# *Chiffres et nombres, ces inconnus auxquels on attribue tant de mérite!*

*Les grandes personnes aiment les chiffres. Quand vous leur parlez d'un nouvel ami, elles ne vous questionnent jamais sur l'essentiel. Elles ne vous disent jamais : Quel est le son de sa voix ? Quels sont les jeux qu'il préfère ? Est-ce qu'il collectionne les papillons ? Elles vous demandent : Quel âge a-t-il ? Combien pèse-t-il ? Combien gagne son père ? Alors seulement elles croient le connaître.*

*Le Petit Prince de Antoine de Saint-Exupéry*

*Les nombres ont-ils un mode d'existence en dehors de la tête de celui qui les pense ?*

*Hubert Reeves*

***Préambule :***

Suite à divers questions de collègues
- sur la divergence numérique lors des simulations de mécanique
- sur les transtypages automatiques dans les langages de programmation
- sur le bien fondé de tel ou tel représentation
....
il m'a semblé nécessaire de clarifier les idées sur les nombres et leur implémentation informatique.

Ces quelques pages sont de fait une compilation réécriture de documents glanés sur le net, agrémentés de touches personnelles. Elles tentent de répondre en partie aux diverses questions soulevées par l'utilisation des nombres en utilisant l'outil informatique.

# *Sommaire :*

# *Mais en fait qu'est-ce qu'un nombre ?*

## Définitions :

Un nombre est le représentant d'une quantité, c'est une représentation abstraite.

Un nombre peut s'écrire (s'afficher sous une forme compréhensible et imprimable) par une combinaison de chiffres (symboles représentant les nombres unitaires d'une base).

## Notion positionnelle de la représentation :

*"Les hommes sont comme les chiffres : ils n'acquièrent de valeur que par leur position."*

*Napoléon Bonaparte*

Dans les différent systèmes de numération ayant eu cours depuis la nuit des temps, nous opposerons deux systèmes : Le système romain et le système actuel. Le système romain est (tous les élèves vous le diront) très compliqué à lire et donc à coder. Il repose sur un processus additif et soustractif des symboles. Il impose un nombre de symbole de plus en plus important pour représenter les nombres de plus en plus grand. De plus, il n'a pas la notion de codage du zéro. Le système arabe (le nôtre), s'appuie sur un set de symbole réduit (autant que la base utilisée) représentant chacun une quantité simple (de un en un). Le poids de ces symboles dans la représentation du nombre est positionnel (habituellement le poids 0 se situe à droite et le poids le plus élevé à gauche de l'écriture) . La notion de quantité vide (le zéro) est représentable.
Un nombre est représentable par une succession de symboles $a_j$ (ex : E5F6)

$$n = \sum_{i>=0} a_j b^i \quad avec \begin{cases} b \text{ la base utilisée}, \\ a_j \text{ les chiffres de la base} \\ i \text{ la position du chiffre dans le nombre représenté} \end{cases}$$

Ex : Le contenu d'un poulailler (en poules) (il s'agit d'une quantité donc d'un nombre dans l'espace mathématique associé) peut s'écrire sous la forme 12 en base 10 (nombre exprimé en base 10 ) soit en utilisant la notation positionnelle des chiffres $1x10^1+2*10^0$ 1 et 2 étant des chiffres de la base.
Dans une autre base, possédant d'autre chiffres, la quantité de poules pourrait se représenter par : en base 2 , 1100 $\{=(1*2^3+1*2^2+0*2^1+0*2^{20})_{(10)}\}$ ; en base $16_{(10)}$, 0C ; en base $6_{(10)}$ , 20 $\{=2*6^1+2*6^0)_{(10)}\}$
L'écriture $1100_{(2)}=12_{(10)}=0C_{(16)}=20_{(6)}$ est donc valide car chaque expression représente la même quantité.

il est bon de rappeler que $\boldsymbol{nimportequoi^0=1}$ avec mais que $\boldsymbol{0^{nimportequoi}=0}$ avec ***nimportequoi*** une valeur réelle non nulle.
Nous avons parlé ici de nombre entiers. En ce qui concerne les nombres fractionnaire, la règle est la même sauf que maintenant nous pouvons parler de poids négatif qui

correspond à la position après la virgule.

