Informatique appliquée au calcul scientifique

Alexis Herault

Table des matières

Codage de l'information et algorithmique

Chapitre I

Représentation des nombres en informatique

Quelques soient les moyens de calcul informatiques mis en œuvre, il appartiennent au domaine du discret et du fini. Ainsi il ne peuvent manipuler et représenter qu'un nombre fini (même s'il est très grand) d'informations. En revanche, en mathématiques nous utilisons couramment les concepts d'exactitude et d'infini. L'utilisation de l'outil informatique impose donc d'adopter des représentations fines pour les quantités numériques (entier, réels, ...) et introduit forcément un erreur due à ces représentations.

1 Représentation des entiers dans une base quelconque

Une fois l'ensemble des entiers naturel construit, cette construction étant en dehors de l'objet de cours, il est nécessaire d'adopter un représentation pour les entiers ne serais-ce que pour les écrire et effectuer des calculs.

Proposition 1 : *Tout entier naturel n peut s'écrire de manière unique :*
$n = \alpha_p b^p + \alpha_{p-1} b^{p-1} + \cdots + \alpha_1 b^1 + \alpha_0 b^0$ *avec* $0 \leq \alpha_i \leq b-1$
b s'appelle la base de numération et $b > 1$.

Déf. : *$\alpha_p \cdots \alpha_1 \alpha_{0_b}$ constitue le développement de n en base b. Quand il n'y a pas d'ambiguïté possible on peut omettre l'indice b.*

Proposition 2 : *Le digit le moins significatif du développement de n en base b, α_0 est égal au reste de la division de n par b.*

Démonstration. Cela résulte immédiatement de la définition de la division euclidienne de n par b : $n = b.q + r = qb^1 + rb^0$ avec $0 \leq r < b$. Par identification avec $n = \alpha_p b^p + \alpha_{p-1} b^{p-1} + \cdots + \alpha_1 b^1 + \alpha_0 b^0$ on obtient immédiatement $\alpha_0 = r$. □

En appliquant de manière itérative ce résultat au quotient de l'étape précédente, on obtient de proche en proches tous les termes du développement de n dans la base b.

Jusqu'à la base 10 les α_i peuvent être représentés par des chiffres, au delà de la base 10 il faudra, pour éviter toute confusion, utiliser des symboles. Dans ce cas une fois les dix chiffres (0,1,...,9) épuisés, on utilise les lettres $A, B, \cdots$.

Parmi l'infinité de représentations possibles certaines revêtent une importance particulière dans la vie courante, représentation en base 10, ou en informatique, représentation binaire (base 2) ou hexadécimales (base 16). Nous nous intéresserons donc essentiellement à ces trois représentations ainsi qu'au passages des unes vers les autres.

1.1 La base 10 : écriture décimale

Le développement en base 10, ou écriture décimale, est le système de numération le plus répandu actuellement et auquel vous êtes tous habitués. L'utilisation de la base 10 provient très probablement du fait que nous ayons 10 doigts ; si nous avions 12 doigts nous compterions en base 12 !

Il n'en a pas toujours été ainsi et vous trouverez ci dessous une liste, non exhaustive, des différents systèmes de numération utilisé à travers les époques et les peuples :

- le système binaire (base 2) utilisé en d'Amérique du Sud et en Océanie
- le système quinaire (base 5) était utilisé parmi les premières civilisations, et jusqu'au XX$^{\text{e}}$ siècle par des peuples africains, mais aussi, partiellement, dans les notations romaine et maya
- le système octal (base 8) utilisé au Mexique.

– le système décimal (base 10) a été utilisé par de nombreuses civilisations, comme les Chinois dès les premiers temps, et, probablement, les Proto-indo-européens. Aujourd'hui, il est de loin le plus répandu.
– le système duodécimal (base 12) utilisé au Népal. On le retrouve, à cause de ses avantages en matière de divisibilité (par 2, 3, 4, 6), pour un certain nombre de monnaies et d'unités de compte courantes en Europe au Moyen Âge, partiellement dans les pays anglo-saxons dans le système d'unité impérial, et dans le commerce. Notre façon de compter les mois et les heures est un vestige de son utilisation.
– le système sexagésimal (base 60) était utilisé pour la numération babylonienne, ainsi que par les Indiens et les Arabes en trigonométrie. Il sert actuellement dans la mesure du temps et des angles.

1.2 La base 2 : écriture binaire

Le binaire, ou base 2, est la base naturelle de représentation d'un nombre en informatique. En effet, dans un ordinateur, les informations (donc aussi les nombres) sont stockées sous forme de suites de bit (binary digit), chaque bit valant 0 ou 1. Le bit est donc l'unité élémentaire d'information en informatique.
D'après la proposition 1, le développement binaire d'un entier naturel n est :

$$n = \sum_{i=0}^{p} \alpha_i 2^i \text{ avec } 0 \leq \alpha_i \leq 1$$

et son écriture binaire est donc :

$$n = \alpha_p \cdots \alpha_1 \alpha_{0_2}$$

Passage du binaire au décimal
Dans ce sens le passage d'une écriture à l'autre est immédiat : il suffit d'effectuer les opérations du développement binaire :

$$1011_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 11_{10}$$

Passage du décimal au binaire
Le passage du décimal au binaire repose sur l'application itérative de la proposition 2.

$$
\begin{aligned}
43 &= 2*21+1 &\rightarrow \alpha_0 = 1\\
21 &= 2*10+1 &\rightarrow \alpha_1 = 1\\
10 &= 2*5+0 &\rightarrow \alpha_2 = 0\\
5 &= 2*2+1 &\rightarrow \alpha_3 = 1\\
2 &= 2*1+0 &\rightarrow \alpha_4 = 0\\
1 &= 2*0+1 &\rightarrow \alpha_5 = 1
\end{aligned}
$$

donc :

$43_{10} = 101011_2$

1.3 La base 16 : écriture hexadécimale

Le binaire, ou base 16, est une autre base utilisée fréquemment informatique. L'utilisation de l'hexadécimal permet de représenter les nombres de manière plus compacte qu'en binaire tout en assurant un passage binaire $\leftrightarrow$ hexadécimal très rapide.
De même que pour le binaire, le développement hexadécimal d'un entier naturel n est :

$$n = \sum_{i=0}^{p} \alpha_i 16^i \text{ avec } 0 \leq \alpha_i \leq 15$$

et son écriture héxadécimale est donc :

$n = \alpha_p \cdots \alpha_1 \alpha_{0_{16}}$

Pour écrire un nombre en hexadécimal il nous font donc disposer de 16 symboles. Pour les dix premiers symboles on utilise les 10 chiffres $0, 1, \cdots, 9$ et pour les six derniers on utilise les six premières lettres de l'alphabet A, B, C, D, E et F.

