Chrysostome Baskiotis

Laurence Lamoulie

ANALYSE NUMÉRIQUE

Fascicule 1

Analyse des erreurs

EISTI

Année 2011

INTRODUCTION

L'objectif de l'analyse numérique est d'obtenir pour un problème qui est exposé en termes mathématiques, des réponses numériques. Par réponse numérique on entend une solution « approchée » du problème, par opposition à la solution « exacte » obtenue par des méthodes mathématiques, à savoir algébriques ou analytiques. Il ne faut pas toutefois restreindre le domaine d'application de l'analyse numérique uniquement aux problèmes qui ont une solution mathématique précise, car, en effet, un autre grand domaine d'utilisation de l'analyse numérique concerne la résolution des problèmes mathématiques pour lesquels on ne connaît pas de façon analytique la solution « exacte ». La matière première de l'analyse numérique est le nombre, qu'il s'agisse d'un entier, d'un réel ou, encore, d'un complexe. Nous commencerons donc l'étude de l'analyse numérique par les nombres et nous nous apercevrons, avec étonnement, que les relations des nombres avec l'ordinateur sont souvent difficiles. En effet, le passage de ce nombre par un ordinateur numérique peut modifier de façon significative sa précision et, par voie de conséquence son exactitude et sa valeur. Un ordinateur « comprend » donc avec des erreurs les nombres réels que nous lui fournissons et aussi quand il effectue des calculs numériques, il n'est pas rare que ces calculs soient entachés d'erreurs. Expliquer, dans le cadre de l'algèbre linéaire, les raisons de ces erreurs, les analyser et les minimiser, autant que faire se peut, est le but de ce cours. Le cours se divise en trois parties : La première concerne l'étude des erreurs provoquée par la représentation tronquée des valeurs numériques par l'ordinateur. Les deux autres parties sont consacrées à l'algèbre linéaire. Ainsi, à la deuxième partie on étudiera les effets de perturbations des valeurs des vecteurs et matrices à la résolution des systèmes d'équations linéaires par des méthodes directes et itératives. Enfin à la troisième partie on étudiera le calcul numérique des valeurs et vecteurs propres. Quelques applications, comme la méthode des moindres carrés ou l'approximation des images par la décomposition en valeurs singulières, seront aussi présentées.

1

ANALYSE DES ERREURS

Un ordinateur M utilise pour la représentation des valeurs numériques un mot machine qui a un nombre fini de bits. Par conséquent l'ensemble des nombre réels R sera « vu » par l'ordinateur comme un sous-ensemble fini $M(R) \subset R$ (i.e. l'ordinateur opère un échantillonnage – au sens traitement du signal du terme – de R). Cette représentation est une application $fl : R \rightarrow M(R)$, telle que pour tout réel x l'application fl fournit une valeur $fl(x)$, appelée nombre-machine. Dans la mesure où, en général, nous avons $x = fl(x)$, il s'ensuit qu'une erreur est commise chaque fois que nous demandons à un ordinateur de manipuler une valeur numérique.

L'objectif de ce chapitre est d'étudier, de façon détaillée, les erreurs numériques provoquées par un ordinateur.

1.1 Introduction

Un nombre réel quelconque en représentation décimale est caractérisé par deux propriétés :

– L'exactitude qui est le nombre de digits([1]) significatifs

– La précision qui est l'ordre de grandeur du dernier digit significatif.

Ainsi pour le nombre 0.0017892 nous avons une exactitude de 5 et une précision de 10^{-7}. Le passage de ce nombre par un ordinateur numérique peut modifier de façon significative sa précision et, par voie de conséquence son exactitude et sa valeur. Un ordinateur « comprend »

1. Pour un chiffre décimal on utilisera en abrégé le mot digit.

donc avec des erreurs les nombres réels que nous lui fournissons et aussi quand il effectue des calculs numériques.

Utiliser donc un ordinateur pour représenter un nombre réel, a comme conséquence de faire (presque) inévitablement une erreur. Pourquoi ? C'est assez simple de comprendre les raisons. La droite des réels $]-\infty, +\infty[$ a la puissance du continu. Cet ensemble infini de nombres nous essayons de le représenter par un ordinateur qui, pour ce faire, utilise des bits en nombre fini et même limité! Dès lors on est devant une situation où entre deux nombres réels consécutifs de l'ordinateur il y a une infinité de nombres réels que l'ordinateur ne peut pas représenter et ne représentera jamais. Il tentera seulement de les approcher en faisant ainsi une erreur sur la valeur exacte du nombre que l'on appellera erreur de représentation.

Plus concrètement, considérons un nombre binaire :

$$b_q b_{q-1} \ldots b_1 . b_{-1} \ldots b_{-p}$$

Nous écrivons ce nombre sous forme normalisée, à savoir en partie entière on a un 0 le reste se trouve en partie non entière (après le point) suivi d'une puissance de 10. Par exemple, pour le nombre précédent, sa forme normalisée est

$$0.b_q b_{q-1} \ldots b_1 b_{-1} \ldots b_{-p} \times 10^{q}$$ [2]

La partie après le point $b_q b_{q-1} \ldots b_1 b_{-1} \ldots b_{-p}$ s'appelle mantisse[3] et pour un nombre réel il a, en général, une infinité d'éléments. Dans notre cas la mantisse est composée de $p+q$ bits, où p peut être infini. Pour stocker un réel dans un ordinateur, il faut pouvoir stocker sa mantisse (et aussi son exposant q, mais pour l'instant on ne se préoccupe de celui-ci). Étant donné qu'un nombre est stocké dans un mot de la mémoire de l'ordinateur et le mot est composé d'un nombre fini de bits, on aboutit à la conclusion que le stockage d'un réel dans un ordinateur dont la mémoire à m bits, avec $m < p+q$, provoque la troncature de sa valeur à $0.b'_1 \ldots b'_{-m}$, avec $b'_1 = b_q$, $b'_2 = b_{q-1}$, ...

Remarquons que nous procédons de la même façon quand on travaille avec des nombres réels et qu'après une opération on décide de s'arrêter après, par exemple, cinq chiffres décimaux.

L'objectif de la première partie du cours est l'étude des erreurs de représentation et leur influence sur les résultats des calculs lors du déroulement d'un algorithme. Nous commençons par l'étude des nombres réels et de leur représentation par un ordinateur.

---

2. En général dans la forme normalisée, on place après le point tous les chiffres significatifs, c'est-à-dire le premier chiffre après la virgule doit être différent de zéro. Par exemple la forme normalisée de 0.0001 est 0.1×10^{-3}.

3. Notons que dans des temps plus anciens on appelait cette partie significande. Apparemment la dimension sémantique dans la terminologie informatique n'est pas au goût du jour. D'où l'introduction du terme mantisse qui signifie partie décimale d'un logarithme.

EN SUBSTANCE

- ***Écriture sous forme normalisée d'un nombre :***
  Soit le nombre 34.10285. ***En écriture normalisée on a*** 0.3410285×10^2, ***c'est-à-dire :***
  - 0 ***en partie entière ;***
  - ***passage du contenu de la partie entière au début de la partie décimale, et***
  - ***multiplication avec une puissance de la base de numérotation*** 10.[a]

- ***Dans un ordinateur les nombres sont stockés sous forme normalisée.***

- ***Un ordinateur dispose d'un nombre limité de places pour stocker les chiffres de la partie décimale d'une valeur sous forme normalisée. Étant donné qu'un réel a, le plus souvent, un nombre infini de chiffres en partie décimale, le stockage exact d'un tel nombre dans un ordinateur est impossible. Le nombre réellement stocké est une approximation du nombre fourni par l'utilisateur.***

a. La base de numérotation sera notée toujours 10, indépendamment du système de numérotation (décimal, binaire, héxadécimal, ...).

1.2 L'ordinateur et les nombres

D'abord il faut se rappeler qu'un ordinateur traite des nombres en base binaire et non pas décimale. Mais ce fait ne doit pas créer un problème, car un nombre binaire est comme un nombre décimal. Il a un point que l'on qualifiera de binaire (équivalent du point décimal) et il est composé de 0 et de 1. Par exemple le nombre décimal 13.125 est l'écriture condensée du nombre $1 \times 10^1 + 3 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$. De même le nombre binaire 11011.001 est l'écriture condensée du nombre $2^4 + 2^3 + 2^1 + 2^0 + 2^{-3}$. Nous savons aussi que les nombres réels peuvent être représentés sous une forme qui est appelée notation scientifique. Ainsi le nombre 13.125 peut se mettre sous la forme 1.3125×10^1. De la même façon nous pouvons avoir une notation scientifique pour les nombres binaires. Par exemple le nombre binaire 11011.001 peut s'écrire à l'aide de cette notation $1.1011001 \times 10^{100}$.

Un ordinateur dispose, pour représenter les nombres, des registres dont le nombre de bits est limité habituellement à 16, 32 ou 64 bits. Donc pour pouvoir représenter les nombres réels par un ordinateur nous devons, si on veut tenir compte du nombre fini de bits d'un registre, introduire deux limitations à la représentation par notation scientifique des nombres :

(1) Le nombre de chiffres qu'on utilise après le point est fixé d'avance. Si par exemple on fixe à 4 ce chiffre, le nombre 11011.001 sera représenté par 1.1011×10^{100}.

(2) Le nombre de chiffres qu'on utilise pour la valeur de l'exposant de la base est fixé d'avance. Par exemple si on fixe, sur la représentation précédente, le nombre de chiffres de l'exposant à 2, on aura finalement comme représentation 1.1011001×10^{10}.

Cette représentation – qui peut être tronquée – des nombres s'appelle forme flottante. Les ordinateurs actuels utilisent pour les nombres flottants le standard IEEE-754 dont les caractéristiques pour les nombres en simple précision sont données par le tableau 2.1.

Signe s	Exposant e	Mantisse m
s = 1 bit	q = 8 bits	p = 23 bits
□	□□□□□□□□	□□□□□□□□□□□□□□□□□□□□□□□
±	$a_1a_2a_3a_4a_5a_6a_7a_8$	$b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}$

TABLE 1.1 – Standard IEEE-754. Simple précision

1.2.1 Conversion décimale – binaire

Nous allons examiner cette structure du nombre flottant en simple précision à l'aide d'un exemple. Prenons le nombre 465.463. Pour le convertir en flottant binaire normalisé on doit

- d'abord convertir séparément la partie entière et la partie décimale ;
- ensuite réunir les deux parties et
- à la fin, de normaliser le résultat de la réunion.

Nous avons ainsi pour la partie entière :

(1) 465 / 2 = 232 Reste = 1

(2) 232 / 2 = 116 Reste = 0

(3) 116 / 2 = 58 Reste = 0

(4) 58 / 2 = 29 Reste = 0

(5) 29 / 1 = 14 Reste = 1

(6) 14 / 2 = 7 Reste = 0

(7) 7 / 2 = 3 Reste = 1

(8) 3 / 2 = 1 Reste = 1

d'où $(465)_{10} = (111010001)_2$.