$$d = \sum_{i>=0} a_j b^i + \sum_{i<0} a_j b^i \text{ avec } \begin{cases} b \text{ la base utilisée,} \\ a_j \text{ les chiffres de la base} \\ i \text{ la position du chiffre dans le nombre représenté} \end{cases}$$

$\sum_{i>=0} a_j b^i$ représente la partie entiere

$\sum_{i<0} a_j b^i$ représente la partie fractionnaire

par exemple $123.56=1*10^2+2*10^1+3*10^0+5*10^{-1}+6*10^{-2}$

## Représentations usuelles des bases :

En informatique, jusqu'alors, seuls quelques bases sont utilisées :
Base 2 : On représente les nombres en ajoutant un b à la fin de sa représentation Ex : 1100b
Base 8 : Octal On ajoute un o (à ne pas confondre avec un 0 à la fin de la représentation Ex 14o.
Base 10 : On ne rajoute rien cette base semble la base naturelle bien qu'elle soit la plus éloignée des systèmes électroniques Ex : 12
Base 16 : Hexadécimale. Plusieurs représentations sont en vogue
- en rajoutant h à la fin de l'écriture Ex : 0Ch
- en préfixant par $ la représentation Ex $0C
- en préfixant par 0x la représentation Ex : 0x0C

## Remarque sur les nombres :

Un nombre représentant une quantité, un système de numération doit pouvoir représenter aussi bien une quantité simple comme 12 œufs (12 et non 11 et non 13 !) comme 1/3 de tarte comme le rapport entre le diamètre d'un cercle et son périmètre pi() (que j'aime à faire apprendre un nombre utile aux sages ….). Force est d'avouer que notre système de numération ne fonctionne pas bien dans les deux derniers cas (surtout dans ce que nos mathématiciens qualifieront d'irrationnel). Ecrire $\pi = 3.1415962$ est manifestement faux mais représente une approximation suffisante pour bien des domaines (de même que $\pi = \frac{22}{7}$). Nous en resterons donc à un codage permettant une approximation suffisamment fine (permettant de négliger les erreurs commises devant la grandeur quantifiée) des quantités que nous manipulons.

# *Mais comment le code-t-on dans un ordinateur ?*

# Représentation d'un nombre dans un ordinateur :

On appelle représentation (ou codification) d'un nombre la façon selon laquelle il est décrit sous forme binaire. La représentation des nombres sur un ordinateur est indispensable pour que celui-ci puisse les stocker, les manipuler. Toutefois le problème est qu'un nombre mathématique peut être infini (aussi grand que l'on veut), mais la représentation d'un nombre dans un ordinateur doit être faite sur un nombre de bits prédéfini. Il s'agit donc de prédéfinir un nombre de bits et la manière de les utiliser pour que ceux-ci servent le plus efficacement possible à représenter l'entité. Ainsi il serait idiot de coder un caractère sur 16 bits (65536 possibilités) alors qu'on en utilise généralement moins de 256...

# Représentation d'un entier naturel :

Un entier naturel est un entier positif ou nul. Le choix à faire (c'est-à-dire le nombre de bits à utiliser) dépend de la fourchette des nombres que l'on désire utiliser. Pour coder des nombres entiers naturels compris entre 0 et 255, il nous suffira de 8 bits (un octet) car $2^8$=256. D'une manière générale un codage sur *n* bits pourra permettre de représenter des nombres entiers naturels compris entre 0 et $2^n$-1.

Pour représenter un nombre entier naturel après avoir défini le nombre de bits sur lequel on le code, il suffit de ranger chaque bit dans la cellule binaire correspondant à son poids binaire de la droite vers la gauche, puis on « remplit » les bits non utilisés par des zéros.

# Représentation d'un entier relatif :

Un entier relatif est un entier pouvant être négatif. Il faut donc coder le nombre de telle façon que l'on puisse savoir s'il s'agit d'un nombre positif ou d'un nombre négatif, et il faut de plus que les règles d'addition soient conservées. L'astuce consiste à utiliser un codage que l'on appelle *complément à deux*.

- **un entier relatif positif ou nul** sera représenté en binaire (base 2) comme un entier naturel, à la seule différence que le bit de poids fort (le bit situé à l'extrême gauche) représente le signe. Il faut donc s'assurer pour un entier positif ou nul qu'il est à zéro (0 correspond à un signe positif, 1 à un signe négatif). Ainsi si on code un entier naturel sur 4 bits, le nombre le plus grand sera 0111 (c'est-à-dire 7 en base décimale).

