Chapitre 1

Représentation d'un nombre en machine, erreurs d'arrondis

Ce chapitre est une introduction à la représentation des nombres en machine et aux erreurs d'arrondis, basé sur [2], [1].

1.1 Un exemple : calcul approché de π

Cet exemple est extrait de [2], [1]. Le nombre π est connu depuis l'antiquité, en tant que méthode de calcul du périmètre du cercle ou de l'aire du disque. Le problème de la quadrature du cercle étudié par les anciens Grecs consiste à construire un carré de même aire qu'un cercle donné à l'aide d'une règle et d'un compas. Ce problème resta insoluble jusqu'au $19^{\text{ème}}$ siècle, où la démonstration de la transcendance de π montra que le problème ne peut être résolu en utilisant une règle et un compas.

Nous savons aujourd'hui que l'aire d'un cercle de rayon r est $A = \pi r^2$. Parmi les solutions proposées pour approcher A, une méthode consiste à construire un polygône dont le nombre de côté augmenterai jusqu'à ce qu'il devienne équivalent au cercle circonscrit. C'est Archimède vers 250 avant J-C qui appliquera cette propriété au calcul des décimales du nombre π, en utilisant à la fois un polygône inscrit et circonscrit au cercle. Il utilise ainsi un algorithme pour le calcul et parvient à l'approximation de π dans l'intervalle $(3 + \frac{1}{7}, 3 + \frac{10}{71})$ en faisant tendre le nombre de côtés jusqu'à 96.

Regardons l'algorithme de calcul par les polygônes inscrits. On considère un cercle de rayon $r = 1$ et on note A_n l'aire associée au polygône inscrit à n côtés. En notant $\alpha_n = \frac{2\pi}{n}$, A_n est égale à n fois l'aire du triangle ABC représenté sur la figure 1.1, c'est-à-dire

$$A_n = n \cos\frac{\alpha_n}{2} \sin\frac{\alpha_n}{2},$$

que l'on peut réécrire

$$A_n = \frac{n}{2}(2\cos\frac{\alpha_n}{2}\sin\frac{\alpha_n}{2}) = \frac{n}{2}\sin\alpha_n = \frac{n}{2}\sin(\frac{2\pi}{n}).$$

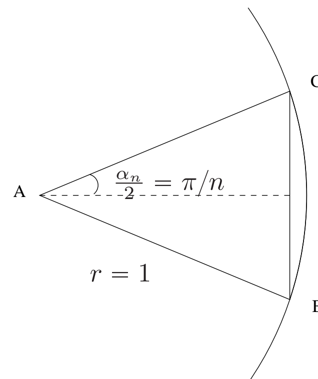

FIG. 1.1 – Quadrature du cercle

Comme on cherche à calculer π à l'aide de A_n, on ne peut pas utiliser l'expression ci-dessus pour calculer A_n, mais on peut exprimer A_{2n} en fonction de A_n en utilisant la relation

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}.$$

Ainsi, en prenant $n = 2^k$, on définit l'approximation de π par récurrence

$$x_k = A_{2^k} = \frac{2^k}{2}s_k, \quad \text{avec } s_k = \sin(\frac{2\pi}{2^k}) = \sqrt{\frac{1-\sqrt{1-s_{k-1}}}{2}}$$

En partant de $k = 2$ (i.e. $n = 4$ et $s = 1$) on obtient l'algorithme suivant :

Algorithm 1.1 Algorithme de calcul de π, version naïve

1: $s \leftarrow 1$, $n \leftarrow 4$ ▷ Initialisations
2: **Tantque** $s > 1e - 10$ **faire** ▷ Arrêt si $s = \sin(\alpha)$ est petit
3: $s \leftarrow sqrt((1 - sqrt(1 - s * s))/2$ ▷ nouvelle valeur de $\sin(\alpha/2)$
4: $n \leftarrow 2 * n$ ▷ nouvelle valeur de n
5: $A \leftarrow (n/2) * s$ ▷ nouvelle valeur de l'aire du polygône
6: **fin Tantque**

On a $\lim_{k\longrightarrow+\infty} x_k = \pi$. Ce n'est pourtant pas du tout ce que l'on va observer sur machine ! Les résultats en Matlab de la table 1.1 montre que l'algorithme commence pas converger vers π puis pour $n > 65536$, l'erreur augmente et finalement on obtient $A_n = 0$! ! "Although the theory and the program are correct, we obtain incorrect answers" ([2]).

