Catastrophes Numériques

Catastrophes Numériques

TROESTLER C.
Cristophe.Troestler@umons.ac.be
http://math.umons.ac.be/

Table des matières

L'arithmétique des ordinateurs renferme quelques pièges qui ont mené à des catastrophes coûteuses financièrement et en vies humaines. Divers exemples réels vous seront présentés tout au long de ces notes. Pour comprendre par quels mécanismes ces catastrophes se produisent, nous allons vous introduire aux caractéristiques et aux limitations principales des calculs sur ordinateur.

La première chose à savoir est qu'il y a deux classes de nombres que les microprocesseurs traitent de manière différente. Il s'agit d'une part des nombres *entiers* et d'autre part des nombres *avec virgule*. Quelques exemples sont donnés dans la table 1. Nous allons examiner ces deux classes de nombres chacune à leur tour.

nombres entiers	nombres avec virgule
0	0,0
1	3,1415
4	$-427{,}278$
-2	$27 \cdot 10^{-4}$

TABLE 1 – Exemples de nombres

1 Nombres entiers

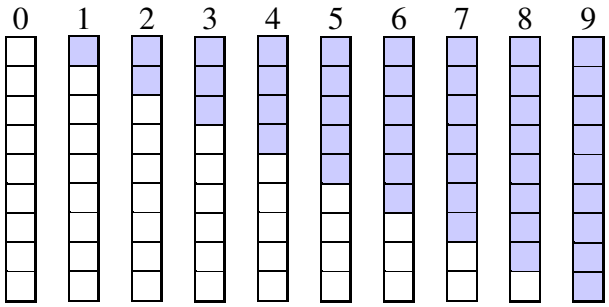
Les nombres entiers sont représentés sous forme binaire par les ordinateurs. Nous allons donc commencer par expliquer ce qu'est l'écriture binaire d'un nombre et comment celle-ci est en correspondance avec notre écriture décimale de tous les jours. Nous apprendrons ensuite comment il est possible de calculer avec ces représentations binaires et découvrirons pourquoi la *notation de position* est une invention de génie. Enfin, nous examinerons quelles sont les limites imposées par les ordinateurs. Elles seront illustrées par l'histoire de l'explosion d'Ariane 5.

1.1 Représentation binaire des nombres entiers

La manière dont on nous a accoutumé à écrire les nombres est appelée représentation *déci*male car elle utilise *dix* symboles : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Pour comprendre ce qui va suivre, il est utile de faire la distinction entre un nombre et sa représentation. Par exemple, si on a dix verres sur une table, le *nombre* de verres a un sens indépendamment de la manière dont on choisi de le transcrire symboliquement. Un même nombre peut être écrit de plusieurs manières. Par exemple, pour le nombre de verres ci-dessus, on peut l'écrire en français « dix », en anglais « ten », en chiffres arabes « 10 », en chiffres romains « X »,... Nous allons ici

apprendre à représenter les nombres avec deux symboles que nous noterons **0** et **1**. Ces symboles seront en gras pour les distinguer des 0 et 1 employés dans l'écriture décimale.

Pour comprendre comment **0** et **1** suffisent, il est utile d'examiner en détail la manière dont les nombres sont écrits en base dix. Les dix symboles nous donnent immédiatement les nombres de 0 à 9. Nous allons les représenter conjointement avec une barrette colorée de neuf cases qui permet de visualiser le nombre sans se soucier de son écriture.



Si on veut aller au delà du nombre 9, on va utiliser une nouvelle position pour indiquer le nombre de blocs de 10 (voir figure 1). On peut exprimer de la sorte les

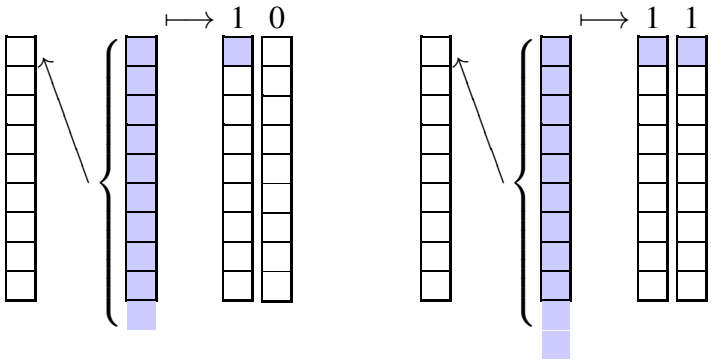


FIGURE 1 – Notation de position

nombres jusqu'à 99. Ensuite, pour aller plus loin, on crée une nouvelle position (voir figure 2). En résumé, une fois qu'on dépasse dix unités, on crée une nouvelle position pour les dizaines ; une fois qu'on dépasse dix dizaines, on crée une nouvelle position pour les centaines et ainsi de suite. Ceci se traduit par les formules suivantes :

$$45 = 4 \cdot 10 + 5 = 4 \cdot 10^1 + 5 \cdot 10^0$$
$$247 = 2 \cdot 100 + 4 \cdot 10 + 7 = 2 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$$

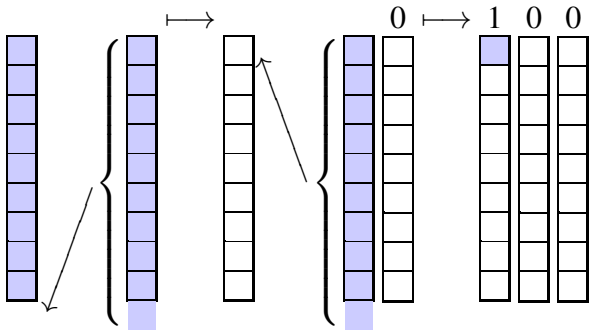


FIGURE 2 – Création d'une seconde position

L'écriture binaire des nombres repose sur les mêmes principes à la seule différence qu'on ne dispose plus que de deux symboles. Ainsi, pour le nombre deux, il faut déjà créer une nouvelle position (voir figure 3).

