Arithmétique virgule flottante

Jean-Michel Muller
CNRS - Laboratoire LIP
(CNRS-INRIA-ENS Lyon-Université de Lyon)
Ecole PRCN – Mars 2013

http://perso.ens-lyon.fr/jean-michel.muller/

Arithmétique virgule flottante

- trop souvent perçue comme un tas de recettes de cuisine ;
- beaucoup de "théorèmes" qui sont vrais. . . souvent ;
- de simples–mais corrects !–modèles tels que le modèle standard

  en l'absence d'overflow/underflow, la valeur calculée de $(a \top b)$ *vaut* $(a \top b) \cdot (1 + \delta), \quad |\delta| \leq 2^{-p}$,

  (en base 2, mantisses de p bits et arrondi au plus près) sont très utiles, mais ne permettent pas de capter des comportements subtils, comme dans

  ```
  s = a + b ; z = s - a ; r = b - z
  ```

  et beaucoup d'autres.
- au passage, est-ce que ces "comportements subtils" sont robustes ?

Propriétés souhaitables d'une arithmétique machine

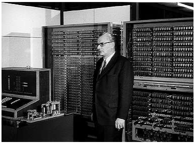
- Vitesse : la météo de demain doit être calculée en moins de 24 heures ;
- Précision, dynamique : par exemple, certaines prédictions de la mécanique quantique et de la relativité générale vérifiées avec erreur relative $\approx 10^{-14}$;
- "taille" : surface de circuit, taille du code, consommation mémoire ;
- Energie consommée : autonomie, chauffe des circuits ;
- Portabilité : les programmes mis au point sur un système doivent pouvoir tourner sur un autre sans requérir des modifications longues et/ou complexes ;
- Simplicité d'implantation et d'utilisation : si une arithmétique est trop ésotérique, personne ne l'utilisera.

Les précurseurs

- système babylonien de base 60 ; règle à calcul : Gunter (1681-1626), Oughtred (1575-1660) ;

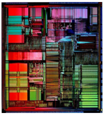
- Leonardo Torres y Quevedo (1914) : implantation électromécanique de la machine de Babbage, avec virgule flottante ;
- Konrad Zuse : Z3 (1941) : base 2, mantisses de 14 bits exposants de 7 bits. Mémoire de 16 nombres. Voir `http://www.epemag.com/zuse/`





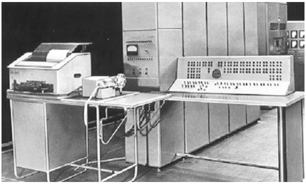
On peut faire du très mauvais travail...

- 1994 : bug de la division du Pentium, 8391667/12582905 donnait 0.666869 ⋯ au lieu de 0.666910 ⋯ ;



- Excel, versions 3.0 à 7.0, entrez 1.40737488355328, vous obtiendrez 0.64
- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;
- Excel'2007 (premières versions), calculez $65535 - 2^{-37}$, vous obtiendrez 100000 ;

Un peu d'exotisme



- Machine Setun, université de Moscou, 1958. 50 exemplaires ;
- base 3 et chiffres -1, 0 et 1. Nombres sur 18 « trits » ;
- idée : base β, nombre de chiffres n, + grand nombre représenté M. Mesure du « coût » : $\beta \times n$.
- minimiser $\beta \times n$ sachant que $\beta^n \approx M$. Si variables réelles, optimum $\beta = e = 2.718\ldots \approx 3$.

Système virgule flottante

$$\text{Paramètres : } \begin{cases} \text{base} & \beta \geq 2 \\ \text{précision} & p \geq 1 \\ \text{exposants extrèmes} & e_{\min}, e_{\max} \end{cases}$$

Un nombre VF fini x est représenté par 2 entiers :

- mantisse entière : M, $|M| \leq \beta^p - 1$;
- exposant e, $e_{\min} \leq e \leq e_{\max}$.

tels que

$$x = M \times \beta^{e+1-p}$$

On appelle mantisse réelle, ou mantisse de x le nombre

$$m = M \times \beta^{1-p},$$

ce qui donne $x = m \times \beta^e$, avec $|m| < \beta$.

Représentation normalisée

Buts : représentation unique et algorithmique plus simple.
La représentation normalisée de x, est celle qui minimise l'exposant. Si $e > e_{\min}$, donne nécessairement $1 \leq |m| < \beta$ et $\beta^{p-1} \leq |M| \leq \beta^b - 1$.

- Nombre *normal* : de valeur absolue $\geq \beta^{e_{\min}}$.
  En base 2 le premier chiffre de mantisse d'un nombre normal est un "1" $\rightarrow$ pas besoin de le mémoriser.
- Nombre sous-normal : de la forme

  $$M \times \beta^{e_{\min}+1-p}.$$

  avec $|M| \leq \beta^{p-1} - 1$. Le premier chiffre de mantisse d'un nombre sous-normal est un zéro.
  Correspond à $\pm 0.xxxxxxxx \times \beta^{e_{\min}}$.

Les sous-normaux compliquent les algorithmes, mais. . .

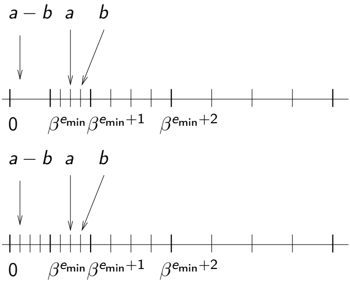


Figure: *En haut : nombres VF normaux. Dans cet ensemble, $a - b$ n'est pas représentable → le calcul $a - b$ donnera 0 en arrondi au plus près. En bas : on ajoute les sous-normaux.*

Système	β	p	$e_{\min}$	$e_{\max}$	+ grand représ.
DEC VAX	2	24	-128	126	$1.7\cdots\times 10^{38}$
(D format)	2	56	-128	126	$1.7\cdots\times 10^{38}$
HP 28, 48G	10	12	-500	498	$9.9\cdots\times 10^{498}$
IBM 370	16	6	-65	62	$7.2\cdots\times 10^{75}$
et 3090	16	14	-65	62	$7.2\cdots\times 10^{75}$
IEEE-754 binary32	2	23+1	-126	127	$3.4\cdots\times 10^{38}$
IEEE-754 binary64	2	52+1	-1022	1023	$1.8\cdots\times 10^{308}$
IEEE-754 binary128	2	112+1	-16382	16383	$1.2\cdots\times 10^{4932}$
IEEE-754 decimal32	10	7	-95	96	$9.99\cdots9\times 10^{96}$
IEEE-754 decimal64	10	16	-383	384	$9.99\cdots9\times 10^{384}$
IEEE-754 decimal128	10	34	-6143	6144	$9.99\cdots9\times 10^{6144}$

binary32 = simple précision binary64 = double précision.

Norme IEEE 754 (1985 et 2008)

- la norme IEEE 754-1985 a mis fin à une pagaille (très mauvaise portabilité des algorithmes numériques) ;
- leader : W. Kahan (père de l'arithmétique des HP35 et Intel 8087) ;
- formats ;
- spécification des opérations, des conversions, etc. ;
- gestion des *exceptions* (max+1, 1/0, $\sqrt{-2}$, 0/0, etc.) ;
- nouvelle révision adoptée en 2008.

