Notation en virgule flottante

démystification

Pascal Romon

Introduction

En programmation, on utilise :

- des **int** (*integer*)
- des **unsigned int**
- des **float** (*flottants* $\neq$ *réels*)
- des **double**
- ...

$\rightarrow$ Comment sont-ils codés dans la machine?
$\rightarrow$ Quelles précautions à prendre pour les utiliser?

Rappels sur les entiers

Écriture binaire (base 2):

$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$	(base 2)
$\mathbf{1} \times 2^3$	$+ \mathbf{0} \times 2^2$	$+ \mathbf{1} \times 2^1$	$+ \mathbf{1} \times 2^0$	(base 10)
$\mathbf{1} \times 8$	$+ \mathbf{0} \times 4$	$+ \mathbf{1} \times 2$	$+ \mathbf{1} \times 1$	(base 10)
8	$+ 0$	$+ 2$	$+ 1$	(base 10)
		$\rightarrow 11$		(base 10)

Rappels sur les entiers

En base 2 :

$x \approx a_m \dots a_2 a_1 a_0$

avec $a_i = 0$ ou 1

Conversion en base 10 :

$$x = a_m 2^m + \dots + a_2 2^2 + a_1 2^1 + a_0 2^0$$

$$x = \sum_{i=0}^{m} a_i 2^i \qquad \text{(avec } a_i = 0 \text{ ou } 1\text{)}$$

Rappels sur les entiers

En base b :

$$x \approx \underbrace{a_m \dots a_2 a_1 a_0}_{\text{en base } b} \quad \text{avec } 0 \leq a_i \leq b-1$$

Conversion en base 10 :

$$x = \underbrace{\sum_{i=0}^{m} a_i b^i}_{a_i \text{ et } b^i \text{ exprimés en base } 10}$$

Rappels sur les entiers

Addition binaire :

$$\begin{array}{rcccc} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ + & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline = & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array} \qquad \begin{array}{rccccc} & & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ + & & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ + & & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \hline = & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}$$

exactement comme en base 10 *!*

Rappels sur les entiers

Opérations binaires :

- Addition
- Soustraction (différent du '$-$' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $\ll$, $\gg$, $\&$ et $|$ (spécifiques au binaire)
- $\&\&$ et $||$ (logique)

Les entiers en C/C++

Les `unsigned int` :

4 octets $=$ 32 bits
de `00000000`$\cdots$`00000000` à `11111111`$\cdots$`11111111`
$\rightarrow$ de 0 à $2^{32}-1 = 4,294,967,295$ (notation virgule anglo-saxonne)

Les entiers en C/C++

Les `int` :

4 octets $=$ 32 bits
1 bit de signe pour les nombres négatifs : $x = s.m$

- $s = 0/1$: bit de signe
- m : nombre binaire codé sur 31 bits (*magnitude*)

Les entiers en C/C++

Les `int` :
En pratique : $x = m - s \times 2^{31}$

- si $s = 0$ $\quad 0 \leq x \leq 2^{31} - 1$
- si $s = 1$ $\quad -2^{31} \leq m - 2^{31} \leq -1$

$\rightarrow$ évite de coder 0 deux fois ($+0$ et -0)

$$\Rightarrow -2147483648 \leq x \leq 2147483647$$

soit environ 4 milliards d'entiers
(même amplitude que `unsigned int`, décalée de -2147483648).

Référence: en.wikipedia.org/wiki/Signed_number_representations

Exemple

Exercice :
Écrire en binaire (8 bits) les nombres décimaux $+2, +1, 0, -1, -2$.
Réponse :

$$0\,0000010 \quad 0\,0000001 \quad 0\,0000000 \quad 1\,1111111 \quad 1\,1111110$$