Symbole	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Valeur	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

TABLE I.1 – Correspondance chiffres hexadécimaux - décimal

Passage du binaire à l'hexadécimal et inversement
Le lien entre le binaire, base 2, et l'hexadécimal, base 16, se voit à partir de l'égalité $2^4 = 16$. Ceci implique que chaque digit (symbole) hexadécimal se code sur 4 bits.

Hex.	0	1	2	3	4	5	6	7
Binaire	0000	0001	0010	0011	0100	0101	0110	0111
Hex.	8	9	A	B	C	D	E	F
Binaire	1000	1001	1010	1011	1100	1101	1110	1111

TABLE I.2 – Correspondance chiffres hexadécimaux - binaire

Pour passer de l'écriture hexadécimale à l'écriture binaire il suffit donc de remplacer chaque digit hexadécimal par sa représentation binaire :

$1BF_{16} = 0001\ 1011\ 1111_2$

Dans le sens inverse il suffit de découper l'écriture binaire en groupe de 4 bits, quitte à rajouter des 0 à gauche si le nombre total de bits n'est divisible par 4, et de remplacer chaque groupe de 4 bits par sa représentation hexadécimale :

$1011011_2 = 0101\ 1011 = 5B_{16}$

Passage de l'hexadécimal au décimal

On procède de la même manière que pour le binaire, en effectuant les opérations du développement hexadécimal tout en tenant comptes des valeurs des différents symboles données dans le tableau ci-dessus :

$1BF_{16} = 1 * 16^2 + 11 * 16^1 + 15 * 16^0 = 447_{10}$

Passage du décimal à l'hexadécimal

Même si l'on peut procéder par divisions successives par 16 en pratique on ne le fait pas. En effet la division par 16 est fastidieuse alors que celle par 2 est immédiate et il est très facile de passer du binaire à l'hexadécimal comme nous l'avons vu ci-dessus. Donc pour passer du décimal à l'hexadécimal on passe par la représentation binaire que l'on écrit ensuite en hexadécimal :

$43_{10} = 101011_2 = 00101011 = 2B_{16}$

2 Codage des différents types numériques en informatique

La représentation des différents types numériques à pendant longtemps été propre à chaque constructeur. A l'heure actuelle, suite à des efforts de normalisation, quasiment tous les constructeurs utilisent les mêmes représentations.

Ce sont celles-ci que nous présenterons ici. De plus, pour des raisons historiques, les bits sont regroupés sous forme d'octet (byte en anglais) comportant 8 bits. Les types numériques ou alphanumériques seront donc codés sur un ou plusieurs octets.

2.1 Les entiers

2.1.1 Les entiers naturels

La représentation machines des entiers naturel repose naturellement sur leur écriture binaire. L'ensemble des entiers que l'on peut ainsi représenter dépend uniquement du nombre de bits utilisés pour le codage. De manière générale si l'on utilise n bits pour le codage d'un entier on peut représenter tous les entiers de 0 à $2^n - 1$. Dans la plupart des cas les entiers naturels sont codés sur :

- 16 bits (2 octets) permettant de représenter les entiers de l'intervalle $[0, \cdots, 65\ 535]$
- 32 bits (4 octets) permettant de représenter les entiers de l'intervalle $[0, \cdots, 4\ 294\ 967\ 295]$
- 64 bits (8 octets) permettant de représenter les entiers de l'intervalle $[0, \cdots, 18\ 446\ 744\ 073\ 709\ 551\ 615]$

2.1.2 Les entiers relatifs ou entiers signés

Quelle que soit la représentation adoptée il faudra inclure le signe dans le codage du nombre, ce qui nous coutera 1 bit et donc en utilisant n bits nous pourrons représenter les entiers de l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$.Il existe plusieurs méthodes de représentation des entiers signés.

Le binaire signé

L'idée la plus simple pour représenter un entier relatif est de coder le signe directement sur le bit de poids fort (le bit le plus à gauche de la représentation binaire) 0 correspondant au signe + et 1 au signe - et de consacrer les bits restants au codage binaire de la valeur absolue du nombre à représenter.
Par exemple, en utilisant 8 bits pour le codage, on aura :

$$x = 13 \quad \rightarrow 0000\ 1101$$
$$x = -13 \rightarrow 1000\ 1101$$

Le décalage

Pour un codage sur n bits, une autre idée consiste à décaler le 0 de $2^{n/2}$. Par exemple, en utilisant 8 bits pour le codage, 0 sera décalé de $2^4 = 16$ et on aura :

$x = 0 \quad \rightarrow 0001\ 0000$
$x = -1 \quad \rightarrow 0000\ 1111$
$x = 1 \quad \rightarrow 0001\ 0001$
$x = -13 \rightarrow 0000\ 0011$

Le complément à 2

Malheureusement aussi bien le binaire signé que le décalage du 0 ont un défaut majeur : ces représentations ne sont pas compatibles avec l'addition. L'utilisation de la représentation dite de complément à 2 permet d'éliminer cette difficulté. Soit x un entier relatif à représenter avec n bits :

- si $x \geq 0$, x est codé en binaire sur $n-1$ bits avec 0 comme bit de poids fort
- si $x < 0$
  - on code $\mid x \mid$ en binaire sur $n-1$ bits avec 0 comme bit de poids fort
  - on inverse les bits du résultat précédent (NON binaire)
  - on ajoute 1 au nombre binaire précédent en ignorant les dépassements

Cette opération correspond au calcul de $2^n - \mid x \mid$, d'où le nom complément à 2.