Juste quelques mots sur les exceptions dans la norme

- l'utilisateur peut (difficilement, et c'est un euphémisme) définir le comportement du programme dans les cas exceptionnels ;
- philosophie par défaut : le calcul doit toujours continuer ;
- le format comporte deux infinis, et deux zéros. Règles intuitives : $1/(+0) = +\infty$, $5 + (-\infty) = -\infty \ldots$;
- tout de même un truc bizarre : $\sqrt{-0} = -0$;
- Not a Number (NaN) : résultat de $\sqrt{-5}$, $(\pm 0)/(\pm 0)$, $(\pm\infty)/(\pm\infty)$, $(\pm 0) \times (\pm\infty)$, NaN $+3$, etc.

Juste quelques mots sur les exceptions dans la norme

- en général, se comporte “bien”

$$1+\frac{3}{x^2}$$

  avec x très grand (de sorte que x^2 “dépasse”) $\rightarrow 1$.
- S’en méfier tout de même. Comportement pour x grand de

$$\frac{x^2}{\sqrt{x^3+1}}$$

  - on devrait obtenir un résultat proche de $\sqrt{x}$;
  - si x^3 “dépasse” mais pas x^2, on obtiendra 0 ;
  - si les 2 dépassent, on obtiendra NaN.

Quelques propriétés des NaN

- se propagent : $x+$ NaN $=$ NaN. Nécessaire (que faire d’autre ?) mais dangereux (embarqué) ;
- comparaisons : toute comparaison impliquant un NaN retourne faux (sauf une, voir ci-dessous)
  $\rightarrow x \geq y$ n’est pas exactement le contraire de $x < y$.
- si x est un NaN, le test “$x = x$” retourne “faux” et le test “$x \neq x$” retourne “vrai”
  $\rightarrow$ fournit un moyen simple de tester qu’une variable est un NaN ;

Mauvaise gestion des exceptions...

- Novembre 1998, navire américain USS Yorktown, on a par erreur tapé un « zéro » sur un clavier → division par 0. Ce problème n'était pas prévu → cascade d'erreurs → arrêt du système de propulsion.

- premier envol... et premier plongeon d'Ariane 5

Codages internes des formats binaires de la norme IEEE-754

- binary32/simple précision : 32 bits (1 de signe, 8 d'exposant, $23 + 1$ de mantisse) ;
- binary64/double précision : 64 bits (1 de signe, 11 d'exposant, $52 + 1$ de mantisse) ;
- binary128/quad précision : 128 bits (1 de signe, 15 d'exposant, $112 + 1$ de mantisse) ;
- le premier bit de mantisse des normaux est un "1", celui des sous-normaux est un "0" → on ne le mémorise pas ;
- exposant biaisé : on représente l'exposant e par l'entier positif $e + b$ (binary32 : $b = 127$, binary64 : $b = 1023$, binary128 : $b = 16383$) ;
- écriture dans l'ordre (signe, exposant biaisé, mantisse) → comparaison lexicographique, comme pour des entiers, et passage au successeur par incrémentation entière.

Exposants réservés

- la plage d'exposants "normaux" utilisables est de
  - -126 à $+127$ (biaisé : 1 à 254) en binary32,
  - -1022 à $+1023$ (biaisé : 1 à 2046) en binary64,
  - -16382 à $+16383$ (biaisé : 1 à 32766) en binary128 ;
- l'exposant tient sur 8 bits (binary32), 11 bits (binary64), ou 15 bits (binary128) ;

$\rightarrow$ on peut représenter les entiers de 0 à 255 (binary32), 0 à 1023 (binary64), ou 0 à 32767 (binary128). Les valeurs extrêmes sont réservées :
  - l'exposant biaisé 0 code 0 et les dénormalisés.
  - l'exposant maximum (255 ou 1023 ou 32767 $\rightarrow$ rien que des "1") code les valeurs spéciales $\pm\infty$ et `NaN`.

Exposants réservés

- l'exposant biaisé 0 code 0 et les dénormalisés.
- l'exposant maximum (255 ou 1023 ou 32767 $\rightarrow$ rien que des "1") code les valeurs spéciales $\pm\infty$ et `NaN`.

-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
NaN	0	11111111	chaîne non nulle
5	0	10000001	01000000000000000000000

Arrondi correct

En général, la somme, le produit, etc. de deux nombres VF n'est pas un nombre VF $\rightarrow$ nécessité de l'arrondir.

Définition 1 (Arrondi correct)

Fonction d'arrondi $x \mapsto \circ(x)$ parmi :

- $\mathrm{RN}(x)$: au plus près (défaut) s'il y en a deux :
  - *round ties to even :* celui dont la mantisse entière est paire ;
  - *round ties to away :* (2008 – recomm. en base 10 seulement) celui de plus grande valeur absolue.
- $\mathrm{RU}(x)$: vers $+\infty$.
- $\mathrm{RD}(x)$: vers $-\infty$.
- $\mathrm{RZ}(x)$: vers zéro.

Une opération dont les entrées sont des nombres VF doit retourner ce qu'on obtiendrait en arrondissant le résultat exact.

Arrondi correct

IEEE-754-1985 : Arrondi correct pour $+$, $-$, $\times$, $\div$, $\sqrt{}$ et certaines conversions. Avantages :

- si le résultat d'une opération est exactement représentable, on l'obtient ;
- si on n'utilise que $+$, $-$, $\times$, $\div$ et $\sqrt{}$, et si l'ordre des opérations ne change pas, l'arithmétique est déterministe : on peut élaborer des algorithmes et des preuves qui utilisent ces spécifications ;
- précision et portabilité améliorées ;
- en jouant avec les arrondis vers $+\infty$ et $-\infty$ $\rightarrow$ bornes certaines sur le résultat d'un calcul.

L'arithmétique VF devient une *structure mathématique en elle même,* que l'on peut étudier.

IEEE-754-2008 : l'arrondi correct est recommandé (mais pas exigé) pour les principales fonctions mathématiques (cos, exp, log, . . .)

Arrondi correct

- l'arrondi correct garantit que l'addition et la multiplication virgule flottante sont commutatives. Pour toute fonction d'arrondi $\circ$:

$$\circ(a+b) = \circ(b+a)$$

- par contre l'associativité n'est pas conservée. Exemple : $a = \beta^{p+1}$, $b = -a$, et $c = 1$, donne

$$\text{RN}\,(a + \text{RN}\,(b+c)) = 0$$

et

$$\text{RN}\,(\text{RN}\,(a+b)+c) = 1.$$

la perte d'associativité est plus "douce" pour la multiplication, sauf en cas de dépassements.

- la distributivité n'est pas préservée non plus.