En effet, $-1 = m - 2^7$ donc

$$m = 2^7 - 1 = 10000000 - 00000001 = 1111111$$

Exercice : comment calculer l'opposé d'un entier signé ?
Réponse : opération $\overline{\text{not}}$ suivie de l'addition de 1.
En effet, si $\bar{m} = \text{not}(m)$ est la magnitude complémentaire,

$$m + \bar{m} = 1111111 = 2^7 - 1$$

Nombre à Virgule Flottante

Introduction :

$$
\begin{array}{lcllll}
(1.25)_{10} & = & 1 \times 10^0 & + 2 \times 10^{-1} & + 5 \times 10^{-2} & \\
 & = & 1 & + 0 & + \frac{1}{4} & \\
 & = & 1 \times 2^0 & + 0 \times \underbrace{2^{-1}}_{\frac{1}{2}} & + 1 \times \underbrace{2^{-2}}_{\frac{1}{4}} & = \quad (1.01)_2
\end{array}
$$

Nombre à Virgule Flottante

Définition :
Le réel x est écrit en virgule flottante de base b à m chiffres si :

- $b \geq 2$
- on peut écrire x sous la forme :

$$x \approx \pm 0.\underbrace{a_1 a_2 \dots a_m}_{\text{mantisse}} b^M \rightarrow \text{exposant}$$

$$x = \pm \sum_{i=1}^{m} a_i b^{M-i} \text{ avec } a_i \in \mathbb{N} \text{ et } 0 \leq a_i < b$$

- $a_1 \neq 0$ pour une écriture normalisée
- $M_{\min} \leq M \leq M_{\max}$ fixe l'amplitude possible

Nombre à Virgule Flottante

$$x \approx \pm 0.a_1 a_2 \dots a_m b^M$$

avec $M_{\min} = -5$ et $M_{\max} = 6$.

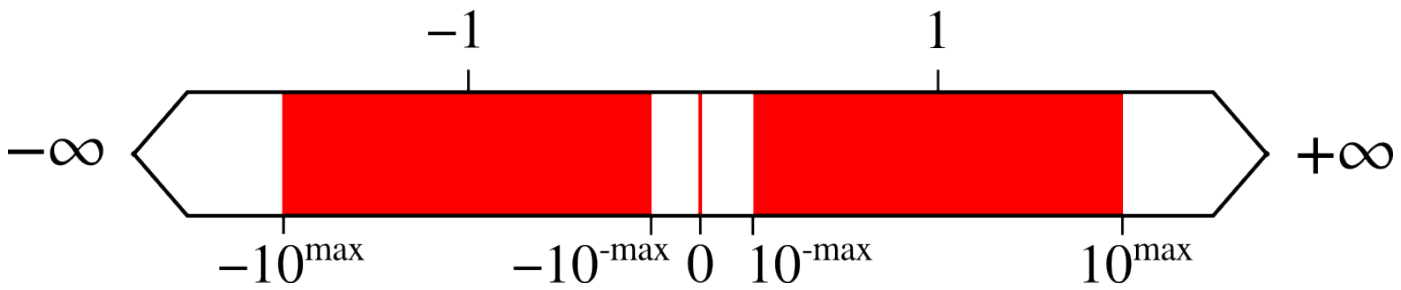
Exemple :

$$x \approx 0.00064$$
$$b = 10$$
$$m = 2$$

$\rightarrow x \approx 0.64 \times 10^{-3}$

Nombre à Virgule Flottante

Limites de cette représentation :



- nombres exprimables

Nombre à Virgule Flottante

Propriété : (version idéale en base b)
Nombre de réels représentables exactement en virgule flottante.

$$x = \pm 0, a_1 a_2 \dots a_m b^M$$

- $\pm \rightarrow 2$
- $a_1 \rightarrow (b-1)$ possibilités car $a_1 \neq 0$
- $a_2 \dots a_m \rightarrow b^{m-1}$ possibilités
- $M \rightarrow$ nombre d'exposants possibles $= M_{\max} - M_{\min} + 1$
- pour $0 \rightarrow +1$ (remarque : $+2$ si on différencie $+0$ de -0)

$$\longrightarrow \text{en tout : } 2(b-1)b^{m-1}(M_{\max} - M_{\min} + 1) + 1$$

Norme IEEE 754

En pratique : `float` norme IEEE 754 (C et C++)

