Épreuve orale Math-info de Centrale. Oraux choisis. Corrigé

```
from math import * ; from scipy import *
import numpy as np ; import matplotlib.pyplot as plt
```

Exercice A

1) $E(X) = \frac{1}{r+1}\sum_{k=0}^{r} k = \frac{r}{2}$ et $G_X(z) = \frac{1}{r+1}(1 + z + ... + z^r)$ et $G_{S_{r,n}}(z) = (G_X(z))^n$.

2) `import numpy.random as rd ; from numpy.polynomial import Polynomial`

a)
```
def S(r,n) :
    L = [1/(r+1) for i in range(r+1)]
    return (Polynomial(L)**n).coef

def loi(r,n) :
    L = S(r,n)
    T = [k for k in range(n*r+1)] ; plt.plot(T,L)
loi(2,5) ; plt.show()     ### on fait ici un test
```

b)
```
def uniforme(r) :  return rd.randint(r+1)

def simul(r,n) :
    s = 0
    for i in range(n) :  s = s + uniforme(r)
    return(s)

def loi2(r,n) :
    T = [k for k in range(n*r+1)]
    V = [0]*(n*r+1)
    N = r*n*100
    for k in range(N) : V[simul(r,n)] += 1/N
    plt.plot(T,V)
loi2(2,5) ; plt.show()     # test
```

3) a)
```
def esperance(f,r,n) :
    s = 0 ; N = 1000
    for _ in range(N) : s = s + f(simul(r,n))
    return s/N
```

b)
```
r = 3 ; n = 15 ;
for alpha in [1,2,3] :
    def f(t) :  return t**alpha
    print(esperance(f,r,n)),f(r/2))
```

c) f est lipschitzienne car f' bornée sur le segment $[0,1]$.

On utilise alors $E\left(f\left(\frac{S_{r,n}}{n}\right)\right) - f_\alpha\left(\frac{r}{2}\right) \leq kE\left(\left|\frac{S_{r,n}}{n} - \frac{r}{2}\right|\right) \leq k\sqrt{V\left(\frac{S_{r,n}}{n}\right)}$, car $\frac{r}{2} = E\left(\frac{S_{r,n}}{n}\right)$.

Exercice B

On considère $f_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!} - 1$.

1) On a $f_n(0) = -1$, $f_n(1) > 0$ et $f_n' > 0$ sur $[0,1]$. Donc x_n existe et est unique.

2)
```
def f(n,x) :
    s = -2 ; y = 1
    for k in range(0,n+1) :  s = s + y ; y = y*x/(k+1)
    return s-1
```

Remarque : La fonction `f(n,x)` renvoie $\sum_{k=0}^{n} \frac{x^k}{k!} - 2$.

```
import scipy.optimize as resol
def xx(n) :
    def g(x) :  return f(n,x)  # il faut définir f_n comme une fonction à UNE variable x
    y = resol.fsolve(g,0)     # 0 est la valeur initiale (méthode de Newton)
    return y[0]   # attention : fsolve renvoie une liste (à un seul élément ici)
print(xx(50))   # on vérifie déjà pour une valeur (qui ici est proche de la limite L = e^{-1})
N = [n for n in range(1,51)]
X = [xx(n) for n in N ]
plt.plot(N,X)
```

3) On a $f_{n+1}(x_{n+1}) = 0 = f_n(x_n) < f_{n+1}(x_n)$, et comme f_{n+1} croît strictement, on a $x_{n+1} < x_n$.

Donc $(x_n)_{n\in\mathbb{N}^*}$ est décroissante et minorée, donc elle converge.

Remarque : On a $\lim_{n\to+\infty} f_n(x) = e^x - 2$. Pour prouver que $L = \ln 2$, le plus simple est de prouver que pour tout $\varepsilon > 0$, on a $f_n(L) < 0$ et $\lim_{n\to+\infty} f_n(L+\varepsilon) > 0$, donc $L \leq x_n \leq L + \varepsilon$ pour n assez grand.

