supérieurs à 99.
34. Ecrire une fonction Python pour calculer le PGCD de deux nombres.
35. Ecrire un **code python** pour calculer les nombres de conditionnement du tableau 1.3.3. Il est recommandé d'utiliser les **formules de Faddev ou de Leverrier** (voir 4.1.5.2) pour calculer l'inverse d'une matrice régulière.

Chapitre 4

Systèmes linéaires [Alfio2004]

On appelle système linéaire d'ordre n (n entier positif), une expression de la forme

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.1}$$

où $\mathbf{A} = (a_{ij})$, $1 \leq i,\, j \leq n$, désigne une matrice de taille $n \times n$ de nombres réels ou complexes, $\mathbf{b} = (b_i)$, $1 \leq i \leq n$, un vecteur colonne réel ou complexe et $\mathbf{x} = (x_i)$, $1 \leq i \leq n$, est le vecteur des inconnues du système. La relation précédente équivaut aux équations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad i = 1, \cdots, n\,. \tag{4.2}$$

La matrice $\mathbf{A}$ est dite *régulière* (ou *non singulière*) si $det(\mathbf{A}) \neq 0$; on a existence et unicité de la solution $\mathbf{x}$ (pour n'importe quel vecteur $\mathbf{b}$ donné) si et seulement si la matrice associée au système linéaire est régulière. Théoriquement, si $\mathbf{A}$ est non singulière, la solution est donnée par *la formule de cramer* :

$$x_i = \frac{det\,(\mathbf{A}_i)}{det\,(\mathbf{A})}, \quad i = 1, \cdots, n \tag{4.3}$$

où $\mathbf{A}_i$ est la matrice obtenue en remplaçant la i-ème colonne de $\mathbf{A}$ par le vecteur $\mathbf{b}$. Cependant l'application de cette formule est inacceptable pour la résolution pratique des systèmes, car son coût est de l'ordre de $(n+1)!$ *floating-point operations (flops)*. En fait, le calcul de chaque déterminant par la formule

$$det(\mathbf{A}) = \sum_{\sigma} (-1)^{\epsilon(\sigma)} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{4.4}$$

(où la somme est étendue à toutes les permutations σ sur n objets et $\epsilon(\sigma)$ désigne la **signature** [1] de la permutation σ) requiert $n!$ flops. Par example, sur un ordinateur effectuant 10^9 flops par seconde, il faudrait $9,6.10^{47}$ années pour résoudre un système linéaire de seulement 50 équations. Il faut donc développer des algorithmes alternatives avec un coût raisonnable. Dans les sections suivantes, plusieurs méthodes sont analysées.

On appelle méthode de résolution **directe** d'un système linéaire un algorithme qui, si l'ordinateur faisait des calculs exacts, donnerait la solution en un nombre fini d'opérations. Il existe aussi des méthodes **itératives** qui consistent à construire une suite de vecteurs $\mathbf{x}_n$ convergeant vers la solution $\mathbf{x}$.

1. En mathématiques, une permutation est dite paire si elle présente un nombre pair d'inversions, impaire sinon. La signature d'une permutation vaut 1 si celle-ci est paire, –1 si elle est impaire.

4.1 Méthodes directes

Pour résoudre $\mathbf{Ax} = \mathbf{b}$, on cherche à écrire $\mathbf{A} = \mathbf{LU}$ où :
— $\mathbf{L}$ est une matrice triangulaire inférieure avec des 1 sur la diagonale,
— $\mathbf{U}$ est une matrice triangulaire supérieure.
La résolution de $\mathbf{Ax} = \mathbf{b}$ est alors ramenée aux résolutions successives des systèmes échelonnés $\mathbf{Ly} = \mathbf{b}$ et $\mathbf{Ux} = \mathbf{y}$.

4.1.1 Résolution des systèmes triangulaires

Une matrice $\mathbf{A} = (a_{ij})$ est *triangulaire supérieure* si

$$a_{ij} = 0 \quad \forall i\,,\ j\ :\ 1 \leq j < i \leq n \tag{4.5}$$

et *triangulaire inférieure* si

$$a_{ij} = 0 \quad \forall i\,,\ j\ :\ 1 \leq i < j \leq n\cdot \tag{4.6}$$

Suivant ces cas, le système à résoudre est dit *système triangulaire supérieur ou inférieur*. Si la matrice $\mathbf{A}$ est régulière et triangulaire alors, comme $det(\mathbf{A}) = \prod a_{ii}$, on déduit que $a_{ii} \neq 0$, pour tout $i = 1, \cdots, n$.

Si $\mathbf{A}$ est triangulaire inférieure, on a

$$x_1 = \frac{b_1}{a_{11}}\,, \tag{4.7}$$

et pour $i = 2,\ 3,\ \cdots,\ n$,

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j\right). \tag{4.8}$$

Cet algorithme est appelé *méthode de descente*.

Si $\mathbf{A}$ est triangulaire supérieure, on a

$$x_n = b_n/a_{nn} \tag{4.9}$$

et pour $i = n-1,\ n-2, \cdots,\ 2,\ 1$

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right). \tag{4.10}$$

Cet algorithme est appelé *méthode de remontée*.

Le nombre de multiplications et de divisions nécessaires dans cet algorithme est de $\frac{n(n+1)}{2}$ et le nombre d'additions et de soustractions est de $\frac{n(n-1)}{2}$, ce qui implique que l'algorithme nécessite n^2 flops.

4.1.2 Méthode d'élimination de Gauss et décomposition LU

La *méthode d'élimination de Gauss* a pour but de transformer le système $\mathbf{Ax} = \mathbf{b}$ en un système équivalent (c'est-à-dire ayant la même solution) de la forme $\mathbf{Ux} = \tilde{\mathbf{b}}$, où $\mathbf{U}$ est une

matrice triangulaire supérieure et $\tilde{\mathbf{b}}$ est un second membre convenablement modifié. Ce dernier système peut alors être résolu par une méthode de substitution rétrograde.

Au cours de la transformation, on utilise essentiellement la propriété selon laquelle on ne change pas la solution du système quand on ajoute à une équation donnée une combinaison linéaire des autres équations.

Soit $\mathbf{A} = (a_{ij})$ une matrice non singulière de dimension $n \times n$. L'*algorithme de Gauss* est le suivant : on pose $\mathbf{A}^{(1)} = \mathbf{A}$, c'est à dire $a_{ij}^{(1)} = a_{ij}$ pour $i, j = 1, \cdots, n$; et pour $k = 1, \cdots, n$, on calcule

$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \quad i = k+1, \cdots, n\,; \tag{4.11}$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}, \quad i = k+1, \cdots, n\,. \tag{4.12}$$

$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)} \qquad i = k+1, \cdots, n\,. \tag{4.13}$$

A la fin de ce procédé, les élements de la matrice U sont définis par

$$u_{ij} = a_{ij}^{(n)} \tag{4.14}$$

tandis que les éléments de la matrice $\mathbf{L}$ sont les termes l_{ij} engendrés par l'algorithme de Gauss. Le coût de cette méthode de factorissation est de l'ordre de $\frac{2n^3}{3}$ flops. En fait, pour passer de $\mathbf{A}^{(k)}$ à $\mathbf{A}^{(k+1)}$, on modifie tous les éléments de $\mathbf{A}^{(k)}$ sauf les premières k lignes et les premières k colonness, qui ne changent pas ; pour chaque élément de $\mathbf{A}^{(k+1)}$, il faut faire une multiplication et une soustraction ; on a donc $2(n-k)^2$ opérations à faire. Au total, pour fabriquer $\mathbf{A}^{(n)}$, il faut

$$\sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{j=1}^{n-1} j^2 = 2\frac{(n-1)n(2n-1)}{6} \sim \frac{2n^3}{3}\cdot \tag{4.15}$$

Remarque. *Pour que l'algorithme de Gauss puisse terminer, il faut que tous les termes $a_{kk}^{(k)}$, qui correspondent aux termes diagonaux $u_{kk}^{(k)}$ de la matrice* $\mathbf{U}$ *et qu'on appelle* ***pivots****, soient non nuls.*

Le fait que la matrice $\mathbf{A}$ ait des entrées non nulles ne prévient pas l'apparition de pivots nuls, comme on remarque dans l'exemple qui suit :

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{A}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -6 & -12 \end{bmatrix} \tag{4.16}$$

Cependant on a :

Proposition 4.1. *La factorisation* $\mathbf{LU}$ *de la matrice carrée* $\mathbf{A}$ *de taille n par la méthode de Gauss existe si et seulement si les sous-matrices principales $\mathbf{A}_i = (a_{hk})$, h, $k = 1, \cdots, i$ sont non-singulières. Cette propriété est satisfaite en particulier dans les cas qui suivent :*

1. $\mathbf{A}$ *est une matrice symétrique définie positive ;*
2. $\mathbf{A}$ *est une matrice à diagonale dominante stricte par lignes, c'est à dire $\forall i = 1, \cdots, n$,*

$$|a_{ii}| > \sum_{j=1,\, j\neq i}^{n} |a_{ij}|\,. \tag{4.17}$$

3. **A** *est une matrice à diagonale stricte par colonnes, c'est-à-dire* $\forall\ j = 1,\ \cdots,\ n,$

$$|a_{jj}| > \sum_{i=1,\, j\neq i}^{n} |a_{ij}|\,. \tag{4.18}$$

Si **A** *est non singulière, sa factorisation* **LU** *est unique.*
Grace à la factorisation **LU**, *on peut calculer le déterminant d'une matrice carrée avec* $O(n^3)$ *operations,vu que*

$$det(\mathbf{A}) = det(\mathbf{L})det(\mathbf{U}) = det(\mathbf{U}) = \prod_{k=1}^{n} u_{kk}, \tag{4.19}$$

puisque le déterminant d'une matrice triangulaire est le produit des termes diagonaux. La commande *MATLAB* **det(A)** exploite ce procédé. selon le théorème 4.1

4.1.3 Problème des pivots

Si au cours de l'algorithme d'élimination de Gauss on trouve un pivot nul à l'étape k , le procédé s'arrête. On peut alors permuter la ligne k avec la ligne $r > k$. La matrice qui permute les lignes k et r d'une matrice quelconque est la matrice qui a des éléments égaux à 1 partout sur la diagonale sauf $p_{rr} = p_{kk} = 0$ et des éléments égaux à 0 partout ailleurs, sauf $p_{rk} = p_{kr} = 1$:

$$\mathbf{P}^{(k,r)} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ \cdots & 0 & \cdots & 1 & \cdots \\ \vdots & & & & \vdots \\ \cdots & 1 & \cdots & 0 & \cdots \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{matrix} 1 \\ \vdots \\ k \\ \vdots \\ r \\ \vdots \\ n \end{matrix} \tag{4.20}$$

On aboutit alors à la factorisation d'une nouvelle matrice, c'est-à-dire

$$\mathbf{LU} = \mathbf{PA}\,, \tag{4.21}$$

où **P** est le produit de toutes les matrices de permutation (des lignes) qu'on a utilisées.

Supposons donc qu'on a trouvé une factorisation du type $\mathbf{PA} = \mathbf{LU}$. Alors, le vecteur $\mathbf{x}$, solution du système $\mathbf{Ax} = \mathbf{b}$, vérifie également $\mathbf{PAx} = \tilde{\mathbf{b}}$, où $\tilde{\mathbf{b}} = \mathbf{Pb}$. Ce deuxième système s'écrit $\mathbf{LUx} = \tilde{\mathbf{b}}$. Donc, comme auparavant, on résout par $\mathbf{Ly} = \tilde{\mathbf{b}}$ et $\mathbf{Ux} = \mathbf{y}$.

Si un pivot est très petit, sans pour autant être nul, le procédé d'élimination ne s'arrête pas, mais le résultat peut être entaché d'erreurs considérables, comme on va le voir dans l'exemple ci-dessous.

Soit ϵ un petit nombre réel et considérons le système de deux équations à deux inconnues

$$\begin{cases} \epsilon x + y = 1 \\ x + y = 2 \end{cases} \tag{4.22}$$

Par élimination sans recherche de pivot, nous obtenons le système équivalent suivant

$$\begin{cases} \epsilon x + y = 1 \\ \left(1 - \frac{1}{\epsilon}\right) y = 2 - \frac{1}{\epsilon} \end{cases} \tag{4.23}$$

On a donc immédiatement la valeur de y :

$$y = \frac{1 - 2\epsilon}{1 - \epsilon} \simeq 1 \,. \tag{4.24}$$

Par substitustion, on obtient x :

$$x = \frac{1 - y}{\epsilon} \simeq 0 \,. \tag{4.25}$$

Essayons maintenant l'élimination après avoir échangé les lignes dans le système :

$$\begin{cases} x + y = 2 \\ \epsilon x + y = 1 \end{cases} \tag{4.26}$$

Le même procédé d'élimination que précédemment conduit au système équivalent suivant

$$\begin{cases} x + y = 2 \\ (1 - \epsilon) y = 1 - 2\epsilon \end{cases} \tag{4.27}$$

qui devient, en flottants,

$$\begin{cases} x + y = 2 \\ y = 1 \end{cases} \tag{4.28}$$

La solution, tout à fait acceptable cette fois-ci, est

$$x = 1 \text{ et } y = 1.$$

L'erreur qu'on faisait avec la première élimination venait de ce qu'on divisait un nombre par le pivot ϵ, ce qui amplifie considérablement les erreurs.

L'algorithme de décomposition de Gauss avec pivotation par lignes permute deux lignes de la matrice $A^{(k)}$ à chaque pas de la décomposition afin que l'élément diagonale $a_{kk}^{(k)}$ de la matrice permutée soit maximal (en valeur absolue). Précisément, au pas k de la décomposition, on trouve l'index r , avec $r \geq k$, tel que

$$\left|a_{rk}^{(k)}\right| = \max_{s \geq k} \left|a_{sk}^{(k)}\right|$$

et on échange les lignes r et k entre elles. Le logiciel *MATLAB* implémente ce dernier algorithme dans la commande **lu**.

4.1.4 Matrices tridiagonales

On considère le cas particulier où la matrice **A** est non singulière et tridiagonale, donnée par

$$\begin{bmatrix} a_1 & c_1 & & 0 \\ b_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & b_n & a_n \end{bmatrix} \tag{4.29}$$

Dans ce cas, les matrices **L** et **U** de factorisation **LU** de **A** sont des matrices bidiagonales de la forme :

$$\mathbf{L} = \begin{bmatrix} 1 & & & 0 \\ \beta_2 & 1 & & \\ & \ddots & \ddots & \\ 0 & & \beta_n & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} \alpha_1 & c_1 & & 0 \\ & \alpha_2 & \ddots & \\ & & \ddots & c_{n-1} \\ 0 & & & \alpha_n \end{bmatrix} \tag{4.30}$$

Les coefficients α_i et β_i peuvent être calculés par les relations :

$$\alpha_1 = a_1,\ \beta_2\alpha_1 = b_2 \Rightarrow \beta_2 = \frac{b_2}{\alpha_1},\ \beta_2 c_1 + \alpha_2 = a_2 \rightarrow \alpha_2 = a_2 - \beta_2 c_1, \cdots \tag{4.31}$$

Donc, on a

$$\alpha_1 = a_1,\ \beta_i = \frac{b_i}{\alpha_{i-1}},\ \alpha_i = a_i - \beta_i c_{i-1}, \quad i = 2, \cdots, n. \tag{4.32}$$

Cet algorithme est connu sous le nom de *algorithme de Thomas*, et le nombre d'opérations effectuées est de l'ordre de n. Avec l'algorithme de Thomas, résoudre un système tridiagonal $\mathbf{Ax} = \mathbf{f}$ revient à résoudre 2 systèmes bidiagonaux $\mathbf{Ly} = \mathbf{f}$ et $\mathbf{Ux} = \mathbf{y}$ avec

$$\mathbf{Ly} = \mathbf{f} \ \leftrightarrow\ y_1 = f_1,\ y_i = f_i - \beta_i y_{i-1}, \quad i = 2, \cdots, n. \tag{4.33}$$

$$\mathbf{Ux} = \mathbf{y} \ \leftrightarrow\ x_n = \frac{y_n}{\alpha_n},\ x_i = \frac{y_i - c_i x_{i+1}}{\alpha_i}, \quad i = n-1, \cdots, 1. \tag{4.34}$$

Le coût de calcul ici est de $3(n-1)$ opérations pour la factorisation et de $5n-4$ opérations pour la substitution, ce qui fait un total de $8n-7$ opérations.

4.1.5 Applications de la factorisation LU

4.1.5.1 Calcul du déterminant d'une matrice

La méthode de *factorisation LU* fournit un procédé rapide de calcul du déterminant de la matrice **A**, qui n'est autre, au signe près, que le produit des pivots, puisque

$$\det(PA) = \det(LU) = \det(L)\det(U) = \det(U) = \left(\prod_{i=1}^{n} u_{ii}\right)$$

Comme $\det(P) = (-1)^\sigma$, où σ est le nombre de permutations dans l'élimination de Gauss, on obtient

$$\det(A) = \frac{\det(PA)}{\det(P)} = (-1)^\sigma \det(U) = (-1)^\sigma \left(\prod_{i=1}^{n} u_{ii}\right)$$

4.1.5.2 Calcul de l'inverse d'une matrice

Si **A** est une matrice $n \times n$ non singulière, notons $\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(n)}$ les colonnes de sa matrice inverse $\mathbf{A}^{-1}$, c'est-à-dire $\mathbf{A}^{-1} = \left(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(n)}\right)$. La relation $\mathbf{AA}^{-1} = \mathbf{I}$ se traduit par les systèmes suivants :

$$\mathbf{Av}^{(k)} = \mathbf{e}^{(k)}, \quad 1 \le k \le n, \tag{4.35}$$

où $\mathbf{e}^{(k)}$ est le vecteur colonne ayant tous les éléments 0 sauf celui sur la ligne k qui est 1 :

$$\mathbf{e}^{(k)} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} 1 \\ \vdots \\ k \\ \vdots \\ n \end{matrix} \tag{4.36}$$

Connaissant les matrices $\mathbf{L}$ et $\mathbf{U}$ qui factorisent la matrice $\mathbf{A}$, résoudre les n systèmes (4.35) gouvernés par la même matrice $\mathbf{A}$, coûte n^3 opérations.