- 4 octets (32 bits)
- $b = 2$, $m = 23$ ($+$ 1 bit de signe)
- M est codé sur 8 bits sans bit de signe (256 valeurs), auquel on retire 127. $\rightarrow M_{\max} = 128$ et $M_{\min} = -127$ (en réalité -127 et 128 sont *réservés*)
- a_1 fantôme (pas codé car supposé égal à 1)

$$x = \pm\left(\underbrace{b^{M-127}}_{\text{pour } a_1} + \sum_{i=2}^{m} a_i b^{M-(i-1)-127}\right)$$

Nombre de réels codables exactement en C/C++

$\simeq$ 2 milliards de réels différents

Norme IEEE 754

En pratique : norme IEEE 754 (C et C++)

On code aussi :

- $+\infty$, $-\infty$
- NaN (Not a Number) exemple: `float a = 8.0/0.0;`
- zéros signés : $+0 \neq -0$

Norme IEEE 754

En pratique : `float` norme IEEE 754 (C et C++)

signe s	exposant M	mantisse m	valeur
0/1	$1 \rightarrow 254$	any	$(-1)^s \times 2^{M-127} \times (1+m)$
0	0	0	$+0$
1	0	0	-0
0	255	0	$+\inf$
1	255	0	$-\inf$
0/1	255	$\neq 0$	NaN

$a_1 = 1$ est un bit fantôme non codé
en représentation normalisée

Norme IEEE 754

$$x = (-1)^s \times 2^{M-127} \times (1+m)$$

Exemples :

$$x = \underbrace{0}_{\pm} \ \underbrace{01111111}_{127} 00000000000000000000000$$

exposant associé au a_1 fantôme : $2^{127-127} = 2^0$

$$\begin{array}{rccccccc} & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & \dots & & \\ x = + & (1) & 0 & 0 & 0 & \cdots & = & 1 \end{array}$$

Attention : le vrai exposant est en fait $M - 126$.

Norme IEEE 754

$$x = (-1)^s \times 2^{M-127} \times (1+m)$$

Exemples :

$$x = \underbrace{0}_{\pm} \ \underbrace{10000000}_{128} 11000000000000000000000$$

exposant associé au a_1 fantôme : $2^{128-127} = 2^1$

$$\begin{array}{rccccccc} & 2^1 & 2^0 & 2^{-1} & 2^{-2} & \dots & & \\ x = + & (1) & 1 & 1 & 0 & \cdots & = & 3.5 \end{array}$$

Norme IEEE 754

Exemples : `float` en norme IEEE 754

signe (1)	exposant (8)	mantisse (23)	
0	00000000	00000000000000000000000	$\rightarrow +0$
1	00000000	00000000000000000000000	$\rightarrow -0$
0	01111111	00000000000000000000000	$\rightarrow 1$
0	10000000	00000000000000000000000	$\rightarrow 2$
0	10000001	00000000000000000000000	$\rightarrow 4$
0	10000001	01000000000000000000000	$\rightarrow 5$
0	10000000	11000000000000000000000	$\rightarrow 3.5$
0	10000001	11000000000000000000000	$\rightarrow 7$
0	10000010	11000000000000000000000	$\rightarrow 14$
0	11111111	10000000000000000000000	$\rightarrow$ NaN
0	11111111	00000000000000000000000	$\rightarrow \infty$

Arrondi et troncature

Valeur tronquée à p décimales :

$$x = \pm 0.a_1 a_2 \dots a_p \mid a_{p+1} \dots a_m.b^M$$

Valeur arrondie (au plus près) à p décimales :

$$x = \pm 0.a_1 a_2 \dots a_{p-1} a_p \dots a_m.b^M$$

$$x' = \pm 0.a_1 a_2 \dots a_{p-1} r_p.b^M$$

avec :

$$\begin{cases} r_p = a_p + 1 & \text{si } a_{p+1} \geq \dfrac{b}{2} \\ r_p = a_p & \text{si } a_{p+1} < \dfrac{b}{2} \end{cases}$$

en base 2 : $r_p = a_p + a_{p+1}$ (attention aux retenues !)

Arrondi et troncature

Exemple :
représenter $x = 0.306 \times 10^2$

$b = 10$

$m = 2$

$M_{\min} = -5$

$M_{\max} = 6$

$\rightarrow$ Troncature : $x = 0,30 \times 10^2$
$\rightarrow$ Arrondi : $x = 0.31 \times 10^2$

Remarque : avec un arrondi, on perd 2 fois moins d'incertitude.