Par exemple, pour une représentation sur 4 bits on a les correspondances suivantes :

Valeur	Représentation
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
⋮	⋮
7	0111

On vérifie sans peine que cette représentation est compatible avec l'addition. Bien entendu c'est cette représentation qui est effectivement utilisée actuellement.

2.2 Les réels ou nombres flottants

L'utilisation du terme "réel" dans le cadre de la représentation des nombres en informatique relève de l'abus de langage. En fait ce terme sous entend nombre "avec des chiffres (en nombre fini) après la virgule", c'est à dire un rationnel au sens des mathématiques. Au niveau machine, un réel sera donc toujours approché par un rationnel. Bien entendu, tout comme pour les entier, on ne pourra représenter qu'un sous ensemble fini et borné des rationnels.

2.2.1 Chiffres significatifs

C'est la notion importante qui justifie l'écriture en virgule flottante. Le but de cette représentation est d'obtenir, pour un encombrement mémoire donné, un éventail de valeurs suffisant avec la plus grande précision possible.

Pour illustrer cette notion et pour plus de commodité nous nous placerons dans le système de numérotation décimal. Supposons que nous disposions de 8 caractères dont la virgule et le signe pour représenter un décimal. Dans

une représentation à virgule fixe, nous ne pourrons écrire que les décimaux du type suivant :

$\pm 123,456$

L'éventail des nombres représentables est très limité. Ainsi le plus petit décimal, en valeur absolue, représentable est :

$+0,00001$

Sur ce dernier exemple 5 caractères sont occupés par la représentation de 0 et le seul chiffre significatif est 1. Il est clair qu'une telle représentation est d'une part particulièrement inefficace et d'autre part d'une précision très limitée. L'idée de la représentation en virgule flottante consiste donc à avoir toujours un maximum de chiffres significatifs.

2.2.2 Codage en virgule flottante

Le principe de la virgule flottante est inspiré de la notation familière aux scientifiques, qui préfèrent écrire $1,97.10^8$ plutôt que 197000000.
L'idée va donc être d'exprimer les réels sous la forme :

$x = m.b^e$

Où m est la mantisse, b la base et e l'exposant.

D'une manière générale le format flottant est défini par :

- le signe s
- la base b
- le nombre de chiffres de la mantisse, p
- la plage de valeurs possibles pour l'exposant $e_{min}, \cdots, e_{max}$

Ainsi tout flottant s'écrit sous la forme :

$$x = (-1)^s.b^e.\left(a_0.b^0 + a_1.b^{-1} + \cdots + a_p.b^{-p}\right) \tag{I.1}$$

avec $s = \pm 1$, $e \in [e_{min}, \cdots, e_{max}]$ et $a_i \in [0, \cdots, b-1]$.

Si $a_0 \neq 0$ l'écriture est dite normalisée, sinon elle est dite dénormalisée. Les nombre dénormalisés comblent le vide de représentation qui existe entre 0 et le plus petit nombre normalisé, mais cela à un prix : le sacrifice des chiffres significatifs. Il résulte de cette écriture que le plus petit écart entre deux mantisses est $\epsilon = b^{-p}$.

Bien entendu une telle représentation ne permet de représenter exactement

qu'un nombre fini de réels, ceux qui peuvent s'écrire exactement sous la forme ci-dessus. Pour tous les autres se pose le problème de l'approximation, autrement dit de l'arrondi.

Considérons $x \in \mathbb{C}$. Choisir une représentation flottante de x revient à choisir une mantisse m de p chiffres telle que :

$(-1)^s.m.b^e = x\,(1 + b^{-p}\alpha)$

$b^{-p}\alpha$ est l'erreur relative sur x. α est appelé erreur d'arrondi normalisée :

- en arrondi au plus près, $\alpha \in [-0.5, 0.5[$
- en arrondi vers 0, $\alpha \in [0, 1[$
- en arrondi vers $\pm\infty$, $\alpha \in [-1, 1[$

2.2.3 La norme IEEE754-1985

La norme IEEE754-1985 impose un format particulier pour le codage des nombres en virgule flottante. Elle est le résultat d'un ensemble de compromis qui, si ils sont parfois surprenants, permettent d'obtenir un codage puissant et efficace. Elle correspond en outre à une norme internationale (IEC559).

Principes généraux

Le codage d'un nombre dans la normalisation IEEE754 est légèrement plus complexe que celui présenté ci-dessus :

- La mantisse m est codée en base 2 avec codage des valeurs négatives par bit de signe. Elle est normalisée et est donc représenté par un triplet (S, N, F)(signe, valeur absolue de la partie entière, valeur absolue de la partie fractionnaire) normalisé de façon à ce que $1 \leq |m| < 2$ (si $|m| \neq 0$). La normalisation implique que $N = 1_2$ ce qui permet de coder N de façon implicite (bit caché). C'est donc la longueur de F qui caractérise la précision relative du code.
- L'exposant e est un entier relatif codé par décalage. e est donc représenté par une valeur entière E avec $e = E - d$ (avec d décalage). On notera que, dans la norme IEEE754, les deux valeurs extrêmes de l'exposant sont réservées au codage de valeurs "non numériques" (voire section 4.4).

On a donc un codage binaire sous la forme d'un triplet (S, F, E) (avec $N = 1_2$ implicitement) ce qui donne :

$$r = S.(1, F).2^{(E-d)} \tag{I.2}$$

La notation (1,F) exprime le fait qu'il faut rajouter le bit caché (issu de la normalisation de la mantisse) devant la partie fractionnaire F de la mantisse pour obtenir la mantisse complète.

