Notation en virgule flottante

démystification

Vincent Nozick

IMAC INGÉNIEUR ESIPE

Université Gustave Eiffel

Introduction

En programmation, on utilise :

- des **int**
- des **unsigned int**
- des **float**
- des **double**
- ...

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

Rappels sur les entiers

Écriture binaire (base 2):

$$\mathbf{1}\quad\mathbf{0}\quad\mathbf{1}\quad\mathbf{1}\quad\text{(base 2)}$$

Rappels sur les entiers

Écriture binaire (base 2):

$$\begin{array}{llll} \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \quad \text{(base 2)} \\ \mathbf{1}\times 2^3 & +\,\mathbf{0}\times 2^2 & +\,\mathbf{1}\times 2^1 & +\,\mathbf{1}\times 2^0 \quad \text{(base 10)} \\ \mathbf{1}\times 8 & +\,\mathbf{0}\times 4 & +\,\mathbf{1}\times 2 & +\,\mathbf{1}\times 1 \quad \text{(base 10)} \\ 8 & +\,0 & +\,2 & +\,1 \quad \text{(base 10)} \\ & & \to 11 & \quad \text{(base 10)} \end{array}$$

Rappels sur les entiers

Expression en base b :

$x = a_m b^m + \ldots + a_2 b^2 + a_1 b + a_0$

avec $a_i \in [0, b[$

Ecriture en base 2 :

$x = a_m 2^m + \ldots + a_2 2^2 + a_1 2^1 + a_0 2^0$ avec $a_i = 0$ ou 1

Conversion en base 10 :

$x = \sum_{i=0}^{m} (a_i 2^i)_{10}$ (avec $a_i = 0$ ou 1)

Rappels sur les entiers

En base b :

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

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 :

$$\begin{array}{cccccc} & \mathbf{1} & {}^{1}\mathbf{0} & \mathbf{1} & \mathbf{1} \\ + & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline = & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array}$$

Rappels sur les entiers

Opérateurs binaires :

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |
- && et ||
- ...

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |
- && et ||
- ...

Opérateurs unaires :

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |
- && et ||
- ...

Opérateurs unaires :

- '−' unaire

Rappels sur les entiers Virgule Flottante C et C++ Arrondi Opérations Erreur numérique En pratique

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |
- && et ||
- ...

Opérateurs unaires :

- '−' unaire
- `abs`

Rappels sur les entiers

Opérateurs binaires :

- Addition
- Soustraction (différent du '−' unaire)
- Multiplication
- Division entière
- Modulo (reste de la division entière)
- pow, pgcd, ...
- $<$, $\leq$, $\geq$, $>$, $==$ et $\neq$
- $<<$, $>>$, & et |
- && et ||
- ...

Opérateurs unaires :

- '−' unaire
- `abs`
- `round`, `floor`, `ceil`

Les entiers en C/C++

Les `unsigned int` :

4 octets $=$ 32 bits
de $00000000 \cdots 00000000$ à $11111111 \cdots 11111111$
$\rightarrow$ de 0 à $2^{32} - 1 = 4,294,967,295$

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

Nombres à Virgule Flottante

Introduction :

$$\begin{aligned}(1,25)_{10} &= 1\times 10^0 + 2\times 10^{-1} + 5\times 10^{-2} \\ &= 1\times 2^0 + 0\times \underbrace{2^{-1}}_{\frac{1}{2}} + 1\times \underbrace{2^{-2}}_{\frac{1}{4}} \quad = \quad (1,01)_2\end{aligned}$$

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 :

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

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

- $a_1 \neq 0$ pour une écriture normalisée
- $M_{min} \leq M \leq M_{max}$

Nombre à Virgule Flottante

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

Exemple :

