Calcul numérique sur ordinateur

Christophe Jermann*

Journée académique de l'IREM PdL - 13/04/2023

DPT INFO Département Informatique de Nantes Université Faculté des Sciences et des Techniques

Nantes Université

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

** exposé conçu en lien avec un fascicule éponyme, réalisé avec Frédéric Goualard, disponible sur le site de l'IREM*

Au cœur d'outils pédagogiques (et du quotidien)...

python™

SCRATCH

Java, javascript,
C/C++, …

LibreOffice

Google
Sheets

Excel

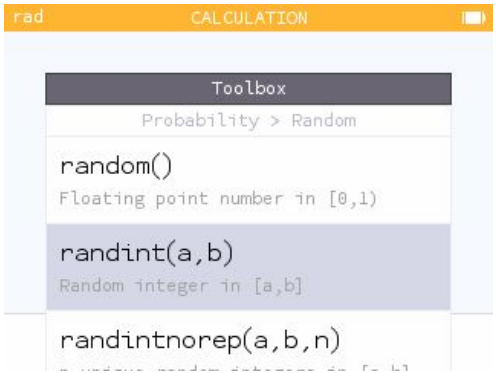
GeoGebra

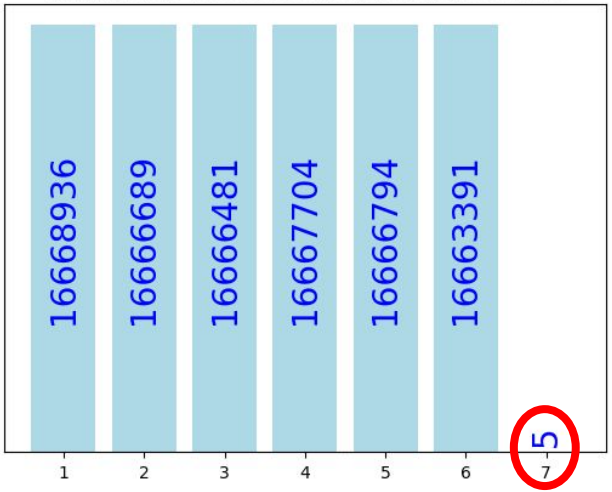
Maple™

MATLAB

… à la fiabilité questionnable…

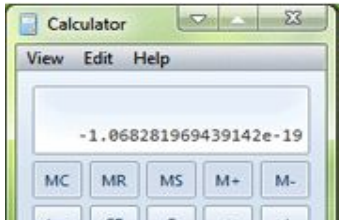
Exemple : résultat de cent millions de lancers de dé à 6 faces au moyen de la calculatrice Numworks[1] … quelque chose cloche ?





[1] résultat similaire reproductible avec Scratch, Snap, et d'autres outils et langages de programmation.

… avec des impacts importants (voire catastrophiques) !

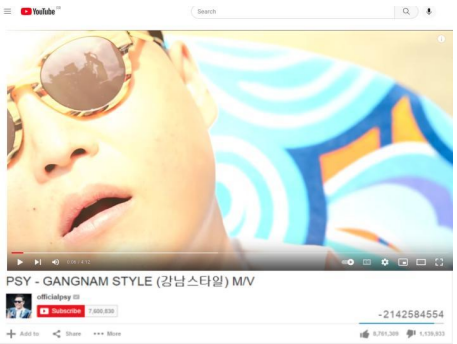


Calculatrice Windows
(2001-2018)

Missiles Patriot
(1991)

Intel Pentium IV
(1994)

Paypal
(2013)



Compteur de vues Youtube
(2014)

Premier vol Ariane 5
(1996)

Bug de l'an 2000
et bientôt, 2038 !

Pourquoi cela ?

Calcul mathématique = ensembles (et nombres) **infinis**

VS

Ordinateur = machine **finie**

Exploiter *au mieux* les possibilités des ordinateurs = **coder** l'information (et les opérations) aussi **efficacement** que possible (espace mémoire, temps de calcul)

⇒ Le codage numérique n'a pas les propriétés mathématiques des ensembles (partiellement) représentés

Le calcul numérique sur ordinateur[2] obéit à ses propres règles !

[2] ou calculatrice, tablette, smartphone, …

Le langage des ordinateurs

Langage des ordinateurs = le **binaire**

1 bit (**b**inary dig**it**) = 2 valeurs, **0** et **1**.

Chaîne binaire = séquence de bits (exemple : 00100111)

Unité typique d'information = octet = 8 bits

⇒ Toute donnée informatique (nombre, texte, image,…) est une chaîne binaire

Exemple : que représente 00110001 00110011 01000010 00110010 ?
Quatre entiers *(49 51 66 50)* ? Un entier *(825 442 866)* ? Une date Unix *(27/02/1996 17:41:06)* ? Un *réel* *(~2.608×10^{-9})* ? Quatre caractères ASCII *(1 3 A 2)* ? Deux caractères UTF16 *(ビル(自))* ?

Codage “numérique”

Pour **interpréter** une chaîne binaire, il faut un **codage**, souvent associé à une **taille**.

