Exemple de document élaboré

Vincent Goulet

12 février 2015

Table des matières

Chapitre 1

Arithmétique des ordinateurs

1.1 Les ordinateurs ne savent pas compter

Le type de bogue le plus fréquemment rapporté dans les forums de discussion de R a trait au fait que le logiciel ne retourne pas le bon résultat lors d'opérations arithmétiques simples.

On devine que les créateurs de R n'ont pas négligé une fonctionnalité aussi fondamentale pour un langage mathématique que le calcul arithmétique. Les supposées erreurs ci-dessus relèvent toutes de la représentation interne des nombres dans un ordinateur et d'erreurs d'arrondi inhérentes aux opérations arithmétiques avec ces nombres. En effet, un des plus célèbres principes de programmation proposés par Kernighan et Plauger (1978) est :

10,0 fois 0,1 ne donne jamais vraiment 1,0.

D'ailleurs, les auteurs des faux rapports de bogues dans `r-help` sont invariablement renvoyés à l'entrée 7.31 de la foire aux questions [1] de R.

Pour quiconque travaille régulièrement avec un ordinateur pour faire du calcul scientifique, connaître globalement le fonctionnement interne d'un ordinateur peut s'avérer précieux pour les raisons suivantes, entre autres :

— éviter les erreurs d'arrondi et de troncature ;

— éviter les dépassements et soupassements de capacité (*overflow* et *underflow*) ;

— optimiser le code informatique.

1. http://cran.r-project.org/doc/FAQ/R-FAQ.html

Ce chapitre se penche sur les grands principes de l'arithmétique des ordinateurs. En fait, les ordinateurs ne savent pas faire d'arithmétique à proprement parler. Ils ne connaissent que deux états : ouvert (1) et fermé (0). Nous en profiterons donc d'abord pour réviser la conversion des nombres décimaux vers, et de, n'importe quelle base. Après avoir, à la section 1.3, introduit les principales unités de mesure en informatique, nous expliquerons la représentation interne des nombres dans un ordinateur en nous concentrant sur la double précision. Nous pourrons ainsi justifier que les ordinateurs ne peuvent représenter qu'un sous-ensemble des nombres réels, d'où les erreurs d'arrondi et de troncature. La section 1.5 explique comment ces erreurs surviennent. On y donne également quelques pistes pour éviter ou pour diminuer l'impact de ces erreurs. Enfin, le chapitre se clôt par un survol d'un sujet quelque peu périphérique : la représentation interne des caractères.

1.2 Conversion de base

La *base*, dans un système de numération, est le nombre de symboles (habituellement les chiffres) qui pourront servir à exprimer des nombres. L'humain s'est habitué à compter en base 10, le système *décimal*, où les symboles sont $0, 1, \ldots, 9$. De nos jours, la grande majorité des ordinateurs travaillent toutefois en base 2, le système *binaire*. Les autres système d'usage courant, principalement en informatique, sont l'*octal* (base 8) et l'*hexadécimal* (base 16). Dans ce dernier système, les symboles utilisés pour les nombres 10–15 sont généralement les lettres A–F. (C'est pourquoi l'on retrouve ces six lettres sur le pavé des calculatrices scientifiques).

Cette section étudie la conversion des nombres entre la base 10 et une autre base quelconque.

Notation et définitions

De manière générale, soit x un nombre (entier pour le moment) dans la base de numération b composé de m chiffres ou symboles, c'est-à-dire

$$x = x_{m-1}x_{m-2}\cdots x_1x_0,$$

où $0 \leq x_i \leq b-1$. On a donc

$$x = \sum_{i=0}^{m-1} x_i b^i. \tag{1.1}$$

Lorsque le contexte ne permet pas de déterminer avec certitude la base d'un nombre, celle-ci est identifiée en indice du nombre par un nombre décimal. Par exemple, 10011_2 est le nombre binaire 10011.

Exemple 1.1. Soit le nombre décimal 348. Selon la notation ci-dessus, on a $x_0 = 8$, $x_1 = 4$, $x_2 = 3$ et $b = 10$. En effet,

$$348 = 3 \times 10^2 + 4 \times 10^1 + 8 \times 10^0.$$

Ce nombre a les représentations suivantes dans d'autres bases. En binaire :

$$\begin{aligned} 101011100_2 = {} & 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 \\ & + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 \\ & + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0. \end{aligned}$$

En octal :

$$534_8 = 5 \times 8^2 + 3 \times 8^1 + 4 \times 8^0.$$

En hexadécimal :

$$15C_{16} = 1 \times 16^2 + 5 \times 16^1 + 12 \times 16^0.$$

Des représentations ci-dessus, l'hexadécimale est la plus compacte : elle permet de représenter avec un seul symbole un nombre binaire comptant jusqu'à quatre chiffres. C'est, entre autres, pourquoi c'est une représentation populaire en informatique. □

Dans un ordinateur réel (par opposition à théorique), l'espace disponible pour stocker un nombre est fini, c'est-à-dire que $m < \infty$. Le plus grand nombre que l'on peut représenter avec m chiffres ou symboles en base b est

$$\begin{aligned} x_{\max} &= \sum_{i=0}^{m-1} (b-1) b^i \\ &= (b-1) \sum_{i=0}^{m-1} b^i \\ &= (b-1) \left(\frac{b^m - 1}{b-1} \right) \\ &= b^m - 1. \end{aligned}$$

Par exemple, le plus grand nombre décimal représentable avec $m = 4$ symboles est

$$x_{\max} = 9\,999 = 10\,000 - 1 = 10^4 - 1,$$

alors que le plus grand nombre binaire est seulement

$$x_{\max} = 1111 = 10000 - 1 = 2^4 - 1 = 15_{10}.$$

Par une extension naturelle de ce qui précède, un nombre composé de $m \geq 0$ symboles dans sa partie entière et $n \geq 0$ symboles dans sa partie fractionnaire est représenté en base b comme

$$\begin{aligned} x &= x_{m-1}x_{m-2}\cdots x_1x_0,x_{-1}x_{-2}\cdots x_{-n} \\ &= \sum_{i=-n}^{m-1} x_i b^i. \end{aligned}$$

Le symbole qui sépare les parties entière et fractionnaire du nombre est la *séparation fractionnaire* (une virgule en français, un point en anglais).