Exercice C

```
import scipy.integrate as integr
def F(x) :
    def g(t) :
        return exp(-t*x)/(1+t)
    return integr.quad(g,0,np.inf)[0]
X = [0.01+0.1*k for k in range(101)]
Y = [F(x) for x in X]
plt.plot(X,Y)
```

On conjecture que f est strictement décroissante de $]0,+\infty[$, que $\lim_{x\to 0} F(x) = +\infty$ et $\lim_{x\to+\infty} F(x) = 0$.

On a (cf cours de dérivation des intégrales) $F'(x) = -\int_0^{+\infty} \frac{t}{1+t}\exp(-tx)\ dt < 0$.

On a $\frac{t}{1+t} = 1 - \frac{1}{1+t}$, donc $F'(x) = F(x) - \frac{1}{x}$. On en déduit F de classe C^∞.

Remarque : *Preuve mathématique* : $0 \leq F(x) \leq \int_0^{+\infty} \exp(-tx)\ dt = \frac{1}{x}$, donc $\lim_{x\to+\infty} F(x) = 0$.

Par une IPP, on a $F(x) = \int_0^{+\infty} \frac{\exp(-u)}{x+u}\ du = -\ln(x) + \int_0^{+\infty} \ln(x+u)\exp(-u)\ du$.

Par cv dominée, $\lim_{x\to 0} \int_0^{+\infty} \ln(x+u)\exp(-u)\ du = \int_0^{+\infty} \ln(u)\exp(-u)\ du$ donc $F(x) = -\ln(x) + O(1)$.

Exercice D

1) Le principe est le suivant : On définit une matrice M, puis on regarde si la matrice de passage renvoyée par la fonction `eig(M)` est inversible (dans ce cas, M est inversible).

Remarque : Attention, il convient d'écrire $|\det(M)| < 10^{-6}$ plutôt que $\det(M) = 0$.

On utilise une fonction `essai()` qui fait *un* test (et renvoie 1 si M diagonalisable), et ensuite on moyenne sur 1000 essais. On obtient donc le programme suivant :

```
import numpy.linalg as alg ; import numpy.random as rd

def essai() :
     X,Y = rd.geometric(0.5,2) ; M = np.array([[0,X-Y],[X+Y,0]])
     V,P = alg.eig(M)
     if abs(alg.det(P)) < 10**(-6) :
         return 0
     else :  return 1
print(essai()) # on fait un test pour s'assurer que le programme tourne
s = 0
for i in range(1000) :  s = s + essai()
print(s/1000) # renvoie 0.69
```

2) Le polynôme caractéristique de M est $\lambda^2 - \Delta$, où $\Delta = X^2 - Y^2$.

Il admet deux racines distinctes ssi $\Delta \neq 0$. Dans ce cas M est diagonalisable dans $\mathcal{M}_2(\mathbb{C})$.

Il admet $\lambda = 0$ comme racine double si $\Delta = 0$. Si M était diagonalisable, elle serait la matrice nulle, ce qui est impossible ici car X et Y ne peuvent être nuls (attention, en effet, elles sont à valeurs dans $\mathbb{N}^*$).

Donc M est diagonalisable ssi $\Delta \neq 0$, donc $1 - r = \sum_{n=1}^{+\infty} P(X = n, Y = n) = \sum_{n=1}^{+\infty} (q^{n-1}p)^2 = \frac{p^2}{1-q^2} = \frac{p}{1+q}$.

On en conclut que $r = 1 - \frac{p}{1+q} = \frac{2q}{1+q}$. Lorsque $p = q = \frac{1}{2}$, on obtient $r = \frac{2}{3} \approx 0.67$

3) M est diagonalisable dans $\mathcal{M}_2(\mathbb{R})$ ssi $\Delta > 0$, c'est-à-dire $X > Y$.

On a $P(X > Y) = \sum_{n=1}^{+\infty} P(Y = n)P(X > n) = \sum_{n=1}^{+\infty} (q^{n-1}p)(q^n) = \frac{pq}{1-q^2} = \frac{q}{1+q} = \frac{1}{2}r$.