- Codage : interprétation d’une chaîne binaire comme un nombre, un texte, une image, …
- Taille : nombre de bits (ou d’octets) à considérer pour (dé)coder une chaîne binaire
  ⚠️Taille ⇒ nombre de valeurs possibles (exemples : 8 bits → 256 chaînes binaires distinctes ; 16 bits → 65536 chaînes ; N bits → 2^N chaînes)

Exemples :

- texte : ASCII (8bits), UTF-8/16/32 (8/16/32 bits), …
- image : RGB (24 bits/pixel), RGBA (32 bits/pixel), …

Nombres : le problème de base

En France (et dans de nombreux pays), les nombres sont notés au moyen des chiffres arabes (0,1,2,3,4,5,6,7,8,9) selon une **notation positionnelle en base 10**
Exemples :

- $6438 = 6\times10^{3} + 4\times10^{2} + 3\times10^{1} + 8\times10^{0}$
- $-1{,}4142\ldots = -1\times10^{0} + 4\times10^{-1} + 1\times10^{-2} + 4\times10^{-3} + 2\times10^{-4} + \ldots$

Pour faire du calcul numérique sur ordinateur, il faut donc **convertir la notation décimale en binaire** (base 2), mais cela induit **des arrondis** :

- dec→bin : certaines notations décimales finies sont codées en notations binaires trop grandes pour l'ordinateur, voire infinies *(on en reparlera)*
- bin→dec : l'affichage décimal d'un nombre binaire stocké dans l'ordinateur peut être inexact (tronqué/arrondi)

⇒ Il ne faut pas croire tout ce que l'on voit à l'écran !

Les entiers

Entiers mathématiques VS informatiques

En mathématiques :

- Naturels ($\mathbb{N}$) ou Relatifs ($\mathbb{Z}$), ensembles infinis dénombrables
- Munis des opérations arithmétiques usuelles (+, −, ×, ÷)
  - Particularités : la soustraction doit rester dans le domaine de définition pour les Naturels ; la division fournit deux résultats, un quotient et un reste

En informatique :

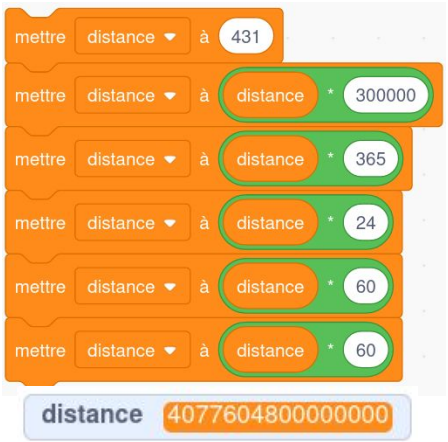
- type signé ou non-signé, de taille fixe (e.g., 32 bits) ou variable *(≠ infinie)*
  ⇒ ensembles **finis**, codages **multiples**
- Munis des opérations arithmétiques par extension du **calcul booléen**

Un premier calcul sur entiers

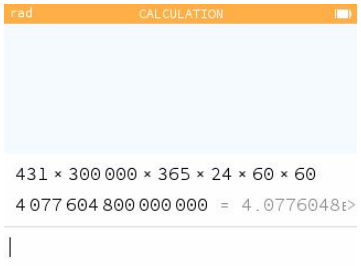
On veut convertir en kilomètres la distance du Soleil à l'Étoile polaire (Alpha Ursæ Minoris) qui vaut approximativement 431 années-lumière.

- Mathématiquement :
  431 al × 300 000 km.s^{-1} × 365 j × 24 h × 60 min × 60 s = 4 077 604 800 000 000 km
- Informatiquement :

Scratch



Numworks



Python

```
>>> 431*300000*365*24*60*60
4077604800000000
```

C/C++

```
int al2km(int d) {
    return d * 300000 * 365 * 24 * 60 * 60;
}
```

```
code$ g++ distance.cpp -o distance.exe
code$ ./distance.exe
 1503883264 ?!?
```

LibreOffice

al	km/s	j	h	m	s	km
431	300000	365	24	60	60	**4077604800000000**

Un autre !

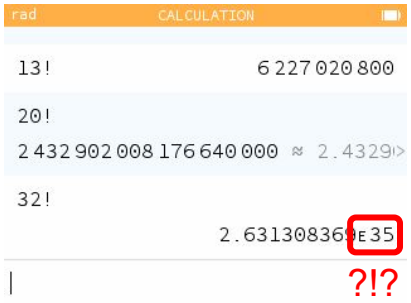
On veut calculer le nombre de donnes possibles à la belote, c-à-d. le nombre de permutations d'un paquet de 32 cartes.

- Mathématiquement : la fonction factorielle fait cela très bien ! :-)
  32! = 263 130 836 933 693 530 167 218 012 160 000 000 *(~2.63×10^35)*
- Informatiquement :

Scratch

?!?

Numworks



?!?

LibreOffice

N	N !
10	3628800
20	2432902008176640000
32	263130836933694000000000000000000000

?!?

Python

```
>>> from math import factorial
>>> factorial(32)
263130836933693530167218012160000000
```

C/C++

```
int factorial(int N) {
    if (N<2) return 1;
    else return N*factorial(N-1);
}
```

```
exposé$ ./fact.exe
 12 : 479001600
 13 : 1932053504
 32 : -2147483648
```

!?!?!?

Mais qu'est-ce qui se passe ?

