Tout ce que vous avez toujours voulu savoir sur le standard IEEE 754

Paul Zimmermann, Inria/LORIA, Nancy

Café Calcul, 10 février 2022

```
SageMath version 9.4, Release Date: 2021-08-22
Using Python 3.9.10. Type "help()" for help.
sage: 0.3-(0.2+0.1)
-5.55111512312578e-17

sage: R53z = RealField(53,rnd='RNDZ')
sage: x = R53z(pi)
sage: x.exact_rational()
884279719003555/281474976710656
sage: x.sign_mantissa_exponent()
(1, 7074237752028440, -51)
sage: x.str(16)
'3.243f6a8885a30'
sage: x == R53z("3.243f6a8885a30",16)
True
```

Plan

- Historique du standard IEEE 754
- Les formats
- Les modes d'arrondi
- Les opérations de base
- Les fonctions mathématiques

Historique du standard IEEE 754

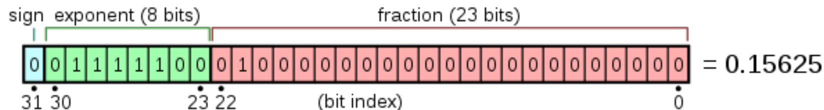
Première version en 1985 sous l'impulsion de William Kahan

$X = (X + X) - X$

Première révision en 2008 : formats décimaux (BID et DPD), fonctions mathématiques recommandées avec arrondi correct

Seconde révision en 2019 : fused multiply-add (FMA).

Le format simple précision (binary32)



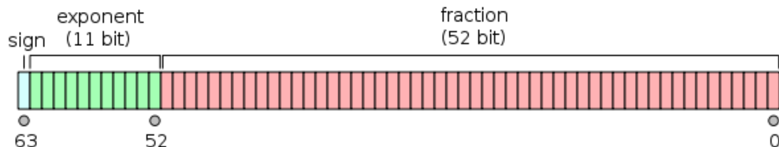
Valeur minimale $2^{-149} \approx 1.40 \cdot 10^{-45}$.

Valeur maximale $2^{128}(1 - 2^{-24}) \approx 3.40 \cdot 10^{38}$.

Précision 24 bits.

`float` en langage C.

Le format double précision (binary64)



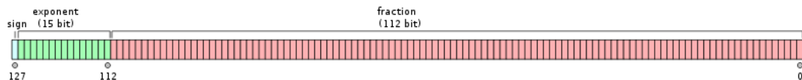
Valeur minimale $2^{-1074} \approx 4.94 \cdot 10^{-324}$.

Valeur maximale $2^{1024}(1 - 2^{-53}) \approx 1.80 \cdot 10^{308}$.

Précision 53 bits.

`double` en langage C.

Le format quadruple précision (binary128)



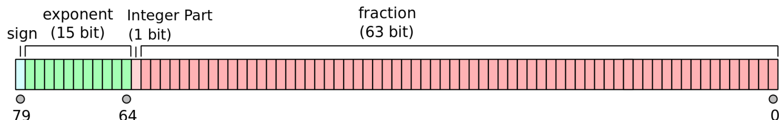
Valeur minimale $2^{-16494} \approx 6.48 \cdot 10^{-4966}$.

Valeur maximale $2^{16384}(1 - 2^{-113}) \approx 1.20 \cdot 10^{4932}$.

Précision 113 bits.

`_Float128` ou `_Quad` en langage C.

Le format double précision étendue



Valeur minimale $2^{-16445} \approx 3.65 \cdot 10^{-4951}$.

Valeur maximale $2^{16384}(1 - 2^{-64}) \approx 1.20 \cdot 10^{4932}$.

Précision 64 bits.

`long double` en langage C.

0.1 n'est pas représentable

```
sage: a=0.1;b=0.2;c=0.3

sage: a.exact_rational()
3602879701896397/36028797018963968
sage: b.exact_rational()
3602879701896397/18014398509481984
sage: c.exact_rational()
5404319552844595/18014398509481984

sage: (c-(a+b)).exact_rational()
-1/36028797018963968
sage: c-(a+b)
-5.55111512312578e-17
```

Les modes d'arrondi

- au plus proche (avec arrondi pair)
- vers zéro
- vers $+\infty$
- vers $-\infty$

