Arithmétique des ordinateurs

INSA-Lyon, département Informatique, troisième année
G. Beslon, LaTeX, février 2012, v1.6

préambule

23 août 1991, Gandsfjorden, Norvège : une plate-forme de forage de plus de cent mille tonnes coule suite à la rupture de la paroi des ballasts, provoquant un séisme de magnitude 3 sur l'échelle de Richter ; coût estimé : 700 millions de dollars. Les experts remontent les causes de la catastrophe jusqu'au logiciel de calcul des éléments finis utilisé pour le dessin des ballasts. Une approximation de calcul dans ce logiciel a conduit à une sous-estimation de la taille des parois du balast de 47%. Compte-tenu de cette sous-estimation, les experts estiment la rupture probable à partir de 62m de profondeur. La catastrophe s'est produite par 65m de fond.

25 février 1991, Dharan, Arabie Saoudite : un missile Scud irakien détruit un baraquement américain ; 28 morts, 100 blessés. En raison d'une erreur d'arrondi sur le calcul du temps (au 24ième bit[1]), la batterie de missiles "Patriot" supposée protéger le base déviait de 0,3 secondes (la vitesse d'un Scud étant de 1676m/s, l'erreur commise sur la position était donc de plus de 500m). Aucun missile anti-missile n'a été lancé. Le bug avait été détecté le 11 février. Le correctif est arrivé à Dharan le 26.

4 juin 1996, Kourou, Guyane Française : après 39 seconde de vol, la fusée Ariane 5 s'autodétruit avec son chargement (coût estimé : 500 millions de dollars). L'autodestruction a été provoquée suite à une correction de l'angle de vol après qu'une vitesse horizontale importante ait été signalée par le Système de Référence Inertielle (SRI). En pratique, la vitesse était normale mais une conversion de "double" (floating-Point, 64 bits) en "short" (integer, 16 bits) avait provoqué un dépassement de capacité. Le SRI de secours utilisant le même code, il provoque une erreur similaire[2].

2001, passage à l'euro : le taux de conversion est très précis (un euro = 1,95583 DM). Supposons une banque allemande dans laquelle un informaticien développe deux fonctions pour la conversion Deutchmark-Euro ...

```
#include <stdio.h>
#include <math.h>
#define TAUX 1.95583 // taux de conversion deutchmark-euro

float dm2euro(float dm) {
  float temp;
  // conversion deutchmark-euro
  temp=dm*TAUX;
  temp*=100.0;
```

[1] L'erreur commise est de 0,000000095 secondes mais elle s'accumule sur l'ensemble du temps d'utilisation de la batterie. Au moment de l'attaque, le système était activé depuis 100 heures.

[2] Le SRI avait été développé pour Ariane 4 dont la vitesse – inférieure à celle d'Ariane 5 – ne provoquait pas de dépassement de capacité. Il est important de remarquer que ce système n'était plus utile sur Ariane 5.

```
  temp=roundf(temp);
  return temp/100.0;
}

float euro2dm(float euro) {
  float temp;
  // conversion euro-deutchmark
  temp=euro/TAUX;
  temp*=100.0;
  temp=roundf(temp);
  return temp/100.0;
}

/* Si, partant d'une fortune initiale de 0.02 DM, je convertis
les Deuchmarks en Euros centime par centime, puis mes Euros
en Deutchmarks centime par centime, etc. */

main() {
  float euro=0;
  float dm=0.02;
  for(;;) {
    while (dm>0.01) { // je vends mes deutchmarks centime par centime
      dm=dm-0.01;
      euro=euro+dm2euro(0.01);
      }
    while (euro>0.01) { // je vends mes euros centime par centime
      euro=euro-0.01;
      dm=dm+euro2dm(0.01);
      }
    printf("ma fortune est : %f dm\n",dm);
    }
}
```

et je suis rapidement milliardaire ...

1 Introduction

Comment manipule-t-on les nombres en informatique (sous-entendu "sur une machine *réelle*") ... et d'ailleurs, qu'est-ce qu'un nombre en informatique ?

Trois aspects à distinguer :

- Le codage des données (transcription binaire, exacte ou approchée d'une valeur particulière)
- La réalisation "matérielle" des opérations élémentaires (généralement : addition, soustraction, multiplication, division auxquelles s'ajoutent les compléments, les comparaisons, ...)
- La réalisation logicielle des opérations de haut niveau. Ce troisième niveau peut lui-même être redécomposé en deux sous-niveaux :

→ Opérateurs et fonctions standards, généralement pris en charge par les langages de haut niveau (C, C++, java, python, ...)

→ Algorithmes numériques explicitement programmés ...

Sans entrer dans les détails (en deux heures !) le but de cette séance de travaux dirigés est de comprendre/approcher :

- Les qualités et les limites des machines généralement utilisées (qualités et limites des représentations utilisées, réalisations éventuellement imparfaites des opérations élémentaires, ...)
- Les qualités espérées (espérables ?) des algorithmes numériques (et les règles de bonne conduite permettant d'obtenir des algorithmes de qualité !)

En gros, le but est de comprendre pourquoi on peut faire n'importe quoi sans le savoir ... et comment on peut éviter de faire n'importe quoi en sachant finalement assez peu de choses[3] ...

2 Petit rappel sur la numérotation de position

Le principe de la numérotation de position est si simple ... qu'on oublie qu'il n'a pas toujours été employé (la numérotation romaine, par exemple, ne l'utilise pas). Dans ce système, en effet, le poids d'un chiffre (et donc sa contribution réelle à la valeur d'un nombre) dépend de sa position dans l'écriture du nombre. Cette numérotation est rendue possible grace à l'introduction d'un chiffre désignant l'absence de contribution (le zéro[4]). Elle permet une écriture extrêmement simple des opérations élémentaires (essayez d'additionner deux nombres écrits en numérotation romaine !).