D'une manière générale le plus grand entier relatif positif codé sur *n* bits sera $2^{n-1}-1$.

- **un entier relatif négatif** grâce au codage en complément à deux.

Principe du *complément à deux* :
Soit à représenter un nombre négatif.

  - Prenons son opposé (son équivalent en positif)
  - On le représente en base 2 sur n-1 bits

- On complémente chaque bit (on inverse, c'est-à-dire que l'on remplace les zéros par des 1 et vice-versa)
- On ajoute 1

On remarquera qu'en ajoutant le nombre et son complément à deux on obtient 0...

Voyons maintenant cela sur un exemple :
On désire coder la valeur -5 sur 8 bits. Il suffit :

- d'écrire 5 en binaire : 00000101
- de complémenter à 1 : 11111010 (revient à effectuer l'opération 00000101 xor 1111111)
- d'ajouter 1 : 11111011 (11111010 + 00000001)
- la représentation binaire de -5 sur 8 bits est 11111011

***Remarques:***
Le bit de poids fort est 1, on a donc bien un nombre négatif.
Si on ajoute 5 et -5 (00000101 et 11111011) on obtient 0 (avec une retenue de 1...).

***Methode informatique :***
Pour obtenir le codage de l'opposé d'un nombre qu'il soit "positif" ou "négatif" il faut faire une opérations booléenne et une addition $-x \Leftrightarrow (x \oplus 0xFF) + 0x01$ ou $\oplus$ (xor) est le or exclusif bit à bit du mot de 8 bits ( on comprendra ici le signe $\Leftrightarrow$ par même représentation). On adaptera facilement pour les mots plus longs.

# Représentation d'un nombre réel :

Il s'agit d'aller représenter un nombre binaire à virgule (par exemple *101,01* qui ne se lit pas *cent un virgule zéro un* puisque c'est un nombre binaire mais *5,25* en décimale) sous la forme 1,XXXXX... * $2^n$ (c'est-à-dire dans notre exemple 1,0101*$2^2$). La norme *IEEE* définit la façon de coder un nombre réel.
Cette norme se propose de coder le nombre sur 32 ou 64 bits et définit trois composantes :

La suite de ce paragraphe se basera sur la représentation 32 bits.

- le signe est représenté par un seul bit, le bit de poids fort (celui le plus à gauche)
- l'exposant est codé sur les 8 bits consécutifs au signe
- la mantisse (les bits situés après la virgule) sur les 23 bits restants

Ainsi le codage se fait sous la forme suivante :
**seeeeeeeemmmmmmmmmmmmmmmmmmmmmm**

- le **s** représente le bit relatif au signe (1 bit)

- les **e** représentent les bits relatifs à l'exposant (8 bits)
- les **m** représentent les bits relatifs à la mantisse (23 bits)

Certaines conditions sont toutefois à respecter pour les exposants :

- l'exposant 00000000 est interdit
- l'exposant 11111111 est interdit. On s'en sert toutefois pour signaler des erreurs, on appelle alors cette configuration du nombre *NaN*, ce qui signifie *Not a number*
- Il faut rajouter 127 (01111111) à l'exposant pour une conversion de décimal vers un nombre réel binaire. Les exposants peuvent ainsi aller de -254 à 255

La formule d'expression des nombres réels est ainsi la suivante:

```
(-1)^S * 2^( E - 127 ) * ( 1 + F )
```

où:

- S est le bit de signe et l'on comprend alors pourquoi 0 est positif ( -1^0=1 ).
- E est l'exposant auquel on doit bien ajouter 127 pour obtenir son équivalent codé.
- F est la partie fractionnaire, la seule que l'on exprime et qui est ajoutée à 1 pour effectuer le calcul.

Voyons ce codage sur un exemple :
Soit à coder la valeur 525,5.