Opérations sur les float

Opérations standard :

- $+$, $-$, $\times$ et $\div$
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$

Opérations plus évoluées :

- $\sqrt{}$, $\exp$, $\log\,(=\ln)$, $\log 10$, $\ldots$
- $\cos$, $\sin$, $\ldots$

Opérations sur les float

Addition et soustraction :

$$x = 0.x_1 x_2 \dots x_m \times 10^{M_x}$$
$$y = 0.y_1 y_2 \dots y_m \times 10^{M_y}$$

$$\hookrightarrow x + y \text{ ou } x - y$$

1. $M = \max(M_x, M_y)$
2. on réécrit x et y en b^M
3. on fait l'opération
4. on réajuste M si nécessaire
5. on arrondit

Opérations sur les float

Addition et soustraction : exemple

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = +0.321 \times 10^2$
$y = +0.440 \times 10^3$

$$x + y = 0.321 \times 10^2 + 0.440 \times 10^3$$

$M = \max(M_x, M_y)$	$\rightarrow M = 3$
on réécrit x et y en b^M	$\rightarrow x + y = (0.0321 + 0.440) \times 10^3$
on fait l'opération	$\rightarrow x + y = 0.4721 \times 10^3$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x + y = 0.472 \times 10^3$

Opérations sur les float

Addition et soustraction : exemple

$b = 10$, $m = 3$, $M_{min} = -5$ et $M_{\max} = 6$
$x = +0.321 \times 10^{2}$
$y = +0.440 \times 10^{-3}$

$$x + y = 0.321 \times 10^{2} + 0.440 \times 10^{-3}$$

$M = \max(M_x, M_y)$	$\rightarrow M = 2$
on réécrit x et y en b^M	$\rightarrow x + y = (0.321 + 0.00000440) \times 10^{2}$
on fait l'opération	$\rightarrow x + y = 0.3210044 \times 10^{2}$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x + y = 0.321 \times 10^{2} = x$

Note : une troncature préalable accélère les calculs (voire annule si $|M_x - M_y| > m$)

Opérations sur les float

Multiplication et division :

$$x = 0.x_1 x_2 \dots x_m \times 10^{M_x}$$
$$y = 0.y_1 y_2 \dots y_m \times 10^{M_y}$$

$$\hookrightarrow x \times y \text{ ou } x \div y$$

- ~~$M = \max(M_x, M_y)$~~
- ~~on réécrit x et y en b^M~~

1. on fait l'opération
2. on réajuste M si nécessaire
3. on arrondit

Opérations sur les float

Multiplication et division : exemple

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = +0.321 \times 10^2$
$y = +0.542 \times 10^1$

$$x \times y = 0.321 \times 10^2 \times 0.542 \times 10^1 = (0.321 \times 0.542) \times 10^{2+1}$$

on fait l'opération	$\rightarrow x \times y = 0.173982 \times 10^3$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x \times y = 0.174 \times 10^3$

Opérations sur les float

Multiplication et division : exemple

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = +0.321 \times 10^2$
$y = +0.142 \times 10^1$

$$x \times y = 0.321 \times 10^2 \times 0.142 \times 10^1 = (0.321 \times 0.142) \times 10^{2+1}$$

on fait l'opération	$\rightarrow x \times y = 0.045582 \times 10^3$
on réajuste M si nécessaire	$\rightarrow M = 2$
on arrondit	$\rightarrow x \times y = 0.456 \times 10^2$

Opérations sur les float

Multiplication et division : exemple

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = +0.321 \times 10^5$
$y = +0.542 \times 10^4$

$$x \times y = 0.321 \times 10^5 \times 0.542 \times 10^4 = (0.321 \times 0.542) \times 10^{5+4}$$

on fait l'opération	$\rightarrow x \times y = 0.173982 \times 10^9$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x \times y = 0.174 \times 10^9 \rightarrow$ dépassement

Erreur numérique

Précision de la machine :
La précision ϵ_m de la machine est le plus petit float (en magnitude) qui, additionné à 1.0 donne un float différent de 1.0.