Conversion vers une base quelconque

Avant de discuter de la conversion de nombres décimaux vers une base quelconque, il convient de définir les notions de *quotient* et de *reste* d'une division.

Le quotient est la partie entière de la division de deux entiers a et d ; sa représentation mathématique habituelle est

$$q = \left\lfloor \frac{a}{d} \right\rfloor, \tag{1.2}$$

où $\lfloor x \rfloor$ est la fonction qui retourne le plus grand entier inférieur ou égal à x. Le reste de la division est simplement la valeur

$$r = a - d \left\lfloor \frac{a}{d} \right\rfloor. \tag{1.3}$$

Évidemment, on a $r \in \{0, 1, \ldots, d-1\}$. Le reste est le résultat de l'opération modulo, notée $r = a \bmod d$.

On remarquera que le premier chiffre en partant de la droite d'un entier décimal est le reste de la division de ce nombre par 10, que le second chiffre est le reste de la division par 10 du quotient de la division précédente, et ainsi de suite. La conversion d'un nombre décimal en une base b implique simplement de diviser par b plutôt que par 10 et de déterminer le symbole dans la base b correspondant au reste de chaque division.

L'algorithme suivant reprend ses idées sous forme plus formelle et en ajoutant le traitement de la partie fractionnaire. Nous démontrerons plus loin qu'il n'est pas réellement nécessaire de savoir effectuer la conversion de la partie fractionnaire d'un nombre réel.

TABLE 1.1 – Conversion du nombre décimal 23,31 en binaire.

(a) partie entière

i	v	$\lfloor v/2 \rfloor$	$v \bmod 2$	x_i
0	23	11	1	1
1	11	5	1	1
2	5	2	1	1
3	2	1	0	0
4	1	0	1	1

(b) partie fractionnaire

i	v	$2v$	$\lfloor 2v \rfloor$	x_{-i}
1	0,31	0,62	0	0
2	0,62	1,24	1	1
3	0,24	0,48	0	0
4	0,48	0,96	0	0
5	0,96	1,92	1	1

Algorithme 1.1 (Conversion de la base 10 vers la base b)**.** *Soit x un nombre réel en base* 10*.*

1. *Poser $i \leftarrow 0$ et $v \leftarrow \lfloor x \rfloor$.*
2. *Répéter les étapes suivantes jusqu'à ce que $v = 0$:*
   a) *Poser $d_i \leftarrow v \bmod b$ et trouver x_i, le symbole dans la base b correspondant à d_i ;*
   b) *Poser $v \leftarrow \lfloor v/b \rfloor$;*
   c) *Poser $i \leftarrow i + 1$.*
3. *Poser $i \leftarrow 1$ et $v \leftarrow x - \lfloor x \rfloor$.*
4. *Répéter les étapes suivantes jusqu'à ce que $v = 0$ ou que $i = n$ (le nombre voulu ou maximal de chiffres après la séparation fractionnaire) :*
   a) *Poser $d_{-i} \leftarrow \lfloor bv \rfloor$ et trouver x_{-i}, le symbole dans la base b correspondant à d_{-i} ;*
   b) *Poser $v \leftarrow bv - d_{-i}$;*
   c) *Poser $i \leftarrow i + 1$.*
5. *Retourner*

$$x_b = x_{m-1}x_{m-2}\cdots x_1x_0{,}x_{-1}x_{-2}\cdots x_{-n}.$$

Exemple 1.2. Soit le nombre décimal 23,31 que l'on convertit en binaire (base 2) avec un maximum de cinq chiffres après la séparation fractionnaire ($n = 5$). Le tableau 1.1 montre le processus en suivant les étapes de l'algorithme 1.1. On obtient le résultat final en combinant la dernière colonne du tableau 1.1 lue de bas en haut avec la dernière colonne du tableau 1.1 lue de haut en bas. Ainsi, la représentation binaire du 23,31 limitée à cinq chiffres après la virgule est 10111,01001.

□

Exemple 1.3. Soit de nouveau la conversion de 23,31 en binaire avec une partie fractionnaire d'au plus cinq chiffres. En suivant les étapes de la remarque ci-dessus, on a :

1. $23{,}31 \times 2^5 = 23{,}31 \times 32 = 745{,}92$;
2. la représentation binaire de $745 = 2^9 + 2^7 + 2^6 + 2^5 + 2^3 + 1$ est 1011101001 ;
3. enfin, en déplaçant la virgule de cinq positions vers la gauche, on obtient 10111,01001, comme à l'exemple 1.2.

□

On tire une conclusion intéressante de la procédure ci-dessus. S'il existe un entier $k \leq n$ tel que la partie fractionnaire de x est un multiple de $1/b^k$, alors la multiplication de x par b^k résultera en un entier. Par conséquent, la représentation de x en base b aura une partie fractionnaire de longueur finie. Ainsi, pour obtenir une représentation binaire finie, la partie fractionnaire d'un nombre réel doit être un multiple de $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ Ce n'était pas le cas à l'exemple 1.3 (la partie fractionnaire étant 0,31), d'où la nécessité de limiter le nombre de chiffres après la virgule.

Conversion en décimal

La conversion d'un nombre en base b vers la base 10 repose essentiellement sur la définition (1.1), mais avec chaque symbole x_i remplacé pour son équivalent décimal. Un algorithme de conversion est le suivant.

Algorithme 1.2 (Conversion de la base b vers la base 10)**.** *Soit x*

$$x_b = x_{m-1}x_{m-2}\cdots x_1x_0{,}x_{-1}x_{-2}\cdots x_{-n}$$

un nombre réel en base b.