- 525,5 est positif donc le 1er bit sera 0.
- Sa représentation en base 2 est la suivante : 1000001101,1
- En normalisant, on trouve : $1,0000011011*2^9$
- On ajoute 127 à l'exposant qui vaut 9 ce qui donne 136, soit en base 2 : 10001000
- La mantisse est composée de la partie décimale de 525,5 en base 2 normalisée, c'est-à-dire 0000011011.
- Comme la mantisse doit occuper 23 bits, il est nécessaire d'ajouter des zéros pour la compléter :

00000110110000000000000

- La représentation du nombre 525,5 en binaire avec la norme IEEE est donc :

0 1000 1000 00000110110000000000000
0100 0100 0000 0011 0110 0000 0000 0000 (4403600 en hexadécimal)

Voici un autre exemple avec un réel négatif :
Soit à coder la valeur -0,625.

- Le bit s vaut 1 car 0,625 est négatif

- 0,625 s'écrit en base 2 de la façon suivante : 0,101
- On souhaite l'écrire sous la forme 1.01 x 2-1
- Par conséquent l'exposant vaut 1111110 car 127 - 1 = 126 (soit 1111110 en binaire)
- la mantisse est 01000000000000000000000 (seuls les chiffres après la virgule sont représentés, le nombre entier étant toujours égal à 1)
- La représentation du nombre 0,625 en binaire avec la norme IEEE est :

1 1111 1110 01000000000000000000000
1111 1111 0010 0000 0000 0000 0000 0000 (FF 20 00 00 en hexadécimal)

# Valeurs particulières :

Nous pouvons remarquer (mais ce n'est pas évident au premier abord) que certaines valeurs ne sont pas représentables par ce standard notamment le 0 et l'infini. Par convention, certaines combinaisons de bits (non normalisées) correspondent à ces valeurs.

- deux valeurs pour 0 : les deux signes, avec exposant=mantisse=0
  Soit
  **s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm**
  1 00000000 000……..0000000 = 0 moins = -0
  0 00000000 000……..0000000 = 0 plus = +0

  On remarquera tout de même que l'ajout bit à bit de deux nombres normalisés donne +0. Le codage du 0 a donc une certaine cohérence.

- infini positif: signe positif, exposant=1...1, mantisse=0
  **s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm**
  0 11111111 000……..0000000 = +∞

- infini negatif: signe negatif, exposant=1...1, mantisse=0
  **s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm**
  1 11111111 000……..0000000 = -∞

- NaN (not a number): signe indifférent, exposant=1...1, mantisse≠0
  2^23-1 possibilités de codage
  **s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm**
  x 11111111 xxxxxxxxxxxxx1x ……..xxxxxxxx = Nan

# Autre codages :

Il existe aussi des flottants codés sur 64 bits (double précision) avec le découpage suivant

- le **s** représente le bit relatif au signe (1 bit)
- les **e** représentent les bits relatifs à l'exposant (16 bits)
- les **m** représentent les bits relatifs à la mantisse (47 bits)

On adaptera facilement les règles du 32 bits au 64 bits. Ce dernier codage permet une plus grande précision et une étendue de codage plus importante.

Certains processeurs arithmétiques ont un codage en interne de 80 bits (série des 8xx87 1s, 15e,64m) mais souvent une conversion en 64 bits s'opérait en entrée sortie du processeur (les langages ne supportant pas la notion de double précision étendue). Ces processeurs utilisent un fpu câblé basé sur l'algorithme CORDIC (C.F. "Petite balade à la frontière des Mathématiques et du Numérique").

# *Conséquences de ces codages*

## Sur les opérations :

Les entiers : Les opérations sur les entiers positifs (non signés unsigned) sont les seuls opérations « naturelles » acceptées par les processeurs et sont exactes (si on ne dépasse pas la capacité de représentation de la quantité par un entier) Le problème d'enroulement des nombres

Les entiers relatifs : Prenons un exemple

- $-5_{(10)}$ représenté par un integer (int) sous le codage la représentation binaire de -5 sur 8 bits est 11111011
- $+8_{(10)}$ représenté par un integer (int) sous le codage la représentation binaire de +8 sur 8 bits est 00001000

  La somme « brute » (par le microprocesseur) de ces deux nombres :

  11111011 + 00001000 =00000011 et une retenue de 1 que l'on oubli

  Ce codage correspond à $+3_{(10)}$.