Erreur relative, unité d'arrondi et "modèle standard"

- Pour simplifier : base 2,
- précision p,
- unité d'arrondi $\mathbf{u} = 2^{-p}$ si arrondi au + près, 2^{-p+1} avec les autres fonctions d'arrondi,
- si x et y sont des nombres VF, et si $\mathrm{op} \in \{+,-,\times,/\}$, alors en l'absence d'overflow et d'underflow,

$$\begin{aligned} \text{RN}\,(x \,\mathrm{op}\, y) &= (x \,\mathrm{op}\, y)(1+\epsilon_1) && \text{(1a)} \\ &= (x \,\mathrm{op}\, y)/(1+\epsilon_2), \qquad |\epsilon_1|, |\epsilon_2| \le \mathbf{u}. && \text{(1b)} \end{aligned}$$

Erreur relative, unité d'arrondi et "modèle standard"

Le modèle standard est très utile pour analyser les algorithmes numériques.
Exemple 1 :

- binary64 (base 2, $p = 53$), et fonction d'arrondi RN ;
- on veut calculer $x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n$ par l'algorithme :

$P \leftarrow x_1$
for $i = 2$ to n **do**
  $P \leftarrow \mathrm{RN}(P \times x_i)$
end for

la valeur finale de P vérifie $P = x_1 x_2 x_3 \ldots x_n \times K$, où

$$\left(1 - 2^{-p}\right)^{n-1} \leq K \leq \left(1 + 2^{-p}\right)^{n-1},$$

l'erreur relative est donc majorée par $\left(1 + 2^{-p}\right)^{n-1} - 1$, Pour $n = 500$, donnera 1.108×10^{-13}.

Erreur relative, unité d'arrondi et "modèle standard"

Exemple 2 (sommation "naïve") : $a_1 + a_2 + \cdots + a_n$ approché par

$$\sigma = \mathrm{RN}\left(\cdots \mathrm{RN}\left(\mathrm{RN}\left(\mathrm{RN}\left(a_1 + a_2\right) + a_3\right) + a_4\right) + \cdots + a_n\right).$$

Base 2 et précision p. On rappelle que $\mathbf{u} = 2^{-p}$. En définissant

$$\gamma_n = \frac{n\mathbf{u}}{1 - n\mathbf{u}},$$

on trouve

$$\left|\sigma - \sum_{i=1}^{n} a_i\right| \leq \gamma_{n-1} \sum_{i=1}^{n} |a_i|. \quad (2)$$

Résultats similaires pour produit scalaire, évaluation de polynômes par le schéma de Horner, etc.

Mais le modèle standard ne fait pas tout

- on oublie que le résultat d'un opération est complètement déterminé,
- on n'utilise pas le fait que certaines opérations sont en fait exactes,
- on perd certaines propriétés telles que la préservation de la commutativité de $+$ et $\times$;

$\longrightarrow$ incapacité à capter des comportements subtils, comme dans

```
s = a + b ; z = s - a ; r = b - z
```

et beaucoup d'autres.

Premier exemple : lemme de Sterbenz

Lemme 1 (Sterbenz)

Base β, avec nombres sous-normaux disponibles. Soient a et b deux nombres VF positifs. Si

$$\frac{a}{2} \leq b \leq 2a$$

alors $a - b$ est un nombre VF.

$\rightarrow$ *Il est calculé exactement dans n'importe lequel des 4 modes d'arrondi.*

Preuve : élémentaire en utilisant la notation $x = M \times \beta^{e+1-p}$.

Erreur de l'addition VF (Møller, Knuth, Dekker)

Premier résultat : représentabilité. RN $(x) = x$ arrondi au plus près.

Lemme 2

Soient a et b deux nombres VF. Soient

$$s = RN(a+b)$$

et

$$r = (a+b) - s.$$

s'il n'y a pas de dépassement de capacité en calculant s, alors r est un nombre VF.

Erreur de l'addition VF (Møller, Knuth, Dekker)

Démonstration : Supposons $|a| \geq |b|$,

1. s est "le" nombre VF le plus proche de $a+b \rightarrow$ il est plus près de $a+b$ que a ne l'est. Donc $|(a+b)-s| \leq |(a+b)-a|$, par conséquent

$$|r| \leq |b|.$$

2. posons $a = M_a \times \beta^{e_a-p+1}$ et $b = M_b \times \beta^{e_b-p+1}$, avec $|M_a|, |M_b| \leq \beta^p - 1$, et $e_a \geq e_b$.
   $a+b$ est multiple de $\beta^{e_b-p+1} \Rightarrow s$ et r sont multiples de β^{e_b-p+1} également $\Rightarrow \exists R \in \mathbb{Z}$ t.q.

$$r = R \times \beta^{e_b-p+1}$$

   mais $|r| \leq |b| \Rightarrow |R| \leq |M_b| \leq \beta^p - 1 \Rightarrow r$ est un nombre VF.

Obtenir r : l'algorithme fast2sum (Dekker)

Théorème 1 (Fast2Sum (Dekker))

$\beta \leq 3$. Soient a et b des nombres VF dont les exposants vérifient $e_a \geq e_b$ (si $|a| \geq |b|$,OK). Algorithme suivant : s et r t.q.

- *$s + r = a + b$ exactement ;*
- *s est "le" nombre VF le plus proche de $a + b$.*

Algorithme 1 (FastTwoSum)

$$s \leftarrow RN(a+b)$$
$$z \leftarrow RN(s-a)$$
$$r \leftarrow RN(b-z)$$

Programme C 1

```
s = a+b;
z = s-a;
r = b-z;
```

Se méfier des compilateurs "optimisants".

Preuve dans un cas simplifié : $\beta = 2$ et $|a| \geq |b|$

$$\begin{aligned} s &= \mathrm{RN}(a+b) \\ z &= \mathrm{RN}(s-a) \\ t &= \mathrm{RN}(b-z) \end{aligned}$$

- si a et b sont de même signe, alors $|a| \leq |a+b| \leq |2a|$ donc (base 2 $\rightarrow 2a$ représentable, arrondi croissant) $|a| \leq |s| \leq |2a| \rightarrow$ (Lemme Sterbenz) $z = s - a$. Comme $r = (a+b) - s$ est représentable et $b - z = r$, on trouve $\mathrm{RN}(b-z) = r$.
- si a et b sont de signes opposés, alors
  1. soit $|b| \geq \frac{1}{2}|a|$, auquel cas (lemme Sterbenz) $a + b$ est exact, donc $s = a + b$, $z = b$ et $t = 0$;
  2. soit $|b| < \frac{1}{2}|a|$, auquel cas $|a+b| > \frac{1}{2}|a|$, donc $s \geq \frac{1}{2}|a|$ (base 2 $\rightarrow \frac{1}{2}a$ est représentable, et l'arrondi est croissant), donc (lemme Sterbenz) $z = \mathrm{RN}(s-a) = s - a = b - r$. Comme $r = (a+b) - s$ est représentable et $b - z = r$, on trouve $\mathrm{RN}(b-z) = r$.

Algorithme TwoSum (Møller-Knuth)

- pas besoin de comparer a et b ;
- 6 opérations au lieu de 3 $\longrightarrow$ moins cher qu'une mauvaise prédiction de branchement en comparant a et b.