Le lemme de Sterbenz

Lemma

Si a et b sont deux nombres flottants de même précision et même signe, et $|a| \leq |b| \leq 2|a|$, alors $b - a$ est exact.

```
sage: a=RR(pi/2)
sage: b=RR(e)
sage: a, b
(1.57079632679490, 2.71828182845905)
sage: c=b-a
sage: c.exact_rational() == b.exact_rational()
      - a.exact_rational()
True
sage: c.exact_rational()
2583907638853853/2251799813685248
```

FastTwoSum

Algorithme FastTwoSum.
$s \leftarrow RN(a + b)$
$e \leftarrow RN(s - a)$
$t \leftarrow RN(b - e)$
Return s, t

Lemma

Si $|a| \geq |b|$, alors les valeurs s, t renvoyées par FastTwoSum vérifient :

$$a + b = s + t.$$

Preuve : $s = a + b + \varepsilon$, $e = b + \varepsilon$ (Sterbenz), $t = -\varepsilon$.

Modes d'arrondi en C

```
#include <stdio.h>
#include <fenv.h>
main() {
   fesetround (FE_DOWNWARD);
   printf ("%a\n", 1.0f / 3.0f);
   fesetround (FE_UPWARD);
   printf ("%a\n", 1.0f / 3.0f);
}

$ gcc -frounding-math e.c -lm
0x1.555554p-2
0x1.555556p-2
```

Double arrondi

Soit $x = 0.31414999$.

Si on arrondit x au plus proche à cinq chiffres, cela donne $y = 0.31415$.

Si on arrondit y au plus proche à quatre chiffres, cela donne $z = 0.3142$ (arrondi pair).

Si on arrondit x directement à quatre chiffres, on obtient $t = 0.3141 \neq z$.

Ne se produit qu'avec l'arrondi au plus proche, et quand le second arrondi utilise la règle de l'arrondi pair.

Arrondi en précision étendue

Sur la machine gcc45 (AMD Athlon II) :

```
#include <stdio.h>
#include <fpu_control.h>
main() {
  double b = 0x1.fffffffffffffp-1;
  printf ("1/b=%la\n", 1 / b);
  int cw = 0xbfff027f; _FPU_SETCW (cw);
  printf ("1/b=%la\n", 1 / b);
}
```

```
$ gcc i.c; ./a.out
1/b=0x1p+0
1/b=0x1.0000000000001p+0
```

Même chose sur `x86_64` avec `-m32`.

Zéros

Le standard IEEE définit deux zéros : $+0$ et -0.

On a $+0 == -0$.

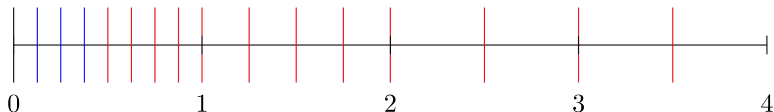
$1/+\infty$ donne $+0$.

$1/-\infty$ donne -0.

Par défaut si on écrit 0 on obtient $+0$.

Les nombres dénormalisés

(subnormal en anglais)



En double précision, le plus petit nombre normal est $a = 2^{-1022}$, et son successeur est $b = 2^{-1022} + 2^{-1074}$.

Sans dénormalisés, $b - a$ ne serait pas représentable (flush to zero).

Avec dénormalisés, $b - a$ est représentable.

Fused Multiply Add

$$\mathrm{fma}(x, y, z) = \circ(xy + z) \qquad \neq \qquad \circ(\circ(xy) + z)$$

Permet d'obtenir plus de précision.

Autre utilisation : obtenir l'erreur exacte d'une multiplication :

$h \leftarrow RN(ab)$

$\ell \leftarrow \mathrm{fma}(a, b, -h)$

$ab = h + \ell$

```
#include <stdio.h>
#include <stdlib.h>
main (int argc, char *argv[]) {
  double a = strtod (argv[1], NULL);
  double b = strtod (argv[2], NULL);
  double s = strtod (argv[3], NULL);
  double t = a * b + s;
  printf ("t=%la\n", t);
}
```

```
$ gcc -O2 fma.c; a=0x1.921fb54442d18p1;
  b=0x1.62e42fefa39efp-1; s=-0x1.16bb24190a0b7p+1;
$ ./a.out $a $b $s
t=0x0p+0
```

```
$ gcc -O2 -march=native fma.c; ./a.out $a $b $s
t=-0x1.ce22e99bf1d3p-53
```

Exceptions

- Invalid (NaN)
- Overflow ($\pm\infty$)
- Underflow (± 0)
- Division by zero
- Inexact

Exemples d'utilisation : Inexact pour vérifier que des calculs entiers restent exacts, Overflow/Underflow pour vérifier qu'on ne sort pas de l'intervalle des exposants.