  En ce qui concerne la différence, c'est une autre histoire… mais retenons que +8-(-5) se code comme +8 plus valeur opposée de (-5) Il suffit donc de savoir convertir un nombre en son opposé (il suffit vous avez dit … mais nous avons déjà développé cela plus haut.)

  Vous avez compris, plus on s'éloigne du codage « naturel » du processeur, plus on remplace du hardware par du software.

Les réels (ou plutôt les nombres à virgule)

Je ne détaillerais pas ici les divers opérations, mais comme déjà dans les microprocesseurs (hors-mis les processeurs arithmétiques) la représentation de ces quantités n'est pas « naturelle », il est nécessaire de faire appel à des bibliothèques spécialisés de calcul sur ces nombres.

Exemple de la somme de deux flottants positifs:

Plusieurs opérations sont à faire

1 : rechercher le nombre ayant l'exposant le plus fort

2 : dé-normaliser le nombre a l'exposant le plus faible pour ramener les nombres au même exposant

3 : faire la somme des mantisses et traiter l'exposant en cas de débordement

4 : re-normaliser le nombre résultant

Je vous laisse imaginer la soustraction et la multiplication ! (sans parler de la division !!)

# Le problème d'enroulement des nombres :

Exemple : sur un codage sur 8 bits avec un octet 1111 1111 $_{(2)}$ + 1 = 0000 0000 $_{(2)}$ et 1 de retenue que l'on oublie. En résumé 255+1 = 0 troublant non !!

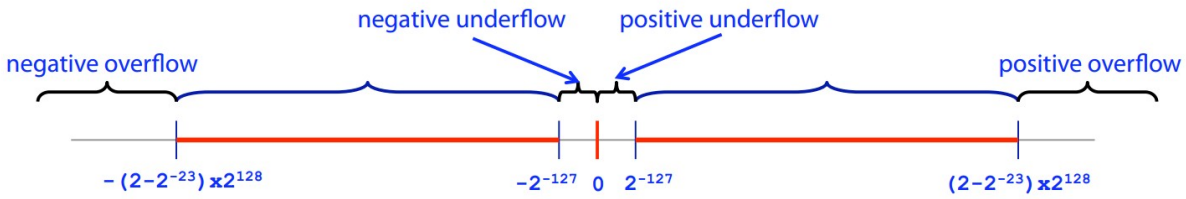
Ce type de débordement peut avoir des conséquences catastrophiques sur les calculs scientifiques et financiers (pensez à votre compte en banque !)

Dans les langages de haut niveau il est possible de gérer les débordements de nombres mais ces opérations sont couteuses en temps machine. Sur les programmes écrits dans des langages proches de la machine (souvent compilés ex C C++ C # assembleur) pour lesquels on cherche la rapidité, il incombe au programmeur de prendre en compte ces débordement et de les traiter en amont ou aval du calcul soit par un code de vérification avant de lancer le calcul, soit par une certitude (due au mécanisme numériquement simulé) que les valeurs que l'on va soumettre au programme permettent de garantir le non débordement lors du calcul.

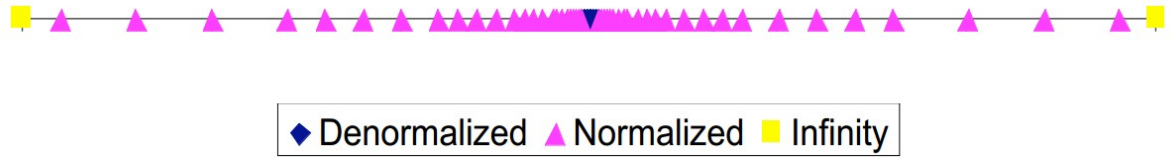
# Etendue et précision relative de ces nombres :

En ce qui concerne les entiers positif ou négatifs, la précision est de 1 unité dans le domaine de validité des nombres (exemple -128 à + 127 pour les octets signés 0 à 255 pour les octets non signés)

Nombre base et représentation informatique binairePour les flottants standards (32 bits) au niveau étendue nous avons le schéma suivant :



En ce qui concerne la densité de représentation nous nous contenterons du graphique ci-dessous établi dans un cas particulier de codage




Niveau exactitude de la représentation, Tout vas mal !! Rien n'est exact !! (enfin presque)