Quelques éléments d'explication avant d'entrer dans le détail :

- Python, Numworks : représentation des entiers à **taille variable**
  + représentation exacte de très grands entiers ; opérations exactes
  – espace mémoire et temps de calcul importants

- Scratch, LibreOffice *(et de nombreux autres outils)* : **pas de représentation des entiers**, utilisation de *"réels"* ± proprement arrondis à l'affichage

- C/C++ *(et Numpy, et beaucoup de langages de programmation)* : représentation des entiers à **taille fixe**
  + représentation exacte ; espace mémoire et temps de calcul optimisés
  – bornes sur les entiers représentables assez *petites* ; calcul à maîtriser

Entiers non-signés à taille fixe

- Tailles usuelles = 8, 16, **32**, 64, 128 bits
  ⇒ borne max = 255, 65 535, **4 294 967 295**, 18 446 744 073 709 551 615 *($<2\times10^{19}$)*, 340 282 366 920 938 463 463 374 607 431 768 211 455 *($<4\times10^{38}$)*
  ⚠ Nombres pas si *"astronomiques"* !!! *(cf. exemples précédents)*

- Codage "naturel" = positionnel en base 2

	Codage $D_{10}\rightarrow B_2$	Décodage $B_2\rightarrow D_{10}$
Principe	$D \div 2 \xrightarrow{\text{quotient}} D' \div 2 \rightarrow \ldots \rightarrow 0$ $\downarrow$ reste $b_0 \; . \; b_1 \ldots b_k = \overleftarrow{B}$	$B \div 2 \xrightarrow{\text{quotient}} B' \div 2 \rightarrow \ldots\ldots\ldots \rightarrow 0$ $\downarrow$ reste $b_0\times2^0+b_1\times2^1+\ldots+b_k\times2^k = D$
Exemples :	$29 \rightarrow 14 \rightarrow 7 \rightarrow 3 \rightarrow 1 \rightarrow 0$ $1 \quad 0 \quad 1 \quad 1 \quad 1 \Rightarrow B=11101$	$11011 \rightarrow 1101 \rightarrow 110 \rightarrow 11 \rightarrow 1 \rightarrow 0$ $1 + 2 + 0 + 8 + 16 \Rightarrow D=27$

Remarque : k ≥ taille ⇒ *non représentable*

Opérations sur ce codage binaire “naturel”

Addition, soustraction, multiplication et division réalisables directement sur le code binaire.

Exemple : addition bit à bit *avec retenues*

b + b’ + retenue précédente = somme + retenue suivante

b	b’	p	s	r
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Illustration :

1 0 1 0

\+ 1 0 1 1

\+ 1 0 1 0 *retenues*

= 1 0 1 0 1

(en décimal : 10 + 11 = 21)

Exception lors d'un calcul

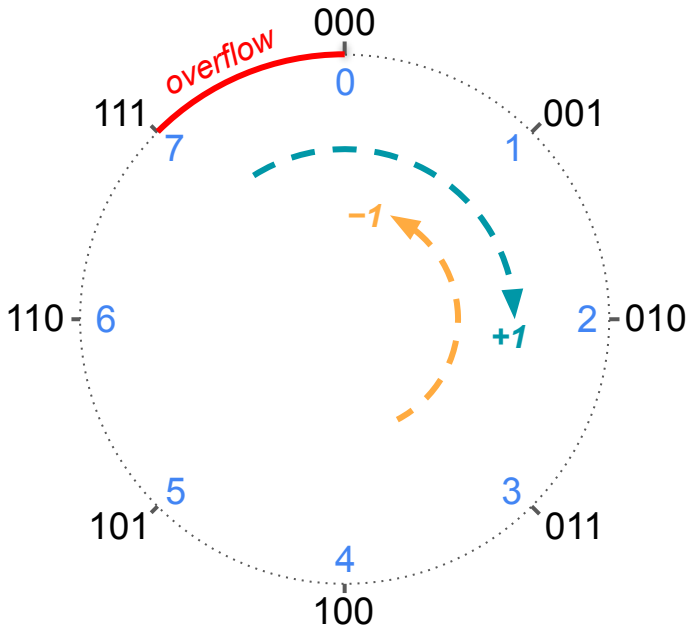
Si le résultat dépasse la borne de taille, on parle de **dépassement de capacité** *(overflow)* : le résultat obtenu est incorrect *(mais l'ordinateur mémorise la situation)*

Exemples : sur 4 bits

1 1 1 1	0 0 0 0	1 0 1 1
+ 0 0 0 1	- 0 0 0 1	× 0 0 1 1
1 1 1 1 *(retenues)*	1 1 1 1 *(retenues)*	1 0 1 1
= 0 0 0 0	= 1 1 1 1	1 0 1 1
		= 0 0 0 1
À l'écran : 15 + 1 = 0	*0 - 1 = 15*	*11 × 3 = 1*

Une arithmétique modulaire !

Le comportement des entiers non-signés peut-être visualisé de façon modulaire :

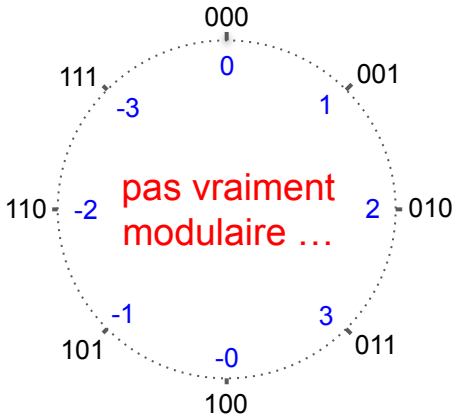


Et les signés ?

