Virgule Flottante: la lente convergence vers la rigueur

Jean-Michel Muller
CNRS - Laboratoire LIP
(CNRS-INRIA-ENS Lyon-Université de Lyon)

Minisymposium «Recherche Reproductible» – CANUM 2016

Arithmétique « virgule flottante »

- trop souvent perçue comme un tas de recettes de cuisine ;
- de simples modèles tels que

  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, $\top \in \{\pm, \times, \div\}$) 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 utilisables sans danger ?

Propriétés souhaitables d'une arithmétique machine

- Critères de performance :
  - Vitesse : la météo de demain doit être calculée en moins de 24 heures ;
  - Précision, dynamique : par exemple, certaines prédictions physiques vérifiées avec erreur relative $\approx 10^{-14}$;
- Critères technologiques
  - « taille » : surface de circuit, taille du code, consommation mémoire ;
  - Energie consommée : autonomie, chauffe des circuits ;
- Critères de coût humain
  - 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.

Système virgule flottante

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

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

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

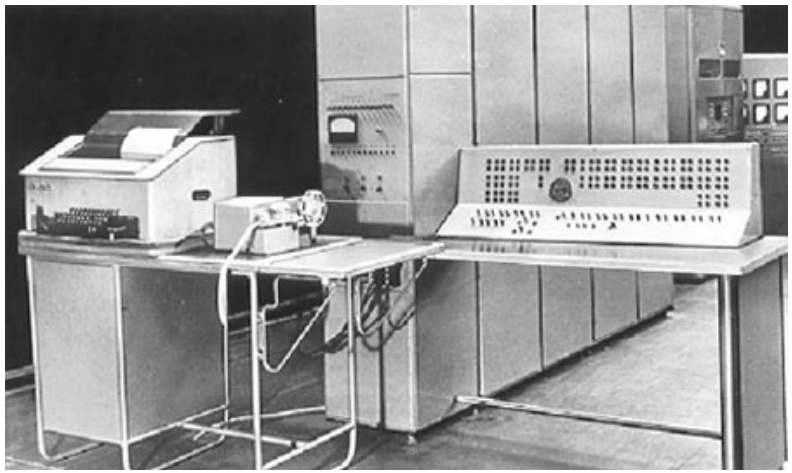
Si plusieurs choix, on prend e minimal sous ces contraintes. On appelle mantisse réelle, ou mantisse de x le nombre

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

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

Quelques étrangetés

- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;
- Maple, version 6.0. Entrez 214748364810, vous obtiendrez 10. Noter que $2147483648 = 2^{31}$.
- Excel'2007 (premières versions), calculez $65535 - 2^{-37}$, vous obtiendrez 100000 ;
- Machine Setun, université de Moscou, 1958 (50 exemplaires) : base 3 et chiffres -1, 0 et 1. Nombres sur 18 «trits».



Leonardo Torres y Quevedo (1852–1936)

- version électromécanique (à relais) de la machine analytique de Babbage ;
- première proposition d'une arithmétique VF ;

L'article de Torres « Essais sur l'Automatique »

- En Français, *Revue Générale des Sciences*, 15 novembre 1915 ;
- contient le paragraphe :

Parfois aussi, pour ne pas avoir à écrire beaucoup de zéros, on écrit les quantités sous la forme $n \times 10^m$.

Nous pourrons simplifier beaucoup cette écriture en établissant arbitrairement ces trois règles très simples :

I. n aura toujours le même nombre de chiffres (six par exemple).

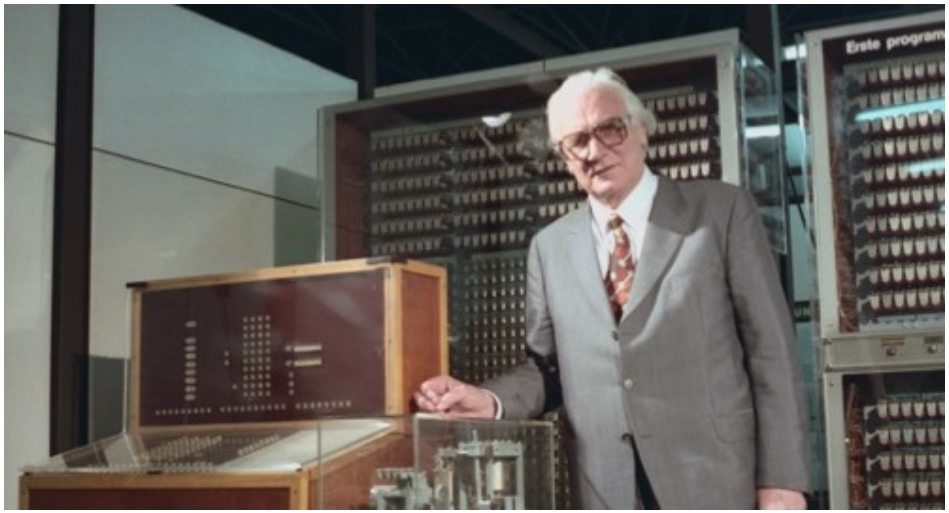
II. Le premier chiffre de n sera de l'ordre des dixièmes, le second des centièmes, etc.