Ceci résulte du codage des valeurs réelles sur un nombre fini de bits, ce que nous allons détailler dans ce chapitre.

1.2 Représentation scientifique des nombres dans différentes bases

Dans cette section nous introduisons les notions de mantisse, exposant, et la façon dont sont représentés les nombres sur une calculatrice ou un ordinateur.

n	A_n	$A_n - \pi$	$\sin(\alpha_n)$
4	2.000000000000000	-1.141592653589793	1.000000000000000
8	2.828427124746190	-0.313165528843603	0.707106781186548
16	3.061467458920719	-0.080125194669074	0.382683432365090
32	3.121445152258053	-0.020147501331740	0.195090322016128
64	3.136548490545941	-0.005044163043852	0.098017140329561
128	3.140331156954739	-0.001261496635054	0.049067674327418
256	3.141277250932757	-0.000315402657036	0.024541228522912
512	3.141513801144145	-0.000078852445648	0.012271538285719
1024	3.141572940367883	-0.000019713221910	0.006135884649156
2048	3.141587725279961	-0.000004928309832	0.003067956762969
4096	3.141591421504635	-0.000001232085158	0.001533980186282
8192	3.141592345611077	-0.000000307978716	0.000766990318753
16384	3.141592576545004	-0.000000077044789	0.000383495187567
32768	3.141592633463248	-0.000000020126545	0.000191747597257
65536	3.141592654807589	0.000000001217796	0.000095873799280
131072	3.141592645321215	-0.000000008268578	0.000047936899495
262144	3.141592607375720	-0.000000046214073	0.000023968449458
524288	3.141592910939673	0.000000257349880	0.000011984225887
1048576	3.141594125195191	0.000001471605398	0.000005992115260
2097152	3.141596553704820	0.000003900115026	0.000002996059946
4194304	3.141596553704820	0.000003900115026	0.000001498029973
8388608	3.141674265021758	0.000081611431964	0.000000749033514
16777216	3.141829681889202	0.000237028299408	0.000000374535284
33554432	3.142451272494134	0.000858618904341	0.000000187304692
67108864	3.142451272494134	0.000858618904341	0.000000093652346
134217728	3.162277660168380	0.020685006578586	0.000000047121609
268435456	3.162277660168380	0.020685006578586	0.000000023560805
536870912	3.464101615137754	0.322508961547961	0.000000012904784
1073741824	4.000000000000000	0.858407346410207	0.000000007450581
2.147484e+09	0.000000000000000	-3.141592653589793	0.000000000000000

TAB. 1.1 – Calcul de π avec l'algorithme naïf 1.1

1.2.1 Partie entière, mantisse et exposant

Exemple en base 10

La base 10 est la base naturelle avec laquelle on travaille et celle que l'on retrouve dans les calculatrices.

Un nombre à virgule, ou nombre décimal, à plusieurs écritures différentes en changeant simplement la position du point décimal et en rajoutant à la fin une puissance de 10 dans l'écriture de ce nombre. La partie à gauche du point décimal est la partie entière, celle à droite avant l'exposant s'appelle la mantisse. Par exemple le nombre $x = 1234.5678$ à plusieurs représentations :

$$x = 1234.5678 = 1234.5678\ 10^0 = 1.2345678\ 10^3 = 0.0012345678\ 10^6, \quad (1.1)$$

avec

- *Partie entière* : 1234

- *Mantisse* : 0.5678 ou 1.2345678 ou 0.0012345678
- *Exposant* : 4 ou 6

Selon le décalage et l'exposant que l'on aura choisi, le couple mantisse-exposant va changer mais le nombre représenté est le même. Afin d'avoir une représentation unique, celle qui sera utilisée c'est la troisième dans (1.1), où la mantisse est 1.2345678 et l'exposant 3. C'est celle où le premier chiffre avant le point décimal dans la mantisse est non nul.