Pour la partie décimale, nous avons :

(1) 0.4063 * 2 = 0.926 Partie enti\'{e}re = 0

(2) 0.926 * 2 = 1.852 Partie enti\'{e}re = 1

(3) 0.852 * 2 = 1.704 Partie enti\'{e}re = 1

(4) 0.704 * 2 = 1.408 Partie enti\'{e}re = 1

(5) 0.408 * 2 = 0.816 Partie enti\'{e}re = 0

(6) 0.816 * 2 = 1.632 Partie enti\'{e}re = 1

(7) 0.632 * 2 = 1.264 Partie enti\'{e}re = 1

(8) 0.264 * 2 = 0.528 Partie enti\'{e}re = 0

(9) 0.528 * 2 = 1.056 Partie enti\'{e}re = 1

(10) 0.056 * 2 = 0.112 Partie enti\'{e}re = 0

(11) 0.112 * 2 = 0.224 Partie enti\'{e}re = 0

(12) 0.224 * 2 = 0.448 Partie enti\'{e}re = 0

(13) 0.448 * 2 = 0.896 Partie enti\'{e}re = 0

(14) 0.896 * 2 = 1.792 Partie enti\'{e}re = 1

(15) 0.792 * 2 = 1.584 Partie enti\'{e}re = 1

(16) 0.584 * 2 = 1.168 Partie enti\'{e}re = 1

(17) 0.168 * 2 = 0.336 Partie enti\'{e}re = 0

(18) 0.336 * 2 = 0.672 Partie enti\'{e}re = 0

(19) 0.672 * 2 = 1.344 Partie enti\'{e}re = 1

(20) 0.344 * 2 = 0.688 Partie enti\'{e}re = 0

(21) 0.688 * 2 = 1.376 Partie enti\'{e}re = 1

(22) 0.376 * 2 = 0.752 Partie enti\'{e}re = 0

(23) 0.752 * 2 = 1.504 Partie enti\'{e}re = 1

(24) 0.504 * 2 = 1.008 Partie enti\'{e}re = 1

(25) 0.008 * 2 = 0.016 Partie enti\'{e}re = 0

(26) 0.016 * 2 = 0.032 Partie enti\'{e}re = 0

(27) 0.032 * 2 = 0.064 Partie enti\'{e}re = 0

(28) 0.064 * 2 = 0.128 Partie enti\'{e}re = 0

(29) 0.128 * 2 = 0.256 Partie enti\'{e}re = 0

(30) 0.256 * 2 = 0.512 Partie enti\'{e}re = 0

(31) .....

d'où $(0.463)_{10} = (0.01110110100000111001010111000000\ \ldots)_2$.

Ainsi nous avons la conversion $(465.463)_{10} = (111010001.01110110100000111001010111000000\ldots)_2$ qui sous forme normalisée devient $0.11101000101110110100000\ldots \times 10^{1001}$. Il faut maintenant transformer ce nombre binaire, qui a un nombre de chiffres infini, en forme binaire flottante en simple précision, c'est-à-dire calculer les valeurs du signe, de l'exposant et de la mantisse.

1.2.2 Élaboration de la forme flottante

Selon le standard IEEE-754, le signe est codé sur un bit qui a la valeur 0 si le nombre est positif et 1 dans le cas contraire. Ici nous avons un nombre positif, donc le bit du signe est égal à 0.

Pour l'exposant le standard utilise 8 bits. Nous pouvons donc disposer, pour le codage de l'exposant, de 256 valeurs différentes. Il faut aussi tenir compte du fait que l'exposant peut être positif ou négatif. On peut, bien sûr, avoir ici aussi un bit de signe. Mais dans ce cas les opérations

d'addition et de soustraction entre différents nombres deviendraient ardues. Comme donc, on ne veut pas avoir un codage avec des nombres positifs et négatifs, on partage l'intervalle de variation de l'exposant en deux parties égales : la première de 0 à 126 consacrée au codage des exposants négatifs et l'autre partie de 128 et 255 consacrée au codage des exposants positifs, la valeur 127 étant réservée à l'exposant 0.

Dans l'exemple précédent, l'exposant est égal à $1001_2 = 9_{10}$. C'est donc une valeur positive. Par conséquent pour l'exposant 9 on rajoute 127 et on obtient ainsi $9 + 127 = 136 = (10001000)_2$.

La mantisse doit être 11101000101110110100000. Remarquons maintenant que si, dans le cas des nombres décimaux, on décidait de placer le premier chiffre significatif (chiffre non nul) avant le point décimal, ce chiffre serait soit 1, soit 2, ..., soit 9. Mais dans le cas des nombres binaires ce chiffre serait obligatoirement 1. Par conséquent on peut toujours, dans le cas des nombres binaires, placer ce chiffre et ne pas le faire apparaître dans le codage (bit caché), ce qui nous permet de gagner un bit supplémentaire à la mantisse pour le codage. Ainsi la mantisse s'écrit 11010001011101101000001 et l'exposant diminue d'une unité et dévient égal à $8 + 127 = 135 = (10000111)_2$ et on obtient donc pour la forme flottante normalisée binaire du nombre 465.463 la représentation suivante :

0	10000111	11010001011010000000000

Si maintenant on fait la démarche inverse, on peut calculer la valeur décimale du nombre binaire que nous venons de créer en utilisant la conversion en simple précision. Nous avons

$$1.11010001011010000000000 \times 10^{10000111 - 01111111} = 111010001.011101101000001$$
$$= 2^8 + 2^7 + 2^6 + 2^4 + 2^0.2^{-2} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-7} + 2^{-9} + 2^{-15}$$
$$= 465.462921142578125$$

Remarquons qu'en vue du décodage en décimal, nous avons incorporer en première position, avant le point, le bit caché qui ne figurait pas dans la représentation flottante.

Bien sûr la question qui vient à l'esprit concerne la capacité de ce système de représenter tous les nombres. En effet nous pouvons envisager que, comme conséquence d'une opération arithmétique, nous avons un nombre très grand (ou très petit) qui ne peut pas être représenté comme un binaire flottant normalisé. Avant d'examiner en détail cette éventualité, il faut établir les valeurs extrêmes que ce système peut représenter ainsi que la résolution du système, c'est-à-dire la valeur de la plus petite différence entre deux nombres que le système est capable de détecter.

EN SUBSTANCE

- ***Forme flottante en simple précision selon le standard IEEE-754 :*** $x = (-1)^S \times M^E$ ***où***
  - x ***le nombre,***
  - S ***le signe stocké sur un bit,***
  - M ***la mantisse stockée sur 23 bits, et***
  - E ***l'exposant stocké sur 8 bits.***

- ***Conversion d'un nombre*** $V.D$ ***en base décimale vers un nombre en base*** β***.- Notons***
  - ***par*** $(V;v,r)_\beta$ ***le résultat de la division de*** V ***par*** β***, c'est-à-dire*** $V = v \times \beta + r$***.***
  - ***par*** $[D;d,f]_\beta$ ***le résultat de la multiplication de*** $0.D$ ***par*** β***, c'st-à-dire*** $d.f = 0.D \times \beta$***.***

  Posons $v_0 = V$ ***et considérons la suite de divisions***

  $$(v_0;v_1,r_0)_\beta, (v_1;v_2,r_1)_\beta, \ldots, (v_{p-1};v_p,r_{p-1})_\beta$$

  avec $r_{p-1} < \beta$***.***

  Considérons aussi la suite de multiplications

  $$[D;d_1,f_{-1}]_\beta, [f_1;d_2,f_2]_\beta, \ldots, [f_{q-1};d_q,f_q]_\beta, \ldots$$

  Alors la conversion du nombre décimal $V.D$ ***en base*** B ***est donnée par***

  $$v_p r_p r_{p-1} \ldots r_1 r_0 . d_1 d_2 \ldots d_q \ldots$$

- ***Conversion d'un nombre*** $v_p r_p r_{p-1} \ldots r_1 r_0 . d_1 d_2 \ldots d_q \ldots$ ***en base*** β ***vers un nombre*** $V.D$ ***en base décimale.- Supposons que la valeur*** β ***de la base est exprimée en base décimale, c'est-à-dire*** $\beta = \beta_{10}$***. Alors***

  $$V.D = v_p \times \beta^p + r_{p-1} \times \beta^{p-1} + \ldots + r_1 \times \beta^1 + r_0 \times \beta^0 . d_1 \times \beta^{-1} + d_2 \times \beta^{-2} + \ldots + d_q \times \beta^{-q} + \ldots$$

- ***Particularité du codage en forme flottante binaire. Le chiffre le plus significatif n'est pas stocké dans un bit (bit caché), mais il est toujours pris en compte lors des calculs et du passage vers la forme flottante décimale.***

1.3 Nombres normalisés et spéciaux

Nous venons de voire que la représentation flottante d'un nombre x est donnée par

$$x = \pm (b_0.b_1 b_2 \ldots b_p) \times 10^{e_1 \ldots e_q}$$

avec $b_0 = 1$ et $1 \leq 1.b_1 b_2 \ldots b_p < 2$. Le fait que nous avons toujours $b_0 = 1$ permet de ne pas stocker b_0 et de reserver les $p = 23$ bits de la mantisse pour la partie décimale et d'augmenter ainsi la précision du codage de 2^{-22} à 2^{-23}. b_0 est appelé le bit caché de la normalisation. Cette technique, qui a été utilisée pour la première fois sur un ordinateur Vax dans les années 70, a aussi un inconvenient. Celui de ne pas pouvoir stocker de façon simple la valeur 0. En effet si nous suivons la procédure pour le codage flottant binaire, qui vient d'être décrite, on s'aperçoit qu'il faut poser $b_0 = 0$! Comme cette solution est inapplicable on conviendra que le nombre

0	00000000	00000000000000000000000

avec $b_0 = 1$ représente effectivement la valeur 0, mais, pour éviter toute ambiguïté, l'exposant 00000000 ne sera utilisé pour aucune autre représentation. Donc le plus petit nombre positif que nous pouvons stocker est

0	00000001	00000000000000000000000

ce qui donne $1.0 \times 2^{1-127} = 2^{-126}$ et qui signifie que le domaine de variation de l'exposant E n'est pas entre -127 et 127 mais entre -126 et 127.

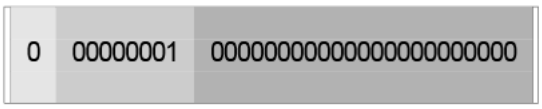
En utilisant les notations du tableau 2.1 on établit le tableau 2.1.

Bits de l'exposant $a_1 \cdots a_8$	La valeur numérique correspondante
$(00000000)_2 = 0$	$\pm (0.b_1 b_2 \cdots b_{23})_2 \times 2^{-127}$
$(00000001)_2 = 1$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = 2$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{-125}$
$\vdots$	$\vdots$
$(01111111)_2 = 127$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{0}$
$(10000000)_2 = 128$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{1}$
$\vdots$	$\vdots$
$(11111101)_2 = 253$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{126}$
$(11111110)_2 = 254$	$\pm (1.b_1 b_2 \cdots b_{23})_2 \times 2^{127}$
$(11111111)_2 = 255$	$\pm \infty$ si $b_1 = b_2 = \cdots = b_{23} = 0$, NaN sinon.

TABLE 1.2 – Nombres du standard IEEE-754, simple précision

Tous les nombres de ce tableau, sauf ceux de la première et de la dernière ligne, sont des nombres flottants binaires normalisés en simple précision. La première ligne indique le codage de la valeur 0. Notons qu'en fonction de la valeur du bit du signe (0 ou 1) nous avons deux zéros, un positif et un négatif. On peut aussi, à l'aide de ce tableau, calculer le plus petit et le plus grand nombre de la représentation en simple précision.

Le plus petit nombre normalisé peut être représenté par

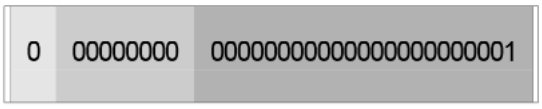


ce qui équivaut à $(1.000\ldots0)_2 = 1\times 10^{00000001} \approx 1.2\times 10^{-38}$. On le notera par la suite m_p.

Le plus grand nombre normalisé est représenté par

ce qui équivaut à $(1.11\ldots1)_2 \times 10^{11111110} = \ 2-2^{-23}\ \times 2^{127} \approx 3.4\times 10^{38}$. On le notera par la suite m_G.

Un examen attentif de cette table montre qu'il existe des nombres soit plus petits, soit plus grands que ceux qui sont normalisés. Ce sont justement les nombres spéciaux que seul l'ordinateur peut utiliser. Par exemple le plus petit nombre non nul qu'on peut stocker est $2^{-149} = [0.00\ldots01] = 0.00\ldots1\times 2^{00000001}$ dont la représentation, en suivant la technique pour la représentation de zéro, doit être



Ces nombres, qu'ils sont plus petits que le plus petit nombre normalisé, i.e. plus petits que 1.2×10^{-38}, sont appelés nombres sous-normalisés. Ils ne peuvent pas être normalisés car, dans ce cas, le résultat serait un exposant qui n'et pas dans le domaine de variation entre -126 et 127. Les nombres sous-normalisés sont moins précis que les nombres normalisés, c'est-à-dire que leur utilisation peut provoquer des erreurs de calcul plus importantes. Si un résultat d'une opération arithmétique conduit à un nombre sous-normalisé, on dit qu'on est en présence d'un underflow.