III. On écrira chaque quantité sous cette forme: n, m.

Ainsi, au lieu de 2435,27 et de 0,00000341862, on écrira respectivement 243527 ; 4 et 341862 ; — 5.

Je n'ai pas indiqué de limite pour la valeur de l'exposant, mais il est évident que, dans tous les

Konrad Zuse (1910–1995) et le Z3 (1941)



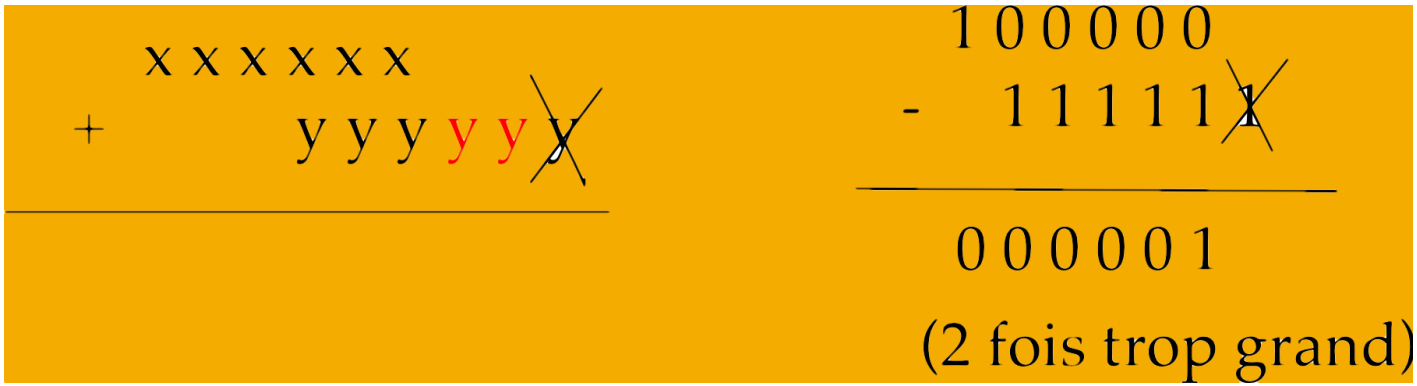
Zuse posant devant une reconstruction du Z3

- Z3 : Arithmétique VF de base 2, nombres sur 22 bits :
  - mantisses de 14 bits ;
  - exposants de 7 bits ;
  - 1 bit de signe ;
- représentations spéciales pour $\pm\infty$ et résultats indéterminés, plus de 40 ans avant IEEE 754 ;
- contrairement aux Z1 (1936–1938) et Z2 (1938), a été complètement opérationnel.

Années 50 et 60 : la VF se généralise... mais c'est la pagaille

- Base : 2, 4, 8, 10, 16... des études de Brent et Cody trancheront ;
- seuils d'over/underflow très différents ;
- pas la même manière de gérer $1/0$, $0/0$, $\sqrt{-1}$, etc. ;
- spécification floue des opérations : on raisonne en « bits de garde »

Alignement lorsqu'on calcule $x + y$ avec $e_x > e_y$:



`(1 - x)` gagnait à être remplacé par `(0.5 - x) + 0.5`

Quelques exemples (Kahan)

- Quand seule la vitesse compte : sur les Crays, l'overflow était calculé à partir des exposants des entrées, en parallèle avec le calcul effectif du produit des mantisses

→ `1 * x` peut faire un overflow ;

- sur les mêmes, seuls 12 bits de x étaient examinés pour détecter une division par 0 lors du calcul y/x

→ `if (x = 0) then z := 17.0 else z := y/x`
pouvait provoquer une erreur « division par zéro »...
mais comme le multiplieur aussi ne regardait que 12 bits pour décider qu'une opérande est nulle,

```
if (1.0 * x = 0) then z := 17.0 else z := y/x
```

→ plus de problème.