On peut facilement montrer que la précision absolue du codage dépend de l'ordre de grandeur du nombre (e) $a < 1 \bullet 2^{-23} \bullet 2^{e} = 2^{e-23}$ et la précision relative p oscile en dent de scie entre deux valeurs $0 < p < \frac{1 \bullet 2^{-23}}{2^{1}} = 2^{-24} = 5.9 \bullet 10^{-8}$

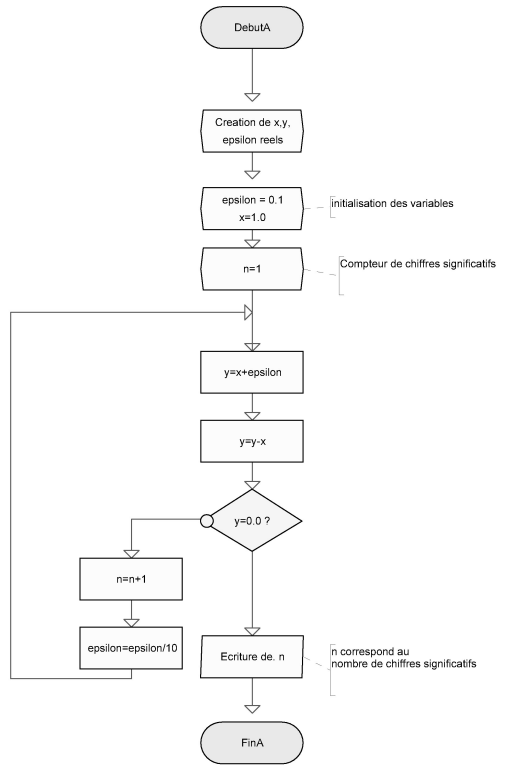
Nota pour un double precision nous obtenons :

la précision absolue du codage $a < 1 \bullet 2^{-53} \bullet 2^{e} = 2^{e-53}$ et la précision relative p est telle que $0 < p < \frac{1 \bullet 2^{-53}}{2^{1}} = 2^{-54} = 5.5 \bullet 10^{-17}$

Le tableau ci apres résume les caractéristiques des types de données les plus classiques

| Type pascal | Type C | Nb byte | e | m | Precision digit (10) | min max | Precision relative |
|---|---|---|---|---|---|---|---|
| Real (turbo pascal) | *Non reconnu* | 6 | 10 | 37 | 11 | $2.9*10^{-39}$ à $1.7*10^{+38}$ | $3.63*10^{-12}$ |
| Single | float | 4 | 8 | 23 | 8 | $3.4*10^{-38}$ à $3.4*10^{38}$ | $5.9*10^{-8}$ |
| Double | double | 8 | 11 | 52 | 16 | $1.7*10^{-308}$ à $1.7*10^{308}$ | $5.5*10^{-17}$ |
| Extended | long double | 10 | 15 | 64 | 20 | $3.4*10^{-4932}$ à $3.4*10^{4932}$ | $2.7*10^{-20}$ |

Un petit algorithme sur votre machine pour déterminer le nombre de chiffres significatifs :




En pseudo code

```
x :=1.0
e :=0.1
n :=1
repeter
      y :=x+e
      y :=y-x
      n :=n+1
      e :=e/10
jusqu'à y=0

afficher n-1
```

sur TI 83

```
EffÉcran
1.0→X
0.1→E
1→N
Lbl 1
X+E→Y
Y-X→Y
If Y=0
Then
Goto 2
End
N+1→N
Output(1,1,"N :
Output(1,4,N
E/10→E
Goto 1
Lbl 2
Pause
EffÉcran
```

L'algorithme proposé ci dessus n'est pas unique, nous en proposons un autre plus concis avec son codage en C

En pseudo code

```
x :=1.0
e :=0.1
n :=1
repeter
      n :=n+1
      e :=e/10
jusqu'à (x=(x+e))
afficher n-1
```

```
// En C, C++
float x = 1.0;
float e = 0.1;
int n =1;
do
  {
   n++;
   e /=10;
  }
while (x!=(x+e))

printf(n-1);
```

Il est facile d'implementer ces algorithmes sur un tableur. Avec "Excel", nous trouvons 15 chiffres significatif, ce qui signifie que Microsoft utilise la représentation double dans son tableur. Comme Excel est écrit en C++, on s'en serait douté.