1. *Poser $x \leftarrow 0$ et $y \leftarrow 0$.*
2. *Pour $i = m-1, m-2, \ldots, 0$, faire les étapes suivantes :*
   a) *Trouver d_i, le nombre décimal correspondant au symbole x_i ;*
   b) *Poser $x \leftarrow xb + d_i$.*
3. *Pour $i = -n, -n+1, \ldots, -1$, faire les étapes suivantes :*
   a) *Trouver d_i, le nombre décimal correspondant au symbole x_i ;*
   b) *Poser $y \leftarrow (y + d_i)/b$.*
4. *Retourner $x + y$.*

TABLE 1.2 – Conversion du nombre binaire 101011,011 en décimal

(a) partie entière

i	d_i	x
5	101011,011	1
4	101011,011	2
3	101011,011	5
2	101011,011	10
1	101011,011	21
0	101011,011	43

(b) partie fractionnaire

i	d_{-i}	y
3	101011,011	0,5
2	101011,011	0,75
1	101011,011	0,375

On peut, ici aussi, aisément éviter de devoir convertir la partie fractionnaire. Il suffit de déplacer la virgule de n positions vers la droite dans le nombre x_b de manière à obtenir un entier, de convertir ce nombre en décimal avec les étapes 1 et 2 de l'algorithme 1.2 et, finalement, de diviser le nombre obtenu par b^n.

Exemple 1.4. On convertit le nombre binaire 101011,011 en décimal. Le tableau 1.2 illustre le processus de conversion selon l'algorithme 1.2 (le chiffre en surbrillance est celui traité lors de chaque étape de l'algorithme). Le résultat final est 43,375.

De manière équivalente, mais plus simple, on peut déplacer la virgule de trois positions vers la droite pour obtenir l'entier binaire 101011011. Ce nombre est $2^8 + 2^6 + 2^4 + 2^3 + 2 + 1 = 347$ en décimal. Enfin, $347 \div 2^3 = 43{,}375$. □

Exemple 1.5. Soit le nombre hexadécimal AC2,3D8. On peut le convertir en base 10 à partir de la définition :

$$\begin{aligned}\text{AC2,3D8} &= 10 \times 16^2 + 12 \times 16 + 2 + 3 \times 16^{-1} + 13 \times 16^{-2} + 8 \times 16^{-3} \\ &= 2\,754{,}240\,234\,375.\end{aligned}$$

□

Conversion avec des bases générales

Il existe de nombreuses bases de numération d'usage courant où chaque symbole est (potentiellement) exprimé dans une base différente. On a qu'à penser à l'heure ou à l'ensemble du système de mesure impérial. Or, il peut s'avérer utile de savoir convertir de et vers une base quelconque. Le matériel

TABLE 1.3 – Conversion du nombre décimal 91 492 dans la base générale [365 24 60 60].

i	v	b_i	$\lfloor v/b_i \rfloor$	$v \bmod b_i$	x_i
0	91 492	60	1 524	52	52
1	1 524	60	25	24	24
2	25	24	1	1	1
3	1	365	0	1	1

de cette section se retrouve dans très peu d'ouvrages d'analyse numérique ou de statistique numérique. Pourtant, il existe quelques applications intéressantes de la conversion de et vers des bases générales, comme nous le verrons.

On peut généraliser la notion de nombre présentée dans les sections précédentes à une collection de «symboles», chacun dans une base différente. On restreint la discussion aux entiers sans perte de généralité. On a donc

$$x = x_{m-1}x_{m-2}\cdots x_1x_0,$$

où x_{m-1} est un nombre en base b_{m-1}, x_{m-2} est un nombre en base b_{m-2}, etc. Nous dirons que le nombre x est exprimé en base $[b_{m-1}\ b_{m-2}\ \ldots\ b_0]$. On a alors

$$x = \sum_{i=0}^{m-1} x_i \prod_{j=-1}^{i-1} b_j, \tag{1.4}$$

avec $b_{-1} = 1$.

Les algorithmes de conversion 1.1 et 1.2 demeurent essentiellement valides ici. Il suffit de remplacer chaque mention de b par b_i.

Exemple 1.6. Le jour de l'année et l'heure du jour est un «nombre» exprimé en base [365 24 60 60]. En utilisant directement l'équation (1.4), le nombre de secondes correspondant à 1 jour, 2 heures, 33 minutes et 20 secondes est :

$$\begin{aligned} \text{1 j 2 h 33 min 20 sec} &= 1 \times (24)(60)(60) + 2 \times (60)(60) + 33 \times 60 + 20 \\ &= 95\,600 \text{ secondes.} \end{aligned}$$

Si l'on fait la conversion en sens inverse, le nombre de jours, heures, minutes et secondes correspondant à 91 492 secondes est, en utilisant l'algorithme 1.1 avec la base [365 24 60 60] : 1 jour, 1 heure, 24 minutes et 52 secondes (voir le tableau 1.3 pour les calculs). □

Les bases générales sont particulièrement utiles pour assigner ou extraire les éléments d'une matrice ou d'un tableau dans un ordre non séquentiel. Typiquement, l'index de l'élément à traiter est le résultat d'un calcul. L'exemple suivant illustre cette idée.

Exemple 1.7. Soit **A** une matrice 4×5 que l'on suppose remplie en ordre lexicographique (par ligne). Il est simple de déterminer, ici, que le 14^{e} élément est a_{34}. Or, on observe que la conversion du nombre $14 - 1 = 13$ dans la base $[4\ 5]$ donne :

$$\begin{aligned} 13 \div 5 = 2 \text{ reste } 3 \quad &\Rightarrow \quad x_0 = 3 \\ 2 \div 4 = 0 \text{ reste } 2 \quad &\Rightarrow \quad x_1 = 2, \end{aligned}$$

soit le «nombre» $(2, 3)$. En additionnant 1 à ce résultat, on obtient précisément la position du 14^{e} élément dans la matrice. Les opérations -1 et $+1$ sont rendues nécessaires par le fait que les lignes et les colonnes sont numérotées à partir de 1, alors que les systèmes de numération débutent à 0. □

1.3 Unités de mesure

Les ordinateurs que nous utilisons couramment fonctionnent en base 2. La plus petite quantité d'information qu'un ordinateur peut traiter est un *bit* (compression de l'anglais *binary digit*). En informatique, un bit est égal à $\boxed{0}$ ou à $\boxed{1}$. Le symbole du bit est b.