Le principe de la numérotation de position repose sur la notion de *base* : la contribution d'un des chiffres du nombre à la valeur de ce dernier est calculée par : $C.B^d$ avec C le chiffre considéré, B la base et d le décalage du chiffre par rapport à l'origine (Par convention, l'origine est placée à la virgule[5] et on compte les décalages en positif vers la gauche et en négatif vers la droite). Dans la vie de tous les jours, on utilise généralement la base 10 (numérotation décimale) mais en informatique on utilise classiquement la base 2 (binaire) ou la base 16 (hexadécimal). La valeur du nombre est donc donnée par la formule :

$$\sum_{i=d_{min}}^{d_{max}} C_i.B^i$$

Ainsi, "cent un virgule un" s'écrit 101,1 en base 10 ($1.10^2 + 0.10^1 + 1.10^0 + 1.10^{-1}$; on notera $101,1_{10}$) mais 101,1 en base 2 ($101,1_2$) ne vaut "que" $1.2^2 + 0.2^1 + 1.2^0 + 1.2^{-1}$ soit $5,5_{10}$ ... On aura remarqué au passage que la numérotation de position ne se limite pas au codage des entiers ...

[3] Contrairement à ce qui est couramment admis, ce type de question se pose rapidement. Outre les exemples proposés en préambule, on peut citer toutes applications de simulation numérique, par exemple en calcul scientifique, mais aussi les jeux vidéo, les images de synthèse (par exemple pour les algorithmes de lancé de rayon) ou la CAO. Des problèmes peuvent même apparaître dans des codes apparemment anodins (voir préambule !).

[4] Le zéro a été "inventé" par les indiens puis adopté par les arabes qui l'introduisirent progressivement en Europe par l'Espagne ... Les romains ne pouvaient donc pas utiliser la numérotation de position car il leur manquait cet élément essentiel [BES03].

[5] Dans toutes les équations de ce document, on notera, par convention "." pour exprimer la multiplication et "," pour exprimer la virgule.

Question : que vaut $101,1_{16}$?

3 Petit rappel sur le codage des entiers

3.1 Codage des entiers naturels

Le codage des entiers naturels en binaire permet, avec une taille de n chiffres binaires (n bits), de représenter tout entier inférieur à 2^n (donc tout entier compris entre 0 et $2^n - 1$). Les tailles les plus courantes en informatique étant 8 bits (un octet ou byte), 16, 32 et 64 bits, il est utile de mémoriser, au moins de façon approximative, les valeurs suivantes :

2^8	256
2^{16}	65.536
2^{32}	4.294.967.296 (environ quatre milliards)
2^{64}	18.446.744.073.709.551.616

3.2 Codage des entiers relatifs

Il y a quatre méthodes couramment utilisées pour représenter les nombres entiers signés sur n bits :

Bit de signe Le bit de poids fort indique le signe ("bit de signe") et les $n - 1$ bits de poids faibles donnent la valeur absolue. En dehors de sa simplicité de lecture, cette technique présente peu d'intérêts et beaucoup d'inconvénients (complexité de l'implémentation des calculs arithmétiques, double codage du zéro, ...).

Complément à un L'opposé du nombre est obtenu par un complément bit à bit (par exemple, si $5_{10} = 0000.0101_2$, on a $-5_{10} = 1111.1010_2$. Dans ce cas, le bit de poids fort indique toujours le signe de l'entier relatif mais, contrairement à la technique du "bit de signe", il est *aussi* utilisé pour le calcul de la valeur. Par rapport au "bit de signe", le complément à un simplifie la gestion des dépassements de capacité (en particulier lors de l'addition binaire). Le principe du complément à un est que les nombres sont ordonnés "sur un cercle" : les incrémentations successives à partir de 0 passent par la valeur (positive) maximale (sur huit bits : $0111.1111_2 = 127_{10}$) puis à la valeur (négative) minimale ($1000.0000_2 = -127_{10}$) pour remonter ensuite vers 0 (1111.1111_2). Le principal inconvénient est qu'il existe deux codages différents pour le zéro ($+0_{10} = 0000.0000_2$ et $-0_{10} = 1111.1111_2$ avec "$+0 = -0 + 1$" ce qui est mathématiquement difficile à accepter !).

Complément à deux L'opposé du nombre est obtenu par un complément à un *suivi* d'une incrémentation (donc $-5_{10} = 1111.1010_2 + 1_2 = 1111.1011_2$). Les nombres sont toujours ordonnés sur un cercle (voir ci-dessus) mais l'incrémentation permet de ne coder qu'une valeur de zéro (0000.0000_2). En revanche, le codage en complément à deux permet de coder une valeur négative "de plus" que les valeurs positives. Il existe donc un nombre qui ne possède pas d'opposé dans ce mode de codage (ainsi, sur 8 bits, $-127_{10} = 1000.0000_2$ a pour opposé la valeur $0111.1111_2 + 1_2 = 1000.0000_2 = -127_{10}$!).

Codage avec décalage Les trois modes de codage présentés ci-dessus conservent la représentation "classique" (représentation de position) pour les entiers positifs. Ce n'est pas le cas pour le codage avec décalage. Dans ce dernier cas, un nombre est représenté par sa valeur augmentée d'une constante : le *décalage* (pour lequel on choisit généralement la valeur 2^{n-1}). Ainsi, sur 8 bits, on a (pour un décalage de $2^{8-1} = 128$) : $-5_{10} \rightarrow -5_{10} + 128_{10} = 123_{10} = 0111.1011_2$).

Les ordinateurs "modernes" utilisent tous le codage en complément à deux pour le codage des entiers relatifs. La principale raison, outre les avantages cités ci-dessus, est que l'addition des entiers relatifs (avec un codage en complément à deux) s'exprime exactement comme l'addition des entiers naturels. Ce codage permet donc de simplifier le jeu d'instruction des ordinateurs.

3.3 Opérations sur les entiers

Les opérations classiquement implémentées sont l'addition, la soustraction, la multiplication et la division. Elles sont généralement implémentées au niveau matériel. Si les opérations d'addition et de soustraction sont relativement simples, cela n'est pas le cas pour la multiplication et la division (en particulier pour les entiers relatifs). Ces opérations sont généralement implémentées en utilisant des décalages à gauche ou à droite (notés << et >>) en utilisant une propriété fondamentale du codage de position : un décalage à gauche (respectivement à droite) multiplie (respectivement divise) la valeur d'un nombre par la valeur de la base. En base deux on a, par exemple (pour 123_{10}) : $<< 0111.1011_2 = 1111.0110_2 = 246_{10} = 123_{10}.2_{10}$

3.4 Limites

La représentation des nombres entiers est exacte. Il en est de même des opérations arithmétiques sur les valeurs entières, sous réserve que :

- Le résultat de l'opération ne dépasse pas les limites du codage utilisé (limite de taille et interprétation éventuelle du signe),
- La division est considérée comme une division entière avec perte du reste entier.