Les différents formats
La norme IEEE754 définit quatre formats différents permettant d'obtenir des précisions et des dynamiques différentes en fonction des besoins :
Simple précision codage sur 32 bits
Double précision codage sur 64 bits
Simple précision étendue codage sur 44 bits
Double précision étendue codage sur 80 bits

En pratique, seuls les types simple précision (float) et double précision (double) sont couramment utilisés en informatique (les types correspondant aux "précisions étendues" sont prévus pour permettre le stockage des résultats temporaires sans propagation d'erreurs d'arrondi). La double précision étendue est implémentée dans les microprocesseurs les plus courants (Intel et Motorola en particulier) mais tous les compilateurs ne permettent pas de l'utiliser. Ainsi, le compilateur Visual C++ de Microsoft traite les long double comme des double bien qu'il ne s'agisse pas du même type de donnée. Le Gnu C Compiler (gcc) autorise l'utilisation du codage en double précision étendue. On notera par ailleurs que, l'alignement des nombres en double précision étendue étant difficilement compatible avec l'utilisation des bus et des registres de 64 bits, un nouveau standard de fait émerge sous l'impulsion des constructeurs (Sun avec le processeur SPARC V8 et Hewlett Packard avec le processeur PA-RISC) : la quadruple précision codée sur 128 bits.

Codage binaire
Le codage binaire prévu dans la norme IEEE754 peut paraître complexe de prime abord ; il a en effet été conçu pour favoriser la rapidité des calculs flottants (et non pour faciliter la lecture "humaine"). C'est donc un codage "orienté processeur" plus qu'un codage "orienté utilisateur".
Tout nombre flottant est composé d'un bit de signe S, de la valeur absolue de la partie fractionnaire de la mantisse[1] F et de l'exposant E. Le codage IEEE normalise le nombre de bits de chacune de ces parties.

1. Du fait de la normalisation, la partie entière, toujours à 1, n'est pas stockée puisqu'elle est implicite (bit caché)

Par ailleurs, afin de permettre la comparaison directe de deux nombres flottants (dans le même format), la mantisse est codée sur les poids faibles et l'exposant sur les poids forts (le bit de poids le plus fort étant utilisé pour le signe).
Le nombre de bits utilisés pour représenter chaque champs d'un flottant est résume dans le tableau I.3 et la position des différents champs sur la figure I.1

	Taille	Signe	Mantisse	Exposant
Simple précision	32	1	23 + 1 (caché)	8
Double précision	64	1	52 + 1 (caché)	11
Double précision étendue	80	1	63 + 1 (caché)	16

TABLE I.3 – Tailles en bits des différents champs d'un flottant

FIGURE I.1 – Positions des différents champs d'un flottant

Codage du signe :
Le codage du bit de signe est extrêmement simple : par convention, le nombre est positif si le bit de signe (bit de poids fort) est à 0 et négatif sinon.

Codage de l'exposant :
L'exposant est un nombre entier relatif, codé sur 8 bits en simple précision et sur 11 bits en double précision. Il est codé par décalage : 127 pour en simple précision et 1023 en double précision. Cependant, toutes les valeurs "codables" ne sont pas utilisées car les valeurs extrêmes (-127 et 128 en simple précision) sont utilisées pour le codage de valeurs spécifiques (voir paragraphe suivant).

Codage de la mantisse :
La mantisse est un nombre fractionnaire, normalisé, codé sur 24 bits en simple précision et sur 53 bits en double précision. La normalisation impose que le premier bit de la mantisse (ici, le premier bit avant la virgule) soit toujours 1

(l'exposant est ajusté en conséquence). Cela permet de garantir la précision maximum. La partie fractionnaire de la mantisse est donc un nombre fractionnaire compris entre 0.0_2 et $0.11111111111111111111111_2$. La partie entière, toujours égale à 1_2 n'est pas codée (bit caché).
On notera que la normalisation, en entraînant la "perte" de la valeur originale de l'exposant, place délibérément la norme IEEE754 du coté des mathématiciens (qui travaillent sur des nombres libres de sémantiques) et non du coté des physiciens (pour lesquels la sémantique des nombres est souvent plus importante que les nombres eux-même). Ainsi, pour un physicien :

0,00 années-lumières + 1 cm = 0,00 années-lumières

tandis que, pour un mathématicien (et donc pour un informaticien) :

0,00 années-lumières + 1 cm = 1 cm.

On voit clairement que la normalisation, parce qu'elle perd "l'ordre de grandeur" de la valeur, risque d'entrainer une perte de signification lors des traitements.

Codes "spécifiques"

Nous avons déjà vu que les exposants "extrêmes" ($E = 0$ et $E = 255$, ou e=-127 et $e = 128$, en simple précision) sont réservés à des usages spécifiques et non au codage des flottants proprement dits. La norme IEEE754 autorise en effet l'utilisation de mantisses "dénormalisées", le codage de valeurs spécifiques "non-numériques" (NaN,$+\infty$ et $+\infty$), ainsi que le codage exact de la valeur zéro.

Valeurs dénormalisées :

Les nombres dénormalisés sont les nombres pour lesquels la partie entière de la mantisse n'est pas fixée à 1. De façon évidente la dénormalisation entraine une perte de précision relative (puisque le nombre de chiffres significatifs de la mantisse n'est plus égal au nombre de chiffres total de la mantisse [2]). Elle est cependant utile pour les nombres dont l'exposant est inférieur à e_{min} qui, sans la dénormalisation ne seraient pas codables. C'est pour cette raison que l'exposant $e_{min} - 1$ (correspondant à $e = -127$ en simple précision, c'est à dire à $E = 0$) est réservé au codage de ce type de valeurs. Dans le cas des valeurs dénormalisées, la formule I.2 n'est plus valide, et est remplacée par :

$$r = S.0, F.2^{-d+1} = S.0, F.2^{e_{min}} \tag{I.3}$$

2. Ainsi, en décimal, avec une mantisse sur trois chiffres, il est évident que la représentation $3,14.10^0$ est plus précise que $0,03.10^2$

Attention l'exposant $e_{min} - 1$ sert au codage des valeurs dénormalisées mais l'exposant pris en compte dans I.3 est bien e_{min}, ceci afin de garantir la continuité de cette représentation avec la représentation normalisée I.2.

