

La partie "code" est corrigée en Python, en commençant par :

```
import numpy as np
import random as rd
```

Exercice 1 Ecriture en base b d'un rationnel S

1. Soient $(r_k)_{k=0}^{+\infty}$ et $(a_k)_{k=1}^{+\infty}$ la suite obtenue par la procédure décrite dans la question. Il suffit de montrer que $\frac{p}{q} = \sum_{k=1}^{+\infty} \frac{a_k}{b^k}$, que $0 \leq a_k \leq b-1$ et qu'il n'existe pas de rang n tel que $a_k = b-1$ pour tout $k \geq n$.

On montre par récurrence que pour tout $n \in \mathbb{N}^*$ on a $\sum_{k=1}^n \frac{a_k}{b^k} = \frac{p}{q} - \frac{r_n}{qb^n}$.

Pour le cas $n=1$, on remarque que par définition de la division euclidienne, on a $r_0b = qa_1 + r_1$, ce qui est bien l'hypothèse de récurrence puisque $r_0 = p$.

Soit $n \in \mathbb{N}$ un entier tel que $\sum_{k=1}^n \frac{a_k}{b^k} = \frac{p}{q} - \frac{r_n}{qb^n}$. Alors

$$r_n = pb^n - q \sum_{k=1}^n a_k b^{n-k}$$

et par construction de r_{n+1} et a_{n+1} on a

$$a_{n+1}q + r_{n+1} = r_nb.$$

En remplaçant la valeur de r_n dans cette égalité et en divisant par qb^{n+1} , on obtient

$$\sum_{k=1}^{n+1} \frac{a_k}{b^k} = \frac{p}{q} - \frac{r_{n+1}}{qb^{n+1}}$$

ce qui achève la preuve par récurrence.

Puisque pour tout $n \geq 1$ l'entier r_n est le reste dans une division euclidienne par q , on a $r_n \leq q-1$. Donc $\frac{r_n}{qb^n} \rightarrow 0$ quand $n \rightarrow +\infty$, et donc on a bien

$$\frac{p}{q} = \sum_{k=1}^{+\infty} \frac{a_k}{b^k}.$$

Par ailleurs, $0 \leq r_nb < (q-1)b$ donc $0 \leq a_{n+1} < b$.

Enfin, on montre par l'absurde qu'il n'existe pas de $n \in \mathbb{N}$ tel que $a_k = b-1$ pour tout $k \geq n$. Supposons que ça soit le cas, alors on a

$$\frac{p}{q} = \sum_{k=1}^{n-1} \frac{a_k}{b^k} + \sum_{k=n}^{+\infty} \frac{b-1}{b^k}$$

donc, en écrivant $l = \sum_{k=1}^{n-1} \frac{a_k}{b^k}$, d'après la formule des sommes géométriques on a

$$\frac{p}{q} = l + \frac{1}{b^n} \frac{b-1}{1 - \frac{1}{b}} = l + \frac{1}{b^{n-1}}.$$

Mais d'après la formule démontrée par récurrence, on a $\frac{p}{q} - l = \frac{r_{n-1}}{qb^{n-1}}$, donc $1 = \frac{r_{n-1}}{q}$, ce qui est impossible car r_{n-1} est le reste dans une division euclidienne par q .

2. D'après la construction donnée à la question 1, on remarque que pour tout n les suites r_{n+1}, r_{n+2}, \dots et a_{n+1}, a_{n+2}, \dots ne dépendent que des nombres r_n, q et b . Mais comme r_n est un entier compris entre 0 et $q-1$, d'après le principe des tiroirs il existe des entiers n, m distincts tels que $r_n = r_m$, donc les suites a_{n+1}, a_{n+2}, \dots et a_{m+1}, a_{m+2}, \dots sont égales. Quitte à échanger n et m on peut supposer que $n < m$. Posons $T = m - n$, par l'égalité des suites on a $a_{n+k} = a_{n+T+k}$ pour tout $k \geq 1$, c'est-à-dire que (a_k) est périodique à partir du rang $n + 1$, de période T .
3. On suit l'algorithme : $b = 2, p = 1$ et $q = 10$. On trouve

n	1	2	3	4	5
r_n	2	4	8	6	2
a_n	0	0	0	1	1

Le reste 2 apparaît pour $n = 1$ et $n = 5$, le reste du développement est donc donné par la répétition du motif 0011 à l'infini : en base 2, l'écriture de $1/10$ est 0.00011001100110011... On peut le trouver en Python avec le code :

```
def devfraction(p, q, n):
    r=list(range(n+1)); r[0]=p;
    a=list(range(n));
    for i in a:
        a[i]=int(r[i]*2/q); r[i+1]=r[i]*2-a[i]*q;
    return a
print( devfraction(1,10,15) )
```

Le flottant correspondant à x en base 2 avec 12 bits est de mantisse 1.10011001100, d'exposant -4 (ou plus strictement, de mantisse 1100110011 d'exposant -13). Ainsi,

$$\hat{x} = \left(1 + \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9}\right) 2^{-4}$$

c'est à dire 0.0999755859375. Le code en Python pour l'obtenir est :

```
def sumbin(a):
    return sum([a[i]*2**(-i) for i in range(len(a)) ])
sumbin([1,1,0,0,1,1,0,0,1,1])*2**(-4)
```

De même, on obtient que $3/10$ s'écrit 0.01001100110011... et que son arrondi à 12 bits de mantisse est 0.2999267578125.