Lorsque le résultat d'une opération dépasse les limites du codage utilisé, il se produit alors un *dépassement de capacité* ou *overflow*.

3.5 Un codage particulier : les puissances négatives de deux

On a vu section 2 que la numérotation de position ne se limitait pas au codage des entiers ... Il existe donc une méthode relativement simple pour coder les nombres fractionnels : les puissances négatives de deux. Il suffit pour cela d'utiliser - mais en base 2 - les mêmes conventions qu'en base 10, à savoir les décalages négatifs de l'exposant à droite de la virgule. Ainsi $0,101101_2 = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} = 0,703125_{10}$. Cette notation n'est cependant pas (ou peu) utilisée en informatique. En effet, elle imposerait une convention de codage supplémentaire peu utile car il s'agit d'une simple changement d'échelle par rapport au codage des entiers ($0,101101_2 = 101101_2/2^6$ car les deux nombres sont équivalents à un décalage à gauche de 6 chiffres près). L'utilisation des puissances négatives de deux est donc laissée aux bons soins du programmeur ...

4 La représentation en virgule flottante

Le rapide élimine le lent, même si le rapide a tort.
W. Kahan

4.1 Motivations et historique

La représentation des valeurs entières (naturelles ou relatives) est très limitée dans sa *dynamique* (c'est-à-dire dans la plage de valeurs utilisable). La raison en est simple : quelle que soit la valeur représentée, sa précision est toujours la même ("un"). Il n'est pas possible d'étendre la plage de valeurs en limitant la précision comme le font, par exemple, les physiciens lorsqu'ils estiment la masse du soleil ou celle de l'électron. Pour de telles représentations, deux solutions s'offrent à l'informaticien : il peut utiliser une représentation *intrinsèquement* dynamique et à précision limitée ou utiliser une représentation à précision fixe (typiquement, les entiers) et gérer algorithmiquement la dynamique par des changements d'échelle au niveau du programme.

Historiquement, les premiers ordinateurs étaient trop lents pour qu'on puisse se permettre de mettre en œuvre des représentations standardisées pour le stockage des valeurs à précision limitée. En effet, l'implémentation des opérations de base est alors beaucoup plus complexe (donc plus lente) que pour les entiers. Il était alors préférable d'augmenter le temps programmeur pour limiter le temps machine[6]. Cependant, avec l'accroissement de la puissance des machines, il est devenu acceptable d'ajouter des composants matériels permettant une gestion "générique" des nombres à précision limitée. Ce type de représentations est apparu graduellement au cours de l'évolution de l'informatique, d'abord limité aux ordinateurs dédiés au calcul scientifique (années 60 et 70), puis, à partir des années 80, sur l'ensemble des machines. Ce déploiement a été accompagné par un processus de normalisation IEEE (Institute of Electrical and Electronics Engineers) qui a débuté en 1977 sous l'impulsion de W. Kahan[7] pour s'achever en 1985.

La représentation en précision limitée est généralement restée dans l'ombre de l'informatique, jusqu'à ce qu'en 1994 un bogue soit détecté dans les tous nouveaux circuits Pentium d'Intel. Ce bogue ne touchait que la division (et les chances d'avoir une division fausse étaient de une sur neuf milliards) mais il a coûté plus de 300 millions de dollars à Intel ...

[6]En 1946, Dans "*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*", Arthur Burks, Herman H. Goldstine et John von Neumann exprimaient ce choix ainsi : *Il apparaît qu'il y a deux raisons majeures pour un système décimal virgule "flottante", qui découlent du fait que le nombre de chiffres dans un mot est une constante fixée par des considérations de réalisation pour chaque machine particulière. La première raison est de conserver autant de chiffres significatifs que possible dans une somme ou un produit, et la seconde est de libérer l'opérateur humain de la corvée d'estimer et d'introduire dans un problème des "facteurs d'échelle" des constantes multiplicatives qui servent à conserver les nombres dans les limites de la machine.*
Nous ne nions pas, naturellement, le fait que du temps humain est perdu à préparer l'introduction des facteurs d'échelle adéquats. Nous disons seulement que le temps ainsi dépensé n'est qu'un tout petit pourcentage du temps total passé à préparer un problème intéressant pour notre machine. Le premier avantage de la virgule flottante est, nous pensons, quelque peu illusoire. Pour avoir une telle représentation virgule flottante, on doit perdre de la capacité mémoire qui pourrait autrement être utilisée pour manipuler davantage de chiffres par mot. Il ne nous semble donc pas clair du tout que les modestes avantages d'une virgule flottante binaire compensent la perte de capacité mémoire et la complexité accrue de l'arithmétique et des circuits de contrôle. (d'après [HEN03], annexe H)

[7]W. Kahan a reçu le prix Turing en 1989 en reconnaissance de ses travaux sur la "virgule flottante". En effet, la norme IEEE754 est en grande partie basée sur ses travaux antérieurs sur l'IBM 7094.

Pourtant, sans aller jusqu'à parler de bogue matériel, une bonne maîtrise de la représentation en précision limitée est nécessaire pour programmer convenablement des applications de calcul numérique, sous peine de réaliser des programmes qui, si ils paraissent justes sur le plan purement *mathématique*, peuvent donner des résultats totalement faux en pratique ...

4.2 Principe de la représentation des "réels" : le codage en virgule flottante

Le codage en virgule flottante est un *codage rationnel approché, à grande dynamique, de précision relative limitée*. Il ne doit donc en aucun cas être confondu avec un *codage exact de réels à dynamique illimitée*[FRE02]. C'est précisément de cette différence que sont issues les contraintes propres à l'utilisation des "flottants" en algorithmie.

