Introduction à l'arithmétique des ordinateurs

Peut-on vraiment calculer avec un ordinateur ?

F. Langrognet

PLAN

PLAN

PLAN

Un calcul simple

vraiment simple...

$$\text{Calcul de } \sum_{i=1}^{n} x$$

Avec $n = 1000$ et

- $x = 0.5$
- $x = 0.25$
- $x = 0.1$
- $x = 0.7$

Un calcul simple

Un peu de C++

```
int main(){
        float x;
        cout<<"Entrez la valeur de x"<<endl;
        cin>>x;
        float res = 0.0;
        for (int i=0;i<1000;i++){
                res+=x;
        }
        cout<<"Somme des x (1000 fois) :"<<setprecision(10)<<res<<endl;
        return 0;
}
```

Un calcul simple

Calculons

$$\sum_{i=1}^{1000} 0.5 = 500$$

$$\sum_{i=1}^{1000} 0.25 = 250$$

$$\sum_{i=1}^{1000} 0.1 = 99.999046$$

$$\sum_{i=1}^{1000} 0.7 = 700.006958$$

Un calcul simple

Calculons

$$\sum_{i=1}^{1000} 0.5 = 500$$

$$\sum_{i=1}^{1000} 0.25 = 250$$

$$\sum_{i=1}^{1000} 0.1 = 99.999046$$

$$\sum_{i=1}^{1000} 0.7 = 700.006958$$

Un calcul simple

Calculons

$$\sum_{i=1}^{1000} 0.5 = 500$$

$$\sum_{i=1}^{1000} 0.25 = 250$$

$$\sum_{i=1}^{1000} 0.1 = 99.999046$$

$$\sum_{i=1}^{1000} 0.7 = 700.006958$$

Un calcul simple

Calculons

$$\sum_{i=1}^{1000} 0.5 = 500$$

$$\sum_{i=1}^{1000} 0.25 = 250$$

$$\sum_{i=1}^{1000} 0.1 = 99.999046$$

$$\sum_{i=1}^{1000} 0.7 = 700.006958$$

Et avec Matlab ?

Ce calcul avec Matlab

```
x = single(0.1);
res = 0.0;
for i=1:1000
        res = res+x;
end;
```

- res = 99.9990
- Pas mieux (même erreur)

Remarque

Tous les calculs (C++ et Matlab) ont été faits en simple précision

un calcul simple

Et avec des double precision ?

On remplace les float par des double et . . .

$$\sum_{i=1}^{1000} 0.1 = 99.999999999985931253$$

On repousse simplement le problème !

un calcul simple

Et avec des double precision ?

On remplace les float par des double et . . .

$$\sum_{i=1}^{1000} 0.1 = 99.99999999985931253$$

On repousse simplement le problème !

Question 1

- Pourquoi les résultats avec $x = 0.5$ et $x = 0.25$ sont-ils justes et ceux avec $x = 0.7$ et $x = 0.1$ faux ?
- 0.7 et 0.1 sont-ils plus dangereux que 0.5 et 0.25 ?

Question 2

Est-ce vraiment grave ?

Exemple concret d'une erreur numérique

- Guerre du Golfe de 1991 : un anti-missile US Patriot dont le programme tournait depuis 100 heures a raté l'interception d'un missile Irakien Scud - 28 morts
- Explication :
  - l'anti missile Patriot incrémentait un compteur toutes les 0.1 secondes
  - 0.1 approché avec erreur 0.0000000953 (codé sur 24 bits)
  - au bout de 100 heures, erreur cumulée 0.34s
  - dans ce laps de temps le Scud parcourt 500 mètres.

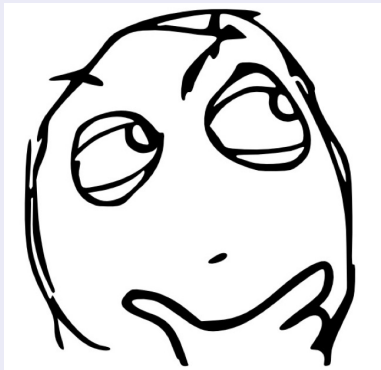
PLAN

Suite de Muller

Soit la suite de Muller définie par :

$$\begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_{n+1} = 111 - \frac{1130}{u_n} + \frac{3000}{u_n . u_{n-1}} \end{cases}$$

- u_n converge vers 6
- Sur n'importe quel système à précision finie on observera une convergence apparente, très rapide, vers 100



Fonction de Rump

Fonction de Rump

$f(a,b) = (333 + \frac{3}{4})b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + \frac{11}{2}b^8 + \frac{a}{2b}$ avec

$a = 77617.0$ et $b = 33096.0$

Résultats

- Simple précision (32 bits) : $\approx -2,34.10^{29}$
- Double précision (64 bits) : $\approx -1,1.10^{21}$
- Avec d'autres parenthésages, sur des architectures différentes
  - Simple précision : $\approx 7,09.10^{29}$ ou encore $\approx +1,17$
  - Double précision : $\approx +1,17$ (souvent)

Quelle est la bonne valeur ?

$$\approx -0.827396059946821368141165095479816291$$

Fonction de Rump

Fonction de Rump

$f(a,b) = (333 + \frac{3}{4})b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + \frac{11}{2}b^8 + \frac{a}{2b}$ avec

$a = 77617.0$ et $b = 33096.0$

Résultats

- Simple précision (32 bits) : $\approx -2,34.10^{29}$
- Double précision (64 bits) : $\approx -1,1.10^{21}$
- Avec d'autres parenthésages, sur des architectures différentes
  - Simple précision : $\approx 7,09.10^{29}$ ou encore $\approx +1,17$
  - Double précision : $\approx +1,17$ (souvent)

Quelle est la bonne valeur ?

$\approx -0.827396059946821368141165095479816291$

Fonction de Rump

Fonction de Rump

$f(a,b) = (333 + \frac{3}{4})b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + \frac{11}{2}b^8 + \frac{a}{2b}$ avec

$a = 77617.0$ et $b = 33096.0$

Résultats

- Simple précision (32 bits) : $\approx -2,34.10^{29}$
- Double précision (64 bits) : $\approx -1,1.10^{21}$
- Avec d'autres parenthésages, sur des architectures différentes
  - Simple précision : $\approx 7,09.10^{29}$ ou encore $\approx +1,17$
  - Double précision : $\approx +1,17$ (souvent)

Quelle est la bonne valeur ?

$\approx -0.827396059946821368141165095479816291$

Fonction de Rump

Fonction de Rump

$f(a,b) = (333 + \frac{3}{4})b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + \frac{11}{2}b^8 + \frac{a}{2b}$ avec

$a = 77617.0$ et $b = 33096.0$

Résultats

- Simple précision (32 bits) : $\approx -2,34.10^{29}$
- Double précision (64 bits) : $\approx -1,1.10^{21}$
- Avec d'autres parenthésages, sur des architectures différentes
  - Simple précision : $\approx 7,09.10^{29}$ ou encore $\approx +1,17$
  - Double précision : $\approx +1,17$ (souvent)

Quelle est la bonne valeur ?

$\approx -0.827396059946821368141165095479816291$

Fonction de Rump

Fonction de Rump

$f(a,b) = (333 + \frac{3}{4})b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + \frac{11}{2}b^8 + \frac{a}{2b}$ avec

$a = 77617.0$ et $b = 33096.0$

Résultats

- Simple précision (32 bits) : $\approx -2,34.10^{29}$
- Double précision (64 bits) : $\approx -1,1.10^{21}$
- Avec d'autres parenthésages, sur des architectures différentes
  - Simple précision : $\approx 7,09.10^{29}$ ou encore $\approx +1,17$
  - Double précision : $\approx +1,17$ (souvent)

Quelle est la bonne valeur ?

$$\approx -0.827396059946821368141165095479816291$$

PLAN

Codes vraiment équivalents ?

$$f(x) = x^2 + \frac{1}{10} sin(x) \qquad f(x) = x * x + 0.1 sin(x)$$

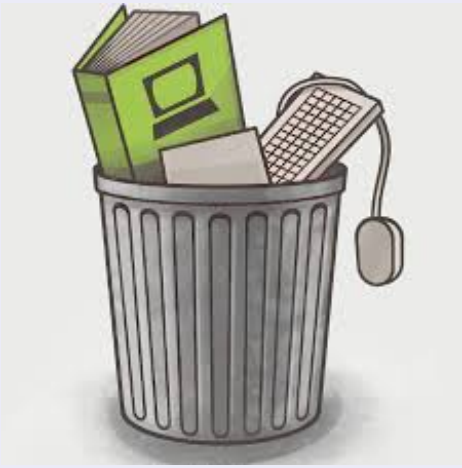
$$g(n) = \sum_{i=1}^{n} \frac{1}{i} \qquad g(n) = \sum_{i=n}^{1} \frac{1}{i}$$

donnent elles toujours les mêmes résultats ?

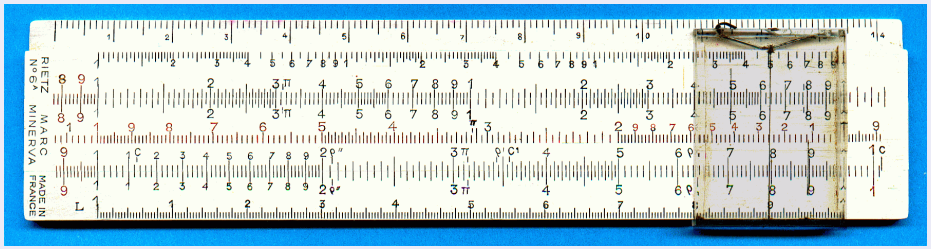
2 codes mathématiquement équivalents peuvent mener à des résultats différents

Question 3

Faut-il jeter son ordinateur à la poubelle ...



... et utiliser la règle à calculs ?



PLAN

PLAN

Rappels sur les bases

Généralités

- En base b, un nombre réel d de $m+n+1$ chiffres (digits) s'écrit $d_m d_{m-1} \ldots d_1 d_0 d_{-1} \ldots d_{-n}$
- Sa valeur se calcule ainsi : $d = \sum_{i=-n}^{m} b^i . d_i$

Base 10

Par exemple 12.34_{10} représente le nombre

$$\sum_{i=-2}^{1} 10^i . d_i = 1.10^1 + 2.10^0 + 3.10^{-1} + 4.10^{-2}$$

$$= (1.10^3 + 2.10^2 + 3.10^1 + 4).10^{-2} \text{ soit } \frac{1234}{100}$$

Base 2

Par exemple 101.11_2 représente le nombre

$$\sum_{i=-2}^{2} 2^i . d_i = 1.2^2 + 0.2^1 + 1.2^0 + 1.2^{-1} + 1.2^{-2}$$

$$= (1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0).2^{-2} \text{ soit } \frac{23}{4}$$

Conséquences

Contraintes

- En base b, on ne peut représenter exactement que les nombres fractionnaires de la forme $\frac{X}{b^k}$ où X et k sont des entiers
- En base 10 : $\frac{X}{10^k}$
- En base 2 : $\frac{X}{2^k}$

Exemples :

- En base 10, $\frac{1}{3}$ ne peut être représenté : $0.[33]_{10}$
- En base 2, $\frac{1}{10}$ ne peut être représenté : $0.000[1100]_2$

PLAN

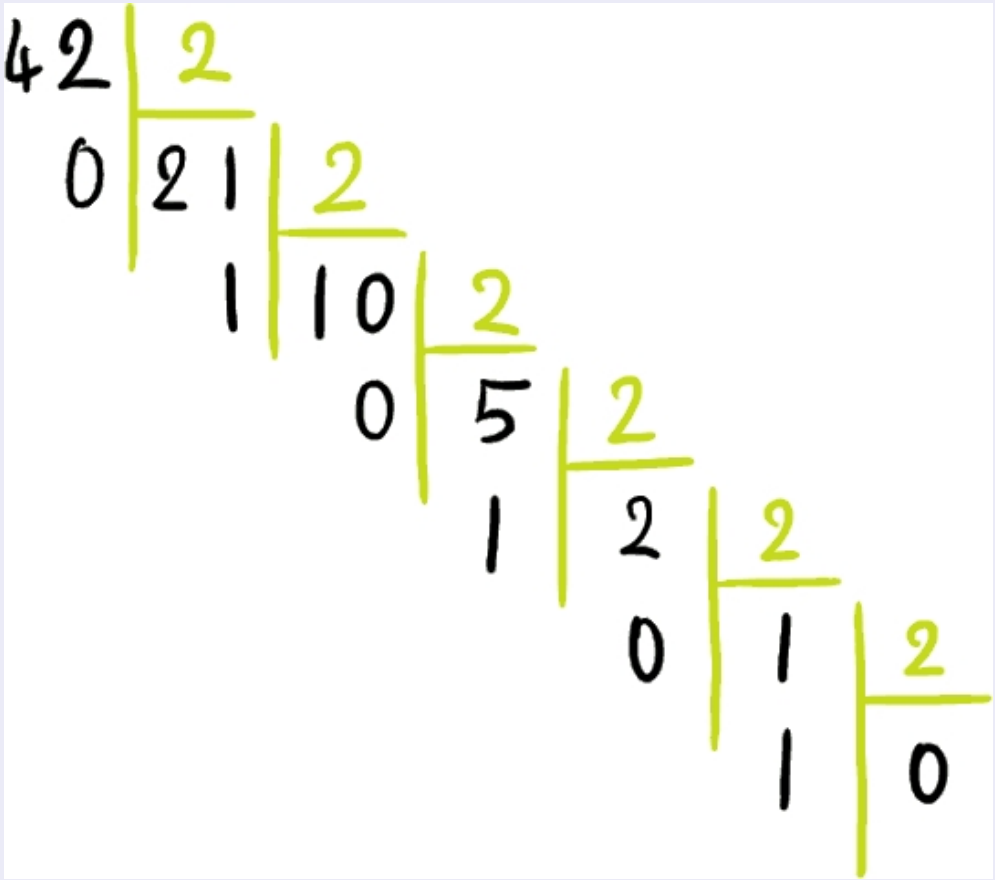
De la base 10 à la base 2

Un entier

- $9_{10} = 1001_2$ car $9 = 8 + 1 = 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0$
- $176_{10} = 10110000_2$ car $176 = 128 + 32 + 16 = 1.2^7 + 1.2^5 + 1.2^4$

En pratique :

$42_{10} = 101010_2$



De la base 10 à la base 2

Un décimal

Le principe est le même sauf qu'après la virgule, on divise par $\frac{1}{2}$ (ou on multiplie par 2)

- $0.375_{10} = 0.011_2$ car
  - $0.375x2 = 0.75$
  - $0.75x2 = 1.5$
  - $0.5x2 = 1.0$
- $0.1_{10} = 0.000[1100]_2$ car
  - $0.1x2 = 0.2$
  - $0.2x2 = 0.4$
  - $0.4x2 = 0.8$
  - $0.8x2 = 1.6$
  - $0.6x2 = 1.2$
  - $0.2x2 = 0.4$
  - $0.4x2 = 0.8$
  - ...
- $9.375_{10} = 1001.011_2$

PLAN

Virgule fixe, virgule flottante

Représentation des réels en virgule fixe

Lorsque le nombre de chiffres après la virgule est fixe quelque soit le réel.

Peu pratique

- Masse d'un électron $\overbrace{0,0.....9}^{28 decimales}$ grammes
- Masse du soleil : $\overbrace{20.....0}^{33 zeros}$ grammes

Virgule fixe, virgule flottante

Représentation des réels en virgule flottante

Exemple : 37.5 peut s'écrire (en base 10) :

- 37500.10^{-3}
- $0.0375.10^{3}$
- ...
- $0.375.10^{2}$ (représentation normalisée)

Représentation normalisée (base 10)

- La partie entière est systématiquement nulle
- Tous les chiffres significatifs sont à droite de la virgule
- La virgule n'a pas besoin d'être représentée
- La mantisse est inférieure à 1
- L'exposant est un entier (positif ou négatif)

Virgule flottante

Et en base 2 ?

Notation normalisée :
Exemple : $11011 = (0,11011).2^{101}$

- La partie entière est systématiquement nulle
- Tous les chiffres significatifs sont à droite de la virgule
- La virgule n'a pas besoin d'être représentée
- La mantisse est inférieure à 1
- L'exposant est un entier (positif ou négatif)

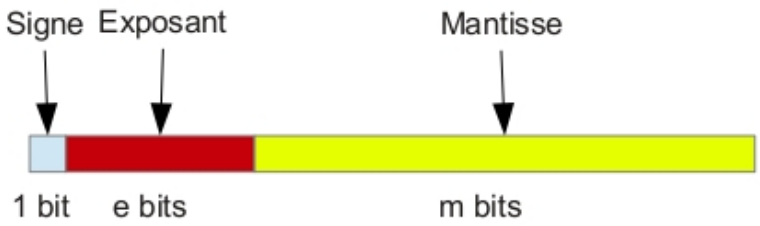
PLAN

Ecriture des nombres réels (base 2)

Mantisse, exposant, signe

Un nombre réel est représenté par un triplet :

- le signe (1 bit)
  Négatif si bit=1, positif sinon
- l'exposant *decalé* (e bits)
  Nombre entier compris entre 0 et $2^e - 1$
- la mantisse (m bits)
  Représente un nombre réel compris entre 0 et 1



Norme IEEE-754 - nombre de bits

Type	Signe	Exposant	Mantisse
float (simple précision)	1	8	23
double (double précision)	1	11	52

Exposant

Décalage ou biais

- L'exposant représente un entier compris entre $-2^{e-1}-1$ et 2^{e-1}
  Exemple : si $e = 8$, plage de valeurs : de -127 à +128 (soit 256 valeurs)
- Pour des raisons pratiques, l'exposant est décalé afin de le stocker sous forme d'un nombre non signé. Ce décalage est de $2^{e-1}-1$
- exposant décalé = exposant réel + décalage
  Exemple : si $e = 8$, plage de valeurs : de 0 à +255 (soit 256 valeurs)
  Le décalage est de 127

Mantisse

Entre 0 et 1 ou entre 1 et 2 ?