Codage du zéro :
Pour des raisons de simplicité évidentes, la norme IEEE754 utilise, pour le codage du zéro, une représentation particulière correspondant au "zéro binaire". On notera que cette représentation correspond en pratique à un nombre dénormalisé "à l'extrême" ce qui permet, dans un calcul, d'arrondir automatiquement vers 0 lorsque cela s'avère nécessaire. On notera par ailleurs, que le codage signé utilisé pour les mantisses (bit de signe) permet de coder deux valeurs de zéro différentes (+0 et -0).

"*Not a Number*" :
La normalisation IEEE754 introduit la notion de "*NaN*" ("*Not a Number*") afin de garantir qu'un calcul donnera toujours un résultat, même si ce résultat est invalide (par exemple lors du calcul de $\sqrt{-1}$). *NaN* est représenté, en binaire, par une valeur d'exposant égale à $e_{max} + 1$ (128 en simple précision) et de mantisse non nulle [3]. Le codage de *NaN* est particulièrement intéressant car il permet de simplifier les formes algorithmiques en "négligeant" les tests systématiques des valeurs intermédiaires (à condition que toute opération arithmétique recevant *NaN* en entrée retourne *NaN* en sortie). Cette dernière caractéristique est d'ailleurs imposée dans la norme (par exemple : $NaN + 3 = NaN$).

Codage des deux infinis :
Si le résultat de $\sqrt{-1}$ est *NaN*, cela n'est pas le cas de 1/0. En effet, la norme IEEE754 permet de coder de façon explicite les deux "valeurs" $+\infty$ et $-\infty$. Elle définit de plus une arithmétique spécifique pour leur utilisation (par exemple : $\frac{1}{\infty} = 0$) pour pouvoir les utiliser dans les calculs courants. La représentation binaire de $\pm\infty$ utilise très logiquement l'exposant $e_{max} + 1$ (128 pour un nombre simple précision) et une mantisse nulle. Le signe est évidemment utilisé pour différentier $+\infty$ et $-\infty$.

Le tableau I.4 résume les différents cas rencontrés dans le codage binaire

3. Les bits de la mantisse peuvent alors être utilisés pour exprimer le type de l'opération ayant produit un *NaN*.

des nombres flottants.

Exposant	Mantisse fractionnaire	Exposant
$e = e_{min} - 1$	$F = 0$	± 0
$e = e_{min} - 1$	$F \neq 0$	$\pm 0, F.2^{e_{min}}$
$e_{min} \leq e \leq e_{max}$	$-$	$\pm 1, F.2^{e}$
$e = e_{max} + 1$	$F = 0$	$\pm \infty$
$e = e_{min} - 1$	$F \neq 0$	NaN

TABLE I.4 – Résumé des valeurs codables

Arrondis
La norme définit quatre modes d'arrondis :

- au plus proche (réel machine le plus proche du réel à coder)
- vers $+\infty$ (réel machine supérieur ou égal au réel à coder)
- vers $-\infty$ (réel machine inférieur ou égal au réel à coder)
- vers 0 (correspond à un arrondi vers $+\infty$ pour les nombres positifs et vers $-\infty$ pour les nombres négatifs)

Lors d'un calcul, il est possible de choisir le mode d'arrondi. Le mode d'arrondi par défaut est l'arrondi au plus proche.

Changer le mode d'arrondi a essentiellement deux usages : tester la robustesse d'un algorithme ou faire du calcul d'intervalle.

Plus un algorithme est robuste, moins il est sensible à des petites différences des données de départ. On effectue le calcul avec plusieurs modes d'arrondis et la comparaison des résultats obtenus permet de tester la robustesse de l'algorithme.

Le calcul d'intervalle consiste à encadrer la valeur exacte d'un calcul en l'effectuant deux fois, une fois avec arrondis vers $+\infty$ et l'autre avec arrondis vers $-\infty$.

2.2.4 Conséquences de ce choix de représentation

Etant donné qu'il existe une infinité de nombre réels, et qu'entre deux d'entre eux il y a aussi une infinité d'autres réels, il est évident que la représentation des nombres flottants (qui n'autorise que le codage d'un nombre limité de valeurs) n'est qu'une approximation (on remarquera d'ailleurs que

les nombres flottants ne sont, par définition, que des nombres fractionnels et certainement pas des nombres réels "pûrs"). Il est donc fondamental de connaître la précision de la représentation utilisée.

Précision
La précision de la représentation est portée par la mantisse mais l'interprétation de cette dernière est modifiée par l'exposant. La précision de la représentation ne peut donc pas être considérée comme absolue : suivant l'ordre de grandeur de la valeur codée, la précision sera totalement différente. C'est pourquoi on parle d'erreur relative de codage. Pour une valeur donnée, l'erreur absolue doit donc être calculée en multipliant l'erreur relative par la valeur de l'exposant.

La précision de la représentation correspond, pour une valeur donnée de e, à la plus petite différence entre deux valeurs codables, c'est-à-dire à l'intervalle au sein duquel les valeurs réelles ne sont pas exprimables. Ainsi, si la mantisse est exprimée sur n bits ($n = 23$ en simple précision, $n = 52$ en double précision et $n = 63$ en double précision étendue), l'erreur relative du codage est égale à la précision portée par le chiffre de poids le plus faible : 2^{-n}. Ceci est illustré dans le tableau I.5.

x	suivant	intervalle
0.0	$1.1754944.10^{-38}$	$1.1754944.10^{-38}$
1.0000000	1.0000001	$1.1920929.10^{-7}$
2.0000000	2.0000002	$2.3841858.10^{-7}$
$1.6000000.10^{1}$	$1.6000002.10^{1}$	$1.9073486.10^{-6}$
$1.2800000.10^{2}$	$1.2800002.10^{2}$	$1.5258789.10^{-5}$
$1.0000000.10^{20}$	$1.0000001.10^{20}$	$8.7960930.10^{12}$
$9.9999997.10^{37}$	$1.0000001.10^{38}$	$1.0141205.10^{31}$

TABLE I.5 – Intervalle entre deux réels représentés exactement (en représentation normalisée)

Il est important de bien faire la différence, dans le codage des flottants, entre l'erreur relative ϵ qui dépend du nombre de bits de la mantisse et le plus petit réel codable (de l'ordre de $1,4012.10^{-45}$ en simple précision, de $4,9406.10^{-324}$ en double précision et de $3,6451.10^{-4951}$ en double précision

étendue).