Le calcul de l'inverse d'une matrice est une opération non seulement coûteuse, mais qui peut aussi s'avérer parfois moins stable que la méthode de Gauss.

Les **formules de Faddev ou de Leverrier** offrent une alternative pour le calcul de l'inverse de $\mathbf{A}$: posons $\mathbf{B}_0 = \mathbf{I}$, et calculons par récurrence

$$\alpha_k = \frac{1}{k}\mathrm{tr}(\mathbf{A}\mathbf{B}_{k-1})\,, \quad \mathbf{B}_k = -\mathbf{A}\mathbf{B}_{k-1} + \alpha_k \mathbf{I}\,, \quad k = 1\,,\ 2\,,\ \cdots\,,\ n. \tag{4.37}$$

Puisque $\mathbf{B}_n = \mathbf{0}$, si $\alpha_n \neq 0$ on obtient

$$\mathbf{A}^{-1} = \frac{1}{\alpha_n}\mathbf{B}_{n-1} \tag{4.38}$$

et le coût de la méthode pour une matrice pleine est de $(n-1)n^3$ flops. Pour plus de détails, voir **FF63** [2] et **Bar89** [3].

4.1.6 Considération sur la précision des méthodes directes pour les systèmes linéaires

Il y a des cas où les méthodes qu'on vient de voir ne marchent pas très bien.

Définition. On définit le conditionnement d'une matrice M symétrique définie positive comme le rapport entre la valeur maximale et la valeur minimale de ses valeurs propres, c'est-à-dire

$$K_2(\mathbf{M}) = \frac{\lambda_{\max}(\mathbf{M})}{\lambda_{\min}(\mathbf{M})}\cdot \tag{4.39}$$

On peut monter que plus le conditionnement de la matrice est grand, plus la solution du système linéaire obtenue par une méthode directe peut être mauvaise. Par exemple, considérons un système linéaire

$$\mathbf{A}\mathbf{x} = \mathbf{b}\cdot \tag{4.40}$$

Si on résout ce système avec un ordinateur, à cause des eurreurs d'arrondis on ne trouve pas la solution exacte mais une solution approchée $\hat{\mathbf{x}}$. On s'attend alors à ce que la solution $\hat{\mathbf{x}}$ soit très proche de $\mathbf{x}$. Toutefois, on peut montrer la relation suivante :

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq K_2(\mathbf{A})\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \tag{4.41}$$

2. Faddeev D. K. and Faddeeva V. N. (1963) *Computational Methods of Linear Algebra*. Freeman, San Francisco and London.

3. Barnett S. (1989) **Leverrier's Algorithm : A New Proof and Extensions**. *Numer. Math.* **7** : 338-352.

où $\mathbf{r}$ est le résidu $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$; on a noté $\|.\|$ la norme euclidienne d'un vecteur. On remarque que, si le conditionnement de $\mathbf{A}$ est grand, la distance $\|\mathbf{x} - \hat{\mathbf{x}}\|$ entre la solution exacte et celle calculée numériquement peut être très grande même si le résidu est très petit. Avec **MATLAB** on peut calculer le conditionnement d'une matrice avec la commande *cond*.

4.1.7 Méthode de Cholesky

La méthode d'André-Louis Cholesky (1875-1918) s'applique aux matrices hermitiennes ou symétriques. Elle est fondée sur le théorème suivant qui affirme que *pour une matrice hermitienne (resp. symétrique) définie positive* $\boldsymbol{A}$*, il existe (au moins) une matrice triangulaire inférieure inversible* $\boldsymbol{L}$ *telle que* $\mathbf{A} = \mathbf{L}\mathbf{L}^\star$ *(resp.* $\mathbf{A} = \mathbf{L}\mathbf{L}^T$*). Si les éléments diagonaux de* $\boldsymbol{L}$ *sont strictement positifs, la matrice L est unique.* Ce théorème est valable pour une matrice triangulaire supérieure $\mathbf{U}$.

Soit A une matrice symétrique définie positive à coefficients réels. Mettons A sous la forme $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ (**décomposition de Cholesky**). En appliquant l'algorithme de la factorisation LU à la i-ième étape ; on calcule la matrice $\mathbf{L}$ par

$$L_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right) \Big/ L_{jj} \qquad j = 1\,,\ \cdots\,,\ i-1\,; \qquad L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \cdot \tag{4.42}$$

On résout ensuite le système $\mathbf{L}\mathbf{y} = \mathbf{b}$ puis $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ par un double balayage :

$$y_i = \left(b_i - \sum_{k=1}^{i-1} L_{ik} y_k \right) \Big/ L_{ii} \qquad i = 1\,,\ \cdots\,,\ n\,; \qquad x_i = \left(y_i - \sum_{k=i+1}^{n} L_{ik} x_k \right) \Big/ L_{ii}\,\cdot \tag{4.43}$$

Décrivons l'algorithme pour obtenir la décomposition de Cholesky. Partant de $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, on a

$$a_{ij} = \sum_{k=1}^{j} l_{ik} l_{jk}, \quad j \le i\cdot \tag{4.44}$$

1. Obtention de la première colonne. On a $a_{11} = l_{11}^2 > 0$, d'où $l_{11} = \sqrt{a_{11}} > 0$.
   Pour $i = 2, \cdots, n$, on a $a_{i1} = l_{i1} l_{11}$, donc $l_{i1} = \frac{a_{i1}}{l_{11}}$.
2. Obtention des colonnes suivantes. Supposons avoir calculé les colonnes $1, \cdots, j-1$. On a

$$a_{jj} = \sum_{k=1}^{j} l_{jk}^2 = l_{j1}^2 + \cdots + l_{j,j-1}^2 + l_{jj}^2\,, \tag{4.45}$$

   d'où

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}\,, \tag{4.46}$$

   ce qui est bien défini grâce au théorème.

   Pour $i = j+1, \cdots, n$,

$$a_{ij} = l_{i1} l_{j1} + \cdots + l_{ij} l_{jj}\,\cdot$$

   Par conséquent,

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) \cdot \tag{4.47}$$

   On remarque que $det(\mathbf{A}) = \prod_{j=1}^{n} l_{jj}^2 \neq 0$, donc $l_{jj} \neq 0$.

Algorithme de cholesky

$1:\ L(1:n,1:n)=0$
$2:\ L(1,1)=\sqrt{a(1,1)}$
$3:\ pour\ i=2\,à\,n\ faire$
$4:\quad L(i,1)=\frac{a(i,1)}{L(1,1)}$
$5:\ fin\ pour$
$6:\ pour\ j=2\,à\,n\ faire$
$7:\quad L(j,j)=\sqrt{A(j,j)-\sum_{k=1}^{j-1}L(j,k)^2}$
$8:\quad pour\ i=j+1\,à\,n\ faire$
$9:\qquad L(i,j)=\frac{1}{L(j,j)}\left(A(i,j)-\sum_{k=1}^{j-1}L(i,k)L(j,k)\right)$
$10:\quad fin\ pour$
$11:\ fin\ pour$

4.2 Exercices

Exercice 1. On veut résoudre le système linéaire $\mathbf{Ax}=\mathbf{b}$ où

$$\mathbf{A}=\begin{bmatrix}1&1&1\\2&2&5\\4&6&8\end{bmatrix}\qquad\text{et}\qquad\mathbf{b}=\begin{bmatrix}1\\2\\5\end{bmatrix}$$

par la méthode d'élimination de Gauss.

a) Vérifier que l'algorithme de Gauss sans pivoting ne peut pas être exécuté jusqu'au bout.
b) Trouver une matrice de permutation $\mathbf{P}$ telle que la matrice $\mathbf{PA}$ soit factorisable.
c) Calculer la factorisation $\mathbf{LU}$ de la matrice $\mathbf{PA}$.
d) Résoudre le système linéaire $\mathbf{Ax}=\mathbf{b}$ en remplaçant $\mathbf{PA}$ par $\mathbf{LU}$ et en utilisant les algorithmes de substitution progressive et rétrograde.

Exercice 2.

1) Pour une matrice quelconque $\mathbf{A}$, montrer que $\rho(\mathbf{A})\leq||\mathbf{A}||$ pour toute norme matricielle. Donner l'expression de $||\mathbf{A}||_\infty$.
2) On suppose $\mathbf{A}$ symétrique définie positive. Montrer que $\rho(\mathbf{A})=||\mathbf{A}||_2$.
3) Soit

$$\mathbf{A}=\begin{bmatrix}1&a&a\\a&1&a\\a&a&1\end{bmatrix}$$

Pour quelle valeur de a $\mathbf{A}$ est-elle définie positive ?

Chapitre 5

Résolution numérique d'équations non linéaires

Soit f : Ω fermé $\subset \mathbb{R}^p \to \mathbb{R}^p$. On cherche $\xi \in \Omega$ tel que $f(\xi) = 0$. Nous supposerons qu'un moyen (une étude graphique, une raison physique, un théorème, l'intuition ou ... la chance !) nous ont permis de voir que ξ était l'unique solution dans Ω . Nous supposerons aussi que f est au moins continue sur Ω.

Nous nous intéresserons seulement à l'approximation des zéros (ou racines dans le cas d'un polynôme) d'une fonction réelle d'une variable réelle, c'est-à-dire, étant donné un intervalle $I \subseteq \mathbb{R}$ et une application f de I dans $\mathbb{R}$, la résolution approchée du problème : *trouver* $\xi \in \mathbb{R}$ *(ou plus généralement* $\mathbb{C}$*) tel que* $f(\xi) = 0$.

Ce problème intervient notamment dans l'étude générale de fonctions d'une variable réelle, qu'elle soit motivée ou non par des applications [1] pour lesquelles des solutions exactes de ce type d'équation ne sont pas connues [2].

1. Essayons néanmoins de donner deux exemples, l'un issu de la physique, l'autre de l'économie. Supposons tout d'abord que l'on cherche à déterminer le volume V occupé par n molécules d'un **gaz de van der Waals** de température T et de pression p. L'*équation d'état* (c'est-à-dire l'équation liant les variables d'état que sont n, p, et V) d'un tel gaz s'écrit

$$\left(p + a\left(\frac{n}{V}\right)^2\right)(V - nb) = nk_\beta T,$$

où les coefficients a (pression de cohésion) et b (*covolume*) dépendent de la nature du gaz et k_β désigne la *constante de Boltzmann*. On est donc amené à résoudre une équation non linéaire d'inconnue V et de fonction

$$f(V) = \left(p + a\left(\frac{n}{V}\right)^2\right)(V - nb) - nk_\beta T = 0.$$

Admettons maintenant que l'on souhaite calculer le **taux de rendement annuel moyen** R d'un fonds de placement et que l'on se retrouve après n années avec un capital d'un montant de M euros. La relation liant M, n, R et V est

$$M = V\sum_{k=1}^{n}(1+R)^k = V\frac{1+R}{R}\big((1+R)^n - 1\big),$$

et on doit alors trouver R tel que

$$f(R) = M - V\sum_{k=1}^{n}(1+R)^k = M - V\frac{1+R}{R}\big((1+R)^n - 1\big) = 0.$$

2. Même dans le cas d'une **équation algébrique**, on rappelle qu'il n'existe pas de méthode de résolution générale à partir du degré cinq.

Toutes les méthodes que nous allons présenter sont *itératives* et consistent donc en la construction d'une suite réelle $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ qui, on l'espère, sera telle que

$$\lim_{k\to+\infty} x^{(k)} = \xi. \tag{5.1}$$

En effet, à la différence du cas des systèmes linéaires, la convergence de ces méthodes itératives dépend en général du choix de la donnée initiale $x^{(0)}$. On verra ainsi qu'on ne sait souvent qu'établir des résultats de *convergence locale*, valables lorsque $x^{(0)}$ appartient à un certain voisinage du zéro ξ.

Après avoir caractérisé la convergence des suites engendrées par des méthodes itératives, en introduisant notamment la notion d'*ordre de convergence*, nous présentons quelques méthodes parmi les plus connues et les plus utilisées : tout d'abord des méthodes dites **d'encadrement** comme celles de *dichotomie* et *de la fausse position*, puis les méthodes *de la corde*, *de Newton*[3]*-Raphson*[4], qui sont toutes deux des **méthodes de point fixe**, et enfin la **méthode de la sécante**. Dans chaque cas, un ou plusieurs résultats de convergence ad hoc sont énoncés. Des méthodes adaptées au cas particulier des **équations algébriques** (c'est-à-dire polynomiales) sont brièvement abordées dans la section 5.5. Signalons que nous nous sommes surtout référés à [Guillaume2009] pour constituer les notes concernant le présent chapitre.

5.1 Comptage et localisation des zéros : la méthode de Sturm

Dans le cas d'une fonction réelle d'une variable réelle, une bonne pratique est de commencer par faire un graphe de la fonction pour avoir une idée sur le nombre de zéros et leur *localisation*. On peut évidemment aussi étudier cette fonction par les moyens analytiques classiques pour *localiser* son(ses) zéro(s) éventuel(s). Dans le cas d'une fonction polynomiale d'une variable réelle, le comptage des racines peut aussi être réalisé au moyen de la **méthode de Sturm**. Celle-ci est fondée sur l'utilisation des **suites de Sturm** [Alain2009]. Notons en passant qu'une suite de polynômes P_0, P_1, $\cdots$, P_s est une **suite de Sturm** sur l'intervalle réel [a,b] lorsque :

1. les racines de P_0 sont simples (ou $P_0(a)\,P_0(b) \neq 0$) ;
2. si $P_0(c) = 0$ et $c \in]a,b[$, alors $\mathbf{sign}(P_1(c)) = -\mathbf{sign}\,(P_0'(c))$;
3. si $P_k(x) = 0$, alors $P_{k-1}(x)P_{k+1}(x) < 0$, et ce pour $1 \leq k \leq s-1$;
4. $P_s(x) \neq 0$ pour tout $x \in [a,b]$.

5.2 Généralités

5.2.1 Ordre de convergence d'une méthode itérative

Afin de pouvoir évaluer à quelle « vitesse » la suite construite par une méthode itérative converge vers sa limite (ce sera souvent un des critères discriminants lors du choix d'une méthode), il nous faut introduire quelques définitions.

Définition 5.1 (ordre d'une suite convergente)**.** *Soit une suite* $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ *de réels convergeant vers une limite* ξ*. On dit que cette suite convergente est d'ordre* $r \geq 1$ *s'il existe deux constantes* C_1 *et* C_2

3. Sir Isaac Newton (4 janvier 1643 - 31 mars 1727) était un philosophe, mathématicien, physicien et astronome anglais. Figure amblématique des Sciences, il est surtout reconnu pour sa théorie de la gravitation universelle et la création du calcul infinitésimal.

4. Joseph Raphson (v. 1648 - v. 1715) était un mathématicien anglais. Son travail le plus notable est son ouvrage *Analysis aequationum universalis*, publié en 1690 et contenant une méthode pour l'approximation d'un zéro d'une fonction d'une variable réelle à valeurs réelles .

vérifiant
$0 < C_1 \leq C_2 < +\infty$ *telles que*

$$C_1 \leq \frac{\left|x^{(k+1)} - \xi\right|}{\left|x^{(k)} - \xi\right|^r} \leq C_2 \quad \forall k \geq k_0, \tag{5.2}$$

où k_0 *appartient à* $\mathbb{N}$.

Par extension, une méthode itérative produisant une suite convergente vérifiant les relations (5.2) sera également dite *d'ordre r*. On notera que, dans plusieurs ouvrages, on trouve l'ordre d'une suite défini uniquement par le fait qu'il existe une constante $C \geq 0$ telle que pour tout $k \geq k_0 \geq 0$, $\left|x^{(k+1)} - \xi\right| \leq C \left|x^{(k)} - \xi\right|^r$. Il faut cependant observer que cette définition n'assure pas l'unicité de r, l'ordre de convergence pouvant éventuellement être plus grand que r. On préférera donc dire dans ce cas que la suite est d'ordre r *au moins*. On remarquera aussi que, si r est égal à 1, on a nécessairement $C_2 < 1$ dans (5.2), faute de quoi la suite ne pourrait pas converger.

La définition 5.1 est très générale et n'exige pas que la suite $\left(\frac{\left|x^{(k+1)}-\xi\right|}{\left|x^{(k)}-\xi\right|}\right)_{k\in\mathbb{N}}$ admette une limite quand k tend vers l'infini. Lorsque c'est le cas, on a coutume de se servir de la définition suivante.

Définition 5.2. *Soit une suite* $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ *de réels convergeant vers une limite* ξ. *On dit que cette suite est convergente d'ordre* r, *avec* $r > 1$, *vers* ξ *s'il existe un réel* $\mu > 0$, *appelé* ***constante asymptotique d'erreur****, tel que*

$$\lim_{k\to+\infty} \frac{\left|x^{(k+1)} - \xi\right|}{\left|x^{(k)} - \xi\right|^r} = \mu. \tag{5.3}$$

Dans le cas particulier où $r = 1$, *on dit que la suite* ***converge linéairement*** *si*

$$\lim_{k\to+\infty} \frac{\left|x^{(k+1)} - \xi\right|}{\left|x^{(k)} - \xi\right|} = \mu, \text{ avec } \mu \in]0, 1[, \tag{5.4}$$

et ***super linéairement*** *(resp.* ***sous-linéairement****) si l'égalité ci-dessus est verifié avec* $\mu = 0$ *(resp.* $\mu = 1$*).*

Ajoutons que la convergence d'ordre 2 est dite *quadratique*, celle d'ordre 3 *cubique*.

Si cette dernière caractérisation est particulièrement adaptée à l'étude pratique de la plupart des méthodes itératives que nous allons présenter dans ce chapitre, elle a comme inconvénient de ne pouvoir permettre de fournir l'ordre d'une suite dont la « vitesse de convergence » est variable, ce qui se traduit par le fait que la limite (5.2) n'existe pas. On a alors recours à une définition « étendue ».