En norme IEEE 754 :

`std::numeric_limits<float>::epsilon()`	$1,19 \times 10^{-7} = 2^{-23}$
`std::numeric_limits<float>::min()`	$1,18 \times 10^{-38} = 2^{-126}$
`std::numeric_limits<float>::max()`	$3,40 \times 10^{38} = 2 \times 2^{127}$
`std::numeric_limits<double>::epsilon()`	$2,22 \times 10^{-16}$
`std::numeric_limits<double>::min()`	$2,23 \times 10^{-308}$
`std::numeric_limits<double>::max()`	$1,80 \times 10^{308}$

Erreur numérique : 2 sources d'erreurs

Erreur de troncature : (*truncation error*)
Erreur générée par les arrondis après une opération binaire.

Erreur de précision : (*rounding error*)
Erreur générée par l'incapacité de représenter parfaitement certain nombres réels avec un `float` (ex: $\pi, \sqrt{2}, \frac{1}{3}$)

Erreur d'incertitude : (*uncertainty*)
Erreur de mesure ou d'estimation.

Erreur numérique

Remarque :
Il arrive qu'il y ait une perte de chiffres significatifs dans la soustraction de deux nombres voisins du fait des arrondis.

La soustraction est une des opérations les plus **dangereuses** en calcul numérique : elle peut amplifier l'erreur relative de façon catastrophique.

Exemple :

- addition : $(x + \epsilon_1) + (x + \epsilon_2) = 2x + err(\epsilon_1 + \epsilon_2)$
  $\hookrightarrow$ l'erreur est petite par rapport à $2x$
- soustraction : $(x + \epsilon_1) - (x + \epsilon_2) = 0 + err(\epsilon_1 - \epsilon_2)$
  $\hookrightarrow$ l'erreur est grande par rapport à 0

Conséquence des arrondis

Associativité : $(x + y) + z = x + (y + z)$

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = y = 0.400 \times 10^0$
$z = 0.100 \times 10^3$

$$\begin{aligned}(x + y) + z &= (0.400 \times 10^0 + 0.400 \times 10^0) + 0.100 \times 10^3 \\ &= 0.800 \times 10^0 + 0.100 \times 10^3 \\ &= 0.1008 \times 10^3 \\ &= 0.101 \times 10^3\end{aligned}$$

Conséquence des arrondis

Associativité : $(x + y) + z = x + (y + z)$

$b = 10$, $m = 3$, $M_{\min} = -5$ et $M_{\max} = 6$
$x = y = 0.400 \times 10^0$
$z = 0.100 \times 10^3$

$$
\begin{aligned}
(x + y) + z &= 0.101 \times 10^3 \\
x + (y + z) &= 0.400 \times 10^0 + (0.400 \times 10^0 + 0.100 \times 10^3) \\
&= 0.400 \times 10^0 + (0.0004 \times 10^3 + 0.100 \times 10^3) \\
&= 0.400 \times 10^0 + 0.1004 \times 10^3 \\
&= 0.400 \times 10^0 + 0.100 \times 10^3 \\
&= 0.100 \times 10^3 = z
\end{aligned}
$$

Conséquence des arrondis

Associativité : $(x + y) + z = x + (y + z)$

Dans notre exemple, on a : $(x + y) + z \neq x + (y + z)$

Remarque :
On a le même problème avec la distributivité de la multiplication.

Conséquence des arrondis

Méthode :

On calcule mieux les opérations sur des valeurs du même ordre de grandeur.

On ajoute d'abord les plus petits éléments entre eux afin qu'ils deviennent suffisamment gros pour ne plus être négligeables devant les autres.

Autrement dit, on fait les sommes du plus petit au plus grand.

En pratique

Précautions à prendre :

- être attentif dans la formulation des équations.
- faire attention aux dépassements.
  $\rightarrow$ normer / conditionner les données
- bien ordonner les opérations.

Et en C/C++ ?

Et en pratique ? : dans du code C/C++

```
float a = 0.0;
float b = -0.0;

cout << "a = " << a;
cout << "b = " << b;
```

$\rightarrow$ a = 0 b = -0

Et en C/C++ ?