En simple précision on a donc une erreur relative de :
$\epsilon = 2^{-23} \simeq 10^{-7}$
En double précision on a :
$\epsilon = 2^{-52} \simeq 10^{-16}$
Enfin, en double précision étendue, on a :
$\epsilon = 2^{-63} \simeq 10^{-20}$

En décimal, les nombres flottants sont donc représentables avec sept chiffres significatifs (en simple précision), seize chiffres significatifs (en double précision) ou 20 chiffres significatifs (en double précision étendue). L'imprécision des représentations peut avoir des conséquences importantes sur le codage des programmes. En effet, du fait de ces approximations, une simple comparaison peut donner des résultats surprenants.

Plus petite et plus grande valeur codable
Comme nous l'avons vu précédemment nous ne pouvons coder qu'un nombre fini de valeur, il y a donc, en valeur absolue, une plus petite (différente de 0) et une plus grande valeur codable. Nous donnons ci-dessous leurs valeurs pour la simple précision.

Le plus petit nombre positif différent de zéro et le plus grand nombre négatif différent de zéro (représentés par une valeur dénormalisée avec $e = e_{min} - 1$, c'est à dire $E = 0_2$, et seul le bit de poids faible de m à 1) sont :
$\pm 2^{-23}.2^{-126} = 2^{-149} \simeq \pm 1,4012985.10^{-45}$

Le plus petit nombre normalisé positif différent de zéro et le plus grand nombre normalisé négatif différent de zéro (représentés avec $e = e_{min}$, c'est à dire $E = 1_2$, et tous les bits de m à 0) sont :
$\pm 2^{-126} = 2^{-149} \simeq \pm 1,17549351.10^{-38}$

Le plus grand nombre positif fini et le plus grand nombre négatif fini (représentés avec $e = e_{max}$, c'est à dire $E = 11111111_2$, et tous les bits de m à 1) sont :
$$\pm 2^{127} \sum_{k=0}^{23} 2^{-k} = \pm 2^{127} \left(2 - 2^{-23}\right) \simeq \pm 3,4028235.10^{38}$$

Imperfection des opérations élémentaires
Il est évident que, si le codage en virgule flottante n'est qu'une approximation des réels, alors les opérations arithmétiques sur les réels ne peuvent plus être considérées comme exactes. En effet, dans le cas général, le résultat d'un calcul devra toujours être considéré comme différent de son résultat théorique : si, en théorie, on a $a+b=c$, on devra toujours considérer en pratique que $a+b=c(1+\epsilon_{ab})$ avec ϵ_{ab} (erreur d'arrondi) dépendant de a et de b.

Addition :
Outre les problèmes de dépassement de capacité (*overflow*) qui peuvent survenir quand l'un des deux opérandes possède l'exposant e_{max}, le principal problème de l'addition est lié à la dénormalisation (mise au même exposant) utilisée lors du calcul. En effet, si les deux opérandes sont trop différents (en ordre de grandeur, c'est-à-dire en exposant), alors la plus petite valeur sera, au cours de la dénormalisation, confondue avec 0 et donc négligée dans le calcul : en résumé, si $a \gg b$ alors $a+b=a$.

Soustraction :
Le problème de la soustraction est plus important dans la mesure ou la soustraction de deux opérandes de valeurs comparables amplifie l'erreur relative : si $a \cong b$, alors l'erreur relative sur $a-b$ est égale à l'erreur absolue sur $a-b$ divisée par le résultat soit :

$$\epsilon_{relatif} = \frac{\epsilon_{absolu}}{|a-b|}$$

Cette imperfection "fondamentale" de la soustraction - qui ne s'exprime, heureusement, que lorsque les deux opérandes sont presque égaux - est à l'origine de beaucoup d'erreurs et d'approximations dans les calculs numériques.

Multiplication/division :
Vu la définition du format flottant, la multiplication et la division sont plus simples que l'addition et la soustraction. Même si elles sont soumises aux erreurs d'arrondi, d'*overflow* et d'*underflow* (sous-dépassement de capacité : un nombre est trop petit pour être représenté sera confondu avec zéro), elles n'entraînent pas de problèmes aussi cruciaux que la soustraction. On restera cependant particulièrement attentif aux dépassements de capacités (dans les deux sens). En effet, dès que l'exposant d'un des deux opérandes est supérieur à $\frac{e_{max}}{2}$ ou inférieur à $2.e_{emin}$, un dépassement de capacité devient possible lors

d'une multiplication et/ou d'une division.

2.2.5 Conséquences sur les algorithmes de calcul

L'arithmétique des opérations "machine" étant différente de l'arithmétique "réelle", il convient d'être méfiant lors de l'enchainement de plusieurs calculs. Ainsi, d'une façon générale, il n'est plus possible de considérer que $(a+b)+c = a+(b+c)$. Considérons, par exemple, les deux calculs suivants :

$$S_1 = \sum_{n=1}^{100000} \frac{1}{n^2}$$

$$S_1 = \sum_{n=100000}^{1} \frac{1}{n^2}$$

En simple précision on obtient $S_1 = 1.64472532$ et $S_2 = 1.64492404$. La différence est donc de l'ordre de 10^{-4} bien que la précision relative, en simple précision, soit de 10^{-7}. On voit donc que, non seulement le résultat du calcul ne peut pas être considéré exact mais que, de plus, les erreurs peuvent se cumuler au fil des calculs. En outre, considérant le résultat théorique $S_1 = S_2 \simeq 1.64492407$, on voit que l'agencement du calcul permet ou non d'obtenir des résultats plus ou moins proches du résultat théorique. Il peut donc être utile de déduire, de ces considérations, des règles d'écriture pour les algorithmes numériques.