- Comme le 1er bit ne peut pas valoir 0 (par définition de la normalisation), en base 2 il vaut toujours 1
- On peut gagner 1 bit de précision en ne représentant pas le bit 1 après la virgule. Ce bit figure implicitement mais n'est pas représenté (bit implicite)
- On peut donc avoir une représentation de la mantisse ne contenant que des zéros
- ATTENTION : ce procédé n'a de sens qu'en base 2

Mantisse réelle = Mantisse stockée + 1

La mantisse réelle sera donc comprise entre 1 et 2

Calcul de la valeur

$$\text{Valeur} = s.2^{e^*}.m^*$$

- s=-1 ou 1
- e^* = exposant réel (exposant décalé - décalage)
- m^* : mantisse rélle (1 + mantisse stockée)

$$\text{Valeur} = s.2^{e-decalage}.(1+m)$$

Sauf exceptions (un peu de patience...)

Exemples (sur 32 bits)

- A = 1 10000010 00110000000000000000000
  - Le bit de signe est 1 : le nombre est négatif
  - exposant : $2^7 + 2^1 - 127 = 130 - 127 = 3$
  - mantisse : $1 + 2^{-3} + 2^{-4} = 1 + 0.125 + 0.0625 = 1.1875$

  $A = -1.1875.2^3 = -9.5$

- B = 0 01111100 01000000000000000000000
  - Le bit de signe est 0 : le nombre est positif
  - exposant : $2^6 + 2^5 + 2^4 + 2^3 + 2^2 - 127 = 124 - 127 = -3$
  - mantisse : $1 + 2^{-2} = 1.25$

  $B = 1.25.2^{-3} = 0.15625$

Retour sur la question 1

- Pourquoi le calcul de $\sum_{i=1}^{100} 0.5$ est-il juste ?
- Pourquoi le calcul de $\sum_{i=1}^{100} 0.1$ est-il faux ?

1^{ers} éléments de réponse

Tous les nombres ne sont pas représentables

Retour sur la question 1

- Pourquoi le calcul de $\sum_{i=1}^{100} 0.5$ est-il juste ?
- Pourquoi le calcul de $\sum_{i=1}^{100} 0.1$ est-il faux ?

1^{ers} éléments de réponse

Tous les nombres ne sont pas représentables

Représentation de 0.5

0.5 est parfaitement représentable

0 01111110 00000000000000000000

- Le bit de signe est 0 : le nombre est positif
- exposant : $2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 - 127 = 126 - 127 = -1$
- mantisse : $1 + 0 = 1$

Soit $= 1.2^{-1} = 0.5$

Représentation de 0.1

0.1 n'est parfaitement représentable

Valeur approchée par excès

0 01111011 10011001100110011001101

- Le bit de signe est 0 : le nombre est positif
- exposant : $2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 - 127 = 123 - 127 = -4$
- mantisse : $1 + 2^{-1} + 2^{-4} + 2^{-5} + ... = 1.600000023841858$

Soit $1.600000023841858.2^{-4} = 0.100000001490116$

Valeur approchée par défaut

0 01111011 10011001100110011001100

- mantisse : $1 + 2^{-1} + 2^{-4} + 2^{-5} + ... = 1.599999904632568 4$

Soit $1.599999904632568 4.2^{-4} = 0.099999940395355 2$

PLAN

Motivations

Le problème du zéro

Avec la formule $s.2^{e^*}.m^*$ où m^* est compris entre 1 et 2, il n'est pas possible de représenter le nombre zéro ! !
Car mantisse réelle = 1 + mantisse stockée

Stocker d'autres valeurs

- $+\infty$, $-\infty$
- Not A Number (NaN)

Nombres normalisés, dénormalisés

Nombres normalisés

Si l'exposant est compris entre 1 et $2^{e^*} - 2$, le nombre est normalisé.
Il est calculé selon la formule : $s.2^{e-decalage}.(1 + m)$

Nombres dénormalisés

Si l'exposant vaut 0 ou $2^{e^*} - 1$, le nombre est dénormalisé.
Il sert à stocker :

- le nombre zéro
- de très petites valeurs (pour plus de précisions vers 0)
- $+\infty$ et $-\infty$
- Not A Number (NaN)

Nombres normalisés, dénormalisés

Résumé

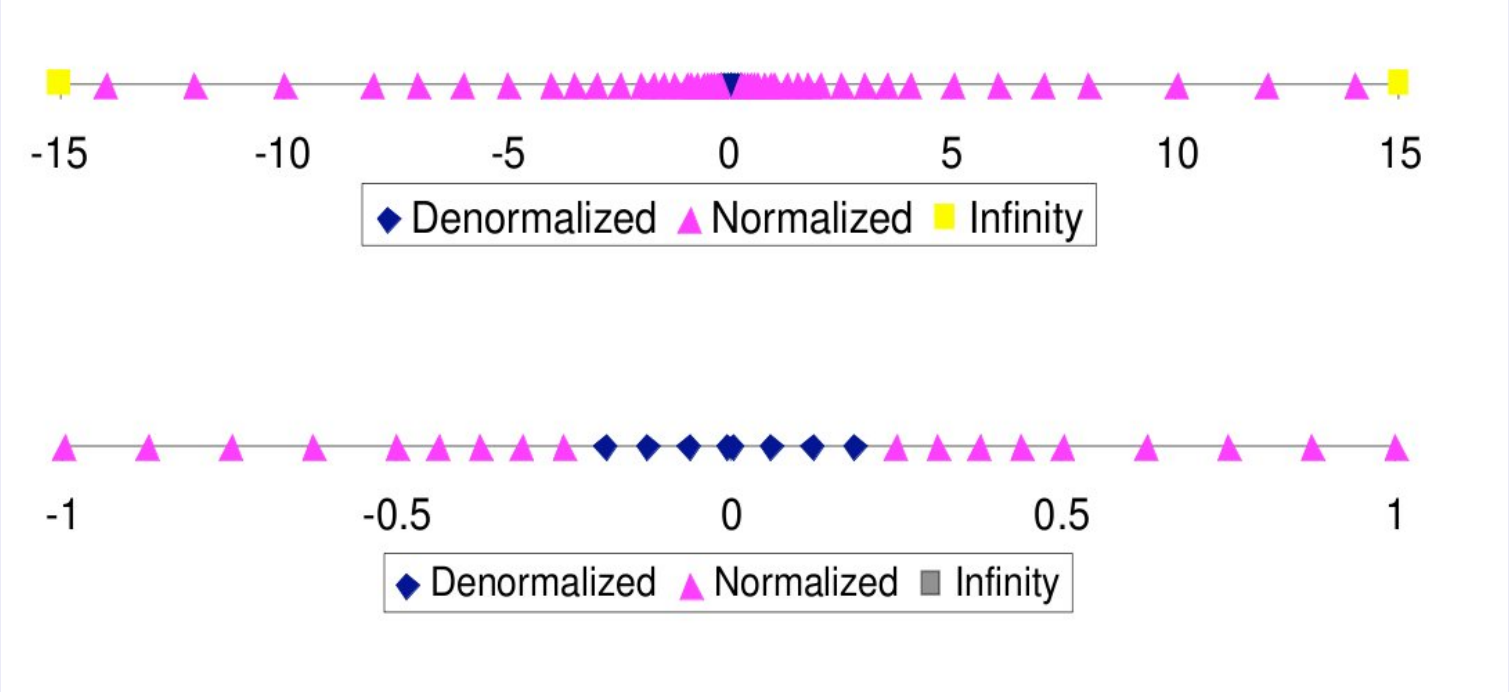
Exposant décalé	Mantisse	Type
0	0	0
0	différente de 0	nombre dénormalisé $s.2^{1-decalage}.m$
de 1 à $2^e - 2$	quelconque	nombre normalisé
$2^e - 1$	0	∞
$2^e - 1$	différente de 0	NaN

Exemple sur 32 bits

8 bits sont réservés pour l'exposant.

- Si l'exposant décalé vaut 0 ou 255, le nombre est dénormalisé
- Si l'exposant décalé est compris entre 1 et 254, il est normalisé

Valeurs représentables



Simple précision

Tableau de valeurs

Type	Exposant	Mantisse	Valeur approchée	écart / préc
Zéro	0000 0000	000 0000 0000 0000 0000 0000	0,0	
Plus petit nb dénormalisé	0000 0000	000 0000 0000 0000 0000 0001	$1,4.10^{-45}$	$1,4.10^{-45}$
Nb dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0010	$2,8.10^{-45}$	$1,4.10^{-45}$
Nb dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0011	$4,2.10^{-45}$	$1,4.10^{-45}$
Plus grand nb dénormalisé	0000 0000	111 1111 1111 1111 1111 1111	$1,17549421.10^{-38}$	
Plus petit nb normalisé	0000 0001	000 0000 0000 0000 0000 0000	$1.17549435.10^{-38}$	$1,4.10^{-45}$
Nb normalisé suivant	0000 0001	000 0000 0000 0000 0000 0001	$1.17549449.10^{-38}$	$1,4.10^{-45}$
Presque 1	0111 1110	111 1111 1111 1111 1111 1111	$0,99999994$	$0,6.10^{-7}$
1	0111 1111	000 0000 0000 0000 0000 0000	1,00000000	
Nombre suivant 1	0111 1111	000 0000 0000 0000 0000 0001	1,00000012	$1,2.10^{-7}$
Plus grand nombre normalisé	1111 1110	111 1111 1111 1111 1111 1111	$3,40282346.10^{38}$	2.10^{31}
∞	1111 1111	000 0000 0000 0000 0000 0000	∞	
Première valeur (dénormalisée) de NaN	1111 1111	000 0000 0000 0000 0000 0001	NaN	
NaN normalisé	1111 1111	010 0000 0000 0000 0000 0000	NaN	
Dernière valeur (dénormalisée) de NaN	1111 1111	011 1111 1111 1111 1111 1111	NaN	

PLAN

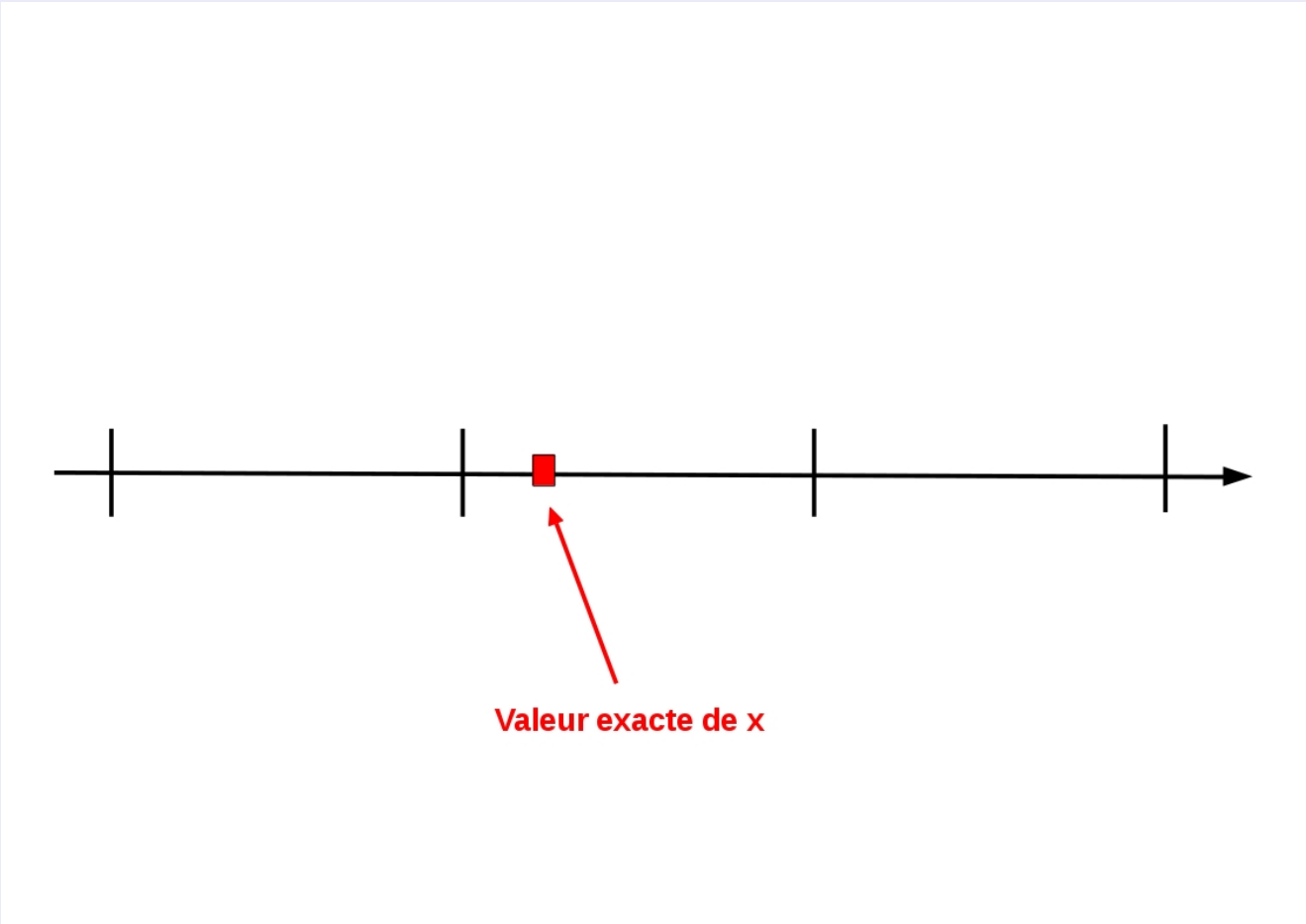
PLAN

Affectations ...
ou la nécessité d'arrondir

Affectation

Choisir le meilleur représentant

Comment passer de $\mathbb{R}$ à $\mathbb{F}$ (ensemble des nombres flottants) ?



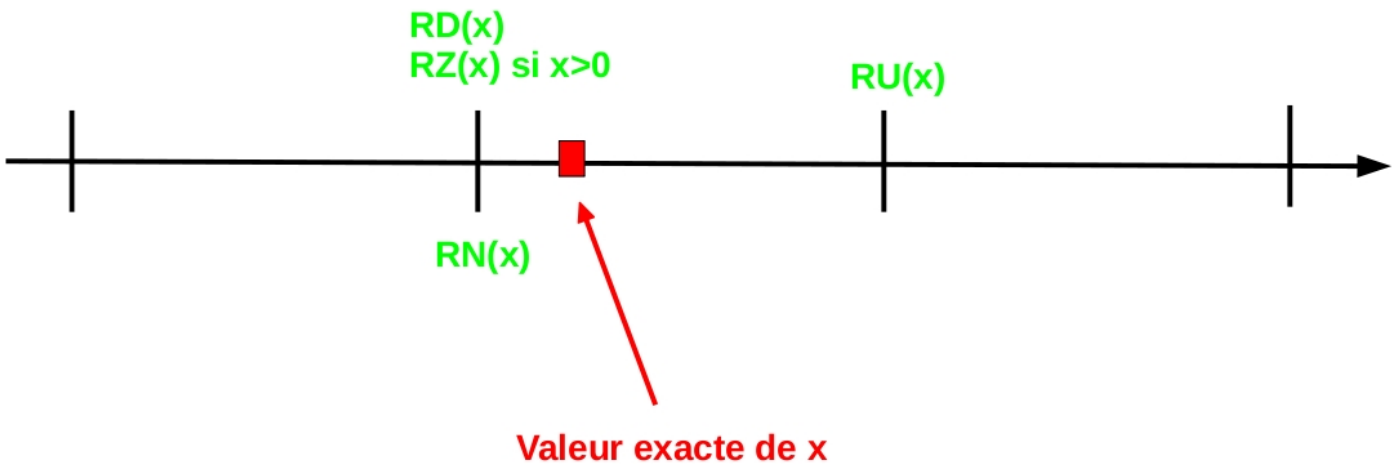
Modes d'arrondis

Arrondis

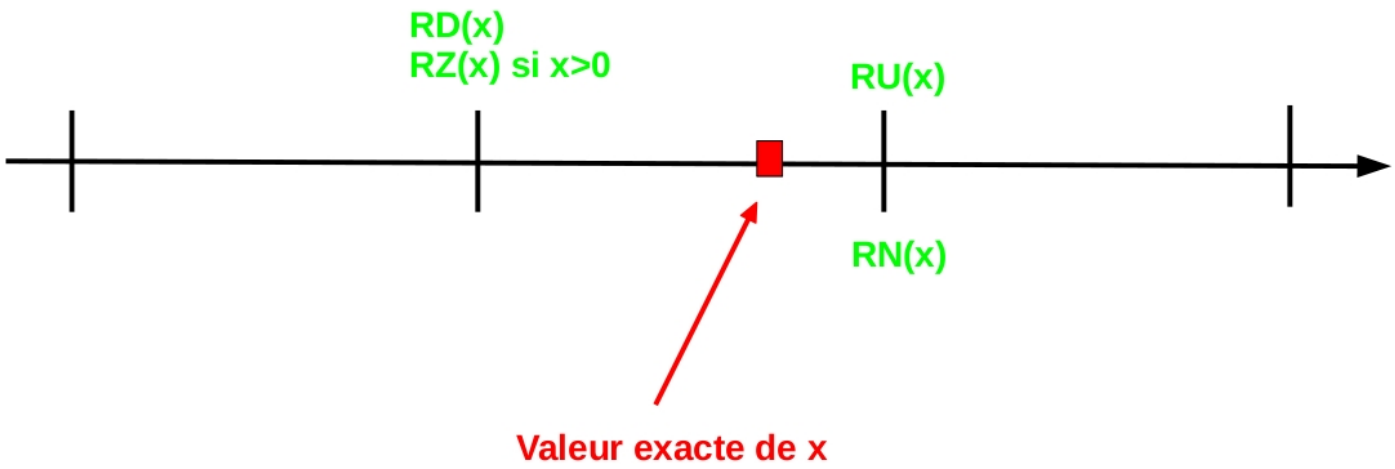
Toutes les opérations (y compris l'affectation) fournissent des valeurs arrondies vers une valeur représentable en machine
Il existe 4 modes d'arrondi (IEEE 754) :

