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 $r_0 b = qa_1 + r_1$, ce qui est bien l'hypothèse de récurence 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

$$a_{n+1}q + r_{n+1} = r_n b \ .$$

   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 $\frac{r_n}{qb^n} \to 0$ quand $n \to +\infty$, et donc on a bien

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

   Par ailleurs, $0 \leq r_n b < (q-1)b$ 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 $r_{n+1}, r_{n+2}, ...$ et $a_{n+1}, a_{n+2}, ...$ 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 $a_{n+1}, a_{n+2}, ...$ et $a_{m+1}, a_{m+2}, ...$ 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 periodique 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 apparait 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 arrondis 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'arrondis, 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-sumbin(devfraction(1,10,23))) x 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 arrondis 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 $\sum_{k=1}^{N} \frac{1}{n} - \log(N)$ 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 beaucoups d'éliminations entre des grands nombres, qui donnent lieux chacuns 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 chacunes 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

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

et on en déduit les inégalités suggérées par l'énoncé. On peut donc, en supposant que les a_k et x sont positifs, considérer que l'erreur relative entre le résultat obtenu et le vrai résultat est de l'ordre de $\gamma_n(u)$.

Exercice 6

1. On a

$$\begin{aligned} I_0 &= \int_0^1 \frac{1}{10+x} dx \\ &= \int_{10}^{11} \frac{1}{y} dy \\ &= \ln(11) - \ln(10) \ . \end{aligned}$$

2. On a

$$\frac{x^n}{10+x} = \frac{x^{n-1}(x+10-10)}{10+x} = x^{n-1} - 10\frac{x^{n-1}}{10+x} \ .$$

On obtient le résultat attendu en intégrant cette égalité entre 0 et 1.
Cela suggère le code suivant pour calculer I_n :

```
def I(n):
        x=np.log(11.0/10.0);
        for i in range(n):
            x=1.0/(i+1)-10.0*x
        return x;
```

On remarque que $I(15)$ est négatif, ce qui est impossible, puis que $I(n)$ diverge, prenant des valeurs négatives et positives de plus en plus grandes. Cela s'explique par le fait que toute erreur d'arrondis faite dans le calcul de I_0 aura été multipliée par 10^n au rang n, et devient donc rapidement visible.

3. On peut estimer que $I_{30} = 0$, puisque x^{30} est très proche de zéro sur la majorité de l'intervalle $[0, 1]$. On définit la fonction suivante :

```
def J(n,N):
    x=0.0;
    for i in range(0,N-n):
        x=1.0/(10*(N-i))-x/10;
    return x;
```

qui retourne une estimation de I_n obtenue par l'approximation $I_N = 0$. On obtient $J(0, 30)$ très proche de $ln(11/10)$, et $J(20, 30) = 0.00434....$ Notons que dans ce cas l'erreur due à l'approximation $I_N = 0$ est divisée par 10^{N-n}, et n'est plus que de l'ordre 10^10 pour le calcul de I_{20} à partir de $I_{30} = 0$.

Exercice 7

1. En utilisant Taylor-Lagrange, on obtient que $|f_1 - f'(x)| \leq \frac{1}{2}Mh$ où M majore la dérivée seconde, et même $|f_2 - f'(x)| \leq \frac{1}{6}Nh^2$ où N majore la dérivée troisième.
2. L'erreur absolue sur le calcul de $f(x+h) - f(x)$ due à l'arrondissement des flottants est de l'ordre de $2\varepsilon|f'(x)|$, ce qui donne une erreur absolue sur le calcul de f_1 de l'ordre de $\frac{2\varepsilon}{h}|f'(x)|$. En supposant qu'on a des fonctions du même ordre, cela donne une erreur relative totale (en prenant en compte l'approximation de Taylor) du type de
$$\frac{2\varepsilon}{h} + \frac{h}{2}.$$
3. En étudiant la fonction $h \mapsto \frac{\varepsilon}{h} + h$, on observe qu'elle atteint son minimum en $h = \sqrt{\varepsilon} \simeq 3e-7$.
4. On teste avec le code suivant :

```
def f(x):
    return np.sqrt(np.exp(np.sin(x)))
def f_1(h):
    return (f(h)-0.5)/h
for i in range(20):
    print f_1(10**(-i))
```

   On obtient ainsi la faleur de $f_1(10^{-k})$ pour k allant de 0 à 20. Par des développements limités successifs, on montre que $f'(0) = \frac{1}{2}$. La valeur de $f_1\left(10^{-k}\right)$ s'en approche effectivement de plus en plus jusqu'à $k = 7$, puis s'en éloigne.
5. On peut approcher la dérivée seconde de f par la formule
$$\frac{f(x+h) + f(x-h) - 2f(x)}{h^2}\ .$$
   D'après Taylor-Lagrange cette formule s'approche de la dérivée seconde avec un reste de l'ordre de h^2. L'erreur est maintenant de la forme
$$\frac{\varepsilon}{h} + h^2.$$
   (à coefficients d'ordre 1 près), ce qui est minimal pour h de l'ordre de $\varepsilon^{\frac{1}{3}}$. La dérivée seconde est donc plus difficile à approximer, puisque l'erreur due à l'approximation augmente plus vite.