Sur la TI83, on trouve 13 chiffres significatifs

# Problèmes relatifs aux algorithmes utilisés :

Dans cette courte partie, nous ne mettrons pas en place une grande théorie sur la stabilité des calculs, mais nous aborderons les problèmes de précision et de justesse des calculs sur un exemple qui a fait couler beaucoup d'encre depuis des siècles.

Le problème mathématique très ancien connu sous le nom de problème de Bâle a été d'évaluer les sommes suivantes :

$$Sr_1 = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \ldots\ldots \frac{1}{n^2}$$

$$Sr_2 = \frac{1}{n^2} + \frac{1}{(n-1)^2} + \ldots . \frac{1}{3^2} + \frac{1}{2^2} + \frac{1}{1^2}$$

et leurs limites pour $n \to \infty$.

Euler a démontré que $\sum_{n=1}^{+\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \approx 1.644934177398681640$

Mathématiquement, de par la commutativité de l'opération +, nous devions avoir $Sr_1 = Sr_2$ or (comme aurait dit 'Brassens' )« la suite lui montra que non ».
Le programme en C écrit pour « arduino » ci-dessous

```
// Euler prog
// ***********
double resultat = double(PI*PI / 6);
// résultat exact suivant "Euler"

void setup()
{
 Serial.begin(9600);
}

float Sr1;
float Sr2;

void loop()
{ int a;
 Serial.println("type g to begin");
 do
 {
  a = Serial.read();
 }
 while (a != 'g');
  for (unsigned long maxi = 10; maxi < 1000000000; maxi = maxi * 10)
  {
   Sr1 = 0;
   Sr2 = 0;
   for (unsigned long i = 1; i < maxi; i++)
    {
        Sr1 = Sr1 + 1.0 / (float(i) * float(i));
        Sr2 = Sr2 + 1.0 / (float(maxi - i) * float(maxi - i));
    }
   }
    Serial.print("maximum : "); Serial.println(maxi);

   Serial.print("Sr1 : ");   Serial.println(Sr1, 9);
   Serial.print("Sr2 : ");   Serial.println(Sr2, 9);
   Serial.print("resultat exact ");
   Serial.println(resultat, 18);
   Serial.println();

  }

}
```

Fourni les résultats suivants :

| maxi | Sr1 | Sr2 |
|---|---|---|
| 10 | 1.539767742 | 1.539767742 |
| 100 | 1.634883975 | 1.634883880 |
| 1000 | 1.643933868 | 1.643933534 |
| 10000 | 1.644725322 | 1.644834041 |
| 100000 | 1.644725322 | 1.644924068 |
| 1000000 | 1.644725322 | 1.644932985 |
| 10000000 | 1.644725322 | 1.644933938 |
| 100000000 | 1.644725322 | 1.644934082 |

résultat exact : 1.644934177398681640

Outre le fait (connu de longue date) que la série converge très lentement, nous constatons que suivant le sens de parcours du calcul (du plus grand au plus petit Sr1 ou du plus petit au plus grand Sr2), nous n'obtenons pas le même résultat. De plus, Sr1 semblent converger vers une valeur fausse. Une étude un peu plus fine nous montre qu'au delà de n=4096, la somme Sr1 n'évoluent plus. $1/(4096)^2 = 5.96 \bullet 10^{-8}$ or $5.96 \bullet 10^{-8} / 1.644 = 3.62 \bullet 10^{-8}$ ce chiffre est inférieur à la précision relative des nombre réels (déclarés en réel simple précision : float). L'ajout de ce terme à la somme initiale n'est donc pas pris en compte. Dans le cas du calcul de Sr2, on ajoute en permanence un terme du même ordre de grandeur que la somme déjà calculée. Les erreurs de calculs dues à l'éviction de nombre trop petit disparaissent alors. Il ne reste que les erreurs d'arrondis.
Il nous a été possible ici de concocter un algorithme plus correct que l'algorithme trivial car nous connaissons déjà beaucoup d'éléments mathématiques sur la suite en question. Dans "la vraie vie", lors des simulations numériques, le résultat est inconnu et le modèle pris en compte s'adapte à toutes les configurations matérielles et numériques. Il en résulte une possibilité d'erreur non négligeable lors des calculs successifs et/ ou des résolutions d'équations multiples. Des techniques de limitations d'erreur existent mais il s'agit de limitation et non d'élimination.