$x = 0,00064$
$b = 10$
$m = 3$
$M_{min} = -5$
$M_{max} = 6$

Nombre à Virgule Flottante

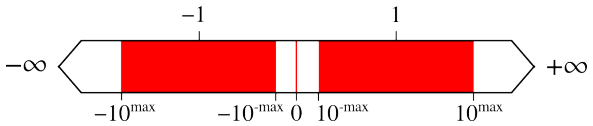
$$x = \pm 0, a_1 a_2 \dots a_m b^M$$

Exemple :

$x = 0,00064$
$b = 10$
$m = 3$
$M_{min} = -5$
$M_{max} = 6$
$\rightarrow x = 0,640.10^{-3}$

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 pour $a_1 \neq 0$
- $a_2 \dots a_m \rightarrow b^{m-1}$ 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 $M_{min} = -127$
- a_1 fantôme (pas codé car supposé égal à 1)

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

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 : $+0$ et -0 exprimés différemment.

Norme IEEE 754

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

signe s	exposant M	mantisse m	valeur
0/1	$1 \to 254$	any	$(-1)^s \times 2^{M-127} \times 1m$
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

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

Exemples :

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

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

	2^0	2^{-1}	2^{-2}	2^{-3}	$\dots$	
$x = +$	(1)	0	0	0	$\cdots$	$= 1$

Norme IEEE 754

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

Exemples :

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

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

	2^1	2^0	2^{-1}	2^{-2}	$\dots$	
$x = +$	(1)	1	1	0	$\cdots$	$= 3.5$

Norme IEEE 754

Exemples : `float` en norme IEEE 754

signe (1) | exposant (8) | mantisse (23)

$0\ 00000000\ 00000000000000000000000 \to +0$
$1\ 00000000\ 00000000000000000000000 \to -0$
$0\ 01111111\ 00000000000000000000000 \to 1$
$0\ 10000000\ 00000000000000000000000 \to 2$
$0\ 10000001\ 00000000000000000000000 \to 4$
$0\ 10000001\ 01000000000000000000000 \to 5$
$0\ 10000000\ 11000000000000000000000 \to 3,5$
$0\ 10000001\ 11000000000000000000000 \to 7$
$0\ 10000010\ 11000000000000000000000 \to 14$
$0\ 11111111\ 10000000000000000000000 \to$ NaN
$0\ 11111111\ 00000000000000000000000 \to \infty$

Nombres à virgule flottante en C/C++

Précision :

- `float` : 32 bits
- `double` : 64 bits
- `long double` : 128 bits

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

en base 2 : $r_p = a_p + a_{p+1}$

Arrondi et troncature

Exemple :
représenter $x = 0,306.10^2$

$b = 10$
$m = 2$
$M_{min} = -5$
$M_{max} = 6$

Arrondi et troncature

Exemple :
représenter $x = 0,306.10^2$

$b = 10$
$m = 2$
$M_{min} = -5$
$M_{max} = 6$

$\rightarrow$ Troncature : $x = 0,30.10^2$
$\rightarrow$ Arrondi : $x = 0,31.10^2$

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, ...
- cos, sin, ...
- ...

Opérations sur les float

Addition et soustraction :

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

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

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 $M_{max} = 6$
$x = +0,321.10^2$
$y = +0,440.10^3$

$$x + y = 0,321.10^2 + 0,440.10^3$$

$M = max(M_x, M_y)$	$\rightarrow M = 3$
on réécrit x et y en b^M	$\rightarrow x + y = (0,0321 + 0,440).10^3$
on fait l'opération	$\rightarrow x + y = 0,4721.10^3$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x + y = 0,472.10^3$

Opérations sur les float

Addition et soustraction : exemple

$b = 10$, $m = 3$, $M_{min} = -5$ et $M_{max} = 6$
$x = +0,321.10^2$
$y = +0,440.10^{-3}$

$$x + y = 0,321.10^2 + 0,440.10^{-3}$$

$M = max(M_x, M_y)$	$\rightarrow M = 2$
on réécrit x et y en b^M	$\rightarrow x + y = (0,321 + 0,00000440).10^2$
on fait l'opération	$\rightarrow x + y = 0,3210044.10^2$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x + y = 0,321.10^2 = x$