Remarque : Le résultat $\frac{1}{2}r$ est logique, car $X^2 - Y^2$ et $Y^2 - X^2$ ayant même loi, on a $P(\Delta > 0) = P(\Delta < 0)$.

Exercice E

```
import numpy.linalg as alg ; from numpy.polynomial import *
```

1) H_n est réelle et symétrique, donc diagonalisable.

2)
```
def factorielle(n) :
    if  n == 0 :  return 0
    else return n*factorielle(n-1)
def H(n) :
    M = np.zeros((n+1,n+1))
    for i in range(n+1) :
        for j in range(n+1) :
            M[i,j] = factorielle(i+j)
    return M
for n in range(1,4) :
    M = H(n)
    print("pour n=", n)
    print("det = ", n,alg.det(M), " ; valeurs propres = ", alg.eigvals(M))
```

3)
```
from numpy.polynomial import Polynomial
import scipy.integrate as integr
def ps(P,Q) :
    def f(t) :  return P(t)*Q(t)*exp(-t)
    return integr.quad(f,0,np.inf)[0]
```

4) Si on considère la matrice P_n exprimant la base $(1, X, ..., X^n)$ dans une base orthonormée, on a $H_n = P_n^T P_n$.

Pour obtenir une telle base orthonormée $(B_0, ..., B_n)$, on peut utiliser le procédé de Gram-Schmidt : il permet de calculer la matrice Q_n de $(B_0, ..., B_n)$ dans la base canonique. Donc on peut prendre $P_n = Q_n^{-1}$:

```
n = 3
X = Polynomial([0,1])
B = [ X**j for j in range(n+1)]
for j in range(n+1) :
    for i in range(j) :
        B[j] = B[j] - ps(X**j,B[i])*B[i]
    B[j] = B[j] / ps(B[j],B[j])**0.5
print(B[3].coef)   ### on affiche un polynôme par ses coefficients
Q = np.zeros((n+1,n+1))
    for j in range(n+1) :
```

```
        for i in range(j+1) :
            Q[i,j] = B[j].coef[i]
P = alg.inv(Q)
```

Exercice F

1) On considère le système d'ordre 1 associé : $\begin{cases} y_0' = y_1 \\ y_1' = -y_1/x - y_0 \end{cases}$, et on représente les (x, y_0)

```
import scipy.integrate as integr

def f(Y,x) :  return np.array([Y[1],-Y[0]-Y[1]/x])
X = np.arange(0.001,20,0.01) ; Y0 = np.array([1,0]) # prendre 0.001 car sinon problème en 0

V = np.array(integr.odeint(f, Y0 , T))
Y = V[:,0]
plt.plot(X,Y) ; plt.show()
```

2) On approche $B(x)$ par $B_N(x) = \sum_{n=0}^{N-1}(-1)^n \dfrac{1}{4^n(n!)^2}x^{2n}$, avec $|B(x) - B_N(x)| \leq \dfrac{x^{2N}}{4^n(N!)^2}$ par le CSSA (le terme général est décroissant pour $n \geq N$ lorsque $x^2 \leq 4N^2$, ce qui est le cas dans la valeur qu'on va trouver ensuite pour $x = 20$ et $N = 37$).

On prend donc N tel que $\dfrac{20^{2N}}{4^n(N!)^2} \leq 10^{-12}$, c'est-à-dire $\dfrac{10^N}{N!} \leq 10^{-6}$.

```
N = 1 ; x = 10
while x > 10**(-6) :  N = N+1 ; x = N*10/n
print(N)   # renvoie 37

def B(x) :
     s = 1 ; a = 1 ; m = 1
     for n in range(1,N) :
     a = (- a) * x**2 / 4 / n **2 ; s = s + a
              # on calcule les coefficients a_n en utilisant la récurrence a_0 = 1 et a_n = -a_{n-1}x^2/(4n^2)
     return(s)
print(B(2))   #   test

X = [ 0.1*k for k in range(201) ] ; B = np.array([B(x) for x in X])
plt.plot(X,B) ; plt.show()
```