Algorithmique numérique

Nicolas Sicard - EFREI - Avril 2006

Avant-propos

C'est au calcul scientifique et ses besoins que l'on doit la naissance des ordinateurs. Aujourd'hui, les ordinateurs et les méthodes numériques sont de plus en plus employés dans le domaine de la physique, des mathématiques et de façon générale dans les sciences de l'ingénieur.

L'analyse numérique consiste à étudier le problème scientifique d'un point de vue mathématique, puis numérique. L'algorithmique numérique en est la phase opérationnelle, consistant à écrire les algorithmes puis les programmes capables de résoudre le problème par les ordinateurs.

Le champ d'application de l'analyse numérique (donc de l'algorithmique numérique) est très vaste. Il s'agit très souvent de calculs scientifiques, présents dans de nombreux domaines de l'industrie : **simulation de phénomènes réels** (physiques, chimiques, biologiques, démographiques...) à des fins de **prévisions** (météorologie, démographie...), de **conception** (*crash test*, aéronautique, pharmaceutique, nucléaire...) ou même de vérifications (mathématiques).

Le problème revient donc le plus souvent à calculer l'état d'un système à un certain point de son évolution (le plus souvent temporelle ou spatiale) en fonction d'un état initial et de paramètres de transformation.

Dans ce cours, nous présenterons un certain nombre de méthodes de résolution connues de problèmes d'analyse numérique et à leur traduction algorithmique. Nous verrons également les contraintes de l'informaticien (ou du numéricien), liées à la simplification du modèle mathématique, à la discrétisation des données et à leur représentation et calcul en machine.

Dans l'introduction, nous préciserons la notion de modélisation et les paramètres du choix d'une solution arithmétique/algorithmique pour un problème donné. Puis (partie II), nous rappellerons brièvement les notions d'algorithmique de l'ordinateur : la représentation des nombres et les notions d'erreur en calcul numérique. La partie III traite des systèmes linéaires et des calculs sur matrices. Nous nous intéresserons ensuite (partie IV) à la représentation des courbes (interpolations, approximations) dont le champ d'application est très vaste (de l'analyse démographique à la synthèse 3D). En partie V, nous présenterons les méthodes classiques de dérivation et d'intégration de fonctions. La dernière partie propose une courte bibliographie des ouvrages et documents à partir desquels ce cours est construit.

NB : merci de signaler toute erreur...

I - Introduction

Le calcul scientifique et les applications de l'analyse numérique en général, recouvrent plusieurs disciplines correspondant aux différentes étapes de résolution du problème.

Modélisation

La première étape consiste à **concevoir une représentation d'un objet et des phénomènes physiques qui s'y déroulent**. Par exemple, on veut représenter un avion, l'air et leurs interactions mutuelles, ou bien un réacteur nucléaire, son combustible et les réactions qui s'y produisent... Il y a alors **plusieurs échelles de description possibles**, au niveau de la représentation des détails. Ainsi en météorologie, on utilise un maillage tri-dimensionnel de l'atmosphère dans lequel chaque point est séparé de plusieurs kilomètres. Au sein d'une molécule, l'industrie parmaceutique travaille au niveau du micron... Pour un même problème, plus la précision est grande, plus la quantité de données et de calculs est grande.

Par ailleurs, un même phénomène peut être décrit de façon différente : pour évaluer une foule, on peut compter les individus un par un ou bien évaluer sa surface et la multiplier par une densité moyenne. Dans le premier cas la précision est maximale (on parle de **modélisation exacte**) mais le temps de calcul très long. Dans l'autre cas la précision est plus faible mais le temps de réponse du système de simulation sera plus rapide (**modélisation approchée**). C'est le rôle du spécialiste dans le domaine concerné (physicien, chimiste, démographe etc.) de **définir le modèle de représentation et sa précision**.

Méthode numérique

Après l'établissement d'un modèle mathématique à partir d'observations ou de propriétés physiques, le rôle du mathématicien (ou numéricien) est de le transformer pour le rendre "compatible" avec les ordinateurs. Il propose alors une méthode de résolution numérique du problème, c'est-à-dire **un moyen de trouver la solution par une suite de calculs qu'un ordinateur pourra effectuer**.

La résolution du problème revient en général à déterminer la valeur d'une ou plusieurs inconnues dans cette description du modèle. Comme le modèle, une méthode numérique peut être exacte, par exemple issue d'une formule donnant la solution et d'une arithmétique exacte. Ainsi l'équation $x+y+z=180°$ permet de connaître la valeur exacte (en degrés) d'un des angles x d'un triangle lorsqu'on connaît les valeurs des deux autres (y et z). De même, le calcul de la valeur d'un polynôme en un point donné relève d'un calcul exact.

Elle peut être également approchée, à l'issue d'un calcul numérique dans lequel interviennent des valeurs approchées (réels), des erreurs d'arrondis *etc.*. ou lorsqu'on cherche une valeur limite par un processus itératif. Les mathématiques ne peuvent pas proposer des solutions (ou modèles) exactes pour tous les problèmes. On cherchera alors une représentation sous la forme de valeurs approchées. Par exemple, certaines fonctions n'ont pas de primitives connues en mathématiques classiques, ou bien certaines équations n'ont pas de solutions déterminées... On utilise alors une méthode numérique qui permet d'obtenir une valeur approchée de la solution.

Informatique

Le rôle de l'informaticien numéricien est de proposer un ensemble programme/architecture qui permette de réaliser les calculs nécessaires à la résolution numérique du problème en un temps le plus court possible avec la précision la plus grande possible. Cela recouvre :

1. *Le choix de la représentation des données* : selon le problème on utilisera des nombres scalaires, ou bien des vecteurs, des matrices... On peut aussi utiliser des structures de données plus élaborées, comme des graphes (mailles, arbres...),

2. *Le choix du couple algorithmique/arithmétique* : il dépend naturellement de la nature du modèle (système linéaire, équations différentielles, séries...), mais aussi de sa précision (degré d'approximation introduite dans la description mathématique), de sa complexité (logarithmique, linéaire, exponentielle), de ses conditions d'application (temps disponible pour le calcul, puissance des ordinateurs, parallélisme *etc.*), ou encore de la précision du résultat exigée.

3. *Les phases de test et de représentation des solutions* : pour tester un algorithme, la méthode la plus simple consiste à vérifier qu'il fournit de "bonnes solutions" pour des problèmes dont on connaît déjà la solution, ou bien de le confronter à des expériences réelles (aérodynamiques, chimiques, biologiques, maquettes...). Il ne faut pas négliger la représentation des solutions car elle donne lieu à l'interprétation des résultats. Elle peut être sous forme numérique (par exemple la proportion d'un métal dans un alliage), sous forme graphique (la courbure d'une aile d'avion), vectorielle (une direction dans l'espace) *etc.*

Le **choix de l'arithmétique** est très important car il intervient dans la stabilité d'un algorithme, c'est-à-dire sa capacité à confiner les erreurs à des ordres de grandeur suffisamment faibles et à ne pas diverger. Il intervient aussi dans sa précision. Par exemple, il n'est pas raisonnable de simuler l'écoulement de l'air autour d'un fuselage d'avion en ne manipulant que des entiers codés sur 16 bits. À l'inverse, le calcul d'un angle de triangle à partir de l'équation $x+y+z=180°$ ne nécessite probablement pas l'emploi de réels codés sur 96 bits (très grande précision). Nous verrons plus en détails les problèmes d'algorithmique des machines et la notion d'erreur en partie II. De façon générale, on utilise des **arithmétiques à précision finie**, à **précision finie extensible** (multi-précision) ou bien encore à **précision infinie** (algorithmes symboliques). Les premières limitent la quantité de nombres représentables, par exemple par un nombre restreint de *bits (avec 8 bits on peut représenter $2^8=256$ valeurs)*. Les secondes consistent à adapter la précision en fonction des besoins lors du calcul. De cette façon, on contribue à une plus grande stabilité des algorithmes. Les dernières n'imposent pas de limite théorique à la précision des nombres manipulés. Cela se traduit par des algorithmes très coûteux en terme d'occupation mémoire et de ressources de calculs.

On distingue deux grands types d'algorithmes : **les algorithmes discrets** et les algorithmes itératifs. Les premiers découlent le plus souvent d'une formule symbolique qui permet de calculer la valeur exacte du résultat par un nombre fini et prévisible d'étapes (avec une algorithmique adaptée...). Le pivot de Gauss en est un bon exemple. Les **algorithmes itératifs** correspondent, eux, au calcul de limites de suites convergentes, par exemple pour calculer le zéro d'une fonction, ou bien pour obtenir le résultat de systèmes d'équations différentielles. On pourra citer les méthodes de Jacobi et Gauss-Seidel pour la résolution de systèmes linéaires.

Lorsqu'on veut résoudre numériquement un problème, on doit donc se poser les questions suivantes :

- Le modèle mathématique choisi est-il exact ou approché ?
- Quel couple algorithme / algorithmique est le plus approprié pour ce modèle ?
- L'algorithme converge-t-il dans ces conditions ? Quelle est alors sa complexité, sa vitesse de convergence ?
- Quelle est la précision du calcul et quelle est la valeur de l'erreur dans le cas d'un résultat approché ?

Dans la suite de ce cours, nous nous intéresserons essentiellement aux étapes 2 et 3. Nous présenterons les principaux modèles mathématiques utilisés en analyse numérique puis les méthodes de calcul et les algorithmes qui en découlent. Quelques définitions et propriétés mathématiques nécessaires à la compréhension de ces modèles sont disponibles en annexe de ce document.

II - Algorithmique d'ordinateur, notions d'erreurs

L'exemple qui suit est un cas d'école représentatif des approximations par erreurs d'arrondis produites par un ordinateur. Soit la formule suivante :

$(10^N+1)-10^N$ où N est un entier positif ou nul.

