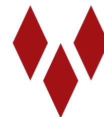
Chrysostome Baskiotis

Laurence Lamoulie

Fascicule 1

ANALYSE NUMÉRIQUE

Analyse des erreurs



Année 2009 – 2010

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

1.1 Introduction . . . 3
1.2 L'ordinateur et les nombres . . . 5
1.2.1 Conversion décimale – binaire . . . 6
1.2.2 Élaboration de la forme flottante . . . 7
1.3 Nombres normalisés et spéciaux . . . 9
1.4 Arithmétique arrondie . . . 12
1.4.1 Exercices . . . 14
1.5 Les nombres réels et les nombres-machine . . . 15
1.5.1 Étude de l'erreur de précision relative . . . 15
1.5.2 Exercices . . . 17

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 $\mathbb{R}$ sera « vu » par l'ordinateur comme un sous-ensemble fini $M(\mathbb{R}) \subset \mathbb{R}$ (i.e. l'ordinateur opère un échantillonnage – au sens traitement du signal du terme – de $\mathbb{R}$). Cette représentation est une application $fl : \mathbb{R} \rightarrow M(\mathbb{R})$, 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 $x \neq fl(x)$, 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 :

$$\mathtt{b}_q\mathtt{b}_{q-1}\ \ldots\ \mathtt{b}_1.\mathtt{b}_{-1}\ \ldots\ \mathtt{b}_{-p}$$

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

$$\mathtt{0.b}_q\mathtt{b}_{q-1}\ \ldots\ \mathtt{b}_1\mathtt{b}_{-1}\ \ldots\ \mathtt{b}_{-p}\ \times\ \mathtt{10}^q$$ [2]

La partie après le point $\mathrm{b}_q\mathrm{b}_{q-1} \ldots \mathrm{b}_1\mathrm{b}_{-1} \ldots \mathrm{b}_{-p}$ s'appelle *mantisse* 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 $0.\mathrm{b'}_1 \ldots \mathrm{b'}_{-m}$, avec $\mathrm{b'}_1 = \mathrm{b}_q$, $\mathrm{b'}_2 = \mathrm{b}_{q-1}, \ldots$.

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}.

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 1.1.

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

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 $(\texttt{0.463})_{10} = (\texttt{0.0111011010000011100101011000000 ...})_2$.

Ainsi nous avons la conversion $(465.463)_{10} = (111010001.0111011010000011100101011000000\ldots)_2$ 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

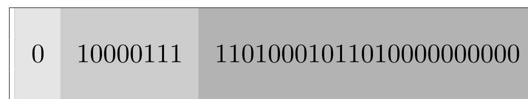
Selon le standard IEEE-754, le signe est codé sur un bit qui a la valeur 0 si le nombre est positif et 1 dans le cas contraire. Ici nous avons un nombre positif, donc le bit du signe est égal à 0.

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 :



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

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

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

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

  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***

  $$\begin{aligned} V.D = & \; v_p \times_\beta^p + r_{p-1} \times_\beta^{p-1} + \ldots + r_1 \times_\beta^1 + r_0 \times_\beta^0 . d_1 \times_\beta^{-1} + d_2 \times_\beta^{-2} \\ & + \ldots + d_q \times_\beta^{-q} + \ldots \end{aligned}$$

- ***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 voir que la représentation flottante d'un nombre x est donnée par

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

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 1.1 on établit le tableau 1.2.

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

0	00000001	00000000000000000000000

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

0	11111110	11111111111111111111111

ce qui équivaut à $(1.11\ldots1)_2 \times 10^{11111110} = \left(2-2^{-23}\right) \times 2^{127} \approx 3.4 \times 10^{38}$. 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

0	00000000	00000000000000000000001

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 1.2. 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 1.2. Cinq cas peuvent se présenter :

(1) Le résultat est un des nombres normalisés du tableau 1.2.

(2) Le résultat x est entre m_p et m_G mais ne fait pas partie des nombres normalisés du tableau 1.2.

(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 1.2. 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

$$b'_i = b_i;\ 1 \leq i \leq 22$$

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 1.2 qui est le plus proche de x et qui est plus petit que x : $b'_{23} = \begin{cases} b_{23}, & \text{si } s = 0 \\ b_{23}+1, & \text{si } s = 1 \end{cases}$, avec modification de l'exposant dans le cas de propagation de la retenue.

(3) Soit il utilisera le nombre flottant x_+ de la table 1.2 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 1.2 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*

$$1 + eps \neq 1$$

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

0	01111111	00000000000000000000000

ce qui montre que ce nombre fait partie du tableau 1.2. En utilisant cette table on peut représenter le nombre normalisé qui est immédiatement supérieur à 1. Il est donné par

0	01111111	00000000000000000000001

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*

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

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

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

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

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

Donc

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

ce qui donne

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

EN SUBSTANCE

- ***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 $1 + eps \neq 1$. Pour la simple précision on a $eps = 2^{-23} \approx 1.192 \times 10^{-7}$.***

1.4.1 Exercices

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

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

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

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

(1) la somme $x + z$ et commenter le résultat ;
Conclusion : $x + z = z$, avec $x \neq 0$! ! !

(2) le produit $x \times z$ et commenter le résultat ;

(3) la division w/z indiquant qu'il y a underflow

EXERCICE 1.3 *Soient trois réels $x = 0.400 \times 10^0 = y$, $z = 0.100 \times 10^3$. En utilisant le système de représentation de l'exercice précédent pour le stockage de ces deux nombres, calculer les deux sommes*

$(x + y) + z$, $x + (y + z)$

et commenter le résultat.

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 $\imath(x)$:

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

– *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* $123.4567 = 0.1234567 \times 10^3 = \left(0.1234 + 0.567 \times 10^{-4}\right) \times 10^3$. *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

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

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 numérotation β et nombre de chiffres de la mantisse p. Alors l'erreur relative de précision $\eta(x)$ est bornée comme suit :*

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

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 :** $\imath(x) = \frac{\Delta x}{fl(x)} = \frac{fl(x) - x}{fl(x)}$
  - **Erreur relative de précision :** $\eta(x) = \frac{\Delta x}{x} = \frac{fl(x) - x}{x} = \frac{fl(x)}{x} - 1$**, 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***

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

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* $x \in \mathbb{R}$. *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*

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

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.*

Table des matières

1 ANALYSE DES ERREURS **3**

1.1 Introduction . . . 3

1.2 L'ordinateur et les nombres . . . 5

1.2.1 Conversion décimale – binaire . . . 6

1.2.2 Élaboration de la forme flottante . . . 7

1.3 Nombres normalisés et spéciaux . . . 9

1.4 Arithmétique arrondie . . . 12

1.4.1 Exercices . . . 14

1.5 Les nombres réels et les nombres-machine . . . 15

1.5.1 Étude de l'erreur de précision relative . . . 15

1.5.2 Exercices . . . 17