Algorithme 2 (TwoSum)

$$
\begin{aligned}
s &\leftarrow RN(a+b)\\
a' &\leftarrow RN(s-b)\\
b' &\leftarrow RN(s-a')\\
\delta_a &\leftarrow RN(a-a')\\
\delta_b &\leftarrow RN(b-b')\\
r &\leftarrow RN(\delta_a+\delta_b)
\end{aligned}
$$

Knuth : $\forall\beta$, en absence d'underflow et d'overflow $a+b=s+r$, et s est le nombre VF le plus proche de $a+b$.

Boldo et al : (preuve formelle) en base 2, marche même si underflow.

Preuves formelles (en Coq) d'algorithmes similaires très pratiques : `http://lipforge.ens-lyon.fr/www/pff/Fast2Sum.html`.

TwoSum est optimal

Supposons qu'un algorithme vérifie :

- pas de tests, ni d'instructions min/max ;
- seulement des additions/soustractions arrondies au + près : à l'étape i, on calcule $\mathrm{RN}(u+v)$ ou $\mathrm{RN}(u-v)$, où u et v sont des variables d'entrée ou des valeurs précédemment calculées.

Si cet algorithme retourne toujours les mêmes résultats que 2Sum, alors il nécessite au moins 6 *additions/soustractions (i.e., autant que 2Sum).*

- **preuve :** most inelegant proof award ;
- 480756 algorithmes avec 5 opérations (après suppression des symétries les plus triviales) ;
- chacun d'entre eux essayé avec 2 valeurs d'entrée bien choisies.

Revenons au calcul de sommes

Compensated summation (Kahan) pour approcher $x_1 + x_2 + \cdots x_n$.

Algorithme 3

$s \leftarrow x_1$
$c \leftarrow 0$
for $i = 2$ *to* n ***do***
  $y \leftarrow RN(x_i - c)$
  $t \leftarrow RN(s + y)$
  $c \leftarrow RN(RN(t - s) - y)$
  $s \leftarrow t$
end for
return s

Additionner n nombres : compensated summation (Kahan)

Le même, réécrit avec des Fast2Sum.

Algorithme 4

$s \leftarrow x_1$
$c \leftarrow 0$
for $i = 2$ *to* n ***do***
  $y \leftarrow \circ(x_i + c)$
  $(s, c) \leftarrow Fast2Sum(s, y)$
end for
return s

$\rightarrow$ variante de l'algorithme naïf, où à chaque pas on ré-injecte l'erreur de l'addition précédente.

Pichat, Ogita, Rump, Oishi

Algorithme 5

$s \leftarrow x_1$
$e \leftarrow 0$
for $i = 2$ *to* n **do**
  $(s, e_i) \leftarrow 2Sum(s, x_i)$
  $e \leftarrow RN(e + e_i)$
end for
return $RN(s + e)$

$\rightarrow$ variante de l'algorithme naïf, où à chaque pas on accumule les erreurs des additions pour les rajouter à la fin.

Pichat, Ogita, Rump, Oishi

On rappelle que $\mathbf{u} = 2^{-p}$ (base 2 et précision p), et que

$$\gamma_n = \frac{n\mathbf{u}}{1 - n\mathbf{u}}.$$

Théorème 2 (Ogita, Rump et Oishi)

En appliquant l'algorithme de P.,O., R., et O. à x_i, $1 \leq i \leq n$, et si $n\mathbf{u} < 1$, alors, même en cas d'underflow (mais sans overflow), le résultat final retourné par l'algorithme, σ, satisfait

$$\left|\sigma - \sum_{i=1}^{n} x_i\right| \leq \mathbf{u}\left|\sum_{i=1}^{n} x_i\right| + \gamma_{n-1}^2 \sum_{i=1}^{n} |x_i|.$$

Table: Erreurs de diverses méthodes pour $x_i = \mathrm{RN}(1/i)$ – donc tous les x_i sont exactement représentables – en arithmétique Binary32 et $n = 100,000$.

méthode	erreur en ulps
ordre croissant	6.86
ordre décroissant	738.9
compensated (Kahan)	0.137
Pichat ; ou Rump, Ogita & Oishi	0.137

Table: Erreurs de diverses méthodes pour $x_i = \mathrm{RN}(cos(i))$ – donc tous les x_i sont exactement représentables – en arithmétique Binary32 et $n = 5000$.

méthode	erreur en ulps
increasing order	18.90625
decreasing order	16.90625
compensated (Kahan)	6.90625
Pichat ; or Ogita, Rump and Oishi	0.09375

Une autre propriété (Kahan)

$$z = \frac{x}{\sqrt{x^2 + y^2}}$$

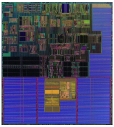
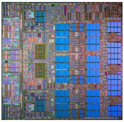
- arrondi correct, arrondi au plus près, base 2 ; pas d'overflow/underflow.
- x et y sont des nombres VF,

La valeur calculée de z est comprise au sens large entre -1 et $+1$.

- Propriété importante car jugée (naïvement !) évidente par la plupart des programmeurs, qui pourront par exemple calculer une fonction de z définie seulement entre -1 et $+1$.
- Le programmeur expérimenté de 2012 rejoint le programmeur naïf de 1980 !

Et les produits ?

- FMA : *fused multiply-add* (fma), calcule $\mathrm{RN}(ab+c)$. RS6000, PowerPC Itanium, Bulldozer, Haswell. Spécifié dans IEEE 754-2008
- si a et b sont des nombres VF, alors $r = ab - \mathrm{RN}(ab)$ est un nombre VF ;
- obtenu par l'algorithme TwoMultFMA $\left\{ \begin{array}{lcl} p & = & \mathrm{RN}(ab) \\ r & = & \mathrm{RN}(ab-p) \end{array} \right.$ $\rightarrow 2$ opérations seulement. $p + r = ab$.
- sans fma, algorithme de Dekker : 17 opérations ($7\ \times$, $10\ \pm$).

Itanium 2   PowerPC 5

$ad - bc$ "naïf" avec un fma

- on a envie de calculer $\hat{w} = \mathrm{RN}(bc)$;
- puis (avec un fma), $\hat{x} = \mathrm{RN}(ad - \hat{w})$.

peut être catastrophique (bien pire que de faire $\mathrm{RN}(\mathrm{RN}(ad) - \mathrm{RN}(bc))$). En effet, considérons le cas :

- $a = b$, et $c = d$;
- ad n'est pas exactement représentable en VF : $\hat{w} = \mathrm{RN}(ad) \neq ad$.

On aura :

- la valeur exacte de $x = ad - bc$ est nulle ;
- $\hat{x} \neq 0$,

$\rightarrow$ l'erreur relative $|\hat{x} - x|/|x|$ est infinie.