Le résultat exact de ce calcul simple dans les réels est 1 et pourtant la plupart des ordinateurs (et autres calculatrices) donneront la valeur 1 lorsque $N < 16$ mais 0 pour $N \geq 16$. Au delà d'une certaine valeur de N, l'ordinateur n'est plus en mesure de "coder de façon différente" les valeurs 10^N et $(10^N + 1)$. La différence des deux est donc nulle. Le problème est comparable lorsqu'on manipule de très petites valeurs : la précision est insuffisante.

1/ Représentation et codage des nombres entiers (rappel)

On rappelle que dans les ordinateurs, les nombres sont représentés sous forme binaire. Une donnée élémentaire d'information (un *bit*) peut avoir deux états, notés 0 et 1. Ces états correspondent à l'absence (0) ou à la présence (1) d'une charge électrique dans un composant. Un nombre, représenté sous forme binaire, est donc codé par un groupe de bits : 8, 16, 32, 64, 128, voire même 256 ou 512... Dans le cas des entiers la représentation et les calculs sont effectués en base 2. Ainsi, le nombre binaire 10110010 (sur 8 bits) vaut :

$$0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = (178)_{10}$$

Les nombres négatifs sont le plus souvent codés en "complément à deux" (ou "complément vrai"), consistant en pratique à inverser tous les bits du nombre et à lui ajouter 1 pour passer d'une valeur positive à la même valeur négative. Cette représentation a pour avantage de préserver les techniques de calcul pour les opérations de base (addition *etc.*).

Ainsi la machine est capable d'effectuer des calculs tout à fait exacts. Mais elle dispose de ressources limitées qui ne permettent pas de manipuler des très grands nombres. Pour des programmes classiques, des entiers codés sur 32 bits suffisent (de -2147483647 à +2147483647). Pour des applications numériques scientifiques, cette limite est atteinte très vite. On utilise alors des entiers codés sur 64 bits (pris en charge nativement par des processeurs 64 bits) ou plus grâce à des bibliothèques de programmation pour la manipulation de grands nombres. La cryptographie est une application typique des très grands entiers. On se reportera utilement au cours d'architecture pour plus d'informations sur le codage des entiers en machine et sur les notions de débordement (*overflow*).

2/ Représentation et codage des nombres réels en virgule flottante

Cas du codage en virgule fixe

La représentation en virgule fixe est en réalité une extension de la représentation des entiers pour la partie décimale du nombre. La partie entière (à gauche de la virgule) est codé comme un entier. La partie décimale (à droite de la virgule) est représentée par les coefficients d'une somme pondérée des puissances négatives de 2, par analogie avec le système décimal : la partie décimale d'un nombre réel est représentée par les coefficients d'une somme pondérée des puissances négatives de 10 :

$$(1,25)_{10} = 1x10^0 + 2x10^{-1} + 5x10^{-2} = 1x2^0 + 0x2^{-1} + 1x2^{-2} = (1,01)_2$$

Dans les faits cette représentation est un peu rigide car le nombre de bits réservés aux parties entière et décimale doit être fixé à l'avance. On lui préfère souvent le codage en virgule flottante, également plus répandu et normalisé.

Comprendre le codage en virgule flottante [1]

Au final, dans un ordinateur classique, toute information est codée sous forme de bits. Mais pour faciliter la compréhension de ce codage, nous partirons de la représentation décimale des nombres réels.

Soit *a* un nombre réel non nul. On sait qu'il peut s'écrire sous la forme (notation scientifique) :

$$a = \pm 10^q \times 0, a_1 a_2 a_3 ... \quad (a_1 \neq 0)$$

ou encore :

$$a = \pm 10^q \sum_{i=1}^{\infty} a_i 10^{-i} \quad (a_1 \neq 0)$$

où les a_i sont des chiffres décimaux et *q* un entier relatif appelé *exposant*.

Par exemple, le nombre réel 12,089 peut s'écrire $0{,}12089 \times 10^2$ et 0,000000256 s'écrire $0{,}256 \times 10^{-6}$.

Le nombre réel :

$$m = \pm \sum_{i=1}^{\infty} a_i 10^{-i}$$

est appelé *mantisse* de *a*.

L'exposant entier q est lui aussi codé par un nombre limité de chiffres. Cela donne, en base 10 :

$$q = \pm \sum_{i=1}^{S} b_i 10^{i-1}$$

Ainsi q ne peut varier qu'entre deux valeurs limites (cf. codage des entiers) : $-N \leq q \leq M$ où N et M sont des entiers. Lorsque $q > M$ on parle de dépassement et quand $q < -N$ on parle de "sous passement" de capacité. Selon les normes et les ordinateurs, un dépassement ou un sous-passement peut provoquer l'arrêt du programme et l'émission d'un message d'erreur. Ou bien la valeur NaN (Not a Number) lors d'un dépassement ou bien 0 lors d'un sous passement peuvent être utilisés pour continuer les calculs.

Dans l'expression de la mantisse, la somme comporte un nombre potentiellement infini de termes, ce qui n'est pas réaliste dans le cadre d'une utilisation informatique où le nombre de chiffres significatifs est fini. On appelle donc représentation en virgule flottante à *t* chiffres significatifs, et on note *fl*(*a*), le nombre réel :

$$fl(a) = \pm 10^q \sum_{i=1}^{t} d_i 10^{-i} \quad (d_1 \neq 0)$$

On utilise aussi le terme de *nombre-machine*.

Pour passer du nombre réel a à sa représentation en virgule flottante $fl(a)$ il faut donc déterminer les valeurs des d_i en fonction des a_i. La première méthode, la plus simple, consiste à tronquer a, c'est à dire à ne conserver que les t premiers chiffres de la mantisse m : $d_i = a_i$ pour i=1,2,...t. Mais l'erreur de troncature peut être importante si a_{i+1} est grand (proche de 9).

Il est donc préférable d'utiliser un arrondi de la mantisse en choisissant le nombre entier $d_1d_2...d_t$ le plus proche du nombre $a_1a_2...a_t,a_{t+1}$, c'est à dire $E(a_1a_2...a_t,a_{t+1}+0,5)$[1]. Pourtant un problème apparaît lorsque a_{t+1} vaut 5, car l'arrondi est toujours à l'entier supérieur. On introduit ce qu'on appelle un *biais*, qui peut être compensé par un arrondi à l'entier supérieur une fois sur deux, et une troncature une fois sur deux (par exemple en utilisant la parité de a_t).

Limites de la représentation [7]

Le nombre fini de chiffres significatifs autorisés dans la représentation des réels en machine implique que seul un sous-ensemble de l'ensemble des réels peut être représenté pour une précision donnée.

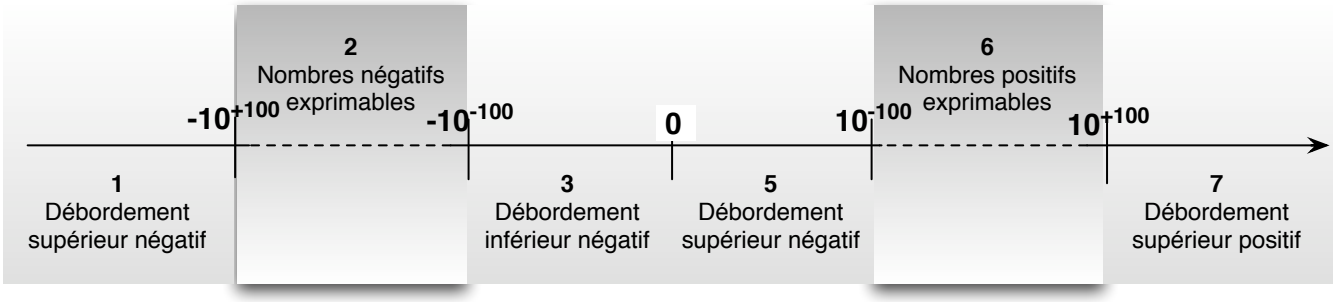
> ***En réalité, un ordinateur ne travaille pas sur l'ensemble R mais bien sur un sous-ensemble de l'ensemble Q des nombres rationnels.***

Voyons comment se présente ce sous-ensemble dans le cas d'une mantisse signée à 3 chiffres et un exposant à 2 chiffres (en base 10). On peut distinguer 7 zones :

1. les nombres négatifs inférieurs à $\mathbf{-0{,}999 \times 10^{99}}$,
2. les nombres négatifs compris entre $\mathbf{-0{,}999 \times 10^{99}}$ et $\mathbf{-0{,}100 \times 10^{-99}}$,
3. les "petits" nombres négatifs, de valeur absolue inférieure à $\mathbf{-0{,}100 \times 10^{-99}}$,
4. **0** (zéro),
5. les "petits" nombres positifs, de valeur absolue inférieure à $\mathbf{-0{,}100 \times 10^{-99}}$,
6. les nombres positifs compris entre $\mathbf{-0{,}100 \times 10^{-99}}$ et $\mathbf{0{,}999 \times 10^{99}}$,
7. les nombres positifs supérieurs à $\mathbf{0{,}999 \times 10^{99}}$

Comme le montre la figure suivante, les réels des catégories 1, 3, 5 et 7 ne sont pas représentables. Lorsqu'un calcul produit un résultat dans les intervalles 1 et 7, on parle de débordement supérieur (négatif ou positif). Lorsque le résultat appartient à la catégorie 3 ou 5, on parle de débordement inférieur (positif ou négatif), qui peut se traduire souvent par un arrondi à 0. Les nombres représentables se trouvent dans les régions 2 et 6.

[1] $E(x)$ = partie entière de x.



L'amplitude des régions 2 et 6 (*i.e.* la différence entre le plus grand et le plus petit nombre représentable de l'intervalle) dépend de l'exposant. Plus il est grand, plus l'amplitude est grande.

En revanche, leur densité (*i.e.* le nombre de réels représentables et différentiables dans ces régions) est directement liée au nombre de chiffres significatifs de la mantisse. Cette densité correspond au degré de précision de la représentation. Pour un même exposant, plus le nombre de chiffres de la mantisse est grand, plus la précision est grande... Si le nombre de chiffres pouvait être infini, cette précision serait infinie. C'est la cas des nombres réels qui forment ce qu'on appelle un *continuum* (entre deux nombres réels x et y, quels qu'ils soient, il existe toujours un nombre réel).

Normalisation

Dans ce qui précède, nous avons passé sous silence un aspect particulier de cette représentation : sans indication particulière, il existe plusieurs représentations possibles pour un même nombre. Par exemple, $0{,}1200 \times 10^{-5}$ et $0{,}0120 \times 10^{-4}$ sont deux représentations possibles du même nombre 0,0000012 avec une mantisse à 4 chiffres. Mais la seconde représentation pose un problème de précision lors de certains calculs. Imaginons par exemple que l'on veuille élever ce nombre au carré, c'est-à-dire le multiplier par lui même. Le calcul consiste à faire $0{,}1200 \times 0{,}1200 \times 10^{-5-5} = 0{,}0144 \times 10^{-10}$ pour la première représentation. Cela ne pose pas de problème car la mantisse dispose de 4 chiffres. Si on utilise la seconde représentation, on obtient $0{,}0120 \times 0{,}0120 \times 10^{-4-4} = 0{,}0001 \times 10^{-8}$. Comme la mantisse ne contient que 4 chiffres, les deux derniers chiffres significatifs du résultat (4 et 4) ont été tronqués ou arrondis. La perte d'information est importante.

Afin d'assurer une précision maximum des calculs, mais aussi une représentation unique des nombres, on a tout intérêt à décaler les chiffres de la mantisse à gauche de façon à ce que le premier chiffre (à gauche) ne soit pas nul. Une telle mantisse est dite ***normalisée***.

Les nombres normalisés sont les plus utilisés en calcul numériques car ils assurent la meilleure stabilité des calculs, comme on vient de le voir. Néanmoins, en cas de débordement inférieur, il est courant d'utiliser les nombres dénormalisés. Ces derniers ont un exposant le plus petit possible (-99 sur deux chiffres) et le premier chiffre de la mantisse est égal à 0. Ils permettent un dépassement graduel vers le bas pour des opérations produisant des résultats inférieurs au plus petit nombre normalisé, plutôt que de le remplacer simplement par un 0.

Et en base 2 ?

Cette représentation des nombres réels en base 10 peut être transposée dans n'importe quelle base, en particulier en base 2. C'est l'objet de la norme IEEE 754. Les nombres à virgule flottante sont représentés sous la forme d'un signe S, d'une mantisse M, d'un exposant E et d'une base b. S, M et E sont des nombres entiers codés en binaire, b vaut le plus souvent 2 (parfois 8 ou 16).

On distingue trois niveaux de précisions :

	Signe	Exposant	Mantisse	Total
Simple précision	*1 bit*	*8 bits*	*23 bits*	*32 bits*
Double précision	*1 bit*	*11 bits*	*52 bits*	*64 bits*
Grande précision	*1 bit*	*15 bits*	*64 bits*	*80 bits*

La valeur d'un nombre réel codé est alors :

- $(-1)^S * (1+M) * 2^{(E-127)}$ en simple précision,
- $(-1)^S * (1+M) * 2^{(E-1023)}$ en double précision.

Ici, plutôt que de coder le signe de l'exposant, on retire systématiquement la valeur maximale que ce dernier pourrait prendre s'il était codé en entier signé (par exemple 127 pour 8 bits). Ainsi on ne stocke que des nombres positifs ou nuls. Par ailleurs, le premier bit de la mantisse, qui vaut toujours 1 dans le cas des nombres normalisés, est tout bonnement implicite (comme sa valeur ne change pas, il est inutile de le représenter en mémoire). Ceci explique l'ajout de la valeur 1 à celle de la mantisse dans l'expression de la valeur réelle représentée.

Extension de la norme

La norme IEEE 754 prévoit également un codage à précision étendue (simple ou double). Pour manipuler des nombres de précision plus importante, on peut utiliser des bibliothèques de programmation adaptées, telles que MPFR (*GNU MP*).

Les calculs en virgule flottante sont très utilisés car ils autorisent une certaine souplesse et la normalisation permet une plus grande interoperabilité ou portabilité des codes entre différentes plates-formes. Néanmoins, ils sont fortement sujets à l'apparition et à la propagation d'erreurs d'approximation (voir l'exemple du début de chapitre), dont nous allons parler dans la suite.

3/ Les erreurs numériques [1]

En calcul numérique, il faut être capable de mesurer l'erreur produite par l'utilisation de la combinaison algorithme / algorithmique choisie. Cela permet d'évaluer la précision et, *in fine*, la pertinence des résultats obtenus. Cette remarque n'a rien d'anodin, comme le prouvent les deux exemples, désormais célèbres, de l'explosion du premier tir de la fusée Ariane 5 et des cibles ratées par les missiles patriotes. On pourra utilement se reporter au rapport de la commission d'enquête pour le cas Ariane [4], et à [5] pour le cas des missiles. Par ailleurs, une étude technique intéressante des erreurs ou inexactitudes des calculs sur des réels peut être consultée en [6] (en anglais).

On peut distinguer deux catégories d'erreurs : les erreurs directement dues à la méthode numérique et les erreurs dues à l'utilisation de l'ordinateur. Les premières proviennent directement des méthodes approchées utilisées par l'analyse numérique. Il faut essayer de déterminer à quel point leurs résultats peuvent s'écarter de la solution exacte, c'est à dire étudier l'*erreur d'approximation*. Les erreurs dues à l'utilisation des ordinateurs sont liées à la représentation des nombres en machine, en particulier par le nombre fini de chiffres significatifs pour coder les nombres. On ajoutera dans ce chapitre la nécessité pour l'algorithmique d'étudier la complexité de son algorithme, c'est à dire le nombre d'opérations

élémentaires nécessaires que nécessite l'obtention de la solution. Cela est important pour minimiser ce nombre d'opérations, prévoir le temps de calcul sur machine sur des problèmes de dimensions plus grandes, mais aussi parfois anticiper les propagations d'erreurs au fil des opérations.

Pour résumer, on peut dire que l'erreur de méthode décroît avec l'augmentation du degré de précision. Mais l'augmentation de cette précision entraîne une augmentation du nombre de calculs, responsable d'une amplification des erreurs liées à l'utilisation des machines. Pour un problème et une précision donnée, l'erreur totale est la somme des deux. Les rôles du numéricien et de l'informaticien sont intimement liés pour déterminer un équilibre entre précision et temps de calcul.

Intéressons-nous plus précisément à la notion d'erreur d'arithmétique, liée à l'utilisation des ordinateurs.

Erreurs d'affectation

L'erreur d'affectation correspond à l'erreur commise lors du codage d'un nombre réel dans l'ordinateur (mémoire ou registre ou unité de calculs). En base 10, elle est mesurée par la différence, en valeur absolue, entre le nombre réel *a* et sa représentation en machine *fl*(*a*) :

$$|a - fl(a)| \leq 5|a|p10^{-t} \tag{2.1}$$

où $p = 1$ lorsqu'on utilise l'arrondi et $p = 2$ lorsqu'on utilise la troncature.

Dans le cas de la troncature, les mantisses de a et fl(a) sont identiques, donc :

$$\begin{aligned}
|a - fl(a)| &= 10^q \times (0, a_1a_2...a_{t+1}a_{t+2}... - 0, a_1a_2...a_t) \\
&= 10^q \times 0, \underbrace{0...0}_{t} a_{t+1}a_{t+2}... \\
&= 10^{q-t} \times 0, a_{t+1}a_{t+2}... \\
&= 10^{q-t} \times 0, a_{t+1}a_{t+2}... \frac{|a|}{|a|} \\
&= 10^{-t} \frac{0,a_{t+1}a_{t+2}...}{0,a_1a_2...}|a| \\
&\leq 10^{1-t}|a|
\end{aligned}$$

car $0,a_{t+1}a_{t+2}... \leq 1$ et $0,a_1a_2... \geq 0,1$.

En binaire, donc pour les ordinateurs réels, on montre que l'erreur d'affectation pour une mantisse de ***t*** bits est majorée par :

$$|a - fl(a)| \leq 2^{-t}|a| \tag{2.2}$$

Opérations arithmétiques élémentaires

Les erreurs apparaissent lors des calculs de l'ordinateur. Ces calculs sur les flottants sont effectués dans une mémoire spéciale appelée accumulateur, qui stocke les résultats le temps du calcul, avant transfert en mémoire centrale. Aujourd'hui, la plupart des ordinateurs disposent d'accumulateurs en double précision, au minimum. Cela signifie que la mantisse de l'accumulateur dispose de $2t$ chiffres (quand la mantisse normale en dispose de t). Voyons comment, pour l'opération arithmétique de base qu'est l'addition, comment évaluer l'erreur commise durant un calcul.

Soient x_1 et x_2 deux nombres réels (opérandes) de mantisses respectives m_1 et m_2. On suppose que $|x_1| \geq |x_2|$. On peut écrire :

$$\begin{aligned} x_1 &= 10^{q_1} m_1 \\ x_2 &= 10^{q_2} m_2 = 10^{q_1} m'_2 \end{aligned}$$

avec

$$m'_2 = 10^{q_2 - q_1} m_2$$

L'idée, ici, est de décaler les chiffres de la mantisse m_2 de telle façon à retrouver une expression avec un même exposant q_1 pour x_1 et x_2. Il suffit alors d'appliquer l'opération élémentaire sur les mantisses m_1 et m'_2. Deux cas se présentent :

- $q_1 - q_2 > t$, ce qui signifie que le premier chiffre significatif (en partant de la gauche) de la mantisse de x_2 est à plus de t chiffres de distance du premier chiffre significatif de la mantisse de x_1 (q_1 est l'exposant commun car le plus grand). On peut dès maintenant conclure que les chiffres significatifs de x_2 ne pourront pas apparaître dans la mantisse de t chiffres du résultat. Donc $fl(x_1+x_2)=x_1$. Par exemple, l'addition de 0,3218x10^6 et de 0,2411x10^1 donne (0,3218 + 0,000002411)x10^6, soit 0,3218x10^6 pour une mantisse de 4 chiffres.

- $q_1 - q_2 \leq t$, alors on peut calculer $m' = m_1 + m'_2$ dans l'accumulateur double précision ($2t$ chiffres). Il ne reste plus qu'à tronquer ou arrondir la mantisse m' à t chiffres ce qui donne $fl(x_1 + x_2)$.

Lors d'un tel calcul, l'erreur n'intervient qu'au moment où on tronque (ou on arrondi) la mantisse de $2t$ chiffres à t chiffres. Donc l'erreur sur l'opération ne peut dépasser l'erreur d'arrondi ou de troncature effectuée sur un nombre en virgule flottante, sous réserve qu'il n'y ait pas de dépassement (inférieur ou supérieur). En effectuant une analyse similaire pour les deux autres opérations on arrive à la même conclusion, d'après (2.1) :

$$\begin{aligned} |(x_1 \pm x_2) - fl(x_1 \pm x_2)| &\leq 5|(x_1 \pm x_2)|p10^{-t} \\ |(x_1 x_2) - fl(x_1 x_2)| &\leq 5|(x_1 x_2)|p10^{-t} \\ |(x_1 / x_2) - fl(x_1 / x_2)| &\leq 5|(x_1 / x_2)|p10^{-t} \end{aligned}$$

avec $p = 1$ dans le cas de l'arrondi et $p = 2$ dans le cas de la troncature.

Une autre conséquence notable, lorsque $q_1 - q_2 > t$, est la **non-associativité de l'addition** : selon l'ordre avec lequel la somme de plusieurs termes est effectuée, le résultat n'est pas toujours le même. Essayez, par exemple avec la somme 1 + 0,0004 + 0,0003 avec une mantisse limitée à 4 chiffres, en utilisant l'arrondi. Vous constaterez que le calcul d'une somme doit être fait par paquets de termes de même ordre de grandeur pour obtenir la meilleure précision.

De façon générale, certaines formulations de calculs seront meilleures que d'autres vis-à-vis de la stabilité numérique de l'algorithme. Ainsi l'opération $1/[x(x+1)]$ sera plus stable numériquement sous cette forme que sous la forme $1/x - 1/(x+1)$. Il faut toujours s'efforcer d'éliminer ces instabilités par voie algébrique, avant l'écriture de l'algorithme.

4/ Propagation des erreurs [1]

Un algorithme de calcul numérique ne se résume pas à une seule opération, mais bien souvent à de longues séquences de calculs. Comme on l'a vu précédemment, une seule opération peut mener à des erreurs potentiellement importantes. On imagine aisément le résultat de ce phénomène propagé sur plu-

sieurs centaines, plusieurs milliers, voire parfois plusieurs millions d'opérations successives du même type ! Nous montrerons le cas particulier du développement de l'exponentielle e^{-x}, mais voyons d'abord comment majorer de façon formelle l'erreur issue d'une simple séquence de produits.

Séquence de produits

D'après la formule (2.1), on sait encadrer l'erreur commise lors du produit de deux flottants x_1 et x_2. On a :

$$|x_1x_2 - fl(x_1x_2)| \leq 5|x_1x_2|p10^{-t}$$

où $p = 1$ dans le cas de l'arrondi et $p = 2$ dans le cas de la troncature.

Cela peut aussi s'écrire :

$$x_1x_2 - 5p(x_1x_2)10^{-t} \leq fl(x_1x_2) \leq x_1x_2 + 5p(x_1x_2)10^{-t}$$

ou encore plus directement :

$$fl(x_1x_2) = (x_1x_2)(1 + a10^{-t})$$

avec $0 \leq |a| \leq 5p$.

On souhaite calculer $fl(x_1x_2...x_n)$ en fonction du produit $x_1x_2...x_n$ par récurrence. Soit $p_1 = x_1$ et $p_i = fl(p_{i-1}x_i)$ pour $i = 2,...n$. D'après la formule précédente, on a donc $p_i = p_{i-1}x_i(1+a_i10^{-t})$, avec $0 \leq |a_i| \leq 5p$, puis

$$p_n = fl(x_1x_2 \dots x_n) = x_1x_2 \dots x_n(1 + a_210^{-t}) \dots (1 + a_n10^{-t})$$

Or, on peut dire qu'il existe un *a'* tel que :

$$(1 + a_210^{-t}) \dots (1 + a_n10^{-t}) \leq (1 + a'10^{-t})^{n-1}$$

avec $0 \leq |a'| \leq 5p$. D'où :

$$fl(x_1x_2 \dots x_n) = x_1x_2 \dots x_n(1 + a10^{-t})^{n-1}$$

avec $0 \leq |a| \leq 5$ dans le cas de l'arrondi et $0 \leq |a| \leq 10$ dans le cas de la troncature.

Pour résumer, on peut dire que l'erreur finale est égale au produit des erreurs commises à chaque produit élémentaire.

> ***Question :*** *évaluer le nombre minimum n de produits à partir duquel l'erreur commise lors du calcul d'une séquence risque de dépasser 1% de la valeur résultante. Peut-on dire que ce genre de calcul est instable ?*

Cas de e^x

Cet exemple montre que le résultat obtenu peut être parfois très éloigné du résultat escompté. Il consiste à évaluer la valeur de e^x (*exponentielle x*) à l'aide de son développement en série, pour des valeurs négatives de x :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots$$

On sait que ce développement converge pour tout x. Il fait intervenir une séquence de sommes, produits et division. C'est un candidat intéressant pour l'étude de la propagation de l'erreur.

Comme la sommation d'une infinité de termes n'est pas possible en pratique, on pourra s'arrêter lorsque le terme $x^k/k!$ devient plus petit que la précision minimum que l'on se fixe, représentée par la valeur ϵ. L'algorithme de calcul est le suivant :

```
REEL S, T, A
S = T = A = 1
FAIRE
     T = T * (X / A)
     S = S + T
     A = A + 1
TANT QUE ( |T| > ϵ )
```

Voici les résultats obtenus par un programme qui implante cet algorithme en langage C, en double précision. Comme $e^{-30} > 10^{-15}$ on choisit $\epsilon = -15$:

x	S	e^x	n	erreur
-10	4.539993e-05	4.539993e-05	52	≃ 0
-15	3.059094e-07	3.059023e-07	67	≃ 0
-20	5.621884e-09	2.061154e-09	82	≃ 60%
-25	-7.129780e-07	1.388794e-11	96	x100
-30	-3.066812e-05	9.357623e-14	119	$x10^9$

On voit que les résultats n'ont plus aucun sens pour $x \leq -20$. Cela s'explique par le fait que, pour x=-30, les termes de la série vont en croissant jusqu'à $x^{30}/30! \simeq 8x10^{11}$. Avec une mantisse de 52 bits ($2^{52} \simeq 4{,}5x10^{15}$), de telles valeurs ne peuvent avoir de précision supérieure à environ $2x10^{-4}$. Le résultat trouvé pour S ne représente en fait que l'accumulation des erreurs d'approximations sur les termes de plus grandes valeurs du développement.

Question : que proposez-vous pour améliorer les résultats pour $x < -15$?

5/ Conditionnement et stabilité numérique

On a vu que le mode de représentation non exact des nombres en machine entraîne deux types d'erreur: des erreurs d'affectation et des erreurs de calcul. Les erreurs d'affectation interviennent dès l'introduction des valeurs dans la machine. Par des variations même très petites des valeurs initiales, elles peuvent entraîner des variations importantes du résultat. On parle de **conditionnement d'un problème**.

On dit qu'un problème est bien conditionné si une petite variation des données initiales entraîne une petite variation sur les résultats. À l'inverse, un problème est dit mal conditionné lorsqu'une petite variation des valeurs initiales entraîne une grande variation sur les résultats.

Ce problème de conditionnement est lié à la méthode de résolution mathématique employée : pour une même arithmétique d'ordinateurs, différentes méthodes peuvent présenter des conditionnements différents. Nous verrons un exemple de problème de conditionnement dans le chapitre suivant.

La notion de **stabilité numérique** est davantage liée à l'arithmétique de l'ordinateur. Elle mesure le degré de propagation des erreurs lors d'une suite d'opérations. Cette stabilité dépend fortement de l'algorithme mis en œuvre pour le calcul d'une formule mathématique. En particulier, l'erreur commise peut varier beaucoup selon l'ordre des opérations : la non-associativité de l'opération d'addition est un bon exemple. À titre d'illustration, vous pouvez vous référer au problème de la résolution d'une équation du second degré, développé en TD.

III - Systèmes linéaires

Dans cette partie on s'intéresse à la résolution de systèmes de *n* équations algébriques à *n* inconnues. Il s'agit de la première partie, et de la partie la plus importante car elle est à la base de la résolution de très nombreux problèmes numériques. Une très grande majorité des problèmes peut se résumer, en fin de compte, à résoudre un système linéaire. Même les systèmes continus à base d'équations différentielles peuvent être approximés par des systèmes linéaires, la plupart du temps. Par ailleurs, les systèmes d'équations issus de problèmes physiques (nucléaire, biologiques etc.) sont souvent importants en terme de quantités de données et de calculs. Cela est lié à l'exigence de précision.

1/ Introduction

Les systèmes linéaires sont présents dans la plupart des problèmes d'ingénieurie, de la mécanique à l'écoulement des fluides en passant par les circuits électriques. Il se traduisent par un ensemble d'équations de type $Ax = b$ où b représente les données, connues, à l'entrée du système (forces mécaniques, courants électriques *etc.*), A représente les caractéristiques du système (par exemple des lois physiques connues...) et x représente la réponse (inconnue) du système. Pour un système donné, A est invariant mais b change selon l'expérience. Pour une matrice A donnée, il est donc utile de proposer des algorithmes qui nécessitent le moins de calculs possibles quelque soit le vecteur d'entrée b.

En algèbre, on apprend qu'un système linéaire peut être résolu par la méthode de Cramer. Mais cette méthode nécessite près de $n!n^2$ opérations élémentaires, soit 4×10^8 pour $n = 10$! Ce n'est pas une solution acceptable pour des systèmes de plusieurs dizaines ou centaines d'inconnues.

En pratique, il existe deux types de méthodes de résolution des systèmes linéaires : les **méthodes directes** ou les **méthodes itératives** (dites aussi indirectes). Dans les méthodes directes, on tente de transformer le système de façon à obtenir des équations plus facile, plus rapide à résoudre. Dans le cas itératif, on part d'une solution supposée que l'on affine petit à petit jusqu'à satisfaire un certain critère de convergence.

Problème mathématique

Pour résumer, le problème mathématique consiste à résoudre l'équation $Ax = b$ où A est une matrice carrée connue, $\boldsymbol{b}$ un vecteur connu et $\boldsymbol{x}$ un vecteur inconnu. En notation matricielle détaillée, cela s'écrit également :

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Le système peut être résolu par l'équation $x = A^{-1}b$ où A^{-1} désigne la matrice inverse de A, et accepte une solution unique, pourvu que le déterminant de A est différent de 0 ($|A| \neq 0$, A non singulière). Cela veut simplement dire qu'aucune ligne de A n'est combinaison linéaire d'un ou plusieurs autres lignes. Dans le cas contraire le système a une infinité de solutions ou bien n'en a aucune (selon le vecteur constant).

De l'importance du conditionnement

Il faut être vigilant lorsque le déterminant de ***A*** est non nul mais très petit : les calculs sur les réels introduisent des erreurs d'arrondis comparables à de légères variations des valeurs des coefficients d'origine de la matrice. Ceci peut conduire à des erreurs beaucoup plus grandes du type de celles qu'on obtient en appliquant les calculs sur une matrice singulière. On dit alors que le système est instable. Reste à définir le sens exact de "très petit" : très petit par rapport à quoi ?

En théorie, on compare le déterminant à la norme[2] de la matrice A, notée $||A||$. On dira que le système est "très petit" si $|A| << ||A||$. Pour le vérifier, on utilise ce qu'on appelle le nombre de conditionnement de la matrice A, noté *cond(A)*, et tel que $cond(A) = ||A||.||A^{-1}||$. Si ce nombre est proche de *1*, la matrice est bien conditionnée, le système est donc stable. Plus il s'accroît, plus le système est instable. Il tend vers l'infini pour une matrice singulière.

En pratique, le nombre de conditionnement est très long à calculer pour de grandes matrices, on préfère donc comparer le déterminant à l'ordre de grandeur moyen des coefficients.

Prenons par exemple le système suivant :

$$\begin{aligned} 2x + y &= 3 \\ 2x + 1,001y &= 0 \end{aligned}$$

dont la solution est x = 1501,5 et y = -3000. On remarque que $|A|$=2x1,001 - 2x1 = 0,002. qui est de trois ordres de grandeur inférieure aux coefficients de la matrice. Le système est donc mal conditionné. On peut le vérifier en remplaçant la deuxième équation par l'équation $2x + 1,002y = 0$. La solution du système est alors donnée par x = 751,5 et y = -1500. Une variation de 0,1% d'un des coefficients produit une variation de 100% sur le résultat !

2/ Méthodes de résolution directe

Ces méthodes permettent de résoudre un système linéaire en un nombre fini d'étapes. De façon générale, le principe consiste à rendre la matrice A triangulaire supérieure. On distingue généralement trois types de **méthodes directes** :

- Le **pivot de Gauss** (*Gauss elimination*) consiste à transformer l'équation initiale $Ax = b$ en une équation de type $Ux = c$ où U est une matrice triangulaire supérieure et ***c*** un vecteur constant (**phase d'élimination**). La dernière équation devient alors triviale (une seule inconnue), les autres équations sont résolues les unes après les autres de bas en haut durant la **phase de remontée**. On détaillera cette méthode dans la suite.

- La **décomposition *LR*** (ou ***LU***) qui transforme l'équation initiale en une équation de type $LUx = b$ où L est une matrice triangulaire inférieure et U une matrice triangulaire supérieure. On se ramène aux résolutions successives de deux équations $Ly = b$ puis $Ux = y$. En utilisant la décomposition de Doolittle, le problème est similaire au pivot de Gauss. La décomposition de Choleski (qui suppose une matrice A symétrique et définie positive) ne s'applique que dans des situations particulières.

---

[2] Il existe plusieurs normes possibles. La plus connue est sans doute la norme infinie :
$||A|| = max_{1 \leq i \leq n} \sum_{j=1}^{n} |A_{ij}|$

- L'**élimination de Gauss-Jordan** consiste à obtenir une équation de type $Ix = c$ à partir de l'équation initiale, où $\boldsymbol{I}$ est la matrice identité. Le résultat est alors directement obtenu dans c.

L'avantage des solutions directes, telles que le pivot de Gauss, est que le nombre d'opérations est fini et on peut facilement le borner. C'est important dans le cadre d'une exécution sur ordinateur car cela permet d'évaluer le temps de calcul de l'algorithme en fonction de la taille du système à priori. L'autre avantage est que l'augmentation de la précision de l'algorithmique de l'ordinateur permet en général d'obtenir des résultats plus précis. Une précision infinie mènerait à un résultat exact.

2.1/ Pivot de Gauss [2]

Phase d'élimination

Cette phase est en fait très simple sur le principe : elle consiste à combiner les équations deux par deux afin de faire disparaître progressivement les inconnues. Cette transformation du système n'affecte pas la solution puisqu'il s'agit de combinaisons linéaires. Ainsi, on effectue l'opération :

$$Eq.(i) \leftarrow Eq.(i) - \lambda \times Eq.(j)$$

L'équation soustraite ($Eq.(j)$) est appelée ***pivot***. Bien sûr, la valeur de $\boldsymbol{\lambda}$ est calculée de façon à annuler le coefficient d'une inconnue. Ainsi le système suivant,

$$\begin{array}{rcll} 4x_1 - 2x_2 + x_3 & = & 11 & (a) \\ -2x_1 + 4x_2 - 2x_3 & = & -16 & (b) \\ x_1 - 2x_2 + 4x_3 & = & 17 & (c) \end{array}$$

sur lequel on applique les deux opérations de combinaison successives

$$\begin{array}{rcll} Eq.(b) & \leftarrow & Eq.(b) - (-2/4) \times Eq.(a) & (1) \\ Eq.(c) & \leftarrow & Eq.(c) - (1/4) \times Eq.(a) & (2) \end{array}$$

donne le système (on a éliminé x_1 dans les équations (b) et (c)) :

$$\begin{array}{rcll} 4x_1 - 2x_2 + x_3 & = & 11 & (a) \\ 3x_2 - 1,5x_3 & = & -10,5 & (b) \\ -1,5x_2 + 3,75x_3 & = & 14,25 & (c) \end{array}$$

L'opération

$$Eq.(c) \leftarrow Eq.(c) - (-0,5) \times Eq.(b)$$

permet finalement d'obtenir le système :

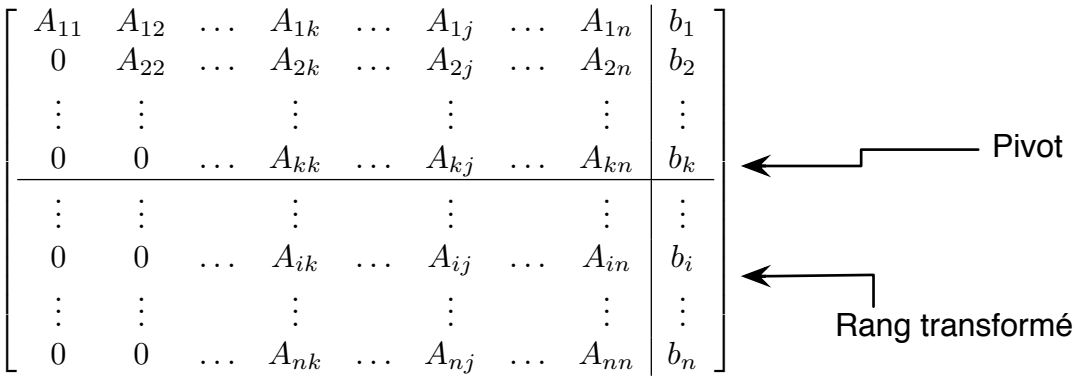
$$\begin{array}{rcll} 4x_1 - 2x_2 + x_3 & = & 11 & (a) \\ 3x_2 - 1,5x_3 & = & -10,5 & (b) \\ 3x_3 & = & 9 & (c) \end{array}$$

Phase de résolution (remontée)

Les inconnues peuvent alors être calculées par substitutions successives lors d'une phase de remontée : l'équation (c) donne x_3 = 9/3 = 3 puis (b) donne x_2 = -6/3 = -2 et x_1 = (11-4-3)/4 = 1.

Ecriture matricielle et algorithme

Revenons à la forme matricielle du problème pour exprimer les calculs. Considérons la matrice étendue constituée des coefficients des équations et des valeurs du vecteur b.



La phase d'élimination revient à faire apparaître des zéros dans la partie triangulaire inférieure de la matrice par combinaisons linéaires successives du pivot (au rang *k*, allant de 1 à *n*-1) avec toutes les équations situées "en-dessous" (aux rangs *i*, allant de *k*+1 à *n*). Le pivot au rang *k* permet d'annuler tous les $k_{ièmes}$ coefficients des équations *k*+1 à *n* en utilisant $\lambda = A_{ik} / A_{kk}$. L'algorithme correspondant peut donc s'écrire :

```
POUR k de 1 à n-1 FAIRE                 /* rang du pivot */
     POUR i de k+1 à n FAIRE            /* rang à transformer */
          lambda = A[i,k] / A[k,k]      /* calcul de lambda */
          POUR j de k à n FAIRE         /* pour tous les coefficients du rang */
               A[i,j] = A[i,j] - lambda * A[k,j]     /* combinaison linéaire */
               b[i] = b[i] - lambda * b[k]           /* pour le vecteur aussi */
          FINPOUR
     FINPOUR
FINPOUR
```

Question : quelles optimisations peut-on apporter à cet algorithme, par exemple pour éliminer des calculs inutiles ? Modifier l'algorithme en conséquence.

À la fin de la phase d'élimination la matrice se présente ainsi (attention, les valeurs des A_{ij} et b_i ont changé) :

$$[A|b] = \left[\begin{array}{ccccc|c} A_{11} & A_{12} & A_{13} & \dots & A_{1n} & b_1 \\ 0 & A_{22} & A_{23} & \dots & A_{2n} & b_2 \\ 0 & 0 & A_{33} & \dots & A_{3n} & b_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & A_{nn} & b_n \end{array}\right]$$

On obtient facilement x_n par le quotient b_n/A_{nn} obtenu à partir du rang *n*. Puis, d'après le rang *n*-1 on a :

$$A_{n-1n-1}x_{n-1} + A_{n-1n}x_n = b_{n-1}$$

d'où

$$x_{n-1} = \frac{b_{n-1} - A_{n-1n}x_n}{A_{n-1n-1}}$$

que l'on généralise facilement au rang *k* par :

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} A_{kj}x_j}{A_{kk}}$$

En supposant que l'on remplace les valeurs du vecteur *b* par celles du vecteur solution *x*, l'algorithme de la phase de remontée (substitution) peut donc s'écrire :

```
POUR k de n à 1 par pas de -1 FAIRE        /* au rang k */
     POUR j de k+1 à n FAIRE               /* substitution des xj connus... */
          b[k] = b[k] - A[k,j]*b[j]        /* ...que l'on soustrait à bk */
     FINPOUR
     b[k] = b[k] / A[k,k]                  /* quotient par Akk */
FINPOUR                                    /* b contient le résultat x */
```

Eléments de complexité

L'algorithme de la phase d'élimination présente trois boucles imbriquées dépendant de *n*, la taille de la matrice. La première boucle est répétée N_1=*n*-1 fois. À chaque itération *k*, la deuxième boucle est elle-même itérée *n-k* fois. Cela représente déjà N_2=(*n*-1)+(*n*-2)+...+2+1 itérations. À chacune de ces itérations, la troisième boucle est itérée *n* fois pour les (*n*-1) premières, (*n*-1) fois pour les (*n*-2) suivantes *etc.* jusqu'à 2 fois pour la dernière. Cela représente donc :

$$N_3 = \sum_{k=1}^{n-1} k(k+1) = \sum_{k=1}^{n-1} k^2 + \sum_{k=1}^{n-1} k = \frac{n(n-1)(2n-1)}{6} + \frac{(n-1)(n-2)}{2} \simeq \frac{n^3}{3} + \frac{n^2}{2}$$

itérations.

La phase de résolution (remontée) présente deux boucles imbriquées dépendant de *n*. La première est répétée *n* fois, la seconde *n-k* fois. On a donc :

$$N_4 = \sum_{k=1}^{n}(n-k) = \sum_{k=1}^{n} n - \sum_{k=1}^{n} k = n^2 - \frac{n(n-1)}{2} = \frac{n(n+1)}{2} \simeq \frac{n^2}{2}$$

D'où la complexité globale N_{pg} de l'algorithme du pivot de Gauss :

$$N_{pg} \simeq \frac{n^3}{3} + n^2$$

Il s'agit d'un polynôme d'ordre 3, donc la complexité du pivot de Gauss est de type $O(n^3)$.

De l'importance du choix du pivot (méthode avec pivot partiel)

Considérons le système linéaire suivant, sous forme de matrice augmentée :

$$[A|b] = \left[\begin{array}{ccc|c} 0 & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{array}\right]$$

La solution de ce système est $x_1 = x_2 = x_3 = 1$, mais l'application de l'algorithme de Gauss ne peut pas fonctionner car $A_{11} = 0$ ce qui entraîne une division par 0.

De même, si le système se présente sous la forme

$$[A|b] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{array}\right]$$

alors l'application de la première phase de l'élimination de Gauss va donner :

$$[A'|b'] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ 0 & 2-1/\varepsilon & -1+1/\varepsilon & 0 \\ 0 & -1+2/\varepsilon & -2/\varepsilon & 1 \end{array}\right]$$

Lorsque ϵ tend vers 0 (*ie.* est très petit), $1/\epsilon$ tend vers l'infini (*ie.* une valeur très grande) et $2\text{-}1/\epsilon$ tend vers $-1/\epsilon$ (cf. II-3/). Ce système est donc équivalent au système suivant s'il est calculé par un ordinateur :

$$[A'|b'] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ 0 & -1/\varepsilon & 1/\varepsilon & 0 \\ 0 & 2/\varepsilon & -2/\varepsilon & 1 \end{array}\right]$$

Or, ce système n'est pas solvable car la deuxième et la troisième équations sont clairement contradictoires. On a donc tout intérêt à faire en sorte que les pivots (les A_{kk} dans la figure ci-dessous) soient les plus grands possibles.

$$\left[\begin{array}{ccccccccc|c} A_{11} & A_{12} & \dots & A_{1k} & \dots & A_{1j} & \dots & A_{1n} & b_1 \\ 0 & A_{22} & \dots & A_{2k} & \dots & A_{2j} & \dots & A_{2n} & b_2 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & A_{kk} & \dots & A_{kj} & \dots & A_{kn} & b_k \\ \hline \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & A_{ik} & \dots & A_{ij} & \dots & A_{in} & b_i \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & A_{nk} & \dots & A_{nj} & \dots & A_{nn} & b_n \end{array}\right]$$

Pivot

Rang transformé

On sait que permuter deux équations d'un même système linéaire (*i.e.* permuter deux lignes dans la matrice correspondante) ne change pas sa solution. Lors de la phase d'élimination du pivot de Gauss, on va donc chercher dans les lignes situées après chaque pivot celle qui présente le plus grand coefficient dans la même colonne (on cherche le max. des A_{jk} pour j de k à n), puis permuter les deux lignes.

***Exercice** : modifier l'algorithme d'élimination de Gauss afin d'introduire la permutation des pivots.*

3/ Méthodes indirectes (itératives) [2]

Les méthodes itératives (ou indirectes) partent d'une valeur initiale du vecteur ***x*** et tentent de converger petit à petit vers la solution exacte itération après itération, tant que le résultat n'est pas stable. Elles ont certains avantages par rapport aux solutions directes. Les principaux sont des besoins en mémoire souvent moindres, une convergence rapide (donc moins de temps de calcul) et parfois même des propriétés d'auto-correction d'erreurs d'arrondis d'une itération à l'autre.

Principe

Les méthodes itératives consistent à exprimer le calcul du vecteur x sous forme d'une relation itérative de type $x^{(k)} = f(x^{(k-1)})$ où $x^{(k)}$ désigne la valeur de x à l'étape k et f désigne une fonction de x. Le problème revient donc à calculer un point fixe obtenu comme limite d'une suite convergence. Or, le système $Ax=b$ peut s'écrire sous la forme $(M-N)x=b$ ou encore $Mx = Nx+b$. On produit alors une suite de la façon suivante :

$$\begin{aligned} x^{(1)} &= M^{-1}Nx^{(0)} + M^{-1}b \\ x^{(2)} &= M^{-1}Nx^{(1)} + M^{-1}b \\ &\vdots \\ x^{(k+1)} &= M^{-1}N^{(k)} + M^{-1}b \end{aligned}$$

où $x^{(k)}$ désigne la valeur de x à l'itération k. La valeur $x^{(0)}$ est donnée.

En général, la matrice A est décomposée sous la forme $A = D - L - U$ où D est une matrice diagonale, U une matrice triangulaire supérieure et L une matrice triangulaire inférieure. De la répartition de D, L et U entre M et N découlent trois méthodes itératives classiques :

1. La méthode de **Jacobi** où $A=M-N$ avec $M=D$ et $N=L+U$,
2. La méthode de **Gauss-Seidel** où $A=M-N$ avec $M=D-L$ et $N=U$,
3. La méthode de **Gauss-Seidel avec relaxation** où $A=M-N$ avec $M=D/\omega-L$ et $N=(1-\omega)D/\omega+U$.

Dans une autre registre il existe également la méthode du gradient conjugué que nous n'aborderons pas dans ce cours. Elle est exposée dans [2].

3.1/ Méthode de Jacobi

Le plus simple, pour comprendre Gauss-Seidel, est de partir de l'expression scalaire des équations $Ax=b$:

$$\sum_{j=1}^{n} A_{ij}x_j = b_i,\ i = 1, 2, \ldots, n$$

On peut extraire le terme contenant x_i :

$$A_{ii}x_i + \sum_{j=1, j\neq i}^{n} A_{ij}x_j = b_i,\ i = 1, 2, \ldots, n$$

Et on déduit l'expression de x_i en fonction des $x_{j,(j\neq i)}$:

$$x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} A_{ij}x_j \right),\ i = 1, 2, \ldots, n$$

On peut remarquer qu'on arriverait à la même formule en partant de l'expression matricielle :

$$x = D^{-1}(L + U)x + D^{-1}b$$

avec la décomposition $M=D$ et $N=L+U$ où $A=M-N$.

***Exercice** : montrez le (ce n'est pas très difficile) !*

On notera bien qu'il est nécessaire que les pivots (A_{ii}) ne soient pas nuls (sinon on a une division par zéro). Dans le cas contraire, on pourra permuter des lignes entre elles à l'instar du pivot de Gauss.

Convergence

Partons de la relation :

$$\begin{aligned} Mx^{(k+1)} &= Nx^{(k)} + b \\ x^{(k+1)} &= M^{-1}Nx^{(k)} + M^{-1}b \end{aligned}$$

L'algorithme converge si la différence entre $x^{(k+1)}$ et $x^{(k)}$ diminue quand k augmente, c'est à dire si :

$$lim_{k\to\infty}||e^{(k)}|| = 0$$

avec

$$e^{(k+1)} = x^{(k+1)} - x^{(k)} = M^{-1}N(x^{(k)} - x^{(k-1)}) = M^{-1}Ne^{(k)}$$

Si l'on note $B=M^{-1}N$, alors

$$e^{(k+1)} = Be^{(k)} = B^k e^{(0)}$$

Ceci implique que l'algorithme converge pour tout vecteur initial $x^{(0)}$ si

$$lim_{k\to\infty}||e^{(k)}|| = 0 \Leftrightarrow lim_{k\to\infty}||B^k|| = 0$$

***Théorème :** Une condition nécessaire et suffisante pour que $lim_{k->\infty}||B^k||=0$ est que le rayon spectral (plus grande valeur propre en valeur absolue) de B soit strictement inférieur à 1.*

Propriété (rappel) : une matrice est dite à **diagonale dominante** si :

$$\forall i, |A_{ii}| > \sum_{j \neq i} |A_{ij}|$$

(en chaque ligne de la matrice, le coefficient situé sur la diagonale est strictement supérieur en valeur absolue à la somme de tous les autres coefficients, en valeurs absolues, de la ligne).

Théorème : *Les méthodes de Jacobi et Gauss-Seidel s'appliquent sur Ax=B et convergent si A est à diagonale dominante.*

Note : lorsque la propriété n'est pas vérifiée sur la matrice d'origine, on peut essayer de permuter les lignes ou les colonnes de la même façon qu'avec le pivot de Gauss.

Algorithme

On commence par choisir le vecteur x_0 de départ, soit à partir d'une solution "intuitée", soit aléatoirement. L'expression précédente permet de calculer, à chaque itération k+1, les valeurs des $x_i^{(k+1)}$ en fonction des valeurs calculées des $x_j^{(k)}{}_{j \neq i}$ à l'itération précédente k. La procédure est itérée jusqu'à ce que la différence entre les deux vecteurs obtenus à deux itérations successives soit suffisamment petite.

$$x_i^{(k+1)} \leftarrow \frac{1}{A_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} A_{ij} x_j^{(k)} \right), \ i = 1, 2, \ldots, n \tag{3.3}$$

3.2/ Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est très proche de la méthode de Jacobi mais elle converge plus rapidement car elle utilise les dernières valeurs des x_i calculées (y compris durant la même itération) et non pas seulement les valeurs calculées à l'itération précédente. La relation itérative s'écrit donc :

$$x_i^{(k+1)} \leftarrow \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right), \ i = 1, 2, \ldots, n \tag{3.4}$$

Exercice : *montrer que l'on retrouve cette relation en partant de l'expression matricielle du problème avec la décomposition A=M-N avec M=D-L et N=U.*

Algorithme

L'algorithme suivant calcule un vecteur `x`, solution du système linéaire donné par la matrice `A` et le vecteur `b`. On donne également `eps`, la précision souhaitée. Il est composé de deux fonctions : `IterX` qui applique une itération à l'ensemble du vecteur `x`, et `GaussSeidel` qui applique itérativement `IterX` et s'arrête quand la différence entre les deux vecteurs est suffisamment petite (on utilise la norme du vecteur d'erreur entre deux itérations**,** càd la racine carrée de son produit vectoriel).

```
PROCEDURE IterX(A : matrice, b : vecteur ; x : vecteur) /* x en entrée/sortie */
DEBUT
     POUR i de 1 à n FAIRE                  /* pour tous les x[i] */
          somme = 0.0                       /* somme de A[i,j]*x[j] */
          POUR j de 1 à n FAIRE
               SI j ≠ i ALORS               /* pour tous j différents de i*/
                    somme = somme + A[i,j]*x[j]
               FINSI
          FINPOUR
          x[i] = (b[i] - somme) / A[i,i]    /* calcul final */
     FINPOUR
FIN
```

```
PROCEDURE GaussSeidel(A : matrice, b : vecteur, epsilon : reel ; X : vecteur)
DEBUT
     oldX : vecteur
     dX : reel
     FAIRE
          oldX = X                                  /* sauvegarde du vecteur x */
          IterX(A,b ; x)                            /* màj du vecteur x */
          dX = sqrt(dot(x - oldX, x - oldX))        /* dot = produit vectoriel */
     TANT QUE (dX < epsilon)
FIN
```

Complexité

Chaque itération de l'algorithme nécessite $n(2n\text{-}1)$ opérations, c'est-à-dire n divisions, $n(n\text{-}1)$ soustractions et multiplications. Ces méthodes sont particulièrement intéressantes, par rapport aux méthodes directes, lorsqu'on se contente d'une dizaine d'itérations sur de très grandes matrices ($n > 100$).

Exercice : *connaissant la complexité de l'algorithme du pivot de Gauss et celle de Jacobi ou Gauss-Seidel, montrer (très approximativement) que pour un petit nombre d'itérations ce genre de méthode itérative nécessite moins de calculs.*

3.3/ Un peu de relaxation...

La vitesse de convergence de la méthode de Gauss-Seidel peut être augmentée dans la plupart des cas par une technique dite de *relaxation*. Le principe consiste à calculer la nouvelle valeur des x_i comme une somme pondérée de leur ancienne valeur (à l'itération précédente) et de la valeur calculée par la formule (3.3). Le calcul itératif se présente alors sous la forme :

$$x_i \leftarrow \frac{\omega}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right) + (1-\omega) x_i, \; i = 1, 2, \ldots, n \qquad (3.5)$$

où ω est appelé *facteur de relaxation*. Si ω=1, il n'y a aucune relaxation ; si ω<1 il s'agit d'une interpolation entre l'ancienne valeur de x_i et la valeur donnée par (3.4) appelée *sous-relaxation* ; si ω>1 il s'agit d'une extrapolation appelée *sur-relaxation*. (on utilise ce type de valeur lorsque le processus à tendance à diverger). Dans tous les cas, si ω>2 le processus diverge.

Il n'existe pas de méthode pratique pour déterminer la valeur optimale du facteur de relaxation ω. Mais on sait en calculer une bonne estimation pendant les calculs, en fonction de la norme du vecteur variation de X à l'itération k : $dX^{(k)} = |X^{(k-1)} - X^{(k)}|$. Il a été montré que, pour k suffisamment grand (k>5) l'expression suivante

$$\omega_{opt} \approx \frac{2}{1 + \sqrt{1 - (dX^{(k+p)}/dX^{k})^{1/p}}} \tag{3.5}$$

où p est un entier positif, est une bonne approximation de ω.

Le principe de l'algorithme de GaussSeidel avec relaxation est donc le suivant :

1. Exécuter k itérations avec ω=1 (par exemple k = 10). À la $k^{ième}$ itération, mémoriser $dX^{(k)}$,
2. Exécuter p autres itérations ($p \geq 1$) et mémoriser $dX^{(k+p)}$ à la dernière itération,
3. Continuer l'algorithme avec $\omega_{opt}=\omega$, où ω_{opt} est calculé par l'expression (3.5).

Voici l'algorithme correspondant, avec p = 1 et k =10 :

```
PROCEDURE IterXRelax(A : matrice, b : vecteur, omega : réel ; x : vecteur)
DEBUT
     POUR i de 1 à n FAIRE                /* pour tous les x[i] */
          somme = 0.0                     /* calcul de la somme de A[i,j]x[j] */
          POUR j de 1 à n FAIRE
               SI j ≠ i ALORS             /* pour tous j différents de i*/
                    somme = somme + A[i,j]*x[j]
               FINSI
          FINPOUR
          X[i] = (omega * (b[i] - somme)) / A[i,i] + (1 - omega) * x[i]
     FINPOUR
FIN
```

```
PROCEDURE GaussSeidelRelax(A : matrice, b : vecteur, eps : réel ; x : vecteur)
DEBUT
     oldX : vecteur
     k, p, nbIter : entier
     omega, dX, dXk : réel
     k = 10 ; p = 1 ; nbIter = 1; omega = 1.0
     FAIRE
          oldX = x                          /* sauvegarde du vecteur x */
          IterXRelax(A,b,omega ; x)         /* màj du vecteur x */
          dX = sqrt(dot(x - oldX, x - oldX))    /* dot = produit vectoriel */
          SI (nbIter == k) ALORS            /* sauvegarde de dX à l'iteration k */
               dXk = dX
          FINSI
          SI (nbIter = k+p) ALORS           /* calcule omega à l'iteration k+p */
               omega = 2 / (1 + sqrt(1 - (dX/dXk)^(1/p)))
          FINSI
     TANT QUE (dX < epsilon)
FIN
```

VI - Bibliographie

[1] *Algorithmique numérique,* C. Brezinski, Editions Ellipses, 1988.

[2] *Numerical Methods in Engineering with MATLAB*, Jaan Kiusalaas, Cambridge University Press, 2005.

[3] *Les codes de calcul, un support indispensable à la sureté de la conception et de l'exploitation des réacteurs [nucléaires]*, Thierry Nkaoua, http://www.ulb.ac.be/polytech/smana/Seances/Synoptique/cs/cs.htm.

[4] *ARIANE 501, Rapport de la commision d'enquête*, http://www.capcomespace.net/dossiers/espace_europeen/ariane/ariane5/AR501/AR501_rapport_denquete.htm.

[5] *The Patriot Missile Failure*, http://www.math.psu.edu/dna/disasters/patriot.html.

[6] *The Perils of Floating Points*, http://www.lahey.com/float.htm.

[7] *Architecture de l'ordinateur*, A. Tanenbaum, Éditions Dunod, 2001.

Annexe - Quelques rappels mathématiques

Une **matrice** est un tableau de nombres de *n* lignes et *m* colonnes (*n* et *m* sont les dimensions):

$$A_{n\times m} = \begin{array}{c} 1 \\ 2 \\ \vdots \\ i \\ \vdots \\ n \end{array} \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1m} \\ a_{21} & a_{22} & & & & a_{2m} \\ \vdots & & \ddots & & & \vdots \\ a_{i1} & & & a_{ij} & & a_{im} \\ \vdots & & & & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nj} & \dots & a_{nm} \end{pmatrix}$$

- Si ***n* = *m*** la matrice est dite carrée.
- Si ***m* = *1*** la matrice est un vecteur (colonne).
- Si ***n* = *1*** la matrice est un vecteur ligne.

Quelques matrices courantes :

- ***I*** est la matrice **identité** (ou unité),
- ***D*** est une matrice **diagonale**,
- ***U*** est **triangulaire supérieure** (*Up*),
- ***L*** est **triangulaire inférieure** (*Low*)

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} L = \begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix} U = \begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix} D = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$$

Limite et convergence :

- Une suite numérique (u_n) converge vers un réel L si pour tout réel $e > 0$, il existe un entier N tel que si $n > N$, alors $| u_n - L | < e$.

- Une fonction numérique f admet la limite L au point x_0 si pour tout réel $e > 0$, il existe un réel h tel que pour tout x vérifiant $0 < | x - x_0 | < h$, on ait $| f(x) - L | < e$. On note alors $lim_{x->x0}\, f(x) = L$.

Continuité et dérivabilité :

- Une fonction ***f*** est **continue en un point x_0** si :

$$lim_{x\to x_0^-} f(x) = lim_{x\to x_0^+} f(x) = L$$

- Une fonction ***f*** est **continue sur un intervalle** si elle est continue en tous points de cet intervalle.

- Une fonction ***f*** est **dérivable en un point x_0** si il existe un réel noté ***f'(x_0)*** tel que :

$$lim_{x\to x_0^-} \frac{f(x) - f(x_0)}{x - x_0} = f'(x_0)$$

- Une fonction ***f*** est **dérivable sur un intervalle** si elle est dérivable en tous points de cet intervalle.
- Une fonction dérivable en un point est continue en ce point.

Dérivées usuelles :

$$\begin{array}{rclrcl} (f+g)' & = & f'+g' & (f-g)' & = & f'-g' \\ (f\times g)' & = & f'\times g + f\times g' & (x^n)' & = & n\times x^{n-1} \\ (f/g)' & = & (f'\times g + f\times g')/g^2 & (f\circ g)' & = & g'\times (f'\circ g) \end{array}$$

Séries entières :

- On appelle **série entière** (réelle ou complexe) toute série numérique de la forme :

$$\sum_{i=0}^{\infty} a_i x^i$$

- Le **domaine de convergence *D*** de la série entière est l'ensemble des x pour lesquels la série converge.

Formule de Taylor : approximation d'une fonction plusieurs fois dérivable au voisinage d'un point par une fonction polynôme dont les coefficient dépendent des dérivées successives de la fonction en ce point.

- Si *n* est un entier naturel et *f* est une fonction définie et dérivable *n* fois sur un intervalle *I* contenant *a* :

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R(x)$$

R(x) est un reste qui dépend de ***x*** et est très petit quand *x* tend vers *a*.

Développements usuels :

Développement	**Domaine de convergence**
$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$	$\{\|x\| < 1\}$
$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$	R
$ln(1-x) = -\sum_{i=1}^{\infty} \frac{x^i}{i}$	$\{\|x\| < 1\}$
$(1+x)^n = \sum_{i=0}^{\infty} C_i^n x^i$	$\{\|x\| < 1\}$

IV - Interpolation et approximation

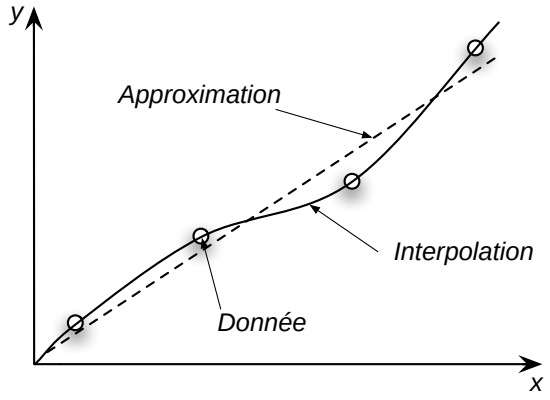
L'expérimentation est un aspect fondamental des sciences. Cela semble évident dans certains domaines comme la physique, mais cela peut être aussi important pour l'étude de propriétés mathématiques (souvent dans le domaines des nombres). Dans tous les cas sont impliqués des ensembles de données discrètes, souvent grands, issus d'observations expérimentales, ou de simulations (calculs numériques). Ces ensembles se présentent sous la forme de couples (x_i, y_i) où x_i peut être une mesure du temps ou bien de l'espace et y_i la mesure d'une propriété (température, distance...).

x_1	x_2	x_3	...	x_n
y_1	y_2	y_3	...	y_n

Lorsque le phénomène est continu (dynamique des fluides, ondes électromagnétiques etc...), une phase d'échantillonnage du signal est nécessaire. Nous n'entrerons pas dans les détails de ce type de problème dans la suite, mais il ne faut pas perdre de l'esprit que cette transformation est rarement sans conséquences sur les calculs ultérieurs.

L'objectif général est d'extraire de ces mesures une propriété mathématique (fonction de x) exploitable plus directement sur le même intervalle (par exemple pour dériver ou intégrer) ou bien pour extrapoler le comportement du phénomène au delà de l'intervalle de mesure.

Interpolation (*interpolation* en anglais) et approximation (*curve fitting* en anglais) sont deux traitements différents. Le premier consiste à construire une courbe continue passant exactement par tous les points du plan xOy. Le second s'applique plutôt à des ensemble de données contenant du "bruit" (*scatter* en anglais), c'est-à-dire des données issues de mesures légèrement perturbées par un ou plusieurs phénomènes extérieurs. Dans ce cas le problème n'est pas de faire passer la courbe par tous les points mesurés mais plutôt de déterminer une fonction de x dont la courbe approxime au mieux les données. La figure suivante illustre cette différence entre interpolation et approximation.



1/ Méthodes d'interpolation polynômiale

La forme la plus simple de fonction d'interpolation est le polynôme : il suffit de trouver les coefficients d'un polynôme passant par tous les points (x_i, y_i). Or il est toujours possible de construire un unique polynôme $P_{n-1}(x)$, de degré n-1, qui passe par n points distincts. Après une présentation de la méthode de Lagrange, nous détaillerons celles de Newton et de Neville.

Méthode de Lagrange

Pour obtenir le polynôme $P_{n-1}(x)$, on peut utiliser la *formule de Langrange* :

$$P_{n-1}(x) = \sum_{i=1}^{n} y_i \ell_i(x) \tag{4.1}$$

où les

$$\ell_i(x) = \frac{x-x_1}{x_i-x_1}.\frac{x-x_2}{x_i-x_2}\cdots\frac{x-x_{i-1}}{x_i-x_{i-1}}.\frac{x-x_{i+1}}{x_i-x_{i+1}}\cdots\frac{x-x_n}{x_i-x_n} = \prod_{j=1,j\neq i}^{n}\frac{x-x_j}{x_i-x_j} \tag{4.2}$$

sont appelées *fonctions cardinales*.

Ainsi, pour n=2, la fonction d'interpolation est la droite qui passe par les deux points (x_1,y_1) et (x_2,y_2) et qui a pour équation :

$$P_1(x) = y_1\ell_1(x) + y_2\ell_2(x)$$

avec

$$\ell_1(x) = \frac{x-x_2}{x_1-x_2} \qquad \text{et} \qquad \ell_2(x) = \frac{x-x_1}{x_2-x_1}$$

Les *fonctions cardinales* constituent elles-mêmes des polynômes de degré n-1 car l'expression générale (4.2) contient n-1 facteurs (il n'y a pas le $i^{ième}$ facteur dans le produit). On peut noter également qu'elle disposent de la propriété suivante :

$$\ell_i(x_j) = \left\{ \begin{array}{ll} 0 & \text{si } i \neq j \\ 1 & \text{si } i = j \end{array} \right\} = \delta_{ij}$$

qu'on appelle le symbole de Kronecker. Cela signifie qu'une fonction cardinale de rang i est nulle pour toute valeur x_j avec $j\neq i$ et qu'elle vaut 1 quand $i=j$.

Cela suffit pour montrer que ce polynôme "passe" par tous les points (x_i,y_i) : on vérifie que pour tout $j\leq n$:

$$P_{n-1}(x_j) = \sum_{i=1}^{n} y_i\ell_i(x_j) = \sum_{i=1}^{n} y_i\delta_{ij} = y_1\delta_{1j}+y_2\delta_{2j}+\ldots+y_j\delta_{jj}+\ldots+y_n\delta_{nj} = y_j$$

La méthode de Lagrange est assez simple sur le principe mais elle ne se prête pas particulièrement bien à l'élaboration d'un algorithme efficace (pour pré-calculer les coefficients ou pour évaluer directement sa valeur en chaque point). On lui préfère souvent l'algorithme issu de la méthode de Newton.

Méthode de Newton

Le polynôme d'interpolation de Newton s'écrit sous la forme :

$$P_{n-1}(x) = a_1 + (x - x_1)a_2 + (x - x_1)(x - x_2)a_3 + \ldots + (x - x_1)(x - x_2)\ldots(x - x_{n-1})a_n \qquad (4.3)$$

L'évaluation de ce polynôme en une valeur *x* quelconque est assez simple et efficace. En effet, on peut factoriser l'expression tel que :

$$P_{n-1}(x) = a_1 + (x - x_1)[a2 + (x - x_2)[a3 + \ldots[(x - x_{n-1})a_n]\ldots]$$

À partir de cette formule, on déduit une méthode d'évaluation récursive définie par :

$$\begin{aligned} P_0(x) &= a_{n-1} \\ P_1(x) &= a_{n-2} + (x - x_{n-1})P_0(x) \\ P_2(x) &= a_{n-3} + (x - x_{n-2})P_1(x) \\ &\vdots \\ P_{n-1}(x) &= a_1 + (x - x_1)P_{n-2}(x) \end{aligned}$$

Cela s'écrit autrement sous la forme :

$$P_0(x) = a_n \qquad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, \ldots, n-1 \qquad (4.4)$$

=> ALGORITHME é