Exemple en base 2

C'est la base que les ordinateurs utilisent. Les chiffres utilisables en base 2 sont 0 et 1 que l'on appelle *bit* pour *binary digit*, les ordinateurs travaillent en binaire. Par exemple

$$39 = 32 + 4 + 2 + 1 = 2^5 + 2^2 + 2^1 + 2^0 = (100111)_2,$$
$$3.625 = 2^1 + 2^0 + 2^{-1} + 2^{-3} = (11.101)_2 = (1.1101)_2\ 2^1$$

Représentation d'un nombre en machine : nombres flottants

De façon générale tout nombre réels x sera représenté dans une base b ($b = 10$ pour une calculatrice $b = 2$ pour un ordinateur) par son signe ($+$ ou $-$), la mantisse m (appelée aussi significande) , la base b et un exposant e tel que le couple (m, e) caractérise le nombre. En faisant varier e, on fait « flotter » la virgule décimale. La limitation fondamentale est que la place mémoire d'un ordinateur est limitée, c'est-à-dire qu'il ne pourra stocker qu'un ensemble fini de nombres. Ainsi un nombre machine réel ou *nombre à virgule flottante* s'écrira :

$$\begin{aligned} \tilde{x} &= \pm m \cdot b^e \\ m &= D.D \cdots D \\ e &= D \cdots D \end{aligned}$$

ou $D \in \{0, 1, ..., b-1\}$ représente un chiffre. Des représentations approchées de π sont : (0.031,2), (3.142,0), (0.003,3) et on observe qu'elles ne donnent pas la même précision. Pour rend la représentation unique et garder la meilleure précision, on utilisera une mantisse *normalisée* : le premier chiffre avant le point décimal dans la mantisse est non nul. Les nombres machine correspondants sont appelés *normalisés.* En base 2, le premier bit dans la mantisse sera donc toujours 1, et on n'écrit pas ce 1 ce qui permet d'économiser un bit. L'exposant est un nombre variant dans un intervalle fini de valeurs admissibles : $L \leqslant e \leqslant U$ (typiquement $L < 0$ et $U > 0$). Le système est donc caractérisé par quatre entiers :

- la base b ($b = 2$),
- le nombre de chiffres t dans la mantisse (en base b),
- l'exposant minimal L et maximal U.

En mathématiques on effectue les calculs avec des nombres réels x provenant de l'intervalle continu $x \in [-\infty, \infty]$. A cause de la limitation ci-dessus, la plupart des réels seront approchés sur un ordinateur. Par exemple, $\frac{1}{3}, \sqrt{2}, \pi$ possèdent

une infinité de décimales et ne peuvent donc pas avoir de représentation exacte en machine. Le plus simple des calculs devient alors approché. L'expérience pratique montre que cette quantité limitée de nombres représentables est largement suffisante pour les calculs. Sur l'ordinateur, les nombres utilisés lors des calculs sont des nombres machine $\tilde{x}$ provenant d'un ensemble discret de nombres machine $\tilde{x} \in \{\tilde{x}_{min}, \ldots, \tilde{x}_{max}\}$. Ainsi, chaque nombre réel x doit être transformé en un nombre machine $\tilde{x}$ afin de pouvoir être utilisé sur un ordinateur. Un exemple est donné sur la figure 1.1.

