Séance 10 :

Récursivité

1. Triangle de Pascal

(a) ***Construire une fonction*** `pascal(n, p)` ***qui renvoie le triangle de pascal de*** $(n+1)$ ***lignes et*** $(p+1)$ ***colonnes par récursivité en utilisant des listes, et la méthode*** `extend` ***sur les listes.***

```
def pascal(n, p):
    if (n == 0):
        return [[1] + [0]*p]
    t = pascal(n-1, p)
    newline = [[1] + [t[-1][k] + t[-1][k-1] for k in range(1, p+1)]]
    t.extend(newline)
    return t
```

(b) ***Afficher le triangle sous la forme d'une pyramide.***

```
def Triangle(n):
    # On recupere les lignes du triangle de Pascal
    P = pascal(n, n)
    LignesTemp = []
    # Mettre en forme 1 par 1
    for ligne in P:
        # On commence par suppimer tous les zeros
        while (0 in ligne):
            ligne.remove(0)

        # On transforme les listes en chaine de caracteres
        a = ''
        for num in ligne:
            a += f'{num} '
        LignesTemp.append(a)

    # On recupere la longueur de la derniere ligne (la plus longue)
    h = len(LignesTemp[n]) + 1

    # Maintenant on va rajouter les espaces en debut de lignes pour centrer
    for ligne in LignesTemp:
        L = len(ligne) + 1

        # Si la ligne est paire, on rajoute d'abord un underscore au milieu
        if (L % 2 == 0):
            ligne = ligne[:L//2-1] + '_' + ligne[L//2-1:]

        # Nombre d'espace a rajouter
        nbSpaces = (h - L) // 2

        # On rajoute les espaces
        ligne = ' ' * nbSpaces + ligne

        # On affiche la ligne
        print(ligne)
```

En exécutant `Tiangle(12)`, on obtient :

```
                        1
                      1  1
                    1  2  1
                  1  3  3  1
                1  4  6  4  1
              1  5  10  10  5  1
            1  6  15  2_0  15  6  1
          1  7  21  35  35  21  7  1
        1  8  28  56  7_0  56  28  8  1
      1  9  36  84  126  126  84  36  9  1
    1  10  45  120  210  252  210  120  45  10  1
  1  11  55  165  330  462  462  330  165  55  11  1
1  12  66  220  495  792  924  792  495  220  66  12  1
```

2. Dichotomie

(a) ***Ecrire une fonction récursive*** `dicho(f, a, b, eps)` ***qui calcule par dichotomie la racine de*** f ***entre*** $a < b$ ***sachant que*** f ***est continue et*** $f(a) \times f(b) < 0$ ***à une erreur eps près.***
Avant tout j'importe `numpy` et `pylab` :

```python
import numpy as np
import matplotlib.pylab as plt
```

Puis je code la fonction demandée :

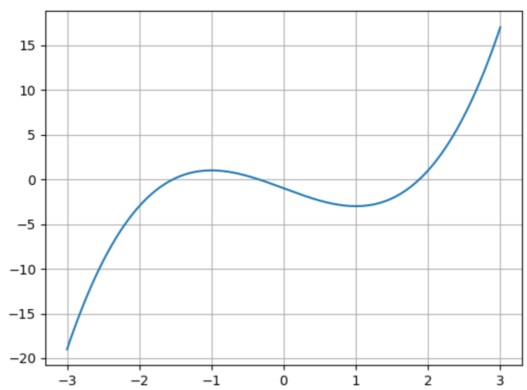
```python
def dicho(f, a, b, eps):
    if(abs(a-b) < 2*eps):
        return((a+b)/2)
    m = (a+b)/2
    if(f(a)*f(m) < 0):
        return (dicho(f, a, m, eps))
    return (dicho(f, m, b, eps))
```

(b) ***Par exemple, trouver les trois racines de*** $P(x) = x^3 - 3x - 1$ ***à*** `eps` $= 10^{-5}$ ***près.***