Un *octet* (*byte*) est un groupe de huit bits. Son symbole est o ou B. L'octet est l'unité de mesure la plus fréquemment utilisée en informatique, en grande partie parce que le codage d'un caractère dans la plupart des langues occidentales requiert un octet ; voir la section 1.6. Les termes pour de plus grandes quantités de bits ou d'octets utilisent généralement les préfixes usuels du système international d'unités. Par exemple :

— 1 kilooctet (ko) est $2^{10} = 1\,024$ octets ;
— 1 mégaoctet (Mo) est $2^{20} = 1\,048\,576$ octets ;
— 1 gigaoctet (Go) est $2^{30} = 1\,073\,741\,824$ octets.

On voit que cette pratique fort répandue entraîne des distorsions par rapport aux définitions usuelles de kilo, méga, giga, etc., et que cette distorsion s'amplifie au fur et à mesure que l'on grimpe dans l'échelle des tailles d'objets. Pour cette raison, on a défini les préfixes kibi, mébi, gibi, etc., mais ceux-ci demeurent peu utilisés à ce jour.

Dans l'industrie de l'informatique, on joue beaucoup avec les unités pour montrer son produit sous un jour favorable. Deux exemples :

1. Les fournisseurs d'accès Internet expriment la vitesse de téléchargement en Mb/sec, alors que la taille des fichiers est généralement affichée en octets. Il faut donc diviser par 8 la vitesse annoncée pour avoir une idée plus juste du temps requis pour télécharger un fichier.
2. Les fabricants de disques durs divisent la capacité réelle de leurs disques par 10^9 pour l'exprimer en gigaoctets. Ainsi, un disque dont la capacité annoncée est de 100 Go ne peut contenir, en fait, que $100 \times 10^9 \div 2^{30} =$ 93,132 Go de données. Cette confusion serait éliminée si les fabricants affichaient plutôt une capacité de 93,132 gibioctets. Moins vendeur...

1.4 Représentation en virgule flottante

Cette section décrit la manière standard de représenter les nombres réels dans les ordinateurs d'usage courant. Il est utile de connaître les grandes lignes afin de comprendre pourquoi les nombres réels n'ont pas tous une représentation exacte dans un ordinateur et comment surviennent et se propagent les erreurs d'arrondi. L'étude du sujet est également intéressante en soi pour constater comment les ingénieurs et les informaticiens sont parvenus à stocker un maximum d'information dans un espace limité.

Tel que mentionné précédemment, la capacité de stockage d'un ordinateur, bien que vaste de nos jours, n'en demeure pas moins limitée. Par conséquent, les nombres y sont représentés par un ensemble de m bits. Habituellement, m est un multiple de 2 et sa valeur détermine le type de nombre. Par exemple, des définitions usuelles sont $m = 2^4 = 16$ pour un entier, $m = 2^5 = 32$ pour un nombre réel en simple précision (*float* dans plusieurs langages de programmation) et $m = 2^6 = 64$ pour un nombre réel en double précision (*double*).

Il existe deux grandes façons de représenter les nombres (réels) à l'aide de m bits.

Virgule fixe Dans cette représentation, la position de la virgule dans le nombre est prédéterminée et, par conséquent, n'a pas besoin d'être conservée en mémoire. On réserve alors m_e positions pour la partie entière et m_f pour la partie fractionnaire (avec $m = m_e + m_f$). La représentation en virgule fixe est particulièrement utile dans les applications financières.

Virgule flottante Dans la représentation des nombres en virgule flottante, celle-ci peut être placée n'importe où dans le nombre. L'étendue de cette représentation est beaucoup plus grande que la virgule fixe, mais ceci au détriment de la précision puisque quelques bits doivent être réservés pour stocker la position de la virgule dans le nombre.

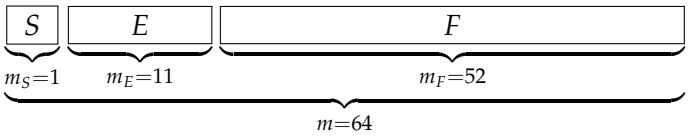

FIGURE 1.1 – Représentation schématique d'un nombre en double précision dans la norme IEEE 754.

La représentation en virgule flottante est celle utilisée dans les applications scientifiques et celle sur laquelle nous nous concentrons dans la suite.

La représentation en virgule flottante est analogue à la notation scientifique. Le nombre réel x est entièrement défini par un bit de signe S, un exposant positif E et une mantisse M tel que

$$x = (-1)^S \times B^{E-e} \times M, \tag{1.5}$$

où B est la base de la représentation et e est un entier prédéterminé appelé le décalage, ou le biais, de l'exposant. Le recours au décalage facilite les calculs internes et évite de devoir réserver un autre bit pour le signe de l'exposant en le stockant sous forme non signée (c'est-à-dire sous forme d'entier positif).

La norme IEEE 754 définit la manière standard de représenter les nombres en virgule flottante dans les ordinateurs (IEEE, 2003; Wikipedia, 2012, pour une excellente présentation). En premier lieu, la norme stipule que la base dans la représentation est $B = 2$. Ceci est implicite et n'est pas stocké où que ce soit dans l'ordinateur. En second lieu, la norme suppose que, pour les nombres dits *normalisés*, la mantisse est toujours de la forme $M = 1{,}F$, où F est un entier binaire. Le bit à gauche de la virgule est appelé le *bit caché* puisqu'il est lui aussi implicite et non stocké dans l'ordinateur.

Nous décrivons plus en détail la norme pour les nombres réels en *double précision* puisque c'est le type avec lequel R travaille toujours [2]. Tout d'abord, un nombre en double précision est stocké dans un mot de $m = 64$ bits (8 octets) divisé ainsi de gauche à droite : $m_S = 1$ bit pour le signe, $m_E = 11$ bits pour l'exposant et $m_F = 52$ bits pour la partie fractionnaire de la mantisse. Le premier bit du mot utilisé pour le signe est appelé le *bit fort*. Voir la figure 1.1 pour une représentation schématique.

2. Il est possible de créer des vrais entiers dans R en ajoutant un suffixe `L` à un nombre entier. Ainsi, `1L` est un nombre entier dans le sens où `is.integer(1L)` est `TRUE`. Cela est relativement peu utilisé en programmation normale.