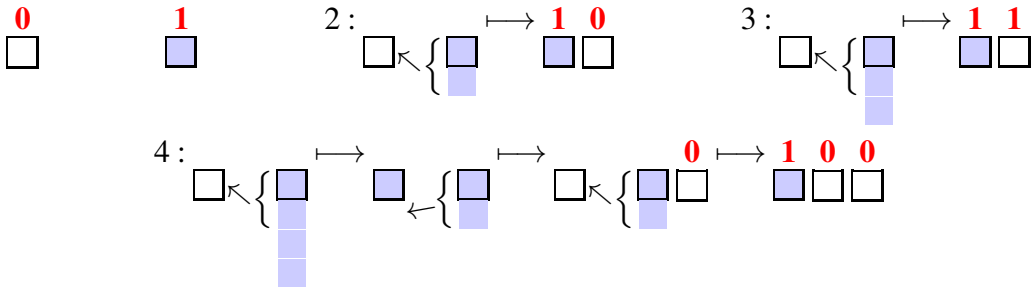


FIGURE 3 – Notation de position appliquée au binaire

En formules, on peut écrire :

$$\mathbf{100} = \mathbf{1}\cdot 4 + \mathbf{0}\cdot 2 + \mathbf{0}\cdot 1 = \mathbf{1}\cdot 2^2 + \mathbf{0}\cdot 2^1 + \mathbf{0}\cdot 2^0 = 4$$
$$\mathbf{110} = \mathbf{1}\cdot 4 + \mathbf{1}\cdot 2 + \mathbf{0}\cdot 1 = \mathbf{1}\cdot 2^2 + \mathbf{1}\cdot 2^1 + \mathbf{0}\cdot 2^0 = 6$$
$$\mathbf{1011} = \mathbf{1}\cdot 8 + \mathbf{0}\cdot 4 + \mathbf{1}\cdot 2 + \mathbf{1}\cdot 1 = \mathbf{1}\cdot 2^3 + \mathbf{0}\cdot 2^2 + \mathbf{1}\cdot 2^1 + \mathbf{1}\cdot 2^0 = 11$$

Grâce à ceci, il est facile de transformer une écriture binaire en une écriture décimale. Comment faire l'inverse ? Comment savoir par exemple quelle est l'écriture binaire du nombre 13 ?

Si on a un nombre entier x, on peut toujours faire sa division entière par 2 et on note le résultat $x \operatorname{div} 2$. Cette division peut avoir un reste, noté $x \bmod 2$. Exemples :

- 13 divisé par 2 donne 6 avec un reste de 1 car $13 = 2\cdot 6+1$. On écrit $13\operatorname{div}2 = 6$ et $13\bmod 2 = 1$;
- 16 divisé par 2 donne 8 avec un reste de 0 car $16 = 2\cdot 8+0$. On écrit $16\operatorname{div}2 = 8$ et $16\bmod 2 = 0$.

Le reste étant toujours strictement inférieur au diviseur, $x\bmod 2$ ne peut valoir que 0 ou 1. De manière générale, si on a une écriture binaire $b_n\dots b_2b_1b_0$ avec b_i qui vaut **0** ou **1**, ce qui correspond au nombre $x = b_n2^n+\cdots+b_22^2+b_12+b_0$, on peut écrire

$$x = b_n2^n+\cdots+b_22^2+b_12+b_0 = 2\big(b_n2^{n-1}+\cdots+b_22+b_1\big)+b_0$$

ce qui signifie que

$$x\operatorname{div}2 = b_n2^{n-1}+\cdots+b_22+b_1 \qquad \text{et} \qquad x\bmod 2 = b_0.$$

Ainsi donc, le digit b_0 est juste la valeur du reste de la division du nombre par 2. Connaître b_1 se fait via le même principe mais sur $x\operatorname{div}2$ au lieu de x :

$$(x\operatorname{div}2)\operatorname{div}2 = b_n2^{n-2}+\cdots+b_2 \qquad \text{et} \qquad (x\operatorname{div}2)\bmod 2 = b_1.$$

On peut continuer de la même manière pour b_2,... La figure 4 donne un exemple la totalité des calculs qui montrent que $x = 13$ a pour expansion binaire **1101**.