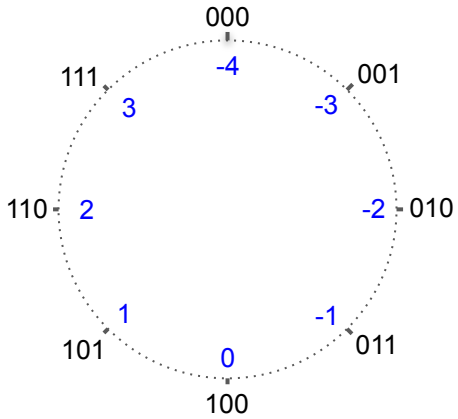
Plusieurs codages possibles :

- bit de signe *(0=positif, 1=négatif)* + codage “naturel”
- codage “naturel” + biais pair ou impair
- complément à 1, ou à 2

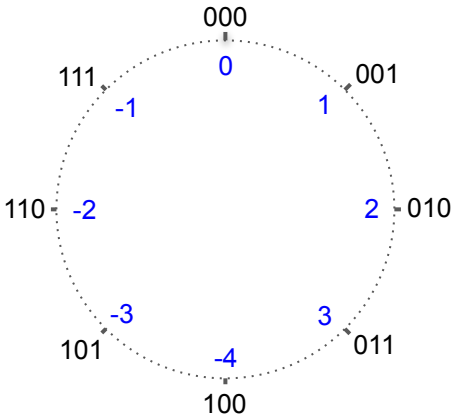
avec bit de signe



avec biais pair



en complément à 2



⚠️ Dans tous les cas, | borne inférieure | ≈ borne supérieure ≈ 2^k pour une taille k

Complément à 2 : propriétés et opérations

- Codage efficace :
  - L'opposé d'un nombre binaire B est ß+1 (ß inverse binaire *(0↔1)* de B)
    ⚠️ L'opposé du plus petit nombre (codé $100\ldots0_2$) n'est pas représentable
  - Le bit de poids fort donne le signe
  - Zéro représenté de façon unique
  - Nombre maximum de valeurs représentées

```
>>> from numpy import int8
>>> int8(127)+int8(1)
-128
>>> int8(-128)-int8(1)
127
>>> abs(int8(-128))
-128
```

- Opérations efficaces :
  - addition et soustraction "naturelles"
  - multiplication quasi-"naturelle" *(vérification de signe)*
  - une division binaire efficace est possible

Retour sur les exemples

- Entiers de taille fixe *(C/C++)* : l'overflow explique le résultat surprenant.
  C'est plutôt la norme (pour des raisons d'efficacité) ⇒ Attention !

- Entiers de taille variable *(Numworks, Python)* : les problèmes sont identiques, mais repoussés à des résultats faramineux *(e.g., 2^{8192} sur Numworks)*
  Réservé à des usages particuliers car inefficace

- *"Réels"* *(Scratch, LibreOffice)* : le nombre de chiffres représentable conditionne l'exactitude du résultat.
  Mathématiquement $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$, mais informatiquement non

Les réels

Réels mathématiques VS informatiques

En mathématiques :

- Décimaux ($\mathbb{D}$), Rationnels ($\mathbb{Q}$) ensembles infinis dénombrables
- Réels ($\mathbb{R}$) ensemble infini indénombrable *(irrationnels : π, $\sqrt{2}$, …)*
- Munis des opérations arithmétiques usuelles (+, −, ×, ÷) et de fonctions élémentaires (racines, logarithmes, trigonométrie,...)

En informatique :

- Virgule fixe ou flottante, de taille fixe (e.g., 32 bits)
  ⇒ ensembles **finis**, codages **multiples**
- Opérations arithmétiques *(usuellement)* par extension du calcul booléen *(hardware)*, fonctions élémentaires par approximation *(software)*

Tracé de polygones

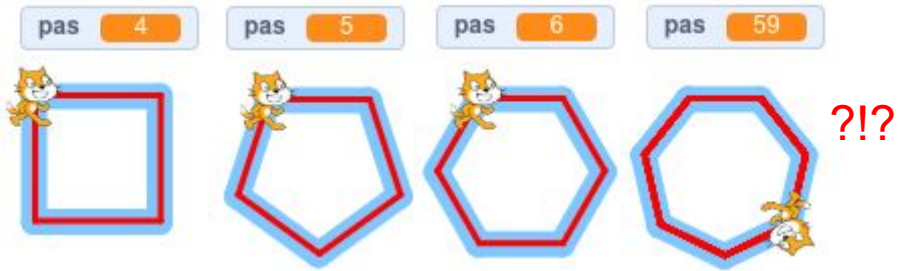
On veut tracer un polygone à N côtés selon deux méthodes :

– “répéter X fois” : aller au premier sommet ; Répéter N fois : tracer un côté, tourner de 360/N degrés ;

– “répéter jusqu’à” : aller au premier sommet ; tracer le premier côté ; Répéter jusqu’à être revenu au premier sommet : tracer un côté, tourner de 360/N degrés ;

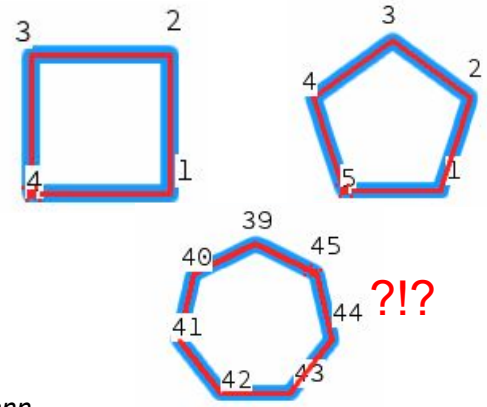
Mathématiquement : c’est équivalent

Scratch



Informatiquement : pas toujours…

Numworks/Turtle



Triangle manquant d'aire

On veut calculer l'aire A d'un triangle isocèle de côtés a, b, c qui se "dégonfle", devenant de plus en plus plat (a=b,c→0) en utilisant la formule de Héron :
$A = \sqrt{s\times(s-a)\times(s-b)\times(s-c)}$ avec $s=(a+b+c)/2$

