# Nombres flottants

INFORMATIQUE COMMUNE - TP n° 2.8 - Olivier Reynet

**À la fin de ce chapitre, je sais :**

☞ sortir de la salle dans laquelle on m'a pris au piège

## A Comment sortir de la salle ?

Ça y est. Vous y êtes. Dernier TP d'informatique de l'année. En fermant la porte derrière vous, un bruit sinistre, un enclenchement de verrou rouillé résonne dans la salle. Vous êtes coincé par un beau soleil de juin, à deux pas de la plage alors que vous avez presque fini les cours !

C'est à ce moment que vous le voyez[1], sur la porte, un petit message de votre professeur[2] : «Les portes peuvent être déverrouillées en suivant ce lien. Le code pour sortir se calcule de la manière suivante :

- faire la somme des résultats des questions B3 et B4,
- multiplier par la partie fractionnaire du résultat de la dernière question de la section C,
- multiplier le tout par l'indice $n_0$ de la limite trouvée à la section D,
- le code est le reste de la division euclidienne de ce nombre par 21062023.

Bonne chance !»

**Solution :**

```
print((102 * 125 * 2097152) % (21062023))
# 10980813  MPSI1  --> 833928360
print((92 * 25 * 2097152) % (21062023))
# 246333 MPSI2  --> 1481139999
```

## B Du rififi dans la sommation

Un élève tente de calculer la somme suivante :

$$\sum_{k=1}^{n} \frac{1}{n} \tag{1}$$

Il lui semble que le résultat vaut un, mais, pas très sûr de ses méthodes de calcul analytique, il décide de vérifier avec Python si son résultat est correct.

1. trop tard…
2. qui n'est plus dans la salle d'ailleurs !

B1. Écrire une fonction de signature `somme64bits(n:int)-> int` qui renvoie cette somme.

**Solution :**

```
def somme64bits(n):
    h = 1/n
    x = 0
    for k in range(n):
        x += h
    return x
```

B2. Écrire une fonction de signature `somme32bits(n:numpy.float32)-> numpy.float32` qui renvoie cette somme calculée avec des flottants codés sur 32 bits grâce à Numpy.

**Solution :**

```
def somme32bits(n):
    h = np.float32(1 / n)
    x = np.float32(0)
    for k in range(n):
        x += np.float32(h)
    return x
```

Le dernier cours d'informatique sur les bonnes pratiques lui donne l'idée suivante : il va tester les deux fonctions précédentes pour toutes les valeurs de n de un à $2^{14}$ inclus.

B3. À la suite de ce test, combien de valeurs de $n$ vérifie $\sum_{k=1}^{n} \frac{1}{n} = 1$ lorsqu'on utilise la fonction `somme64bits(n:int)-> int`?

**Solution :**

```
res64 = [k for k in range(1, 2**14 + 1) if somme64bits(k) - 1 == 0]
```

B4. Même question lorsqu'on utilise la fonction `somme32bits(n)`?

**Solution :**

```
res32 = [k for k in range(1, 2**14 + 1) if somme32bits(k) - 1 == 0]
```

B5. Interpréter ces résultats.

**Solution :** Ces résultats sont dû à l'effet des arrondis sur la sommation des flottants. Plus on code les flottants de manière précise, plus il y a de tests invalides...Les résultats peuvent être testés grâce des fonctions numpy comme `is_close` : cette fonction teste si deux nombres sont égaux moyennant une certaine tolérance.

## C Du bruit dans l'intégration

On souhaite connaître l'évolution d'un système dynamique après une seconde de fonctionnement. Ce système obéit à l'équation d'évolution suivante :

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 \tag{2}$$

Par ailleurs, la valeur initiale de $x$ est connue : $x(0) = 0$.

Grâce à ses cours de mathématiques et sa science du calcul, un élève se doute que $x$ vaut 1 à la fin du temps écoulé. Cependant, il souhaite vérifier en codant avec Python la résolution numérique. Grâce à ses cours de physique, il parvient à écrire le code suivant :

```
def f(x):
    return 1

def one_step(f, x, dt):
    return x + dt * f(x)

def euler(f, x0, dt, T=1):
    t = 0
    x = x0
    n = int(T / dt)
    for _ in range(n):
        x = one_step(f, x, dt)
    return x
```

C1. À l'aide de la méthode d'Euler, simuler ce système pendant une seconde par pas de 0,1 seconde.

**Solution :**

```
print(euler(0,0.1,T=1))
```

C2. En divisant à chaque fois par 10, faire varier le pas d'intégration de 0,1 à $10^{-6}$ et mémoriser les résultats obtenus.

**Solution :**

```
def euler_test():
    dt = 0.1
    S = []
    while dt > 1e-7:
        S.append((dt, euler(0, dt)))
        dt = dt / 10
    return S
# [(0.1, 0.9999999999999999), (0.01, 1.0000000000000007), (0.001,
    1.0000000000000007), (0.0001, 0.9999999999999062), (1e-05,
    0.9999899999980838), (1.0000000000000002e-06, 0.9999990000079182),
    (1.0000000000000002e-07, 0.99999989975017)]
```

C3. Sachant que x devrait valoir 1 au bout d'une seconde, calculer l'erreur absolue commise pour chaque pas d'intégration de 0,1 à $10^{-6}$ .

**Solution :**

```
def ea(S, val):
    E = []
    for dt, e in S:
        E.append((dt,abs(val-e)))
    return E
#[(0.1, 1.1102230246251565e-16), (0.01, 6.661338147750939e-16), (0.001,
    6.661338147750939e-16), (0.0001, 9.381384558082573e-14), (1e-05,
    1.000000191619943e-05), (1.0000000000000002e-06, 9.999920818071217e-07),
     (1.0000000000000002e-07, 1.0024982999290444e-07)]
```

C4. Expliquer le phénomène observé sur l'erreur absolue.