Définition 5.3. *On dit qu'une suite* $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ *de réels* ***converge avec un ordre*** r ***au moins*** *vers une limite* ξ *s'il existe une suite* $\left(\varepsilon^{(k)}\right)_{k\in\mathbb{N}}$ *vérifiant*

$$\left|x^{(k)} - \xi\right| \leq \varepsilon^{(k)}, \ \forall k \in \mathbb{N}, \tag{5.5}$$

et un réel $\nu > 0$ *tel que*

$$\lim_{k\to+\infty} \frac{\varepsilon^{(k+1)}}{{\varepsilon^{(k)}}^r} = \nu. \tag{5.6}$$

On remarquera l'ajout du qualificatif *au moins* dans la définition 5.3 qui provient du fait que l'on a dû procéder à une majoration par une suite convergeant vers zéro avec un ordre r au sens de la définition 5.2. Bien évidemment, on retrouve la définition 5.2 si l'on a égalité dans (5.5), mais ceci est souvent impossible à obtenir en pratique.

Finissons en indiquant que les notions d'ordre et de constante asymptotique d'erreur ne sont pas purement théoriques et sont en relation avec le nombre de chiffres exacts obtenus dans l'approximation de ξ. Posons en effet $\delta^{(k)} = -log_{10}\left(\left|x^{(k)} - \xi\right|\right)$; $\delta^{(k)}$ est alors le nombre de chiffres significatifs décimaux exacts de $x^{(k)}$. Pour k suffisamment grand, on a

$$\delta^{(k+1)} \approx r\delta^{(k)} - \log_{10}(\mu)\cdot \tag{5.7}$$

On voit donc que si r est égal à un, on ajoute environ $-\log_{10}(\mu)$ chiffres significatifs à chaque itération. Par exemple, si $\mu = 0.999$ alors $-\log_{10}(\mu) \approx 4,34 \times 10^{-4}$ et il faudra près de 2500 itérations pour gagner une seule décimale. Par contre, si r est strictement plus grand que un, on multiplie environ par r le nombre de chiffres significatifs à chaque itération. Ceci montre clairement l'intérêt des méthodes d'ordre plus grand que un.

5.2.2 Critères d'arrêt

En cas de convergence, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ construite par la méthode itérative tend vers le zéro ξ quand k tend vers l'infini. Pour l'utilisation pratique d'une telle méthode, il faut introduire un *critère d'arrêt* pour interrompre le processus itératif lorsque l'approximation courante de ξ est jugé « satisfaisante ». Pour cela, on a principalement le choix entre deux types de critères (imposer un nombre maximum d'itérations constituant une troisième possibilité) : l'un basé sur l'incrément et l'autre sur le résidu.

Soit $\varepsilon > 0$ la tolérance fixée pour le calcul approché de ξ. Dans le cas d'un *contrôle de l'incrément*, les itérations s'achèvent dès que

$$\left|x^{(k+1)} - x^{(k)}\right| < \varepsilon. \tag{5.8}$$

Si l'on choisit de *contrôler le résidu*, on met fin aux itérations dès que

$$\left|f\left(x^{(k)}\right)\right| < \varepsilon. \tag{5.9}$$

Selon le cas, chacun de ces critères peut s'avérer soit trop restrictif, soit trop optimiste.

5.3 Méthodes d'encadrement

Cette première classe de méthodes repose sur la propriété fondamentale suivante, relative à l'existence de zéros d'une application d'une variable rélle à valeurs réelles.

Théorème 5.1 (existence d'un zéro d'une fonction continue)**.** *Soit un intervalle non vide [a,b] de* $\mathbb{R}$ *et f une application continue de [a,b] dans* $\mathbb{R}$ *vérifiant* $f(a)\,f(b) < 0$*. Alors il existe* $\xi \in]a,b[$ *tel que* $f(\xi) = 0$*.*

Démonstration. Si $f(a) < 0$, on a $0 \in]f(a), f(b)[$, sinon $f(a) > 0$ et alors $0 \in]f(b), f(a)[$. Dans ces deux cas, le résultat est une conséquence du théorème des valeurs intermédiaires [5]. Ce théorème donne dans certains cas l'existence de solutions d'équations et est à la base de techniques de résolutions approchées comme la recherche dichotomique ou la bissection.

5.3.1 Méthode de dichotomie

La *méthode de dichotomie* (ou *méthode de la bissection*) suppose que la fonction f est continue sur un intervalle [a,b], n'admet qu'un seul zéro $\xi \in]a, b[$ et vérifie $f(a)f(b) < 0$.

Son principe est le suivant. On pose $a^{(0)} = a$, $b^{(0)} = b$; on note $x^{(0)} = \frac{1}{2}\left(a^{(0)} + b^{(0)}\right)$ le milieu de l'intervalle de départ et on évalue la fonction f en ce point. Si $f\left(x^{(0)}\right) = 0$, le point $x^{(0)}$ est le zéro de f et le problème est résolu. Sinon, si $f\left(a^{(0)}\right) f\left(x^{(0)}\right) < 0$, alors le zéro ξ est contenu dans l'intervalle $]a^{(0)}, x^{(0)}[$ alors qu'il appartient à $]x^{(0)}, b^{(0)}[$ si $f\left(x^{(0)}\right) f\left(b^{(0)}\right) < 0$. On réitère ensuite ce processus sur l'intervalle $[a^{(1)}, b^{(1)}]$, avec $a^{(1)} = a^{(0)}$ et $b^{(1)} = x^{(0)}$ dans le premier cas, ou $a^{(1)} = x^{(0)}$ et $b^{(1)} = b^{(0)}$ dans le second, et ainsi de suite...

De cette façon, on construit de manière récurrente trois suites $\left(a^{(k)}\right)_{k\in\mathbb{N}}$, $\left(b^{(k)}\right)_{k\in\mathbb{N}}$ et $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ telles que $a^{(0)} = a$, $b^{(0)} = b$ et vérifiant, pour un entier naturel k :

- $x^{(k)} = \frac{a^{(k)}+b^{(k)}}{2}$,
- $a^{(k+1)} = a^{(k)}$ et $b^{(k+1)} = x^{(k)}$ si $f\left(a^{(k)}\right) f\left(x^{(k)}\right) < 0$,
- $a^{(k+1)} = x^{(k)}$ et $b^{(k+1)} = b^{(k)}$ si $f\left(x^{(k)}\right) f\left(b^{(k)}\right) < 0$.

La figure 5.1 illustre la mise en oeuvre de la méthode.

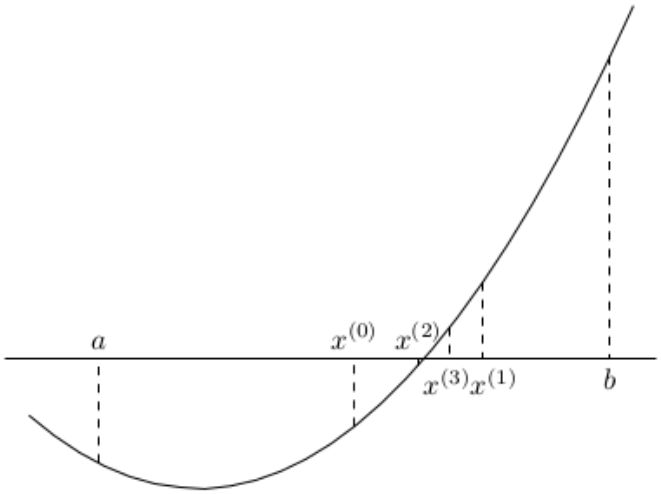


FIGURE 5.1 – Construction des premiers itérés de la méthode de dichotomie

.

5. Le **théorème des valeurs intermédiaires** (**TVI**) ou **de Bolzano** est un théorème important en analyse et concerne des fonctions continues sur un intervalle. Il indique que si une fonction continue sur un intervalle prend deux valeurs m et n, alors elle prend toutes les valeurs intermédiaires entre m et n. Il est souvent énoncé comme suit : *Pour toute fonction $f : [a, b] \to \mathbb{R}$ et tout réel u compris entre $f(a)$ et $f(b)$, il existe au moins un réel c compris entre a et b tel que $f(c) = u$.*

Cas particulier (**Théorème de Bolzano**)

Si $f(a)\, f(b) \leq 0$, il existe au moins un réel $c \in [a, b]$ tel que $f(c) = 0$ (car 0 est compris entre $f(a)$ et $f(b)$)

Concernant la convergence de cette méthode, on a le résultat suivant, dont la preuve est laissée en exercice.

Proposition 5.1. *Soit f une fonction continue sur un intervale [a,b], vérifiant $f(a)f(b) < 0$, et soit $\xi \in]a,b[$ l'unique solution de l'équation $f(x) = 0$. Alors, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ construite par la méthode de dichotomie converge vers ξ et on a l'estimation*

$$\left|x^{(k)} - \xi\right| \leq \frac{b-a}{2^{k+1}}, \ \forall k \in \mathbb{N}. \tag{5.10}$$

Il ressort de cette proposition que la méthode de dichotomie converge de manière certaine : c'est une méthode *globalement convergente*. L'estimation d'erreur (5.10) fournit par ailleurs directement un critère d'arrêt pour la méthode, puisque, à précision ε donnée, cette dernière permet d'approcher ξ en un nombre prévisible d'itérations. On voit en effet que, pour avoir $\left|x^{(k)} - \xi\right| \leq \varepsilon$, il faut que

$$\frac{b-a}{2^{k+1}} \leq \varepsilon \Leftrightarrow k \geq \frac{\ln\left(\frac{b-a}{\varepsilon}\right)}{\ln(2)} - 1\cdot \tag{5.11}$$

Ainsi, pour améliorer la précision de l'approximation du zéro d'un ordre de grandeur, c'est-à-dire trouver $k > j$ tel que $\left|x^{(k)} - \xi\right| = \frac{1}{10}\left|x^{(j)} - \xi\right|$, il faut effectuer $k - j = \frac{\ln(10)}{\ln(2)} \simeq 3.32$ itérations. La convergence de cet algorithme est donc *lente*. Enfin, la méthode de dichotomie ne garantit pas une réduction monotone de l'erreur absolue d'une itération à l'autre, comme on le constate sur la figure 5.2 où la méthode de dichotomie est appliquée sur un polynôme de Legendre [6] de degré 5. Ce n'est donc pas une méthode d'ordre un au sens de la définition 5.1.

On gardera donc à l'esprit que la méthode de dichotomie est une méthode robuste permettant d'obtenir une approximation raisonable du zéro ξ pouvant servir à l'initialisation d'une méthode dont la convergence est plus rapide mais seulement *locale*, comme la méthode de Newton-Raphson (voir la section 5.4.4).

6. Adrien-Marie Legendre (18 septembre 1752- 9 janvier 1833) était français. On lui doit d'importantes contributions en théorie des nombres, en statistique, en algèbre et en analyse, ainsi qu'en mécanique. Il est aussi célèbre pour être l'auteur des *Élément de géométrie*, un traité publié pour la première fois en 1794 reprenant et modernisant les *Éléménts* d'Euclide

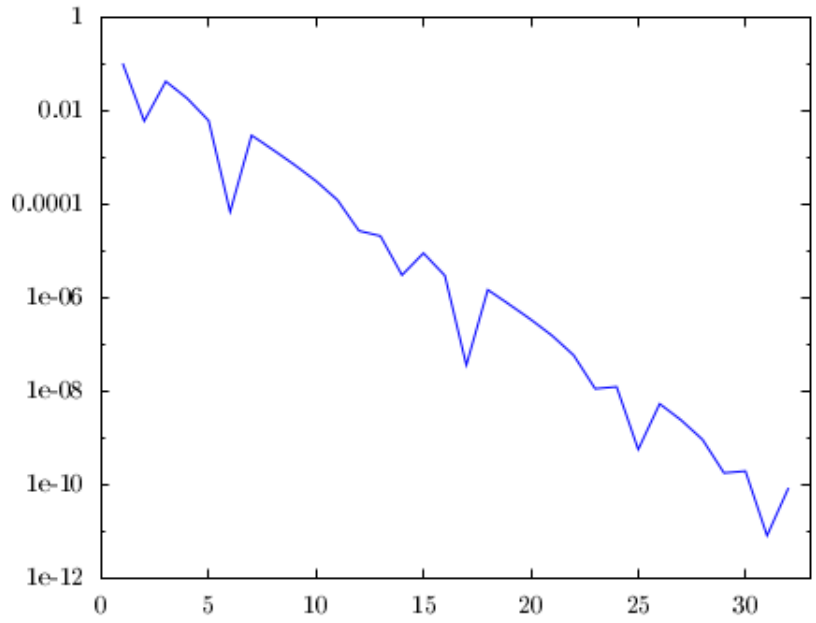


FIGURE 5.2 – Historique de la convergence, c'est-à-dire le tracé de l'erreur $\left|x^{(k)} - \xi\right|$ en fonction de k, de la méthode de dichotomie pour l'approximation de la racine $\xi = 0,9061798459...$ du polynôme de Legendre de degré 5, $P_5(x) = \frac{x}{8}\left(63x^4 - 70x^2 + 15\right)$, dont les racines se situent dans l'intevalle $]-1,1[$. On a choisi les bornes $a = 0,6$ et $b = 1$ pour l'intervalle d'encadrement initial et une précision de 10^{-10} pour le test d'arrêt, qui est atteinte après 31 itérations (à comparer à la valeur 30,89735... de l'estimation (5.11). On observe que l'erreur a un comportement oscillant, mais diminue néamoins en moyenne.

Exemple. On utilise la méthode de dichotomie pour approcher la racine du polynôme $f(x) = x^3 + 2x^2 - 3x - 1$ contenue dans l'intervalle $[1,2]$ (cette fonction est en effet continue et on a $f(1) = -1$ et $f(2) = 9$), avec une précision égale à 10^{-4}. Le tableau suivant donne les valeurs respectives des bornes $a^{(k)}$ et $b^{(k)}$ de l'intervalle d'encadrement, de l'approximation $x^{(k)}$ de la racine et de $f\left(x^{(k)}\right)$ en fonction du numéro k de l'itération.

5.3.2 Méthode de la fausse position

La *méthode de la fausse position*, dite encore méthode *regula falsi*, est une méthode d'encadrement combinant les possibilités de la méthode de dichotomie avec celles de la méthode de la sécante, que nous introduisons dans la section 5.4.5. L'idée est d'utiliser l'information fournie par les valeurs de la fonction f aux extrémités de l'intervalle d'encadrement pour remédier à la lente vitesse de convergence de la méthode de dichotomie (cette dernière ne tenant compte que du signe de la fonction). Sous des hypothèses raisonables de régularité sur f, on peut en effet montrer que la convergence de cette méthode est *linéaire*.

Comme précédemment, cette méthode suppose connus deux points a et b vérifiant $f(a)f(b) < 0$ et servant d'initialisation à la suite d'intervalles $[a^{(k)}, b^{(k)}]$, $k \geq 0$, contenant un zéro de la fonction f. Le procédé de construction des intervalles emboîtés est alors le même que pour la méthode de dichotomie, à l'exception du choix de $x^{(k)}$, qui est à présent donné par l'abscisse du point d'intersection de la droite passant par les points $\left(a^{(k)}, f\left(a^{(k)}\right)\right)$ et $\left(b^{(k)}, f\left(b^{(k)}\right)\right)$ avec l'axe des abscisses, c'est-à-dire

$$x^{(k)} = a^{(k)} - \frac{a^{(k)} - b^{(k)}}{f\left(a^{(k)}\right) - f\left(b^{(k)}\right)} f\left(a^{(k)}\right) = b^{(k)} - \frac{b^{(k)} - a^{(k)}}{f\left(b^{(k)}\right) - f\left(a^{(k)}\right)} f\left(b^{(k)}\right) = \frac{f\left(a^{(k)}\right)b^{(k)} - f\left(b^{(k)}\right)a^{(k)}}{f\left(a^{(k)}\right) - f\left(b^{(k)}\right)}. \qquad (5.12)$$

On a représenté sur la figure 5.4 la construction des premières approximations $x^{(k)}$ ainsi trouvées.

k	$a^{(k)}$	$b^{(k)}$	$x^{(k)}$	$f(x^{(k)})$
0	1	2	1,5	2,375
1	1	1,5	1,25	0,328125
2	1	1,25	1,125	−0,419922
3	1,125	1,25	1,1875	−0,067627
4	1,1875	1,25	1,21875	0,124725
5	1,1875	1,21875	1,203125	0,02718
6	1,1875	1,203125	1,195312	−0,020564
7	1,195312	1,203125	1,199219	0,003222
8	1,195312	1,199219	1,197266	−0,008692
9	1,197266	1,199219	1,198242	−0,00274
10	1,198242	1,199219	1,19873	0,000239
11	1,198242	1,19873	1,198486	−0,001251
12	1,198486	1,19873	1,198608	−0,000506
13	1,198608	1,19873	1,198669	−0,000133

FIGURE 5.3 – Recherche du zéro par la méthode de *dichotomie*.

Cette méthode apparaît comme plus « flexible » que la méthode de dichotomie, le point $x^{(k)}$ ainsi construit étant plus proche de l'extrémité de l'intervalle $[a^{(k)}, b^{(k)}]$ en laquelle la valeur de la fonction $|f|$ est la plus petite. Par ailleurs, si f est une fonction linéaire, on voit que le zéro est obtenu après une itération plutôt qu'une infinité.

Indiquons que si la mesure de l'intervalle d'encadrement $[a^{(k)}, b^{(k)}]$ ainsi obtenu décroit bien lorsque k tend vers l'infini, elle ne tend pas nécessairement, à la difference de la méthode de dichotomie, vers zéro, comme l'illustre l'exemple ci-dessous.