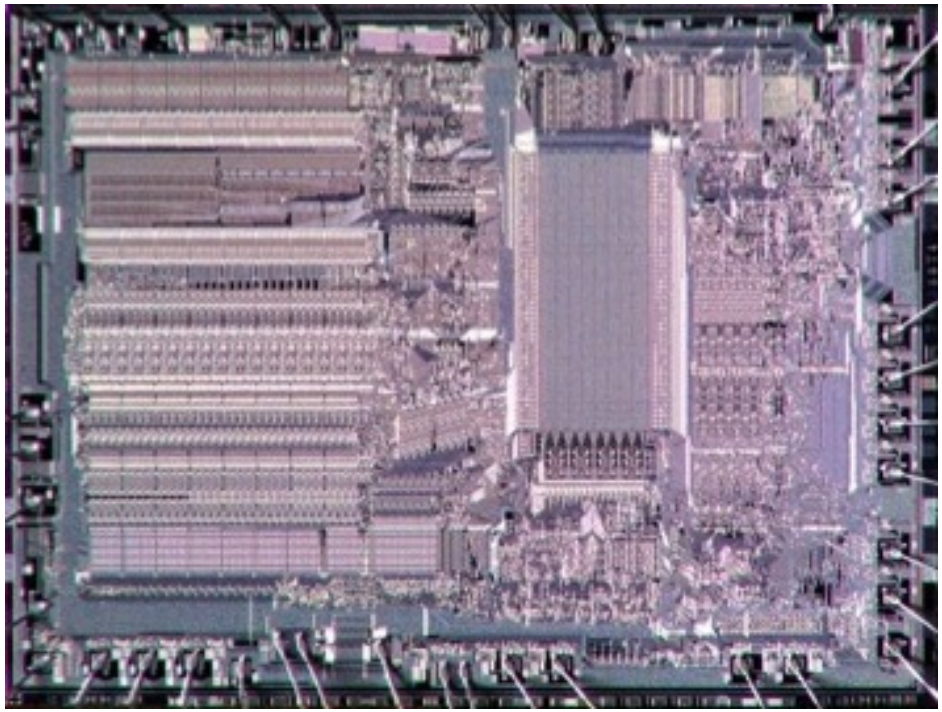
William Kahan

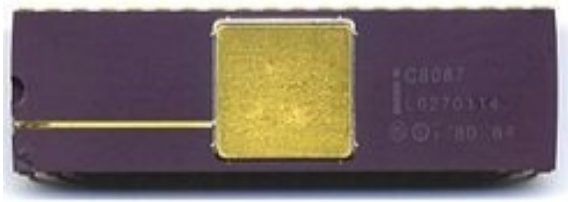
- PhD, Univ. Toronto, 1958 ;
- à programmé à peu près toutes les machines de l'époque ;
- a contribué à la conception de la calculatrice HP35 (1972) ;
- arithmétique VF du 8087 ;
- en parallèle (1977), 1ères discussions autour du futur standard IEEE 754.

La HP35 : l'objet qui a « tué » la règle à calcul

Encore beaucoup d'informations à `http://www.eecs.berkeley.edu/~wkahan`

Le 8087 d'Intel (1980)





- coprocesseur du 8086 ;
- $ 200 environ ;
- fonctions élémentaires en VF (Cordic) ;
- première « presque » implantation de ce qui sera 5 ans + tard IEEE 754 ;
- entrée de la VF rapide et propre dans le monde du personal computing ;
- format « étendu » avec mantisses de 64 bits (+ grande taille qui ne change pas le temps de cycle) ;
- 5 MHz.

IEEE 754-1985

- choix de la base 2, de formats (32 bits, 64 bits) ;
- deux idées fortes :
  - système clos : même les opérations « illicites » $(1/0\,;\,\sqrt{-5})$ fournissent un résultat, qui doit pouvoir être réutilisable en entrée ;
  - arrondi correct : une fonction d'arrondi $\circ$ étant choisie, le calcul en machine de $a \star b$ donne

$$\circ(a \star b)$$

Juste quelques mots sur les exceptions dans la norme

- philosophie par défaut : le calcul doit toujours continuer ;
- 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.