TABLE 1.4 – Valeurs spéciales dans la norme IEEE 754

S	E	F	Représentation binaire	Valeur spéciale
0	0	0	0 | 00000000000 | $00\cdots00$	0
1	0	0	1 | 00000000000 | $00\cdots00$	-0
0	2 047	0	0 | 11111111111 | $00\cdots00$	$+\infty$
1	2 047	0	1 | 11111111111 | $00\cdots00$	$-\infty$
0 ou 1	2 047	$F \neq 0$	S | 11111111111 | F	`NaN`[a]

[a] *Not a number*, par exemple $\frac{0}{0}$ ou $\frac{\infty}{\infty}$.

On remarque que le bit caché confère à la mantisse une longueur de mot effective (ou précision) de 53 bits.

Le décalage est $e = 2^{m_e-1} - 1 = 2^{10} - 1 = 1\,023$. Avec une longueur de mot de 11 bits, les valeurs possibles de E vont de 0 à $2^{11} - 1 = 2\,047$. Cependant, les valeurs 0 et 2 047 sont réservées pour des usages spéciaux ; voir le tableau 1.4. Par conséquent, les valeurs possibles de l'exposant (une fois décalé) pour les nombres en double précision vont de $-1\,022$ à $1\,023$.

Exemple 1.8. Tel qu'expliqué à la section 1.2, le nombre $\frac{1}{2}$ possède une représentation binaire exacte :

$$\begin{aligned}\frac{1}{2} &= 2^{-1} \\ &= (-1)^0 \times 2^{1\,022-1\,023} \times 1{,}0.\end{aligned}$$

Puisque la représentation de l'exposant est $1\,022_{10} = 1111111110_2$, la représentation interne de $\frac{1}{2}$ selon la norme IEEE 754 est

$$\boxed{0}\,\boxed{01111111110}\,\boxed{00\cdots00}.$$

En revanche, le nombre $\frac{1}{10}$ n'a pas une représentation binaire finie. En utilisant l'algorithme 1.1, on trouve (le côté droit de l'égalité étant en binaire) :

$$\begin{aligned}\frac{1}{10} &= 0{,}0001100110011001100110011001\ldots \\ &= 2^{-4} \times 1{,}1001100110011001\ldots \\ &= (-1)^0 \times 2^{1\,019-1\,023} \times 1{,}1001100110011001\ldots\end{aligned}$$

Or, $1\,019_{10} = 1111111011_2$, d'où la représentation interne finie de $\frac{1}{10}$ est

$$\boxed{0}\boxed{01111111011}\boxed{1001\cdots1001}.$$

Si l'on reconvertit ce nombre en décimal, on obtient

$$2^{-4}\times\left(1+\sum_{k=0}^{12}(2^{-4k-1}+2^{-4k-4})\right),$$

un nombre près de, mais pas exactement égal à, $\frac{1}{10}$ (le programme de calcul en multiprécision `bc` donne 0,0999999999999999166). Dès lors, la multiplication de ce nombre par 10 ne donnera pas 1. Voilà qui justifie le principe de programmation énoncé à la page 1. □

Les principales caractéristiques des nombres en double précision sont les suivantes.

1. Soit $x_{\max}$ le plus grand nombre représentable. Parce que l'exposant $E = 2\,047$ est réservé, on a

$$\begin{aligned}x_{\max} &= \boxed{0}\boxed{11111111110}\boxed{11\cdots11}\\ &= (-1)^0\times 2^{1\,023}\times 1{,}11\cdots11\\ &= 2^{1\,023}\times(2-2^{-52})\\ &= 1{,}797\,693\times10^{308}.\end{aligned}$$

2. Soit $x_{\min}$ le plus petit nombre normalisé représentable. Parce que l'exposant $E = 0$ est réservé, on a

$$\begin{aligned}x_{\min} &= \boxed{0}\boxed{00000000001}\boxed{00\cdots00}\\ &= (-1)^0\times 2^{-1\,022}\times 1{,}00\cdots00\\ &= 2^{-1\,022}\\ &= 2{,}225\,074\times10^{-308}.\end{aligned}$$

   Afin de rendre la transition vers 0 moins abrupte, la norme IEEE 754 définit également des nombres *dénormalisés*. Ceux-ci sont identifiés par $E = 0$ et une partie fractionnaire F non nulle. Cependant, par définition, les nombres dénormalisés ont $E - e = -1\,022$ et leur bit caché est 0. Par conséquent, le plus petit nombre dénormalisé est stocké sous la forme

$$\boxed{0}\boxed{00000000000}\boxed{00\cdots01}$$

et sa valeur est

$$\begin{aligned} x_{\min} &= (-1)^0 \times 2^{-1\,022} \times 0{,}00\cdots01 \\ &= 2^{-1\,022} \times 2^{-52} \\ &= 2^{-1\,074} \\ &= 4{,}940\,656 \times 10^{-324}. \end{aligned}$$

3. On a la même étendue pour les nombres négatifs, soit

$$[-1{,}797\,693 \times 10^{308}, -2{,}225\,074 \times 10^{-308}]$$

ou

$$[-1{,}797\,693 \times 10^{308}, -4{,}940\,656 \times 10^{-324}].$$

en considérant les nombres dénormalisés.

4. Soit ε la plus petite valeur tel que $1 + \varepsilon \neq 1$ dans la représentation en virgule flottante. Cette valeur est appelée l'*epsilon de la machine* ou la *précision de la machine*. Or, puisque

$$1 = (-1)^0 \times 2^0 \times 1{,}00\cdots00,$$

le nombre suivant est

$$(-1)^0 \times 2^0 \times 1{,}00\cdots01.$$

Par conséquent, on a $\varepsilon = 2^{-52} = 2{,}220\,446 \times 10^{-16}$.

5. Tout nombre x représente en fait un intervalle de $\mathbb{R}$. Par exemple, tout nombre dans l'intervalle $[1, 1 + \varepsilon)$ est représenté par le nombre 1 dans l'ordinateur.