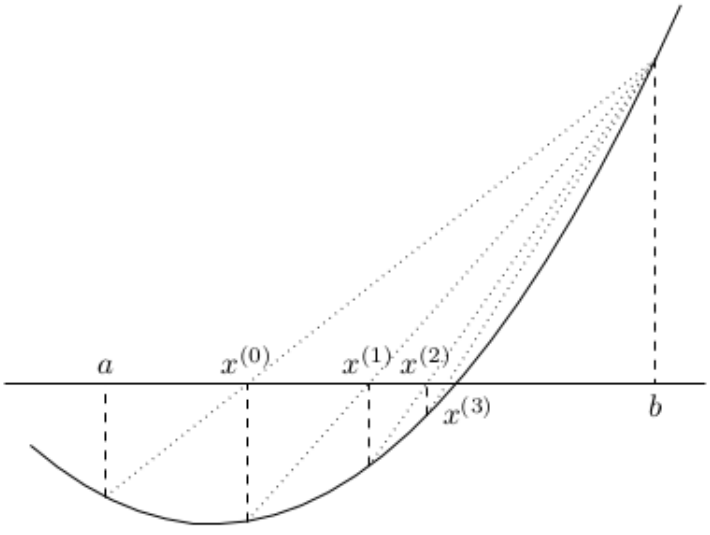


FIGURE 5.4 – Construction des premiers itérés de la méthode de la fausse position.

Exemple : On reprend l'exemple précédent en utilisant cette fois la méthode de la *fausse position*. Le tableau ci-après présente les résultats obtenus.

k	$a^{(k)}$	$b^{(k)}$	$x^{(k)}$	$f(x^{(k)})$
0	1	2	1,1	−0,549
1	1,1	2	1,151744	−0,274401
2	1,151744	2	1,176841	−0,130742
3	1,176841	2	1,188628	−0,060876
4	1,188628	2	1,194079	−0,028041
5	1,194079	2	1,196582	−0,012852
6	1,196582	2	1,197728	−0,005877
7	1,197728	2	1,198251	−0,002685
8	1,198251	2	1,19849	−0,001226
9	1,19849	2	1,1986	−0,00056
10	1,1986	2	1,198649	−0,000255

FIGURE 5.5 – Recherche du zéro par la méthode de la *fausse position*.

On observe que la borne de droite de l'intervalle d'encadrement initial est conservée tout au long du calcul.

De fait, compte tenu des hypothèses sur f, on peut voir que la méthode conduit inévitablement à partir d'un certain rang à l'une des configurations présentées à la figure 5.6, pour chacune desquelles l'une des deux bornes de l'intervalle d'encadrement n'est jamais modifiée tandis que l'autre converge de manière monotone vers le zéro de la fonction. La méthode se comporte alors comme une *méthode de point fixe* (comparer à ce titre (5.13) avec (5.17)).

Sous des hypothèses de régularité légèrement restrictives [7] sur f, on peut établir le résultat de convergence suivant pour la méthode de la fausse position.

Théorème 5.2. *Soit f une fonction de classe $\mathscr{C}^2$ sur un intervalle $[a,b]$, vérifiant $f(a)f(b)<0$, et soit $\xi \in]a,b[$ l'unique solution de l'équation f(x)=0. Alors, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ construite par la méthode de la fausse position converge linéairement vers ξ.*

Démonstration. Si f est une fonction affine, la méthode converge en une étape. Sinon, l'une des configurations illustrées à la figure 5.6 est obligatoirement atteinte par la méthode à partir d'un certain rang et l'on peut se ramener sans perte de généralité au cas où l'une des bornes de l'intervalle de départ reste fixe tout au long du processus itératif.

Supposons à présent que $f'(x)>0$ (f croissante) et $f''(x)>0$ (f convexe) sur l'intervalle $[a,b]$ (c'est la première configuration décrite plus haut). On remplace alors à l'étape $k+1$, $k\geq 0$, l'intervalle $[x^{(k)},b]$ par $[x^{(k+1)},b]$, où la borne $x^{(k+1)}$ est donnée (en choisissant, de manière un peu abusive, $x^{(0)}$ égal à a) par la formule

$$x^{(k+1)} = x^{(k)} - \frac{b-x^{(k)}}{f(b)-f\left(x^{(k)}\right)} f\left(x^{(k)}\right),\ k\geq 0\cdot \tag{5.13}$$

Étudions à présent la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ en posant $g(x) = x - \frac{b-x}{f(b)-f(x)}f(x)$, d'où $x^{(k+1)} = g\left(x^{(k)}\right)$, $\forall k\in\mathbb{N}$. La fonction g est de manière évidente de classe $\mathscr{C}^1$ sur $[a,b[$ et continue en b, avec

7. L'hypothèse « f dérivable » est en effet suffisante pour établir le résultat, mais demanderait une preuve plus élaborée.

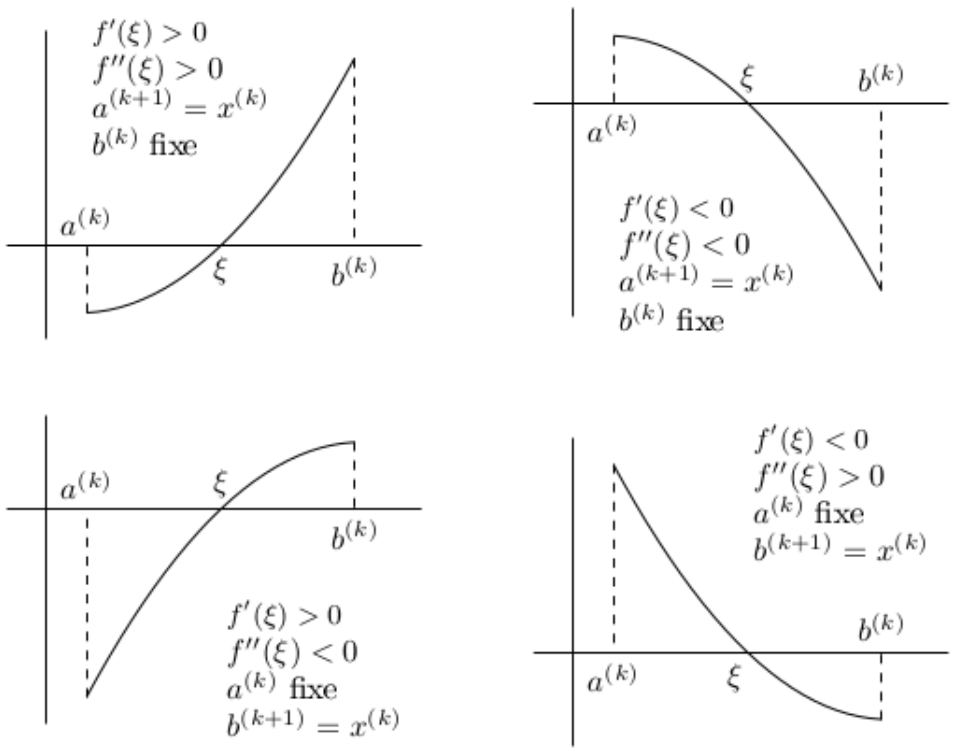


FIGURE 5.6 – Différentes configurations atteintes par la méthode de la fausse position à partir d'un certain rang.

$g(b) = b - \frac{f(b)}{f'(b)}$. On a par ailleurs

$$g'(x) = 1 - \frac{b-x}{f(b)-f(x)} f'(x) + \frac{f(b)-f(x)-(b-x)f'(x)}{(f(b)-f(x))^2} f(x) = \frac{f(b)-f(x)-(b-x)f'(x)}{(f(b)-f(x))^2} f(b),\ \forall x \in [a,b[, \qquad (5.14)$$

dont on déduit la continuité de g' en b, avec $g'(b) = \frac{f(b)f''(b)}{2f'(b)^2}$. L'application g est donc de classe $\mathscr{C}^1$ sur $[a,b]$. La fonction f étant supposée convexe sur $[a,b]$, on a $f(b) - f(x) - (b-x)f'(x) \geq 0$, $\forall x \in [a,b]$, ainsi que $f(b) > 0$, puisque f est croissante et $f(a)f(b) < 0$. Par conséquent, g est croissante sur $[a,b]$ et alors $g([a,b]) \subset [g(a), g(b)]$. Enfin, on utilise la croissance de f et le fait que $f(a) < 0$ et $f(b) > 0$ pour montrer que $g(a) = a - \frac{b-a}{f(b)-f(a)]} f(a) \geq a$ et $g(b) = b - \frac{f(b)}{f'(b)} \leq b$.

La suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ est donc croissante et majorée par b ; elle converge vers une limite $\ell \in [a,b]$, qui vérifie, par continuité de g, $g(\ell) = \ell$. Puisque $x^{(0)} = a < \xi$, on a, par récurrence, $x^{(k)} < \xi$, $\forall k \in \mathbb{N}$, et donc $\ell \leq \xi$, d'où $\ell \in]a,b[$ et, par suite, $f(\ell) = 0$ donc $\ell = \xi$, par unicité de ξ.

Pour prouver que la convergence est linéaire, on doit montrer que

$$0 < \lim_{k\to+\infty} \frac{x^{(k+1)} - \xi}{x^{(k)} - \xi} < 1\cdot \qquad (5.15)$$

Or, le théorème des accroissement finis [8] et la continuité de la fonction g' impliquent que

$$\lim_{k\to+\infty} \frac{x^{(k+1)}-\xi}{x^{(k)}-\xi} = g'(\xi) = 1 - \frac{b-\xi}{f(b)-f(\xi)} f'(\xi) \tag{5.16}$$

et, f étant strictement convexe, il est facile de voir que la pente de la droite passant par les points $(\xi, 0)$ et $(b, f(b))$ est strictement plus grande que celle de la tangente à la courbe représentative de $f$au point ξ, d'où la conclusion.

La même technique de démonstration s'adapte pour traiter les trois cas (pour lesquels les signes de $f'(x)$ et $f''(x)$ sont constants sur $[a, b]$) restants, ce qui achève la preuve.

On notera que le critère d'arrêt des itérations doit nécessairement être basé sur la valeur du résidu $f\left(x^{(k)}\right)$, puisque la longueur de l'intervalle d'encadrement du zéro de$f$ ne tend pas nécessairement vers zéro.

5.4 Méthodes de point fixe

Les méthodes d'approximation de zéros introduites dans la suite se passent de l'hypothèse de changement de signe de f en ξ et ne consistent pas en la construction d'une suite d'intervalles contenant le zéro de la fonction ; bien qu'étant aussi des méthodes itératives, ce ne sont pas des méthodes d'encadrement. Rien ne garantit d'ailleurs que la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ produite par l'un des algorithmes présentés prend ses valeurs dans un intervalle fixé *à priori*.

D'autre part, comme nous l'avons déjà vu avec la méthode de la fausse position, prendre en compte les informations données par les valeurs de la fonction f et même, dans le cas où celle-ci est différentiable, celles de sa dérivée aux points $x^{(k)}$, $k \in \mathbb{N}$, peut conduire à des propriétés de convergence ameliorées. On verra que les méthodes présentées exploitent ce « principe » sous différentes formes.

Les sections 5.4.3 et 5.4.4 sont respectivement consacrées aux méthodes de la corde et de Newton-Raphson, qui sont ensuite analysées à la section 5.4 dans le cadre général des méthodes de point fixe. La méthode de la sécante est introduite et analysée dans la section 5.4.5.

5.4.1 Principe

La famille de méthodes que nous allons maintenant introduire utilise le fait que le problème $f(x) = 0$ peut toujours ramener au problème équivalent $x - g(x) = 0$, pour lequel on a le résultat suivant.

Théorème 5.3 (« théorème du point fixe de Brouwer »). [9] *Soit $[a, b]$ un intervalle non vide de $\mathbb{R}$ et g une application continue de $[a, b]$ dans lui-même. Alors, il existe un point ξ de $[a, b]$, appelé* ***point fixe de la fonction*** *g, vérifiant $g(\xi) = \xi$.*

8. Pour toute fonction réelle d'une variable réelle $f : [a, b] \to \mathbb{R}$ (a et b réels tels que $a < b$), supposée **continue** sur l'intervalle fermé $[a, b]$ et **dérivable** sur l'intervalle ouvert $]a, b[$, il existe un réel c dans $]a, b[$ vérifiant

$$\frac{f(b)-f(a)}{b-a} = f'(c).$$

Graphiquement, le **théorème des accroissements finis** indique que, pour toute droite sécante en deux points à une courbe différentiable, il existe une tangente parallèle à la sécante.

On peut illustrer ainsi le théorème : « Si un véhicule parcourt une distance à la vitesse moyenne de 60 km/h, alors son compteur (censé indiquer avec une précision infinie la vitesse instantanée) a indiqué au moins une fois la vitesse précise de 60 km/h. »

9. Luitzen Egbertus Jan Brouwer (27 février 1881-2 décembre 1996) était mathématicien et philosolphe néerlandais. Ses apports concernèrent principalement la topologie et la logique formelle.

Démonstration. Posons $f(x) = x - g(x)$. On a alors $f(a) = a - g(a) \leq 0$ et $f(b) = b - g(b) \geq 0$, puisque $g(x) \in [a, b]$ pour tout $x \in [a, b]$. Par conséquent, f est une fonction continue sur [a,b], telle que $f(a)f(b) \leq 0$. Le théorème 5.1 assure alors l'existence d'un point ξ dans $[a, b]$ tel que $0 = f(\xi) = \xi - g(\xi)$.

Bien entendu, toute équation de la forme $f(x) = 0$ peut s'écrire sous la forme $x = g(x)$ en posant $g(x) = x + f(x)$, mais cela ne garantit en rien que la fonction auxiliaire g ainsi définie satisfait les hypothèses du théorème 5.3. Il existe cependant de nombreuses façons de construire g à partir de f, comme le montre l'exemple ci-après, et il suffit donc de trouver une transformation adaptée.

Exemple. Considérons la fonction $f(x) = e^x - 2x - 1$ sur l'intervalle $[1, 2]$. Nous avons $f(1) < 0$ et $f(2) > 0$, f possède donc bien un zéro sur l'intervalle[1, 2]. Soit $g(x) = \frac{1}{2}(e^x - 1)$. L'équation $x = g(x)$ est bien équivalente à $f(x) = 0$, mais g, bien que continue, n'est pas à valeurs de $[1, 2]$ dans lui-même. Réécrivons à présent le problème en posant $g(x) = \ln(2x + 1)$. Cette dernière fonction est continue et croissante sur l'intervalle $[1, 2]$, à valeurs dans lui-même. Elle satisfait donc les conditions du théorème 5.3.

Nous venons de montrer que, sous certaines conditions, approcher les zéros d'une fonction f revient à approcher les points fixes d'une fonction g, sans que l'on sache pour autant traiter ce nouveau problème. Une méthode courante pour la détermination de point fixe se résume à la construction d'une suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ par le procédé itératif suivant : étant donné $x^{(0)}$ (appartenant à [a,b]), on pose

$$x^{(k+1)} = g\left(x^{(k)}\right),\ k \geq 0\cdot \tag{5.17}$$

On dit que la relation (5.17) est une *itération de poin fixe*. La méthode d'approximation résultante est appelée *méthode de point fixe* ou bien encore *méthode des approximations successives*. Si la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par (5.17) converge, cela ne peut être que vers un point fixe de g. En effet, en posant

$$\lim_{k\to+\infty} x^{(k)} = \xi\,,$$

nous avons que

$$\xi = \lim_{k\to+\infty} x^{(k+1)} = \lim_{k\to+\infty} g\left(x^{(k)}\right) = g\left(\lim_{k\to+\infty} x^{(k)}\right) = g(\xi)\,, \tag{5.18}$$

la deuxième égalité provenant de la définition (5.17) de la suite récurrente, et la troisième étant une conséquence de la continuité de g.

5.4.2 Quelques résultats de convergence

Le choix de la fonction g pour mettre en oeuvre cette méthode n'étant pas unique, celui-ci est alors motivé par les exigences du théorème 5.4 qui donne des conditions *suffisantes* sur g de convergence vers un zéro de la fonction f. Avant de l'énoncer, rappelons tout d'abord la notion d'*application contractante*.

Définition 5.4 (application contractante)**.** *Soit $[a, b]$ un intervalle non vide de $\mathbb{R}$ et g une application de $[a, b]$ dans $\mathbb{R}$. On dit que g est une application* ***contractante*** *si et seulement si il existe une constante K telle que $0 < K < 1$ vérifiant*

$$|g(x) - g(y)| \leq K|x - y|,\ \forall x \in [a, b]\,,\ \forall y \in [a, b]\cdot \tag{5.19}$$

On notera que la *constante de Lipschitz*[10] de g n'est autre que la plus petite constante K vérifiant la condition (5.19).

10. Rudolph Otto Sigismund Lipschitz(14 mai 1832- 7 octobre 1903) était un mathématicien allemand. Son travail s'étend sur des domaines aussi variés que la théorie des nombres, l'analyse, la géométrie différentielle et la mécanique classique.

Le résultat suivant est une application dans le cas réel du *théorème du point fixe de Banach*[11](également attribué à Picard[12]), dont l'énoncé général vaut pour toute application contractante définie sur un *espace métrique complet.*

Théorème 5.4. *Soit $[a,b]$ un intervalle non vide de $\mathbb{R}$ et g une application contractante de $[a,b]$ dans lui-même. Alors, la fonction g possède un unique point fixe ξ dans $[a,b]$. De plus, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par la relation (5.17) converge, pour toute initialisation $x^{(0)}$ dans $[a,b]$, vers ce point fixe et l'on a les deux estimations suivantes :*

$$\left|x^{(k)}-\xi\right| \leq K^k \left|x^{(0)}-\xi\right|, \ \forall k \geq 0, \tag{5.20}$$

$$\left|x^{(k+1)}-\xi\right| \leq \frac{K}{1-K}\left|x^{(k)}-x^{(k-1)}\right|, \ \forall k \geq 1. \tag{5.21}$$

Démonstration. On commence par montrer que la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ est une suite de Cauchy. En effet, pour tout entier k non nul, on a

$$\left|x^{(k+1)}-x^{(k)}\right| = \left|g\left(x^{(k)}\right)-g\left(x^{(k-1)}\right)\right| \leq K\left|x^{(k)}-x^{(k-1)}\right|, \tag{5.22}$$

par hypothèse, et on obtient par récurrence que

$$\left|x^{(k+1)}-x^{(k)}\right| = K^k\left|x^{(1)}-x^{(0)}\right|, \ \forall k \in \mathbb{N}\cdot \tag{5.23}$$

On en déduit, par une application répétée de l'inégalité triangulaire, que, $\forall k \in \mathbb{N}$, $\forall p > 2$,

$$\begin{aligned}\left|x^{(k+p)}-x^{(k)}\right| &\leq \left|x^{(k+p)}-x^{(k+p-1)}\right| + \left|x^{(k+p-1)}-x^{(k+p-2)}\right| + \cdots + \left|x^{(k+1)}-x^{(k)}\right| \\ &\leq \left(K^{p-1}+K^{p-2}+\cdots+1\right)\left|x^{(k+1)}-x^{(k)}\right| \\ &\leq \frac{1-K^p}{1-K}K^k\left|x^{(1)}-x^{(0)}\right|,\end{aligned} \tag{5.24}$$

le dérnier membre tendant vers zéro lorsque k tend vers l'infini. La suite réelle $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ converge donc vers une limite ξ dans $[a,b]$. L'application g étant continue[13], on déduit alors par un passage à la limite dans (5.17) que $\xi = g(\xi)$.