# *Et les sacrosaints ensembles $\mathbb{N}$ $\mathbb{Z}$ $\mathbb{D}$ $\mathbb{R}$ des maths, ou sont ils dans tout cela ?*

Faisons tout de suite « la peau » aux décimaux. Vous l'avez compris, de par son nom, les décimaux on comme point de départ la base 10. D est l'ensemble des nombres sous la forme $X/10^k$. Comme dans nos "machines à compter" la base est de deux, les décimaux n'ont pas lieu d'être car 10 n'est pas une puissance de 2. Nous pourrions imaginer un ensemble de nombre représentable sous la forme $X/2^k$ qui lui, pourrait être représenté dans nos machines binaires mais est-ce vraiment utile?

L'étendue de représentation d'une quantité étant limité par le nombre de bits du codage, aucun des ensemble restant ne peut exister en informatique au mieux il s'agit de sous-ensembles bornés des ensembles de base.

Le problème de la granulométrie de représentation (non continuité de la représentation des quantités) nous amène à une représentation perlée (ou le vide représente beaucoup plus que la présence) d'un sous ensemble des réels.

En bref, notre système de représentation est une scorie devant l'univers des nombres possibles, mais a le mérite de nous permettre généralement de représenter avec une erreur acceptable les quantités que nous manipulons

Nous noterons

N* le sous ensemble de N représentable par notre codage (les entiers positifs)

Z* le sous ensemble de N représentable par notre codage (les entiers relatifs)
B* le sous ensemble de R représentable par notre codage (des Binomaux ** ? par comparaison aux décimaux ) décomposable comme suit :

$\sum a_i 2^i$ *avec* $i \in [r,l], r-l < 24$ (*nb de bits de la mantisse*), $r >= -127, l <= 127$

Les plus grosses différences entre ces ensembles et leurs "homologues" mathématiques sont qu'ils sont bornés et dénombrables.

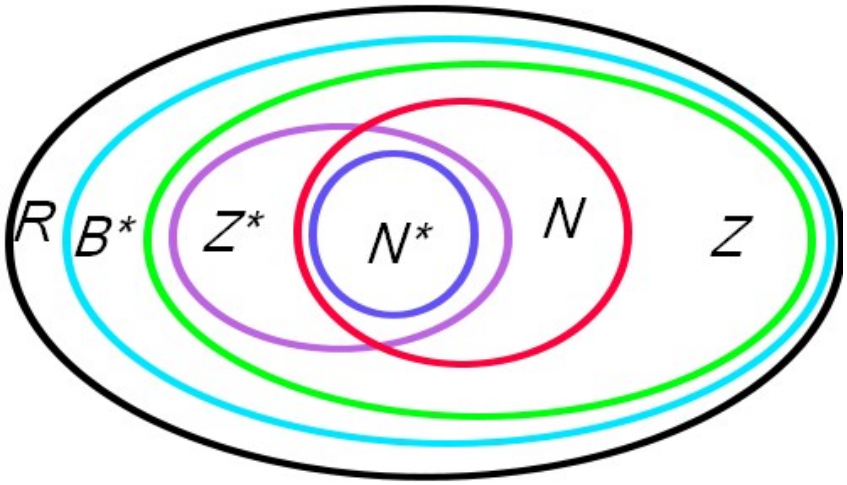
$N^* \subset Z^* \subset B^*$

$N^* \subset N$

$Z^* \subset Z$

$B^* \subset R$

Et pour ceux qui préfèrent les dessins aux longs discours :




** Les *Binormaux* ne sont pas une dénomination reconnue par les mathématiciens, mais il fallait bien donner un nom à cet ensemble !!

# *Bibliographie*


http://www.commentcamarche.net/contents/100-representation-des-nombres-entiers-et-reels

http://lslwww.epfl.ch/pages/teaching/cours_lsl/intro/4.Reels.pdf

http://e-ressources.univ-avignon.fr/assembleur/co/11_3.html

https://fr.wikipedia.org/wiki/Problème_de_Bâle

https://en.wikipedia.org/wiki/X87



Auteur : Yves Cordier, DDFPT Lycée Altitude, 3 rue Marius Chancel, 05100 Briançon