La représentation en virgule flottante est une représentation de type $r = (m; e)_B$ avec :

B La *base* utilisée pour la représentation. En notation scientifique classique, la base utilisée est la base 10 (on note alors $0,453.10^{23}$). En informatique, on utilise (classiquement) la base 2 mais d'autres bases ont pu être utilisées sur d'autres architectures (en particulier la base 16 chez IBM) ou dans d'autres cultures/civilisations (par exemple la base 60 chez les Babyloniens ou la base 20 chez les Gaulois). Les bases 10 et 2 sont cependant les seules utilisées en informatique, la première parce qu'elle est la plus favorables aux échanges homme-machine et la seconde parce qu'elle permet d'obtenir la plus grande précision dans les calculs à plus faible coût.

m La *mantisse* (ou significande) est un nombre à p chiffres exprimant les chiffres significatifs du nombre. La mantisse est dite *normalisée* si et seulement si on a toujours soit $m = 0$ soit $Max/B \leq \|m\| < Max$. La normalisation permet de garantir l'unicité de la représentation du nombre et la "stabilité" de la précision. La taille de la mantisse (p) permet alors de connaître la *précision relative* du codage : $\epsilon_r = B^{-p}$.

e L'*exposant* est un entier (compris entre e_{min} et e_{max}) codant l'ordre de grandeur du nombre. Il caractérise la *dynamique* du codage : $B^{e_{max}-e_{min}}$.

Connaissant B, m et e on obtient r avec :

$$r = m.B^e \tag{1}$$

On notera que les modes de codage adoptés pour les nombres m et e ne sont pas imposés à ce stade. Un codage en virgule flottante peut donc prendre des formes très différentes suivant les codages "internes" utilisés (choix des modes de représentation, choix des longueurs de codage, ...). Il y a donc un réel besoin de normalisation afin d'uniformiser les codages et de permettre les échanges de codes et de données entre machines hétérogènes[8].

[8]Ces besoins de standardisation sont devenus particulièrement cruciaux avec l'apparition des réseaux informatiques.

4.3 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[9]).

4.3.1 Principes généraux

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

- La mantisse est codée en base 2 avec codage des valeurs négatives par bit de signe. Elle est normalisée (avec $Max = 2_{10}$). m 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 le coder de façon implicite (bit caché). C'est donc la longueur de F qui caractérise la précision relative du code.
- L'exposant est un entier relatif codé par décalage. e est donc représenté par une valeur entière E avec $e = E - d$ (avec d le décalage du code). 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) (plus la valeur implicite $N = 1_2$) ce qui donne :

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

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.

4.3.2 Les différents formats

La norme IEEE754 définie 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, Visual Studio traite les *long double* comme des *double* bien qu'il ne s'agisse pas du même type de donnée[10]. Le Gnu C Compiler (gcc) autorise l'utilisation du codage en double précision étendue.

[9] http ://www.iso.ch

[10] Previous 16-bit versions of Microsoft C/C++ and Microsoft Visual C++ supported the long double, 80-bit precision data type. In Win32 programming, however, the long double data type maps to the

4.3.3 Le 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 étant composé d'un bit de signe (S), de la valeur absolue de la partie fractionnaire de la mantisse[11] (F) et de l'exposant E, le codage IEEE normalise le nombre de bits de chacune de ces parties. 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).

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

Codage du signe :

Le codage du bit de signe est extrêmement simple (logiquement !) : 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 section suivante).

double, 64-bit precision data type. The Microsoft run-time library provides long double versions of the math functions only for backward compatibility. The long double function prototypes are identical to the prototypes for their double counterparts, except that the long double data type replaces the double data type. The long double versions of these functions should not be used in new code. Source : http ://msdn.microsoft.com/fr-fr/library/9cx8xs15.aspx

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

Question : quelle est la valeur, en décimal, des exposants suivants et à quelles puissances de 10 correspondent-ils ?

$e_1 = 10000000_2$
$e_2 = 00000001_2$
$e_3 = 10000010_2$
$e_4 = 01111111_2$
$e_5 = 01101001_2$
$e_6 = 11111111_2$
$e_7 = 00000000_2$

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 après 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 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 traitement.

Question : quelle est la valeur, en décimal, des mantisses suivantes ? (en simple précision)

$m_1 = (1)00000000000000000000000_2$
$m_2 = (1)10000000000000000000000_2$
$m_3 = (1)10101101011111110010100_2$

La normalisation de la mantisse est un avantage indéniable, en termes de précision *relative* du codage (elle permet aussi de simplifier l'implémentation matérielle des opérations arithmétiques), cependant elle en reduit la précision *absolue*. En effet, elle interdit le codage de nombres inférieurs à $0.1_2.2^{e_{min}}$ (avec $e_{min} = -126$ en simple précision et -1022 en double précision). Dans ce cas, la norme IEEE754 autorise la *dénormalisation* (voir section suivante).

Quelques exemples :

Question : quelle est la valeur, en décimal, des nombres suivants ? (en simple précision)

$f_1 = 01000000000000000000000000000000_2$
$f_2 = 01000001010000000000000000000000_2$
$f_3 = 10111111100000000000000000000000_2$
$f_4 = 00110100110101101011111110010100_2$

4.3.4 La norme IEEE754-2008

L'évolution des architectures machine a conduit à l'élargissement de la norme IEEE754 à de nouveaux types de données. Après plusieurs années de discussions, la norme initiale de 1985 a été actualisée sous la forme de la norme IEEE754-2008. Dans ces grandes lignes, cette norme conserve les principes de la norme de 1985, en particulier les deux types classiques (simple précision, double précision ; la double-précision étendue n'est pas incluse dans la norme mais proposée pour augmenter la précision des calculs lorsque cela s'avère nécessaire). Le principal apport de cette révision est en fait l'ajout d'un type "quadruple précision" codé sur 128 bits et qui permet donc d'implémenter des calculs avec une précision beaucoup plus importante que les nombres en double précision. Le format de la quadruple précision est similaire à celui des simple et double précision mais les nombres de bits de la mantisse et de l'exposant sont les suivants :

	Longueur	Signe	Mantisse	Exposant
quadruple précision	128 bits	1 bit	112 bits + 1 (bit caché)	15 bits

En allongeant considérablement la taille de la mantisse, le codage en quadruple précision permet d'atteindre une précision de 34 chiffres significatifs environ (en décimal). Les exposants, codés sur 15 bits, permettent d'augmenter la dynamique du codage. En quadruple précision l'exposant décimal maximal est d'environ ± 4930.

Pour le moment la quadruple précision n'est quasiment jamais implémentée en matériel et on doit donc faire appel à des librairies pour utiliser ce type. La plupart des compilateurs C n'implémentent pas la quadruple précision, les "long double" étant traités comme des double précision étendus (c'est le cas de gcc[12]), voire comme de simples "double" (Microsoft Visual C++). La quadruple précision est disponible en Fortran (compilateurs GNU et Intel).