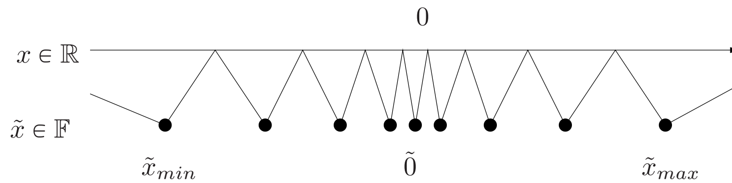

FIG. 1.2 – Représentation des nombres réels $\mathbb{R}$ par les nombres machine $\mathbb{F}$

Prenons un exemple, beaucoup trop simple pour être utilisé mais pour fixer les idées : $t = 3, \ L = -1, \ U = 2$. Dans ce cas on a 3 chiffres significatifs et 33 nombres dans le système $\mathbb{F}$. Ils se répartissent avec 0 d'une part, 16 nombres négatifs que l'on ne représente pas ici, et 16 nombres positifs représentés sur la figure 1.3.

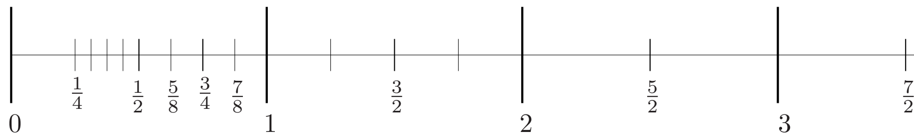

FIG. 1.3 – Nombres positifs de $\mathbb{F}$ dans le cas $t = 3, \ L = -1, \ U = 2$

Dans cet exemple, l'écriture en binaire des nombres entre $\frac{1}{2}$ et 1 est

$$\frac{1}{2} = (0.100)_2, \quad \frac{3}{4} = \frac{1}{2} + \frac{1}{4} = (0.110)_2,$$
$$\frac{5}{8} = \frac{1}{2} + \frac{1}{8} = (0.101)_2, \quad \frac{7}{8} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = (0.111)_2.$$

On obtient ensuite les autres nombres en multipliant par une puissance de 2. Le plus grand nombre représentable dans ce système est $\frac{7}{2}$ (en particulier 4 n'est pas représentable). On remarque que les nombres ne sont pas espacés régulièrement. Ils sont beaucoup plus resserrés du côté de 0 entre $\frac{1}{4}$ et $\frac{1}{2}$ que entre 1 et 2 et encore plus qu'entre 2 et 3. Plus précisément, chaque fois que l'on passe par une puissance de 2, l'espacement absolu est multiplié par 2, mais l'espacement relatif reste constant ce qui est une bonne chose pour un calcul d'ingéniérie car on a besoin d'une précision absolue beaucoup plus grande pour des nombres petits (autour de un millième par exemple) que des nombres très grands (de l'ordre du million par exemple). Mais la précision ou l'erreur relative sera du même ordre. L'erreur absolue ou relative est définie à section 1.4.1.

Précision machine. elle est décrite par le nombre machine *eps*. *eps* est le plus petit nombre machine tel que $1 + eps > 1$ sur la machine. C'est la distance entre l'entier 1 et le nombre machine $\tilde{x} \in \mathbb{F}$ le plus proche, qui lui est supérieur. Dans l'exemple précédent $eps = 1/4$.

Sur une calculatrice

Le système utilisé est la base 10 ($b = 10$). Typiquement, il y a 10 chiffres pour la mantisse et 2 pour l'exposant ($L = -99$ et $U = 99$).

- Le plus grand nombre machine

$$\tilde{x}_{max} = 9.999999999 \times 10^{+99}$$

- Le plus petit nombre machine

$$\tilde{x}_{min} = -9.999999999 \times 10^{+99}$$

- Le plus petit nombre machine strictement positif

$$\tilde{x}_{+} = 1.000000000 \times 10^{-99}$$

  Notez qu'avec des nombres dénormalisés, ce nombre serait $0.000000001 \times 10^{-99}$, c'est-à-dire avec seulement un chiffre significatif !

Les différences de représentation des nombres flottants d'un ordinateur à un autre obligeaient à reprendre les programmes de calcul scientifique pour les porter d'une machine à une autre. Pour assurer la compatibilité entre les machines, depuis 1985 une norme a été proposée par l'IEEE (Institute of Electrical and Electronics Engineers), c'est la norme 754.

1.3 Nombres flottants : le système IEEE 754

Le système IEEE 754 est un standard pour la représentation des nombres à virgule flottante en binaire. Il définit les formats de représentation des nombres à virgule flottante (signe, mantisse, exposant, nombres dénormalisés) et valeurs spéciales (infinis et NaN). Le bit de poids fort est le bit de signe. Cela signifie que si ce bit est à 1, le nombre est négatif, et s'il est à 0, le nombre est positif. Les N_e bits suivants représentent l'exposant décalé, et les N_m bits suivants représentent la mantisse.

L'exposant est décalé de $2^{N_e-1} - 1$ (N_e représente le nombre de bits de l'exposant), afin de le stocker sous forme d'un nombre non signé.

1.3.1 simple précision

C'est le format 32 bits : 1 bit de signe, $N_e = 8$ bits d'exposant (-126 à 127), 23 bits de mantisse comme sur le tableau 1.3.1. L'exposant est décalé de $2^{N_e-1} - 1 = 2^7 - 1 = 127$.

signe s	*exposant e*	*mantisse m*
S	$EEEEEEEE$	$FFFFFFFFFFFFFFFFFFFFFFF$
0	1 — 8	9 — 31

Tab. 1.2 – Représentation en simple précision

$\tilde{x}$	exposant e	mantisse m
$\tilde{x} = 0$ (si S=0) $\tilde{x} = -0$ (si S=1)	$e = 0$	$m = 0$
Nombre *normalisé* $\tilde{x} = (-1)^S \times 2^{e-127} \times 1.m$	$0 < e < 255$	quelconque
Nombre *dénormalisé* $\tilde{x} = (-1)^S \times 2^{e-126} \times 0.m$	$e = 0$	$m \neq 0$
$\tilde{x} =$ `Inf` (si S=0) $\tilde{x} = -$`Inf` (si S=1)	$e = 255$	$m = 0$
$\tilde{x} =$ `NaN` *(Not a Number)*	$e = 255$	$m \neq 0$

Tab. 1.3 – Représentation en simple précision

- Le *plus petit nombre positif normalisé* différent de zéro, et le *plus grand nombre négatif normalisé* différent de zéro sont : $\pm 2^{-126} = \pm 1,175494351 \times 10^{-38}$
- Le *plus grand nombre positif fini*, et le *plus petit nombre négatif fini* sont : $\pm(2^{24} - 1) \times 2^{104} = \pm 3,4028235 \times 10^{38}$

$\tilde{x}$	signe	exposant e	mantisse m	valeur
zéro	0	0000 0000	000 0000 0000 0000 0000 0000	0,0
	1	0000 0000	000 0000 0000 0000 0000 0000	-0,0
1	0	0111 1111	000 0000 0000 0000 0000 0000	1,0
Plus grand nombre normalisé	0	1111 1110	111 1111 1111 1111 1111 1111	$3,4 10^{38}$
Plus petit $\tilde{x} \geqslant 0$ normalisé	0	0000 0001	000 0000 0000 0000 0000 0000	2^{-126}
Plus petit $\tilde{x} \geqslant 0$ dénormalisé	0	0000 0000	000 0000 0000 0000 0000 0001	2^{-149}
Infini	0	1111 1111	000 0000 0000 0000 0000 0000	`Inf`
	1	1111 1111	000 0000 0000 0000 0000 0000	`-Inf`
NaN	0	1111 1111	010 0000 0000 0000 0000 0000	`NaN`
2	0	1000 0000	000 0000 0000 0000 0000 0000	2,0
exemple 1	0	1000 0001	101 0000 0000 0000 0000 0000	6,5
exemple 2	1	1000 0001	101 0000 0000 0000 0000 0000	-6,5
exemple 3	1	1000 0101	110 1101 0100 0000 0000 0000	-118,625

Tab. 1.4 – Exemples en simple précision

Détaillons l'exemple 3 : on code le nombre décimal -118,625 en utilisant le système IEEE 754.

1. C'est un nombre négatif, le bit de signe est donc "1",
2. On écrit le nombre (sans le signe) en binaire. Nous obtenons 1110110,101,
3. On décale la virgule vers la gauche, en laissant seulement un 1 sur sa gauche (nombre flottant normalisé) : $1110110,101 = 1,110110101 \times 2^6$. La mantisse est la partie à droite de la virgule, remplie de 0 vers la droite pour obtenir 23 bits. Cela donne 11011010100000000000000,
4. L'exposant est égal à 6, et nous devons le convertir en binaire et le décaler. Pour le format 32-bit IEEE 754, le décalage est 127. Donc $6 + 127 = 133$. En binaire, cela donne 10000101.

1.3.2 Double précision

C'est le format 64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse. L'exposant est décalé de $2^{N_e-1} - 1 = 2^{10} - 1 = 1023$.

signe s	exposant e		mantisse m	
S	$EEEEEEEEEEE$		$FFFFF...FFFFFF$	
0	1	11	12	63

Tab. 1.5 – Représentation en double précision

$\tilde{x}$	exposant e	mantisse m
$\tilde{x} = 0$ (si S=0) $\tilde{x} = -0$ (si S=1)	$e = 0$	$m = 0$
Nombre *normalisé* $\tilde{x} = (-1)^S \times 2^{e-1023} \times 1.m$	$0 < e < 2047$	quelconque
Nombre *dénormalisé* $\tilde{x} = (-1)^S \times 2^{e-1022} \times 0.m$	$e = 0$	$m \neq 0$
$\tilde{x} =$ `Inf` (si S=0) $\tilde{x} = -$`Inf` (si S=1)	$e = 2047$	$m = 0$
$\tilde{x} =$ `NaN` *(Not a Number)*	$e = 2047$	$m \neq 0$

Tab. 1.6 – Représentation en double précision

- La précision machine est $eps = 2^{-52}$.
- Le *plus grand nombre positif fini*, et le *plus petit nombre négatif fini* sont : $\tilde{x}^{\pm}_{max} = \pm(2^{1024} - 2^{971}) = \pm 1,7976931348623157 \times 10^{308}$
- **Overflow** : L'intervalle de calcul est $[\tilde{x}^{-}_{max}, \tilde{x}^{+}_{max}]$. Si un calcul produit un nombre x qui n'est pas dans cet intervalle, on dit qu'il y a *overflow*. La valeur de x est alors mise à $\pm$`Inf`. Cela peut arriver au milieu du calcul, même si le résultat final peut être représenté par un nombre machine.
- Le *plus petit nombre positif normalisé* différent de zéro, et le *plus grand nombre négatif normalisé* différent de zéro sont : $\tilde{x}^{\pm}_{min} = \pm 2^{-1022} = \pm 2,2250738585072020 \times 10^{-308}$
- Le système IEEE permet les calculs avec des nombres dénormalisés dans l'intervalle $[\tilde{x}^{+}_{min} * eps, \tilde{x}^{+}_{min}]$.
- **Underflow** : Si un calcul produit un nombre positif x qui est plus petit que $\tilde{x}^{+}_{min} * eps$, on dit qu'il y a *underflow*. Néanmoins, le calcul ne s'arrête pas dans ce cas, il continue avec la valeur de x mise à zéro.

1.3.3 En MATLAB

En MATLAB, les calculs réels sont en double précision par défaut. La fonction `single` peut être utilisée pour convertir les nombres en simple précision. Pour voir la représentation des nombres réels en MATLAB, on peut les afficher au format hexadécimal avec la commande *format hex*.

Le système hexadécimal est celui en base 16 et utilise 16 symboles : 0 à 9 pour représenter les valeurs de 0 à 9, et A, B, C, D, E, F pour représenter les valeurs

de 10 à 15. La lecture s'effectue de droite à gauche. La valeur vaut la somme des chiffres affectés de poids correspondant aux puissances successives du nombre 16. Par exemple, $5EB52_{16}$ vaut $2*16^0+5*16^1+11*16^2+14*16^3+5*16^4 = 387922$. Pour passer du binaire au format hexadecimal, c'est facile en regardant la chaîne binaire en groupe de 4 chiffres, et en représentant chaque groupe en un chiffre hexadécimal. Par exemple,

$$\begin{aligned}387922 = 01011110101101010010_2 &= 0101 \quad 1110 \quad 1011 \quad 0101 \quad 0010_2\\ &= 5 \quad E \quad B \quad 5 \quad 2_{16}\\ &= 5EB52_{16}\end{aligned}$$

La conversion de l'hexadécimal au binaire est le processus inverse.

1.4 Calculs sur les nombres flottants

1.4.1 Erreurs d'arrondi

Si $\tilde{x}$ et $\tilde{y}$ sont deux nombres machine, alors $z = \tilde{x} \times \tilde{y}$ ne correspondra pas en général à un nombre machine puisque le produit demande une quantité double de chiffres. Le résultat sera un nombre machine $\tilde{z}$ proche de z.

On définit l'*erreur absolue* entre un nombre réel x et le nombre machine correspondant $\tilde{x}$ par

$$r_a = |x - \tilde{x}|.$$

L'*erreur relative* entre ces nombres (si $x \neq 0$) est définie par

$$r = \frac{|x - \tilde{x}|}{|x|}.$$

Opérations machine : On désigne par *flop* (de l'anglais *floating operation*) une opération élémentaire à virgule flottante (addition, soustraction, multiplication ou division) de l'ordinateur. Sur les calculateurs actuels on peut s'attendre à la précision suivante, obtenue dans les opérations basiques :

$$\tilde{x}\tilde{\oplus}\tilde{y} = (\tilde{x} \oplus \tilde{y})(1 + r)$$

où $|r| < eps$, la précision machine, et $\oplus$ représente l'opération exacte, $\oplus \in \{+, -, *, /\}$ et $\tilde{\oplus}$ représente l'opération de l'ordinateur (*flop*).

1.4.2 Associativité

L'associativité des opérations élémentaires comme par exemple l'addition :

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

n'est plus valide en arithmétique finie. Par exemple, avec 6 chiffres de précision, si on prend les trois nombres

$$x = 1.23456e - 3, \quad y = 1.00000e0, \quad z = -y,$$

on obtient $(x + y) + z = (0.00123 + 1.00000e0) - 1.00000e0 = 1.23000e - 3$ alors que $x + (y + z) = x = 1.23456e - 3$. Il est donc essentiel de considérer l'ordre des opérations et faire attention où l'on met les parenthèses.

1.4.3 Monotonicité

Supposons que l'on a une fonction f strictement croissante sur un intervalle $[a, b]$. Peut-on assurer en arithmétique finie que

$$\tilde{x} < \tilde{y} \Rightarrow f(\tilde{x}) < f(\tilde{y})?$$

En général non. Dans la norme IEEE les *fonctions standard* sont implémentées de façon à respecter la monotonicité (mais pas la stricte monotonicité).

1.4.4 Erreurs d'annulation

Ce sont les erreurs dûes à l'annulation numérique de chiffres significatifs, quand les nombres ne sont représentés qu'avec une quantité finie de chiffres, comme les nombres machine. Il est donc important en pratique d'être attentifs aux signes dans les expressions, comme l'exemple suivant le montre :

Exemple : on cherche à évaluer sur l'ordinateur de façon précise, pour de petites valeurs de x, la fonction

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

Pour $|x| < \sqrt{eps}$, on risque d'avoir le nombre $\sqrt{1 - x^2}$ remplacé par 1, par la machine, et donc lors du calcul de $f(x)$, on risque d'effectuer une division par 0. Par exemple, pour $x = \frac{\sqrt{eps}}{2}$, on obtient :

```
>> f=inline('1./(1-sqrt(1-x.^2))','x');
>> f(0.5*sqrt(eps))

ans =

   Inf
```

On ne peut donc pas évaluer précisément $f(x)$ sur l'ordinateur dans ce cas. Maintenant, si on multiplie dans $f(x)$ le numérateur et le dénominateur par $1 + \sqrt{1 - x^2}$, on obtient

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

Cette fois, on peut évaluer $f(x)$ de façon précise :

```
>> f=inline('(1+sqrt(1-x.^2))./x.^2','x')

>> f(0.5*sqrt(eps))

ans =

   3.6029e+16
```

C'est ce qui se passe dans la cas du calcul de π avec l'algorithme naïf 1.1. En utilisant la formule

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}},$$

comme $\sin\alpha_n \to 0$, le numérateur à droite est de la forme

$$1-\sqrt{1-\varepsilon^2}, \quad \text{avec } \varepsilon = \sin\alpha_n \text{ petit },$$

donc sujet aux erreurs d'annulation. Pour y palier, il faut reformuler les équations de façon à s'affranchir des erreurs d'annulations, par exemple en multipliant le numérateur et le dénominateur par $(1+\sqrt{1-\sin^2\alpha_n})$.

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}} = \sqrt{\frac{1-(1-\sin^2\alpha_n)}{2(1+\sqrt{1-\sin^2\alpha_n})}} = \frac{\sin\alpha_n}{\sqrt{2(1+\sqrt{1+\sin^2\alpha_n})}}.$$

1.5 Quelques catastrophes dûes à l'arithmétique flottante

Il y a un petit nombre "connu" de catastrophes dans la vie réelle qui sont attribuables à une mauvaise gestion de l'arithmétique des ordinateurs (erreurs d'arondis, d'annulation), voir [5]. Dans le premier exemple ci-dessous cela c'est payé en vies humaines.

Missile Patriot

En février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits, donc 24 chiffres binaires. Le temps était compté par l'horloge interne du système en 1/10 de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire : 1/10 = 0,1 (dans le système décimal) = 0,0001100110011001100110011... (dans le système binaire). L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque 1/10 de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui avait entraîné une accumulation des erreurs d'arrondi de 0,34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible. Ce qu'il aurait fallu faire c'était redémarrer régulièrement le système de guidage du missile.

Explosion d'Ariane 5

Le 4 juin 1996, une fusée Ariane 5, a son premier lancement, a explosé 40 secondes après l'allumage. La fusée et son chargement avaient coûté 500 millions de dollars. La commission d'enquête a rendu son rapport au bout de deux semaines. Il s'agissait d'une erreur de programmation dans le système inertiel de référence. À un moment donné, un nombre codé en virgule flottante sur 64 bits (qui représentait la vitesse horizontale de la fusée par rapport à la plate-forme de tir) était converti en un entier sur 16 bits. Malheureusement, le nombre en question était plus grand que 32768 (overflow), le plus grand entier que l'on peut coder sur 16 bits, et la conversion a été incorrecte.

Bourse de Vancouver

Un autre exemple où les erreurs de calcul on conduit à une erreur notable est le cas de l'indice de la Bourse de Vancouver. En 1982, elle a crée un nouvel indice avec une valeur nominale de 1000. Après chaque transaction boursière, cet indice était recalculé et tronqué après le troisième chiffre décimal et, au bout de 22 mois, la valeur obtenue était 524,881, alors que la valeur correcte était 1098.811. Cette différence s'explique par le fait que toutes les erreurs d'arrondi étaient dans le même sens : l'opération de troncature diminuait à chaque fois la valeur de l'indice

Bibliographie

[1] J. P. Demailly. *Analyse Numérique et Equations Différentielles*, PUG, 1994.

[2] M.J. Gander and W. Gander. *Scientific Computing. An introduction using MAPLE and MATLAB*, to appear.

[3] A. Quarteroni, R. Sacco and F. Saleri. *Méthodes numériques pour le calcul scientifique. Programmes en MATLAB*, Springer, 2000.

[4] *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, `http ://docs.sun.com/source/806-3568/ncg_goldberg.html`, 1991.

[5] `http ://www5.in.tum.de/~huckle/bugse.html`