$ad - bc$ précis avec un fma (base 2)

Algorithme de Kahan pour $x = ad - bc$:

$\hat{w} \leftarrow \mathrm{RN}(bc)$
$e \leftarrow \mathrm{RN}(\hat{w} - bc)$
$\hat{f} \leftarrow \mathrm{RN}(ad - \hat{w})$
$\hat{x} \leftarrow \mathrm{RN}(\hat{f} + e)$
Retourner $\hat{x}$

$\mathbf{u} = 2^{-p}$

- avec le "modèle standard" :

$$|\hat{x} - x| \leq J|x|$$

où $J = 2\mathbf{u} + \mathbf{u}^2 + (\mathbf{u} + \mathbf{u}^2)\mathbf{u}\frac{|bc|}{|x|}$ $\rightarrow$ précis tant que $\mathbf{u}|bc| \not\gg |x|$

- en utilisant les propriétés de RN (Jeannerod, Louvet, M., 2011)

$$|\hat{x} - x| \leq 2\mathbf{u}|x|$$

asymptotiquement optimal.

$\rightarrow$ $\times$ et $\div$ complexes.

Erreur d'un FMA

- En collaboration avec Sylvie Boldo (2005) ;
- $\beta = 2$, $p \geq 3$, fma, ni underflow ni overflow ;
- a, x, y : nombres VF ;
- un fma calcule $r_1 = \mathrm{RN}(ax + y)$;
- Deux questions :
  - combien faut-il de nombres VF pour représenter $r_1 - (ax + y)$?
  - peut-on les calculer facilement ?
- Réponses :
  - deux nombres ;
  - il faut 19 opérations (1 TwoMultFMA, 2 TwoSum, 2 additions, 1 FastTwoSum) ;
  - Je n'avais aucune confiance en notre preuve avant que Sylvie Boldo la transcrive en Coq.

Peut servir à faire des "évaluations de polynômes compensées".

L'algorithme de Malcolm-Gentleman

Arithmétique VF de base β. Fonction d'arrondi $\circ \in \{\mathrm{RN}, \mathrm{RU}, \mathrm{RD}, \mathrm{RZ}\}$.

Algorithme 6

$A \leftarrow 1.0$
$B \leftarrow 1.0$
while $\circ(\circ(A + 1.0) - A) = 1.0$ ***do***
  $A \leftarrow \circ(2 \times A)$
end while
while $\circ(\circ(A + B) - A) \neq B$ ***do***
  $B \leftarrow \circ(B + 1.0)$
end while
return B

Cet algorithme calcule β

- Base β, précision p, $A_i = A$ après $i$ème passage dans le 1er `while`;
- Récurrence $\rightarrow$ si $2^i \leq \beta^p - 1$, alors $A_i = 2^i$ exactement. Donne $A_i + 1 \leq \beta^p \rightarrow \circ(A_i + 1.0) = A_i + 1$.
- On en déduit $\circ(\circ(A_i + 1.0) - A_i) = \circ((A_i + 1) - A_i) = 1$. Donc tant que $2^i \leq \beta^p - 1$,on reste dans la 1ère boucle.
- Considérons le plus petit j t.q. $2^j \geq \beta^p$. On a $A_j = \circ(2A_{j-1}) = \circ(2 \times 2^{j-1}) = \circ(2^j)$. Comme $\beta \geq 2$, on déduit

$$\beta^p \leq A_j < \beta^{p+1}.$$

- Donc le successeur VF de A_j est $A_j + \beta \rightarrow \circ(A_j + 1.0)$ vaudra soit A_j soit $A_j + \beta$ donc $\circ(\circ(A_j + 1.0) - A_j)$ vaudra 0 ou β : dans tous les cas, il sera $\neq 1.0 \rightarrow$ on quitte la boucle.

Donc à la fin de la 1ère boucle **while**, A vérifie $\beta^p \leq A < \beta^{p+1}$.

Considérons la 2ème boucle **while**.

- On a vu que le successeur VF de A est $A + \beta$.
- Donc tant que $B < \beta$, $\circ(A + B)$ vaut soit A soit $A + \beta \rightarrow \circ(\circ(A + B) - A)$ vaut 0 ou β : dans les 2 cas on reste dans la boucle.
- Dès que $B = \beta$, $\circ(A + B)$ vaut exactement $A + B$, donc $\circ(\circ(A + B) - A) = B$.

On quitte donc la 2ème boucle dès que $B = \beta$

Les erreurs les plus dangereuses sont idiotes

- la sonde Mars Climate Orbiter s'est écrasée sur Mars en 1999 ;
- une partie des développeurs des logiciels supposait que l'unité de mesure était le mètre ;
- l'autre partie croyait que c'était le pied.

Donc tout va pour le mieux dans le meilleur des mondes...

- arrondi correct $\rightarrow$ arithmétique "déterministe" ;
- on calcule facilement l'erreur d'une addition ou multiplication VF ;
- on peut "ré-injecter" cette erreur plus tard $\rightarrow$ sommes, produits scalaires, évaluations de polynômes, ... précis
- déjà de nombreux tels algorithmes compensés, sûrement d'autres à venir.

...sauf que la vie n'est pas si simple !

Arithmétique déterministe ?

Programme C :

```
double a = 1848874847.0;
double b = 19954562207.0;
double c;
c = a * b;
printf("c = %20.19e\n", c);
return 0;
```

Selon l'environnement, `3.6893488147419103232e+19` ou `3.6893488147419111424e+19` (nombre Binary64 le plus proche du résultat exact).

Arithmétique déterministe ?

option de compilation gcc	résultat
par défaut	c = 3.6893488147419103232e+19
-mfpmath=387	c = 3.6893488147419103232e+19
-march=pentium4 -mfpmath=sse	c = 3.6893488147419111424e+19

Table: Résultats sur plateforme Linux/Debian Etch 32-bit Intel, avec gcc 4.1.2 20061115. Le défaut est d'utiliser les registres 387.

Note : le "bon résultat" est `3.6893488147419111424e+19`.

Doubles arrondis

- plusieurs formats VF dans un même environnement → difficile de savoir dans quel format certaines opérations sont faites ;
- peut rendre le résultat d'une suite d'opérations difficile à prédire ;

Supposons que routes les variables déclarées soient du même format. Deux phénomènes peuvent se produire si un + grand format est disponible :

- variables implicites t.q. le résultat de "`a+b`" dans "`d = (a+b)*c`" : difficile de savoir dans quel format elles sont calculées ;
- variables explicites : peuvent être d'abord arrondies dans le format de destination → conduit à un double arrondi.

Que s'est-il produit dans l'exemple ?

La valeur de `a*b` est 36893488147419107329. En binaire :

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}\,10000000000}^{64\text{ bits}}\,01$$

Le produit est d'abord arrondi au format INTEL "double-extended", on obtient

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}\,10000000000}^{64\text{ bits}} \times 4$$

quand cette valeur intermédiaire est arrondie au format de destination binary64, cela donne (arrondi : RN)

$$\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}} \times 2^{13}$$

$$= 36893488147419103232_{10},$$

→ produit arrondi vers le bas, alors qu'il aurait dû être arrondi vers le haut.

Est-ce un problème ?