4.4 Codes "spécifiques" :

Nous avons déjà vu que les exposants "extrêmes" ($E = 0$ et $E = 255$ 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.

[12]Les compilateurs GNU ont cependant été dotés d'une librairie "libquadmath" en février 2011 : http ://gcc.gnu.org/onlinedocs/libquadmath/.

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[13]). 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 la valeur binaire 00000000_2 en simple précision) est réservé au codage de ce type de valeurs. Dans le cas des valeurs dénormalisées, la formule de calcul données section 4.3.1 n'est plus valide ; elle est remplacée par : $r = S.0, F.2^{E+1-d}$

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 $e_{max} + 1$ (128 en simple précision) dont la mantisse est non nulle[14].

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 défini de plus une arithmétique spécifique pour leur utilisation (par exemple : $1/\infty = 0$) 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 $+\infty$ de $-\infty$.

Le tableau suivant résume les différents cas rencontrés dans le codage binaire des nombres flottants.

[13] Ainsi, en décimal, avec une mantisse sur trois chiffres, on conçoit bien que la valeur $3,14.10^0$ est plus précise que $0,03.10^2$!

[14] Les bits de la mantisse peuvent alors être utilisés pour exprimer le type de l'opération ayant produit NaN.

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

4.5 Codage des opérations élémentaires

Contrairement au cas des entiers naturels ou relatifs pour lesquels l'addition est beaucoup plus simple (et plus rapide !) que la multiplication, dans le cas des nombres flottants, c'est cette dernière qui est la plus simple.

Ainsi, la multiplication (et, conséquemment la division) de deux flottants $r_1 = S_1.m_1.2^{e_1}$ et $r_2 = S_2.m_2.2^{e_2}$ peut être obtenue très simplement en utilisant via deux multiplications et une addition entières : $r_1.r_2 = (S_1.S_2).(m_1.m_2).2^{e_1+e_2}$. On remarquera cependant que l'addition des deux exposants ne peut pas être réalisée avec un circuit additionneur "classique" puisque le signe est codé par décalage.

L'addition et la soustraction, en revanche, ne peuvent être réalisées que sur des nombres exprimés avec le même exposant. Il est donc nécessaire de réaliser une dénormalisation préalable de la plus petite valeur en décalant sa mantisse vers la droite (en décrémentant l'exposant de un à chaque décalage) jusqu'à ce que les deux exposants soient égaux. On peut alors additionner (ou soustraire) les deux mantisses.

4.6 Imperfection du codage

Étant 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 peut donc être utile de connaître la précision de la représentation utilisée.

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 $n = 113$ en quadruple précision), l'erreur *relative* du codage est égale à la précision portée par le chiffre de poids le plus faible : 2^{-n}. 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 $3,3621.10^{-4932}$ en quadruple précision[15]).

En simple précision on a donc une erreur relative[16] de :
$\epsilon = 2^{-23} \cong 10^{-7}$
En double précision on a :
$\epsilon = 2^{-52} \cong 10^{-16}$
Enfin, en quadruple précision, on a :
$\epsilon = 2^{-113} \cong 10^{-34}$

En décimal, les nombres flottants sont donc représentables avec sept chiffres significatifs (en simple précision), seize chiffres significatifs (en double précision) ou 34 chiffres significatifs (en quadruple précision). Il est donc totalement inutile (voire aberrant !) d'écrire, en "C" : `float pi = 3.1415926535 ;` puisque, dans ce cas, la représentation binaire est : $01000000010010010000111111011011_2$.

L'imprécision des réprésentations 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. Considérons, par exemple le code C suivant :

[15] Les valeurs maximales codables sont : $3,4028.10^{38}$ en simple précision, $1,7976.10^{308}$ en double précision et $1,1897.10^{4932}$ en quadruple précision.

[16] Il s'agit bien évidemment d'une erreur relative *maximale*.

```
#include <stdio.h>
int main()
{
  float pi_1 = 3.1415926535;
  float pi_2 = 3.14159265;
  double pi_3 =  3.1415926535;
  if (pi_1 == pi_2)
    printf("pi_1 == pi_2\n");
  else
    printf("pi_1 != pi_2\n");
  if (pi_1 == pi_3)
    printf("pi_1 == pi_3\n");
  else
    printf("pi_1 != pi_3\n");
  return 0;
}
```

l'exécution de ce code affiche : `pi_1 == pi_2` (ce qui ne serait plus le cas si on supprimait le dernier chiffre de `pi_2`) et `pi_1 != pi_3`!!!

Question : au fait, que vaut $01000000010010010000111111011011_2$ en base 10 ?

4.7 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.

Ainsi, le calcul suivant :

```
#include <stdio.h>
int main()
{
  float pi_1 = 3.14159265;
  float pi_2 = pi_1/100.0;
  if (pi_1==pi_2*100.0)
    printf("pi==pi\n");
  else
    printf("pi!=pi\n");
}
```

affiche tranquillement `pi !=pi`!!!

Ce type de comportements est dû à l'imperfection du codage mais aussi aux erreurs d'arrondi. 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. Ainsi, dans le code précédent, si on utilise `128` comme diviseur et multiplicateur, alors le calcul donne effectivement, et contre toute attente, `pi==pi` !

4.7.1 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 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 $a >> b$ alors $a + b = a$.

4.7.2 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 de $a - b$ est égale à l'erreur *absolue* de $a - b$ divisée par le résultat soit :

$$\epsilon_{relatif} = \frac{\epsilon_{absolu}}{\|a - b\|} \tag{3}$$

Cette imperfection "fondamentale" de la soustraction - qui ne s'exprime, heureusement, que lorsque les deux opérandes sont presqu'égaux - est à l'origine de beaucoup d'erreurs et d'approximations dans les calculs numériques (voir section 4.8.2).

4.7.3 Multiplication/division

Nous avons vu que la multiplication et la division sont, pour les flottants, plus simples que l'addition et la soustraction. De même, 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 $e_{max}/2$ ou inférieur à $2.e_{min}$, un dépassement de capacité devient possible lors d'une multiplication et/ou d'une division.