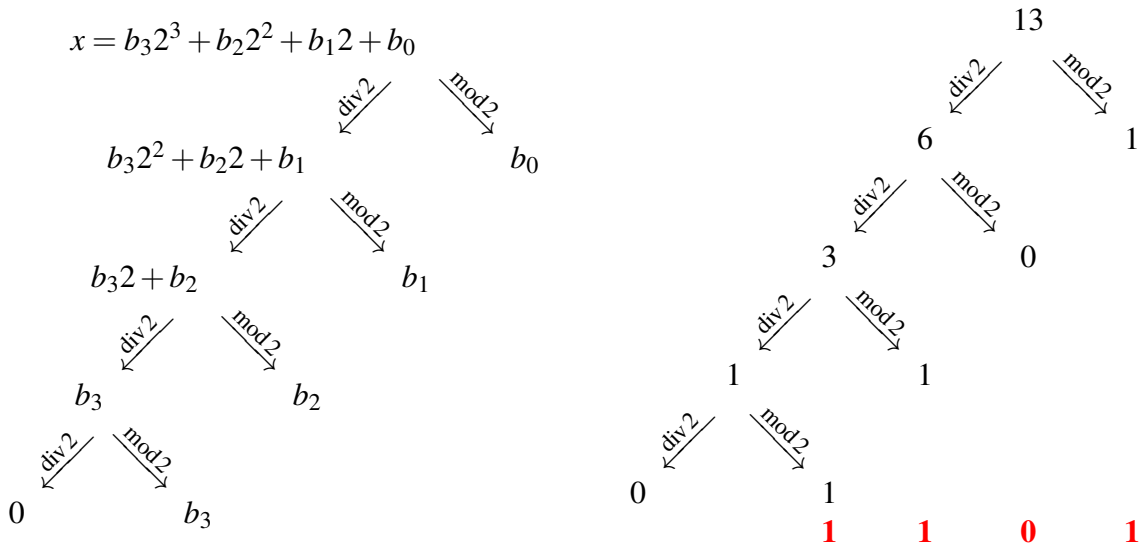
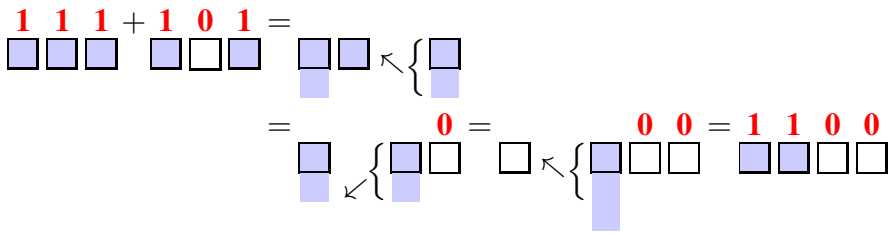


FIGURE 4 – Expansion binaire de 13

1.2 Calculs sur des représentations binaires

Effectuer des calculs avec les représentations binaires suit les mêmes principes qu'avec les représentations décimales. En effet, ce qui permet des calculs aisés n'est pas la base choisie (dix ou deux) mais la notation de position. Illustrons ceci sur l'addition de 7 et 5, en binaire **111** et **101** :



Les « transitions » marquées par des flèches correspondent aux reports dans l'écriture habituelle.

$$\begin{array}{r} 111 \\ 101 \\ \hline \end{array} \rightarrow \begin{array}{r} {\scriptstyle 1} \\ 111 \\ 101 \\ \hline 0 \end{array} \rightarrow \begin{array}{r} {\scriptstyle 1\ }{\scriptstyle 1} \\ 111 \\ 101 \\ \hline 00 \end{array} \rightarrow \begin{array}{r} {\scriptstyle 1} \\ 111 \\ 101 \\ \hline 100 \end{array} \rightarrow \begin{array}{r} \\ 111 \\ 101 \\ \hline 1100 \end{array}$$

La multiplication se fait également selon les règles habituelles :

$$\begin{array}{r} 111 \\ \times 101 \\ \hline 111 \\ +000 \\ +111 \\ \hline 100011 \end{array}$$

En fait on voit que c'est même plus facile qu'en base dix puisqu'on ne multiplie que par 0 et 1 !

La soustraction et la division (Euclidienne) se font aussi comme vous en avez l'habitude.

1.3 Limitations des ordinateurs

Les ordinateurs actuels travaillent uniquement avec deux valeurs de tension, une tension « basse » et une tension « haute ». C'est pourquoi la représentation binaire des nombres est bien adaptée : la tension basse sert à coder le **0** et la tension haute le **1**. En d'autres mots, la plus petite information pouvant être contenue

dans un ordinateur correspond à un *digit binaire* (**0** ou **1**) et est appelée *bit* (qui est l'abréviation de *binary digit*). En général les bits ne sont pas traités individuellement mais par groupes de 8 ou de multiples de 8. Un groupe de 8 bits est appelé un *byte*.

Les entiers sont en général codés sur 2 ou 4 bytes, c'est-à-dire sur 16 ou 32 bits. Leur représentation est simple : un bit est réservé pour le signe et les bits restants contiennent l'écriture binaire du nombre [1] (voir figure 5). Quel est le plus

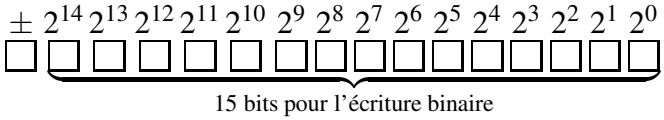
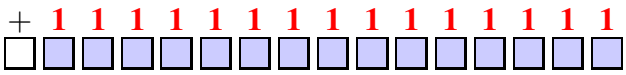
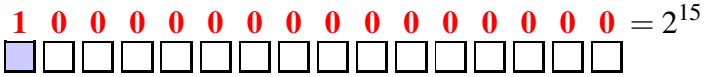


FIGURE 5 – Représentation d'un entier sur 16 bits

grand nombre qu'on peut stocker sur 16 bits ? Clairement, c'est le nombre où tous les bits sont à **1** :



Si on ajoute **1** à ce nombre, on trouve :



Dès lors, le plus grand nombre représentable sur 16 bits est $2^{15} - 1 = 32767$.