- vers moins $+\infty$ (RU)
- vers plus $-\infty$ (RD)
- vers 0 (RZ)
- Au plus proche (RN)

Affectation et arrondis



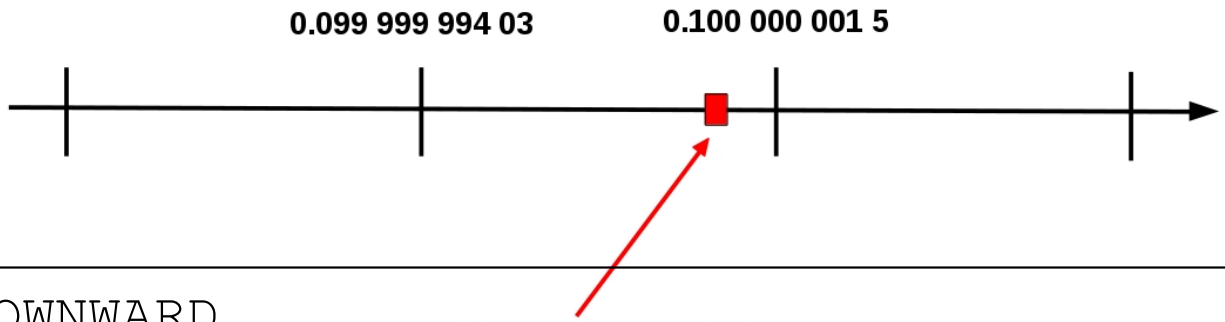
Affectations et arrondis



Exemple : affectation

```
void affectation(){
        float a;
        cout<<"Entrez la valeur de a"<<endl;
        cin>>a;
        cout<<"a : "<<a<<endl<<endl;
}

int main(){
  cout<<setprecision(10);

  cout<<"mode FE_DOWNWARD"<<endl;
  fesetround(FE_DOWNWARD);
  affectation();

  cout<<"mode FE_UPWARD"<<endl;
  fesetround(FE_UPWARD);
  affectation();

  cout<<"mode FE_TONEAREST"<<endl;
  fesetround(FE_TONEAREST);
  affectation();

  return 0;
}
```

Exemple : affectation de 0.1




```
mode FE_DOWNWARD
Entrez la valeur de a
0.1
a : 0.09999999403

mode FE_UPWARD
Entrez la valeur de a
0.1
a : 0.1000000015

mode FE_TONEAREST
Entrez la valeur de a
0.1
a : 0.1000000015
```

Question 4

La meilleure représentation de x est-elle toujours la plus proche ?

Arrondir ”correctement” le résultat d’un calcul

Arrondis corrects

arrondis corrects

Idée : éviter la propagation des erreurs d'arrondis à chaque opération
Soit

- x et y 2 nombres exactement représentables en machine
- $\odot$ une opération
- $\diamond$ le mode d'arrondi

La norme IEEE 754 exige que le resultat d'une operation $x \odot y$ soit égal à $\diamond(x \odot_{exact} y)$
Le résultat doit être le même que si on effectuait le calcul en précision infinie puis on arrondissait ce résultat.

La norme IEEE 754 n'impose l'arrondi correct que pour les opérations : $+$, $-$, $*$, $/$, $\sqrt{}$

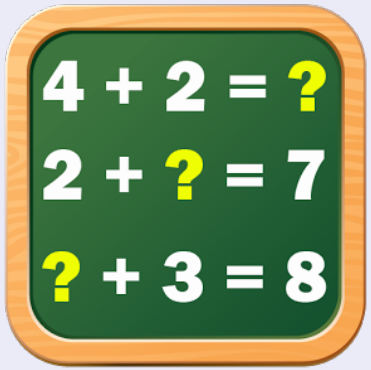
PLAN

Addition

Principe (pour les nombres normalisés)

Etapes pour l'addition (/soustraction) de deux réels S = A + B :

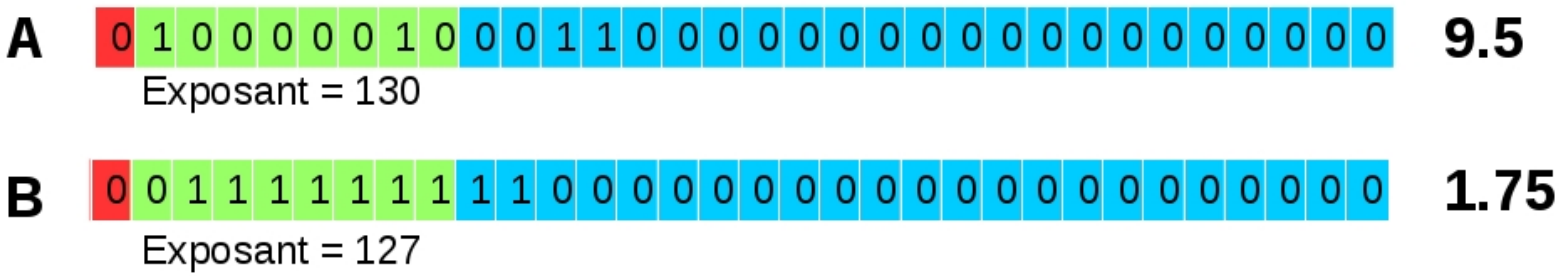
- Alignement des mantisses si les exposants de A et B sont différents
- Addition ou soustraction des mantisses alignées
- Renormalisation de la mantisse de la somme S si elle n'est pas normalisée (et gestion du bit implicite)



Exemples d'addition

Exemple : $9.5 + 1.75$

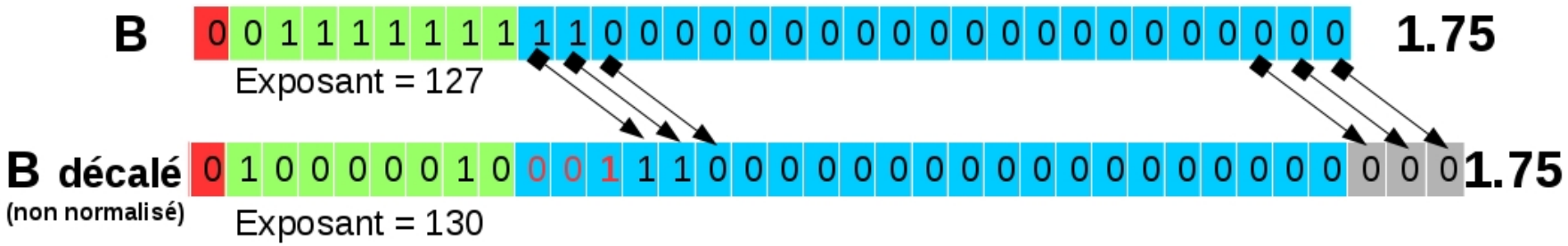
Représentation



Exemple : 9.5 + 1.75

1. Alignement des mantisses

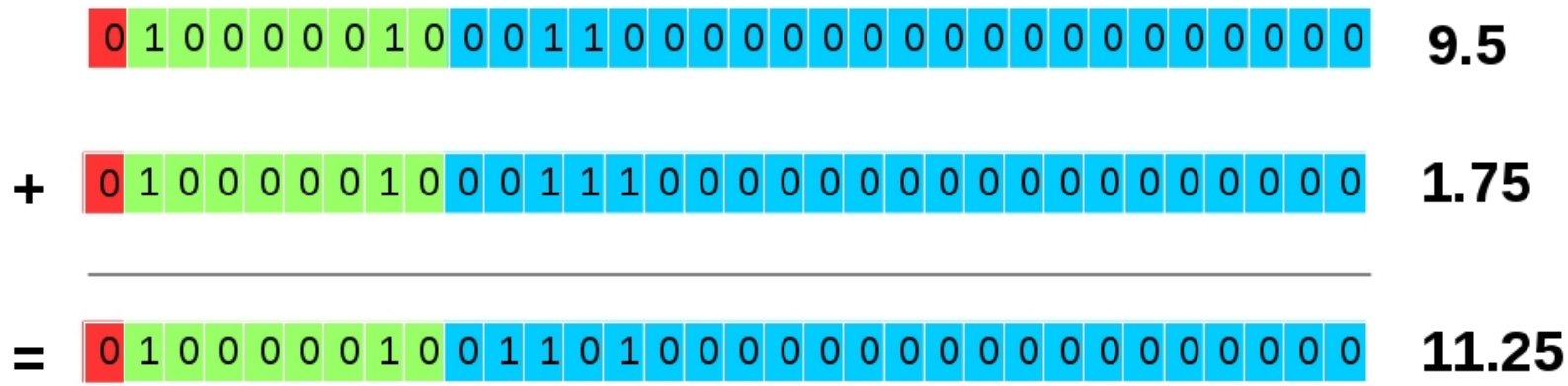
On décale vers la droite la mantisse du nombre ayant le plus petit exposant (ici B). Le nombre de décalages correspond à la différence entre les exposants



- Lors du décalage, il ne faut pas oublier de réintroduire le 1 (bit implicite)
- Après avoir réintroduit le 1, on introduit des 0
- Les bits qui "sortent" de la mantisse lors du décalage sont perdus : on peut donc perdre donc de l'information (pas ici puisque ce ne sont que des 0)

Exemple : $9.5 + 1.75$

2. Somme des mantisses



On a bien $9.5 + 1.75 = 11.25$

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a + b;
  cout<<"c = a + b : "<<c<<endl;

  return 0;
}
```

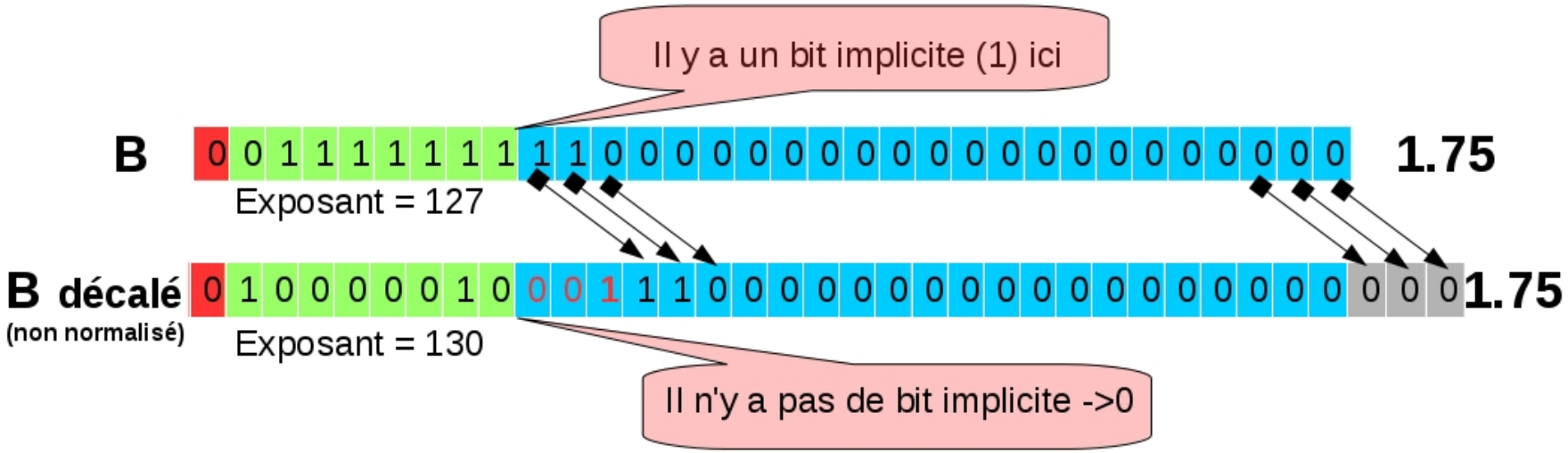
Vérification (suite)