Supposons à présent que g possède deux points fixes ξ et ζ dans l'intervalle $[a,b]$. On a alors

$$0 \leq |\xi-\zeta| = |g(\xi)-g(\zeta)| \leq K|\xi-\zeta|, \tag{5.25}$$

d'où $\xi = \zeta$ puisque $K < 1$. La première estimation se prouve alors par récurrence sur k en écrivant que

$$\left|x^{(k)}-\xi\right| = \left|g\left(x^{(k-1)}\right)-g(\xi)\right| \leq \left|x^{(k-1)}-\xi\right|, \forall k \geq 1, \tag{5.26}$$

et la seconde est obtenue en utilisant que

$$\left|x^{(k+p)}-x^{(k)}\right| \leq \frac{1-K^p}{1-K}\left|x^{(k+1)}-x^{(k)}\right| \leq \frac{1-K^p}{1-K}K\left|x^{(k)}-x^{(k-1)}\right|, \ \forall k \geq 1, \ \forall p \geq 1 \tag{5.27}$$

11. Stefan Banach (30 mars 1892- 31 août 1945) était un mathématicien polonais. Il est l'un des fondateurs de l'analyse fonctionnelle moderne et introduisit notamment des espaces vectoriels normés complets, aujourd'hui appelés *espaces de Banach*, lors de son étude des espaces vectoriels topologiques. Plusieurs importants théorèmes et un célèbre paradoxe sont associés à son nom.

12. Charles Émile Picard (24 juillet 1856-11 décembre 1941) était un mathématicien français, également philosophe et historien des sciences. Il est l'auteur de deux difficiles théorèmes en analyse complexe et fut le premier à utiliser le théorème du point fixe de Banach dans une méthode d'approximations successives de solutions d'équations différentielles ou d'équations aux dérivées partielles.

13. C'est par hypothèse une application K-lipschitzienne.

et en faisant tendre p vers l'infini.

Sous les hypothèses du théorème 5.4, la convergence des itérations de point fixe est assurée quel que soit le choix de la valeur initiale $x^{(0)}$ dans l'intervalle $[a, b]$: c'est donc un nouvel exemple de convergence *globale*. Par ailleurs, l'un des intérêts de ce résultat est de donner une estimation de la vitesse de convergence de la suite vers sa limite, la première inégalité montrant en effet que la convergence est *géométrique*. La seconde inégalité est aussi particulièrement utile d'un point de vue applicatif, car elle fournit à chaque étape un majorant de la distance à la limite (sans pour autant la connaître) en fonction d'une quantité connue. Il est alors posssible de majorer le nombre d'itérations que l'on doit effectuer pour approcher le point fixe ξ avec une précision donnée.

Corollaire 1. *Considérons la méthode de point fixe définie par la relation (5.17), la fonction g vérifiant les hypothèses du théorème 5.4. Étant données une précision $\varepsilon > 0$ et une initialisation $x^{(0)}$ dans l'intervalle $[a, b]$, soit $k_0(\varepsilon)$ le plus petit entier tel que*

$$\left|x^{(k)} - \xi\right| \leq \varepsilon,\ \forall k \geq k_0(\varepsilon) \tag{5.28}$$

On a alors la majoration

$$k_0(\varepsilon) \leq \left\lfloor \frac{\ln(\varepsilon) + \ln(1-K) - \ln\left(\left|x^{(1)} - x^{(0)}\right|\right)}{\ln(K)} \right\rfloor + 1\,, \tag{5.29}$$

où, pour tout réel x, $\lfloor x \rfloor$ désigne la partie entière par défaut de x.

Démonstration. En utilisant l'inégalité triangulaire et l'inégalité (5.21) pour $k = 1$, on trouve que

$$\left|x^{(0)} - \xi\right| \leq \left|x^{(0)} - x^{(1)}\right| + \left|x^{(1)} - \xi\right| \leq \left|x^{(0)} - x^{(1)}\right| + K\left|x^{(0)} - \xi\right|, \tag{5.30}$$

d'où

$$\left|x^{(0)} - \xi\right| \leq \frac{K}{1-K}\left|x^{(0)} - x^{(1)}\right|. \tag{5.31}$$

En substituant cette expression dans (5.21), on obtient que

$$\left|x^{(k)} - \xi\right| \leq \frac{K^k}{1-K}\left|x^{(0)} - x^{(1)}\right|, \tag{5.32}$$

et on aura en particulier $\left|x^{(k)} - \xi\right| \leq \varepsilon$ si k est tel que

$$\frac{K^k}{1-K}\left|x^{(0)} - x^{(1)}\right| \leq \varepsilon. \tag{5.33}$$

En prenant le logaritme néperien de chacun des membres de cette dernière inégalité, on arrive à

$$k \geq \frac{\ln(\varepsilon) + \ln(1-K) - \ln\left(\left|x^{(1)} - x^{(0)}\right|\right)}{\ln(K)}, \tag{5.34}$$

dont on déduit le résultat.

Dans la pratique, vérifier que l'application g est K-lipschitzienne n'est pas toujours aisé. Lorsque g est une fonction de classe $\mathscr{C}^1$ sur l'intervalle $[a, b]$, il est possible d'utiliser la caractérisation suivante.

Proposition 5.2. *Soit $[a, b]$ un intervalle non vide de $\mathbb{R}$ et g une fonction de classe $\mathscr{C}^1$ définie de $[a, b]$ dans lui-même vérifiant*

$$|g'(x)| \leq K < 1,\ \forall x \in [a, b]. \tag{5.35}$$

Alors, g est une application contractante sur $[a, b]$.

Démonstration. D'après le théorème des accroissements finis, pour tous x et y contenus dans l'intervalle $[a, b]$ et distants, on sait qu'il existe un réel c strictement compris entre x et y tel que

$$|g(x) - g(y)| = |g'(c)||x - y|, \tag{5.36}$$

d'où le résultat.

On est alors en mesure d'affiner le résultat de convergence globale précédent dans ce chapitre dans ce cas particulier.

Théorème 5.5. *Soit $[a, b]$ un intervalle non vide dans $\mathbb{R}$ et g une application satisfaisant les hypothèses de la proposition 5.2. Alors, la fonction g possède un unique point fixe ξ dans $[a, b]$ et la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par (5.17) converge, pour toute initialisation $x^{(0)}$ dans $[a, b]$, vers ce point fixe. De plus, on a*

$$\lim_{k\to+\infty} \frac{x^{(k+1)} - \xi}{x^{(k)} - \xi} = g'(\xi), \tag{5.37}$$

la convergence est donc au moins linéaire.

Démonstration. La proposition 5.2 établissant que g est une application contractante sur $[a, b]$, les conclusions de théorème 5.4 sont valides et il ne reste qu'à prouver l'égalité (5.37). En vertu du théorème des accroissements finis, il existe pour tout $k \geq 0$, un réel $\eta^{(k)}$ strictement compris entre $x^{(k)}$ et ξ tel que

$$x^{(k+1)} - \xi = g\left(x^{(k)}\right) - g(\xi) = g'\left(\eta^{(k)}\right)\left(x^{(k)} - \xi\right) \tag{5.38}$$

La suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ convergeant vers ξ, cette égalité implique que

$$\lim_{k\to+\infty} \frac{x^{(k+1)} - \xi}{x^{(k)} - \xi} = \lim_{k\to+\infty} g'\left(\eta^{(k)}\right) = g'(\xi). \tag{5.39}$$

On notera que ce théorème assure une convergence au *moins linéaire* de la méthode de point fixe. La quantité $|g'(\xi)|$ est appelée, par comparaison avec la constante C apparaissant dans (5.2), *facteur de convergence asymptotique* de la méthode.

Encore une fois, il est souvent difficile en pratique de déterminer *a priori* un intervalle $[a, b]$ sur lequel les hypothèses de la proposition 5.2 sont vérifiées. Il est néanmoins possible de se contenter d'hypothèses plus faibles, au prix d'un résultat de convergence seulement *locale*.

Théorème 5.6. *Soit $[a, b]$ un intervalle non vide de $\mathbb{R}$, une fonction g continue de $[a, b]$ dans lui-même et ξ un point fixe de g dans $[a, b]$. On suppose de plus que g admet une dérivée continue dans un voisinage de ξ, avec $|g'(\xi)| < 1$. Alors, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par (5.17) converge vers ξ, pour toute initialisation $x^{(0)}$ choisie suffisamment proche de ξ.*

Démonstration. Par hypothèses sur la fonction g, il existe un réel $h > 0$ tel que g' est continue sur l'intervalle $[\xi - h, \xi + h]$. Puisque $|g'(\xi)| < 1$, on peut alors trouver un intervalle $I_\delta = [\xi - \delta, \xi + \delta]$, avec $0 < \delta < h$, tel que $|g'(x)| \leq L$, avec $L < 1$, pour tout x appartenant à I_δ. Pour cela, il suffit de poser $L = \frac{1}{2}(1 + |g'(\xi)|)$ et d'utiliser la continuité de g' pour choisir $\delta \leq h$ de manière à ce que

$$|g'(x) - g'(\xi)| \leq \frac{1}{2}(1 - |g'(\xi)|),\ \forall x \in I_\delta. \tag{5.40}$$

On en déduit alors que

$$|g'(x)| \leq |g'(x) - g'(\xi)| + |g'(\xi)| \leq \frac{1}{2}(1 - |g'(\xi)|) + |g'(\xi)| = L,\ \forall x \in I_\delta. \tag{5.41}$$

Supposons à présent que, pour un entier k donné, le terme $x^{(k)}$ de la suite définie par la relation de récurrence (5.17) appartient à I_δ. On a alors, en vertu du théorème des accroissements finis,

$$x^{(k+1)} - \xi = g\left(x^{(k)}\right) - \xi = g\left(x^{(k)}\right) - g(\xi) = g'\left(\eta^{(k)}\right)\left(x^{(k)} - \xi\right), \tag{5.42}$$

avec $\eta^{(k)}$ compris entre $x^{(k)}$ et ξ, d'où

$$\left|x^{(k+1)} - \xi\right| \leq L\left|x^{(k)} - \xi\right|, \tag{5.43}$$

et $x^{(k+1)}$ appartient lui aussi à I_δ. On montre alors par récurrence que, si $x^{(0)}$ appartient à I_δ, alors $x^{(k)}$ également, $\forall k \geq 0$, et que

$$\left|x^{(k)} - \xi\right| \leq L^k\left|x^{(0)} - \xi\right|, \tag{5.44}$$

ce qui implique que la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$.

On peut observer que, si $|g'(\xi)| > 1$ et si $x^{(k)}$ est suffisamment proche de ξ pour avoir $|g'\left(x^{(k)}\right) > 1$, on obtient $\left|x^{(k+1)} - \xi\right| > \left|x^{(k)} - \xi\right|$ et la convergence ne peut alors avoir lieu (sauf si $x^{(k)} = \xi$. Dans le cas où $\left|g'(x^{(k)})\right| > 1$, il peut y avoir convergence ou divergence selon les cas considérés. Cette remarque et le thórème 5.6 conduisent à l'introduction des définitiions suivantes.

Définition 5.5. *Soit $[a, b]$ un intervalle non vide de $\mathbb{R}$, une fonction g continue de $[a, b]$ dans lui-même et ξ un point fixe de g dans $[a, b]$. On dit que ξ est* ***un point fixe attractif*** *si la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par l'itération de point fixe (5.17) converge pour toute initialisation $x^{(0)}$ suffisamment proche de ξ. Réciproquement, si cette suite ne converge pour aucune initialisation $x^{(0)}$ dans un voisinage de ξ, exceptée $x^{(0)} = \xi$, le point fixe est dit* ***répulsif****.*

5.4.3 Méthode de relaxation ou de la corde

Nous avons vu dans la section 5.4.1 que l'on pouvait obtenir de diverses manières une fonction g dont les points fixes sont les zéros de la fonction f. Beaucoup de méthodes parmi les plus courantes s'appuient néamoins sur le choix de la forme suivante

$$g(x) = x + h(x)f(x), \tag{5.45}$$

avec h une fonction satisfaisant $0 < |h(x)| < +\infty$ sur le domaine de définition (ou plus généralement sur un intervalle contenant un zéro) de f. Sous cette hypothèse, on verifie facilement que tout zéro de f est point fixe de g, et vice versa.

Le choix le plus simple pour la fonction h est alors celui conduisant à la *méthode de relaxation*, qui consiste en la construction d'une suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ satisfaisant la relation de récurrence

$$x^{(k+1)} = x^{(k)} - \lambda f\left(x^{(k)}\right), \forall k \geq 0, \tag{5.46}$$

avec λ un réel fixé, la valeur de $x^{(0)}$ étant donné.

En supposant f différentiable dans un voisinage de son zéro ξ, rien ne garantit que la méthode converge si $f'(\xi) = 0$ mais on voit qu'on peut facilement assurer la convergence locale de cette méthode si ξ est un zéro simple et λ est tel que $0 < \lambda f'(\xi) < 2$. Ceci est rigoureusement établi dans le théorème suivant.

Théorème 5.7. *Soit f une fonction réelle de classe $\mathscr{C}^1$ dans un voisinage du réel ξ tel que $f(\xi) = 0$. Supposons que $f'(\xi) \neq 0$. Alors il existe un ensemble de réels λ tel que la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par (5.46) converge au moins linéairement vers ξ, pour toute initialisation $x^{(0)}$ choisie suffisamment proche de ξ.*

Démonstration. Supposons que $f'(\xi) > 0$, la preuve étant identique, aux changements de signe près, si $f'(\xi) < 0$. La fonction f' étant continue dans un voisinage de ξ, on peut trouver un réel $\delta > 0$ tel que $f'(x) \geq \frac{1}{2} f'(\xi)$ dans l'intervalle $I_\delta = [\xi - \delta, \xi + \delta]$. Posons

$$M = \max_{x \in I_\delta} f'(x)\cdot$$

On a alors

$$1 - \lambda M \leq 1 - \lambda f'(x) \leq 1 - \frac{\lambda}{2} f'(\xi),\ \forall x \in I_\delta. \tag{5.47}$$

On choisit alors λ de façon à ce que $\lambda M - 1 = 1 - \frac{\lambda}{2} f'(\xi)$, c'est-à- dire

$$\lambda = \frac{4}{2M + f'(\xi)}\cdot \tag{5.48}$$

En posant $g(x) = x - \lambda f(x)$, on obtient que

$$g'(x) \leq \frac{2M - f'(\xi)}{2M + f'(\xi)} < 1,\ \forall x \in I_\delta, \tag{5.49}$$

et la convergence se déduit alors du théorème 5.5.

D'un point de vue géométrique, le point $x^{(k+1)}$ dans (5.46) est, à chaque itération, l'abscisse du point d'intersection entre la droite de pente $1/\lambda$ passant par le point $\left(x^{(k)}, f\left(x^{(k)}\right)\right)$ et l'axe des abscisses. Elle est pour cette raison aussi appelée *méthode de la corde*, le nouvel itéré de la suite étant déterminé par la corde de pente constante joignant un point de la courbe de la fonction f à l'axe des abscisses. Connaissant un intervalle d'encadrement $[a, b]$ de ξ, on a coutume de définir la méthode de la corde par

$$x^{(k+1)} = x^{(k)} - \frac{b - a}{f(b) - f(a)} f\left(x^{(k)}\right),\ \forall k \geq 0, \tag{5.50}$$

avec $x^{(0)}$ donné dans $[a, b]$. Sous les hypothèses du théorème 5.7, la méthode converge si l'intervalle $[a, b]$ est telle que

$$b - a < 2\frac{f(b) - f(a)}{f'(\xi)}. \tag{5.51}$$

On remarque que la méthode de la corde converge en une itération si f est affine.

5.4.4 Méthode de Newton-Raphson

En supposant que la fonction f est de classe $\mathscr{C}^1$ et que ξ est un zéro simple, la *méthode de Newton-Raphson* fait le choix

$$h(x) = -\frac{1}{f'(x)} \tag{5.52}$$

dans (5.45). La relation de récurrence définissant cette méthode est alors

$$x^{(k+1)} = x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)},\ \forall k \geq 0, \tag{5.53}$$

l'initialisation $x^{(0)}$ étant donnée.

Cette méthode peut être interprétée comme une *linéarisation de l'équation* $f(x) = 0$ *au point* $x = x^{(k)}$. En effet, si l'on remplace $f(x)$ au voisinage du point $x^{(k)}$ par l'approximation affine obtenue en tronquant au premier ordre le développement de Taylor de f en $x^{(k)}$ et qu'on résoud l'équation linéaire résultante

$$f\left(x^{(k)}\right) + \left(x - x^{(k)}\right) f'\left(x^{(k)}\right) = 0\,, \tag{5.54}$$

en notant sa solution $x^{(k+1)}$, on retrouve l'égalité (5.53). Il en résulte que, géométriquement parlant, le point $x^{(k+1)}$ est l'abscisse du point d'intersection entre la tangente à la courbe de f au point $\left(x^{(k)}, f\left(x^{(k)}\right)\right)$ et l'axe des abscisses (voir figure 5.7).

Par rapport à toutes les méthodes introduites jusqu'à présent, on pourra remarquer que la méthode de Newton nécessite à chaque itération l'évaluation des deux fonctions f et f' au point courant $x^{(k)}$. Cet effort est compensé par une vitesse de convergence accrue, puisque cette méthode est d'ordre deux.