Par un raisonnement analogue, on conclut que le plus grand nombre représentable sur 32 bits est $2^{31} - 1 = 2147483647$. En résumé :

# bits	# bytes	nombres représentables
16	2	$-32\,767$ à $32\,767$
32	4	$-2\,147\,483\,647$ à $2\,147\,483\,647$

Que se passe-t-il si un nombre est tellement grand qu'il ne peut tenir sur 16 ou 32 bits ? Par exemple, sur 16 bits, que se passe-t-il si on fait $32767 + 1$? Sur la plupart des processeurs « grand public », cela va « boucler », c'est-à-dire que si on ajoute 1 au plus grand entier représentable, on va obtenir le plus petit entier négatif représentable. Sur d'autres machines, essayer de manipuler des nombres trop grands déclenche un « évènement exceptionnel » qui doit être explicitement géré par le programmeur ou qui, à défaut, arrête le programme. C'est ce qui s'est passé avec Ariane 5.

1. Ceci est une simplification. Le codage utilisé est un peu différent ce qui donne des plages allant de -32768 à 32767 pour 16 bits et de -2147483648 à 2147483647 pour 32 bits.

1.4 L'explosion d'Ariane 5

D'après le rapport d'enquête :

> On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $ 7 billion. The destroyed rocket and its cargo were valued at $ 500 million.
>
> The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and *design errors in the software of the inertial reference system*.
>
> The internal SRI software exception was caused during execution of a data *conversion from 64-bit floating point to 16-bit signed integer* value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.

En résumé le système de référence inertiel (SRI) d'Ariane 5 a essayé de convertir un nombre en virgule flottante [2] de 64 bits en un entier sur 16 bits. Ce nombre était trop grand pour être codé par un entier de 16 bits. Un évènement exceptionnel a donc été déclenché mais le code ne comprenait pas d'instructions pour le gérer... L'ironie veut que ce code provenait d'Ariane 4 et n'était pas nécessaire après le décollage...

2 Nombres à virgule fixe

Comme leur nom l'indique, les nombres à virgule fixe possèdent un nombre fixé de digits après la virgule. Ils sont aussi représentés sous forme binaire à l'intérieur des ordinateurs. C'est pourquoi, nous allons étendre l'analyse que nous avons faite plus haut pour les entiers. L'intérêt des nombres à virgule fixe est que les calculs se font sans erreurs. Nous verrons cependant que certains nombres qui ont une représentation décimale finie ont une représentation binaire infinie. Ceci engendre une erreur *au départ*. C'est là l'origine du mauvais fonctionnement des antimissiles Patriots.

2. On abordera ceux-ci à la section 3.

2.1 Représentation binaire

Nous avons vu que dans l'écriture positionnelle des nombres entiers, les digits donnent les poids des puissances de la base. Il en va de même pour les nombres avec virgule, sauf que les puissances des digits après la virgule sont négatives. Plus concrètement :

$$14{,}25 = 1\cdot 10 + 4 + 2\tfrac{1}{10} + 5\tfrac{1}{100} = 1\cdot 10 + 4\cdot 10^0 + 2\cdot 10^{-1} + 5\cdot 10^{-2}.$$

De la même manière, en binaire :

$$\mathbf{10}{,}\mathbf{11} = \mathbf{1}\cdot 2^1 + \mathbf{0}\cdot 2^0 + \mathbf{1}\cdot 2^{-1} + \mathbf{1}\cdot 2^{-2} = 2 + 0 + \tfrac{1}{2} + \tfrac{1}{4} = 1{,}75.$$

Ainsi, comme c'était déjà le cas pour les nombres entiers, il est très facile de convertir une écriture binaire en une écriture décimale.

Comment fait-on l'inverse ? Étant donné un nombre x, comment trouve-t-on sa représentation binaire ? Tout d'abord, remarquons que nous pouvons séparer la partie *entière* (celle avant la virgule) et la partie *fractionnaire* (celle après la virgule). Par exemple, pour convertir 14,25 en binaire, nous pouvons trouver la représentation de 14 en binaire, à savoir **1110**, la représentation de 0,25 en binaire, qui est **0**,**01** et les « mettre bout à bout » : **1110**,**01**. De manière générale, pour un nombre $x \geqslant 0$, nous noterons sa partie entière par $\lfloor x \rfloor$ et sa partie fractionnaire par $\mathrm{fr}(x) = x - \lfloor x \rfloor$. La partie entière $\lfloor x \rfloor$ étant un nombre entier, nous savons comment trouver sa représentation binaire. Reste à s'occuper de $\mathrm{fr}(x)$. Puisque c'est la partie après la virgule de x, elle doit s'écrire en binaire comme

$$\mathrm{fr}(x) = b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \cdots = 0{,}b_{-1}b_{-2}b_{-3}\ldots$$

pour certains b_i qui valent **0** ou **1**. Comme pour les entiers, nous allons dégager les opérations qui vont nous donner les valeurs des b_i. Remarquons que

$$2\cdot \mathrm{fr}(x) = b_{-1} + \underbrace{b_{-2}2^{-1} + b_{-3}2^{-2} + \cdots}_{<1} = b_{-1}{,}b_{-2}b_{-3}\ldots$$

Par conséquent,

$$\lfloor 2\cdot \mathrm{fr}(x) \rfloor = b_{-1} \qquad \text{et} \qquad \mathrm{fr}\big(2\cdot \mathrm{fr}(x)\big) = 0{,}b_{-2}b_{-3}\ldots$$

Ainsi, appliquées à $\mathrm{fr}(x)$, les opérations $\lfloor 2\cdot \rfloor$ et $\mathrm{fr}(2\cdot)$ donnent respectivement les valeurs de b_{-1} et un nombre similaire à celui de départ mais où les décimales sont décalées vers la gauche. Pour avoir la valeur de b_{-2}, il suffit de répéter ces opérations sur $x' := \mathrm{fr}(2\,\mathrm{fr}(x)) = 0{,}b_{-2}b_{-3}\ldots$:

$$\lfloor 2\cdot x' \rfloor = b_{-2} \qquad \text{et} \qquad \mathrm{fr}(2\cdot x') = 0{,}b_{-3}\ldots$$

On continue de la sorte jusqu'à ce qu'on ait épuisé tous les digits b_i, si du moins cela arrive. En effet, tout comme $1/3 = 0{,}333333\ldots$ a une expansion décimale illimitée, certains nombres ont une expansion binaire infinie.