- Dans la plupart des applications, sans incidence ;
- peut rendre le résultat de programmes numériques difficile à prédire (exemples intéressants dûs à Monniaux) ;
- la plupart des compilateurs permettent de s'affranchir du problème. Cependant,
  - parfois mal documenté ;
  - restriction à la portabilité ;
  - impact possible sur la vitesse et la précision

→ voir quelles propriétés restent varies en présence de doubles arrondis (par exemple : quels algorithmes de sommation continuent à être très précis).

Pas de problème avec les instructions SSE, et IEEE 754-2008 améliore la situation.

Exemple : 2Sum et doubles arrondis

Format cible de précision p ; format plus large de précision $p + p'$.

Algorithme 7 (2Sum-with-double-roundings(a, b))

(1) $s \leftarrow RN_p(RN_{p+p'}(a+b))$ *or* $RN_p(a+b)$
(2) $a' \leftarrow RN_p(RN_{p+p'}(s-b))$ *or* $RN_p(s-b))$
(3) $b' \leftarrow \circ(s-a')$
(4) $\delta_a \leftarrow RN_p(RN_{p+p'}(a-a'))$ *or* $RN_p(a-a')$
(5) $\delta_b \leftarrow RN_p(RN_{p+p'}(b-b'))$ *or* $RN_p(b-b')$
(6) $t \leftarrow RN_p(RN_{p+p'}(\delta_a+\delta_b))$ *or* $RN_p(\delta_a+\delta_b)$

$\circ(u)$: $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, ou n'importe quel "arrondi fidèle".

Exemple : 2Sum et doubles arrondis

Théorème 2

$p \geq 4$ et $p' \geq 2$. Si a et b sont des nombres VF de précision p, en l'absence d'overflow, l'algorithme **??** *vérifie $t = RN_p(a + b - s)$.*

$\rightarrow$ de nombreuses propriétés restent vraies ou ne demandent que peu de modifications ;

- suivre les travaux de Sylvie Boldo (`http://www.lri.fr/~sboldo/`), sur des preuves "hardware-independent" de programmes.

ULP : Unit in the Last Place

Base β, précision p. Dans ce qui suit, x est un réel, et X un nombre VF censé l'approcher.

Définition 3

Si $x \in [\beta^e, \beta^{e+1})$ alors $\mathrm{ulp}\,(x) = \beta^{\max(e, e_{\min}) - p + 1}$.

En gros :

- distance entre 2 nombres VF au voisinage de x.
- arrondi au + près $\simeq$ erreur majorée par $1/2\,\mathrm{ulp}$.

ULP : Unit in the Last Place

Propriété 1

En base 2,

$$|X - x| < \frac{1}{2}\, ulp(x) \Rightarrow X = RN(x).$$

Pas vrai en base ≥ 3. Pas vrai non plus (même en base 2) si on remplace $\mathrm{ulp}(x)$ par $\mathrm{ulp}(X)$.

Propriété 2

Dans n'importe quelle base,

$$X = RN(x) \Rightarrow |X - x| \leq \frac{1}{2}\, ulp(x).$$

Division par itération de Newton-Raphson avec un FMA

Version simplifiée d'un algorithme utilisé sur l'Itanium d'Intel/HP. Précision p, base 2. Ici : $1/b$ seulement, où $1 \leq b < 2$ (mantisses).

- Itération de Newton-Raphson pour calculer $1/b$:
  $y_{n+1} = y_n - f(y_n)/f'(y_n)$, où $f(y) = 1/y - b$, ce qui donne

$$y_{n+1} = y_n(2 - by_n)$$

- $y_0 \approx 1/b$ obtenu par lecture dans une table adressée par les premiers (typiquement de 6 à 10) bits de b ;
- l'itération de NR est décomposée en 2 instructions FMA :

$$\begin{cases} e_n & = & \mathrm{RN}(1 - by_n) \\ y_{n+1} & = & \mathrm{RN}(y_n + e_n y_n) \end{cases}$$

Noter que $e_{n+1} \approx e_n^2$.

Propriété 3

Si

$$\left|\frac{1}{b} - y_n\right| < \alpha 2^{-k},$$

où $1/2 < \alpha \leq 1$ *et* $k \geq 1$*, alors*

$$\begin{aligned}\left|\frac{1}{b} - y_{n+1}\right| &< b\left(\frac{1}{b} - y_n\right)^2 + 2^{-k-p} + 2^{-p-1} \\ &< 2^{-2k+1}\alpha^2 + 2^{-k-p} + 2^{-p-1}\end{aligned}$$

$\Rightarrow$ il semble qu'on peut s'approcher arbitrairement près de 2^{-p-1} (i.e., $1/2\,\mathrm{ulp}\,(1/b)$), sans pouvoir montrer une borne inférieure à $1/2\,\mathrm{ulp}\,(1/b)$.

Exemple : double précision du standard IEEE-754

Supposons $p = 53$ et $|y_0 - \frac{1}{b}| < 2^{-8}$ (petite table), on trouve

- $|y_1 - 1/b| < 0.501 \times 2^{-14}$
- $|y_2 - 1/b| < 0.51 \times 2^{-28}$
- $|y_3 - 1/b| < 0.57 \times 2^{-53} = 0.57\,\mathrm{ulp}\,(1/b)$

Aller plus loin ?

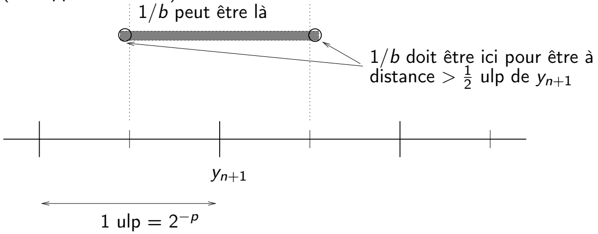
Propriété 4

Si y_n *approche* $1/b$ *avec une erreur* $< 1\,ulp\,(1/b) = 2^{-p}$*, alors, puisque* b *est un multiple de* 2^{-p+1} *et* y_n *est un multiple de* 2^{-p}*,* $1 - by_n$ *est un multiple de* 2^{-2p+1}*.*
Mais $|1 - by_n| < 2^{-p+1} \rightarrow 1 - by_n$ *est un nombre VF de précision* p
$\rightarrow$ *calculé exactement avec un FMA.*

$$\Rightarrow \left|\frac{1}{b} - y_{n+1}\right| < b\left(\frac{1}{b} - y_n\right)^2 + 2^{-p-1}.$$

$$\left| y_n - \frac{1}{b} \right| < \alpha 2^{-p} \Rightarrow \left| y_{n+1} - \frac{1}{b} \right| < b\alpha^2 2^{-2p} + 2^{-p-1}$$

(en supposant $\alpha < 1$)



Qu'en déduit-on ?