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()`	$1,19 \times 10^{-7}$
`std::numeric_limits<float>::min()`	$1,18 \times 10^{-38}$
`std::numeric_limits<float>::max()`	$3,40 \times 10^{38}$
`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()`	$1,80 \times 10^{308}$

Opérations sur les float

Multiplication et division :

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

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

- ~~$M = max(M_x, M_y)$~~
- ~~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 $M_{max} = 6$
$x = +0,321.10^2$
$y = +0,542.10^1$

$$x \times y = 0,321.10^2 \times 0,542.10^1 = (0,321 \times 0,542).10^{2+1}$$

on fait l'opération	$\rightarrow x \times y = 0,173982.10^3$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x \times y = 0,174.10^3$

Opérations sur les float

Multiplication et division : exemple

$b = 10$, $m = 3$, $M_{min} = -5$ et $M_{max} = 6$
$x = +0,321.10^5$
$y = +0,542.10^4$

$$x \times y = 0,321.10^5 \times 0,542.10^4 = (0,321 \times 0,542).10^{5+4}$$

on fait l'opération	$\rightarrow x \times y = 0,173982.10^9$
on réajuste M si nécessaire	$\rightarrow$ pas besoin
on arrondit	$\rightarrow x \times y = 0,174.10^9 \rightarrow$ dépassement

Erreur numérique : 3 sources d'erreurs

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

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 $M_{max} = 6$
$x = y = 0,400.10^0$
$z = 0,100.10^3$

$$\begin{aligned}(x + y) + z &= (0,400.10^0 + 0,400.10^0) + 0,100.10^3 \\ &= 0,800.10^0 + 0,100.10^3 \\ &= 0,1008.10^3 \\ &= 0,101.10^3\end{aligned}$$

Conséquence des arrondis

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

$b = 10$, $m = 3$, $M_{min} = -5$ et $M_{max} = 6$
$x = y = 0,400.10^0$
$z = 0,100.10^3$

$$\begin{aligned}(x + y) + z &= 0,101.10^3 \\ x + (y + z) &= 0,400.10^0 + (0,400.10^0 + 0,100.10^3) \\ &= 0,400.10^0 + (0,0004.10^3 + 0,100.10^3) \\ &= 0,400.10^0 + 0,1004.10^3 \\ &= 0,400.10^0 + 0,100.10^3 \\ &= 0,100.10^3 = z\end{aligned}$$

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.
  → 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;
```

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;
```

```
→ 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";
```


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";
```

```
→ a == b
```


Et en C/C++ ?

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

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

if(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 ? : dans du code C/C++

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

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

→ `if( fabs(a-b) < epsilon)` OK !

Alternative : `if` $\frac{|a-b|}{|a|} < \epsilon$

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

→ `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

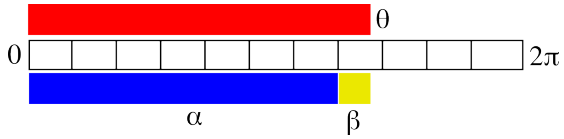
```
float FastInvSqrt(float x) {
  float xhalf = 0.5f * x;
  int i = *(int*)&x; // evil floating point bit level hack
  i = 0x5f3759df - (i >> 1); // what the fuck?
  x = *(float*)&i;
  x = x*(1.5f-(xhalf*x*x));
  return x;
}
```

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

→ normer un vecteur par exemple : $\frac{\mathbf{x}}{\sqrt{\mathbf{x}\cdot\mathbf{x}}}$

sinus et cosinus rapides (à chaque époque ses astuces)

- on utilise 2 LUT : $\sin$ et $\cos$ avec n cases sur $[0, 2\pi[$
- $\theta = \alpha + \beta$



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

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

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

→ 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`