Attaquons nous à $x = 0{,}1 = 1/10$. Ce nombre n'a pas de partie entière, $\lfloor x \rfloor = 0$, et donc $\mathrm{fr}(x) = x$. Appliquons la suite d'opérations ci-dessus pour trouver l'expansion binaire de x. Les calculs sont présentés graphiquement à la figure 6. On voit que $x = \mathbf{0{,}00011001100110011}\ldots$ Ceci est au cœur de l'explication du mauvais fonctionnement du missile Patriot.

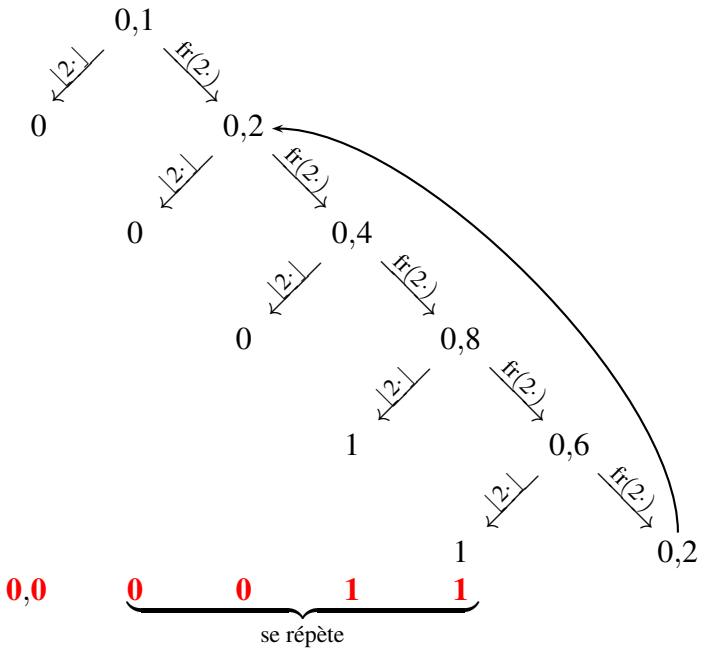


FIGURE 6 – Expansion binaire de 1/10

2.2 Le défaut des missiles « Patriot »

Voici un extrait du rapport du groupe d'experts qui a analysé le problème :

> On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.
>
> It turns out that the cause was an *inaccurate calculation of the time since boot due to computer arithmetic errors*.

> Specifically, the *time in tenths of second* as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a *24 bit* fixed point register [...] the Patriot battery had been up around *100 hours* [...]

Ce que ce rapport explique c'est qu'un compteur, appelons le n, à l'intérieur du missile contenait le nombre de dixièmes de secondes depuis sa mise en route et que le temps écoulé était obtenu en multipliant n par $1/10$ où $1/10$ est stocké (au mieux) avec 24 digits binaires après la virgule. Le nombre stocké est donc [3] :

$$x := \underbrace{0{,}000110011001100110011001}_{24\text{digits}}$$

et l'erreur commise est

$$\frac{1}{10} - x = \underbrace{0{,}000000000000000000000000}_{24\text{digits}} 1100110011001100110011 \ldots \approx 0{,}0000000953\ldots$$

Or le missile était en marche depuis environ 100 heures. Par conséquent, le compteur n contenait le nombre de dixièmes de secondes dans 100 heures, à savoir

$$n = 100 \cdots 60 \cdots 60 \cdot 10 = 3\,600\,000$$

L'ordinateur du missile devait calculer $n \cdot 0{,}1$ mais, au lieu de cela, il a calculé $n \cdot x$. Il a donc commis une erreur de

$$n\frac{1}{10} - bx \approx n \cdot 0{,}0000000953\ldots \approx 0.343\ldots$$

Comme un missile Patriot va a une vitesse moyenne de 1676 mètres par seconde, en 0,34 secondes il parcourt environ 570 mètres. Ceci explique que le missile Scud a détruire n'ai pas été intercepté.

3 Nombres à virgule flottante

Les nombres à virgule fixe ne sont pas suffisants dans de nombreuses situations, en particulier pour des calculs scientifiques. En effet, ils ne peuvent prendre ni de très grandes ni de très petites valeurs positives. C'est pourquoi tous les processeurs actuels incluent un autre type d'arithmétique dit en « virgule flottante ».

3. On suppose pour la simplicité que le nombre est tronqué plutôt qu'arrondis. Cela n'a pas d'importance pour le reste de l'analyse

3.1 La norme IEEE-754

Dans les années 1960–1970, chaque microprocesseur avait sa propre implémentation de l'arithmétique en virgule flottante, incompatible avec celle des autres. Ceci à lui seul est déjà gênant car cela rend difficile l'écriture de programmes pouvant fonctionner sur diverses machines mais, de plus, nombre de ces implémentations avaient leurs bizarreries :

- sur certaines machines, certains nombres se comportaient comme zéro pour la multiplication et la division mais pas pour l'addition ou les comparaisons ;
- sur d'autres, multiplier un nombre par 1,0 lui faisait perdre ses quatres derniers bits ;
- sur beaucoup d'ordinateurs, on pouvait avoir $x - y = 0$ alors que $x \neq y$;
- pour éviter que les programmes ne donnent des résultats grossièrement inexacts, il fallait mettre au bon endroit des instructions d'affectation telles que $x \leftarrow (x + x) - x$;
- etc.