Mathématiquement : étant donné les propriétés du triangle, on peut réécrire la formule générale* en $A = c/4\times\sqrt{4a^2-c^2}$, formulation rigoureusement équivalente

Informatiquement : on calcule avec les deux formules pour $a=b=10^6$ et $c=10^{-k}$

LibreOffice

k	0	1	2	3	4	5	6	7	8	9	10
Héron	500000	50000	5000	500	50,000031	5,0000381	0,5000038	0,0499422	0,0050059	0	0
Simplifiée	500000	50000	5000	500	50	5	0,5	0,05	0,005	0,0005	5E-05

Python

```
exposé$ python triangle.py
k         Héron                    Simplifiée
0         499999.9999999375        499999.9999999375
1         50000.00004656606        49999.99999999994
9         0.0004656612873077392    0.0005000000000000001
10        0.0                      5.0000000000000016e-05
```

* Kahan, W. (2014). *Miscalculating area and angles of a needle-like triangle.*

Sprint en série

Si Achille couvre à chaque instant la moitié de la distance le séparant de la ligne d'arrivée, combien de temps son sprint durera-t-il ?

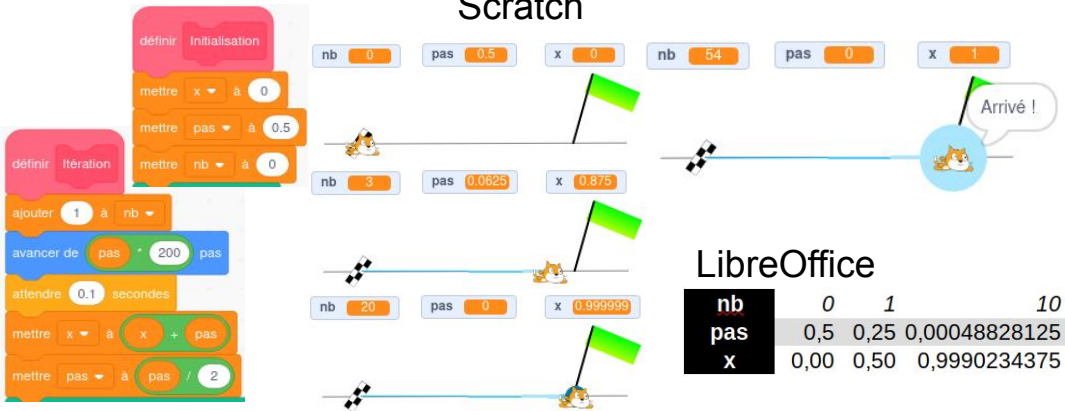
Mathématiquement : Cette série bien connue converge vers 1 … à l'infini !

$u_n = u_{n-1}/2$, $u_0 = ½$: suite géométrique de raison ½ ⇒ terme général : $u_n = ½^{n+1}$

somme partielle : $S_n = \Sigma_{k=0..n}\, ½^{k+1} = ½\,(1-½^{n+1})\,/\,(1-½) = 1 - 2^{-n} < 1$ *pour tout* n ; $lim_{n\to\infty} S_n = 1$

Informatiquement :

Scratch



Python

```
>>> x, pas, nb = 0, 0.5, 0
>>> while x<1:
...     x, pas, nb = x+pas, pas/2, nb+1
...
>>> print(x,pas,nb)
1.0 2.7755575615628914e-17 54
```

LibreOffice

nb	0	1	10	25	50	51
pas	0,5	0,25	0,00048828125	1,49011611938477E-08	4,4408920985006E-16	2,2204460492503E-16
x	0,00	0,50	0,9990234375	0,999999970197678	0,999999999999999	1

Début d'explication

- Pour le calcul réel, les nombres à **virgule flottante** sont utilisés par la majorité des outils informatiques et langages de programmation.

- Ces nombres utilisent un codage standardisé en taille fixe K (e.g., K=64 bits) ;
  ⇒ ils ne peuvent représenter plus de $\mathbf{2^K}$ **valeurs distinctes** (e.g., $<2\times10^{19}$) ;
  ⇒ toutes ces valeurs ont un **nombre fini de chiffres significatifs** (e.g., <17).

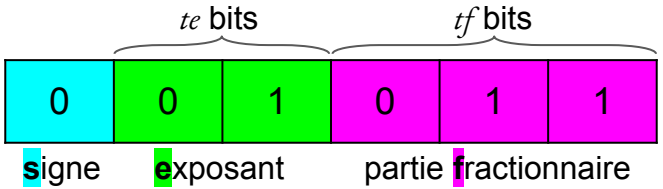
- Ce codage permet la représentation de nombres très petit (e.g., 10^{-300}) et très grands (e.g., 10^{300}) ;
  ⇒ les calculs doivent être réalisés sur des valeurs **homogènes** ;
  ⇒ les résultats doivent être **arrondis** pour être représentables.