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 :

- 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.
- RU (x) : vers $+\infty$.
- RD (x) : vers $-\infty$.
- 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.

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

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

Premier résultat : représentabilité. $\mathrm{RN}(x) = x$ arrondi au plus près.

Lemme 1

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.

Obtenir r : l'algorithme fast2sum (Dekker)

Théorème 1 (Fast2Sum (Dekker))

(base ≤ 3) Soient a et b des nombres VF vérifiant $|a| \geq |b|$. 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 ».

Algorithme TwoSum (Møller-Knuth)

- pas besoin de comparer a et b ;
- 6 opérations au lieu de 3 $\rightarrow$ 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.

Exemple : calcul de $x_1 + x_2 + \cdots + x_n$

Ogita, Rump, Oishi :

Algorithme 3

$s \leftarrow x_1$
$e \leftarrow 0$
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.

TwoSum et Fast2Sum sont « Robustes »

2Sum avec arrondis quelconques : $\circ_1, \circ_2, \ldots, \circ_6$ sont des fonctions d'arrondi (RU, RD, RZ, RN). Base 2 et précision p.

(1) $s \leftarrow \circ_1(a+b)$
(2) $a' \leftarrow \circ_2(s-b)$
(3) $b' \leftarrow \circ_3(s-a')$
(4) $\delta_a \leftarrow \circ_4(a-a')$
(5) $\delta_b \leftarrow \circ_5(b-b')$
(6) $t \leftarrow \circ_6(\delta_a+\delta_b)$

Théorème 2 (Boldo, Graillat, M.)

En l'absence d'overflow, si $p \geq 4$, s et t vérifient

$$\left|\Big((a+b)-s\Big)-t\right| < 2^{-2p+2}\cdot|s|$$

De plus si $e_s - e_b \leq p-1$ alors t est un des 2 nombres VF qui encadrent $(a+b)-s$.

Théorème 3 (Boldo, Graillat, M.)

Si $|a| < \Omega$ et s'il n'y a pas d'overflow à la ligne (1), il n'y en a pas aux lignes (2) à (6).

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 $\begin{cases} p = \mathrm{RN}(ab) \\ r = \mathrm{RN}(ab - p) \end{cases} \rightarrow 2$ opérations seulement. $p + r = ab$. $\times$, 10 $\pm$).

$ad - bc$ «naïf» avec un fma

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

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 : $w = \mathrm{RN}(ad) \neq ad$.

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 $x = ad - bc$:

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

- 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.

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

Arithmétique (presque) bien spécifiée

Dès 1985, est prêt pour prouver rigoureusement les algorithmes utilisés, sauf qu'à l'époque...

- les ingénieurs/scientifiques n'en éprouvent pas vraiment le besoin : ils font de la simulation tranquilles au sol ;
- les outils de preuve formelle ne sont pas encore prêts ;
- culte du secret sur les algorithmes utilisés ;
- aucune spécification des fonctions transcendantes simples : exp, log, sin, cos, etc.
- ...et puis chez Intel, Motorola, etc. il y avait à ce moment là un côté « tonton bricoleur » sympathique mais dangereux.

Automne 1994 : le « bug » du Pentium

- erreur dans l'algorithme de division (SRT de base 4) ;
- nombreux quotients faux. Pire cas : 4195835.0/3145727.0 donne 1.33373906802 au lieu de 1.3338204491 ;
- tempête électronique sur Internet ;
- Intel a dû remplacer les Pentium défectueux (coût : peut-être 400M$) ;
- la vraie perte a été en termes d'image de marque.

Après ceci : vrai changement de stratégie

- fin du secret sur les algorithmes VF : division de l'Itanium publiée dans les actes d'Arith14 (1999) ;
- preuve formelle : Intel embauche Harrison, AMD embauche Russinoff.

Que sont les Pentium devenus ?

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.852200849767188801772 7\cdots$
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	$-0.8522008497671 8879$
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

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.852200849767188801772 7\cdots$
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	$-0.8522008497671887 9$
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

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.852200849767188801772 7\cdots$
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	$-0.8522008497671 8879$
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

Le tableau de la honte

Système	$\sin\left(10^{22}\right)$
résultat exact	$-0.852200849767188801772 7\cdots$
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	$-0.8522008497671 8879$
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

Le dilemme du fabricant de tables

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

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

On a

$$2^{53+x} = 9018742077413030.9999999999999999805240837303\cdots$$

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.