Clairement, une norme bien pensée était nécessaire. C'est pourquoi le standard IEEE-754 a été développé, principalement grâce aux efforts du Prof. W. Kahan. Ses caractéristiques principales [4] sont :

- formats standardisés ;
- les opérations élémentaires ($+$, $-$, $\cdot$, $/$) doivent donner la meilleure réponse possible ;
- diverses formes d'arrondis ;
- introduction des quantités $+\infty$, $-\infty$ et NaN (not a number).

Ce dernier point est en général mal connu. Par exemple lorsque le programme (écrit en langage C) de la figure 7 est exécuté, il doit produire un résultat du type :

```
1/+0 = inf
1/-0 = -inf
0/0  = nan
```

Si au lieu de cela le programme s'arrête brutalement sans rien afficher, changez de compilateur [5], celui que vous avez ne vous permet pas d'exploiter toutes les caractéristiques de votre processeur.

4. La liste donnée n'est pas exhaustive !
5. Nous vous recommendons MinGW (http://www.mingw.org/).

```c
#include <stdio.h>

int main(void)
{
  printf("1/+0 = %f\n", 1. / 0.);
  printf("1/-0 = %f\n", 1. /(-0.));
  printf("0/0  = %f\n", 0. / 0.);

  return 0;
}
```

FIGURE 7 – Programme montrant les quantités $\pm\infty$ et NaN

3.2 Représentation binaire

Vous connaissez certainement l'écriture scientifique des nombres car celle-ci est présente sur de nombreuses machines à calculer. Par exemple, plutôt que 17 310 000, on écrit $1{,}731 \cdot 10^7$ ou 1,731 E7. Cela permet d'avoir tout de suite une idée de l'ordre de grandeur du nombre. Le format adopté par la norme et par tous les ordinateurs actuels peut être caractérisé de « notation scientifique binaire ». En effet, un nombre x peut s'écrire sous la forme

$$x = m2^e \qquad \text{où } (1 \leqslant |m| < 2 \text{ ou } m = 0) \text{ et } e \in \mathbb{Z}.$$

Le nombre m est appelé la *mantisse* et e l'*exposant*. Ce sont ces deux nombres qui sont stockés, avec un nombre de digits fixés. Par exemple $\mathbf{101{,}0101} = \mathbf{1{,}010101} \cdot 2^{\mathbf{10}}$ sera encodé par les valeurs de la mantisse **1010101** et de l'exposant **10**.

Typiquement, les nombres flottants « double précision » sont stockés sur 64 bits, la mantisse sur 53 et l'exposant sur 11, ce qui permet des nombres aussi petits que $2 \cdot 10^{-308}$ et aussi grands que 10^{308}.

Le principal problème des nombres à virgule flottante est que les calculs ne peuvent être exacts, même si les nombres au départ sont stockés sans erreur (à contrario par exemple de $1/10$ vu plus haut). Par exemple, avec une mantisse de trois digits, le résultat de l'addition de $\mathbf{1} = \mathbf{1{,}00} \cdot 2^{\mathbf{0}}$ et $\mathbf{0{,}0011} = \mathbf{1{,}10} \cdot 2^{\mathbf{-11}}$ est **1,0011** et celui-ci ne peut être exactement encodé par une mantisse à trois digits. C'est ici que les *modes d'arrondi* entrent en scène.

- De manière générale, on choisit d'arrondir *au plus proche*, ici cela revient à arrondir **1,0011** en **1,01** car il est plus proche de **1,0011** que **1,00**.
- On peut aussi choisir d'arrondir de manière dirigée : soit systématiquement vers le haut (c'est-à-dire au nombre encodable immédiatement supérieur) ou

vers la bas (c'est-à-dire au nombre encodable immédiatement inférieur).

Les arrondis dirigés peuvent être utiles dans certaines situations particulières mais ils doivent en général être évités. L'histoire du stock exchange de Vancouver illustre cela.

3.3 Un nouvel indice au stock exchange de Vancouver

Voici les faits :

> En 1982, le stock exchange de Vancouver a créé un nouvel indice initialisé à 1000,000. Cet indice était mis à jour après chaque transaction. Vingt-deux mois plus tard, celui-ci avait chuté à 520. Sa vraie valeur aurait dû être 1098,892.

Ce problème était dû au fait que le résultat de chaque opération était tronqué au lieu d'être arrondi au plus proche. Tronquer le résultat peut paraître naturel puisqu'on ne veut que trois digits après la virgule. Cependant, cela revient à faire un arrondi vers le bas et donc à perdre chaque fois les décimales négligées. À force d'être répétée, la perte devient substantielle.

3.4 Pertes de précision

Considérons la fonction suivante qui a l'air innocente :

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

Des manipulations algébriques élémentaires montrent que $f(x) = 1$. Cependant, nous avons l'intention de demander à la machine d'effectuer le calcul de $f(x)$ comme indiqué : d'abord faire $x+1$ puis retrancher 1 et enfin diviser ce résultat par x. L'ordinateur donnera–il 1 comme réponse ? Pour le voir, nous avons tracé à la figure 8 le graphe de f en calculant une centaine de points dans l'intervalle $[-1,1]$. Tout semble aller pour le mieux : le graphe de la figure 8 est bien celui de la fonction constante 1. Pourtant, si on fait un zoom autour de zéro, on obtient la graphe de la figure 9. Il n'a plus rien à voir avec celui de la constante 1 ! Que s'est-il passé ?