Nombres à virgule flottante IEEE 754 (1/3)

Un standard pour le codage et les opérations : la **norme IEEE 754**

- Codage : deux paramètres, te=taille exposant, tf=taille partie fractionnaire

e = entier signé
avec biais impair
(biais = $2^{te-1}-1$)



f = codage “naturel” en
puissances négatives
(bit unité implicite = 1)

Exemple de décodage : $(-1)^{0} \times 2^{(0\times2^{1}+1\times2^{0}\ \text{biais}\ -1)} \times (1+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}) = 1{,}375$

Exemple de codage :

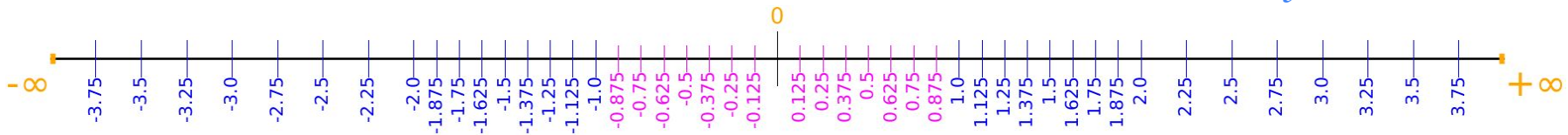
-3,5 → s = 1
signifiant = 11,1 → 1,11 ⇒ flottant = 1 10 110
e = 10 ← décalage=1 ⇒ exposant=1+biais = 2

Nombres à virgule flottante IEEE 754 (2/3)

Certains exposants représentent des valeurs particulières :

- e = 0 *(valeur d'exposant signé = -biais)*
  - f = 0 ⇒ représente la valeur *0* *(±0 selon signe)*
  - f ≠ 0 ⇒ nombre *"dénormalisé"* : le bit implicite devant la partie fractionnaire est 0 et non plus 1 ; codage/décodage de f inchangé, mais e considéré comme valant 1.
- e = *11…1* *(valeur d'exposant signé* = biais+1*)*
  - f = 0 ⇒ infini *(valeur = ±∞ selon signe)*
  - f ≠ 0 ⇒ NaN *(Not a number)*

Exemple : Valeur décimale de tous les flottants sur 6 bits avec te=2, tf=3



soit $2\times((2^2-2)\times2^3+2^3+1)-1$=**49** valeurs distinctes *(sans Nan, -0=+0)* pour **64** codes

Nombres à virgule flottante IEEE 754 (3/3)

- Tailles standards : 32 bits *(simple te=8, te=23)*, 64 bits *(double te=11, te=52)*, 128 bits *(quadruple te=15, te=112)* + tailles intermédiaires pour les calculs

  ⚠️ *Des tailles plus petites (8,16 bits), non standardisées, apparaissent aujourd'hui, notamment pour les applications en apprentissage profond*

  ⇒ Valeurs décimales minimales (non nulle) et maximales encodables :

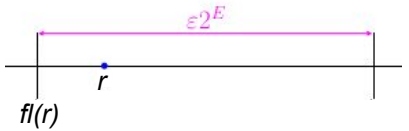
  - simple : $\pm 1.4013 \times 10^{-45}$ – $\pm 3.40282 \times 10^{38}$
  - double : $\pm 4.94066 \times 10^{-324}$ – $\pm 1.79769 \times 10^{308}$
  - quadruple : $\pm 6.47518 \times 10^{-4966}$ – $\pm 1.18973 \times 10^{4932}$

Calcul IEEE 754

- Le standard impose les opérations *“arithmétiques”* sur les flottants (+, −, ×, ÷, mod, √, …) plus des opérations de comparaison (<, =) *(-0=0, NaN⇒faux)* et de manipulation (suivant/précédent, signe, …)
- Il recommande des fonctions élémentaires (exponentiation, logarithmes, trigonométriques, …)
- Toutes les opérations doivent traiter tous les flottants (NaN, dénormalisés, infinis, …) *(opération “indéfinie” ⇒ NaN ou infini)*
- Le calcul flottant diffère cependant du calcul réel pour trois raisons principales, toutes dues à la finitude de représentation :
  - les erreurs d’**arrondi** ;
  - le phénomène d’**absorption** ;
  - le phénomène de ***cancellation***.

Arrondi

- Lors du codage ou d'un calcul, une valeur *r* peut être non représentable
  ⇒ il faut l'arrondir au flottant *fl(r) le plus proche,*
  i.e., vérifiant $|fl(r)-r| \leq \varepsilon 2^{E-1}$ *(ε = précision machine = 2^{-tf})*

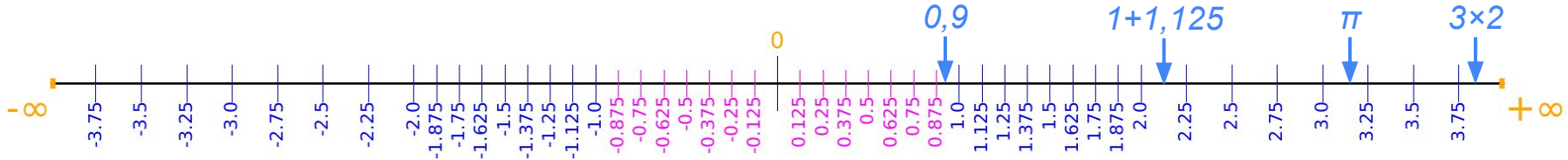


Exemples (sur 6 bits): $fl(\pi) = fl(11{,}00100100001\ldots) = 0\ 11\ 101$ *(valeur =* $3{,}25_{10}$*)*

(underflow) $fl(0{,}9) = fl(0{,}1110011\underline{0011}\ldots) = 0\ 00\ 111$ *(valeur =* $0{,}875_{10}$*)*

$fl(1+1{,}125) = fl(10{,}001) = 0\ 10\ 000$ *(valeur =* $2{,}0_{10}$*)*

(overflow) $fl(3\times 2) = fl(110{,}0) = 0\ 11\ 000$ *(valeur =* $+\infty$*)*



⚠ L'arrondi doit être pris après chaque opération flottante

Exemple : le calcul réel sin(a+b×c) → fl(sin(fl(fl(a)+fl(fl(b)×fl(c)))))

Absorption et *cancellation*

- La méthode d'addition (et de soustraction) flottante consiste à aligner les exposants des opérandes
  ⇒ Perte de précision de l'opérande la plus petite, pouvant aller jusqu'à son annulation : absorption !