Les fonctions mathématiques

IEEE 754 n'impose pas l'arrondi correct (sauf pour `sqrt`).

Erreurs maximales pour `coshf` en ulps (units in last place) :

GLIBC 2.34	IML 2021.2.0	AMD 3.7	Newlib 4.1.0	OpenLibm 0.7.5	Musl 1.2.2	Apple 11.3.1	CUDA 11.2.2
1.89	0.506	∞	2.51	1.36	1.03	0.589	2.34

Constant folding

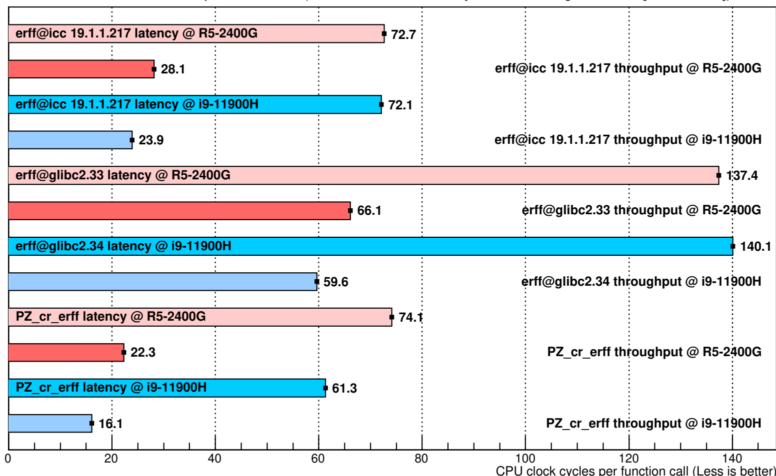
```
#include <stdio.h>
#include <math.h>

main() {
  printf ("%a\n", j0f (0x1.33d152p+1f));
}

$ gcc j0.c; ./a.out
0x1.e4c48ep-25

$ gcc -fno-builtin j0.c -lm; ./a.out
0x1.00209ap-24
```

Dépendance du hardware

```
#include <stdio.h>
#include <math.h>
int main() {
  double x = 0x1.01825ca7da7e5p+0;
  double y = acosh (x);
  printf ("x=%la y=%la\n", x, y);
}

$ icc -fno-builtin test_acosh.c

$ ./a.out # Intel Xeon E5-2680
x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cbp-4

$ ./a.out # AMD EPYC 7452, same binary file
x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cap-4
```

Le projet CORE-MATH

Fournir des implantations avec arrondi correct pour inclusion dans les bibliothèques mathématiques (libms) existantes.



Prochaine révision

Devrait avoir lieu en 2029.

Mailing list `STDS-754@LISTSERV.IEEE.ORG`

`listserv.ieee.org/cgi-bin/wa?SUBED1=STDS-754&A=1`

C Floating-Point Interest Group `cfp-interest@ucbtest.org`

`mailman.oakapple.net/mailman/listinfo/cfp-interest`

Recherche en France

GT ARITH du GDR IM.

Équipe de Jean-Michel Muller (AriC, Lyon).

Équipe PEQUAN au LIP6.

Équipe TOCCATA (Inria Saclay, preuve formelle).

Équipe DALI (LIRMM, Perpignan).

Équipe Caramba (Inria Nancy).

Take Home Message

- opérations de base $(+, -, \times, \div, \sqrt{\cdot})$: arrondi correct, un seul résultat possible, portabilité et reproductibilité
- fonctions mathématiques : cela dépend de la libm utilisée, de sa version et même du hardware ! Ne pas faire une confiance aveugle aux libms !
- dans tous les cas, vérifier qu'il n'y a aucun NaN, ni débordement (underflow, overflow)
- pour obtenir plus de garantie sur la précision des calculs : utiliser une plus grande précision, ou bien de l'arithmétique d'intervalles, ou de la précision arbitraire (GNU MPFR).

Références

What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg, Computing Surveys, 1991.

An Interview with the Old Man of Floating-Point, 1998, https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html

The pitfalls of verifying floating-point computations, David Monniaux, ACM TOPLAS, 2008.

Accuracy of Mathematical Functions in Single, Double, Double Extended, and Quadruple Precision, Vincenzo Innocente and Paul Zimmermann, 2021.

The CORE-MATH Project, https://core-math.gitlabpages.inria.fr/.