6. On a un ensemble fini de nombres pour représenter $\mathbb{R}$. Or, cet ensemble est plus dense près de 0 que pour les grandes valeurs. En effet, le nombre suivant $x_{\min}$ est

$$\begin{aligned} x^+_{\min} &= (1 + \varepsilon) \times 2^{-1\,022} \\ &= x_{\min} + \varepsilon \times 2^{-1\,022} \\ &= x_{\min} + 2^{-1\,074}, \end{aligned}$$

alors que le nombre précédant $x_{\max}$ est

$$\begin{aligned} x_{\max}^{-} &= (2 - 2^{-52} - \varepsilon) \times 2^{1\,023} \\ &= x_{\max} - \varepsilon \times 2^{1\,023} \\ &= x_{\max} - 2^{971}. \end{aligned}$$

L'écart entre les deux plus petits nombres représentables est donc de $2^{-1\,024} \approx 10^{-324}$, alors que celui entre les deux plus grands est de $2^{971} \approx 10^{292}$!

Tout calcul impliquant un nombre $x > x_{\max}$ ($x < -x_{\max}$) entraîne un *dépassement de capacité*. Habituellement, cela entraîne l'arrêt immédiat des calculs et un résultat de $+\infty$ ($-\infty$). À l'autre bout du spectre, un calcul impliquant un nombre $x < x_{\min}$ entraîne un *soupassement de capacité* et le résultat est habituellement considéré égal à 0.

R conserve dans une liste nommée `.Machine` les valeurs mentionnées ci-dessus — ainsi que plusieurs autres — pour l'architecture de l'ordinateur courant. Par exemple, on confirme les valeurs de ε_m, $x_{\min}$, $x_{\max}$, B, m_f et m_e, dans l'ordre, pour l'architecture x86 :

```
> .Machine[c(1, 3:6, 11)]

$double.eps
[1] 2.220446e-16

$double.xmin
[1] 2.225074e-308

$double.xmax
[1] 1.797693e+308

$double.base
[1] 2

$double.digits
[1] 53

$double.exponent
[1] 11
```

TABLE 1.5 – Représentation en virgule flottante simplifiée

x	$\mathrm{fl}(x)$
2,5	$0{,}25000 \times 10^{1}$
$-42{,}182$	$-0{,}42182 \times 10^{2}$
0,214356	$0{,}21436 \times 10^{0}$

1.5 Éléments d'arithmétique en virgule flottante

La section précédente expliquait pourquoi les nombres stockés dans un ordinateur ne sont pas toujours ceux que l'on croit — ou que l'on voudrait. Puisque l'ordinateur ne peut représenter tous les nombres réels, toute opération arithmétique avec des nombres en virgule flottante implique une erreur d'arrondi ou de troncature (selon l'architecture de l'ordinateur) dont il importe de tenir compte lors de la mise en oeuvre de certains algorithmes. Cette section présente quelques grands principes d'arithmétique en virgule flottante ainsi que de bons usages pour minimiser les erreurs d'arrondi. Consulter les ouvrages Monahan (2001); Burden et Faires (2011); Knuth (1997), entre autres, pour une discussion plus complète de ce vaste sujet.

Aux fins de cette section, on note $\mathrm{fl}(x)$ la représentation en virgule flottante du nombre nombre x et l'on définit la représentation simplifiée

$$\mathrm{fl}(x) = (-1)^{S} \times 10^{E} \times 0{,}F, \tag{1.6}$$

où F compte cinq chiffres significatifs, dont le dernier est arrondi. Le tableau 1.5 fournit quelques exemples.

Remarque. Dans la norme IEEE 754, les règles d'arrondi de base sont :

1. arrondir à la valeur la plus près (0,14 devient 0,1 et 0,16 devient 0,2) ;
2. arrondir un nombre exactement à mi-chemin au nombre avec un dernier chiffre significatif pair ou nul (0,05 devient 0,0 alors que 0,15 devient 0,2).

Erreurs d'arrondi ou de troncature

Lors de l'addition et de la soustraction de nombres en virgule flottante, on rend les exposants égaux en déplaçant les bits de la mantisse du plus petit nombre vers la droite. Il en résulte une perte de bits significatifs et donc une *erreur d'arrondi*. Dans les cas extrêmes où les deux opérandes sont de

grandeur très différente, le plus petit opérande devient nul suite à la perte de tous ses bits significatifs.

La situation pour la multiplication et la division est quelque peu différente, mais la source d'erreur d'arrondi demeure la même.

À toute fin pratique, tout calcul fait naître de l'erreur d'arrondi et celle-ci augmente avec le nombre d'opérations. Les quelques principes de programmation ci-dessous, lorsque suivis par le programmeur prudent et consciencieux, permettent d'atténuer l'impact de l'erreur d'arrondi.

1. L'addition et la soustraction en virgule flottante ne sont pas associatives. Additionner les nombres en ordre croissant de grandeur afin d'accumuler des bits significatifs.
2. Éviter de soustraire un petit nombre d'un grand, ou alors le faire le plus tard possible dans la chaîne des calculs.
3. Pour calculer une somme alternée, additionner tous les termes positifs et tous les termes négatifs, puis faire la soustraction.
4. Multiplier et diviser des nombres d'un même ordre de grandeur. Si x et y sont presque égaux, le calcul x/y sera plus précis en virgule flottante que $y^{-1}x$.

Les exemples ci-dessous illustrent ces principes.

Exemple 1.9. Soit $x = 7$, $y = 4$ et $z = 100\,000$. Dans la représentation simplifiée (1.6), on a $\mathrm{fl}(x) = 0{,}70000 \times 10^1$, $\mathrm{fl}(y) = 0{,}40000 \times 10^1$ et $\mathrm{fl}(z) = 0{,}10000 \times 10^6$. Or, $x + y + z = 100\,011$ et

$$\begin{aligned}[\mathrm{fl}(x) + \mathrm{fl}(y)] + \mathrm{fl}(z) &= (0{,}70000 \times 10^1 + 0{,}40000 \times 10^1) + 0{,}10000 \times 10^6 \\ &= 0{,}11000 \times 10^2 + 0{,}10000 \times 10^6 \\ &= 0{,}00001 \times 10^6 + 0{,}10000 \times 10^6 \\ &= 0{,}10001 \times 10^6,\end{aligned}$$

soit un résultat raisonnablement précis. Cependant,

$$\begin{aligned}\mathrm{fl}(x) + [\mathrm{fl}(y) + \mathrm{fl}(z)] &= 0{,}70000 \times 10^1 + (0{,}00000 \times 10^6 + 0{,}10000 \times 10^6) \\ &= 0{,}00000 \times 10^6 + 0{,}10000 \times 10^6 \\ &= 0{,}10000 \times 10^6.\end{aligned}$$