Exemple (sur 6 bits) : $fl(3+0.125) = fl(1.1\times2^1+0.001\times2^0) = fl(1.1\times2^1+0.000\times2^1) = 1.1\times2^1$ (valeur = 3.0_{10})

- La soustraction de flottants très proches peut annuler la plupart des bits de leurs parties fractionnaires
  ⇒ Perte de signification du résultat, composé de bits possiblement arrondis lors de précédents calculs : cancellation !

Exemple (sur 6 bits) : $fl(3+0.375-3) = fl(fl(1.1\times2^1+0.011\times2^0)-1.1\times2^1) = fl(1.110\times2^1-1.1\times2^1) = 0.1\times2^0$ (valeur = 0.5_{10}, soit 33% d'écart au résultat réel !)

Retour sur les exemples

- Les arrondis expliquent les problèmes observés lors du tracé de polygone *($fl(\Sigma_{1..7} fl(360 \div 7) \neq 360$)* et du tirage de dé *(Numworks/randint, p3 : sur 32 bits, $fl((2^{32}-k)/2^{32})=1$ pour $k=1..128 \Rightarrow$ 1 chance sur 33×10^6 de tirer 7)*.

- L'absorption explique le calcul d'aire qui s'annule subitement *($fl(10^6+10^6+10^{-10})=10^6$)* et le sprint d'Achille trop tôt interrompu *($fl(1-2^{-53}+2^{-54})=1$)*.

- La cancellation *(et les arrondis)* explique le calcul d'aire complètement erroné *($fl(10^6+10^{-10}-10^6) \neq 1e\text{-}10$)*.

Tout est perdu ?

Le calcul flottant est entaché d'erreur vis-à-vis du calcul réel, doit-on jeter les supercalculateurs, tel Jean Zay à 36,85Pflop/s, à la poubelle ?

Non, car ce calcul est **predictible** et explicable !

Exemple : IEEE 754 impose des opérations arithmétiques arrondies au flottant près.

On peut ainsi prédire le degré d'approximation de l'évaluation d'un calcul C : on parle de **calcul d'erreur**, absolue (|fl(C)-C|) ou relative (|fl(C)-C|/|C|).

Exemple : erreur relative pour la somme S de 3 flottants a, b et c

Trois formulations : a+(b+c) ; b+(a+c) ; c+(a+b)

$fl(a+fl(b+c)) = (a+(b+c)\times(1+\delta_1))\times(1+\delta_2)$ avec $\delta_i \leq \varepsilon/2$ *(= precision machine ÷ 2)*

$= S \times (1 + \delta_2 + (b+c)/S\ \delta_1 + (b+c)/S\ \delta_1\delta_2)$

$\Rightarrow$ erreur relative $\approx \delta_2 + (b+c)/S\ \delta_1$ *(on néglige les erreurs d'ordre 2)*

Cela permet de choisir la formulation selon les magnitudes de (a+b), (b+c) et (a+c).

Et au-delà des flottants ?

Si les performances ne sont pas un souci, il existe de nombreuses alternatives au calcul flottant :

- calcul rationnel *(e.g., Python/fractions)*
- calcul décimal (généralement en précision *arbitraire*[3]) *(e.g., Python/decimal)*
- calcul à virgule fixe ou flottante en précision *arbitraire*[3] *(e.g., Python/gmpy2)*

Il existe aussi des couches logicielles au-dessus du calcul flottant visant à le rendre plus rigoureux

- calcul stochastique (arrondis aléatoires)
- calcul par intervalles (encadrement garanti)

Domaine de recherche encore très actif !

[3] arbitraire≠infinie … et souvent il faut fixer la précision cible avant le calcul

Conclusion

Mon exposé en 180 *(dixièmes de)* secondes

- L’incarnation physique des nombres en machine les rend obligatoirement finis.

- Le calcul informatique obéit à des règles précises, prédictibles, explicables, mais différentes du calcul mathématique.

- Cela concerne les entiers, les réels, et plus généralement toutes les familles de nombres dans tous les outils et sur tous les supports numériques.

Vous ne pourrez plus dire que vous n’étiez pas prévenus ! 😉

Des questions ?

Des réponses ! →

(et des éléments historiques, des exemples illustratifs, des détails sur les procédures de calcul de Python, Numworks, LibreOffice, …)

Le calcul sur ordinateur

Frédéric Goualard et Christophe Jermann

Département d'Informatique, Nantes Université
LS2N UMR CNRS 6004

2023-04-13, v. 1.0

Je sais que deux et deux font quatre — et serais heureux de le prouver, aussi, si je le pouvais — bien que je dois dire que si, par un quelconque expédient, je pouvais transformer 2 et 2 en cinq, cela me donnerait un bien plus grand plaisir. — Lord Byron, In [1, p. 159], 1813–1814 (trad. de l'anglais par les auteurs).

Peut-on tirer un 7 avec un dé à six faces ? Un nombre peut-il être différent de lui-même ? Si a et b sont deux entiers strictement positifs, leur produit peut-il être inférieur à a et b ? Autant de questions apparemment triviales qui se révèlent bien plus complexes lorsqu'un ordinateur entre en jeu.