4.8 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 codes suivants pour le calcul de

$$S = \sum_{n=1}^{100000} \frac{1}{n^2} \tag{4}$$

Code 1 :

```
#include <stdio.h>
int main()
{
  float n;
  float s;
  for (n=1;n<=100000;n++)
    s+=(1.0/(n*n));
  printf("s=%f\n",s);
}
```

Code 2 :

```
#include <stdio.h>
int main()
{
  float n;
  float s;
  for (n=100000;n>=1;n--)
    s+=(1.0/(n*n));
  printf("s=%f\n",s);
}
```

Le premier code donne comme résultat : `1.64472532` tandis que le deuxième donne : `1.64492404`. La différence est donc de l'ordre de 10^{-4} bien que la précision relative du codage `float` en C 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 pour S ($S \cong 1.64492407$), 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.

4.8.1 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 à *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. Dès 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 ${\epsilon_m}$[17]) sera approximé avec une erreur de l'ordre de $\sqrt{N}.\epsilon_m$ (le terme $\sqrt{N}$ correspond à la distance moyenne parcourue après une marche aléatoire de N pas de longueur moyenne ϵ_m réalisés aléatoirement

[17] L'indice m est là pour rappeler que l'erreur moyenne dépend de la longueur de la mantisse

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 $N.\epsilon_m$.
- 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 :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5}$$

peut conduire à des résultats peu fiables lorsque $ac << b^2$.

Par exemple, le programme suivant, destiné au calcul de e^x par le calcul de la série entière de l'exponentielle[18] donne rapidement des résultats totalement erronés.

```
#include <stdio.h>
int main()
{
  float x=-1.0;
  float t=1.0;
  float e=1.0;
  int n;
  for (n=1;n<=2000;n++)
    {
      t=t*x/(float)n;
      e+=t;
    }
  printf("e(%f)=%8.12f\n",x,e);
  return(0);
}
```

Le tableau et la figure ci-après donnent les résultats calculés par ce programme et les résultats théoriques pour e^x. Ils montrent comment l'algorithme diverge lorsque x décroit :

x	Résultat	e^x
-1	0.3678793	0.367879...
-5	0.0067384	0.0067379...
-10	-0.0000625	0.0000454...
-20	-2.7566754	0.0000000020612...
-30	-24368.556	0.000000000000093576...
-40	-730834432	0.0000000000000000042...

[18] $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + ... + \frac{x^i}{i!} + ...$

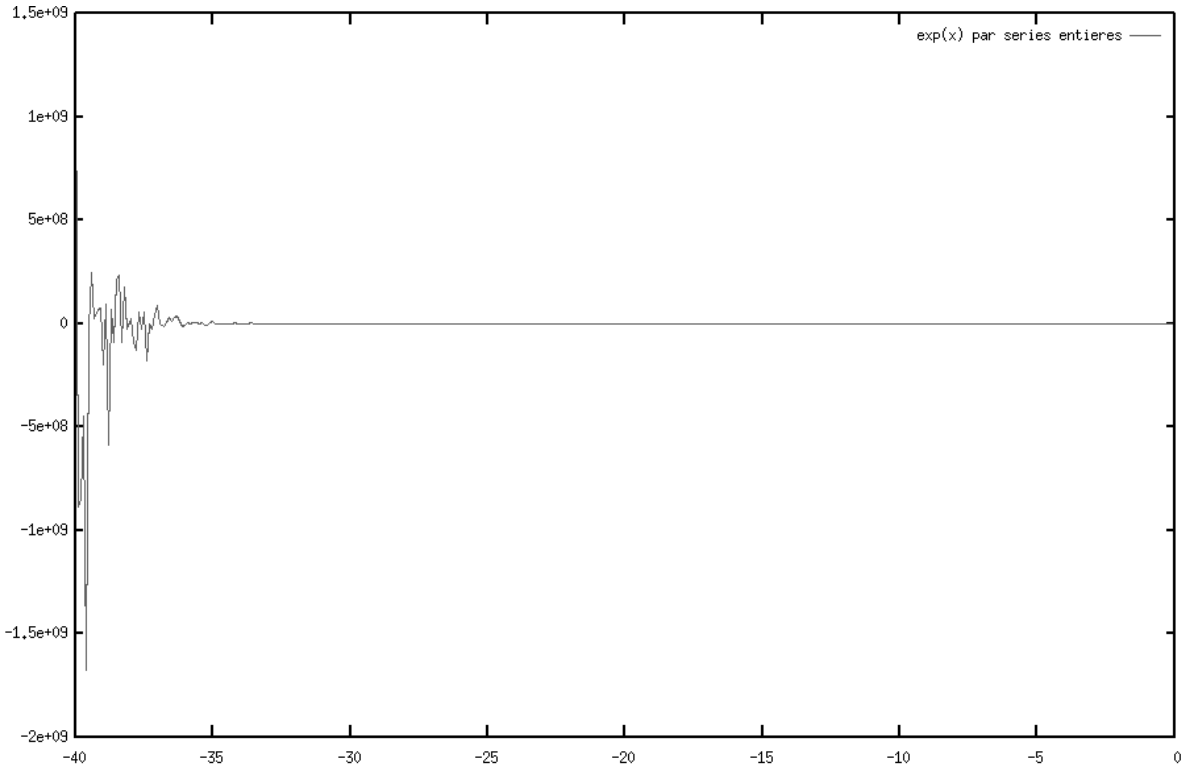

FIG. 1 – Calcul de l'exponentielle par séries entières.

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 des valeurs de n variant entre 1 et 20 (c'est-à-dire pour les premiers termes du développement en séries entières). Les figures ci-dessous montrent que pour des valeurs de x "relativement" grandes (c'est-à-dire supérieures à 5), $\frac{x^n}{n!}$ atteint rapidement des valeurs énormes. Or, pour des valeurs de x négatives, on remarque que le développement en séries entières 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 pour des valeurs positives de x ne poserait donc pas les mêmes problèmes.

4.8.2 Conditionnement d'un algorithme numérique

Le bon conditionnement d'un algorithme permet de limiter la propagation des erreurs. Soit un 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 C, le "nombre de conditions du calcul" [FAL03]. Pour une valeur x donnée, on a :

$$C = \left\| \frac{xf'(x)}{f(x)} \right\| \tag{6}$$

Si C est grand, alors le calcul est "mal conditionné" et risque de produire des erreurs accumulatives.