4. Les nombres 0.3 et 0.1 sont arrondis à 53 bits de mantisse dans Python. Le flottant correspondant à $0.1+0.1+0.1$ a une erreur correspondant à trois fois celle commise par l'arrondi, qui n'est pas exactement égale à l'erreur sur 0.3. On obtient $5.5511e-17$. Notons que dans d'autres cas l'erreur est trop petite et n'est pas affichée par Python (par exemple $0.5 - (0.1 + 0.1 + 0.1 + 0.1 + 0.1)$).

Exercice 2 Durant la première guerre du Golfe, une batterie anti-missile Patriot a raté l'interception d'un missile Scud qui causa la mort de 28 personnes. Nous allons essayer de comprendre pourquoi.

L'ordinateur de la batterie Patriot représente les nombres par des nombres à virgule fixe en base 2 avec une précision de 23 chiffres après la virgule. L'horloge de la batterie compte le temps en dixième de seconde. Pour obtenir le temps en seconde, le programme divise le temps donné par l'horloge par dix.

1) Avec les fonctions Python définies à l'exercice 1, on obtient une erreur de

$$(0.1 - \text{sumbin}(\text{devfraction}(1, 10, 23))) \times 10$$

c'est-à-dire une erreur d'au moins 10^{-7} secondes par seconde. 2) On calcule $100 * 60^2$ multiplié par l'erreur, on trouve environ 0.03. Cela suppose que le fonctionnement de l'horloge du scud est d'ajouter +0.1 à son compteur à chaque dixième de seconde, indiqués par une autre horloge. 3) L'erreur de distance est environ $1676 * 0.03 \simeq 54$ mètres par seconde de vol du scud.

Exercice 3 Les calculs exactes donnent des résultats... exacts. Pour les calculs approchés, tout dépend de la taille de la mantisse. Le nombre 10^{20} correspond à environ $1.0101... \times 2^{66}$ (utiliser `np.binary_repr`), donc le +1 dans $10^{20} + 1$ est arrondi dans un calcul avec une mantisse de moins de 66 chiffres de mantisse. Le premier calcul approché donne donc 0.0 alors que le vrai résultat est 1.

Pour le coefficient binomial, le résultat exact est $1000 * 999 / 2 = 499500$. Si on essaye de faire le calcul approché sans simplifier, il faut créer une fonction factoriel :

```
def factoriel(n):
    a=1.0
    for i in range(n):
        a=(i+1)*a
    return a;
```

On remarque alors que $1000! \simeq \sqrt{2\pi} \left(\frac{1000}{e}\right)^{1000}$ est plus grand que 10^{1024} donc compté comme l'infini, de même que 998!. Le résultat donné par Python est "nan" (forme indéterminée).

Enfin, on sait que $\sum_{k=1}^N \frac{1}{k} - \log(N)$ converge quand N tends vers l'infini, vers la constante $\gamma \simeq 0.577$ d'Euler-Mascheroni. Théoriquement, si on fait un calcul approché pour N dépassant 2^r , les valeurs $1/n$ pour n supérieur à 2^r seront arrondies dans la somme, alors que $\log(N)$ continue de croître, et donc le résultat diverge vers l'infini. En pratique avec $r = 53$ bits de mantisse on atteint difficilement le moment où le résultat se met à

diverger. (rq : utiliser `np.log` pour le logarithme sous Python, ayant importé le package `numpy` en tant que `np`).

Exercice 4 Expressions les plus judicieuses :

- $B = \sin^2(10^{-8})$ est plus judicieux, il donnera un flottant avec un exposant plus petit. (utiliser `np.cos` du package `numpy` importé comme `np`).
- $B = \sum_{n=0}^{10^9-1} \frac{1}{10^9-n}$ est plus judicieux, on commence par sommer les petits nombre pour éviter qu'ils soient arrondis (la somme des petits nombres est non négligeable, et ne sera pas arrondie à la fin).
- $B = \frac{1}{e^{10}} \approx \left(\sum_{n=0}^{30} \frac{10^n}{n!}\right)^{-1}$, dans l'autre cas il y a beaucoup d'éliminations entre des grands nombres, qui donnent lieux chacun à de grandes erreurs.

On a

$$\frac{1}{\sqrt{10^{10}+1}-\sqrt{10^{10}-1}} = \frac{\sqrt{10^{10}+1}+\sqrt{10^{10}-1}}{10^{10}+1-(10^{10}-1)} = \frac{\sqrt{10^{10}+1}+\sqrt{10^{10}-1}}{2}$$

Exercice 5 : évaluation d'un polynôme en un réel.

1. A l'étape 1 on a une multiplication, à chacune des n étapes suivantes on a 2 multiplications et 1 addition, donc $2n+1$ multiplications et n additions au total.
2. Méthode de Hörner
On fait n multiplications et n additions.
3. Le programme par la méthode "naive" :

```
def naifpol(a, x):
    n=len(a)-1;
    s=a[0]; u=x; v=a[1]*u;
    for i in range(n-1):
        s=s+v; u=u*x; v=a[i+2]*u
    return s+v;
```

et par la méthode de Hörner :

```
def hornerpol(a, x):
    n=len(a)-1
    p=a[n]
    for i in range(n):
        p=a[n-i-1]+x*p
    return p;
```

Pas de différence sensible de temps d'exécution pour un polynome de taille 100, ce temps reste très variable d'une exécution à l'autre pour de mêmes polynomes.

4. Montrer que $(1-u)^{k-1} \leq \left|\frac{\hat{u}_k}{u_k}\right| \leq (1+u)^{k-1}$ par récurrence. On en déduit que

$$\leq \left|\frac{\hat{v}_k}{v_k} - 1\right| \leq \gamma_k(u)$$