```
Valeur de a :
9.5
a est stockee avec la valeur : 9.5

Valeur de b :
1.75
b est stockee avec la valeur : 1.75

c = a + b : 11.25
```

Et le bit implicite ?

Gestion du bit implicite lors d'une addition

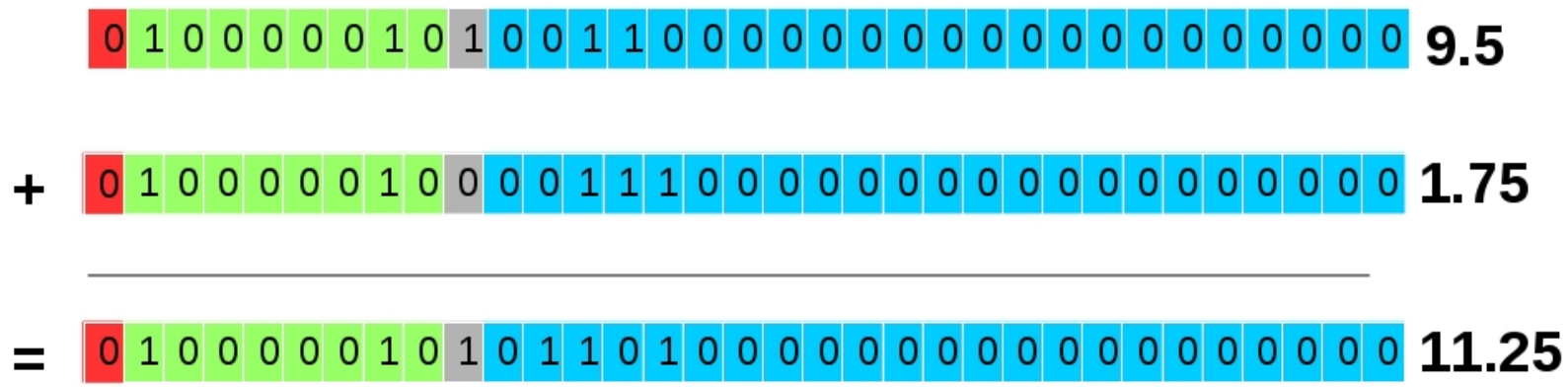
Alignement des mantisses



La 1re écriture inclut le bit implicite, mais pas la 2^{e}

Gestion du bit implicite lors d'une addition

Somme des mantisses



L'addition des bits *cachés* (/implictes) ne pose pas de problème ici (1+0=1), on obtient un nombre avec un bit implicite.
Ceci fonctionne car il y a eu un décalage de mantisse pour l'un des opérandes.

Addition

Formalisation (si décalage)

Soit $A = (1 + M_A).2^{E_A}$ et $B = (1 + M_B).2^{E_B}$

Après alignement des mantisses (si B a été décalé), on a $B = M'_B.2^{E_A}$ (car le bit implicite a été intégré à M'_B)

Et donc $A + B = (1 + M_A).2^{E_A} + M'_B.2^{E_A}$

soit $A + B = (1 + M_A + M'_B).2^{E_A}$

Or $C = (1 + M_C).2^{E_C}$

On voit qu'il suffit de faire la somme des 2 mantisses M_A et M'_B

Addition

Formalisation (en l'absence de décalage)

Soit $A = (1 + M_A).2^{E_A}$ et $B = (1 + M_B).2^{E_B}$

S'il n'y a pas de décalage , on a $E_A = E_B$

Et donc $A + B = (1 + M_A).2^{E_A} + (1 + M_B).2^{E_A}$

soit $A + B = (2 + M_A + M_B).2^{E_A}$

$= (1 + \frac{M_A+M_B}{2}).2^{E_A+1}$

Or $C = (1 + M_C).2^{E_C}$

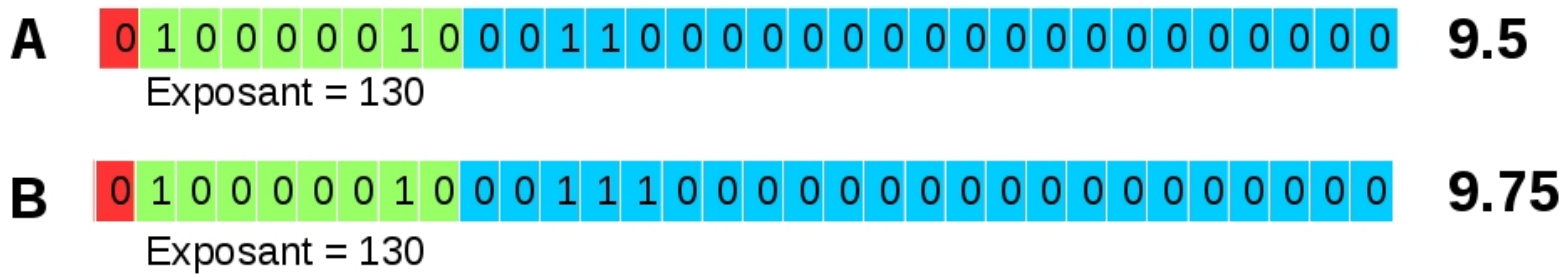
On voit

- qu'il faut ajouter 1 à l'exposant ($E_C = E_A + 1$)
- qu'il faut prendre la moitié de la somme des mantisses
  En pratique, on additionne les mantisses et on décale vers la droite d'un bit

Exemple d'addition sans décalage

$9.5 + 9.75$

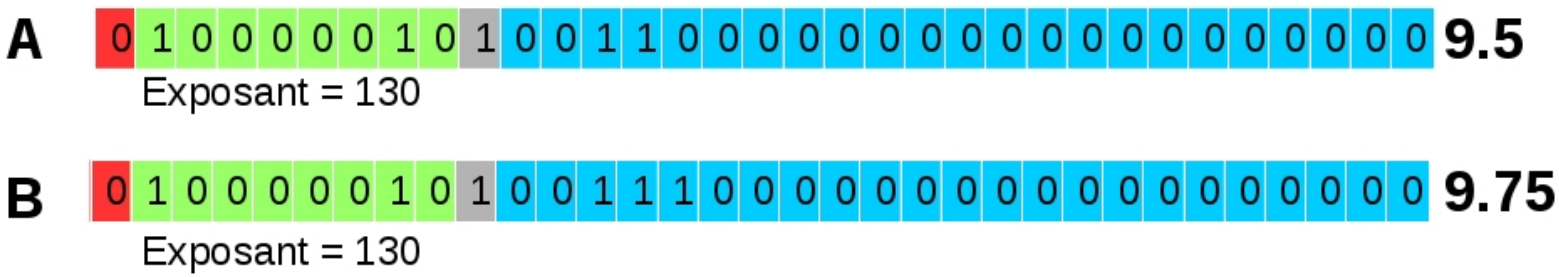
Représentation



Exemple d'addition sans décalage

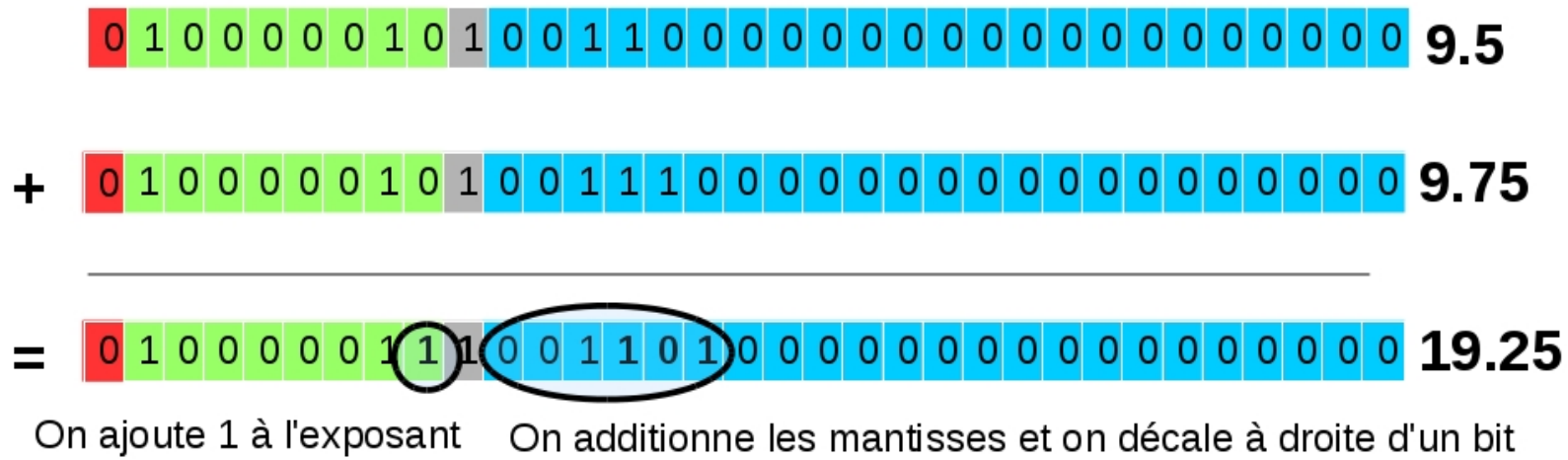
$9.5 + 9.75$

Représentation avec le bit implicite



Exemple d'addition sans décalage

normalisation de la mantisse



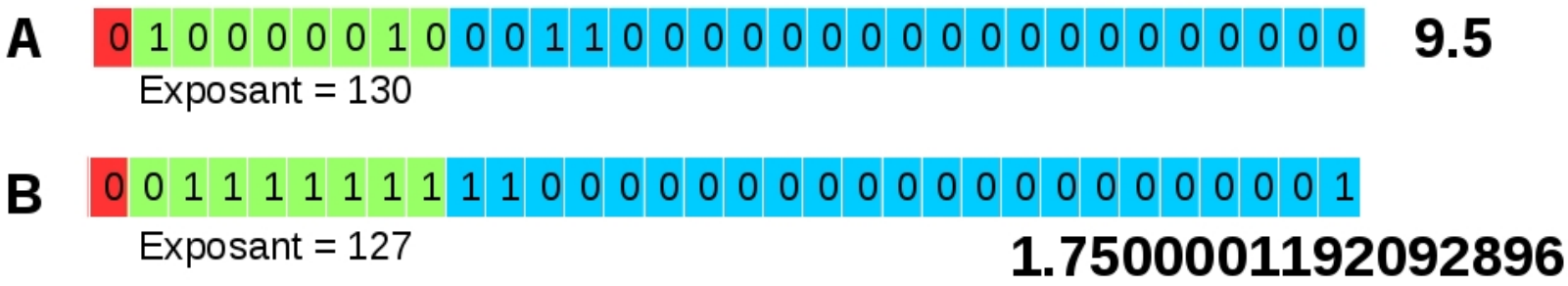
- On ajoute 1 à l'exposant
- On additionne les mantisses et on décale vers la droite d'un bit

Note : en décalant d'un bit vers la droite, on perd le dernier bit (perte de précision)

Perte de précision

Exemple : $9.5 + 1.7500001192092896$

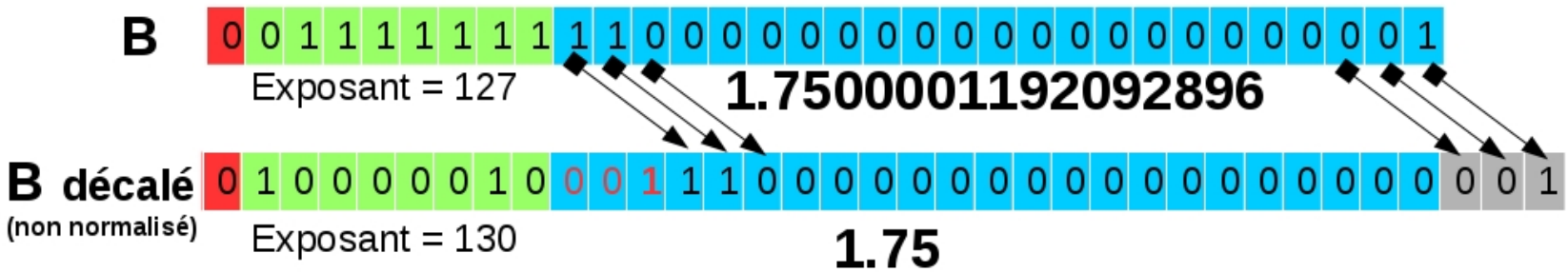
Représentation



Exemple : 9.5 + 1.7500001192092896

1. Alignement des mantisses

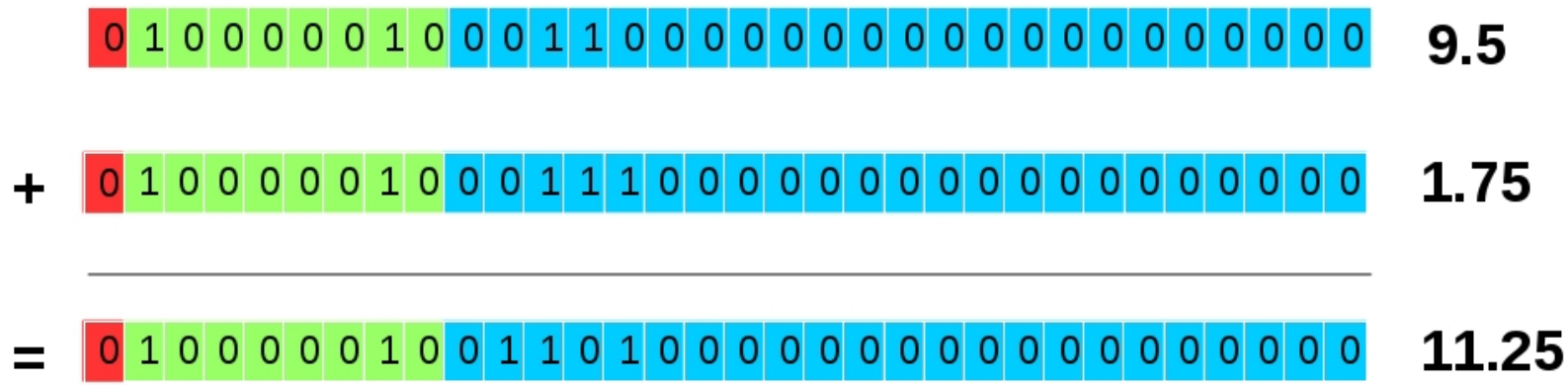
On décale vers la droite la mantisse du nombre ayant le plus petit exposant (ici B). Le nombre de décalages correspond à la différence entre les exposants



- Lors du décalage, il ne faut pas oublier de réintroduire le 1 (bit implicite)
- Après avoir réintroduit le 1, on introduit des 0
- Les bits qui "sortent" de la mantisse lors du décalage sont perdus : on perd donc de l'information

Exemple : $9.5 + 1.7500001192092896$

2. Somme des mantisses



On a $9.5 + 1.7500001192092896 = 11.25$

- On a perdu en précision
- Mais 11.25 est le meilleur représentant pour 11.2500001192092896
- L'addition respecte bien l'arrondi correct

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a + b;
  cout<<"c = a + b : "<<c<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
9.5
a est stockee avec la valeur : 9.5

Valeur de b :
1.7500001192092896
b est stockee avec la valeur : 1.7500001192092896

c = a + b : 11.25
```

Soustraction

soustraction

Formalisation (si décalage)

Soit $A = (1 + M_A).2^{E_A}$ et $B = (1 + M_B).2^{E_B}$

Après alignement des mantisses (si B a été décalé), on a $B = M'_B.2^{E_A}$ (car le bit implicite a été intégré à M'_B)

Et donc $A - B = (1 + M_A).2^{E_A} - M'_B.2^{E_A}$

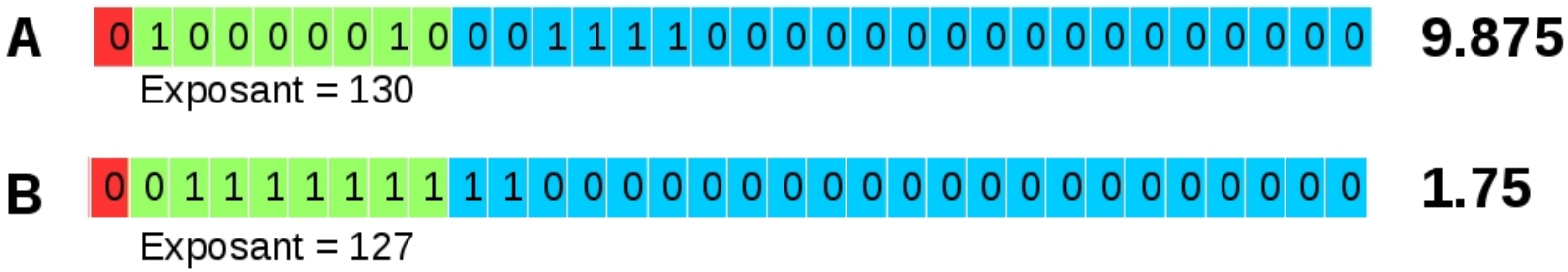
soit $A + B = (1 + M_A - M'_B).2^{E_A}$

Or $C = (1 + M_C).2^{E_C}$

On voit qu'il suffit de faire la différence des 2 mantisses M_A et M'_B

Exemple : 9.875 – 1.75

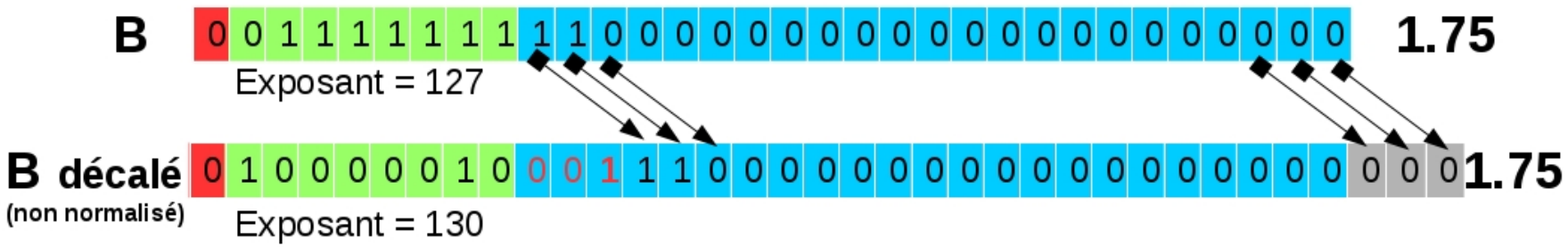
Représentation



Exemple : 9.875 – 1.75

1. Alignement des mantisses

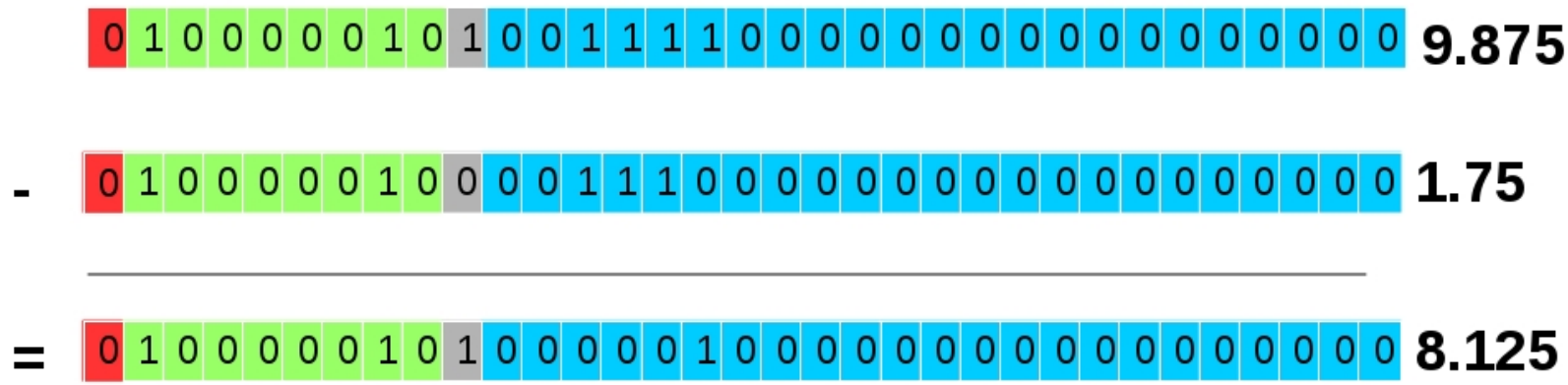
On décale vers la droite la mantisse du nombre ayant le plus petit exposant (ici B). Le nombre de décalages correspond à la différence entre les exposants



- Lors du décalage, il ne faut pas oublier de réintroduire le 1 (bit implicite)
- Après avoir réintroduit le 1, on introduit des 0
- Les bits qui "sortent" de la mantisse lors du décalage sont perdus : on peut donc perdre donc de l'information (pas ici puisque ce ne sont que des 0)

Exemple : 9.875 − 1.75

2. Soustraction des mantisses



On a bien $9.875 - 1.75 = 8.125$

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a - b;
  cout<<"c = a - b : "<<c<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
9.875
a est stockee avec la valeur : 9.875