Théorème 5.8. *Soit f une fonction réelle de classe $\mathscr{C}^1$ dans un voisinage du réel ξ tel que $f(\xi) = 0$. Supposons que $f'(\xi) \neq 0$. Alors la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ définie par (5.53) converge au moins quadratiquement vers ξ, pour toute initialisation $x^{(0)}$ choisie suffisamment proche de ξ.*

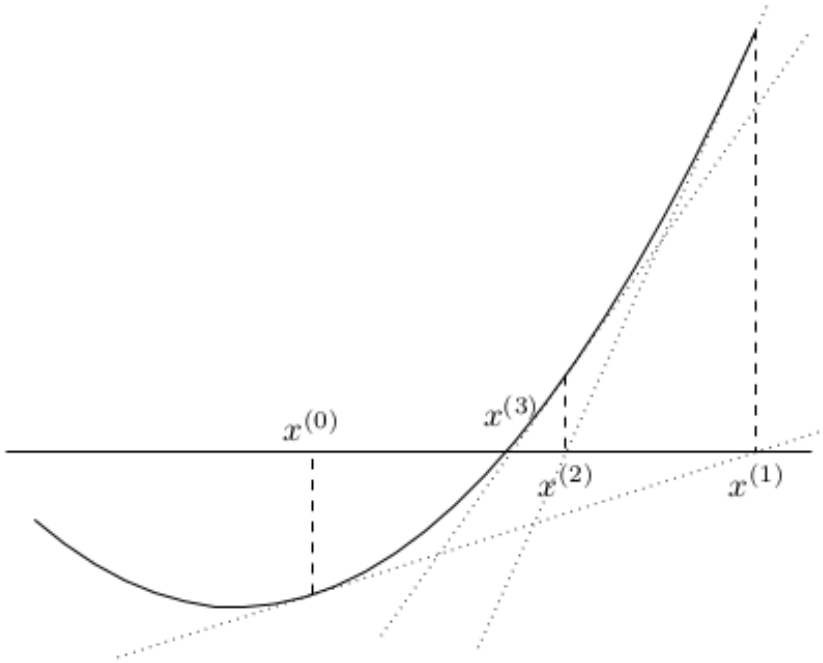


FIGURE 5.7 – Construction des premiers itérés de la méthode de Newton-Raphson.

5.4.5 Méthode de la sécante

La *méthode de la sécante* peut être considérée comme une variante de la méthode de la corde, dans laquelle la pente de la corde est mise à jour à chaque itération, ou bien une modification de la méthode de la fausse position permettant de se passer de l'hypothèse sur le signe de la fonction f aux extrémités de l'intervalle d'encadrement initial (il n'ya d'ailleurs plus besoin de connaitre un tel intervalle). On peut aussi le voir comme une méthode de Newton dans laquelle la donnée de la dérivée $f'\left(x^{(k)}\right)$ serait remplacée par une approximation obtenue par une différence finie. C'est l'une des méthodes que l'on peut employer lorsque la dérivée de f est compliquée, voire impossible [14], à calculer ou encore coûteuse à évaluer.

A partir de la donnée de deux valeurs initiale $x^{(-1)}$ et $x^{(0)}$, telles que $x^{(-1)} \neq x^{(0)}$, on utilise la relation de récurrence

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f\left(x^{(k)}\right) - f\left(x^{(k-1)}\right)} f\left(x^{(k)}\right), \ \forall k \geq 0, \tag{5.55}$$

pour obtenir les approximations successives du zéro recherché.

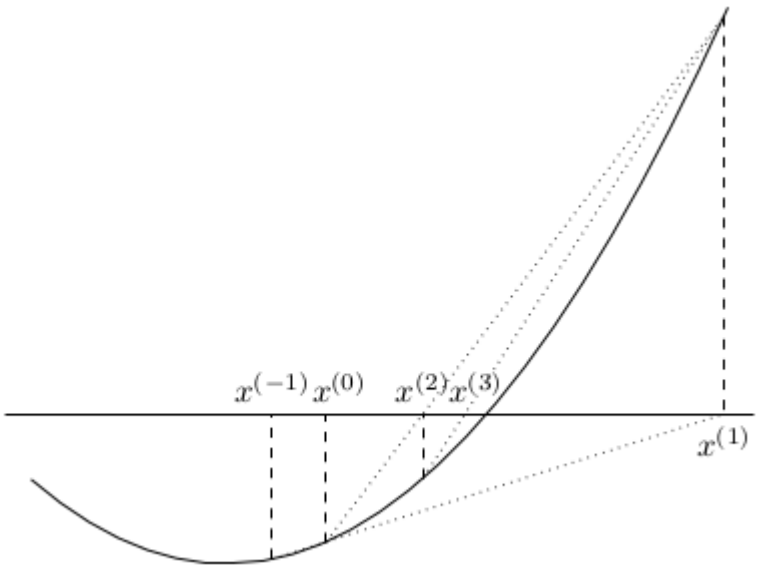


FIGURE 5.8 – Construction des premiers itérés de la méthode de la sécante.

Bien que l'on doive disposer de deux estimations de ξ avant de pouvoir utiliser la relation de récurrence (5.55), cette méthode ne requiert à chaque étape qu'une seule évaluation de fonction, ce qui est un avantage par rapport à la méthode de Newton, dont la relation (5.53) demande de connaître les valeurs de $f\left(x^{(k)}\right)$ et de $f'\left(x^{(k)}\right)$. Cependant, à la différence de la méthode de la fausse position, rien n'assure qu'au moins un zéro de f se trouve entre $x^{(k-1)}$ et $x^{(k)}$, pour tout $k \in \mathbb{N}$. Enfin, comparée à la méthode de la corde, elle nécessite le calcul de « mise à jour » du quotient apparaissant dans (5.55). Le bénéfice tiré de cet effort supplémentaire est bien une vitesse de convergence *superlinéaire*, mais cette convergence n'est plus que *locale*, comme le montre le résultat suivant [15].

Théorème 5.9. *Supposons que f est une fonction de classe $\mathscr{C}^2$ dans un voisinage d'un zéro simple ξ. Alors, si les données $x^{(-1)}$ et $x^{(0)}$, avec $x^{(-1)} \neq x^{(0)}$, choisie dans ce voisinage, sont suffisamment*

14. C'est le cas si la fonction f n'est connue qu'*implicitement*, par exemple lorsque c'est la solution d'une équation differentielle et x est un paramètre de la donnée initiale du problème associé.

15. Notons qu'on peut utiliser les techniques introduites pour les méthodes de point fixe pour établir un résultat de convergence, la relation (5.55) pouvant s'écrire sous la forme (5.17) voulue.

proches de ξ, la suite définie par (5.55) converge vers ξ avec un ordre (au moins) égal à $\frac{1}{2}(1+\sqrt{5}) = 1,6180339887\cdots$

Démonstration. Nous allons tout d'abord prouver la convergence locale de la méthode. À cette fin, introduisons, pour $\delta > 0$, l'ensemble $\{I_\delta = x \in \mathbb{R} || x - \xi| \leq \delta\}$ et supposons que f est de classe $\mathbb{C}^2$ dans ce voisinage de ξ. Pour δ suffisamment petit, définissons

$$M(\delta) = \max_{\substack{s\in I_\delta \\ t\in I_\delta}} \left| \frac{f''(s)}{2f'(t)} \right|, \tag{5.56}$$

et supposons que δ est tel que [16]

$$\delta M(\delta) < 1. \tag{5.57}$$

Le nombre ξ est l'unique zéro de f contenu dans I_δ. En effet, en appliquant la formule de Taylor-Lagrange à l'ordre deux à f au point ξ, on trouve que

$$f(x) = f(\xi) + (x-\xi)f'(\xi) + \frac{1}{2}(x-\xi)^2 f''(c), \tag{5.58}$$

avec c compris entre x et ξ. Si $x \in I_\delta$, On a également $c \in I_\delta$ et on obtient

$$f(x) = (x-\xi)f'(\xi)\left(1 + (x-\xi)\frac{f''(c)}{2f'(\xi)}\right). \tag{5.59}$$

Si $x \in I_\delta$ et $x \neq \xi$, les trois facteurs dans le membre de droite sont tous différents de zéro (le dernier parce que $\left|(x-\xi)\frac{f''(c)}{2f'(\xi)}\right| < \delta M(\delta) < 1$) et la fonction f ne s'annule qu'en ξ sur l'intervalle I_δ.

Montrons à présent que, quelles que soient les initialisations $x^{(-1)}$ et $x^{(0)}$, avec $x^{(-1)} \neq x^{(0)}$, dans I_δ, la suite $\left(x^{(k)}\right)_{k\in\mathbb{N}}$ construite par la méthode de la sécante converge vers ξ en prouvant que, pour tout $k \geq 0$, $x^{(k)}$ appartient à I_δ et que deux itérés successifs $x^{(k)}$ et $x^{(k-1)}$ sont distincts, sauf si $f\left(x^{(k)}\right) = 0$ pour k donné, auquel cas la méthode aura convergé en un nombre fini d'itérations. On raisonne par récurrence, le résultat étant vrai par hypothèse pour $k = 0$. Supposons que $x^{(k)}$ et $x^{(k-1)}$ appartiennent à I_δ, avec $x^{(k)} \neq x^{(k-1)}$, pour $k \geq 1$. Utilisons (5.55) pour obtenir une relation faisant intervenir les trois erreurs consécutives $\left(x^{(i)} - \xi\right), i = k-1,\ k,\ k+1$. En soustrayant ξ dans chaque membre de (5.55) et en se servant que $f(\xi) = 0$, il vient

$$x^{(k+1)} - \xi = x^{(k)} - \xi - \frac{x^{(k)} - x^{(k-1)}}{f\left(x^{(k)}\right) - f\left(x^{(k-1)}\right)} f\left(x^{(k)}\right) = \left(x^{(k)} - \xi\right) \frac{f\left[x^{(k-1)}, x^{(k)}\right] - f\left[x^{(k)}, \xi\right]}{f\left[x^{(k-1)}, x^{(k)}\right]}, \tag{5.60}$$

où l'on a noté, en employant la notation des *différences divisées* qui sera vue en **bac3**,

$$f[x,y] = \frac{f(x) - f(y)}{x - y}. \tag{5.61}$$

Par la relation de récurrence pour les différences divisées, la dernière égalité se réécrit alors

$$x^{(k+1)} - \xi = \left(x^{(k)} - \xi\right)\left(x^{(k-1)} - \xi\right) \frac{f\left[x^{(k-1)}, x^{(k)}, \xi\right]}{f\left[x^{(k-1)}, x^{(k)}\right]}. \tag{5.62}$$

Par application du théorème des accroissements finis, il existe $\zeta^{(k)}$, compris entre $x^{(k-1)}$ et $x^{(k)}$, et $\eta^{(k)}$, contenu dans le plus petit intervalle auquel appartiennent $x^{(k-1)}$,
$x^{(k)}$ et ξ, tels que

$$f\left[x^{(k-1)}, x^{(k)}\right] \text{ et } f\left[x^{(k-1)}, x^{(k)}, \xi\right] = \frac{1}{2} f''\left(\eta^{(k)}\right). \tag{5.63}$$

16. Notons que $\lim_{\delta\to 0} M(\delta) = \left|\frac{f''(s)}{2f'(\xi)}\right| < +\infty$, on peut donc bien satsfaire la condition (5.57) pour δ assez petit.

On en déduit alors que

$$x^{(k+1)} - \xi = \left(x^{(k)} - \xi\right)\left(x^{(k-1)} - \xi\right)\frac{f''\left(\eta^{(k)}\right)}{2f'\left(\zeta^{(k)}\right)}, \tag{5.64}$$

d'où

$$\left|x^{(k+1)} - \xi\right| \leq \delta^2 \left|\frac{f''\left(\eta^{(k)}\right)}{2f'\left(\zeta^{(k)}\right)}\right| \leq \delta(\delta M(\delta)) < \delta, \tag{5.65}$$

et $x^{(k+1)}$ appartient à I_δ. Par ailleurs, il est clair d'après la relation (5.55) que $x^{(k+1)}$ est différent de $x^{(k)}$, excepté si $f\left(x^{(k)}\right)$ est nulle.

En revenant à (5.64), il vient que

$$\left|x^{(k+1)} - \xi\right| \leq \delta M(\delta)\left|x^{(k)} - \xi\right|, \ \forall k \geq 0, \tag{5.66}$$

et donc

$$\left|x^{(k+1)} - \xi\right| \leq (\delta M(\delta))^{k+1}\left|x^{(0)} - \xi\right|, \ \forall k \geq 0, \tag{5.67}$$

ce qui permet de prouver que la méthode converge.

Il reste à vérifier que l'ordre de convergence de la méthode est au moins égal à $r = \frac{1}{2}(1+\sqrt{5})$. On remarque tout d'abord que r satisfait

$$r^2 = r + 1\cdot \tag{5.68}$$

On déduit ensuite de (5.64) que

$$\left|x^{(k+1)} - \xi\right| \leq M(\delta)\left|x^{(k)} - \xi\right|\left|x^{(k-1)} - \xi\right|, \ \forall k \geq 0. \tag{5.69}$$

En posant $E^{(k)} = M(\delta)|x^{(k)} - \xi|$, $\forall k \geq 0$, on obtient, après multiplication de l'inégalité ci-dessus par $M(\delta)$, la relation

$$E^{(k+1)} \leq E^{(k)}E^{(k-1)}, \ \forall k \geq 0. \tag{5.70}$$

Soit $E = \max\left(E^{(-1)}, E^{(0)^{\frac{1}{r}}}\right)$. On va établir par récurrence que

$$E^{(k)} \leq E^{r^{k+1}}, \ \forall k \geq 0. \tag{5.71}$$

Cette inégalité est en effet trivialement verifiée pour $k = 0$. En la posant vrai jusqu'au rang k, $k \geq 1$, elle est également vraie au rang $k-1$ et l'on a

$$E^{(k+1)} \leq E^{r^{k+1}}E^{r^k} = E^{r^k(r+1)} = E^{r^k r^2} = E^{r^{k+2}}. \tag{5.72}$$

Le résultat est donc valable pour tout entier positif k. En revenant à la définition de $E^{(k)}$, on obtient que

$$\left|x^{(k)} - \xi\right| \varepsilon^{(k)}, \text{ avec } \varepsilon^{(k)} = \frac{1}{M(\delta)}E^{r^{k+1}}, \ \forall k \geq 0, \tag{5.73}$$

avec $E < 1$ par hypothèses sur δ, $x^{(-1)}$ et $x^{(0)}$. Il reste à remarquer que

$$\frac{\varepsilon^{(k+1)}}{\varepsilon^{(k)^r}} = M(\delta)^{r-1}\frac{E^{r^{k+2}}}{E^{r^{k+1}r}} = M(\delta)^{r-1}, \ \forall k \geq 0 \tag{5.74}$$

et à utiliser la définition 5.3 pour conclure.

5.5 Méthodes pour les équations algébriques

Dans cette dernière section, nous considérons la résolution numérique d'équations algébriques, c'est-à-dire le cas pour lequel l'application f est un polynôme p_n de degré $n \geq 0$:

$$p_n(x) = \sum_{i=0}^{n} a_i x^i, \tag{5.75}$$

les coefficients $a_i, i = 0, \cdots, n$, étant des nombres réels données.

S'il est trivial de résoudre les équations algébriques du premier degré [17] et que la forme des solutions des équations de second dégré [18] est bien connue, il existe aussi des expressions analytiques pour les solutions des équations de degré trois et quatre, publiées par Cardano [19] en 1545 dans son *Artis Magnae, Sive de Regulis Algebraicis Liber Unus* (les formules étant respectivement dues à del Ferro [20] et Tartaglia [21] pour le troisième et à Ferrari [22] pour le quatrième). Par contre, le théorème d'Abel-Ruffini indique qu'il existe des polynômes de degré supérieur ou égal à cinq dont les racines ne s'expriment pas par radicaux. Le recours à une approche numérique se trouve par conséquent complètement motivé.

5.5.1 Évaluation des polynômes et de leurs dérivées

Nous allons à présent décrire la *méhode de Horner* [23], qui permet l'évaluation efficace d'un polynôme et de sa dérivée en un point donné. Celle-ci repose sur le fait que tout polynôme $p_n \in \mathbb{P}_n$ peut s'écrire sous la forme

$$p_n(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\cdots)). \tag{5.76}$$

Si les formes (5.75) et (5.76) sont algébriquement équivalentes, la première nécessite n additions et $2n-1$ multiplications alors que la séconde ne requiert que n additions et n multiplications.

L'algorithme pour évaluer le polynôme p_n en un point z se résume au calcul de n constantes b_i, $i = 0, \cdots, n$, définies de la manière suivante :

$$b_n = a_n\,, \tag{5.77}$$

$$b_i = a_i + b_{i+1} z,\ i = n-1, n-2, \cdots, 0, \tag{5.78}$$

avec $b_0 = p_n(z)$.

Application. Évaluons le polynôme $7x^4 + 5x^3 - 2x^2 + 8$ au point $z = 0,5$ par la méthode de Horner. On a : $b_4 = 7$, $b_3 = 5 + 7 * 0,5 = 8,5$, $b_2 = -2 + 8,5 * 0,5 = 2,25$, $b_1 = 0 + 2,25 * 0,5 = 1,125$ et $b_0 = 8 + 1,125 * 0,5 = 8,5625$, d'où la valeur $8,5625$.

Il est à noter qu'on peut organiser ces calculs successifs de cet algorithme dans un tableau ayant pour première ligne les coefficients $a_i, i = n, n-1, \cdots, 0$, du polynôme à évaluer, et comme seconde

17. Ce sont les équations du type $ax + b = 0$, avec $a \neq 0$, dont la solution est donné par $x = -\frac{b}{a}$.

18. Ce sont des équations de la forme $ax^2 + bx + c = 0$, avec $a \neq 0$, dont les solutions sont données par $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

19. Girolamo Cardano (24 septembre 1501-21 septembre 1576) était un mathématicien, médecin et astrologue italien. Ses travaux en algèbre, et plus précisément ses contributions à la résolution des équations algébriques du troisième degré, eurent pour conséquence l'émergence des nombres imaginaires.