En effectuant les opérations dans un mauvais ordre, on obtient $x + y + z = z$ même si $x \neq 0$ et $y \neq 0$. □

Exemple 1.10. Soit $x = 7$, $y = 100\,006$ et $z = 100\,002$. On a $z - y - x = -11$. Or,

$$\begin{aligned}\mathrm{fl}(z) - [\mathrm{fl}(y) + \mathrm{fl}(x)] &= 0{,}10000 \times 10^6 - (0{,}10000 \times 10^6 + 0{,}00000 \times 10^6) \\ &= 0{,}00000 \times 10^0,\end{aligned}$$

alors que

$$\begin{aligned}[\mathrm{fl}(z) - \mathrm{fl}(y)] - \mathrm{fl}(x) &= (0{,}10000 \times 10^6 - 0{,}10000 \times 10^6) - 0{,}70000 \times 10^1 \\ &= 0{,}00000 \times 10^1 - 0{,}70000 \times 10^1 \\ &= -0{,}70000 \times 10^1.\end{aligned}$$

□

L'exemple suivant est adapté de Monahan (2001).

Exemple 1.11. L'évaluation numérique de probabilités loin dans les queues d'une fonction de répartition peut s'avérer imprécise selon la technique employée. Par exemple, la fonction de répartition de la distribution logistique est

$$F(x) = (1 + e^{-x})^{-1}.$$

La vraie valeur de $\Pr[X > 6]$ est $1 - F(6) = 1 - (1 + e^{-6})^{-1} = 0{,}002\,472\,623$. L'évaluation de cette probabilité dans notre représentation en virgule flottante avec la formule ci-dessus est

$$\begin{aligned}1 - [1 \div (1 + \mathrm{fl}(e^{-6}))] &= 1 - [1 \div (1 + 0{,}24788 \times 10^{-2})] \\ &= 1 - [0{,}10000 \times 10^1 \div 0{,}10025 \times 10^1)] \\ &= 1 - 0{,}99751 \times 10^0 \\ &= 0{,}10000 \times 10^1 - 0{,}09975 \times 10^1 \\ &= 0{,}25000 \times 10^{-2}.\end{aligned}$$

(Nous n'avons pas écrit les entiers en virgule flottante ci-dessus pour alléger la notation.) Toutefois, si l'on utilise plutôt la formule équivalente

$$\begin{aligned}1 - F(6) &= \frac{e^{-6}}{1 + e^{-6}} \\ &= \frac{1}{1 + e^6}\end{aligned}$$

dans laquelle l'opération de soustraction entre deux nombres presque égaux est déjà effectuée, on obtient le résultat bien plus précis

$$\begin{aligned} 1 \div (1 + \mathrm{fl}(e^6)) &= 1 \div (1 + 0{,}40343 \times 10^3) \\ &= 1 \div (0{,}00100 \times 10^3 + 0{,}40343 \times 10^3) \\ &= 1 \div 0{,}40443 \times 10^3 \\ &= 0{,}24726 \times 10^{-2}. \end{aligned}$$

□

Les deux principes ci-dessous permettent d'éviter des dépassements ou des soupassements de capacité et d'améliorer la précision des calculs.

1. Chercher des formules mathématiques équivalentes évitant de devoir traiter des très grands ou des très petits nombres.
2. Travailler en échelle logarithmique. Par exemple, le produit de deux grands nombres x et y dépassera la capacité plus tard et demeurera précis plus longtemps s'il est calculé sous la forme $e^{\log x + \log y}$.

Exemple 1.12. Soit $X_1, \ldots, X_n$ un échantillon aléatoire tiré d'une distribution de Pareto translatée, dont la fonction de répartition est

$$F(x; \mu, \alpha) = 1 - \left(\frac{\mu}{x}\right)^{\alpha}, \quad x > \mu.$$

On peut démontrer que l'estimateur du maximum de vraisemblance de α est

$$\hat{\alpha} = \frac{n}{\ln(X_1 \cdots X_n / X_{(1)}^n)},$$

où $X_{(1)} = \min(X_1, \ldots, X_n)$. Soit `x` un objet R contenant un échantillon de 1 000 valeurs d'une distribution Pareto translatée de moyenne 5 000 (le contenu de cet objet n'est pas affiché ici pour des raisons évidentes). Le calcul de l'estimateur $\hat{\alpha}$ directement par la formule entraîne rapidement un dépassement de capacité, tant lors du produit $X_1 \cdots X_n$ que lors de l'opération $X_{(1)}^n$:

```
> prod(x)

[1] Inf

> min(x)^length(x)

[1] Inf
```

Le résultat est donc `NaN` :

```
> length(x)/log(prod(x)/min(x)^length(x))

[1] NaN
```

Selon la grandeur des données dans l'échantillon, la formule équivalente

$$\hat{\alpha} = \frac{n}{\ln \prod_{i=1}^{n} X_i / X_{(1)}}$$

peut éviter le dépassement de capacité. Elle n'est toutefois d'aucun secours dans le présent exemple :

```
> length(x)/log(prod(x/min(x)))

[1] 0
```

On remarquera de plus que cette approche nécessite un grand nombre de multiplications (voir la section 1.5), en plus d'ouvrir la porte à des erreurs d'arrondi.

Pour obtenir une réponse on utilisera plutôt une autre formule algébriquement équivalente :

$$\hat{\alpha} = \frac{n}{\sum_{i=1}^{n} \ln X_i - n \ln X_{(1)}}.$$

On obtient alors

```
> length(x)/(sum(log(x)) - length(x) * log(min(x)))

[1] 1.405529
```

Cet exemple illustre combien des calculs *algébriquement* équivalents ne sont pas nécessairement *numériquement* équivalents. □

Coût des opérations

Les ordinateurs modernes disposent d'une unité de calcul en virgule flottante (FPU) intégrée au processeur. Cette unité est évidemment très optimisée et, par conséquent, elle accélère beaucoup le calcul en virgule flottante. Néanmoins, certaines opérations sont plus coûteuses à réaliser que d'autres en termes de temps de calcul. À titre indicatif, on trouve au tableau 1.6 le coût relatif approximatif de quelques opérations en virgule flottante.