Problème 8 (différences finies en dimension 1)

1. Fixons $k \in \{1, ..., N\}$ et calculons le vecteur $x = Au_k$. On note $\tau_k = \frac{k\pi}{N+1}$. Pour $1 \leq n \leq N$ notons x_n le n-ième coefficient de x. D'après la forme de la matrice, on a, pour tout $j = 2, ..., N-1$,
$$x_j = 2\sin(j\tau_k) - \sin((j-1)\tau_k) - \sin((j+1)\tau_k)\ .$$

Notons que pour $j = 1$ on a $\sin((j-1)\tau_k) = 0$ et pour $j = N$ on a $= \sin((j+1)\tau_k) = 0$, et donc la formule précédente pour x_j est aussi vérifiée pour $j = 1$ et $j = N$. En utilisant $\sin(x) = Im\left(e^{ix}\right)$ on obtient

$$\begin{aligned} x_j &= Im\left(e^{ji\tau_k}\left(2 - e^{i\tau_k} - e^{-i\tau_k}\right)\right) \\ &= Im\left(e^{ji\tau_k} 2(1 - \cos(\tau_k))\right) \\ &= 2(1 - \cos(\tau))\sin(j\tau_k) \ . \end{aligned}$$

Ainsi, $x = \lambda_k u_k$, la valeur propre étant

$$\lambda_k = 2\left(1 - \cos\left(\frac{k\pi}{N+1}\right)\right) \ .$$

Pour $k = 1, ..., N$, on a $0 < \frac{k\pi}{N+1} \leq \pi$ et cos est strictement décroissant sur $[0, \pi]$, donc ces valeurs propres sont distinctes. Puisqu'il y en a N, ce sont toutes les valeurs propres. De plus, A est symétrique, et les λ_k sont strictement positifs, donc A est une matrice définie positive ; comme la somme d'une matrice définie positive et d'une matrice semi-define positive est toujours définie positive, cela implique que $A+\tilde{C}$ est une matrice définie positive pour toute matrice $\tilde{C}$ diagonale à coefficients positifs ou nuls. En particulier $A+\tilde{C}$ est à valeurs propres strictement positives et est inversible.

2. On a $u''(nh) \simeq \frac{2u_n - u_{n-1} - u_{n+1}}{2h} = \frac{1}{2h}[Au]_n$ pour $2 \leq n \leq N-1$, et les conditions au bord $u(0) = u(1) = 0$ montrent que $u''(nh) \simeq \frac{1}{2h}[Au]_n$ est toujours valide pour $n = 1, N$. L'équation discrétisée devient donc

$$-A\vec{u} + C\vec{u} = \vec{f} \ . \tag{1}$$

3. Effectuer la résolution numérique par une méthode de résolution directe. On pourra utiliser `A:=matrix(N,N);` pour créer une matrice dense de taille N, puis `A[0..N-1,0..N-1]:=2.0` pour mettre la diagonale à 2, etc.

4. Observez le temps de calcul en fonction de N, par exemple pour $c(x) = x, f(x) = (1 + 2x - x^2)e^x + x^2 - x$
Représenter graphiquement l'erreur avec la solution exacte $u(x) = (1-x)(e^x - 1)$.

5. Méthodes itératives : on peut utiliser la méthode de Jacobi pour obtenir une valeur approchée de $\vec{u} = (u_1, ..., u_N)$ plus rapidement lorsque N est grand. On écrit $\tilde{A} = D + (\tilde{A} - D)$ où D est la partie diagonale de $\tilde{A}$, puis le système $\tilde{A}\vec{u} = b$ en remplaçant $\vec{u}$ par deux termes successifs d'une suite récurrente v_k (à valeurs dans $\mathbb{R}^N$) :

$$Dv_{k+1} + (\tilde{A} - D)v_k = b \ \Rightarrow \ v_{k+1} = v_k + D^{-1}(b - \tilde{A}v_k)$$

Il peut alors être judicieux d'utiliser une matrice creuse pour stocker $\tilde{A}$, quel est le nombre d'opérations à faire par itération ?
On peut justifier la convergence de v_k lorsque $C = 0$ (et la vitesse de convergence)

en regardant la norme euclidienne de la matrice $I - D^{-1}\tilde{A}$ en se plaçant dans la base orthonormée de vecteurs propres de $\tilde{A}$. Dans le cas général, on peut appliquer le théorème de convergence de la section sur les méthodes alternatives au pivot du cours. Voir aussi la méthode de Gauss-Seidel et relaxation.