20. Scipione del Ferro (6 février 1465-5 novembre 1526) était un mathématicien italien. Il est célèbre pour avoir été le premier à trouver la méthode de résolution des équations algébriques du troisième degré sans terme quadratique.

21. Niccolò Fontana Tartaglia (vers 1499- 13 décembre 1557) était un mathématiicien italien. Il fût l'un des premiers à utiliser les mathématiques en balistique, pour l'étude des trajectoires de boulets de canon.

22. Lodovico Ferrari (2 février 1522- 5 octobre 1565) était un mathématicien italien. Élève de Cardano, il est à l'origine de la méthode de résolution des équations algébriques du quatrième degré.

23. William George Horner (1786- 22 septembre 1837) était un mathématicien anglais. Il est connu pour sa méthode permettant l'approximation des racines d'un polynôme et pour l'invention en 1834 du *zootrope*, un appareil optique donnant l'illusion du mouvement.

ligne les coefficients $b_i,\ i = n, n-1, \cdots, 0$. Ainsi, chaque élément de la seconde ligne est obtenu en multipliant l'élément situé à sa gauche par z et en ajoutant au résultat l'élément situé au dessus.

Application. Pour l'exemple d'application précédent, on trouve le tableau suivant [24]

	7	5	-2	0	8
0	7	8,5	2,25	1,125	8,5625

Remarquons que les opérations employées par la méthode sont celles d'un procédé de *division synthétique*. En effet, si l'on réalise la division euclidienne de $p_n(x)$ par $(x - z)$, il vient

$$p_n(x) = (x - z)q_{n-1}(x) + r_0, \tag{5.79}$$

où le quotient $q_{n-1} \in \mathbb{P}_{n-1}$ est un polynôme dépendant de z par l'intermédiaire de ses coefficients, puisque

$$q_{n-1}(x) = \sum_{i=1}^{n} b_i x^{i-1}, \tag{5.80}$$

et le reste r_0 est une constante telle que $r_0 = b_0 = p_n(z)$. Ainsi, la méthode de Horner fournit un moyen simple d'effectuer très rapidement la division euclidienne d'un polynôme par un monôme de degré un.

Application. Effectuons la division euclidienne du polynôme $4x^3-7x^2+3x-5$ par $x-2$. En construisant un tableau comme précédemment, soit

	4	-7	3	-5
0	4	1	5	5

on obtient $4x^3 - 7x^2 + 3x - 5 = (x - 2)(4x^2 + x + 5) + 5$.

Appliquons de nouveau la méthode pour effectuer la division du quotient q_{n-1} par $(x-z)$. On trouve

$$q_{n-1}(x) = (x - z)q_{n-2}(x) + r_1, \tag{5.81}$$

avec $q_{n-2} \in \mathbb{P}_{n-2}$ et r_1 une constante, avec

$$q_{n-2}(x) = \sum_{i=2}^{n} c_i x^{i-2} \text{ et } r_1 = c_1, \tag{5.82}$$

les coefficients $c_i, i = 1, \cdots, n$, étant définis par

$$c_n = b_n, \tag{5.83}$$

$$c_i = b_i + c_{i+1}z,\ i = n-1, n-1, \cdots, 1. \tag{5.84}$$

On a par ailleurs

$$p_n(x) = (x - z)^2 q_{n-2}(x) + r_1(x - z) + r_0, \tag{5.85}$$

et, en dérivant cette dernière égalité, on trouve que $r_1 = c_1 = p'_n(z)$. On en déduit un procédé itératif permettant d'évaluer toutes les dérivées du polynôme p_n au point z. On arrive en effet à

$$p_n(x) = r_n(x - z)^n + \cdots + r_1(x - z) + r_0, \tag{5.86}$$

24. Dans ce tableau, on a ajouté une première colonne contenant 0 à la deuxième ligne afin de pouvoir réaliser la même opération pour obtenir tous les coefficients $b_i, i = 0, \cdots, n$, y compris b_n.

après $n+1$ itérations de la méthode que l'on peut résumer dans un tableau synthétique comme on l'a déjà fait

$$\begin{array}{c|cccccc} & a_n & a_{n-1} & \cdots & a_2 & a_1 & a_0 \\ \hline 0 & b_n & b_{n-1} & \cdots & b_2 & b_1 & r_0 \\ 0 & c_n & c_{n-1} & \cdots & c_2 & r_1 & \\ . & . & . & \cdots & r_2 & & \\ \vdots & \vdots & \vdots & \iddots & & & \\ . & . & r_{n-1} & & & & \\ 0 & r_n & & & & & \end{array} \tag{5.87}$$

dans lequel tous les élément n'appartenant pas à la première ligne (contenant les seuls coefficients connus initialement) ou à la première colonne sont obtenus en multipliant l'élément situé à gauche par z et en ajoutant le résultat de cette opération à l'élément situé au-dessus. Par dérivations successives de (5.86), on montre que

$$r_j = \frac{1}{j!} p_n^{(j)}(z),\ j = 0, \cdots, n, \tag{5.88}$$

où $p_n^{(j)}$ désigne la $j^{ième}$ dérivée du polynôme p_n.

Le calcul de l'ensemble du tableau (5.87) demande $\frac{1}{2}(n^2+n)$ additions et autant de multiplications.

5.5.2 Méthode de Newton-Horner

Compte tenu des remarques de la section précédente, on voit que la méthode de Newton peut judicieusement être adaptée pour la recherche des racines d'un polynôme, en exploitant la méthode de Horner étant donné que pour calculer le quotient apparaissant dans (5.53), c'est-à-dire

$$z^{(k+1)} = z^{(k)} - \frac{p\left(z^{(k)}\right)}{p'\left(z^{(k)}\right)}, \tag{5.89}$$

on a seulement besoin des deux premières colonnes du tableau synthétique. En effet, si q_{n-1} est le polynôme associé à p_n, il vient en dérivant par rapport à x

$$p'_n(x) = q_{n-1}(x; z) + (x - z) q'_{n-1}(x; z), \tag{5.90}$$

d'où $p'_n(z) = q_{n-1}(z; z)$. Grâce à cette identité, la méthode de Newton-Horner pour l'approximation d'une racine r_j prend la forme suivante : étant donné une estimation initiale $r_j^{(0)}$ de la racine, calculer

$$r_j^{(k+1)} = r_j^{(k)} - \frac{p_n\left(r_j^{(k)}\right)}{p'_n\left(r_j^{(k)}\right)} = r_j^{(k)} - \frac{p_n\left(r_j^{(k)}\right)}{q_{n-1}\left(r_j^{(k)}; r_j^{(k)}\right)},\ k \geq 0. \tag{5.91}$$

Pour un polynôme de degré n, le coût de chaque itération de l'algorithme est égal à $4n$. Si la racine est complexe, il est nécessaire de travailler en arithmétique complexe et de prendre la donnée initiale dans $\mathbb{C}$.

Annexes

Annexe A

Rappels d'algèbre linéaire

A.1 Définitions

Définition A.1. *Soit* $\mathbf{A} \in \mathbb{C}^{m\times n}$*. La matrice* $\mathbf{B} = \mathbf{A}^* \in \mathbb{C}^{m\times n}$ *est appelée* adjointe *(ou* transposée conjuguée*) de* $\mathbf{A}$ *si* $b_{ij} = \bar{a}_{ji}$*, où* $\bar{a}_{ji}$ *est le complexe conjugué de* a_{ji}*.*

On a : $(\mathbf{A}+\mathbf{B})^* = \mathbf{A}^* + \mathbf{B}^*$, $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$ et $(\alpha\mathbf{A})^* = \bar{\alpha}\mathbf{A}^*$ $\forall\alpha \in \mathbb{C}$*.*

Définition A.2. *Une matrice* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *est dite* symétrique *si* $\mathbf{A} = \mathbf{A}^T$*, et* antisymétrique *si* $\mathbf{A} = -\mathbf{A}^T$*. Elle est dite* orthogonale *si* $\mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T = \mathbf{I}$*, c'est-à-dire si* $\mathbf{A}^{-1} = \mathbf{A}^T$*.*

Les matrices de permutation sont orthogonales et le produit de matrices orthogonales est orthogonal.

Définition A.3. *Une matrice* $\mathbf{A} \in \mathbb{C}^{n\times n}$ *est dite* hermitienne *ou* autoadjointe *si* $\mathbf{A}^T = \bar{\mathbf{A}}$*, c'est-à-dire si* $\mathbf{A}^* = \mathbf{A}$*, et elle est dite* unitaire *si* $\mathbf{A}^*\mathbf{A} = \mathbf{A}\mathbf{A}^* = \mathbf{I}$*. Enfin, si* $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A}$*,* $\mathbf{A}$ *est dite* normale*.*

Par conséquent, une matrice unitaire est telle que $\mathbf{A}^{-1} = \mathbf{A}^*$*. Naturellement, une matrice unitaire est également normale, mais elle n'est en général pas hermitienne.*

On notera enfin que les coefficients diagonaux d'une matrice hermitienne sont nécessairement réels.

A.2 Trace et déterminant d'une matrice

Considérons une matrice carrée $\mathbf{A}$ d'ordre n. La **trace** de cette matrice est la somme des coefficients diagonaux de $\mathbf{A}$:

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}. \tag{A.1}$$

Dans le cas des matrices carrées d'ordre n, on a les propriétés suivantes pour le déterminant :

$$\det(\mathbf{A}) = \det(\mathbf{A}^T), \quad \det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B}), \tag{A.2}$$

$$\det\left(\mathbf{A}^{-1}\right) = \frac{1}{\det(\mathbf{A})} \tag{A.3}$$

$$\det(\mathbf{A}^*) = \overline{\det(\mathbf{A})}, \quad \det(\alpha\mathbf{A}) = \alpha^n\det(\mathbf{A}), \ \forall\alpha \in K \tag{A.4}$$

avec $K = \mathbb{R}$ ou $K = \mathbb{C}$.

De plus, si deux lignes ou deux colonnes d'une matrice coïncident, le déterminant de cette matrice est nul. Quand on échange deux lignes (ou deux colonnes), on change le signe du déterminant. Enfin, le déterminant d'une matrice diagonale est le produit des éléments diagonaux.

A.3 Matrices semblables

Il est utile, aussi bien pour des raisons théoriques que pour les calculs, d'établir une relation entre les matrices possédant les mêmes valeurs propres. Ceci nous amène à introduire la notion de *matrices semblables.*

Définition A.4. *Soient* $\mathbf{A}$ *et* $\mathbf{C}$ *deux matrices carrées de même ordre,* $\mathbf{C}$ *étant supposée inversible. On dit que les matrices* $\mathbf{A}$ *et* $\mathbf{C}^{-1}\mathbf{AC}$ *sont* semblables*, et on appelle* similitude *la transformation qui à* $\mathbf{A}$ *associe* $\mathbf{C}^{-1}\mathbf{AC}$*. De plus, on dit que* $\mathbf{A}$ *et* $\mathbf{C}^{-1}\mathbf{AC}$ *sont* unitairement semblables *si* $\mathbf{C}$ *est unitaire.*

Deux matrices semblables possèdent le même spectre et le même polynôme caractéristique. En effet, il est facile de vérifier que si $(\lambda, \mathbf{x})$ est un couple de valeur propre et vecteur propre de $\mathbf{A}$, il en est de même de $(\lambda, \mathbf{C}^{-1}\mathbf{x})$ pour la matrice $\mathbf{C}^{-1}\mathbf{AC}$ puisque

$$\left(\mathbf{C}^{-1}\mathbf{AC}\right)\mathbf{C}^{-1}\mathbf{x} = \mathbf{C}^{-1}\mathbf{Ax} = \lambda\mathbf{C}^{-1}\mathbf{x}. \tag{A.5}$$

A.4 Matrices définies positives

Définition A.5. *Une matrice* $\mathbf{A} \in \mathbb{C}^{n\times n}$ *est* définie positive *sur* $\mathbb{C}^n$ *si* $(\mathbf{Ax}, \mathbf{x})$ *est un nombre réel strictement positif* $\forall \mathbf{x} \in \mathbb{C}^n\,,\ \mathbf{x} \neq \mathbf{0}$*. Une matrice* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *est* définie positive *sur* $\mathbb{R}^n$ *si* $(\mathbf{Ax}, \mathbf{x})$ *est un nombre réel strictement positif* $\forall \mathbf{x} \in \mathbb{R}^n\,,\ \mathbf{x} \neq \mathbf{0}$*. Si l'inégalité stricte est remplacée par une inégalité au sens large* ($\geq$)*, la matrice est dite* semi-définie positive*.*

Exemple : Les matrices définies positives sur $\mathbb{R}^n$ ne sont pas nécessairement symétriques. C'est le cas par exemple des matrices de la forme

$$\mathbf{A} = \begin{bmatrix} 2 & \alpha \\ -2-\alpha & 2 \end{bmatrix} \tag{A.6}$$

avec $\alpha \neq -1$. En effet, pour tout vecteur non nul $\mathbf{x} = (x_1, x_2)^T$ de $\mathbb{R}^2$,

$$(\mathbf{Ax}, \mathbf{x}) = 2(x_1^2 + x_2^2 - x_1x_2) > 0. \tag{A.7}$$

Remarquer que $\mathbf{A}$ n'est pas définie positive sur $\mathbb{C}^2$. En effet, en prenant un vecteur complexe $\mathbf{x}$, le nombre $(\mathbf{Ax}, \mathbf{x})$ n'est en général pas réel.

Définition A.6. *Soit* $\mathbf{A} \in \mathbb{R}^{n\times n}$*. Les matrices*

$$\mathbf{A}_S = \frac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right), \quad \mathbf{A}_{SS} = \frac{1}{2}\left(\mathbf{A} - \mathbf{A}^T\right)$$

sont appelées respectivement partie symétrique *et* partie antisymétrique *de* $\mathbf{A}$*. Evidemment,* $\mathbf{A} = \mathbf{A}_S + \mathbf{A}_{SS}$*. Si* $\mathbf{A} \in \mathbb{C}^{n\times n}$*, on modifie les définitions comme suit :*

$$\mathbf{A}_S = \frac{1}{2}(\mathbf{A} + \mathbf{A}^*) \quad et \quad \mathbf{A}_{SS} = \frac{1}{2}(\mathbf{A} - \mathbf{A}^*).$$

On a la propriété suivante :

Propriété A.1. *Une matrice réelle* $\mathbf{A}$ *d'ordre* n *est définie positive si et seulement si sa partie symétrique* $\mathbf{A}_S$ *est définie positive.*

Notons pour terminer cet annexe que les matrices qui sont définies positives sur $\mathbb{C}^n$ satisfont la propriété caractéristique suivante :

Propriété A.2. *Une matrice carrée* $\mathbf{A}$ *d'ordre n est définie positive sur* $\mathbb{C}^n$ *si et seulement si elle est hermitienne et a des valeurs propres réelles. Ainsi, une matrice définie positive est inversible.*

Dans le cas de matrices réelles définies positives sur $\mathbb{R}^n$, des résultats plus spécifiques que ceux énoncés jusqu'à présent sont seulement valables quand la matrice est aussi symétrique (c'est la raison pour laquelle beaucoup d'ouvrages ne traitent que des matrices symétriques définies positives).

A.5 Produits scalaires vectoriels et normes matricielles

Définition A.7. *Un produit scalaire sur un K-espace vectoriel V est une application (.,.) de* $V \times V$ *sur K qui possède les propriétés suivantes :*

1. *elle est linéaire par rapport aux vecteurs de V, c'est-à-dire*

$$(\gamma\mathbf{x} + \lambda\mathbf{z}, \mathbf{y}) = \gamma(\mathbf{x}, \mathbf{y}) + \lambda(\mathbf{z}, \mathbf{y}), \ \forall \mathbf{x}, \mathbf{z} \in V, \ \forall \gamma, \lambda \in K;$$

2. *elle est hermitienne, c'est-à-dire* $(\mathbf{y}, \mathbf{x}) = \overline{(\mathbf{x}, \mathbf{y})}, \ \forall \mathbf{x}, \mathbf{y} \in V$ *;*
3. *elle est définie positive, c'est-à-dire* $(\mathbf{x}, \mathbf{x}) > 0, \ \forall \mathbf{x} \neq \mathbf{0}$ *(autrement-dit* $(\mathbf{x}, \mathbf{x}) \geq 0$ *et* $(\mathbf{x}, \mathbf{x}) = 0$ *si et seulement si* $\mathbf{x} = \mathbf{0}$*).*

Dans le cas où $V = \mathbb{C}^n$ (ou $\mathbb{R}^n$), un exemple est donné par le produit scalaire euclidien classique

$$(\mathbf{x}, \mathbf{y}) = \mathbf{y}^*\mathbf{x} = \sum_{i=1}^{n} x_i \overline{y_i}.$$

Pour toute matrice carrée $\mathbf{A}$ d'ordre n et pour tout $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$, on a alors la relation suivante :

$$(\mathbf{Ax}, \mathbf{y}) = (\mathbf{x}, \mathbf{A}^*\mathbf{y}). \tag{A.8}$$

En particulier, puisque pour toute matrice $\mathbf{Q} \in \mathbb{C}^{n\times n}$, $(\mathbf{Qx}, \mathbf{Qy}) = (\mathbf{x}, \mathbf{Q}^*\mathbf{Qy})$, on a :

Propriété A.3. *Les matrices unitaires préservent le produit scalaire euclidien. En d'autres termes, pour toute matrice unitaire* $\mathbf{Q}$ *et pour tous vecteurs* $\mathbf{x}$ *et* $\mathbf{y}$*, on a* $(\mathbf{Qx}, \mathbf{Qy}) = (\mathbf{x}, \mathbf{y})$*.*

Définition A.8. *Soit V un espace vectoriel sur K. On dit qu'une application* $||\cdot||$ *de V dans* $\mathbb{R}$ *est une norme sur V si :*

1. *(i)* $||\mathbf{v}|| \geq 0 \ \forall \mathbf{v} \in V$ et (ii) $||\mathbf{v}|| = 0$ *si et seulement si* $\mathbf{v} = \mathbf{0}$ *;*
2. $||\alpha\mathbf{v}|| = |\alpha|||\mathbf{v}|| \quad \forall \alpha \in K, \ \forall \mathbf{v} \in V$ *(propriété d'homogénéité) ;*
3. $||\mathbf{v} + \mathbf{w}|| \leq ||\mathbf{v}|| + ||\mathbf{w}|| \quad \forall \mathbf{v}, \ \mathbf{w} \in V$ *(inégalité triangulaire),*

   où $|\alpha|$ *désigne la valeur absolue (resp. le module) de* α *si* $K = \mathbb{R}$ *(resp.* $K = \mathbb{C}$*).*

On définit la **p-norme** (ou **norme de Hölder**) par

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \qquad \text{pour } 1 \leq p \leq \infty, \tag{A.9}$$

où les x_i sont les composantes du vecteur $\mathbf{x}$.