On peut calculer C pour les opérations courantes, ce qui donne :

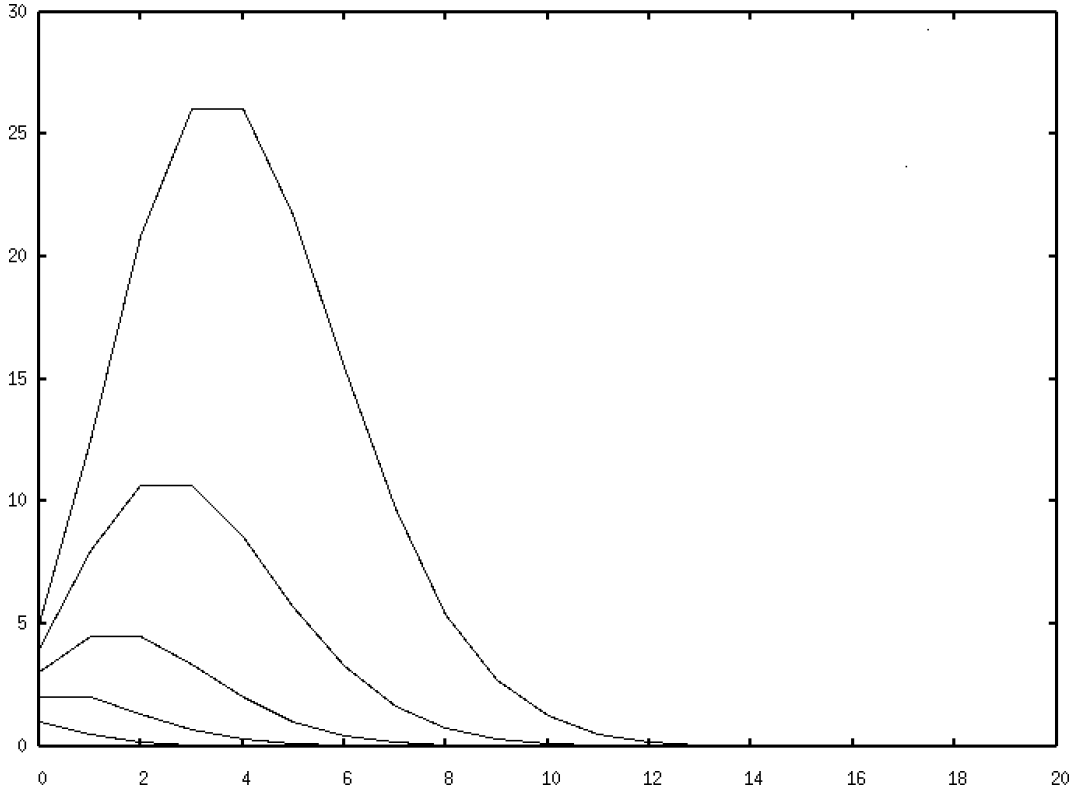

FIG. 2 – $\frac{x^n}{n!}$ pour x variant de 1 à 5 (de bas en haut)

addition $f(x) = a + x$ (avec a et x positifs), soit $f'(x) = 1$ et $C = x/(a + x)$. On a donc toujours $C < 1$; le problème est bien conditionné.

soustraction $f(x) = a - x$ (avec a et x positifs), soit $f'(x) = -1$ et $C = -x/(a - x)$. C peut donc être très grand si $a \cong x$. Dans ce cas le problème est mal conditionné. *Il faut donc toujours veiller à ne pas soustraire deux nombres proches l'un de l'autre.*

multiplication $f(x) = ax$ (avec a et x positifs), soit $f'(x) = a$ et $C = xa/ax = 1$. Le problème est bien conditionné.

inversion $f(x) = x^{-1}$, soit $f'(x) = (-x)^{-2}$ et $C = x.(-x)^{-2}/x^{-1} = -1$. Le problème est bien conditionné.

4.8.3 Stabilité d'un algorithme numérique

Le conditionnement d'un calcul n'est qu'une condition nécessaire pour garantir que les erreurs ne seront pas propagées exagérément (donc que la réponse sera 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 $f_{calc}(x) = f_{theorique}(x + \epsilon)$ avec ϵ petit). Ainsi, alors que e^x est un calcul bien conditionné ($C = x.e^x/e^x = x$), l'algorithme de calcul utilisé ci-dessus n'est pas stable, en tout cas pas pour des valeurs trop grandes.

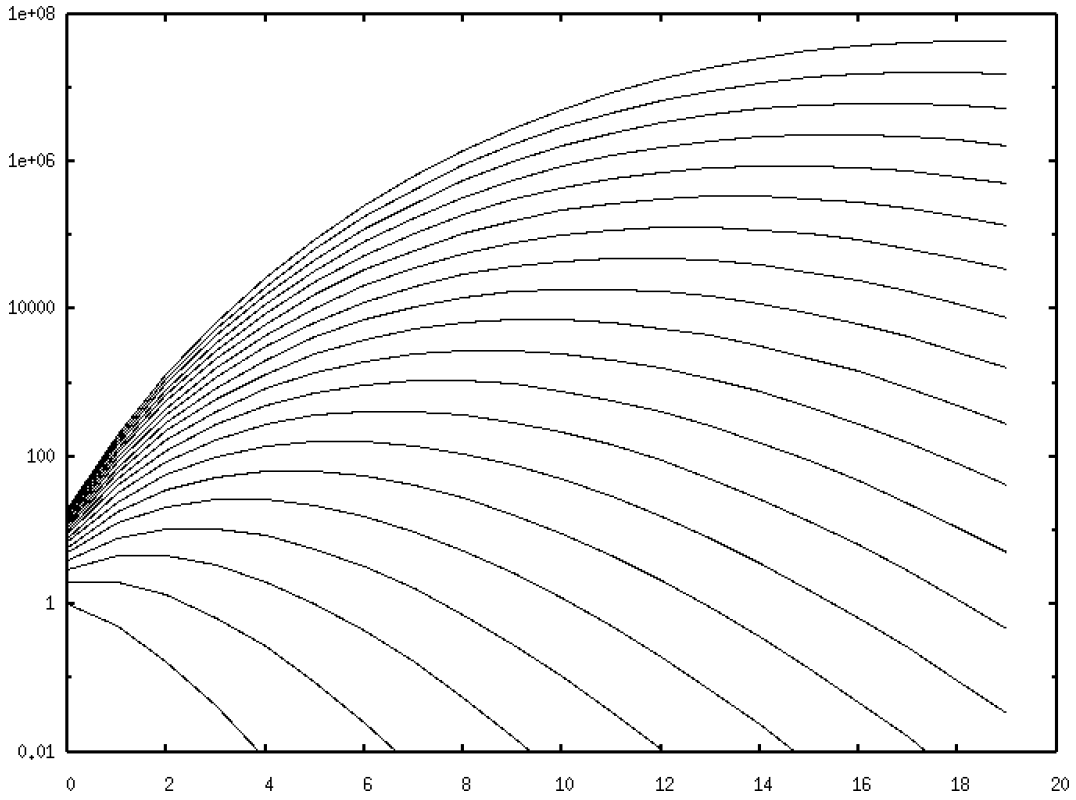

FIG. 3 – $\frac{x^n}{n!}$ pour x variant de 1 à 20 (de bas en haut)

4.8.4 Exemple d'adaptation d'un algorithme numérique

L'instabilité numérique de l'algorithme utilisé pour le calcul de e^x (sauf pour les valeurs comprises entre -1 et $+1$) permet de proposer un algorithme stable pour le calcul de l'exponentielle :

- Séparer x en partie entière N et partie fractionnaire F
- Calculer e^N par multiplications successives (la multiplication est stable et bien conditionnée)
- Calculer e^F par l'algorithme précédent (stable dans ce cas et bien conditionné)
- Multiplier les deux résultats : $e^N.e^F = e^{N+F} = e^x$

5 Conclusion et règles d'hygiène

Nous avons vu que la réalisation d'algorithmes numériques *exacts* n'est généralement pas possible puisqu'il n'est pas possible de coder *exactement* l'ensemble des réels. Il est donc fondamental de conserver à l'esprit que l'arithmétique *machine* n'est aucunement équivalente à l'arithmétique mathématique. Conséquemment, tout algorithme numérique (sur les flottants et, dans une moindre mesure sur les entiers) est succeptible de produire 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 principes théoriques énoncés ci-dessus :

- Choisir le type des données en connaissance de cause, 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 rejeté sauf en cas de raison majeure.**

- **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 surprenant, par exemple lors des tests (d'autant plus que les règles de conversion changent suivant les langages).
- Conserver systématiquement une profonde **méfiance à l'égard de la soustraction**, en particulier si les deux valeurs peuvent être proches.
- Autant que faire se peut **éviter l'utilisation de variables temporaires inutiles** (c'est souvent lors de la sortie du processeur mathématique qu'est fait l'arrondi).
- **Attention aux tests d'égalité entre flottants** qui peuvent donner des résultats suprenants (en particulier dans les instructions de branchement).
- **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 toutes façons à rien si l'algorithme utilisé entraîne *mathématiquement* des approximations ... ou s'il est *algorithmiquement* faux ...

Références

[BES03] R. Beslon, D. Lignon, *Les maths cent problèmes*, Le polygraphe, 2003

[CAZ03] A. Cazes, J. Delacroix, *Architecture des machines et des systèmes informatiques*, Dunod, 2003

[FAL03] G. Falquet, *Algorithmes et structures de données*, Cours de l'Université de Genève, *available on Internet : http ://cui.unige.ch/isi/cours/std/*

[FRE02] L. Frécon, *TD Arithmétique réelle*, Cours de l'INSA de Lyon, 2002

[HEN03] J.L. Hennessy, D.A. Patterson, *Architecture des Ordinateurs (troisième édition)*, Vuibert, 2003

[PRE97] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C, V2.08*, Cambridge University Press, 1997

[ZAN98] P. Zanella, Y. Ligier, *Architecture et technologie des ordinateurs (troisième édition)*, Dunod, 1998

Et pour finir, un petit tour de chasse aux trolls ...

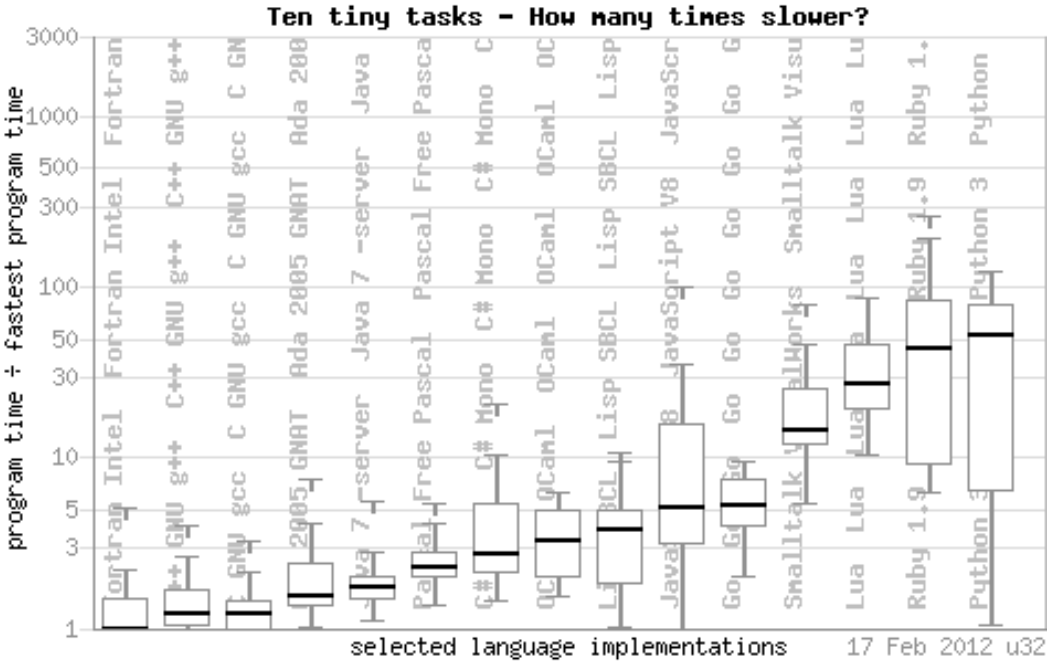

FIG. 4 – The Computer Language Benchmarks Game, http ://shootout.alioth.debian.org/