On pourrait penser que l'intervalle du zoom, à savoir $[-2\cdot 10^{-19}, 2\cdot 10^{19}]$, est trop petit et qu'avec des nombres aussi petits, la machine a du mal a les représenter et donc a faire des calculs exacts. C'est cependant oublier que les nombres sont encodés en notation scientifique et que c'est l'exposant qui donne la petitesse du nombre. La mantisse gardant toujours le même nombre de digits, il n'y a pas de

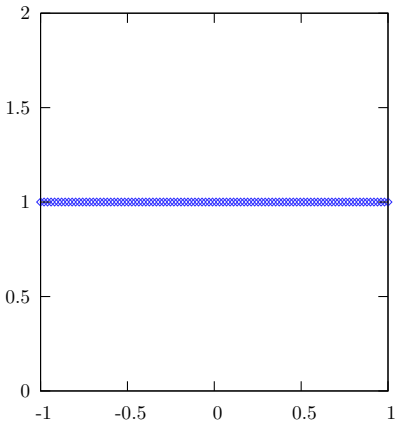


FIGURE 8 – Graphe de f

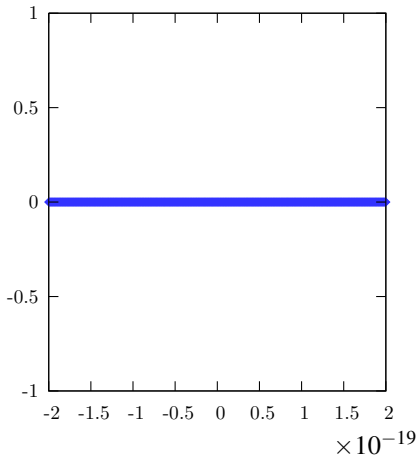


FIGURE 9 – Graphe de f (zoom)

raison pour qu'un nombre avec un exposant négatif soit moins bien représenté qu'un nombre avec un exposant positif. Et puis, l'ordinateur peut traiter des nombres tels que 10^{-307} ; en comparaison, 10^{-19} n'est pas si petit...

Le problème ne vient pas tellement de la petitesse de x que du calcul qu'on lui fait subir. Pour nous rendre compte de ce qui se passe sur un exemple manipulable, considérons une mantisse de trois bits. Le nombre **1** est codé comme $1{,}\mathbf{00}\cdot 2^{\mathbf{0}}$. Prenons $x = \mathbf{0}{,}\mathbf{0001}$. Ce nombre est représentable avec trois digits de mantisse : $x = \mathbf{1}{,}\mathbf{00}\cdot 2^{-\mathbf{100}}$. Alors

$$1 + x = \mathbf{1}{,}\mathbf{0001}$$

Ce résultat n'est *pas* représentable avec une mantisse de trois bits. Il faut donc l'approximer. Le nombre avec une mantisse de trois bits le plus proche de $1+x$ est $\mathbf{1}{,}\mathbf{00}$. Ainsi, l'ordinateur — qui fait de son mieux mais est limité à trois bits de mantisse — pensera que le résultat de l'opération $1+x$ est 1. Ensuite lorsqu'il retirera 1, il obtiendra 0 (au lieu de x) et par conséquent retournera 0 comme valeur de $f(x)$ (au lieu de 1).

Un raisonnement analogue (juste avec une mantisse comportant plus de bits) explique pourquoi l'ordinateur donne 0 comme valeur de $f(0{,}5\cdot 10^{-19})$ au lieu de 1.

Tous nos malheurs viennent du fait que, dans l'expression $1+x$ avec x petit, le x est absorbé dans le 1 et qu'ensuite, lorsqu'on soustrait 1 de cette expression, on voudrait récupérer x mais ce n'est pas possible car il a disparu... En fait le cœur du problème n'est pas tant le fait que x soit absorbé par 1 — après tout l'ordinateur ne peut faire mieux lorsqu'on lui demande de calculer $1+x$ — que le fait qu'on retranche 1 de $1+x$. En effet, $1+x$ et 1 sont des nombres très proches lorsque x est

petit, c'est-à-dire que les premiers digits de leur mantisse sont les mêmes. Dans leur différence, il ne restera donc que les digits à la fin de la mantisse. Mais ce sont précisément ceux qui sont influencés par les arrondis et donc qui contiennent des erreurs...

Ce qu'on vient d'apprendre peut être résumé par le slogan suivant :

> Lorsqu'on calcule avec des nombres en virgule flottante, il faut éviter de soustraire deux nombres très proches car cela peut générer des pertes de précision importantes.

Comment éviter ces pertes de précision ? En transformant algébriquement l'expression à calculer pour faire disparaître la soustraction à problème ! Pour la fonction f ci-dessus, c'est facile : après tout, nous n'avions pas besoin de calculer 1 de façon si alambiquée !