Un résultat de Baker (1975)

- $\alpha = i/j$, $\beta = r/s$, with $i, j, r, s < 2^p$;
- $C = 2^{800}$;

$$|\alpha - \log(\beta)| > (p2^p)^{-Cp \log p}$$

Application : Pour évaluer ln et exp en double précision ($p = 53$) avec arrondi correct, il suffit de calculer une approximation précise sur environ

$$10^{244} \text{ bits}$$

Heureusement, en pratique, on constate que c'est beaucoup moins (≈ 100).

Résultats

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

Interval	worst case (binary)
$[-\infty, -2^{-30}]$	$\exp(-1.11101101001100011000111011111011011000100111111101010 \times 2^{-27})$ $= 1.11111111111111111111111100\cdots0111000100\quad 1\quad 1^{59}0001\ldots \times 2^{-1}$
$[-2^{-30}, 0)$	$\exp(-1.00000000000000000000000000000000000000000000000000001 \times 2^{-51})$ $= 1.11111111111111\cdots1111111111111100\quad 0\quad 0^{100}1010\ldots \times 2^{-1}$
$(0, +2^{-30}]$	$\exp(1.11111111111111111111111111111111111111111111111111111 \times 2^{-53})$ $= 1.00000000000000000000000000000000000000000000000000000\quad 1\quad 1^{104}0101\ldots$
$[2^{-30}, +\infty]$	$\exp(1.01111111111111100111111111111111011100000000000100100 \times 2^{-32})$ $= 1.00000000000000000000000000000001011111111111111101000\quad 0\quad 0^{57}1101\ldots$ $\exp(1.10000000000000010111111111111111011011111111111011100 \times 2^{-32})$ $= 1.00000000000000000000000000000001100000000000000010111\quad 1\quad 1^{57}0010\ldots$ $\exp(1.10011110100111001011101111111101011000001000000001011 \times 2^{-31})$ $= 1.00000000000000000000000000000001100111101001110010111\quad 1\quad 0^{57}1010\ldots$ $\exp(110.00001111010100101111001101111010111011001111110100)$ $= 110101100.01010000101101000000100111001000101011101110\quad 0\quad 0^{57}1000\ldots$

Résultats

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

Interval	worst case (binary)
$[2^{-\mathbf{1074}}, 1)$	$\log(1.11101010011100011101100001011100111011100000000100000 \times 2^{-\mathbf{509}})$
	$= -101100000.00101001011010100110011010110100001011111111 \quad 1 \quad 1^{\mathbf{60}}0000\ldots$
	$\log(1.10010100011101101110001100000100110011010111110001111 \times 2^{-\mathbf{384}})$
	$= -100001001.10110110000011001010111101000111101100110101 \quad 1 \quad 0^{\mathbf{60}}1010\ldots$
	$\log(1.00100110111010011100010011010011001001111001011000000 \times 2^{-\mathbf{232}})$
	$= -10100000.10101011001011000010010111100110100001000000100 \quad 0 \quad 0^{\mathbf{60}}1001\ldots$
	$\log(1.01100001001110010101010111011100100000000010111110000 \times 2^{-\mathbf{35}})$
	$= -10111.11110000001011111001101110101111011000000011010 1 \quad 0 \quad 1^{\mathbf{60}}0011\ldots$
$(1, 2^{\mathbf{1024}}]$	$\log(1.01100010101010001000011000010011011000101001101101 10 \times 2^{\mathbf{678}})$
	$= 111010110.01000111100111101011101001111100100101110001 \quad 0 \quad 0^{\mathbf{64}}1110\ldots$

Du progrès

- arrondi correct des fonctions les plus communes faisable à coût raisonnable ;

→ recommandé (pas exigé) dans la norme IEEE 754-2008 ;

- algorithmes publiés et preuve formelle → bien plus grande confiance.

La fin de l'adolescence pour l'arithmétique virgule flottante ?
À l'avenir :

- Outils : on ne fera plus d'opérateurs (matériels ou logiciels) mais des générateurs d'opérateurs (exploration d'un vaste espace de possibilités, correction par construction, conception adaptée à l'application) ;
- challenge : marier reproductibilité et parallélisme massif sans «tuer» la qualité au nom de la reproductibilité.