Propagation des erreurs

La plupart des algorithmes numériques sont basés sur des calculs de suites et/ou de développements récurrents ou non. La conséquence principale d'une telle structure algorithmique est que les résultats intermédiaires sont réutilisés à chaque pas de temps ce qui conduit l'algorithme 'a propager les erreurs d'arrondi au fur et à mesure de l'avancée des calculs. Dans ce cas, le résultat final est approximé en fonction des différents arrondis réalisés au cours du calcul. D'es lors, la solution idéale serait que les arrondis soient totalement indépendants entre eux et centrés sur zéro (en d'autres termes, les arrondis ont autant de chances d'avoir lieu "vers le haut" que "vers le bas"). Dans ce cas, le résultat final (issu de N opérations d'erreur moyenne ϵ) sera approximé avec une erreur de l'ordre de $\epsilon\sqrt{N}$ (le terme $\sqrt{N}$ correspond à la distance moyenne parcourue après une marche aléatoire de N pas de lon-

gueur moyenne ϵ réalisés aléatoirement dans les deux sens). Ce cas idéal est cependant rare et ne doit être considéré qu'avec une très grande prudence pour au moins deux raisons :

- Il est très courant que, soit l'algorithme utilisé, soit les particularités de l'implémentation ne biaise l'orientation des erreurs d'arrondi. Dans ce cas, l'erreur totale sera de l'ordre de $N.\epsilon$.
- Dans certaines conditions, particulièrement défavorables mais malheureusement très courantes, les erreurs initiales vont s'accumuler rapidement car elles vont être démultipliées par les calculs ultérieurs. C'est en particulier le cas lorsque le programme est amené à soustraire deux nombres très proches puis à réutiliser le résultat de cette soustraction.

Malheureusement de telles situations sont assez courantes dans les algorithmes numériques. Ainsi, la "simple" résolution d'une équation du second degré par la formule classique :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

peut conduire à des résultats totalement erronés si $ac \ll b^2$.

Par exemple la calcul de e^x à partir des 2000 premiers termes de son développement en série entière [4] donne rapidement des résultats totalement erronés (tab. I.6).

x	calcul	e^x
-1	0.3678793	$0.367879\cdots$
-5	0.0067384	$0.0067379\cdots$
-10	-0.0000625	$0.0000454\cdots$
-20	-2.7566754	$0.0000000020612\cdots$
-30	-24368.556	$0.000000000000093576\cdots$
-40	-730834432	$0.0000000000000000042\cdots$

TABLE I.6 – Calcul de e^x à partir de son développement en série entière

L'explication de la divergence de cet algorithme est relativement simple ; il suffit d'observer l'évolution de $\frac{x^n}{n!}$ pour différentes valeurs de x et pour les premiers termes du développement en série entière. Les figures I.2 et I.3

4. $e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$

montrent que pour des valeurs de x "relativement" grandes, $\frac{x^n}{n!}$ atteint rapidement des valeurs importantes pour n petit. Or, pour des valeurs négatives de x, on remarque que le développement en série entière comporte alternativement des additions et des soustractions. L'algorithme passe alors par des soustractions de valeurs proches mais très grandes en regard du résultat final ce qui amplifie l'erreur relative (voir ci-dessus). Le même algorithme utilisé pour calculer des valeurs positives de x ne poserait donc pas les mêmes problèmes.

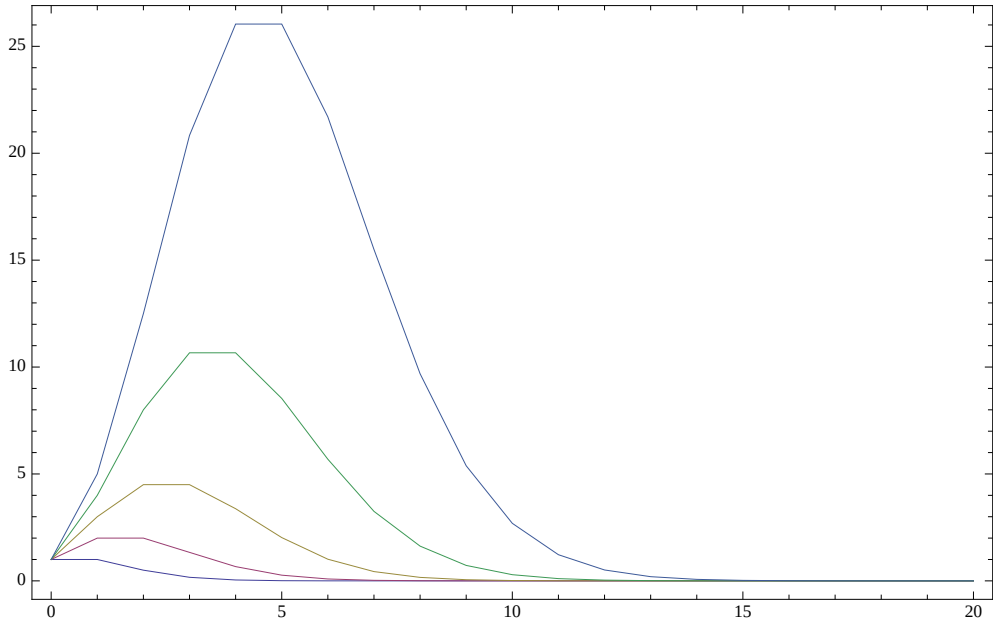

FIGURE I.2 – Valeurs successives de $\frac{x^n}{n!}$ pour $x \in \{1, \cdots, 5\}$ (de bas en haut) et n variant de 0 à 20