Valeur de b :
1.75
b est stockee avec la valeur : 1.75

c = a - b : 8.125
```

soustraction

Formalisation (en l'absence de décalage)

Soit $A = (1 + M_A).2^{E_A}$ et $B = (1 + M_B).2^{E_B}$

S'il n'y a pas de décalage , on a $E_A = E_B$

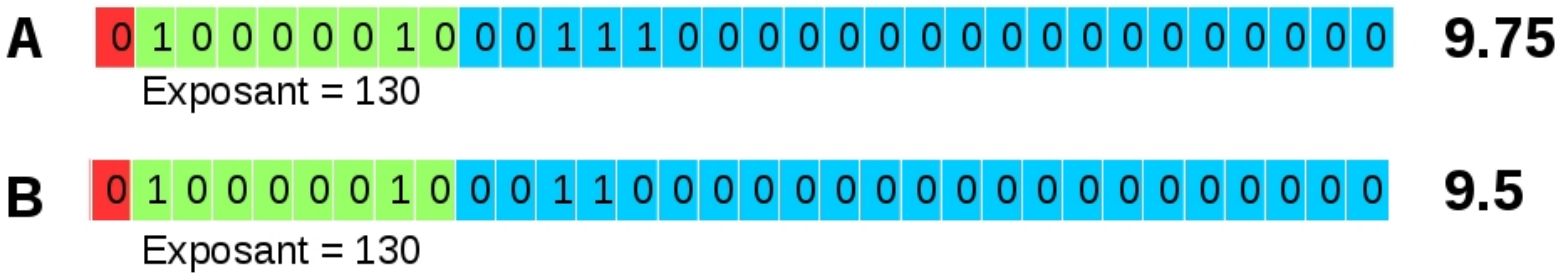
Et donc $A - B = (1 + M_A).2^{E_A} - (1 + M_B).2^{E_A}$

soit $A - B = (M_A - M_B).2^{E_A}$

- Il faut soustraire les mantisses
- Mais on a perdu le bit implicite. Il faut donc re-normaliser la mantisse
  - en la décalant vers la gauche (pour mettre le 1^{er} bit de valeur 1 dans le bit implicite)
  - et mettre à jour l'exposant en conséquence

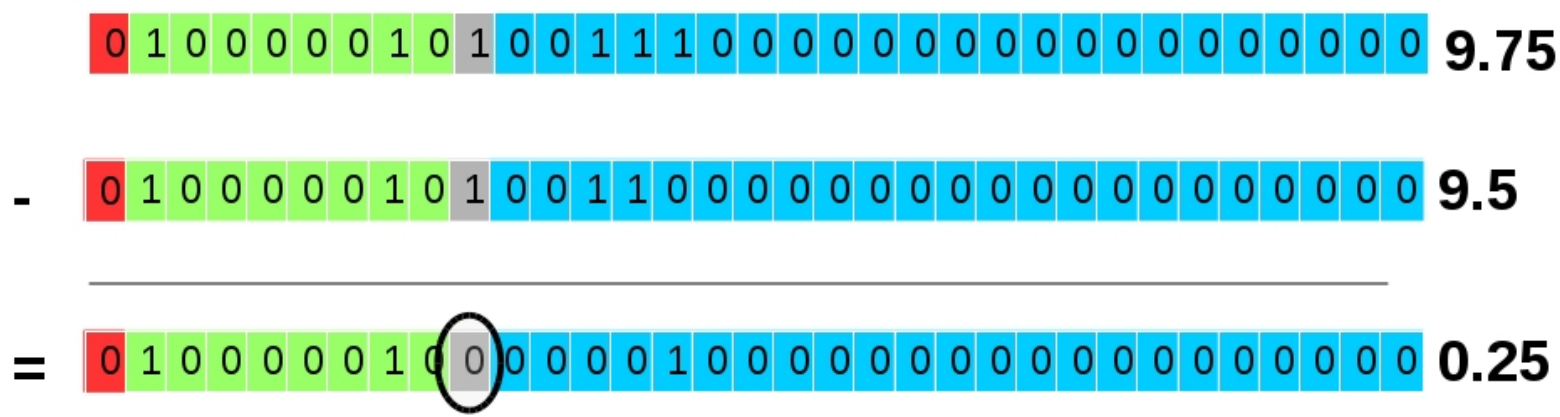
Exemple : 9.75 − 9.5

Représentation



Exemple : 9.75 − 9.5

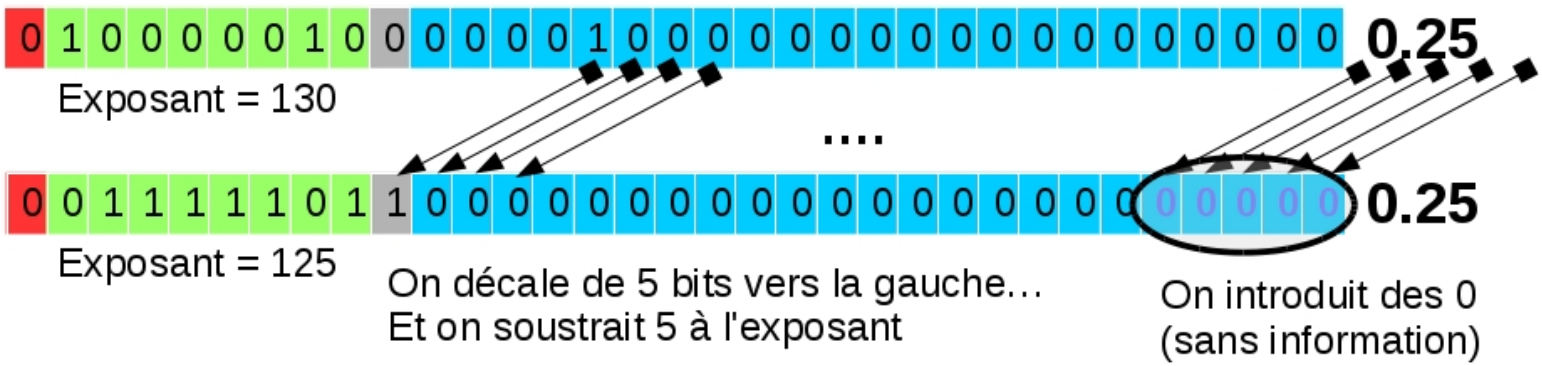
Soustraction des mantisses



On a perdu le bit implicite

Exemple : 9.75 − 9.5

3. Re-normalisation de la mantisse



On a bien $9.75 - 9.5 = 0.25$

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a - b;
  cout<<"c = a - b : "<<c<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
9.75
a est stockee avec la valeur : 9.75

Valeur de b :
9.5
b est stockee avec la valeur : 9.5

c = a - b : 0.25
```

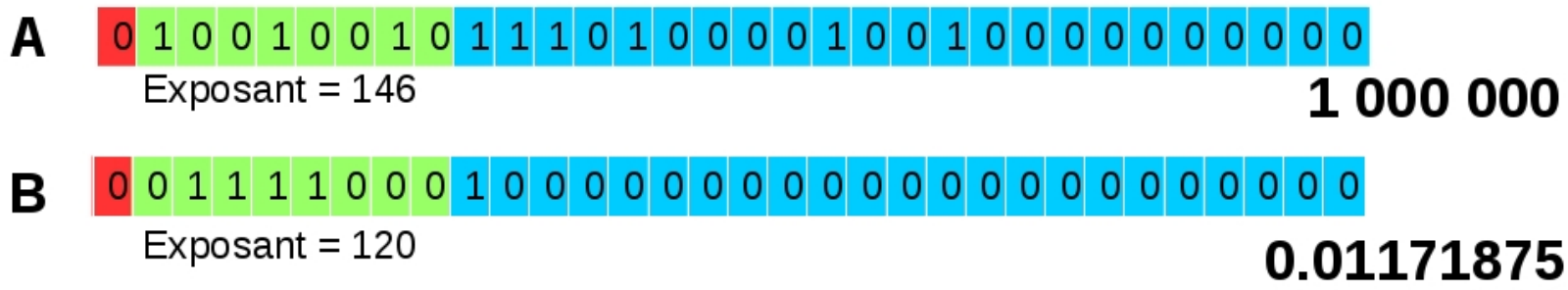
PLAN

Absorption

Absorption

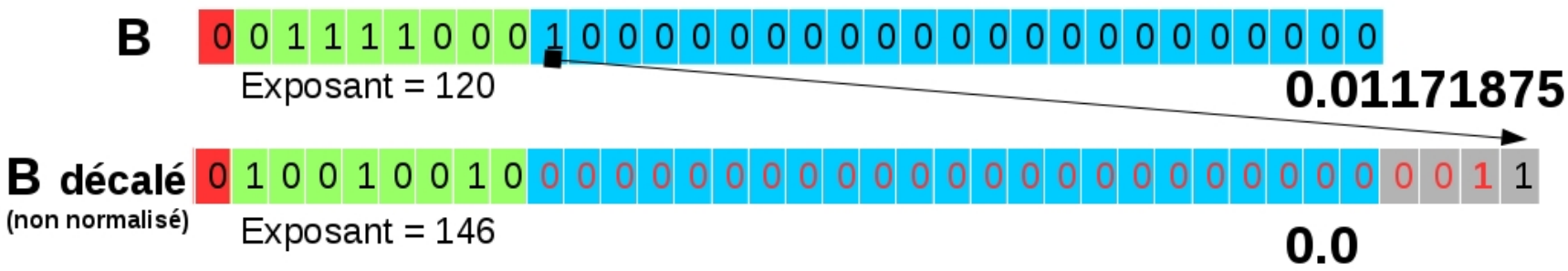
Lors d'une addition de 2 nombres ayant des ordres de grandeur très différents, on peut perdre toute l'information du plus petit.

Exemple



Absorption

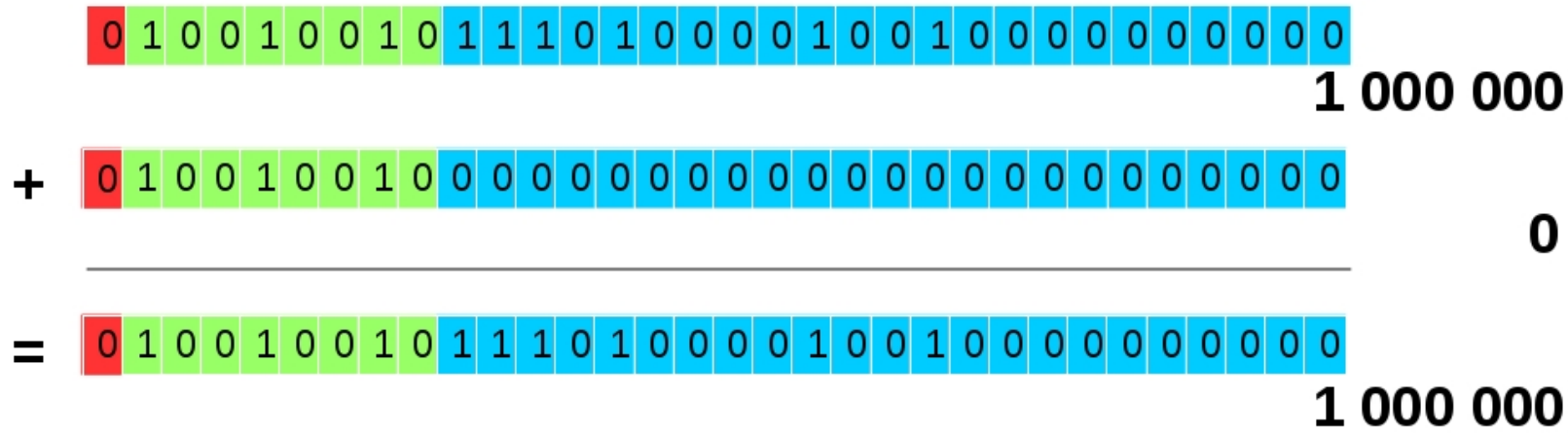
1. Alignement des mantisses



Lors de l'alignement on perd toute l'information sur B

Absorption

2. Somme des mantisses



On a $10^6 + 0.01171875 = 10^6$!

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a + b;
  cout<<"c = a + b : "<<c<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
1000000
a est stockee avec la valeur : 1000000

Valeur de b :
0.01171875
b est stockee avec la valeur : 0.01171875

c = a + b : 1000000
```

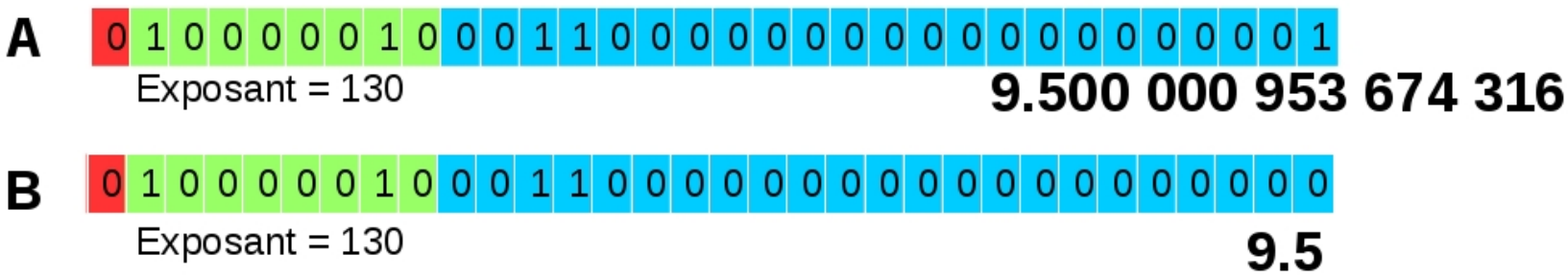
Cancellation

cancellation

Contexte

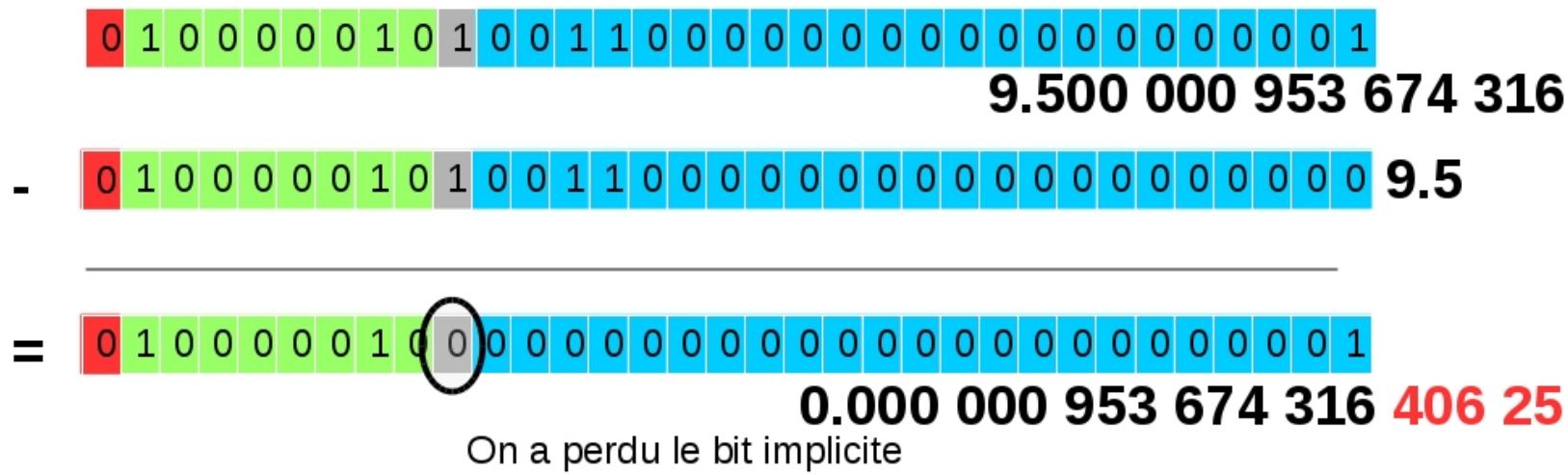
Soustraction de 2 nombres très proches

Exemple



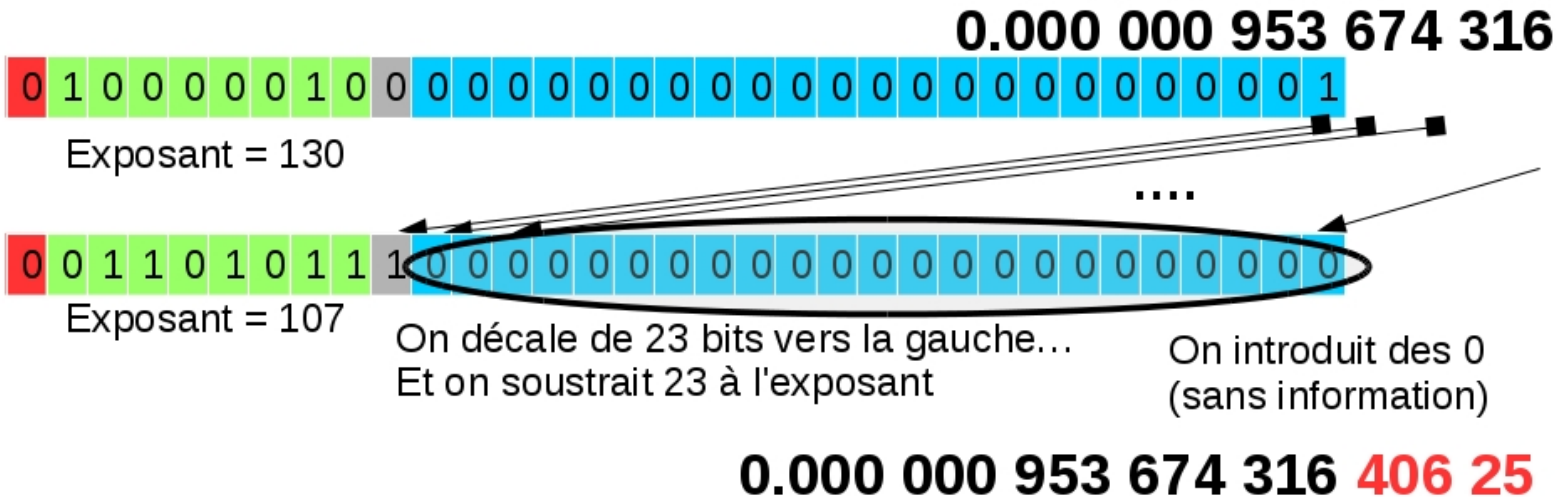
cancellation

Soustraction des mantisses



cancellation

Re-normalisation



- Le résultat respecte l'arrondi correct
  C'est le plus proche représentant du vrai résultat
- Mais on a introduit des 0 arbitrairement (manque d'information)
- Aucun bit de la mantisse n'est significatif

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a - b;
  cout<<"c = a - b : "<<c<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
9.500000953674316
a est stockee avec la valeur : 9.500000953674316

Valeur de b :
9.5
b est stockee avec la valeur : 9.5

c = a - b : 9.5367431640625e-07
```