- pour être à distance $> 1/2$ ulp de y_{n+1}, $1/b$ doit être à moins de $b\alpha^2 2^{-2p} < b2^{-2p}$ du milieu de 2 nombres VF consécutifs ;
- la distance entre y_n et $1/b$ doit donc être de la forme $2^{-p-1} + \epsilon$, avec $|\epsilon| < b2^{-2p}$;
- implique $\alpha < \frac{1}{2} + b2^{-p}$ et donc

$$\left| y_{n+1} - \frac{1}{b} \right| < \left(\frac{1}{2} + b2^{-p} \right)^2 b2^{-2p} + 2^{-p-1}$$

- donc, pour être à distance $> 1/2$ ulp de y_{n+1}, $1/b$ doit être à moins de $\left(\frac{1}{2} + b2^{-p}\right)^2 b2^{-2p}$ du milieu de 2 nombres VF consécutifs.

- b est un nombre VF entre 1 et 2 $\Rightarrow b = B/2^{p-1}$ où $B \in \mathbb{N}$, $2^{p-1} < B \leq 2^p - 1$;
- le milieu de 2 nombres VF consécutifs au voisinage de $1/b$ est de la forme $g = (2G+1)/2^{p+1}$ où $G \in \mathbb{N}$, $2^{p-1} \leq G < 2^p - 1$;
- on en déduit
$$\left| g - \frac{1}{b} \right| = \left| \frac{2BG + B - 2^{2p}}{B.2^{p+1}} \right|$$
- la distance entre $1/b$ et le milieu de 2 nombres VF consécutifs est un multiple non nul de $1/(B.2^{p+1}) = 2^{-2p}/b$.

Distance entre $\frac{1}{b}$ et g, quand $\left|\frac{1}{b} - y_{n+1}\right| > \frac{1}{2}\,\mathrm{ulp}\left(\frac{1}{b}\right)$

- est de la forme $k2^{-2p}/b$, $k \in \mathbb{Z}$, $k \neq 0$;
- on doit avoir
$$\frac{|k| \cdot 2^{-2p}}{b} < \left(\frac{1}{2} + b2^{-p}\right)^2 b2^{-2p}$$
donc
$$|k| < \left(\frac{1}{2} + b2^{-p}\right)^2 b^2$$
- comme $b < 2$, dès que $p \geq 4$, la seule solution est $|k| = 1$;
- de plus, pour $|k| = 1$, des manipulations élémentaires montrent que la seule solution est
$$b = 2 - 2^{-p+1}.$$

Comment procède-t-on ?

- on veut
$$B = 2^p - 1,$$
$$2^{p-1} \leq G \leq 2^p - 1$$
$$B(2G+1) = 2^{2p} \pm 1$$
Une seule solution : $B = 2^p - 1$ et $G = 2^{p-1}$: vient de $2^{2p} - 1 = (2^p - 1)(2^p + 1)$;
- à l'exception de ce B (et donc de la valeur correspondante $b = B/2^{p-1}$ de b), on est certains que $y_{n+1} = \mathrm{RN}\,(1/b)$;
- pour $B = 2^p - 1$: on essaye l'algorithme avec les 2 valeurs de y_n à moins de 1 ulp de $1/b$ (i.e. $1/2$ et $1/2 + 2^{-p}$). En pratique, marche (sinon : trucs "sales").

Application : double précision ($p = 53$)

On part de y_0 tel que $|y_0 - \frac{1}{b}| < 2^{-8}$. On calcule :

$$\begin{array}{rcll}
e_0 & = & \mathrm{RN}\,(1 - by_0) & \\
y_1 & = & \mathrm{RN}\,(y_0 + e_0 y_0) & \\
e_1 & = & \mathrm{RN}\,(1 - by_1) & \\
y_2 & = & \mathrm{RN}\,(y_1 + e_1 y_1) & \\
e_2 & = & \mathrm{RN}\,(1 - by_1) & \\
y_3 & = & \mathrm{RN}\,(y_2 + e_2 y_2) & \text{error} \leq 0.57 \text{ ulps} \\
e_3 & = & \mathrm{RN}\,(1 - by_2) & \\
y_4 & = & \mathrm{RN}\,(y_3 + e_3 y_3) & 1/b \text{ arrondi au plus près}
\end{array}$$

En pratique : deux itérations

Itérations de Markstein

$$\begin{cases} e_n & = & \text{RN}\,(1 - by_n) \\ y_{n+1} & = & \text{RN}\,(y_n + e_n y_n) \end{cases}$$

Plus précise ("auto-correctrice"), séquentielle

Itérations de Goldschmidt

$$\begin{cases} e_{n+1} & = & \text{RN}\,(e_n^2) \\ y_{n+1} & = & \text{RN}\,(y_n + e_n y_n) \end{cases}$$

Moins précis, plus rapide (parallèle)

En pratique : on commence aver des itérations de Goldschmidt, et on passe à des itérations de Markstein pour les derniers pas.

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.8522008497671888017727\cdots$
HP 48 GX	-0.852200849762
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\cdots$
matlab V.4.2 c.1 pour Macintosh	0.8740
matlab V.4.2 c.1 pour SPARC	-0.8522
Silicon Graphics Indy	$0.87402806\cdots$
SPARC	-0.85220084976718879
IBM RS/6000 AIX 3005	$-0.852200849\cdots$
DECstation 3100	NaN
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

Jusqu'à août 2008, fonctions « élémentaires » non normalisées.

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.8522008497671888017727\cdots$
HP 48 GX	-0.852200849762
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\cdots$
matlab V.4.2 c.1 pour Macintosh	0.8740
matlab V.4.2 c.1 pour SPARC	-0.8522
Silicon Graphics Indy	$0.87402806\cdots$
SPARC	-0.85220084976718879
IBM RS/6000 AIX 3005	$-0.852200849\cdots$
DECstation 3100	NaN
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

Jusqu'à août 2008, fonctions « élémentaires » non normalisées.

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.8522008497671888017727\cdots$
HP 48 GX	-0.852200849762
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\cdots$
matlab V.4.2 c.1 pour Macintosh	0.8740
matlab V.4.2 c.1 pour SPARC	-0.8522
Silicon Graphics Indy	$0.87402806\cdots$
SPARC	-0.85220084976718879
IBM RS/6000 AIX 3005	$-0.852200849\cdots$
DECstation 3100	NaN
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

Jusqu'à août 2008, fonctions « élémentaires » non normalisées.

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.8522008497671888017727\cdots$
HP 48 GX	-0.852200849762
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\cdots$
matlab V.4.2 c.1 pour Macintosh	0.8740
matlab V.4.2 c.1 pour SPARC	-0.8522
Silicon Graphics Indy	$0.87402806\cdots$
SPARC	-0.85220084976718879
IBM RS/6000 AIX 3005	$-0.852200849\cdots$
DECstation 3100	NaN
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

Jusqu'à août 2008, fonctions « élémentaires » non normalisées.

Le dilemme du fabricant de tables

Considérons le nombre binary64 ($\beta = 2, p = 53$)

$$x = \frac{8520761231538509}{2^{62}}$$

On a

$$2^{53+x} = 9018742077413030.999999999999999998805240837303\cdots$$

de même, pour :

$$x = 9.407822313572878 \times 10^{-2},$$

on a

$$e^x = \underbrace{1.098645682066338}_{16 \text{ chiffres}}\,5\,\underbrace{0000000000000000}_{16 \text{ zéros}}\,2780\cdots$$

Et alors ?