**Solution :** Le choix d'un pas d'intégration plus petit engendre une augmentation de l'erreur absolue, ce qui paraît contre intuitif, à moins qu'on sache comment les flottants sont codés en machines.

0,1 n'est pas un nombre dyadique, on ne peut donc pas le représenter en binaire d'une manière exacte sur une nombre fini de bits. C'est donc une approximation de 0,1 et de ses puissances qui est codée en machine. L'erreur absolue mesurée est donc entachée du bruit numérique causé par l'approximation de 0,1 et de ses puissances en base 2.

Cet exemple montre que le bruit numérique engendré par un pas de simulation non dyadique n'est pas négligeable. Prendre plus de points dans ce cas n'est pas une solution.

C5. Proposer un pas d'intégration compris entre 0,1 et 0,2 qui annule l'erreur absolue dans ce cas.

**Solution :** On peut choisir par exemple 0,125 et diviser à chaque fois le pas par deux plutôt que par 10. Le bruit numérique n'est plus perceptible dans ce cas!

```
def bin_euler_test():
    dt = 0.125
    S = []
    while dt > 1e-7:
        S.append((dt, euler(0, dt)))
        dt = dt / 2
    return S

Sbin = bin_euler_test()
print(ea(Sbin, 1))
#[(0.125, 0.0), (0.0625, 0.0), (0.03125, 0.0), (0.015625, 0.0), (0.0078125,
    0.0), (0.00390625, 0.0), (0.001953125, 0.0), (0.0009765625, 0.0),
    (0.00048828125, 0.0), (0.000244140625, 0.0), (0.0001220703125, 0.0),
    (6.103515625e-05, 0.0), (3.0517578125e-05, 0.0), (1.52587890625e-05,
    0.0), (7.62939453125e-06, 0.0), (3.814697265625e-06, 0.0),
    (1.9073486328125e-06, 0.0), (9.5367431640625e-07, 0.0),
    (4.76837158203125e-07, 0.0), (2.384185791015625e-07, 0.0),
    (1.1920928955078125e-07, 0.0)]
```

## D Un scandale en série

La série harmonique s'écrit :

$$H_n = \sum_{k \geqslant 1} \frac{1}{k} \tag{3}$$

D1. La série harmonique est-elle convergente ?

**Solution :** Non, cf. cours de mathématiques !

D2. Écrire une fonction de signature `f32_harm(n:int)-> numpy.float32` qui calcule la somme des n premiers termes de la série harmonique dans l'ordre croissant des $k$ en utilisant des flottants codés sur 32 bits.

**Solution :**

```
def f32_sharm(n):
    s = np.float32(0)
    for k in range(1, n + 1):
        s += np.float32(1 / k)
    return s
```

D3. Écrire une fonction de signature `rf32_harm(n:int)-> numpy.float32` qui calcule la somme des n premiers termes de la série harmonique dans l'ordre décroissants des $k$ en utilisant des flottants codés sur 32 bits.

**Solution :**

```
def rf32_sharm(n):
    s = np.float32(0)
    for k in range(n + 1, 0, -1):
        s += np.float32(1 / k)
    return s
```

D4. Y-a-t-il un intérêt à procéder dans le sens croissant ou décroissant. Pourquoi ?

**Solution :** Oui, il vaut mieux additionner les plus petits avant de les additionner avec les nombres plus grands afin de limiter l'impact du phénomène d'absorption.

D5. Montrer que, si on s'y prend mal, la série harmonique calculée sur des flottants 32 bits converge et atteint sa limite. Estimer l'indice $n_0$ à partir duquel la limite est atteinte.

**Solution :** S'y prendre mal, c'est commencer par sommer les gros en premier.

La somme commence par le terme 1 et puis on ajoute des termes positifs, de plus en plus petits. Ceci implique que l'exposant utilisé pour représenter le résultat sera au moins 0, c'est-à-dire $H_n = 1,m \times 2^0$, avec $m$ codée sur 23 bits. La limite des 23 bits de mantisse va être atteinte pour un certain indice $n$ : on ne pourra plus représenter $1/n$ avec 23 bits de mantisse et un exposant de 0. Tous les termes suivants seront nuls et la somme cesse donc de croître. Elle «converge» artificiellement à cause de la représentation des nombres flottants au delà-de $n = 2^{23}$.

D6. Trouver expérimentalement cette limite $n_0$.

**Solution :**

Au-delà de 22030 termes, la somme dépasse les 10. Donc, cela signifie que l'exposant utilisé sera 3 car 10 en binaire s'écrit $1010_2$ : $H_{n\geqslant 22030} = 1,m \times 2^3$.

À partir d'un certain indice $n_0$, le nombre $1/n_0$ va être de l'ordre de $2^{-21}$ et avec un exposant à 3 la limite de précision des 23 bits en binaire sera dépassée car le plus petit représentable est alors $2^{-20}$.

La limite sera donc atteinte pour $n_0 = 2^{21}$.

```
def test_limits_32():
    for n in range(2097150, 2097160,1):
        print(n,f32_sharm(n))
# 2097150 15.403681
# 2097151 15.403682
# 2097152 15.403683
# 2097153 15.403683
# 2097154 15.403683
# 2097155 15.403683
# 2097156 15.403683
# 2097157 15.403683
# 2097158 15.403683
# 2097159 15.403683
```