Et en pratique ? : dans du code C/C++

```
float a = 0.0;
float b = -0.0;

if(a==b) cout << "a==b";
else cout << "a != b";
```

$\rightarrow$ `a == b`

Et en C/C++ ?

Et en pratique ? : dans du code C/C++

```
float a = ...;
float b = ...;

...

if(a==b) ...
```

→ PAS OK !

Et en C/C++ ?

Et en pratique ? : dans du code C/C++

```
float a = ...;
float b = ...;

...

if(a==b) ...
```

→ `if( fabs(a-b) < epsilon)` OK !

Et en C/C++ ?

Et en pratique ?

Temps d'exécution d'opérations standards :

$+$	0.8 ns
$-$	0.8 ns
$\times$	1,3 ns
$\div$	5,8 ns

PC Intel Core i7-2600, 3.40GHz

$\rightarrow$ `float a = x / 2.0;` vs. `float a = x * 0.5;`

Et en C/C++ ?

Et en pratique ?

Quelques opérations :

```
#include<limits>
float inf = std::numeric_limits<float>::infinity();
```

```
std::cout << inf - inf;    →  -Nan
std::cout << inf + 42;     →  inf
std::cout << 1.0/inf;      →  0
std::cout << sqrt(inf);    →  inf
std::cout << sqrt(-1.0);   →  -NaN
```

Et en C/C++ ?

Et en mode warrior ?

Quake III Arena

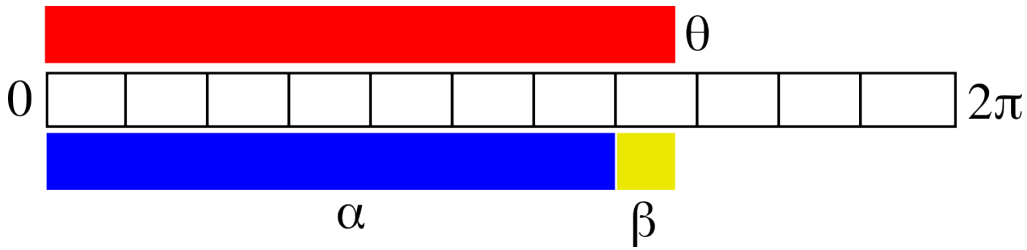
```
float FastInvSqrt(float x) {
  float xhalf = 0.5f * x;
  int i = *(int*)&x; // evil floating point bit level hack
  i = 0x5f3759df - (i >> 1); // what the fuck?
  x = *(float*)&i;
  x = x*(1.5f-(xhalf*x*x));
  return x;
}
```

Pour calculer une excellente approximation de $\frac{1}{\sqrt{x}}$

$\rightarrow$ normer un vecteur par exemple

sinus et cosinus rapides

- on utilise des *Look Up Tables* (LUT) : $\sin$ et $\cos$ avec n cases sur $[0, 2\pi[$
- $\theta = \alpha + \beta$ où $\alpha = \frac{2\pi}{n} \left\lfloor \frac{n|\theta|}{2\pi} \right\rfloor$ et $\beta = \theta - \alpha$



- $\cos\alpha$ et $\sin\alpha$ sont donnés exactement par les LUT
- β petit $\Rightarrow$ développements limités :

$$\sin\beta = \beta - \frac{\beta^3}{3!} + \frac{\beta^5}{5!} \qquad \cos\beta = 1 - \frac{\beta^2}{2!} + \frac{\beta^4}{4!}$$

$$\sin\theta = \sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$

$$\cos\theta = \cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$$

$\rightarrow$ légèrement moins précis, mais même temps de calcul que $\sin$ ou $\cos$

Optimisation du compilateur

Code :

```
float v[1] = {498.255};
float b = 498.255;

v[0] = v[0] / b;

std::cout<< std::setprecision(20) << v[0];
```

Résultats :

- avec `g++ -O2 -ffast-math main.cpp` → 0.99999994
- avec `g++ main.cpp` → 1

`-ffast-math` inclue `-freciprocal-math`

Complexité

Complexité et temps d'exécution :

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Growth rates of common functions measured in nanoseconds

The Algorithm Design manual, Steven Skiena