TABLE 1.6 – Coût relatif de quelques opérations en virgule flottante

Opération arithmétique	Coût relatif
Addition et soustraction	1,0
Multiplication	1,3
Division	3,0
Racine carrée	4,0
Logarithme	15,4

À titre d'exemple, considérons le calcul d'une simple moyenne arithmétique

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

Si l'on effectue le calcul tel qu'écrit ci-dessus, cela requiert $n - 1$ additions et une division. En revanche, le calcul équivalent

$$\bar{x} = \sum_{i=1}^{n} \frac{x_i}{n},$$

requiert $n - 1$ divisions et $n - 1$ additions. Pour n grand utiliser la première approche plutôt que la seconde fera une différence.

1.6 Codage de caractères

Pendant que l'on discute de la représentation interne des nombres dans un ordinateur, on peut toucher un mot de la représentation interne, ou le codage, des caractères. Ce sujet demeure une cible mouvante puisque de nouveaux standards apparaissent encore après de nombreuses années de systèmes incompatibles et basés sur la langue anglaise.

Peu importe le système retenu, les caractères doivent être codés en binaire dans l'ordinateur. La partie la plus difficile consiste à créer un système standard permettant aux ordinateurs et autres appareils numériques de communiquer entre eux. Le premier standard d'usage courant fut le Code américain normalisé pour l'échange d'information, mieux connu sous son acronyme ASCII (ANSI, 1986). La norme ASCII définit des codes numériques pour 128 caractères, soit 95 affichables (lettres, chiffres, symboles divers et l'espace) et 33 non affichables (essentiellement des caractères de contrôle tels que saut de ligne, retour de chariot ou espacement arrière).

À titre d'exemple, les lettres majuscules A–Z occupent les cases 65–90 (1000001–1011010 en binaire) alors que les lettres minuscules a–z occupent les cases 97–122 (1100001–1101010 en binaire). On constate que les versions majuscule et minuscule d'une même lettre ne diffèrent, dans leur représentation binaire, que par leur second bit le plus significatif. Le changement de casse d'une lettre ou d'un mot est donc une opération très simple et rapide.

La représentation ASCII ne requiert en soi que sept bits. Néanmoins, on a rapidement codé les caractères sur un octet (huit bits) puisqu'il s'agit du type de donnée natif des ordinateurs depuis les années 1970. L'ajout d'un bit a créé de l'espace pour 128 caractères additionnels (cases 128–255). Une myriade de systèmes de codage différents sont alors apparus pour supporter les caractères des langues autres que l'anglais (les caractères accentués, par exemple) ainsi que d'autres symboles graphiques. La norme ISO 8859-1, ou Latin 1 (ISO, 1998), a fini par s'imposer comme l'un des standards les plus répandus. Ces listes de codes fixes se révèlent toutefois problématiques lors de l'apparition d'un nouveau symbole. Par exemple, pour faire de la place pour le symbole de l'euro à la fin des années 1990, il a fallu retirer un symbole de ISO 8859-1. Avec quelques autres changements, la nouvelle liste de code est devenue ISO 8859-15. De plus, comment doit-on traiter les langues CJC (chinois, japonais, coréen) qui comptent des milliers de symboles différents ?

Depuis le début des années 1990, la mondialisation de l'informatique a suscité un effort concerté de création d'un système de codage standard couvrant la presque totalité des systèmes d'écriture du monde. Le système devait aussi permettre la composition de textes en plusieurs langues, par exemple en français et dans une écriture de droite à gauche comme l'hébreu ou l'arabe. Le standard ainsi créé est Unicode (Unicode Consortium, 2007). Le plus populaire système de codage capable de représenter l'ensemble des caractères Unicode est *Unicode Transformation Format* 8 bits, ou UTF-8 (Unicode Consortium, 2007, section 3.9). Dans l'UTF-8, chaque caractère est codé sur une suite d'un à quatre octets et les premiers 128 caractères sont identiques à la norme ASCII. L'UTF-8 est le système de codage par défaut dans MacOS X et les plus récentes distributions GNU/Linux.

Le système R supporte les caractères multi-octets de manière assez exhaustive. Sans doute aidés en cela par le fait qu'il s'agit d'un projet international, les développeurs de R ont mis beaucoup d'effort pour assurer l'internationalisation et la localisation du logiciel ; voir Ripley (2005).

Bibliographie

ANSI. 1986, *ANSI X3.4-1986: Coded Character Set. 7-Bit American Standard Code for Information Interchange*, American National Standards Institute, New York.

Burden, R. L. et J. D. Faires. 2011, *Numerical Analysis*, 9e éd., Brooks/Cole, ISBN 978-0-5387335-1-9.

Unicode Consortium, T. 2007, *The Unicode Standard, Version 5.0.0*, Addison-Wesley, Boston, ISBN 0-32148091-0.

IEEE. 2003, *754-1985 IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, Piscataway, NJ.

ISO. 1998, *ISO/IEC 8859-1:1998. Information Technology – 8-bit Single-Byte Coded Graphic Character Sets – Part 1: Latin Alphabet No. 1*, International Organization for Standardization, Geneva.

Kernighan, B. W. et P. J. Plauger. 1978, *The Elements of Programming Style*, 2e éd., McGraw-Hill, ISBN ISBN 0-07034207-5.

Knuth, D. E. 1997, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, MA.

Monahan, J. F. 2001, *Numerical Methods of Statistics*, Cambridge University Press, Cambridge, UK, ISBN 0-52179168-5.

Ripley, B. D. 2005, «Internationalization features of R 2.1.0», *R News*, vol. 5, no 1, p. 2–7. URL http://cran.r-project.org/doc/Rnews/.

Wikipedia. 2012, «IEEE 754-2008 — Wikipedia, the free encyclopedia», URL http://en.wikipedia.org/wiki/IEEE_floating-point_standard.