Cancellation et absorption

Cancellation et absorption

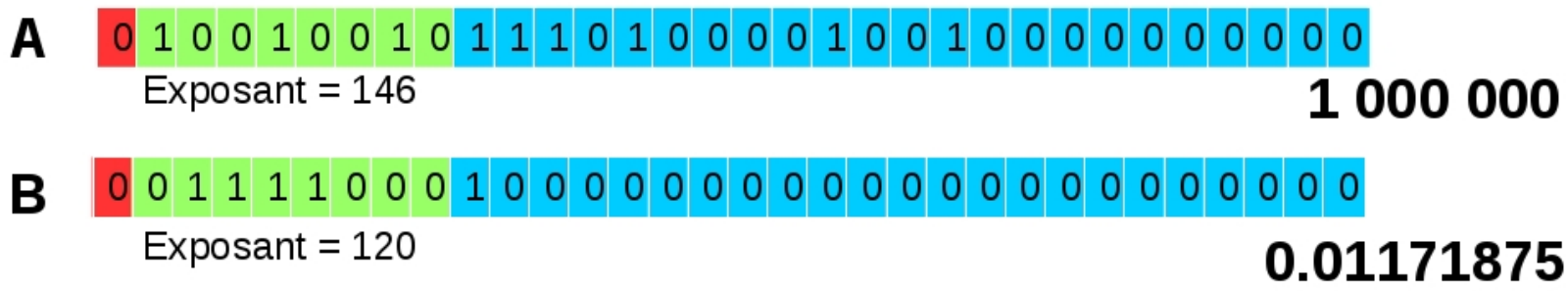
- Lors d'une cancellation, on introduit arbitrairement des 0 dans la mantisse (car on ne dispose pas de plus de précision sur les opérandes)
- Si les opérandes sont issus d'un calcul précédent ils ont pu subir une perte de précision (en cas d'absorption par exemple)

Cette perte de précision va alors être ré-intégrée et amplifiée.
Dans ce cas, les bits abitrairement introduits (des 0) ne sont pas les bons.

Exemple : $(10^6 + 0.01171875) - 10^6$

$10^6 + 0.01171875$

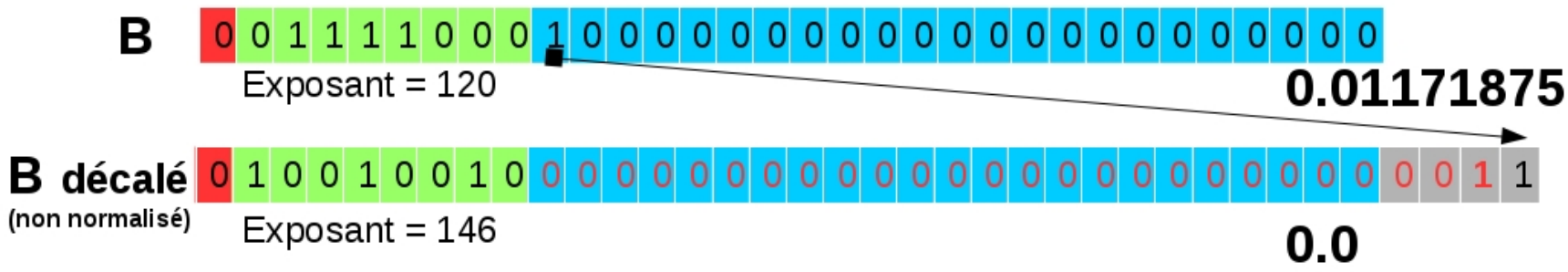
Représentation



Exemple : $(10^6 + 0.01171875) - 10^6$

$10^6 + 0.01171875$

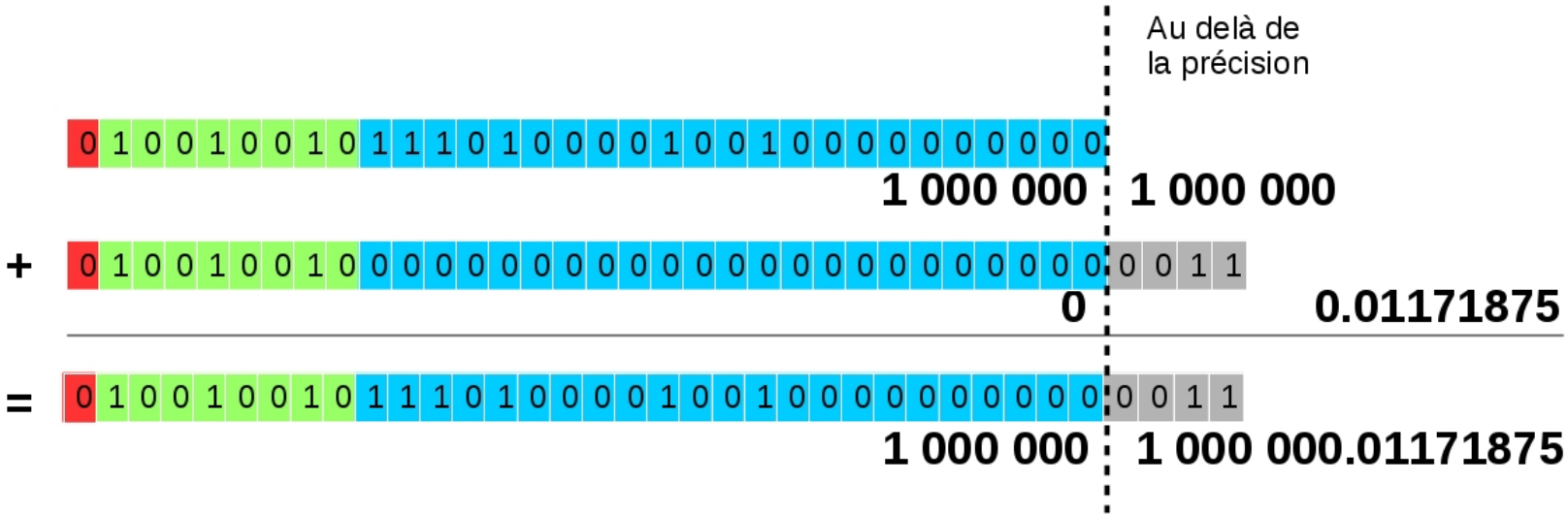
Alignement de la mantisse



Exemple : $(10^6 + 0.01171875) - 10^6$

$10^6 + 0.01171875$

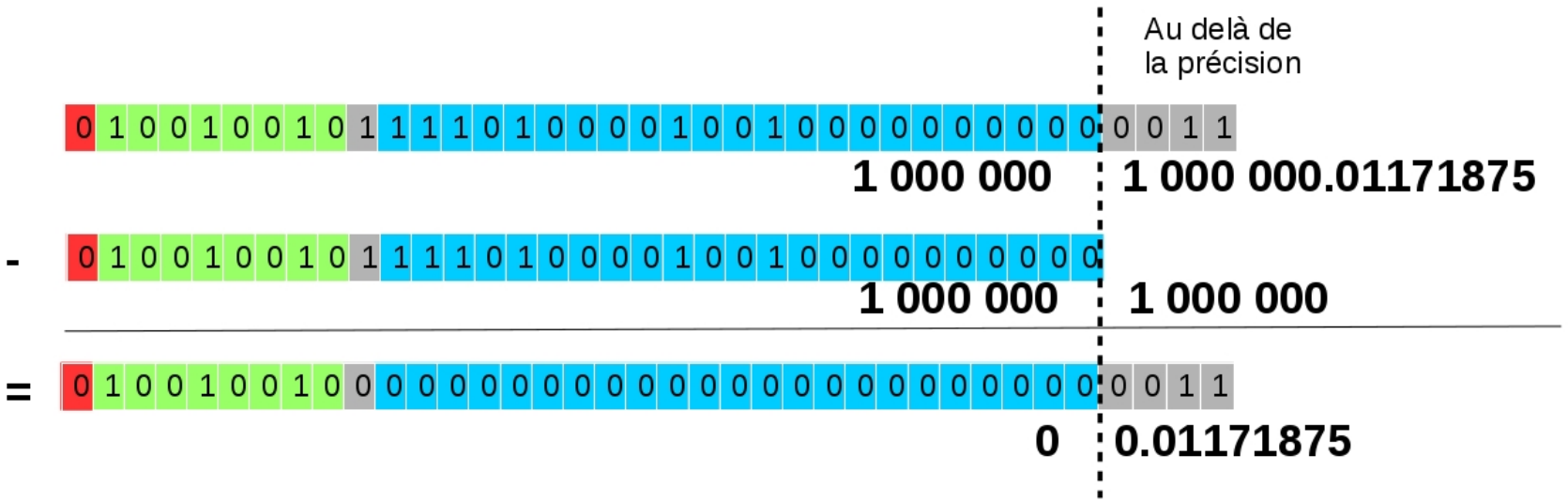
Somme des mantisses



Exemple : $(10^6 + 0.01171875) - 10^6$

$(10^6 + 0.01171875) - 10^6$

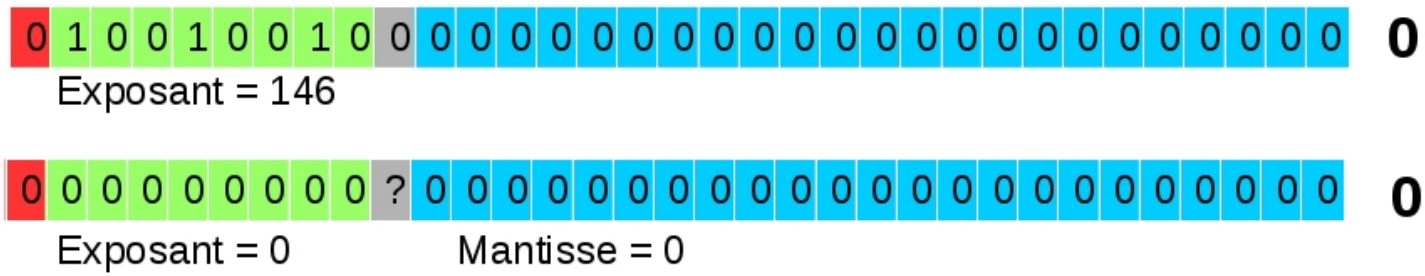
Somme des mantisses



Exemple : $(10^6 + 0.01171875) - 10^6$

Renormalisation

Il reste à écrire la valeur 0 sous sa forme dénormalisée



On a donc $(10^6 + 0.01171875) - 10^6 = 0$

Vérification

```
int main(){
  cout<<setprecision(17);

  float a, b, c,d;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"a est stockee avec la valeur : "<<a<<endl<<endl;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"b est stockee avec la valeur : "<<b<<endl<<endl;

  c = a + b;
  cout<<"c = a + b : "<<c<<endl;

  d = c - a;
  cout<<"d = (a + b) - a : "<<d<<endl;

  return 0;
}
```

Vérification (suite)

```
Valeur de a :
1000000
a est stockee avec la valeur : 1000000

Valeur de b :
0.01171875
b est stockee avec la valeur : 0.01171875

c = a + b : 1000000

d = (a + b) - a : 0
```

Impact de l'ordre des opérations sur le résultat

... ou comment minimiser les effets de l'absorption

Impact de l'ordre des opérations

Somme des inverses de i

- De 1 à N : $$\sum_{i=1}^{n} \frac{1}{i} \qquad (1)$$
- ou de N à 1 : $$\sum_{i=n}^{1} \frac{1}{i} \qquad (2)$$

Impact de l'ordre des opérations

Somme des inverses de i (de 1 à N)

En simple précision, on obtient :

N	10^5	10^6	10^7	10^8
valeur exacte	12.09015	14.39273	16.69531	18.99790
$1 \rightarrow N$	12.09085	14.35736	15.40368	15.40368
$N \rightarrow 1$	12.09015	14.39265	16.68603	18.80792

- $\sum_{i=1}^{n} \frac{1}{i}$: Pour n grand, on finira par additionner des réels d'ordres de grandeur très différents : absorptions
- $\sum_{i=n}^{1} \frac{1}{i}$: Les ordres de grandeur dans les additions seront semblables

Propriétés algébriques

Propriétés algébriques dans $\mathbb{F}$

Addition et multiplication

- La commutativité est respectée pour l'addition et la multiplication
  - $a + b = b + a$
  - $a * b = b * a$
- L'associativité n'est pas respectée (en général) ni pour l'addition, ni pour la multiplication
  - $(a + b) + c \stackrel{?}{=} a + (b + c)$
  - $(a * b) * c \stackrel{?}{=} a * (b * c)$
- La distributivité n'est pas respectée (en général) entre la multiplication et l'addition
  - $a(b + c) \stackrel{?}{=} ab + ac$

PLAN

Opérations mathématiques et arrondi

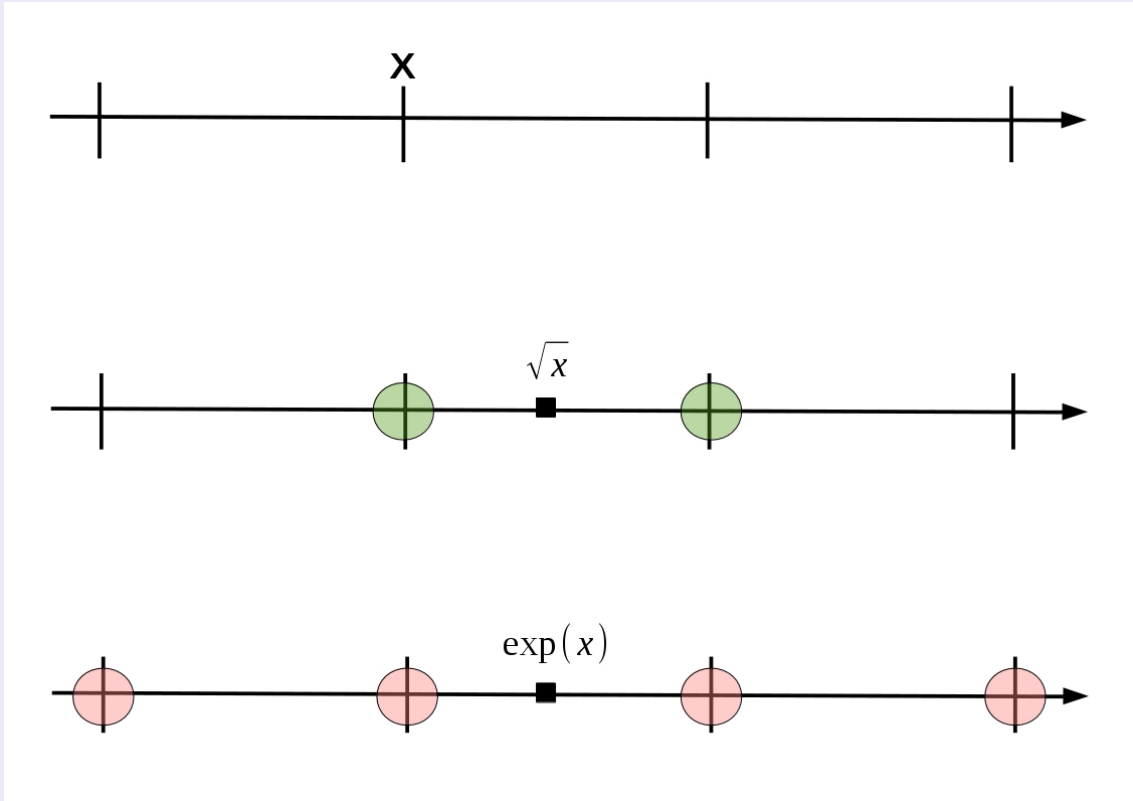
Arrondi correct et norme

- La norme IEEE 754 n'impose l'arrondi correct que pour les opérations : $+, -, *, /, \sqrt{}$
- Pour toutes les autres opérations, la norme conseille l'arrondi correct.

Pour ces fonctions, on ne peut donc jamais être sûr que le résultat d'une opération fournit le meilleur résultat dans $\mathbb{F}$.