Conditionnement d'un algorithme numérique

Le bon conditionnement d'un algorithme permet de limiter la propagation des erreurs. Soit le calcul $y = f(x)$, on dit que le calcul est mal conditionné si une petite variation de x entraîne une grande variation de y. Le conditionnement est caractérisé par le "nombre de condition du calcul", C défini, pour chaque

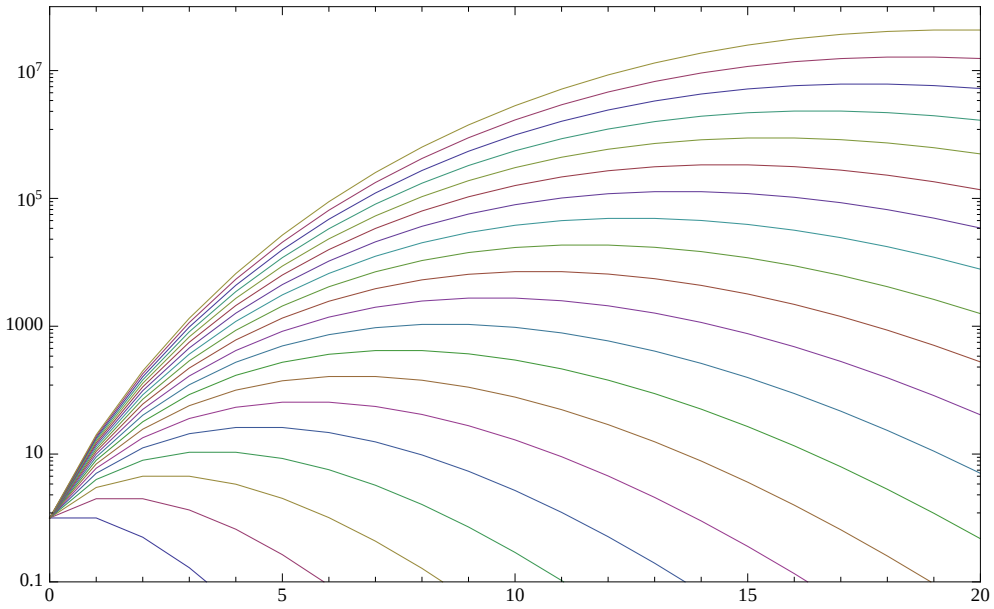

FIGURE I.3 – Valeurs successives de $\frac{x^n}{n!}$ pour $x \in \{1, \cdots, 20\}$ (de bas en haut) et n variant de 0 à 20

valeur de x, par :

$$C = \left| \frac{xf'(x)}{f(x)} \right| \tag{I.4}$$

Si C est grand (> 1), alors le calcul est "mal conditionné" et risque de produire des erreurs cumulatives.

En calculant C pour les quatre opérations (addition, soustraction, multiplication, division) on s'aperçoit qu'elles sont toutes bien conditionnées exception faite de la soustraction.

Stabilité d'un algorithme numérique
Le conditionnement d'un calcul n'est qu'une condition nécessaire pour garantir que les erreurs ne soient pas propagées exagérément (donc que la réponse soit précise). En effet, il est aussi nécessaire que l'algorithme soit numériquement stable.

Un algorithme est dit stable si la valeur calculée $f_{calc}(x)$ est la solution exacte d'un calcul $f_{theorique}(y)$ avec y proche de x (donc si $f_{calc}(x) = f_{theorique}(y + \epsilon)$ avec ϵ petit).

Exemple
Reprenons l'exemple du calcul de $f(x) = e^x$. Le nombre de condition de ce calcul est $C = \left|\frac{xe^x}{e^x}\right| = |x|$. Ce calcul est mal conditionné pour $x \notin [-1, 1]$ et comme nous l'avons vu plus n'est pas stable pour les valeurs négatives inférieures à -1. Ces considérations nous mettent sur la voie d'un algorithme stable et bien conditionné pour le calcul de l'exponentielle :

- x est décomposé en partie entière N et partie fractionnaire F
- e^N est calculé par multiplications successives (la multiplication est stable et bien conditionnée)
- e^F est calculé avec le développement en série entière qui dans ce cas ($F \in [0, 1[$) est stable et bien conditionné
- enfin $e^x = e^{(N+F)} = e^N.e^F$

3 Conclusion

Tout le codage d'information et tout calcul se fait au niveau machine en utilisant une représentation binaire.

La réalisation d'algorithmes numériques exacts n'est généralement pas possible puisqu'il est impossible de coder exactement l'ensemble des réels. Il est donc fondamental de conserver à l'esprit que l'arithmétique machine n'est pas équivalente à l'arithmétique mathématique. Par conséquent, tout algorithme numérique (sur les flottants et, dans une moindre mesure sur les entiers) produit un résultat approximatif (au mieux) voire totalement faux. Pour éviter de se retrouver dans ce dernier cas (et pour limiter l'importance du premier), quelques règles courantes peuvent être déduites des considérations présentées dans ce chapitre :

- **Il faut choisir avec soin le type des données**, d'une part pour éviter les débordements (dans le cas des entiers), d'autre part pour garantir une précision suffisante (dans le cas des flottants). **D'une façon générale, le format "simple précision" doit être considéré comme peu fiable et évité** sauf en cas de raison majeure.
- **Il faut faire attention aux conversions de type**, en particulier aux

réductions (conversion d'un type vers un type plus court) mais aussi aux élargissements plus ou moins implicites qui peuvent provoquer des résultats surprenants, par exemple lors des tests (d'autant plus que les règles de conversion changent suivant les langages).

- **Il faut faire attention lors des soustraction**, en particulier si les deux valeurs soustraites peuvent être proches.
- **Attention aux tests d'égalité entre flottants** qui peuvent donner des résultats surprenants.
- Ne pas faire une confiance aveugle aux compilateurs mais aussi aux librairies, aux codes récupérés, ... Suivant les cas, il pourra être judicieux de savoir exactement ce que fait le code ...

Reste que toutes ces précautions ne serviront de toute façon à rien si l'algorithme utilisé entraîne mathématiquement des approximations ... ou s'il est faux ...