Les nombres entiers et les nombres à virgule flottante — ou simplement, flottants — censés représenter les ensembles d'entiers naturels et relatifs ainsi que les nombres réels sur les ordinateurs n'en possèdent pas toutes les propriétés. Méconnaitre cet état de fait, c'est s'exposer à traduire servilement les méthodes mathématiques sur les entiers et les réels en des algorithmes aux résultats erronés et ainsi considérer l'arithmétique sur les ordinateurs comme de la magie, où chaque résultat ne découlerait pas logiquement du précédent. Nous décrivons dans ce fascicule les règles simples vérifiées par l'arithmétique entière et en virgule flottante sur les ordinateurs et montrons comment utiliser ces règles pour écrire de vraies traductions des méthodes mathématiques adaptées aux capacités des ordinateurs et établir des preuves sur les résultats.

Figure 1 : Poster de Yakov Guminer, 1931. Source : Wikimedia Commons.

[1] Lord Byron. 'Alas! the love of Women!' Byron's Letters and Journals. Sous la dir. de Leslie A. Marchand. T. 3. The Belknap Press of Harvard University Press, 1974

Comment lire ce fascicule ? La section 1 justifie historiquement l'utilisation de la représentation binaire pour stocker et manipuler l'information dans un ordinateur. La section 2 présente la représention des nombres entiers ; la section 2.1 rappelle la notion de représentation positionnelle des nombres ; elle est indispensable à la compréhension de la suite mais peut être omise par les lecteurs et lectrices qui la maîtrisent déjà. La section 3 présente les différentes représentations des nombres réels ; l'accent est mis sur la représentation à virgule flottante (section 3.2) suivant le standard IEEE 754. Les bonnes pratiques à adopter ainsi que la quantification *a priori* des erreurs de calcul sont présentées dans la section 3.2.2. Les exemples sont, pour l'essentiel, tirés des applications et outils potentiellement utilisés à l'école élémentaire, au collège, ou au lycée : calculatrice Numworks, langages Python 3, Scratch et Snap!, tableurs LibreOffice Calc, Google Sheets et Microsoft Excel, … Sauf indication contraire, tous les exemples de code informatique sont écrits en Python 3.

1

Disponible sur le site de l'IREM PdL : https://irem.univ-nantes.fr/

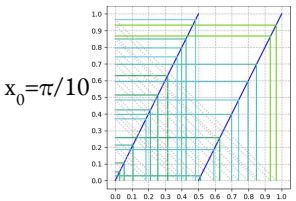
Bonus de fin de générique (de fin)

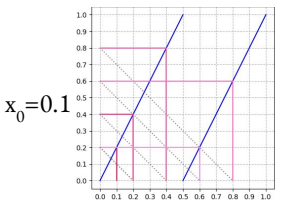
La fonction dyadique

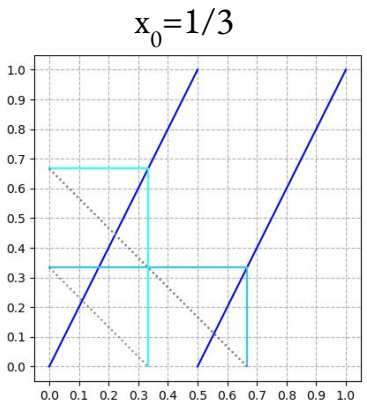
La fonction $f(x) = 2x \bmod 1$ permet de définir un système dynamique $x_{k+1} = f(x_k)$, de condition initiale $x_0 \in [0,1[$, aussi appelé *décalage de Bernoulli*.

Mathématiquement :

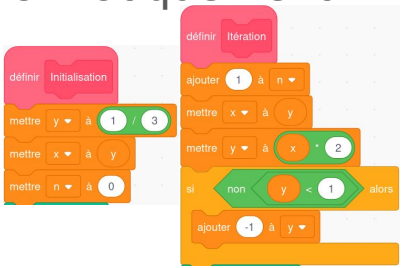
- $x_0 \in \mathbb{R} \Rightarrow$ orbite apériodique
- $x_0 \in \mathbb{Q} \Rightarrow$ orbite périodique

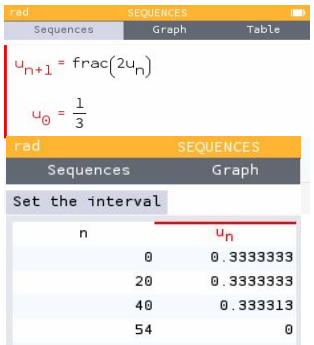


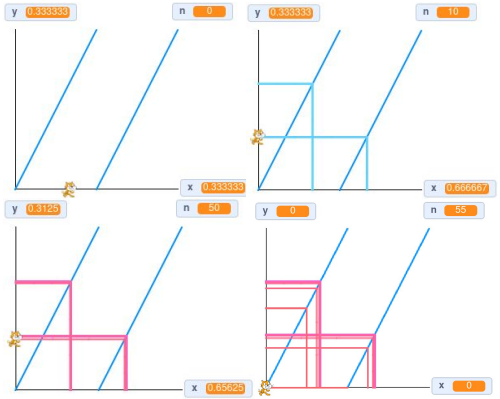




Informatiquement :








```
code$ python DyadicMap.py
Using floats:
    0 : 0.3333333333333333
    20 : 0.3333333333139308
    40 : 0.33331298828125
    54 : 0.0
Using rationals:
    0 : 1/3
    20 : 1/3
    40 : 1/3
    54 : 1/3
```