L'arrondi correct n'est pas garanti pour une opération mathématique (autre que $+, -, *, /, \sqrt{}$)

Arrondis corrects / non corrects



- Pour $\sqrt{x}$, l'arrondi correct est assuré
  Le résultat est connu (/assuré) (au mode d'arrondi près)
- Pour $exp(x)$, l'arrondi correct n'est pas assuré
  On ne peut pas connaître la valeur retournée

Opérations mathématiques et arrondi

L'arrondi correct : un problème difficile et/ou coûteux

- Difficulté : Garantir l'arrondi correct des fonctions mathématiques est un problème difficile en général.
- Coût : Le nombre de chiffres supplémentaires nécessaires pour déterminer l'arrondi est délicat à contrôler a priori, et peut être grand.

Ex : $exp(0.0940782231357287 8) = 1.0986456820663385000000000000000278...$

Il faut donc calculer plus de 30 chiffres décimaux du résultat pour en donner l'arrondi correct à une précision de 16 chiffres significatifs, en arrondi au plus près.

$$exp(0.0940782231357287 8) \approx 1.0986456820663 39$$

Exceptions

Exceptions

Exceptions

La norme IEEE 754 introduit les exceptions suivantes :

- invalid operation : se produit lorsqu'une opération interdite est effectuée ; le résultat du calcul en question sera NaN
- division by zero : se produit lorsqu'on tente de diviser un nombre (non nul) par zéro ; le résultat sera $+\infty$ ou $-\infty$
- overflow : se produit lorsque le résultat d'un calcul est trop grand (en valeur absolue) pour être stocké ; on arrondit à $+\infty$ ou $-\infty$
- underflow : se produit lorsque le résultat d'un calcul est trop petit (en valeur absolue) pour être stocké ; on arrondit à zéro
- inexact : se produit lorsqu'on effectue un autre arrondi pour pouvoir stocker un nombre flottant (parce que le résultat a plus de chiffres significatifs qu'on peut en stocker).

comparaisons

comparaisons

La comparaison entre 2 réels doit considérée avec la plus grande prudence !

- Ne jamais tester if $x == y$ ... surtout si x et y sont le résultat d'un calcul précédent
- Attention aux tests d'arrêt if $|x_n - x_{n-1}| < \epsilon$ ... ou même if $\frac{|x_n - x_{n-1}|}{|x_n|} < \epsilon$ ...
- Attention aux algorithmes dépendant de la comparaison entre 2 réels
  Ex : if $(x < y)$ <Traitement1> else <Traitement2>

PLAN

Envrionnement logiciel et matériel

Le traitement de la précision des calculs se situe sur l'ensemble de l'environnement logiciel et matériel

- Processeur
- Système d'exploitation
- Langage de programmation
- Compilateur
- Développeur

Processeur

Chaque processeur

- réalise des opérations de calcul (Floating Point Unit)
- possède ses propres registres (pour stocker des flottants)
- gère les exceptions
- écrit les résultats en mémoire

Système d'exploitation

Le système d'exploitation

- transmet les exceptions
- calcule les opérations (/fonctions) non gérées par le processeur
- gère le statut du calcul flottant : precision, mode d'arrondi

Langage de programmation

Les langages de programmation n'ont pas toujours les mêmes règles.
Certains imposent un ordre (C++ sauf avec certaines options...), d'autres non (Fortran)

Exemple

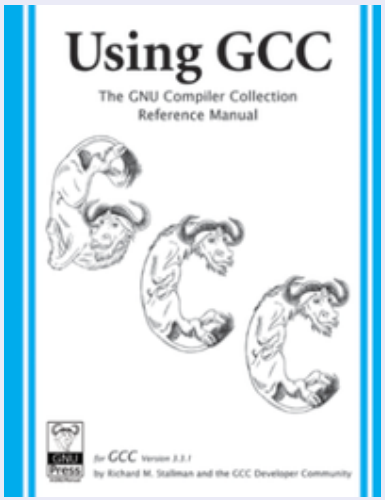
$$x = a + b + c + d$$

- Dans quel ordre seront effectués les calculs ?
- Quel est l'impact sur le résultat ?

Compilateur

Le Compilateur

- possède des centaines d'options
- certaines options permettent de préserver la sémantique du langage
- mais ne sont pas toujours (/rarement ?) activées par défaut
- Marketing : les valeurs par défaut doivent optimiser la vitesse (...et pas la précision)



Développeur

- Est censé maîtrisé tout ce qui précède
- Et ... est tenu pour responsable de tout problème sur le logiciel, y compris la perte de précision

PLAN

PLAN

Précision arbitraire (ou multiprécison)

Principe

- Utiliser des flottants codés sur plus de bits
- Le nombre de bits est paramétrable par l'utilisateur (précision arbitraire)
  Rappel : simple précision (32 bits) et double précision (64 bits)
- La *"précision"* n'est définie que par le nombre de bits qui sont utilisés pour stocker les réels (et pas sur les calculs)

Tous les problèmes de l'arithmétique flottante sont toujours présents.
On *"repousse"* les problèmes.

Exemple : GMP

The GNU Multiple Precision Arithmetic Library
Cf : http ://gmplib.org

Arrondis corrects pour toutes les fonctions

Principe

- Proposer une bibliothèque mathématique dans laquelle toutes les fonctions mathématiques garantissent l'arrondi correct
- Pour chaque fonction, on doit connaître le nombre de bits nécessaires pour assurer l'arrondi correct en simple et double précision.

  Ex : Il a été démontré qu'avec 118 bits de précision (mantisse), on pouvait garantir l'arrondi correct en double précision du logarithme.

Exemple : CRlibm

Correctly Rounded Mathematical LIBrary
Cf : http ://lipforge.ens-lyon.fr/www/crlibm/

Précision arbitraire **ou** arrondis corrects pour toutes les fonctions ?

Principe

Faut-il vraiment choisir entre le fromage et le dessert ?

Précision arbitraire **et** arrondis corrects pour toutes les fonctions

Principe

- Coder les flottants sur un nombre arbitraire de bits
- Garantir l'arrondi correct pour toutes les fonctions mathématiques

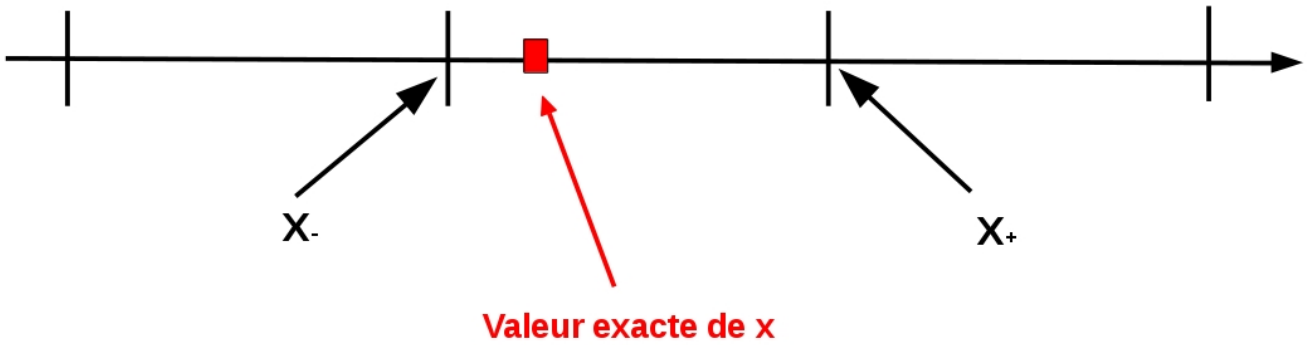
Exemple : MPFR

Cf : http ://www.mpfr.org/

PLAN

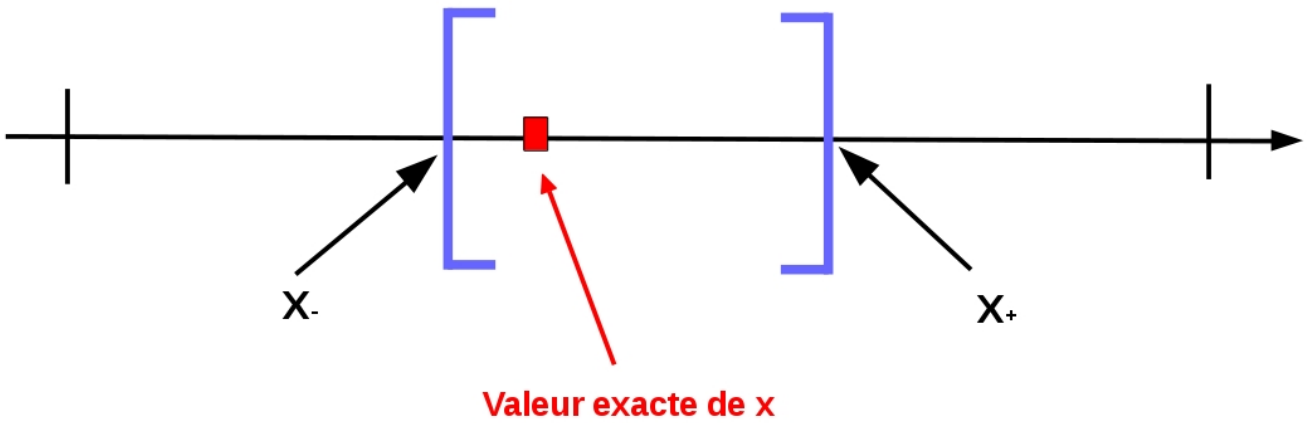
Principe

Comment choisir la meilleure représentation de x ?



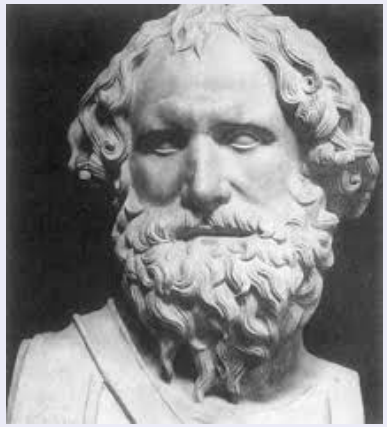
Principe

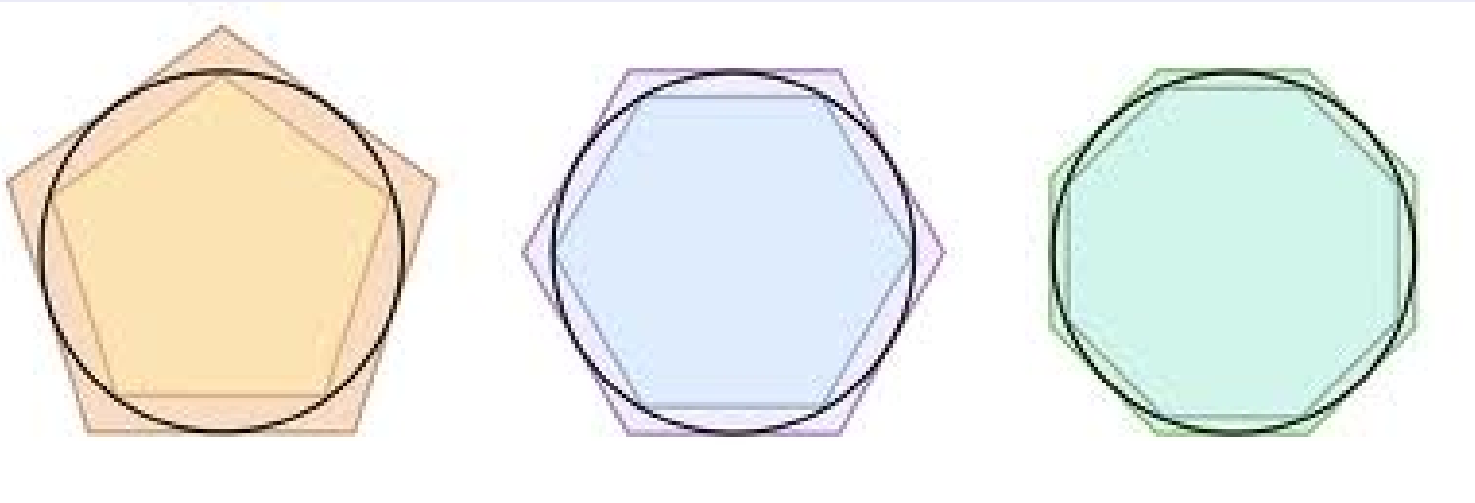
Et si on choisissait l'intervalle ?



Une idée pas si nouvelle...

Archimède utilisait l'arithmétique par intervalles





Formalisation

Formule générale

$$X \diamond Y = Hull\{x \diamond y; x \in X, y \in Y\}$$

Opérations courantes

$$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y} \ , \ \overline{x} + \overline{y}]$$

$$[\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] = [\underline{x} - \overline{y} \ , \ \overline{x} - \underline{y}]$$

$$[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] = [min(\underline{x} \times \underline{y} \ , \ \underline{x} \times \overline{y} \ , \ \overline{x} \times \underline{y} \ , \ \overline{x} \times \overline{y}) \ , \ max(id)]$$

$$\begin{aligned} [\underline{x}, \overline{x}]^2 &= [min(\underline{x}^2, \overline{x}^2) \ , \ max(\underline{x}^2, \overline{x}^2)] \ si \ 0 \notin [\underline{x}, \overline{x}] \\ &= [0, max(\underline{x}^2, \overline{x}^2)] \ sinon \end{aligned}$$

Arithmétique d'intervalles

Mise en oeuvre

- En utilisant l'arithmétique flottante IEEE-754
- et en jouant sur les modes d'arrondis

$$\text{Ex : } \sqrt{[2,3]} = [\nabla\sqrt{2}, \Delta\sqrt{3}]$$

MPFI

Une bibliothèque d'arithmétique par intervalles multi-précision basée sur la bibliothèque MPFR.

Inconvénients

- Sur-estimation : l'intervalle résultat peut être (très) grand
  $[-\infty, +\infty]$ contient le résultat ! !
- Efficacité
  La nécessité de changer souvent le mode d'arrondi a un impact important sur l'efficacité
- Perte de dépendance entre variables

$$X - X = \{x - y; x \in X, y \in X\}$$
$$\neq \{x - x; x \in X\} = \{0\}$$

Propriétés

- L'associativité et la commutativité sont respectées
- Mais certaines propriétés sont perdus
  - La soustraction n'est pas l'opération inverse de l'addition
    Ex : $X - X \neq [0]$
  - La division n'est pas l'opération inverse de la multiplication
  - $[\underline{x}, \overline{x}]^2 \neq [\underline{x}, \overline{x}] \times [\underline{x}, \overline{x}]$

Propriété fondamentale

Le vrai résultat (inconnu) est contenu dans l'intervalle résultat calculé

PLAN

Quel est l'impact du choix du mode d'arrondi sur le résultat ?

Mode d'arrondi

Rappel

Choisir un mode d'arrondi, c'est choisir un représentant $X \in \mathbb{F}$ de $x \in \mathbb{R}$.
Il existe 4 modes d'arrondi.

Impact

Quel est l'impact du choix du mode d'arrondi sur le résultat ?

Quel est l'impact du choix du mode d'arrondi sur le résultat ?

1^{er} exemple

$$0.3x^2 + 2.1x + 3.675 = 0$$

Impact du choix de mode d'arrondi

Mode d'arrondi : au plus près

```
int main(){
  fesetround(FE_TONEAREST);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"Valeur de c : "<<endl;
  cin>>c;

  float delta = b*b - 4*a*c;
  cout<<endl<<"delta : "<<delta<<endl;

  return 0;
}
```

Impact du choix de mode d'arrondi

```
Valeur de a :
0.3
Valeur de b :
2.1
Valeur de c :
3.675

delta : -9.53674e-07
```

2 racines complexes

Impact du choix de mode d'arrondi

Mode d'arrondi : vers $+\infty$

```
int main(){
  fesetround(FE_UPWARD);

  float a, b, c;
  cout<<"Valeur de a : "<<endl;
  cin>>a;
  cout<<"Valeur de b : "<<endl;
  cin>>b;
  cout<<"Valeur de c : "<<endl;
  cin>>c;

  float delta = b*b - 4*a*c;
  cout<<endl<<"delta : "<<delta<<endl;

  return 0;
}
```

Impact du choix de mode d'arrondi

```
Valeur de a :
0.3
Valeur de b :
2.1
Valeur de c :
3.675

delta : 0
```

1 seule racine : 3.5

Quel est l'impact du choix du mode d'arrondi sur le résultat ?

2^e exemple

$$\sum_{i=1}^{n} 0.1$$

Impact du choix de mode d'arrondi

Mode d'arrondi : au plus près

```
using namespace std;

int main(){
        fesetround(FE_TONEAREST);
        float x;
        cout<<"Entrez la valeur de x"<<endl;
        cin>>x;
        float res = 0.0;
        for (int i=0;i<1000;i++){
                res+=x;
        }
        cout<<"Somme des x (1000 fois) :"<<setprecision(10)<<res<<endl;


        return 0;
}
```

Impact du choix de mode d'arrondi

```
 Entrez la valeur de x
0.1
Somme des x (1000 fois) : 99.99904633
```

Erreur : 1.10^{-3}

Quel est l'impact du choix du mode d'arrondi sur le résultat ?