Pires cas pour l'arrondi de la fonction 2^x, et des nombres binary64, et pour celui de e^x et des nombres decimal64.

Arrondi correct des fonctions élémentaires

- base 2, précision p ;
- nombre VF x et entier m (avec $m > p$) $\rightarrow$ approximation y de $f(x)$ dont l'erreur sur la mantisse est $\leq 2^{-m}$.
- peut être fait avec format intermédiaire plus grand, avec TwoSum, TwoMultFMA, etc.
- obtenir un arrondi correct de $f(x)$ à partir de y : impossible si $f(x)$ est trop proche d'un point où l'arrondi change (en arrondi au plus près : milieu de 2 nombres VF consécutifs).

$\rightarrow$ il faut trouver—quand il existe—le plus petit m qui convienne pour tous les nombres VF.

Résultats

Table: *Pires cas pour l'exponentielle de nombres double précision.*

Interval	worst case (binary)
$[-\infty, -2^{-30}]$	$\exp(-1.1110110100110001100011101111110110110001001111101010 \times 2^{-27})$ $= 1.111111111111111111111111100 \cdots 0111000100 \quad 1 \quad 1^{59}0001\ldots \times 2^{-1}$
$[-2^{-30}, 0)$	$\exp(-1.0000000000000000000000000000000000000000000000000001 \times 2^{-51})$ $= 1.111111111111111 \cdots 11111111111111100 \quad 0 \quad 0^{100}1010\ldots \times 2^{-1}$
$(0, +2^{-30}]$	$\exp(1.1111111111111111111111111111111111111111111111111111 \times 2^{-53})$ $= 1.0000000000000000000000000000000000000000000000000000 \quad 1 \quad 1^{104}0101\ldots$
$[2^{-30}, +\infty]$	$\exp(1.0111111111111110011111111111111011100000000000100100 \times 2^{-32})$ $= 1.0000000000000000000000000000000101111111111111101000 \quad 0 \quad 0^{57}1101\ldots$
	$\exp(1.1000000000000001011111111111111011011111111111011100 \times 2^{-32})$ $= 1.0000000000000000000000000000000110000000000000010111 \quad 1 \quad 1^{57}0010\ldots$
	$\exp(1.1001111010011100101110111111110101100000010000001011 \times 2^{-31})$ $= 1.0000000000000000000000000000001100111101001110010111 \quad 1 \quad 0^{57}1010\ldots$
	$\exp(110.00001111010100101111001101111010111011001111110100)$ $= 110101100.01010000101101000000100111001000101011101110 \quad 0 \quad 0^{57}1000\ldots$

Résultats

Table: *Pires cas pour les logarithmes de nombres double précision.*

Interval	worst case (binary)
$[2^{-1074}, 1)$	$\log(1.1110101001110001110110000101110011101110000000100000 \times 2^{-509})$ $= -101100000.0010100101101010011001101011010000101111111\ \ 1\ \ 1^{60}0000\ldots$
	$\log(1.1001010001110110111000110000010011001101011111000111 \times 2^{-384})$ $= -100001001.1011011000001100101011110100011110110011010 1\ \ 1\ \ 0^{60}1010\ldots$
	$\log(1.0010011011101001110001001101001100100111100101100000 \times 2^{-232})$ $= -10100000.1010101100101100001001011110011010000100001 00\ \ 0\ \ 0^{60}1001\ldots$
	$\log(1.0110000100111001010101011101110010000000001011111000 \times 2^{-35})$ $= -10111.111100000001011111001101110101111011000000011010 1\ \ 0\ \ 1^{60}0011\ldots$
$(1, 2^{1024}]$	$\log(1.0110001010101000100001100001001101100010100110110110 \times 2^{678})$ $= 111010110.0100011110011110101110100111110010010111000 1\ \ 0\ \ 0^{64}1110\ldots$

Conclusion

- l'arrondi correct des fonctions les plus communes est faisable à coût raisonnable ;
- recommandé dans la norme IEEE 754-2008 (juin 2008) ;
- bibliothèque CRLIBM disponible à `https://lipforge.ens-lyon.fr/projects/crlibm/`

Quelques logiciels utiles

- CRLIBM : fonctions mathématiques avec arrondi correct.
  http://lipforge.ens-lyon.fr/projects/crlibm/
  La documentation qui explique les méthodes utilisées est à
  http:
  //lipforge.ens-lyon.fr/frs/download.php/41/crlibm-0.10.pdf
- GAPPA : outil de vérifications de propriétés VF (p.ex. bornes) et de génération de preuves formelles.
  http://lipforge.ens-lyon.fr/www/gappa/
- MPFR : arithmétique multi-précision avec arrondi correct.
  http://www.mpfr.org/
- MPFI (basé sur MPFR) : arith. d'intervalles multi-précision.
  http://perso.ens-lyon.fr/nathalie.revol/software.html
- PARI/GP : calculs rapides en arithmétique (factorisations, théorie algébrique des nombres, courbes elliptiques. . .).
  http://pari.math.u-bordeaux.fr/

Tester votre environnement virgule flottante

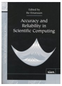
- PARANOIA (W. Kahan et ses élèves) :
  http://www.netlib.org/paranoia/
- UCB Test (plus récent, plus complet) :
  http://www.netlib.org/fp/ucbtest.tgz
- MPCHECK : test des fonctions mathématiques :
  http://www.loria.fr/~zimmerma/free/
- méthodes de recherches de "cas difficiles" pour les opérateurs arithmétiques : John Harrison (factorisation), Michael Parks (remontée de Hensel) ;

La virgule flottante sur le web

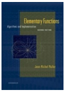
- le site de W. Kahan (père de la norme IEEE 754, de l'arithmétique du 8087 et de la HP35) :
  http://http.cs.berkeley.edu/~wkahan/
- le site de D. Hough sur la révision de la norme
  http://www.validlab.com/754R/
- l'article de Goldberg "What every computer scientist should know about Floating-Point arithmetic"
  http://www.validlab.com/goldberg/paper.pdf
- l'équipe AriC du LIP (ENS Lyon)
  http://www.ens-lyon.fr/LIP/AriC/
- l'équipe Pequan du LIP6 (Paris 6)
  http://www.lip6.fr/recherche/team.php?id=120
- l'équipe CACAO du Loria (Nancy)
  http://www.loria.fr/equipes/cacao/
- ma propre page
  http://perso.ens-lyon.fr/jean-michel.muller/

Une minute de pub

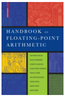
Michael Overton
Numerical Computing with IEEE Floating Point Arithmetic
Siam, 2001



Bo Einarsson
Accuracy and Reliability in Scientific Computing
Siam, 2005



Jean-Michel Muller
Elementary Functions, algorithms and implementation, 2ème édition
Birkhauser Boston, 2006



Brisebarre, de Dinechin, Jeannerod, Lefèvre, Melquiond, Muller (coordinator), Revol, Stehlé and Torres
A Handbook of Floating-Point Arithmetic
Birkhauser Boston, 2010.