Voici un exemple qui provient d'une situation réelle. Le but est de calculer les premières décimales de π, c'est-à-dire l'aire d'un disque de rayon 1. Pour approcher cette aire on va inscrire dans le disque des polygones réguliers avec un nombre de plus en plus grand de cotés. Plus le nombre de cotés sera grand, plus l'aire du polygone sera proche de celle du disque. Plus spécifiquement, on va commencer par inscrire un carré. Son coté est de longueur $c_1 := \sqrt{2}$ et son aire vaut donc $a_1 := 2$. Ensuite on va doubler le nombre de cotés pour obtenir un polygone à 8 cotés. Notons la longueur de son coté c_2 et son aire a_2. On recommence à doubler le nombre de cotés pour obtenir un polygone de 16 cotés. Écrivons c_3 la longueur de son coté et a_3 son aire. Et ainsi de suite. Par un raisonnement qui sort du cadre de ces notes, on peut établir une dépendance entre c_n, c_{n+1} et a_{n+1} :

$$c_{n+1} = \sqrt{2 - \sqrt{4 - c_n^2}} \qquad\qquad a_{n+1} = 2^n c_n.$$

On veut utiliser ces formules pour estimer π. D'après ce qu'on a dit, plus le nombre de cotés est grand, c'est-à-dire plus n est grand[6], plus l'aire a_n doit se rapprocher de π. Pourtant, si on applique naïvement les formules ci-dessus, on trouve comme résultats ceux du tableau 2. L'aire a_n devient zéro ! Ceci est dû au fait que, quand n est grand, la longueur du coté c_n est proche de zéro et donc

$$\sqrt{4 - c_n^2} \approx \sqrt{4} = 2.$$

Or dans la formule ci-dessus pour obtenir c_{n+1}, on soustrait $\sqrt{4 - c_n^2}$ de 2 ce qui provoque les résultats erronés.

6. Le nombre de cotés ne vaut pas n mais 2^{n+1}.

Pour éviter cette soustraction, nous allons changer en une formule équivalente l'expression de c_{n+1} en fonction de c_n :

$$c_{n+1} = \sqrt{\left(2-\sqrt{4-c_n^2}\right)\frac{2+\sqrt{4-c_n^2}}{2+\sqrt{4-c_n^2}}} = \sqrt{\frac{2^2-(4-c_n^2)}{2+\sqrt{4-c_n^2}}}$$
$$= \sqrt{\frac{c_n^2}{2+\sqrt{4-c_n^2}}} = \frac{c_n}{\sqrt{2+\sqrt{4-c_n^2}}}$$

Cette dernière expression ne contient plus de soustraction problématique. Le tableau 3 est construit en l'utilisant. On voit que la perte de précision a disparu.

4 Pour en savoir plus...

4.1 Ariane 5

http://www.cnes.fr/activites/vehicules/lanceurs/ariane5/1Qualification.htm
http://java.sun.com/people/jag/Ariane5.html
http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

4.2 Missiles ≪ patriot ≫

http://www.math.psu.edu/dna/disasters/patriot.html
http://catless.ncl.ac.uk/Risks/20.85.html#subj2.1

4.3 Collections de bugs

http://gala.univ-perp.fr/∼langlois/rounding_error.html
http://dutita0.twi.tudelft.nl/users/vuik/wi211/disasters.html
http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html

4.4 Sur les calculs ≪ avec virgule ≫

http://www.lahey.com/float.htm
http://www.cs.berkeley.edu/ wkahan/
http://www.ens-lyon.fr/ jmmuller/

n	c_n	a_n
0	2.0000000000	2.0000000000
1	1.4142135624	2.8284271247
2	0.7653668647	3.0614674589
3	0.3901806440	3.1214451523
4	0.1960342807	3.1365484905
5	0.0981353487	3.1403311570
6	0.0490824570	3.1412772509
7	0.0245430766	3.1415138011
8	0.0122717693	3.1415729404
9	0.0061359135	3.1415877253
10	0.0030679604	3.1415914215
11	0.0015339806	3.1415923456
12	0.0007669904	3.1415925765
13	0.0003834952	3.1415926335
14	0.0001917476	3.1415926548
15	0.0000958738	3.1415926453
16	0.0000479369	3.1415926074
17	0.0000239685	3.1415929109
18	0.0000119842	3.1415941252
19	0.0000059921	3.1415965537
20	0.0000029961	3.1415965537
21	0.0000014981	3.1416742650
22	0.0000007491	3.1418296819
23	0.0000003746	3.1424512725
24	0.0000001873	3.1424512725
25	0.0000000942	3.1622776602
26	0.0000000471	3.1622776602
27	0.0000000258	3.4641016151
28	0.0000000149	4.0000000000
29	0.0000000000	0.0000000000
30	0.0000000000	0.0000000000

TABLE 2 – Estimation naïve de π

n	c_n	a_n
0	2.0000000000	2.0000000000
1	1.4142135624	2.8284271247
2	0.7653668647	3.0614674589
3	0.3901806440	3.1214451523
4	0.1960342807	3.1365484905
5	0.0981353487	3.1403311570
6	0.0490824570	3.1412772509
7	0.0245430766	3.1415138011
8	0.0122717693	3.1415729404
9	0.0061359135	3.1415877253
10	0.0030679604	3.1415914215
11	0.0015339806	3.1415923456
12	0.0007669904	3.1415925766
13	0.0003834952	3.1415926343
14	0.0001917476	3.1415926488
15	0.0000958738	3.1415926524
16	0.0000479369	3.1415926533
17	0.0000239684	3.1415926535
18	0.0000119842	3.1415926536
19	0.0000059921	3.1415926536
20	0.0000029961	3.1415926536
21	0.0000014980	3.1415926536
22	0.0000007490	3.1415926536
23	0.0000003745	3.1415926536
24	0.0000001873	3.1415926536
25	0.0000000936	3.1415926536
26	0.0000000468	3.1415926536
27	0.0000000234	3.1415926536
28	0.0000000117	3.1415926536
29	0.0000000059	3.1415926536
30	0.0000000029	3.1415926536

TABLE 3 – Meilleure estimation de π