Remarquer que la limite de $||\mathbf{x}||_p$ quand p tend vers l'infini existe, est finie et égale au maximum des modules des composantes de $\mathbf{x}$. Cette limite définit à son tour une norme, appelée **norme infinie** (ou **norme du maximum**), donnée par

$$||\mathbf{x}||_\infty = \max_{1\leq i\leq n} |x_i|\cdot \tag{A.10}$$

Quand on prend $p = 2$ dans (A.9), on retrouve la définition classique de la **norme euclidienne**

$$||\mathbf{x}||_2 = (\mathbf{x},\mathbf{x})^{1/2} = \left(\sum_{i=1}^{n} |x_i|^2\right)^{1/2} = \left(\mathbf{x}^T\mathbf{x}\right)^{1/2}\cdot \tag{A.11}$$

Définition A.9 (Norme matricielle)**.** *Une* norme matricielle *est une application* $||\cdot|| : \mathbb{R}^{m\times n} \to \mathbb{R}$ *telle que :*

1. $||\mathbf{A}|| \geq 0\ \forall \mathbf{A} \in \mathbb{R}^{m\times n}$ et $||\mathbf{A}|| = 0$ *si et seulement si* $\mathbf{A} = \mathbf{0}$ *;*
2. $||\alpha\mathbf{A}|| = |\alpha|||\mathbf{A}||\quad \forall\alpha \in \mathbb{R},\ \forall \mathbf{A} \in \mathbb{R}^{m\times n}$ *(propriété d'homogénéité) ;*
3. $||\mathbf{A}+\mathbf{B}|| \leq ||\mathbf{A}|| + ||\mathbf{B}||\quad \forall \mathbf{A}\,,\ \mathbf{B} \in \mathbb{R}^{m\times n}$ *(inégalité triangulaire).*

Annexe B

Quelques programmes Python pour illustrer certains algorithmes

B.1 Programme Python pour générer le tableau 5.3

```
from mpmath import*
mp.dps = 300
zero=mpf(0)
two=mpf(2)
fact1=mpf(2.0)
fact2=mpf(3.0)
fact3=mpf(1.0)
TOL=mpf(1e-10)
def fonction_f(x):
    valeur=x**3+fact1*x**2-fact2*x-fact3
    return valeur
a0=mpf(1.0)
b0=mpf(2.0)
a = a0
b = b0
x0=(a0+b0)/two
val_f=fonction_f(x0)
f_sortie=open('fichier_7.dat','w')
i = 0
f_sortie.write("  %i   %.18f   %.18f   %.18f   %.18f\n"%(i,a,b,x0,val_f))
imax=1000
for i in range(1,imax+1):
    if(fonction_f(a0)*fonction_f(x0)<zero):
        a=a0
        b=x0
    elif(fonction_f(x0)*fonction_f(b0)<zero):
        a=x0
        b=b0
    x=(a+b)/two
    if abs(x-x0)<TOL:
        break
    val_f=fonction_f(x)
    f_sortie.write("  %i   %.18f   %.18f   %.18f   %.18f\n"%(i,a,b,x,val_f))
    a0=a
    b0=b
    x0=x
f_sortie.close()
```

B.2 Programme Python pour générer les données du tableau 5.2

```
import mpmath
mpmath.mp.dps =  18
zero=mpmath.mpf(0)
two=mpmath.mpf(2)
quinze=mpmath.mpf(15)
septante=mpmath.mpf(70)
sixtrois=mpmath.mpf(63)
fact=mpmath.mpf(0.125)
xvrai=mpmath.mpf(0.9061798459)
TOL=mpmath.mpf(0.0000000001)
def fonction_f(x):
     valeur=fact*x*(sixtrois*x**4-septante*x**2+quinze)
     return valeur
a0=mpmath.mpf(0.6)
b0=mpmath.mpf(1)
x0=(a0+b0)/two
val_f=fonction_f(x0)
i=0
diff=abs(x0-xvrai)
f_sortie=open('fichier_6.dat','w')
f_sortie.write('   %i  %.18f\n'%(i,diff))
imax=100
for i in range(1,imax+1):
     if(fonction_f(a0)*fonction_f(x0)<zero):
          a=a0
          b=x0
     elif(fonction_f(x0)*fonction_f(b0)<zero):
         a=x0
         b=b0
     x=(a+b)/two
     diff=abs(x-xvrai)
     f_sortie.write('   %i  %.18f\n'%(i,diff))
     if abs(x-x0)<TOL:
        break
     a0=a
     b0=b
     x0=x
f_sortie.close()
```

B.3 Recherche du zéro par la méthode de la fausse position (tableau 5.5)

```
from mpmath import*
mp.dps = 300
zero=mpf(0)
two=mpf(2)
fact1=mpf(2.0)
fact2=mpf(3.0)
fact3=mpf(1.0)
TOL=mpf(1e-10)
def fonction_f(x):
    valeur=x**3+fact1*x**2-fact2*x-fact3
    return valeur
a0=mpf(1.0)
b0=mpf(2.0)
a = a0
b = b0
valf_a=fonction_f(a)
valf_b=fonction_f(b)
x0=(valf_a*b-valf_b*a)/(valf_a-valf_b)
val_f=fonction_f(x0)
f_sortie=open('fichier_regula.dat','w')
i = 0
f_sortie.write("  %i   %.18f   %.18f   %.18f   %.18f\n"%(i,a,b,x0,val_f))
imax=1000
for i in range(1,imax+1):
    if(fonction_f(a0)*fonction_f(x0)<zero):
        a=a0
        b=x0
    elif(fonction_f(x0)*fonction_f(b0)<zero):
        a=x0
        b=b0
    valf_a=fonction_f(a)
    valf_b=fonction_f(b)
    x=(valf_a*b-valf_b*a)/(valf_a-valf_b)
    if abs(x-x0)<TOL:
        break
    val_f=fonction_f(x)
    f_sortie.write("  %i   %.18f   %.18f   %.18f   %.18f\n"%(i,a,b,x,val_f))
    a0=a
    b0=b
    x0=x
f_sortie.close()
```

B.4 Programme Python pour le calcul du nombre de conditionnement

```
import numpy as np
from mpmath import hilbert
import mpmath
mpmath.mp.dps = 64
# Cette fonction calcule la matrice B_k en utilisant le processus
#de Faddev et return la matrie B_k ainsi que les coefficeients alpha_k
def alpha_bk_matrix(A):
    mA_row = A.rows
    mA_col = A.cols
    B = mpmath.eye(mA_row)
    I = mpmath.eye(mA_row)
    for k in range(1,mA_row):
        C = A*B
        tr=mpmath.mpf(0.0)
        for i in range(mA_row):
            tr += C[i,i]
        alpha_k = (mpmath.mpf(1.)/k)*tr
        B = -A*B + alpha_k*I
    C = A*B
    tr=mpmath.mpf(0.0)
    for i in range(mA_row):
        tr += C[i,i]
    alpha_k = (mpmath.mpf(1.)/mA_row)*tr
    return alpha_k,B
def inv_matrix(A,alpha_k):
    mat = A/alpha_k
    return mat

if __name__=="__main__":
    print("\n\n%4s\t%10.6s\t%10.6s\t%10.6s\t"%('n','cond1','cond2','cond_e'))
    print("%s"%"="*len("          %6s\t%10.1s\t%10.1s\t%10.1s\t"%('n','cond1',
    'cond2','cond_e')))
    for i in range(2,7):
        h = hilbert(i)
        alpha,C = alpha_bk_matrix(h)
        inv_h = inv_matrix(C,alpha)
        inv_h_bis = h**-1
        cond1 = mpmath.mnorm(h,1)*mpmath.mnorm(inv_h_bis,1)
        norm_e = mpmath.mnorm(h,p='f')* mpmath.mnorm(inv_h_bis,p='f')
        eig_vec_h = mpmath.eig(h*h)[0] #Eigen vectors of h
        eig_vec_h1 = mpmath.eig(inv_h_bis*inv_h_bis)[0] #Eigen vectors of inverse h
        eig_vec_hAbs = mpmath.zeros(1,i)
        eig_vec_h1Abs = mpmath.zeros(1,i)
        norm_h = mpmath.norm(h*h,2)
        norm_h1 = mpmath.norm(inv_h_bis*inv_h_bis,2)
        cond2 = mpmath.sqrt(norm_h)*mpmath.sqrt(norm_h1)
        print("%4i\t%10.1f\t%10.1f\t%10.1f\t"%(i,cond1,cond2,norm_e))
```

Le tableau B.1 contient les résulats de ce programme. Nous constatons que ces derniers coïncident avec ceux figurant dans le tableau de la page 16 des notes de cours pour la **1-norme** et la norme euclidienne. Pour la **2-norme**, la coïncidence est bonne seulement pour n=2 et n=3. A partir de n=4, les résultats divergent. Nous pensons que nos résultats sont meilleurs que ceux de la page susmentionnée car ceux-ci sont en contradiction avec le fait que la matrice de Hilbert est mal conditionnée. Signalons que nous avons travaillé avec une précision de 64 chiffres.

n	cond1	cond2	cond_e
2	27.0	19.3	19.3
3	748.0	524.1	526.2
4	28375.0	15514.4	15613.8
5	943656.0	476644.6	480849.1
6	29070279.0	14952935.9	15118987.1

FIGURE B.1 – Résultats du programme python pour le calcul du nombre de conditionnement

Annexe C

Quelques exercices concernant les TPE

1. Les éléments d'une matrice réelle symétrique **S** sont définis comme suit :

$$S[\ell,k] = \begin{cases} 0 \text{ si } (\ell\text{+k}) \text{ est impaire} \\ \sqrt{\frac{(\ell+1)(\ell+2)}{(k+1)(k+2)}} \quad \text{si } (\ell\text{+k}) \text{ est paire et } k \geq \ell \geq 0 \end{cases}$$

   Ecrire une fonction **Python** pour calculer les éléments de la matrice **S**. On précise que **S** est une matrice carrée de taille ns où ns est un nombre entier naturel. **A.N** : $ns = 20$.

2. Ecrire un programme **Python** pour faire la **décomposition de Cholsky** de la matrice **S** (Pages 44-45 des notes de cours). **A.N.** : $ns = 10$. Il importe de préciser que ***les formules de la page 37 doivent être modifiées compte tenu du fait qu'en Python, les indices des éléments de matrice commencent par 0 au lieu de 1***.

3. Ecrire un programme **Python** pour calculer l'inverse de la matrice carrée **S** à l'aide des **formules de Faddev** ou **Leverrier** (Page 42 des notes de cours). **A.N.** : $ns = 10$.

4. Obtenir la dernière ligne du tableau de la page 17 des notes de cours au moyen d'un program **Python**. Vous devrez vous servir des **formules de Faddev** ou **Leverrier** (Page 42 des notes de cours) pour calculer l'inverse d'une matrice carrée.

5. Ecrire un program **Python** servant à calculer les données nécessaires pour reproduire la figure 5.2 (page 52 des notes de cours).

   Ecrire un programme **gnuplot** pour reproduire la figure en question à partir des résultats fournis par votre programme **Python**.

6. Ecrire un programme **Python** pour approcher, au moyen de la **méthode de dichotomie**, la racine du polynôme $f(x) = x^3 + 0.75\,x^2 + 2.5\,x - 25$ contenue dans l'intervalle $[2,\ 3]$ avec une précision de 10^{-16}. Les résultats devront être présentés dans un tableau ayant la forme de celle du tableau de la page 53 des notes de cours. Comparer le nombre d'itérations nécessaires pour calculer la racine en question à celui obtenu à partir de l'expression (5.11) (page 51 des notes de cours).

7. Ecrire un programme **Python** pour approcher, au moyen de la **méthode regula falsi**, la racine du polynôme $f(x) = x^3 + 0.75\,x^2 + 2.5\,x - 25$ contenue dans l'intervalle $[2,\ 3]$ avec une précision de 10^{-16}. Les résultats devront être présentés dans un tableau ayant la forme de celle du tableau de la page 51 des notes de cours. Comparer le nombre d'itérations nécessaires pour calculer la racine en question à celui obtenu à partir de l'expression (5.11) (page 51 des notes de cours).

8. Ecrire un programme **Python** pour approcher, au moyen de la **méthode de la corde** [formule (5.50)], la racine du polynôme $f(x) = x^3 + 0.75\,x^2 + 2.5\,x - 25$ contenue dans l'intervalle $[2,\ 3]$

avec une précision de 10^{-16}. Les résultats devront être présentés dans un tableau ayant la forme de celle du tableau de la page 53 des notes de cours. Comparer le nombre d'itérations nécessaires pour calculer la racine en question à celui obtenu à partir de l'expression (5.11) (page 51 des notes de cours). On précise que $x^{(0)} = 2.05$.

9. Ecrire un programme **Python** pour approcher, au moyen de la **méthode de Newton-Raphson**, la racine du polynôme $f(x) = x^3 + 0.75\,x^2 + 2.5\,x - 25$ contenue dans l'intervalle $[2,\ 3]$ avec une précision de 10^{-16}. Les résultats devront être présentés dans un tableau ayant la forme de celle du tableau de la page 53 des notes de cours. Comparer le nombre d'itérations nécessaires pour calculer la racine en question à celui obtenu à partir de l'expression (5.11) (page 51 des notes de cours). On précise que $x^{(0)} = 2.05$.

10. Ecrire un program **Python** pour reproduire le tableau de la page 53 des notes de cours.

11. On considère la fonction f définie par :

$$f(x) = \mathrm{e}^x - 2x - 1\cdot$$

(a) Montrer que l'équation $f(x) = 0$ admet une unique racine α sur l'intervalle $[1,\ 2]$.
(b) Montrer que cette équation équivaut à $x = g(x)$ avec $g(x) = \ln(2x + 1)$.
(c) Montrer que la méthode du point fixe (des approximations successives)

$$\begin{cases} x_{n+1} = g(x_n) \qquad n = 0,\ 1,\ 2,\ \cdots \\ x_0 = 1.2 \end{cases}$$

converge vers la racine de l'équation. Il faut établir un tableau à quatre colonnes. La $1^{\text{ère}}$ colonne contient les valeurs de n, la seconde les valeurs de x_n, la $3^{\text{ème}}$ les valeurs de x_{n+1} et la dernière les valeurs de la différence $x_{n+1} - x_n$. Une précision de 10 chiffres au moins après la virgule est exigée. Le tableau en question doit être généré au moyen d'un programme informatique écrit en python.

12. On considère l'intégrale $S = \int_a^b f(x)\mathrm{d}x$. On se donne une subdivision

$a = x_0,\ x_1,\ x_2,\ \cdots,\ x_n = b$ de l'intervalle $[a,\ b]$ et on pose $x_j = a + jh$ et $h = (b - a)/n$.
Dans la méthode des rectangles, on remplace la fonction f par une fonction constante par morceaux. Soit g la fonction définie par

$$g(x) = f(x_j) \quad \text{si } x \in [x_j,\ x_{j+1}]\cdot$$

On montre que si l'on pose $S = \int_a^b g(x)\mathrm{d}x$, on a

$$S = h\sum_{j=0}^{n-1} f(a + jh)\cdot$$

Calculer (à l'aide d'un programme informatique écrit en python) par la méthode des rectangles (formule ci-dessus) l'intégrale $I = \int_0^1 (x^2+1)\mathrm{d}x$ pour $n = 10$ (un dessin est nécessaire), et évaluer l'erreur commise.

Bibliographie

[Alain2009] Alain Yger, Jacques-Arthur Weil et al., **Mathématiques L3 appliquées**. Cours avec 500 tests et exercices corrigés, (Pearson Education, Pans, 2009).

[Alfio2004] Alfio Quarteroni, Riccardo Sacco and Fausto Saleri , **Méthodes Numériques. Algorithmes, analyse et applications** (Springer-Verlag Italia, Milano 2004).

[Fibonacci] Nils Berglund (2005) , **Récréations mathématiques. La suite de Fibonacci** (Université du Sud Toulon–Var, novembre 2005),
`http://www.univ-orleans.fr/mapmo/membres/berglund/fibonacci.pdf`

[Franck2005] Franck Jedrzejewski , **Introductions aux méthodes numériques,** 2^e **éd.** (Springer-Verlag France, Paris 2005).

[Gilles1997] Gilles Dowek , **Le langage mathématique et les langages de programmation**, Colloque *Voir, entendre, raisonner, calculer* (Cité des sciences et de l'industrie, La Villette, Paris, 1997).
`https://www.lsv.fr/~dowek/Philo/langagelangage.pdf`

[Guillaume2009] Guillaume Legendre, **Méthodes numériques. Introduction à l'analyse numérique et au calcul scientifique**. Cours de Deuxième année de licence de Mathématiques et Informatique appliquées à l'Economie et à l'Entreprise (MI2E) à l'université de Paris-Dauphine, année académique 2009-2010.
`http://users.metu.edu.tr/baver/Cours.NUM.pdf`

[Jean-Marc2002] Jean-Marc Huré et Didier Pelat, **Méthodes numériques. Elements d'un premier parcours**. Cours destiné aux étudiants de DEA Astrophysique & Méthodes associées aux université Paris 7 et 11, année académique 2002-2003
`https://media4.obspm.fr/public/M2R/supports/CoursMN.pdf`

[Laurent2015] Laurent Signac, **Introduction à l'algorithmique et à la programmation avec Python**, `https://deptinfo-ensip.univ-poitiers.fr`