Nous pouvons aussi envisager des résultats des opérations arithmétiques qui sont supérieurs au plus grand nombre normalisé, à savoir 3.4×10^{38}. Dans ce cas tous les bits de l'exposant sont égaux à 1. Cette situation est décrite par dernière ligne du tableau 2.1. Deux cas peuvent se présenter : Soit la mantisse est nulle et dans ce cas on dit que la valeur représentée par ce flottant est l'infini – positif ou négatif, selon le bit du signe. Soit la mantisse n'est pas nulle et dans ce cas on est en présence d'un non-nombre, noté NaN pour Not a Number.

EN SUBSTANCE

Il y a cinq catégories de nombres flottants binaires selon les valeurs des bits de l'exposant et de la mantisse.

- ***Nombres normalisés : l'exposant a au moins un bit différent de 0 et aussi au moins un bit égal à 0.***
- ***Nombres sous-normalisés : L'exposant est nul et la mantisse a au moins un bit différent de 0.***
- ***Zéro : L'exposant et la mantisse sont nuls. Il y un zéro positif et un zéro négatif en fonction de la valeur du bit du signe.***
- ***Nan : L'exposant est composé exclusivement de 1 et la mantisse a au moins un bit différent de 0.***
- ***Infini : L'exposant est composé exclusivement de 1 et la mantisse de 0. Il y un infini positif et un négatif, en fonction de la valeur du bit du signe.***

1.4 Arithmétique arrondie

D'après ce qui vient d'être présenté aux deux sections précédentes, il s'ensuit que le résultat x d'une opération arithmétique avec des réels, peut rarement faire partie des nombres représentés par le tableau 2.1. Cinq cas peuvent se présenter :

(1) Le résultat est un des nombres normalisés du tableau 2.1.

(2) Le résultat x est entre m_p et m_G mais ne fait pas partie des nombres normalisés du tableau 2.1.

(3) Le résultat x est un nombre sous-normalisé.

(4) Le résultat x est une valeur infinie.

(5) Le résultat x est un NaN.

Les trois derniers cas ne nous intéressent pas. L'ordinateur indiquera le problème et, soit il poursuivra l'exécution du programme, soit il s'arrêtera. Le premier cas ne pose aucun problème. Nous examinerons donc le deuxième cas. Il est évident que pour pouvoir poursuivre l'exécution du programme, l'ordinateur assimilera le résultat x, dont la représentation binaire est $0.b_1b_2\cdots b_{22}b_{23}b_{24}\cdots$, à un des nombres normalisés qui sont présents dans la table 2.1. La raison pour laquelle x ne fait pas partie des nombres flottants normalisés est que sa représentation binaire a des bits b_i non nuls pour $i > p = 23$. L'opération donc de normalisation du nombre x consiste à trouver une représentation binaire avec 23 bits $0.b_1b_2\cdots b_{22}b_{23}$ avec

$$b_i = b_i\,;\ 1 \le i \le 22$$

Le 23-ème bit et, éventuellement, l'exposant E seront modifiés. Il y a quatre possibilités pour cette modification :

(1) Soit l'ordinateur ne fait aucune modification : $b_{23} = b_{23}$ et l'exposant reste inchangé.

(2) Soit il utilisera le nombre flottant x_- de la table 2.1 qui est le plus proche de x et qui est plus petit que x : $b_{23} = \begin{matrix} b_{23}, & \text{si } s = 0 \\ b_{23} + 1, & \text{si } s = 1 \end{matrix}$, avec modification de l'exposant dans le cas de propagation de la retenue.

(3) Soit il utilisera le nombre flottant x_+ de la table 2.1 qui est le plus proche de x et qui est plus grand que x : $b_{23} = \begin{cases} b_{23}+1 & \text{si } s=0 \\ b_{23}, & \text{si } s=1 \end{cases}$, avec modification de l'exposant dans le cas de propagation de la retenue.

(4) Soit il utilisera le nombre flottant $x_\pm$ de la table 2.1 qui est le plus proche de x : $b_{23} = b_{23} + b_{24}$ avec modification de l'exposant dans le cas de propagation de la retenue.

Cette opération s'appelle opération d'arrondi et le standard IEEE-754 préconise d'utiliser l'arrondi vers le nombre $x_\pm$ le plus proche de x qu'on appellera par la suite représentation par l'arrondi (le plus proche). Nous noterons par $fl(x)$ le résultat de cet arrondi et on aura, selon ce qui vient d'être dit, $|fl(x) - x| = \min\{|x_- - x|, |x_+ - x|\}$. Il y a des systèmes d'exploitation qui utilisent aussi la représentation par x_- qu'on appellera représentation par troncature.

Afin d'évaluer l'erreur de calcul provoquée par l'arrondi nous devons connaître la résolution de notre système de représentation des flottants normalisés ou, comme on dit, la précision de la machine.

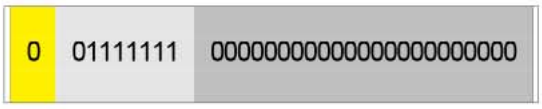
DÉFINITION 1.4.1 La précision eps d'un ordinateur est le plus petit nombre positif normalisé tel que

$$1 + eps = 1$$

On peut aussi envisager la précision de la machine comme étant la plus grande erreur relative de précision qui peut arriver lors de la représentation par troncature d'un nombre réel.

Il ne faut pas confondre ce nombre avec le plus petit nombre flottant positif que l'ordinateur peut représenter qui, comme nous avons vu précédemment, est un nombre flottant sous-normalisé.

Pour calculer la précision de la machine on commence par la représentation binaire normalisé, avec le bit caché, de la valeur $1 = (1.00\ldots0)_2 \times 10^0$. On a donc



ce qui montre que ce nombre fait partie du tableau 2.1. En utilisant cette table on peut représenter le nombre normalisé qui est immédiatement supérieur à 1. Il est donné par

qui équivaut en décimal à $1 + 2^{-23}$. La différence entre ce deux nombres est égale à 2^{-23} qui est la précision de la machine. Nous avons donc pour la précision de la machine le fait suivant :

FAIT 1.1 La précision eps d'une machine simple précision qui suit la norme IEEE-754 est

$$\text{eps} = 2^{-23} \approx 1.192 \times 10^{-7}$$

Nous pouvons maintenant avoir une idée concernant l'erreur d'arrondi. Si x est la valeur qu'on veut représenter selon le standard IEEE-754, alors on la remplacera soit par x_-, soit par x_+. Cette représentation est notée $fl(x)$. La valeur absolue de la différence entre $fl(x)$ et x n'est pas supérieure à la moitié de la différence entre x_+ et x_-. Ainsi, si le nombre x s'écrit en binaire

$$x = (b_0.b_1b_2 \ldots b_{23}b_{24} \cdots)_2 \times 2^E$$

avec $b_0 = 1$, alors sa représentation machine sera

$$fl(x) = (b_0.b_1b_2 \ldots b_{23})_2 \times 10^E$$

Donc

$$|fl(x) - x| \leq 2^{-24} \times 10^E$$

ce qui donne

$$|fl(x) - x| \leq \frac{1}{2}\text{eps} \times 10^E$$

EN SUBSTANCE

- ***Il y a quatre types de nombres flottants binaires : nombres normalisés, sous-normalisés, infini et NaN.***
- ***À tout nombre réel correspond un nombre machine, noté*** $fl(x)$ ***et qui, pour le standard IEEE-754, est donné par l'approximation*** $fl(x) = x_\pm$.
- ***La précision*** eps ***de la machine est le plus petit nombre normalisé tel que*** $1 + \text{eps} = 1$. ***Pour la simple précision on a*** $\text{eps} = 2^{-23} \approx 1.192 \times 10^{-7}$.

1.4.1 Exercices

EXERCICE 1.1 Soit un système de représentation flottant avec base β, mantisse à p places, ex^posant E, avec $E_m \leq E \leq E_M$.

Calculer le nombre de valeurs normalisées qui peuvent être représntées par ce système.

Application : $\beta = 10, p = 3, E_m = -15, E_M = 16$.

EXERCICE 1.2 Soient trois réels $x = 0.125 \times 10^6$, $y = 0.437 \times 10^{12}$, $z = 0.215 \times 10^{-10}$. En utilisant le système de représentation de l'exercice précédent, calculer

(1) la somme x + y et commenter le résultat.

(2) le produit x × y et commenter le résultat.

(3) le quotient z/ y et commenter le résultat.

EXERCICE 1.3 Soient trois réels $x = 0.4\times 10^{0}$, $y = 0.4\times 10^{0}$, $z = 0.1\times 10^{3}$. En utilisant le système de représentation de l'exercice précédent, calculer les deux sommes

$(x + y) + z$ et $x + (y + z)$

Conclusion.

1.5 Les nombres réels et les nombres-machine

Pour l'étude des différences entre un réel x et le nombre-machine $fl(x)$ qui est sa représentation dans l'ordinateur, nous utilisons trois types d'erreurs :

– Erreur de représentation, notée Δx ou $\varepsilon(x)$:

$$\Delta x = fl(x) - x \tag{1.5.1}$$

ou encore l'erreur absolue de représentation :

$$|\Delta x| = |fl(x) - x|$$

– Erreur relative de représentation, notée $\iota(x)$:

$$\iota(x) = \frac{\Delta x}{fl(x)} = \frac{fl(x) - x}{fl(x)} \tag{1.5.2}$$

– Erreur relative de précision, notée $\eta(x)$:

$$\eta(x) = \frac{\Delta x}{x} = \frac{fl(x) - x}{x} = \frac{fl(x)}{x} - 1 \tag{1.5.3}$$

ou encore l'erreur relative absolue de précision

$$|\eta(x)| = \frac{|\Delta x|}{|x|} = \frac{|fl(x) - x|}{|x|}$$

L'erreur relative de précision permet d'exprimer le nombre-machine comme suit :

$$fl(x) = x(1 + \eta(x)) \tag{1.5.4}$$

1.5.1 Étude de l'erreur de précision relative

Nous avons vu qu'n ordinateur utilise pour la représentation des valeurs réelles un mot machine dont les bits sont partagés en

– un bit de signe,
– p bits de mantisse, et
– q bits d'exposant.

Dans la suite on considère que la base de numérotation de l'ordinateur est β.

EXEMPLE 1.5.1 Considérons un ordinateur avec base de numérotation $\beta = 10$ et nombre de chiffres de la mantisse $p = 4$. Nous avons $123.4567 = 0.1234567\times 10^{3} = 0.1234 + 0.567\times 10^{-4}\times 10^{3}$. Si l'ordinateur approche les valeurs réelles par troncature, alors $m(x) = 0.1234\times 10^{3}$. Si l'approximation se fait par arrondi, alors $m(x) = 0.1235\times 10^{3}$.

Cet exemple nous aide à comprendre que si nous avons un réel x, alors il sera représenté sous forme normalisée comme suit :

$$x = s \times w \times \beta^n \tag{1.5.5}$$

où s est le signe. On peut, encore écrire :

$$x = s \times \left(r + t \times \beta^{-p} \right) \times \beta^n \text{ , avec } \beta^{-1} \leq |r| < 1 \text{ et } 0 \leq |t| < 1 \tag{1.5.6}$$

Si le nombre-machine représentant x est obtenu par troncature, nous avons

$$fl(x) = s \times r \times \beta^n \tag{1.5.7}$$

tandis que pour approximation par arrondi, nous avons

$$fl(x) = \begin{array}{ll} s \times r \times \beta^n, & \text{si } |x - r \times \beta^n| < |x - \left(r \times \beta^n + \beta^{-p} \right)| \\ s \times r \times \beta^n + \beta^{-p}, & \text{sinon} \end{array}$$

Nous avons, pour l'erreur relative de précision, le résultat suivant :

THÉORÈME 1.5.1 (de la précision relative).- Soit un calculateur avec base de numération β et nombre de chiffres de la mantisse p. Alors l'erreur relative de précision $\eta(x)$ est bornée comme suit :

$$|\eta(x)| \leq \begin{array}{ll} \beta^{1-p}, & \text{si approximation par troncature} \\ 0.5 \times \beta^{1-p}, & \text{si approximation par arrondi} \end{array} \quad ; x \in R \tag{1.5.8}$$

DÉMONSTRATION. D'après (1.5.6) et (1.5.7) on a

$$\Delta x = fl(x) - x = t \times \beta^{-p} \times \beta^n \tag{1.5.9}$$

d'où

$$|\eta(x)| = \frac{|\Delta x|}{|x|} = \frac{|t \times \beta^{-p} \times \beta^n|}{|r + t \times \beta^{-p} \times \beta^n|} = \frac{|t \times \beta^{-p}|}{|r + t \times \beta^{-p}|} \leq \frac{|t|}{|r|} \times \beta^{-p} \leq \beta^{1-p} \tag{1.5.10}$$

car $\beta^{-1} \leq |r| < 1$, $0 \leq |t| < 1$ et $\frac{|t|}{|r|} \leq \frac{1}{\beta^{-1}} = \beta$ dans le cas de troncature.