❏ D'abord je trace la fonction à étudier :

```python
def Trace(f, borneMin, borneMax):
    X = np.linspace(borneMin, borneMax, 1000)
    Y = f(X)
    plt.plot(X, Y)
    plt.grid(True, which='both')
    plt.show()


Trace(f, -3, 3)
```



- Je note grossièrement les 3 intervalles contenant les racines, et la précision de recherche :

```python
# Parametres
INTERVAL = [(-2, -1), (-1, 1), (1, 2)]
PRES = 10**(-5)
```

- Je calcule les racines à 10^{-5} près :

```python
def getRooth(intervals, eps):
    racines = []
    for a, b in intervals:
        racines.append(dicho(f, a, b, eps))

    return racines


getRooth(INTERVAL, PRES)
```

Je trouve :

$$\begin{cases} \alpha &= \texttt{-1.5320816040039062} \\ \beta &= \texttt{-0.34729766845703125} \\ \gamma &= \texttt{1.8793869018554688} \end{cases}$$

(c) ***On reprend l'exemple du DS 3 :***

$$R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 3 \\ 0 & 1 & 0 \end{pmatrix}$$

Définir les matrice R et D diagonales telles que R semblable à D ainsi que la matrice de passage P. Calculer par la diagonalisation M^{10} et vérifier que l'erreur sur les coefficients est en pourcentage des plus grands coefficients de l'ordre de eps.
Vérifier que l'erreur pour M^n est de l'ordre de $\frac{n}{10}eps$ pour $n \in \{20; 50; 100; 1000\}$

- J'ai :

$$R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 3 \\ 0 & 1 & 0 \end{pmatrix} \quad D = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{pmatrix} \quad P = \begin{pmatrix} 1 & 1 & 1 \\ \alpha^2 & \beta^2 & \gamma^2 \\ \alpha & \beta & \gamma \end{pmatrix}$$

De plus, $M = PDP^{-1} \Rightarrow M^n = PD^nP^{-1}$ avec $M = R$ à 10^{-5} près.

- Je définit la fonction `getRPowN` qui calcule $M^n = PD^nP^{-1}$:

```python
def getRPowN(racines, n):
    # On definit D grace aux valeurs propres
    Dn = np.array([
        [racines[0]**n, 0, 0],
        [0, racines[1]**n, 0],
        [0, 0, racines[2]**n]
    ])

    # On definit la matrice de passage P grace aux racines
    P = np.array([[1, 1, 1],
                  [racines[0]**2, racines[1]**2, racines[2]**2],
                  [racines[0], racines[1], racines[2]]])

    # On cacule l'inverse de P grace a numpy
    P1 = np.linalg.inv(P)

    # On retourne M puissance n
    return np.dot(np.dot(P, Dn), P1)
```

❏ De même, je définit la fonction `puiss` qui calcule R^n :

```
def puiss(A, n):
    U = np.eye(3)
    for i in range(n):
        U = np.dot(U, A)
    return U
```

❏ Je définit la fonction qui calcule l'erreur en pourcentage :

```
def getError(n):
    # On definit R
    R = np.array([[0, 0, 1], [1, 0, 3], [0, 1, 0]])

    # On cherche les valeurs propres de R
    racines = getRooth(INTERVAL, PRES)

    # On calcule les matrices a la puissance 10
    Mn = getRPowN(racines, n)
    Rn = puiss(R, n)

    # On calcule l'erreur absolue :
    Er_abs = np.abs(Mn - Rn)

    # On calcule l'erreur relative :
    A = np.max(Rn)
    B = np.max(Er_abs)

    # On calcule le pourcentage :
    per = B / A / PRES

    return float(f'{per:.2f}')
```

❏ Je calcule l'erreur pour `n = 10` et trouve `1.62 %`, ce qui correspond à ce qu'il fallait trouver.

❏ Je recommence, mais n fois, avec $n \in \{20, 50, 100, 1000\}$, et je trace les résultats :

```
def TraceError(N):
    X, Y = [], []

    for n in N:
        er = getError(n)
        X.append(n)
        Y.append(er)
    plt.plot(X, Y, 'rx')
    plt.show()
```

`TraceError([20, 50, 100, 1000])` donne une droite de pente $\frac{eps}{10}$: