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

Caroline Japhet

Version du 23 septembre 2021

Table des matières

Références :

[1] J. P. Demailly. *Analyse Numérique et Equations Différentielles*, PUG, 1994.
[2] W. Gander, M.J. Gander and F. Kwok, Scientific computing : an introduction using Maple and MATLAB, Springer, Cham, 2014.
[3] T. Huckle, Collection of software bugs, `http://www5.in.tum.de/~huckle/bugse.html`
[4] A. Quarteroni, F. Saleri, and P. Gervasio, Calcul scientifique, 2ème édition, Springer, 2010

Ce document est une introduction à la représentation des nombres en machine et aux erreurs d'arrondis, basée sur les références [1,2,4].

1 Un exemple : calcul approché de π

Cet exemple est extrait de [1,2]. Le nombre π est connu depuis l'antiquité, en tant que méthode de calcul du périmètre du cercle ou de l'aire du disque. 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, c'est-à-dire

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

que l'on peut réécrire

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

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

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

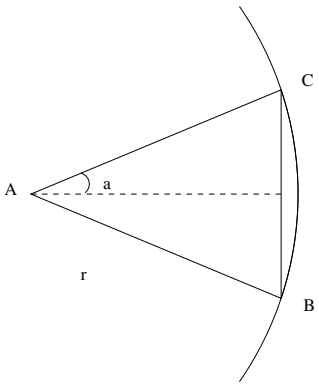

FIGURE 1 – Quadrature du cercle

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

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

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 du tableau 1 montrent que l'algorithme commence pas converger vers π puis pour $n > 65536$, l'erreur augmente et finalement on obtient $A_n = 0$! !

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

TABLE 1 – Calcul de π avec l'algorithme naïf 1.1

2 Représentation scientifique des nombres dans différentes bases

Un 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 :

$$x = 1234.5678 \cdot 10^0 = 1.2345678 \cdot 10^3 = 0.0012345678 \cdot 10^6,$$

avec la *partie entière* : 1234, la *mantisse* : 0.5678 ou 1.2345678 ou 0.0012345678, et *l'exposant* : 3 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, on utilisera celle où le premier chiffre avant le point décimal dans la mantisse est non nul, c'est-à-dire celle où la mantisse est 1.2345678 et l'exposant 3.

Un 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

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

Représentation d'un nombre en machine : nombres flottants

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 :

$$\boxed{\tilde{x} = (-1)^s \cdot (0.a_1 a_2 \dots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0} \tag{2.1}$$

ou s vaut 0 ou 1, β est la *base* (entier supérieur ou égal à 2), m est un entier appelé la *mantisse* dont la longueur t est le nombre maximum de chiffres stockés a_i (compris entre 0 et $\beta-1$) et e est un entier appelé *exposant* (variant dans un intervalle fini $]L, U[$, avec $L < 0$ et $U > 0$). En faisant varier e, on fait « flotter » la virgule décimale, c'est pour cela que les nombres de la forme (2.1) sont appelés *nombre à virgule flottante*. Les nombres $a_1 a_2 \dots a_p$ avec $p \leqslant t$ sont appelés les p *premiers chiffres significatifs* de $\tilde{x}$. En MATLAB le format `long e` est le format le plus proche de cette représentation.
La condition $a_1 \neq 0$ assure qu'un nombre a une représentation unique en gardant la meilleure précision. Par exemple, sans cette restriction, le nombre π pourrait être représenté (dans le système décimal avec 4 chiffres de précision) par : $0.031 \cdot 10^2$, $3.142 \cdot 10^0$, $0.003 \cdot 10^3$, chaque réprésentation ne donnant pas la même précision. En base 2, le premier bit dans la mantisse sera donc toujours 1, et on n'écrit pas ce 1 pour économiser un bit.
L'ensemble $\mathbb{F}$ des nombres à virgule flottante est donc caractérisé par quatre entiers :

- la base β ($\beta = 2$),
- le nombre de chiffres t dans la mantisse (en base β),
- 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 $\tilde{x} \in \{\tilde{x}_{min}, \dots, \tilde{x}_{max}\}$. 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.

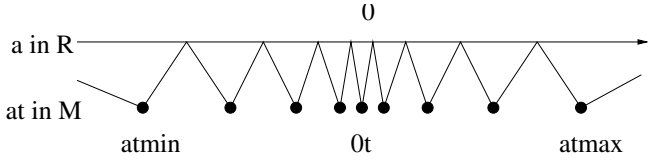

FIGURE 2 – Représentation des nombres réels $\mathbb{R}$ par les nombres machine $\mathbb{F}$

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 3.

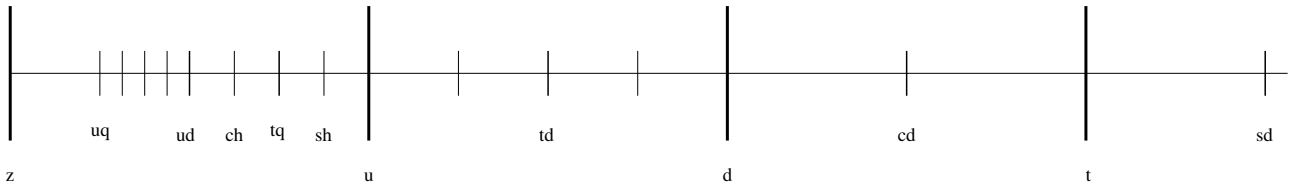

FIGURE 3 – Nombres positifs de $\mathbb{F}$ dans le cas $t = 3,\ L = -1,\ U = 2$

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.

Précision machine

L'*erreur relative* (ou *erreur d'arrondi*) entre un nombre réel $x \neq 0$ et son représentant $\tilde{x}$ dans $\mathbb{F}$ est

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

Cette erreur d'arrondi est petite car

$$\boxed{\frac{|x - \tilde{x}|}{|x|} \leqslant \frac{1}{2}\epsilon_M,} \tag{2.2}$$

où $\epsilon_M = \beta^{1-t}$ est la distance entre l'entier 1 et le nombre machine $\tilde{x} \in \mathbb{F}$ le plus proche, qui lui est supérieur. Ainsi ϵ_M est le plus petit nombre machine tel que $1+\epsilon_M > 1$ sur la machine. Dans l'exemple précédent $\epsilon_M = 1/4$. En MATLAB la commande `eps` fournit la valeur de $\epsilon_M = 2^{-52} = 2.22 \cdot 10^{-16}$.
Notons que dans (2.2) on regarde l'*erreur relative* sur x, et non pas l'*erreur absolue* $|x - \tilde{x}|$ qui ne tient pas compte de l'ordre de grandeur de x.
La relation (2.2) signifie que l' erreur relative maximale que l'ordinateur peut commettre en réprésentant un nombre réel par un nombre dans $\mathbb{F}$ est $\frac{1}{2}\epsilon_M$.
Le nombre 0 n'appartient pas à $\mathbb{F}$ (car il faudrait prendre $a_1 = 0$ dans (2.1)), il est traité à part. De plus, comme L et U sont finis, les plus petit et plus grand nombres réels positifs de $\mathbb{F}$ sont respectivement donnés par

$$\tilde{x}_{\min} = \beta^{L-1}, \quad \tilde{x}_{\max} = \beta^U(1 - \beta^{-t}),$$

ce qui donne, en MATLAB, avec les commandes `realmin` et `realmax`

$$\tilde{x}_{\min} = 2.225073858507201 \cdot 10^{-308}, \quad \tilde{x}_{\max} = 1.797693134862316 \cdot 10^{+308}.$$

Un nombre positif plus petit que $\tilde{x}_{\min}$ produit un message d'erreur appelé *underflow* et est traité soit de façon particulière soit remplacé par `0`. Un nombre positif plus grand que $\tilde{x}_{\max}$ produit un message d'erreur appelé *overflow* et est remplacé par `Inf` (qui est la représentation de $+\infty$ dans l'ordinateur). Les quantités indéterminées comme $0/0$ où ∞/∞ n'ont pas leur place dans $\mathbb{F}$: elles produisent ce que l'on appelle un `NaN` ("*Not a Number*") dans MATLAB.

3 Calculs sur les nombres flottants, erreurs d'arrondis

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. De façon générale, les opérations algébriques sur $\mathbb{F}$ ne sont pas le mêmes que sur $\mathbb{R}$.

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.

Associativité, distributivité

L'associativité des opérations élémentaires comme par exemple l'addition : $(x + y) + z = x + (y + z)$,
ou la distributivité (par exemple de la multiplication par rapport à l'addition) : $x * (y + z) = (x * y) + (x * z)$,
ne sont plus valides en arithmétique finie !

Par exemple, avec 6 chiffres de précision, si on prend les trois nombres

$$x = 1.2345 \cdot 10^{-3}, \quad y = 1.00000 \cdot 10^{0}, \quad z = -y,$$

on obtient $(x+y)+z = (0.00123+1.00000)-1.00000 = 1.23000 \cdot 10^{-3}$ alors que $x+(y+z) = x = 1.23456 \cdot 10^{-3}$.
Il est donc essentiel de considérer l'**ordre des opérations** et faire attention où l'on met les parenthèses.

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. Sur un ordinateur, les *fonctions standard* sont implémentées de façon à respecter la monotonicité (mais pas la stricte monotonicité).

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 :

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 $|x| < \sqrt{\epsilon_M}$, 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 $x = \frac{\sqrt{\epsilon_M}}{2}$, on obtient en MATLAB :

```
>> f=@(x)(1./(1-sqrt(1-x.^2)));
>> 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

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

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

```
>> f=@(x)((1+sqrt(1-x.^2))./x.^2);
>> 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 $\sqrt{(1+\sqrt{1-\sin^2\alpha_n})}$:

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

Algorithm 3.2 Algorithme de calcul de π, version stable

1: $s \leftarrow 1$, $n \leftarrow 4$ ▷ Initialisations
2: **Tantque** $s > 1e-10$ **faire** ▷ Arrêt si $s = \sin(\alpha)$ est petit
3: $s \leftarrow s/sqrt(2*(1+sqrt((1+s)*(1-s))))$ ▷ 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**

Les résultats en MATLAB du tableau 3 montrent que l'algorithme 3.2 converge vers π.

n	A_n	$A_n-\pi$	$\sin(\alpha_n)$
4	2.000000000000000	-1.141592653589793	1.000000000000000
8	2.828427124746190	-0.313165528843603	0.707106781186547
16	3.061467458920718	-0.080125194669075	0.382683432365090
32	3.121445152258052	-0.020147501331741	0.195090322016128
64	3.136548490545939	-0.005044163043854	0.098017140329561
128	3.140331156954753	-0.001261496635041	0.049067674327418
256	3.141277250932772	-0.000315402657021	0.024541228522912
512	3.141513801144301	-0.000078852445492	0.012271538285720
1024	3.141572940367091	-0.000019713222702	0.006135884649154
2048	3.141587725277160	-0.000004928312634	0.003067956762966
4096	3.141591421511200	-0.000001232078593	0.001533980186285
8192	3.141592345570118	-0.000000308019676	0.000766990318743
16384	3.141592576584872	-0.000000077004921	0.000383495187571
32768	3.141592634338563	-0.000000019251230	0.000191747597311
65536	3.141592648776986	-0.000000004812807	0.000095873799096
131072	3.141592652386591	-0.000000001203202	0.000047936899603
262144	3.141592653288993	-0.000000000300800	0.000023968449808
524288	3.141592653514593	-0.000000000075200	0.000011984224905
1048576	3.141592653570993	-0.000000000018800	0.000005992112453
2097152	3.141592653585094	-0.000000000004699	0.000002996056226
4194304	3.141592653588619	-0.000000000001174	0.000001498028113
8388608	3.141592653589500	-0.000000000000293	0.000000749014057
16777216	3.141592653589721	-0.000000000000072	0.000000374507028
33554432	3.141592653589776	-0.000000000000017	0.000000187253514
67108864	3.141592653589790	-0.000000000000004	0.000000093626757
134217728	3.141592653589793	0.000000000000000	0.000000046813379
268435456	3.141592653589794	0.000000000000001	0.000000023406689
536870912	3.141592653589794	0.000000000000001	0.000000011703345
1073741824	3.141592653589794	0.000000000000001	0.000000005851672

TABLE 2 – Calcul de π avec l'algorithme stable 3.2

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 la référence [3]. On peut citer par exemple l'explosion de la fusée Ariane 5 le 4 juin 1996, dûe à une erreur d'*overflow* dans l'ordinateur de bord.