Dans le cas de l'arrondi, nous avons $0 \leq |t| < 1/2$ et donc $\frac{|t|}{|r|} \leq \frac{1}{2}\beta^{1-p}$. ∎

Dans les calculs on utilise la valeur absolue de l'erreur de précision relative $|\eta(x)|$.

Parfois cette quantité est considérée comme étant la précision de la machine.

EN SUBSTANCE

- ***Il y a trois types d'erreur de représentation :***
  - ***Erreur de représentation :*** $\Delta x = fl(x) - x$
  - ***Erreur relative de représentation :*** $\iota(x) = \dfrac{\Delta x}{fl(x)} = \dfrac{fl(x) - x}{fl(x)}$
  - ***Erreur relative de précision :*** $\eta(x) = \dfrac{\Delta x}{x} = \dfrac{fl(x) - x}{x} = \dfrac{fl(x)}{x} - 1$, ***avec*** $fl(x) = x(1 + \eta(x))$.
- ***En règle générale, pour l'analyse des erreurs on utilise la valeur absolue de l'erreur de précision relative, qui pour le standard IEEE-754 est***

$$\eta(x) \leq \begin{cases} \beta^{1-p}, & \textit{si approximation par troncature} \\ 0.5 \times \beta^{1-p}, & \textit{si approximation par arrondi} \end{cases}$$

1.5.2 Exercices

EXERCICE 1.4 Soit le nombre 387.62000_{10}.

(1) Calculer sa valeur en hexadécimal.

(2) Convertir la valeur hexadécimale en décimale.

(3) Calculer l'erreur absolue de représentation.

(4) Calculer l'erreur relative absolue de précision.

EXERCICE 1.5 Considérons une machine décimale avec mantisse à 4 chiffres. Calculer l'erreur de représentation et l'erreur relative de représentation pour les nombres suivants :

9.023506, 158.26 et 0.001588946.

EXERCICE 1.6 Soient les nombres : 15.2750, 358.937 et 5233.618.

(1) Convertir ces nombres en hexadécimal.

(2) Convertir les nombres hexadécimaux ainsi obtenus en nombres décimaux avec exactitude de 8.

(3) Calculer l'erreur relative de cette dernière conversion sous la forme $x.xx \times 10^{-7}$.

N.B. Pour la conversion en héxadécimal on utilisera le même nombre de chiffres héxadécimaux après la virgule qu'en valeur décimale.

EXERCICE 1.7 Soit un nombre réel $x \in R$. Nous pouvons l'écrire sous la forme :

$$x = s \cdot m \cdot \beta^e, \text{ avec } \frac{1}{b} \leq m < 1$$

où s est le signe, m est la mantisse considérée sans limitation de bits, β est la base et e est l'exposant qui est une valeur entière. Dans un ordinateur, x sera représenté comme un flottant normalisé sous la forme

$$fl(x) = s \cdot M \cdot \beta^E, \text{ avec } fl(x) \in M(R)$$

où M est la mantisse limitée à p digits, E est l'exposant limité à q digits. Ainsi pour le codage en nombre flottants, on utilise $N = p + q + 1$ digits.

Dans la suite, on prendra $\beta = 2$.

On se propose de calculer l'erreur de la représentaion pour différentes opérations arithmétiques.

(1) Erreur de l'affectation : Montrer que

$$x - fl(x) = s \cdot 2^{E-p} \cdot \alpha$$

avec

(a) $\alpha \in [-0.5, 0.5[$ si représentation par arrondi.

(b) $\alpha \in [0, 1[$ si représentation par troncature.

(2) Calculer l'erreur d'une opération d'addition.

(3) Même chose pour l'opération de soustraction.

(4) Même chose pour l'opération de multiplication.

(5) Même chose pour l'opération de division.

2

Le standard IEEE 754

Les développements du chapitre précédent, nous permettent de conclure qu'un ordinateur tente de représenter l'infini (les nombres réels R) par des moyens finis (les nombres-machine M). Cette représenation peut se faire de plusieurs manières et au début de l'informatique c'éatit effectivement le cas : avec presque chaque ordinateur, on s'ingénuait à faire une présentation particulière. Bien sûr la comparaison des résultats numériques entre différents ordinateurs dévenait impossible et la portabilité des programmes faisait appel plus à la chance qu'à la raison. Celle situation a pris fin en 1985 avec l'apparition du standard IEEE-754 qui équipe aujourd'hui la quasi totamité des ordinateurs, au point de parler d'une norme IEEE-754 et non pas d'un standard. Pour établir ce standard il a fallu choisir, entre autre, le type du codage, les opérations à utiliser et la méthode d'arrondi.

Ainsi le standard IEE-754 :

- code en virgule flottante avec bit caché ;
- permet le calcul des quatre opérations arithmétiques et de l'extraction de la racine carrée ;
- préconise l'utilisation de l'arrondi au plus près, mais il accepte les quatre modes d'arrondi, présentés en ??, et
- considère chaque résultat comme exact, c'est-à-dire n'effectue pas la propagation des erreurs sur des calculs qui se suivent [1].

Nous présentons dans la suite de manière détaillée le standard IEEE-754.

1. ce qui ne signifie pas que, lors d'une suite des calculs, il n'y a pas de propagation d'erreurs due aux arrondis, mais simplement que, si on a fait un premier calcul, le calcul suivant considérera le résultat du premier comme exact et il ne tiendra pas compte, pour le nouveau calcul, de l'erreur d'arrondi du premier.

2.1 Formats de IEEE-754

Selon la relation (1.5.6), la représentation d'un nombre flottant est donnée par

$$x = s \times M \times \beta^E \tag{2.1.1}$$

où s est le signe, M est la mantisse normalisée stockée sur p bits, β est la base et E est l'exposant qui est un nombre entier non négatif stocké sur q bits. La prise en compte de cette représentation par un ordinateur numérique, impose que :

- la mantisse normalisée ait une précision de $p-1$ bits[2], et
- la valeur de E se trouve dans un intervalle $[E_{min}, E_{max}]$ déterminé par le nombre de bits q de l'exposant.

Concrètement un réel x peut se mettre sous la forme

$$x = (-1)^s \times \quad r + t \times \beta^{-(p+1)} \quad \times \beta^E \quad , \text{ où } 1 \leq r < \beta \text{ et } 0 \leq t < 1$$

En numérotation binaire (c'est-à-dire $\beta = 2$) le réel x s'écrit

$$(-1)^s \times b_0.b_1 b_2 \ldots b_{p-1} \times \beta^E$$

avec

- $s = 0$ ou 1 ;
- $b_0 = 1$;
- $b_i = 0$ ou $1, i = 1, \ldots, p-1$;
- $\beta = 2$, la base et
- E l'exposant, avec $E \in [E_{min}, E_{max}]$.

Comme le standard IEEE-754 utilise le codage avec bit caché, le réel x s'écrit selon ce standard

$$(-1)^s \times .b_1 b_2 \ldots b_{p-1} \times \beta^E$$

Pour calculer l'intervalle de variation $[E_{min}, E_{max}]$ de la valeur de l'exposant on utilise le biais qui est défini par la relation :

$$\text{biais} = B = 0\underbrace{1\ldots1}_{(q-1)\text{fois}} = 2^{(q-1)} - 1 \tag{2.1.2}$$

Donc le calcul en binaire de l'exposant se fait en rajoutant le biais à la valeur de la conversion en binaire de E, i.e. contenu de l'exposant = biais + $(E)_2$, où $(E)_2$ signifie la valeur E en base de numérotation 2.

La table 2.1 de la page suivante présente les différents formats du standard IEEE-754

L'application des formats est différente selon le langage de programmation. Nous donnons dans la table 1.2 les déclarations à faire selon le format choisi :

2. En effet selon le standard IEEE-754 p est le nombre de bits pour la mantisse et le signe. Étant donné que le signe occupe un bit dans tous les formats, il reste pour la mantisse $p-1$ bits.

Paramètre	Simple précision	Simple précision étendue	Double précision	Double précision étendue	Quadruple (Extension (du standard)	Étendue (Introduite (par Intel)
Nombre de bits de la mantisse plus le signe (p)	24	≥ 32	53	≥ 64	113	64
Nombre de chiffres décimaux exacts ($p/\log_2 10$)	7.22	≥ 9.63	15.95	≥ 19.26	34.01	19.26
Type du codage	Bit caché	Non spécifié	Bit caché	Non spécifié	Bit caché	Bit explicite
Nombre de bits pour la mantisse ($p-1$)	23	≥ 31	52	≥ 63	112	64
E_{max} E_{min}	+127 -126	≥ +1023 ≤ -1022	+1023 -1022	≥ +16383 ≤ -16382	+16383 -16382	+16383 -16382
Biais de l'exposant	+127	Non spécifié	+1023	Non spécifié	+16383	+16383
Nombre de bits pour l'exposant	8	≥ 11	11	≥ 15	15	15
Nombre de bits pour le signe	1	1	1	1	1	1
Nombre de bits pour le format	32	≥ 43	64	≥ 79	128	80
Valeur max décimale ($2E_{max}+1$)	3.4028E+38	≥ 1.7976E+308	1.7976E+308	≥ 1.1897E+4932	1.1897E+4932	1.1897E+4932
Valeur min décimale ($2E_{min}$)	1.1754E-38	≤ 2.2250E-308	2.2250E-308	≤ 3.3621E-4932	3.3621E-4932	3.3621E-4932
Valeur min décimale dénormalisée ($2E_{min}-p+1$)	1.4012E-45	≤ 1.0361E-317	4.9406E-324	≤ 3.6451E-4951	6.4751E-4966	1.8225E-4951

TABLE 2.3 – Les formats du standard IEEE-754

Langage	Simple précision	Double précision	Double étendue	Quadruple précision	Etendue
SCILAb		Par défaut	En interne 80 bits		
FORTRAN	`REAL*4`	`REAL*8`		`REAL*16`	`REAL*10`
C, C++	`float`	`double`		`long double`	`long double`

TABLE 2.4 – Déclarations des formats selon le langage de programmation

2.1.1 Format simple précision

Les caractéristiques de ce format sont les suivantes :

Les exposants `00000000` et `11111111` sont à usage reservé et ne sont pas utilisés pour la conversion. Par conséquent nous avons $E_{min} = -\text{biais}+1$ et $E_{max} = \text{biais}$.

Un mot de 32 bits pour lequel tous ses bits sont à 0, représente la valeur décimale 0,0 . Il est évident que, en fonction de la valeur du signe s, on a un zéro positif et un zéro négatif.

Pour l'infini nous avons les représentations suivantes :

– `0 11111111 0000000 00000000000000000` $= +\infty = 7F800000_{16}$

– 1 11111111 0000000 00000000000000000 = $-\infty$ = $FF800000_{16}$.

Tous les nombres avec représentation

s 11111111 fffffffffffffffffffffff

s'appellent NaN – Not a Number – s'il y a au moins un f qui est différent de zéro. Ce sont toujours des résultats indéfinis produits par une erreur de calcul, e.g. du type 0/0 ou log(0). En particulier nous avons les deux NaN suivants :

– Not-a-Number-Signaling (NaNS) dont la représentation est :
  0 11111111 0111111 11111111111111111 = NaNS = $7FBFFFFF_{16}$.
– Not-a-Number-Quiet (NaNQ) dont la représentation est :
  0 11111111 1000000 00000000000000000 = NaNQ = $7FC00000_{16}$.

NaNS engendre un arrêt anormal (trap) de l'exécution du programme tandis que NaNQ permet la poursuite de l'exécution du programme. Tout calcul numérique avec NaN comme entrée produit NaN comme sortie, c'est-à-dire nous avons, par exemple, $0 \times NaN = NaN$.

Nous avons donc pour un nombre décimal V, la représentation flottante suivante en simple précision :

s xxxxxxxx fffffffffffffffffffffff = s E F

où s représent le bit du signe, E représent l'exposant biaisé et F la fraction décimale. Cette valeur est déterminée comme suit :

– Si E = 255 et F = 0, alors V = NaN.
– Si E = 255 et F = 0 et s=0, alors V = ∞ .
– Si E = 255 et F = 0 et s=1, alors V = $-\infty$.
– Si E = 0 et F = 0 et s=0, alors V = 0.
– Si E = 0 et F = 0 et s=1, alors V = -0.
– Si 0 < E < 255, alors $V = (-1)^s \times 2^{X-127} \times (1.F)$
– Si E=0 et F = 0,, alors $V = (-1)^s \times 2^{-126} \times (1.F)$

Le dernier cas représente ce que nous appelons des nombres sous-normalisés auxquels nous n'avons pas accès mais qui sont utilisés de façon interne pour les calculs.

La table 1.3 présente un sommaire du codage en simple précision.

2.2 Exercices

EXERCICE 2.1 Calculer pour le standard IEEE-754 en simple précision

(1) le biais ;

(2) les valeurs E_{min} et E_{max} ;

(3) la valeur de la plus grande mantisse ;

(4) le plus grand nombre que nous pouvons représenter ;

(5) le plus petit nombre positif que nous pouvons représenter ;

(6) la précision des valeurs numériques représentées par ce standard.

EXERCICE 2.2 Calculer la représentation au standard IEEE-754 simple précision des flottants 5.75 et 0.1. Évaluer l'erreur de représentation.

Nom	Signe (s) 1 [31]	Exposant (E) 8 [30-23]	Mantisse (m) 23 [22-0]	Intervalle en héxa	Intervalle	Intervalle décimal
-Nan Quiet	1	11...11	11...11 ⋮ 10...01	FFFFFFFF ⋮ FFC00001		
Indéterminé	1	11...11	10...00	FFC00000		
-Nan Signalé	1	11...11	01..11 ⋮ 00...01	FFBFFFFF ⋮ FF800001		
-Infinity (Overflow négatif)	1	11...11	00...00	FF800000	$< -(2-2^{-23})\times 2^{127}$	$\leq -3.4028235677973365E+38$
Normalisé négatif $-1.m\times 2^{(E-127)}$	1	11...10 ⋮ 00...01	11...11 ⋮ 00...00	FF7FFFFF ⋮ 80800000	$-(2-2^{-23})\times 2^{127}$ -2^{-126}	-3.4028234663852886E+38 -1.1754943508222875E-38
Dénormalisé négatif $-0.m\times 2^{-126}$	1	00...00	11...11 ⋮ 00...01	807FFFFF ⋮ 80000001	$-(1-2^{-23})\times 2^{-126}$ $-(1+2^{-52})\times 2^{-150}$ $=-2^{-149}/2$	-1.1754942106924411E-38 -7.0064923216240862E-46 = -1.4012984643248170E-45
Underflow négatif	1	00...00	00...00	80000000	2^{-150} ⋮ <-0	-7.0064923216240861E-46 <-0
-0	1	00...00	00...00	80000000	-0	-0
0	0	00...00	00...00	00000000	0	0
Underflow positif	0	00...00	00...00	00000000	>0 ⋮ 2^{-150}	>0 ⋮ 7.0064923216240861E-46
Dénormalisé positif $0.m\times 2^{-126}$	0	00...01	00...01 ⋮ 11...11	00000001 ⋮ 007FFFFF	$(1+2^{-52})\times 2^{-150}$ $=2^{-149}/2$ $(1-2^{-23}\times 2^{-126}$	7.0064923216240862E-46 = 1.4012984643248170E-45 1.1754942106924411E-38
Normalisé positif $1.m\times 2^{e-127}$	0	00...01 ⋮ 11...10	00...00 ⋮ 11...11	00800000 ⋮ 7F7FFFFF	2-126 ⋮ ⋮	1.1754943508222875E-38 ⋮ 3.4028234663852886E+38
+Infini (Overflow positif)	0	11...11	00...00	7F800000	$>(2-2^{-23})\times 2^{127}$	$\geq 3.4028235677973365E+38$
+NaN signalé	0	11...11	00...01 ⋮ 01...11	7F800001 ⋮ 7FBFFFFF		
+NaN quiet	0	11...11	10...00 ⋮ 11...11	7FC00000 ⋮ 7FFFFFFF		

TABLE 2.5 – Simple précision

EXERCICE 2.3 Répéter pour la double précision les calculs de l'exc. 2.1.

EXERCICE 2.4 Répéter pour la double précision les calculs de l'exc. 2.2.

EXERCICE 2.5 Considérons un système de représentation binaire pour des nombres réels non négatifs avec trois bits pour l'exposant et trois bits pour la mantisse.

(1) Présenter les différents exposants.

(2) évaluer les valeurs binaires et leur équivalent en décimal de toutes les mantisses indépendamment de l'exposant et du bit caché.

(3) Calculer le plus petit et le plus grand nombre que nous pouvons stocker.

(4) évaluer la précision de la machine.

(5) Calculer l'intervalle de la droite des réels qui peut être représenté par ce système et faire un diagramme de la distribution des nombres normalisés flottants de cet intervalle. Commentaires.

(6) Répétez la même chose pour les nombres sous-normalisés. Commentaires.

(7) Quel est le nombre de valeurs flottantes différentes que nous pouvons représenter par ce système? Généraliser.

(8) Comparer, en utilisant cet exemple, le système de représentation en virgule flottante avec celui en virgule fixe et établir les avantages et les inconvénients de chaque système.

2.3 Exercices de laboratoire

Pour résoudre les exercices suivants, il faut utiliser la version 4.4 de Scilab qu'il faut télécharger du site de ScicosLab : `http://www.scicoslab.org`.

EXERCICE 2.6 Écrire un programme en Scilab qui permet de calculer la précision eps de votre ordinateur.

EXERCICE 2.7 Écrire un programme en Scilab qui permet d'évaluer

(1) le nombre de places p de la mantisse et

(2) la base de numérotation

de votre ordinateur.

EXERCICE 2.8 (D'APRÈS WALTER GANDER) Voici deux programmes qui calculent le plus petit nombre positif normalisé

```
clear();
format("e",22);

  x = 1;
  t = x;
  while %eps *  t/2 > 0
    t = %eps * t /2;
    if (t >0) then
      x = t;
    end;
  end;
  x
```

et

```
clear();
format("e",22);
  t = 1;
  while %eps *  t > 0
    ta = t;
    t = t/2;
  end;
  t = ta
```

Le premier fournit le résultat `x = 8.094771541462983-320` qui est faux, tandis que le second donne la bonne réponse `t = 2.225073858507201-308` qui est la réponse correcte.

Expliquer les raisons de ces résultats.

3

PROPAGATION DES ERREURS

Dans les chapitres précédents nous avons examiné les différentes représentations des nombres réels par un ordinateur et nous avons en particulier étudier le standard IEEE-754. Du fait que les nombres-machine $M(R)$ constituent sous-ensemble fini de R, un nombre réel $x \in R$ quelconque ne correspond pas en général avec sa représentation $fl(x) \in M(R)$. Il s'ensuit un erreur de représentation $\Delta x = fl(x) - x$, de la part de l'ordinateur, concernant la valeur de x.

Dans ce chapitre, nous allons étudier les repercussions de ces erreurs de représentation aux résultats finals des calculs lors de l'exécution du code d'un algorithme par un ordinateur.

3.1 Sources des erreurs

Pour réaliser un projet technique, nous avons, d'abord, besoin de préciser le système physique qui est l'objet du projet.[1] Ensuite pour pouvoir commander ce système ou anticiper son comportement, nous devons procéder à une modélisation mathématique. Soit donc Σ une mesure lue à l'aide d'un capteur (supposé fournir des valeurs exactes) et caractéristique d'une grandeur du système. Notons par S la valeur correspondante de Σ obtenue par le modèle mathématique à l'aide d'une formule analytique. Il est improbable que les deux valeurs soient identiques. Car le modèle mathématique est, en règle générale, une idéalisation de la réalité physique. Ici il faut comprendre le mot "idéalisation" comme étant une simplification. Par exemple on néglige,

1. Parfois un projet contient plusieurs systèmes physiques.

dans certains cas, la résistance de l'air. Nous avons donc une première source d'erreurs qui sont des erreurs de la modélisation mathématique.

La valeur S est obtenue après un calcul d'une formule analytique, par conséquent elle est une solution exacte. Néanmoins, il est rare que cette formule analytique soit utilisée telle quelle pour obtenir les valeurs S car il est très difficile à calculer. On établit donc d'autres formules, plus faciles à calculer, et qui sont des approximations de la formule initiale. Par exemple, un système avec un comportement dynamique non linéaire sur une plage de fonctionnement donnée, on peut le résoudre en linéarisant son comportement et en faisant des calculs successifs sur des plages de fonctionnement de largeur réduite. On obtient ainsi une solution S_a qui est une approximation de la solution mathématique exacte S. Nous avons donc un deuxième source d'erreurs qui sont des erreurs d'approximation.

Si nous voulons utiliser un ordinateur pour calculer S_a, il faut décrire de façon systématique la démarche du calcul de S_a à l'aide d'un algorithme et en tenant compte du fonctionnement discret de l'ordinateur. Ce qui signifie d'une part que les formules contenant des valeurs continues doivent se transformer en formules discrètes et, d'autre part, la convergence d'une quantité vers une autre doit se comprendre comme étant une différence inférieure à une seuil petit certes, mains non infinitésimal. La solution fournie par l'algorithme est S_{ad}, différente de S_a. Nous sommes ainsi en présence d'une troisième source d'erreurs qui sont des erreurs de discrétisation.

Pour que l'algorithme que nous avons élaborer, puisse être introduit dans un ordinateur à des fins de calcul, il faut le coder, c'est-à-dire le traduire dans un langage de programmation. L'exécution du code par l'ordinateur donnera un résultat $fl(S_{ad})$ qui sera entachée des erreurs de calcul numérique. Nous avons donc une quatrième source d'erreurs, qui sont les erreurs du calcul numérique. Ces erreurs dépendent bien sûr de l'ordinateur, mais dépendent aussi, et pour beaucoup, du codage réalisé, à savoir que, selon le codage, l'erreur numérique peut être plus ou moins importante.

La figure suivante resume le passage d'un système physique à sa simulation par ordinateur.

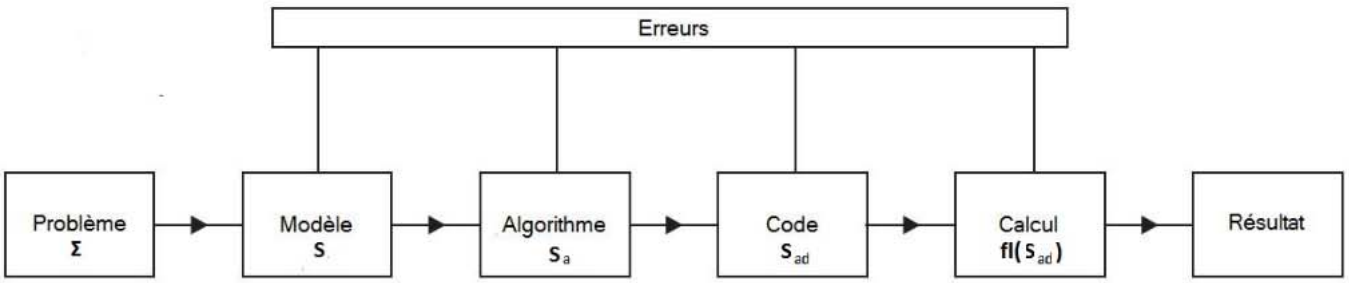


FIGURE 3.1 – Du système physique au résultat de la simulation

En analyse numérique on se préoccupe essentiellement des erreurs du calcul numérique et aussi des erreurs de discrétisation.

Dans la suite nous allons présenter les erreurs numériques lors d'une opération arithmétique isolée. Ensuite on abordera la propagation des erreurs numériques quand on réalise plusieurs calculs successivement.

EN SUBSTANCE

Sources d'erreurs

- ***Modélisation mathématique du système physique***
- ***Approximation des fonctions analytiques du modèle mathématique.***
- ***Discrétisation des fonctions obtenues par approximation des fonctions analytiques, afin qu'elles soient traitées par ordinateur.***
- ***Calcul numérique par ordinateur.***

L'analyse numérique étudie la dernière source d'erreurs et aussi l'avant-dernière.

3.2 Erreurs lors des opérations arithmétiques

Nous allons examiner la qualité du résultat d'une opération arithmétique c'est-à-dire soit d'une addition, soit d'une soustraction, soit d'une multiplication, soit d'une division ou encore d'une extraction de la racine carrée.

FAIT 3.1 Le résultat d'une opération arithmétique $\otimes$ entre deux valeurs réelles a et b, est entaché d'une erreur qui est due d'une part aux erreurs de représentation des nombres a et b et, d'autre part, à l'erreur de représentation du résultat, c'est-à-dire :

$$\text{résultat de l'opération } a \otimes b = fl(fl(a) \otimes fl(b)) \qquad (3.2.1)$$

On peut décomposer cette erreur en deux parties :

- Une erreur d'entrée, due à l'erreur de représentation du résultat de l'opération, notée $\eta^I(a \otimes b)$.
- Une erreur due au calcul et qui sera notée $\eta^C(a \otimes b)$.

L'erreur du calcul est inhérente à la machine et à l'opération et nous ne pouvons pas modifier sa valeur. Par contre l'erreur d'entrée dépend de l'ordre des calculs et, par conséquent, pour chaque série de calculs il y a un ordonnancement de ces calculs qui conduit à une erreur d'entrée minimale.

Les théorèmes suivants précisent les erreurs d'entrée pour les opérations arithmétiques usuelles.

THÉORÈME 3.2.1 (de l'erreur pour les sommes) .- Soit la somme

$$S = x_1 + x_2 + \cdots \tag{3.2.2}$$

L'erreur absolue de l'entrée pour cette somme est

$$\Delta^I S = \Delta^I x_1 + \Delta^I x_2 + \cdots \tag{3.2.3}$$

et l'erreur de précision de l'entrée est

$$\eta^I(S) = \frac{x_1}{S}\eta^I(x_1) + \frac{x_2}{S}\eta^I(x_2) + \cdots \tag{3.2.4}$$

DÉMONSTRATION. D'après (3.2.2) nous avons

$$m(S) = S + \Delta^I S + \Delta^C S = m(x_1) + m(x_2) + \cdots + \Delta^C S = x_1 + \Delta^I x_1 + x_2 + \Delta^I x_2 + \cdots + \Delta^C S$$

d'où on obtient (3.2.3). D'après (1.5.3) nous avons

$$\Delta x = x \cdot \eta(x)$$

et (3.2.3) donne :

$$S \cdot \eta^I(S) = x_1\eta^I(x_1) + x_2\eta^I(x_2) + \cdots$$

d'où on obtient (3.2.4). ∎

En Analyse Numérique on s'intéresse à l'erreur maximale, i.e. à la situation qui arrive quand toutes les erreurs se rajoutent. On essaie donc de borner l'erreur d'entrée afin d'obtenir l'erreur maximale. Nous avons dans le cas de la somme :

$$|\eta^I(S)| \le |\frac{x_1}{S}||\eta^I(x_1)| + |\frac{x_2}{S}||\eta^I(x_2)| + \cdots \le \frac{1}{|S|}(|x_1| + |x_2| + \cdots)\cdot eps$$

d'où finalement

$$|\eta^I(S)| \le \frac{\sum_i |x_i|}{|\sum_i x_i|} \times eps \tag{3.2.5}$$

Cette borne montre que si $|\sum_i x_i|$ est petit par rapport à $\sum_i |x_i| \cdot eps$, l'erreur de précision de l'entrée peut devenir importante.

THÉORÈME 3.2.2 (de l'erreur pour le produit).- Soit le produit

$$P = x_1 \cdot x_2 \cdot \ldots \quad \text{avec } x_i = 0\ \forall i \tag{3.2.6}$$

L'erreur absolue de l'entrée est

$$\Delta^I P = P \cdot \left(\frac{\Delta^I x_1}{x_1} + \frac{\Delta^I x_2}{x_2} + \cdots\right) \tag{3.2.7}$$

et l'erreur de précision de l'entrée est

$$\eta^I(P) = \eta^I(x_1) + \eta^I(x_2) + \cdots \tag{3.2.8}$$

DÉMONSTRATION. D'après (3.2.2) nous avons

$$\begin{aligned} m(P) = P + \Delta^I P + \Delta^C P &= m(x_1)\cdot m(x_2)\cdot\ldots + \Delta^C P \\ &= x_1 + \Delta^I x_1 \cdot x_2 + \Delta^I x_2 \cdot\ldots + \Delta^C P \\ &= x_1\cdot x_2\cdot x_3\cdot\ldots + \Delta^I x_1 (x_2\cdot x_3\cdot\ldots) + \Delta^I x_2 (x_1\cdot x_3\cdot\ldots) + \cdots \\ &\quad + \Delta^I x_1 \Delta^I x_2 (x_3\cdot\ldots) + \Delta^I x_2 \Delta^I x_3 (x_1\cdot\ldots) + \cdots \\ &\quad + \Delta^I x_1\cdot\Delta^I x_2\cdot\Delta^I x_3\cdot\ldots + \Delta^C P \end{aligned}$$

Si on néglige les termes qui ont comme facteurs les produits $\Delta^I x_1 \Delta^I x_2, \ldots; \Delta^I x_1 \Delta^I x_2 \Delta^I x_3, \ldots;$ $\Delta^I x_1\cdot\Delta^I x_2\cdot\Delta^I x_3\cdot\ldots$, on obtient

$$\begin{aligned} \Delta^I P &= \Delta^I x_1 (x_2\cdot x_3\cdot\ldots) + \Delta^I x_2 (x_1\cdot x_3\cdot\ldots) + \cdots \\ &= \tfrac{\Delta^I x_1}{x_1} (x_1\cdot x_2\cdot x_3\cdot\ldots) + \tfrac{\Delta^I x_2}{x_2} (x_1\cdot x_2\cdot x_3\cdot\ldots) + \cdots \end{aligned}$$

d'où on a (3.2.7). De cette dernière on a

$$\frac{\Delta^I P}{P} = \frac{\Delta^I x_1}{x_1} + \frac{\Delta^I x_2}{x_2} + \cdots$$

qui donne (3.2.8). ∎

De ce théorème nous avons

$$\eta^I(P) \le N\cdot eps$$

où N est le nombre de facteurs dans le produit P.

THÉORÈME 3.2.3 (de l'erreur pour la division).- Soit

$$Q = \frac{a}{b};\ b = 0$$

L'erreur absolue de l'entrée est

$$\Delta^I Q = \frac{b\Delta^I a - a\Delta^I b}{b^2} \tag{3.2.9}$$

et l'erreur de précision à l'entrée est

$$\eta^I(Q) = \eta^I(a) - \eta^I(b) \tag{3.2.10}$$

DÉMONSTRATION. Nous avons

$$m(Q) = Q + \Delta^I Q + \Delta^C Q = \frac{a + \Delta^I a}{b + \Delta^I b} + \Delta^C Q \Rightarrow$$

$$Q + \Delta^I Q \quad b + \Delta^I b = a + \Delta^I a \Rightarrow$$

$$b\Delta^I Q = \Delta^I a - Q\Delta^I b = \Delta^I a - \frac{a}{b}\Delta^I b \Rightarrow$$

$$\Delta^I Q = \frac{b\Delta^I a - a\Delta^I b}{b^2} \Rightarrow \frac{\Delta^I Q}{Q} = \frac{b\Delta^I a - a\Delta^I b}{b^2\frac{a}{b}}$$

d'où on obtient (3.2.9) et (3.2.10). ∎

Nous avons de ce théorème que la borne maximale pour l'erreur relative de la division est

$$|\eta^I(Q)| \leq 2 \cdot eps$$

Le tableau ci-après fournit les erreurs relatives d'entrée pour les opérations élémentaires entre deux nombres.

Opération	Erreur d'entrée
addition $a+b$	$\eta^I(a \pm b) = \frac{a}{a \pm b}\eta^I(a) \pm \frac{b}{a \pm b}\eta^I(b)$
multiplication $a \times b$	$\eta^I(a \times b) = \eta^I(a) + \eta^I(b)$
division a/b	$\eta^I(a/b) = \eta^I(a) - \eta^I(b)$
racine carrée $\sqrt{a}$	$\eta^I(a) = 0.5 \cdot \eta^I(a)$

TABLE 3.6 – Erreurs des opérations arithmétiques

3.2.1 Exercice

EXERCICE 3.1 Soient trois nombres réels positifs a, b, c avec $a > b > c$. On calcule leur somme de deux façons :

(1) $S = (a+b)+c$

(2) $S = a+(b+c)$

Quel résultat est le meilleur ?

3.3 Les algorithmes et les erreurs

Nous allons maintenant étudier l'influence des erreurs du calcul numérique sur le résultat final d'une suite d'opérations numériques, effectuées selon un algorithme. Pour ce faire, nous examinerons la propagation des erreurs d'une opération numérique à la suivante. L'étude se fera à l'aide de deux notions : le nombre-condition et la stabilité.

3.3.1 Nombre-condition

Le nombre-condition d'une fonction f décrit la sensibilité de la valeur de la fonction f (x) aux variations de la valeur de l'argument x. Elle est calculée par la variation relative maximale de la valeur de f (x) pour une variation relative correspondante de la valeur de x. De façon formelle on peut poser

$$\text{condition ou nombre condition } \kappa(x) = \max_{V(x)} \frac{\frac{f(x)-f(x)}{f(x)}}{\frac{x-x}{x}}$$

où V(x) est un voisinage de x . Si |x − x | est petit, alors on a

$$\kappa(x) \quad \frac{f(x)}{f(x)} \cdot x$$

Si κ (x) est grand, on dit que la fonction f (x) est mal conditionnée. Ce résultat peut s'appliquer quand x = f l(x) et, si f (x) est mal conditionnée, l'erreur de calcul numérique de la valeur de f peut être grande.

Remarquons que le nombre-condition est indépendant de la méthode que nous utilisons pour calculer la valeur de la fonction f au point x. Il s'agit donc d'une mesure de la « qualité numérique » de la fonction f .

3.3.2 Stabilité

La stabilité d'un algorithme exprime le fait que pour des petites variations des valeurs des entrées de l'algorithme, les valeurs de sortie de cet algorithme présentent aussi des petites variations.

Évaluer la stabilité d'un algorithme n'est pas chose facile. Il faut décomposer l'algorithme φ pour le calcul de la fonction f , en transformations élémentaires $\varphi^{(k)}; k = ,1,\ldots,r$, c'est-à-dire $\varphi = \varphi^{(r)} \circ \varphi^{(r-1)} \circ \cdots \circ \varphi^{(1)}$. Une transformation élémentaire est une transformation qui se limite à une seule opération arithmétique. Par exemple si nous avons à calculer la fonction f (x) = ax + b, nous avons deux transformations élémentaires

(1) $\varphi^{(1)} : t_1 \leftarrow a \times x$

(2) $\varphi^{(2)} : t_2 \leftarrow t_1 + b$

de sorte que $f = \varphi^{(2)} \circ \varphi^{(1)}$.

Ensuite il faut évaluer le nombre-condition pour chaque $\varphi^{(k)}; k = ,1,\ldots,r$. S'il n'y a pas de transformation qui est mal conditionnée, alors l'algorithme est stable.

Remarquons ici que la stabilité d'un algorithme dépend de l'algorithme pour le calcul de f , c'est-à-dire de la méthode utilisée pour son calcul. Elle mesure donc la « qualité numérique » de l'algorithme. Il va de soi que pour la même fonction à calculer, on préférera un algorithme qui est stable.

EN SUBSTANCE

- ***La « qualité numérique » d'une fonction f est évaluée à l'aide du nombre-condition***

$$\kappa(x) = \frac{\frac{f(x)-f(x)}{f(x)}}{\frac{x-x}{x}} \qquad \frac{f'(x)}{f(x)} \cdot x$$

Si le nombre-condition est petit, la fonction est bien conditionnée. Dans le cas contraire elle est mal conditionnée.

- ***La « qualité numérique » d'un algorithme est évaluée par sa stabilité, c'est-à-dire par le fait que les fonctions utilisées par l'algorithme sont bien conditionnées.***

3.3.3 Exercices

EXERCICE 3.2 Calculer le nombre-condition des fonctions

(1) $f(x) = \sqrt{x}$

(2) $f(x) = \frac{1}{1-x^2}$

EXERCICE 3.3 Soit la fonction

$$f(x) = \sqrt{x+1} - \sqrt{x}; x \in R$$

(1) Décomposer en transformations élémentaires le calcul de f .

(2) Calculer pour chaque transformation, son nombre condition.

(3) Établir la condition qui rend l'algorithme instable.

(4) En utilisant le résultat précédent, établir un autre algorithme qui est toujours stable.

3.4 Formalisation de la notion de l'algorithme

Pour appliquer les notions que nous avons vu à la section précédente, à un algorithme, nous devons, d'abord, éétudier ce que c'est qu'un algorithme du point de vue de l'analyse numérique et ensuite examiner la propagation des erreurs.

Un algorithme, du point de vue de l'analyse numérique, est une transformation $\varphi : D \rightarrow R^m$ avec $D \subseteq R^n$, qui fait passer d'un ensemble des valeurs initiales (données) $x^T = [x_1, \ldots, x_n]$ à un ensemble des valeurs finales (résultats) $y^T = [y_1, \ldots, y_m]$. La transformation φ est décomposée en une suite des transformations élémentaires $\varphi = \varphi^{(r)} \circ \cdots \circ \varphi^{(1)}$, avec $\varphi^{(k)} : D_{k-1} \rightarrow R^{n_k}$ et $D_{k-1} \subseteq R^{n_{k-1}}$, telle que $\varphi^{(k)} \; x^{(k-1)} = x^{(k)}; k = 1, \ldots, r$.

Si on remplace x par $fl(x)$[2], alors l'application de l'algorithme fournit le résultat $fl(y) = fl(\varphi(fl(x)))$. Nous avons donc pour le résultat de l'algorithme une erreur absolue $\Delta y = fl(y) - y$

2. $fl(x) = [fl(x_1), \ldots, fl(x_n)]$ est le vecteur composé par les nombres-machine du vecteur $x = [x_1, \ldots, x_n]$

qui s'exprime à l'aide de la relation :

$$\Delta y_i = m(y_i) - y_i = \varphi_i(fl(x)) - \varphi_i(x)$$
$$\sum_{j=1}^{n} (fl(x_j) - x_j)\frac{\partial \varphi_i(x)}{\partial x_j} = \sum_{j=1}^{n} \frac{\partial \varphi_i(x)}{\partial x_j}\Delta x_j \; ; i = 1, \ldots, m \qquad (3.4.1)$$

ou, sous forme matricielle :

$$\Delta y = \begin{bmatrix} \Delta y_1 \\ \vdots \\ \Delta y_m \end{bmatrix} \quad \begin{bmatrix} \frac{\partial \varphi_1}{\partial x_1} & \cdots & \frac{\partial \varphi_1}{\partial x_n} \\ \vdots & \cdots & \vdots \\ \frac{\partial \varphi_m}{\partial x_1} & \cdots & \frac{\partial \varphi_m}{\partial x_n} \end{bmatrix} \quad \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \end{bmatrix} = J[\varphi(x)]\Delta x \qquad (3.4.2)$$

où $J[\varphi(x)]$ est le jacobien de φ. La quantité $\frac{\partial \varphi_i(x)}{\partial x_j}$ représente la sensibilité avec laquelle y_i réagit aux perturbations Δx_j de x_j.

L'erreur relative est donnée par :

$$\eta(y_i) = \sum_{j=1}^{n} \frac{x_j}{\varphi_i(x)}\frac{\partial \varphi_i}{\partial x_j}\eta(x_j) = \sum_{j=1}^{n} \kappa(x_j)\eta(x_j) \quad ; \quad y_i = 0, x_j = 0, i = 1, \ldots, m \qquad (3.4.3)$$

La valeur $\kappa(x_j) = \frac{x_j}{\varphi_i(x)}\frac{\partial \varphi_i}{\partial x_j}$ indique la manière avec laquelle une erreur relative en x_j affecte l'erreur relative en y_i. Ces valeurs ont l'avantage de ne pas dépendre de la grandeur des valeurs x_j et y_i. On a appelé ces valeurs nombres-condition.

EN SUBSTANCE

- ***L'erreur du résultat*** $y = [y_1, \ldots, y_m]$ ***d'un algorithme avec comme entrées*** $x = [x_1, \ldots, x_n]$ ***est***

$$\Delta y = \begin{bmatrix} \Delta y_1 \\ \vdots \\ \Delta y_m \end{bmatrix} \quad \begin{bmatrix} \frac{\partial \varphi_1}{\partial x_1} & \cdots & \frac{\partial \varphi_1}{\partial x_n} \\ \vdots & \cdots & \vdots \\ \frac{\partial \varphi_m}{\partial x_1} & \cdots & \frac{\partial \varphi_m}{\partial x_n} \end{bmatrix} \quad \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \end{bmatrix} = J[\varphi(x)] \cdot \Delta x$$

où $J[\varphi(x)]$ ***est le jacobien de*** φ.

- ***L'erreur relative du résultat est***

$$\eta(y) = \begin{bmatrix} \eta(y_1) \\ \vdots \\ \eta(y_m) \end{bmatrix} = \begin{bmatrix} \kappa(\varphi_1, x_1) & \cdots & \kappa(\varphi_1, x_n) \\ \vdots & \cdots & \vdots \\ \kappa(\varphi_m, x_1) & \cdots & \kappa(\varphi_m, x_n) \end{bmatrix} \begin{bmatrix} \eta(x_1) \\ \vdots \\ \eta(x_n) \end{bmatrix} = \kappa(\varphi, x) \cdot \eta(x)$$

où $\kappa(\varphi_i, x_j)$ ***le nombre-condition rélatif à la fonction*** φ_i ***relatif au point*** x_j **.**

3.5 Propagation des Erreurs

Nous allons maintenant étudier la propagation des erreurs de calcul lors de l'exécution d'un algorithme.

FAIT 3.2 Le déroulement d'un algorithme se fait selon le schéma suivant :

$$x = x^{(0)} \to \varphi^{(1)}\ x^{(0)}\ = x^{(1)} \to \cdots \to \varphi^{(r)}\ x^{(r-1)}\ = x^{(r)} = y \qquad (3.5.1)$$

Si on s'arrête à la k-ième étape ($1 \le k \le r$) nous avons effectué la partie

$$x = x^{(0)} \to \varphi^{(1)}\ x^{(0)}\ = x^{(1)} \to \cdots \to \varphi^{(k)}\ x^{(k-1)}\ = x^{(k)} \qquad (3.5.2)$$

et il nous reste à effectuer la partie :

$$x^{(k)} \to \varphi^{(k+1)}\ x^{(k)}\ = x^{(k+1)} \to \cdots \to \varphi^{(r)}\ x^{(r-1)}\ = x^{(r)} = y \qquad (3.5.3)$$

Notons par $\psi^{(k)}$ l'application qui reste à faire après k étapes, i.e.

$$\psi^{(k)} = \varphi^{(r)} \circ \varphi^{(r-1)} \circ \cdots \circ \varphi^{(k+1)} : D_k \to R^m \ ; \ k = 1, 2, \ldots r-1 \qquad (3.5.4)$$

Notons par $J\varphi^{(k)} \cdot J\psi^{(l)}$ le jacobien de la composition $\varphi^{(k)} \circ \psi^{(l)}$. Nous savons que nous avons :

$$J(f \circ g) = Jf(g(x)) \cdot Jg(x) \qquad (3.5.5)$$

ainsi que

$$J\varphi(x) = J\varphi^{(r)}\ x^{(r-1)}\ \cdot J\varphi^{(r-1)}\ x^{(r-2)}\ \cdots\cdot J\varphi^{(1)}\ x^{(r-1)}$$

$$J\psi^{(k)}\ x^{(k)}\ = J\varphi^{(r)}\ x^{(r-1)}\ \cdot J\varphi^{(r-1)}\ x^{(r-2)}\ \cdots\cdot J\ \varphi^{(k+1)}\ x^{(k)} \qquad k = 0, 1, \ldots, r \qquad (3.5.6)$$

Pour étudier la propagation des erreurs lors du déroulement d'un algorithme, examinons ce qui se passe quand on réalise une application élémentaire $\varphi^{(k)}$. Nous avons, à la place de la valeur $x^{(k+1)} = \varphi^{(k+1)}\ x^{(k)}$, la valeur

$$\mathrm{fl}\ x^{(k+1)}\ = \mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} \qquad (3.5.7)$$

L'erreur absolue $\Delta x^{(k+1)} = \mathrm{fl}\ x^{(k+1)}\ - x^{(k+1)}$ devient donc :

$$\Delta x^{(k+1)} = \mathrm{fl}\ x^{(k+1)}\ - x^{(k+1)} = \mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ - \varphi^{(k+1)}\ x^{(k)} \qquad (3.5.8)$$

qui peut aussi s'écrire sous la forme suivante :

$$\Delta x^{(k+1)} = \mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ - \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ + \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ - \varphi^{(k+1)}\ x^{(k)} \qquad (3.5.9)$$

De (3.4.2) nous avons :

$$\varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ - \varphi^{(k+1)}\ x^{(k)}\ J\varphi^{(k+1)}\ x^{(k)}\ \Delta x^{(k)} \qquad (3.5.10)$$

La machine ne peut pas forcement coder la valeur $\varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}$ et nous aurons donc un nombre-machine $\mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}$ qui représentera cette valeur. Comme

$$\varphi^{(k+1)}\ x^{(k)} = \varphi_1^{(k+1)}\ x^{(k)}\ , \varphi_2^{(k)}\ x^{(k)}\ , \cdots , \varphi_{n_{k+1}}^{(k)}\ x^{(k)} \tag{3.5.11}$$

nous avons pour la représentation machine les relations :

$$\mathrm{fl}\ \varphi_i^{(k+1)}\ \mathrm{fl}\ x^{(k)} = \varphi_i^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ \ 1+\eta_i\ \mathrm{fl}\ x^{(k)} \quad ; i = 1, \ldots, n_k \tag{3.5.12}$$

où $\eta_i\ x^{(k)}$ est l'erreur relative d'arrondi, engendrée durant le calcul de la i-ième composante de l'application élémentaire $\varphi^{(k+1)}$ et telle que $|\ \eta_i\ x^{(k)}\ | \leq$ eps. Sous forme matricielle (3.5.12) s'écrit :

$$\mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} = (I + H_{k+1}) \cdot \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} \tag{3.5.13}$$

avec

$$H_{k+1} = \mathrm{diag}\ \ \eta_1\ \mathrm{fl}\ x^{(k)}\ , \eta_2\ \mathrm{fl}\ x^{(k)}\ , \ldots , \eta_{n_k}\ \mathrm{fl}\ x^{(k)} \tag{3.5.14}$$

Par conséquent le premier terme de la partie droite de (3.5.9) s'écrit :

$$\mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} - \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} = H_{k+1} \cdot \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} \tag{3.5.15}$$

D'autre part

$$H_{k+1} \cdot \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ \ H_{k+1} \cdot \varphi^{(k+1)}\ x^{(k)} = H_{k+1} \cdot x^{(k+1)} \tag{3.5.16}$$

car les erreurs dues aux différences entre $\mathrm{fl}\ x^{(k)}$ et $x^{(k)}$ sont multipliées par les erreurs relatives qui sont les éléments de la matrice H_{k+1} et deviennent ainsi négligeables. Par conséquent nous avons :

$$\mathrm{fl}\ \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)} - \varphi^{(k+1)}\ \mathrm{fl}\ x^{(k)}\ \ H_{k+1} \cdot x^{(k+1)} \tag{3.5.17}$$

relation qui permet de calculer H_{k+1}. La quantité $H_{k+1} \cdot x^{(k+1)}$ est l'erreur absolue entre le résultat de la (k + 1)-ième application élémentaire et sa représentation machine. Les éléments diagonaux de H_{k+1} représentent les erreurs relatives correspondantes. En utilisant (3.5.9), (3.5.10) et (3.5.17) nous avons :

$$\Delta x^{(k+1)}\ \ J\varphi^{(k+1)}\ x^{(k)} \cdot \Delta x^{(k)} + H_{k+1} \cdot x^{(k+1)}\ ; \quad \Delta x^{(0)} = \Delta x \tag{3.5.18}$$

Pour l'erreur absolue du résultat de l'algorithme on utilise (3.5.6) et on obtient :

$$\Delta y\ \ J\varphi(x) \cdot \Delta x + J\psi^{(1)}(x^{(1)}) \cdot H_1 \cdot x^{(1)} + \cdots + J\psi^{(r-1)}\ x^{(r-1)} \cdot H_{r-1} \cdot x^{(r-1)} + H_r y \tag{3.5.19}$$

De (3.5.19) on constate que ce sont les jacobiens $J\psi^{(k)}$ de l'application restante $\psi^{(k)}$ qui sont importants pour l'évaluation des effets des erreurs d'arrondi intermédiaires H_k sur le résultat final, car, pour deux algorithmes différents, $J\varphi(x)$ reste inchangé. Ainsi l'effet total de l'arrondi pour un algorithme donné A est fourni par

$$E_r(A,x) = J\psi^{(1)}(x^{(1)}) \cdot H_1 \cdot x^{(1)} + \cdots + J\psi^{(r-1)}\ x^{(r-1)} \cdot H_{r-1} \cdot x^{(r-1)} + H_r y \tag{3.5.20}$$

FAIT 3.3 Un algorithme A est plus crédible qu'un autre algorithme $\tilde{A}$ si $E_r(A,x) \leq E_r(\tilde{A},x)$ pour le même ensemble de données x .

D'autre part on peut admettre que $|H_r \cdot y| \leq |y| \cdot eps$ et, si l'erreur d'entrée des données est $\Delta^I(x)$, nous avons $\Delta^I(x) \leq |x| \cdot eps$. Par conséquent pour chaque algorithme nous avons une erreur d'entrée

$$\Delta^I y = [|J\varphi(x)||x| + |y|] \cdot eps \tag{3.5.21}$$

Appelons la quantité $\Delta^I y$ erreur inhérente du résultat y . Cette erreur est inévitable quand nous utilisons un ordinateur. En tenant compte de l'erreur inhérente du résultat, nous pouvons spécialiser la notion de la stabilité d'un algorithme comme suit : Un algorithme est numériquement stable si les erreurs intermédiaires d'arrondi $H_k \cdot x^{(k)}$ sont de même ordre de grandeur que l'erreur inhérente, nous disons que l'algorithme φ.

FAIT 3.4 L'objectif principal de l'Analyse Numérique est d'élaborer des algorithmes qui sont numériquement stables.

3.5.1 Exercice

EXERCICE 3.4 Considérons le calcul

$$y = a^2 - b^2$$

selon deux algorithmes :

- Algorithme 1 : $y \leftarrow a^2 - b^2$
- Algorithme 2 : $y \leftarrow (a+b)\cdot(a-b)$

En utilisant les formules de la propagation des erreurs

(1) Calculer l'erreur Δy pour chacun de deux algorithmes.

(2) Trouver s'il y a un algorithme qui est plus crédible que l'autre.

(3) Étudier la stabilité numérique des algorithmes.

4

ERREURS DIRECTE ET INVERSE

L'étude des erreurs de calcul d'un algorithme est difficile à mener, car les résultats de ces calculs sont dans M (R) qui est une structure "pauvre". Par exemple les propriétés de l'associativité, de commutativité ou de la distributivité sont absentes. L'idée est de pouvoir plonger dans R les calculs qui se font dans M (R).

Supposons qu'en exécutant l'algorithme $y = f(x)$ nous obtenons le résultat $f\,l(y)$, que l'on notera dans cette section par y qui est une approximation de la valeur de y. La question qui se pose concerne la qualité de y. D'après Wilkinson, nous pouvons poser cette question de deux façons :

– Mesurer la différence entre y et y. Il s'agit en réalité de l'erreur que nous commettons quand nous calculons la solution numérique (c'est-à-dire à l'aide d'un ordinateur) du problème. Nous partons de x, nous appliquons f , qui à cause de la précsion finie de l'ordinateur devient f et nous obtenons comme résultat y au lieu de y. Il est donc legitime d'appeler cette erreur, erreur directe. L'erreur relative de précision $\eta(y) = \frac{y-y}{y}$, donnée par (1.4.3), peut être considérée comme une erreur de la précision relative directe.

– Nous pouvons aussi s'interroger sur les données en entrée $x = x + \Delta x$ qui ont permis à l'algorithme de nous fournir comme résultat y, c'est à dire on cherche à évaluer x tel que $y = f(x)$. Cette approche revient à considérer que la solution obtenue y est la solution exacte calculée sur une entrée perturbée x, à l'aide d'un algorithme perturbé f . C'est une sorte de problème inverse qui n'a pas forcement une solution unique. Nous sommes ici en présence d'une erreur dont la cause se trouve au commencement du calcul mais dont nous nous rendons compte après le calcul et par un retour en arrière. En effet nous pensons que nous commençons nos calculs avec x et nous appliquons f , mais nous obtenons comme résultat y au lieu de y et nous nous demandons, par retour en arrière, quelle était la valeur de départ $x = x + \Delta x$ que nous avons utilisé, à la place de x, et qui nous a permis d'avoir ce

résultat y. Cette erreur $|\Delta x|$ (ou, encore, $\frac{|\Delta x|}{|x|}$) sur les données en entrée est appelée erreur inverse.

Cette deuxième façon de voir les choses est plus avantageuse que la première car elle permet de considérer que les erreurs d'arrondi ne sont pas des conséquences quasi-fortuites du mécanisme de stockage en mémoire des informations numériques par les ordinateurs, mais plutôt des conséquences des perturbations subies par les données. Ainsi le problème de borner supérieurement l'erreur relative de précision devient un problème de la théorie de perturbations qui dispose d'un arsenal des techniques et des méthodes très développé.

Nous allons développer plus en détail ces résultats.

4.1 Analyses directe et inverse

Supposons que nous voulons calculer numériquement, à l'aide d'un ordinateur, la valeur y de la fonction $f(x), x \in R$, avec f représente la transformation des données induite par un algorithme (c'est-à-dire une méthode de calcul). Nous avons donc le problème

$$(P) \ : \ y = f(x)$$

Supposons encore qu'il existe au moins une solution

$$x = g(y)$$

En général on considère f et g continues.

Le calcul par ordinateur aboutit au résultat suivant

$$P \ : \ y = f(x); \quad y \in M(R)$$

pour lequel la solution est donnée par

$$x = g(y); \quad x \in M(R)$$

Nous pouvons aussi envisager que ce résultat y, puisse être obtenu comme solution exacte (c'est-à-dire dans R) d'un problème perturbé P, qui est du même type que (P), à savoir

$$P \ : \ y = f(x + \Delta x)$$

tel que

$$x + \Delta x = x = g(y); \quad x + \Delta x \in R \text{ solution exacte}$$

avec $x = x + \Delta x$, $y = y + \Delta y$ et $f(x) = f\,l(f(x))$.

L'analyse directe s'applique à l'exécution de l'algorithme f pour la donnée x. Cette analyse appliquée à la propagation des erreurs, étudiée au chapitre précédent, fournit une majoration de la différence entre la valeur exact y du résultat et son calcul y. Cette différence est l'erreur directe qui nous informe sur la valeur de la précision avec laquelle le problème a été résolu.

L'analyse directe présente un certain nombre de difficultés.

– La fonction f est rarement connue avec précision.

– Le calcul de la propagation de l'erreur directe lors de l'exécution d'un algorithme est en général complqué et, si les formules de calcul sont complexes, on doit faire des approximations, en linéarisant, par exemple, des relations non linéaires.
– Le résultat y est dans l'ensemble M (R) qui a une structure pauvre étant donné qu'il est depourvue des propriétes fondamentales des opérations arithmétiques comme, par exemple, la commutativité.

Pour éviter ces difficultés, nous pouvons analyser l'erreur $x - x = \Delta x$ du calcul $x = g(y)$ dans R. C'est le principe de l'analyse inverse due à Wilkinson qui permet ainsi d'étudier l'erreur de calcul comme un problème de l'analyse mathématique. La valeur y est considérée comme étant issue du calcul exact de f appliquée à un point perturbé $x + \Delta x$ de x. Donc $y = f(x + \Delta x)$ et on cherche l'erreur inverse Δx dns un ensemble de perturbations admissible qu'il soit inclus dans l'espace des valeurs x. Comme donc y est considéré en tant que résultat de f sur une donnée perturbée $x + \Delta x$, cette approche permet de savoir quel était en réalité, le problème résolu. De plus, en connaissant ou en estimant la perturbation Δx en entrée, nous pouvons étudier l'erreur à la sortie en utilisant la thorie des perturbations de l'analyse.

4.2 Relations entre les erreurs directe et inverse

Nous pouvons établir une relation entre les erreurs directes et inverse, en utilisant le nombre-condition.

Faisons l'hypothèse que la fonction $y = f(x)$ est deux fois continûment différentiable. Alors nous avons

$$y(y) - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(x + \xi\Delta x)}{2!}(\Delta x)^2, \qquad \xi \in]0, 1[$$

d'où nous obtenons

$$\frac{y(y) - y}{y} = \frac{xf'(x)}{f(x)}\frac{\Delta x}{x} + \frac{f''(x + \xi\Delta x)}{y \cdot 2!}(\Delta x)^2$$

Comme nous avons déjà vu, le premier terme du second membre est le nombre-condition de f. Nous pouvons donc de cette relation et des définitions des erreurs directe et inverse, obtenir l'inégalité suivante :

FAIT 4.1

$$\text{erreur directe} \leq \text{nombre-condition} \times \text{erreur inverse} \qquad (4.2.1)$$

Nous avons ainsi comme conclusion qu'une erreur inverse faible ne garantit pas que l'erreur directe soit, à son tour, faible. Il faudrait en plus, que le nombre-condition soit aussi faible, c'est-à-dire que le problème soit bien conditionné.

On dit qu'un algorithme est fiable si

$$\text{erreur inverse} \sim \text{eps} \qquad (4.2.2)$$

RÉFÉRENCES

Les livres et articles utilisés pour la redaction de ces notes sur l'Analyse des Erreurs sont les suivants :

G. ENGELN-MÜLLGES, F. UHLIG : Numerical algorithms with Fortran, Springer, 1996

J. STOER, R. BULIRSCH : Introduction to numerical analysis, Second ed., Springer-Verlag, 1992

W. PRESS et al. : Numerical recipies in Fortran, Second edition, Cambridge Un.P., 1992

J. H. WILKINSON : Rounding errors in algebraic processes, Dover, 1994

J. H. WILKINSON : The algebraic eigenvalue problem, Clarendon Press, 1965

F. B. HILDEBRAND : Introduction to numerical analysis, Dover, 1987.

N. J. HIGHAM : Accurancy and stability of numerical algorithms, Siam, 1996

D. E. KNUTH : heart of computer programming, vol. 2, Seminumerical algorithms, second ed., Addison-Wesley, 1981

J.-M. MULLER ET AL. : Handbook of Floating-Point Arithmetic, Birkhäuser, 2010

D. GOLDBERG : What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, v.23, no 1, pp.5-48, 1991

W. KAHAN : Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, 1966

G. HANROT, V. LEFÈVRE, S. PUTOT, P. ZIMMERMANN : Calcul Numérique Certifié, INRIA Octobre 2007

Table des matières