Mode d'arrondi : vers $+\infty$

```
using namespace std;

int main(){
        fesetround(FE_UPWARD);
        float x;
        cout<<"Entrez la valeur de x"<<endl;
        cin>>x;
        float res = 0.0;
        for (int i=0;i<1000;i++){
                res+=x;
        }
        cout<<"Somme des x (1000 fois) :"<<setprecision(10)<<res<<endl;


        return 0;
}
```

Impact du choix de mode d'arrondi

```
 Entrez la valeur de x
0.1
Somme des x (1000 fois) : 100.0030442
```

Erreur : 3.10^{-3}

Et en changeant le mode d'arrondi (aléatoirement) à chaque calcul ?

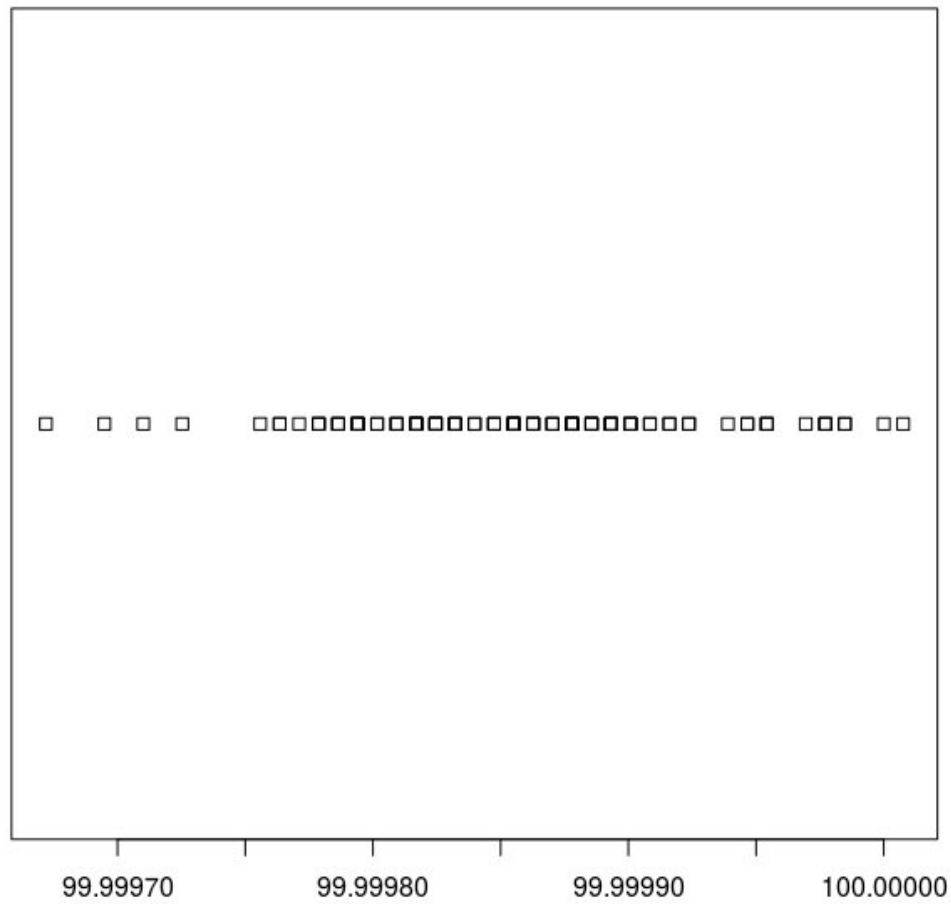
Et avec un mode d'arrondi aléatoire ?

```
void changement_mode_arrondi(){
  int v = rand()%4;
  if (v == 0) fesetround(FE_DOWNWARD);
  if (v == 1) fesetround(FE_UPWARD);
  if (v == 2) fesetround(FE_TONEAREST);
  if (v == 3) fesetround(FE_TOWARDZERO);
}

int main(){
        srand (time(NULL));
        float x=0.1;
        float res = 0.0;

        for (int nbEssai=0; nbEssai<100; nbEssai++){
          res = 0.0;
          for (int i=0;i<1000;i++){
            changement_mode_arrondi();
              res+=x;
              x=0.1;
          }
          cout<<setprecision(10)<<res<<endl;
        }
```

Et avec un mode d'arrondi aléatoire ?

résultat pour 100 essais

- Erreur : de 3.10^{-4} à 0 !
- Les résultats sont globalement meilleurs
- Il arrive que l'on trouve le bon résultat



Impact du choix de mode d'arrondi

Constats

- Le choix du mode d'arrondi (fixe pour toute l'exécution du programme) a un impact sur le résultat
- Changer le mode d'arrondi à chaque opération permet d'obtenir une valeur parmi l'ensemble des résultats possibles
- En ré-itérant, on obtient un échantillon de résultats possibles

C'est la base de l'arithmétique stochastique

arithmétique stochastique

Le but de l'arithmétique stochastique n'est pas d'améliorer la précision du résultat.
Il s'agit de faire propager différemment les erreurs d'arrondi pour, ensuite, à partir des échantillons du résultat, pouvoir estimer son nombre de chiffres significatifs exacts

Formalisation

Perte de précsion

Soit n le nombre d'opérations d'un calcul.
Soit R, le résultat obtenu ($R \in \mathbb{F}$) et r le vrai résultat ($r \in \mathbb{R}$)
Il a été montré que

$$R \approx r + \sum_{i=1}^{n} g_i(d).2^{-p}.\alpha_i + O(2^{-2p}) \quad (1)$$

Avec
p : le nombre de bits de la mantisse (y compris le bit implicite)
$g_i(d)$: coefficients dépendant uniquement des données
α_i : erreurs d'arrondis lors du i^e calcul

Nombre de chiffres significatifs

On montre ainsi que C_R, le nombre de chiffres significatifs exacts du résultat numérique R s'écrit

$$C_R = log_2|\frac{R-r}{r}| = p - log_2|\sum_{i=1}^{n} g_i(d).\frac{\alpha_i}{r}| \quad (2)$$

La perte de précision lors d'un calcul sur ordinateur est indépendante de la précision utilisée pour ce calcul

Estimation de la précision

Variable aléatoire

En exécutant N fois un calcul, on obtient donc N tirages de la variable aléatoire modélisée par $r + \sum_{i=1}^{n} g_i(d).2^{-p}.\alpha_i$

hypothèses

- Les α_i sont des variables aléatoires indépendantes identiquement distribuées (iid).
- La distribution commune des α_i est uniforme sur $[-1; 1]$ donc centrée.

Estimation de la précision

conséquences

- L'espérance mathématique de la variable R est le résultat mathématique exact r
- La distribution de R est quasi-gaussienne.

Il s'agit donc d'estimer la moyenne d'une variable aléatoire gaussienne à partir d'un échantillon (-> Test de Student)

$$\text{Ainsi } C_{\overline{R}} = log_{10}(\frac{\sqrt{N}.|\overline{R}|}{\sigma.\tau_\beta})$$

Avec
$\overline{R} = \frac{1}{N}.\sum_{i=1}^{n} R_i$ *et* $\sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{n}(R_i - \overline{R})^2$

τ_β est la valeur d'une distribution de Student's avec $N-1$ degrés de liberté et un niveau de probabilité β

Conséquences

- En pratique, 2 ou 3 exécutions suffisent.
- Pour augmenter la précision d'un bit, il faudrait multiplier la taille de l'échantillon par 100

Avec $\beta = 0.95$ *et* $N = 3$

- La probabilité de sur-estimer le nombre de bits exacts de 1 est de 0.00054
- La probabilité de sous-estimer le nombre de bits exacts de 1 est de 0.29

En choisissant un intervalle de 95%, on garantit un nombre de bits exacts avec une très grande probabilité (0.99946) (même si on est souvent pessimiste sur un bit).

L'arithmétique stochastique en pratique :

la bibliothèque Cadna

Cadna

Cadna

Cadna permet d'estimer la propagation des erreurs d'arrondis d'un programme scientifique.

- Estimation de la qualité numérique d'un résultat
- Contrôle des instructions de branchement pouvant propvoquer des pertes de précision
- Détection des cancellations et des absorptions

cf : http ://www.lip6.fr/cadna

En pratique

- Tous les réels (float et double) sont remplacés par des types de Cadna. Chaque réel est représenté par 3 valeurs différentes.
- Tous les opérateurs, toutes les fonctions sont surchargés. Les calculs se font donc tous dans Cadna.
- Pour chaque calcul, on change aléatoirement le mode d'arrondi.

Exemple d'utilisation de Cadna (1)

$$0.3x^2 + 2.1x + 3.675 = 0$$

Exemple d'utilisation de Cadna (1)

```
CADNA software --- University P. et M. Curie
--- LIP6
...
-----------------------------------------
d = @.0
Discriminant is zero.
The double solution is 0.349999E+01
-----------------------------------------
...
There are 1 numerical instabilities
0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE INTRINSIC FUNCTION(S)
1 UNSTABLE CANCELLATION(S)
```

Exemple d'utilisation de Cadna (1)

Commentaires

- L'arithmétique flottante ne détecte pas que $d = 0$.
  On chosit donc la mauvaise branche du programme.
  Le résultat est faux.
- Avec Cadna, l'imprécision sur d est telle que l'on ne peut pas le distinguer de la valeur 0.
  Le test *if d==0* est vrai et on choisit la bonne branche.
  Le résultat est correct.

Exemple d'utilisation de Cadna (2)

$$\sum_{i=1}^{n} 0.1$$

Exemple d'utilisation de Cadna (2)

Code avec Cadna

```
#include <cadna.h>

int main(){
        cadna_init(-1);

        float_st x;
        cout<<"Entrez la valeur de x"<<endl;
        cin>>x;
        float_st res = 0.0;
        for (int i=0;i<1000;i++){
                res+=x;
        }
        cout<<"Somme des x (1000 fois) :"<<res<<endl;

        cadna_end();

        return 0;
}
```

Exemple d'utilisation de Cadna (2)

```
CADNA_C 1.1.9 software ---
University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
-----------------------------------------
Entrez la valeur de x
0.1
Somme des x (1000 fois) : 0.100000E+003
-----------------------------------------
...
No instability detected
-----------------------------------------
```

On peut donc affirmer que le résultat est 100 avec une précision d'au moins 10^{-3}

Exemple d'utilisation de Cadna (3)

Suite de Muller

$$\begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_{n+1} = 111 - \frac{1130}{u_n} + \frac{3000}{u_n . u_{n-1}} \end{cases}$$

Suite de Muller

Code sans Cadna

```
int main(){

  double u[30];
  u[0] = 2;
  u[1] = -4;

  for(int i=2;i<30;i++){
    u[i] = 111. - 1130./u[i-1] + 3000./(u[i-1]*u[i-2]);
    cout<<"u["<<i<<"]"<<u[i]<<endl;
  }
  return 0;
}
```

Suite de Muller - sans cadna

```
u[2]18.5
u[3]9.37838
u[4]7.80115
u[5]7.15441
u[6]6.80678
u[7]6.59263
u[8]6.44947
u[9]6.34845
u[10]6.27444
u[11]6.2187
u[12]6.17585
u[13]6.14263
u[14]6.12025
u[15]6.16609
u[16]7.23502
u[17]22.0621
u[18]78.5756
u[19]98.3495
u[20]99.8986
u[21]99.9939
```

Suite de Muller

Code avec Cadna

```
#include <cadna.h>

int main(){

  cadna_init(-1);

  double_st u[30];
  u[0] = 2;
  u[1] = -4;

  for(int i=2;i<30;i++){
    u[i] = 111. - 1130./u[i-1] + 3000./(u[i-1]*u[i-2]);
    cout<<"u["<<i<<"]"<<u[i]<<endl;
  }
  cadna_end();
  return 0;
}
```

Suite de Muller - avec cadna

```
...
u[2] 0.18500000000000E+002
u[3] 0.9378378378378E+001
u[4] 0.7801152737752E+001
u[5] 0.715441448097E+001
u[6] 0.68067847369E+001
u[7] 0.6592632768E+001
u[8] 0.644946593E+001
u[9] 0.63484520E+001
u[10] 0.6274437E+001
u[11] 0.62186E+001
u[12] 0.6175E+001
u[13] 0.613E+001
u[14] 0.60E+001
u[15] @.0
u[16] @.0
u[17] @.0
...
```

Suite de Muller - avec cadna

```
...
u[17] @.0
u[18] @.0
u[19] 0.99E+002
u[20] 0.999E+002
u[21] 0.9999E+002
u[22] 0.99999E+002
u[23] 0.999999E+002
u[24] 0.99999999E+002
-----------------------------------------
....
There are 12 numerical instabilities
9 UNSTABLE DIVISION(S)
3 UNSTABLE MULTIPLICATION(S)
-----------------------------------------
```

PLAN

Validation, tests de non régression

Validation, tests de non régression

Contexte, objectifs

- Vérifier qu'il n'y a pas eu de dégradation/régression des fonctionnalités par rapport à la version précédente
- Vérifier que des modifications n'ont pas entraîné d'effets de bord, de nature à dégrader la qualité d'une version antérieurement validée

Lien avec la précision, la reproductibilité

- Cette validation repose sur la comparaison entre les résultats obtenus pour la version testée et les résultats *de référence*
- Mais comment comparer des résultats qui peuvent avoir été obtenus dans des environnements différents ?
- Faut-il être accepter une tolérance dans la comparaison ? Faut-il figer l'environnement ?

Lien utile

DEVELOP R6

Rencontre DevelopR6 sur les tests

Date et lieu

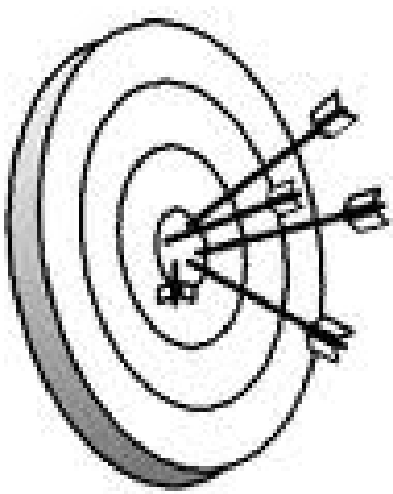
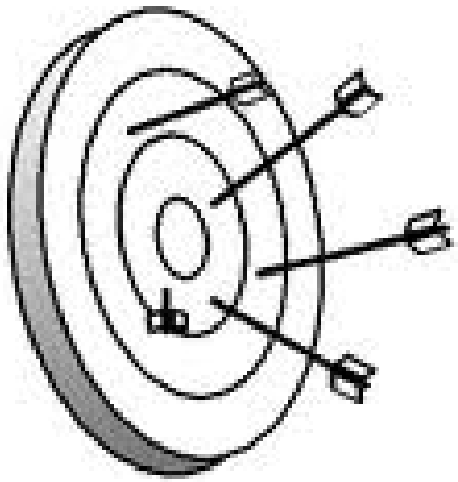
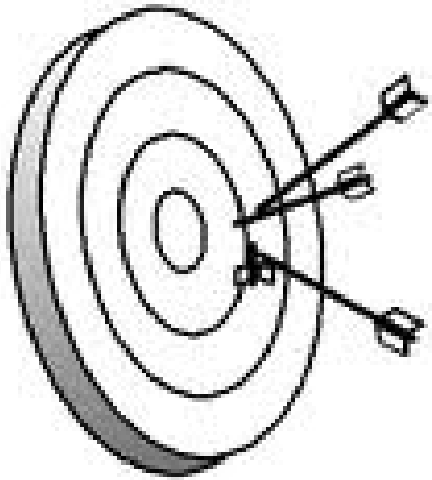
- Jeudi 5 juin 2014 de 10H à 17H
- Besançon, Centre Diocésain (20, rue Mégevand)

Programme

- 10H : Accueil - Café
- 10H25 : Présentation de la journée, le comité de pilotage
- 10H30 : Les tests dans la vie d'un logiciel, F. Bouquet
- 11H15 : Test unitaire C++ avec CppUnit, S. Diakité
- 11H40 : Retour d'expérience sur la bibliothèque fortran selalib, P. Navaro
- 12H10 : Repas
- 14H : Test et couverture de code Java avec JUnit et SonarQube, G. Vuidel
- 14H20 : Tests et résultats numériques : duo explosif ou indissociable ?, F. Langrognet
- 14H45 : Tests de non-régression dans le développement d'une application web graphique, C. Villa
- 15H10 : TDD par l'exemple, G. Billey
- 15H35 : Utilisation des tests dans le cadre du développement de la plateforme ISTEX, C. Niederlender
- 16H : Discussion & Café

http ://developr6.dr6.cnrs.fr/manifestations/007-tests/langrognet

Reproductibilité

Reproductibilité

Reproductibilité numérique

- L'environnement logiciel et matériel peut avoir un impact sur les résultats et donc sur leur reproductibilité
- Pouvoir reproduire les résultats d'un calcul est parfois indispensable (pour les reviewers par exemple)
- Les questions de reproductibilité et de précision sont différentes
  Un résultat peut être parfaitement reproductible mais totalement faux
- Faut-il être accepter une tolérance dans la reproductibilité ? Si oui, comment la définir ?

Reproductibilité

- La question de la reproductibilité des résultats de la recherche ne se limite pas à la capacité de reproduire les résultats numériques.
- La reproductibilité concerne toutes les disciplines (biologie, physique, ...).
- La reproductibilité est l'un des fondements de la recherche.

Lien utile

Ecole "précision et reproductibilité en calcul numérique"

http ://calcul.math.cnrs.fr/spip.php ?article220

Conclusion

Peut-on vraiment calculer avec un ordinateur ?

Conclusion

Les risques sont présents.

Mieux vaut les connaître.