TP1correction

December 18, 2020

TP1 - Initiation à Python - Correction

[1]:
```
import numpy as np
from math import *
import matplotlib.pyplot as plt
```

1 - Manipulation de vecteurs et matrices

“x” est un vecteur ligne.

“y” est un vecteur colonne.

La transposition de “Y” reste un vecteur ligne car c’est un tableau 1D, alors que la transposition de “YY” est un vecteur colonne car c’est un tableau 2D.

La différence entre “z” et “Z” est le typage de l’objet, on a respectivement une matrice et un tableau 2D.

La différence entre “t” et “u” est que “np.linspace” inclut les deux bornes contrairement à “np.arange” qui exclut la deuxième borne (à l’instar des boucles “for”).

La commande “np.zeros_like(y)” permet de construire une matrice de la même dimension que “y” remplie de zéros.

Il faut faire attention à bien mettre un tuple (a,b) en argument des fonctions “np.zeros” et “np.ones” afin d’avoir des éléments 2D.

Pour extraire des coordonnées on utilise “:” qui exclut la deuxième borne. On peut aussi mettre entre crochet un tableau d’indices comme dans “u[t]”.

La différence entre “z**2” et “Z**2” est que dans le premier cas, on fait le produit matriciel et dans le deuxième on fait une multiplication composante par composante. (même phénomène pour “z*np.eye(2)” et “Z*np.eye(2)”).

Dès qu’on a des “np.array” on fait les opérations composante par composante.

Exercice 1. Définition de matrices.

a.

[2]:
```
P = np.zeros((5,5))
for i in range (5):#boucle pour i entre 0 et 4 (inclus)
    for j in range (5):
        P[i,j] = cos(i * pi / 5) + sin(j * pi / 7)
```

```
P
```

[2]:
```
array([[ 1.        ,  1.43388374,  1.78183148,  1.97492791,  1.97492791],
       [ 0.80901699,  1.24290073,  1.59084848,  1.78394491,  1.78394491],
       [ 0.30901699,  0.74290073,  1.09084848,  1.28394491,  1.28394491],
       [-0.30901699,  0.12486674,  0.47281449,  0.66591092,  0.66591092],
       [-0.80901699, -0.37513326, -0.02718551,  0.16591092,  0.16591092]])
```

b.

[3]:
```
C = np.array([cos(j * pi / 5) for j in range (5)]) #ligne des cinq valeurs␣
↪cos(j pi/5)
D = np.array([sin(i * pi / 7) for i in range (5)]) #ligne des cinq valeurs␣
↪sin(i pi/7)
E = np.ones((5, 1)) #une colonne de cinq 1
P = C.T * E.T + E * D #la multiplication à gauche par la colonne de 1 permet␣
↪d'avoir une matrice avec 5 lignes identiques et la multiplication à droite␣
↪par une ligne de 1 permet d'avoir une matrice avec 5 colonnes identiques (on␣
↪rappelle que l'opération A.T renvoie la matrice transposée de A)
P
```

[3]:
```
array([[1.        , 1.24290073, 1.09084848, 0.66591092, 0.16591092],
       [1.        , 1.24290073, 1.09084848, 0.66591092, 0.16591092],
       [1.        , 1.24290073, 1.09084848, 0.66591092, 0.16591092],
       [1.        , 1.24290073, 1.09084848, 0.66591092, 0.16591092],
       [1.        , 1.24290073, 1.09084848, 0.66591092, 0.16591092]])
```

c.

[4]:
```
L = 2 * np.diag(np.ones(10), 0) - np.diag(np.ones(9), -1) - np.diag(np.ones(9),␣
↪1)
L
```

[4]:
```
array([[ 2., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [-1.,  2., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0., -1.,  2., -1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0., -1.,  2., -1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1.,  2., -1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0., -1.,  2., -1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0., -1.,  2., -1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0., -1.,  2., -1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  2., -1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  2.]])
```

Exercice 2. Précision machine.

a.

[5]:
```
U  = np.array([1 + 2**(-n) for n in range (61)])
U
```

[5]:
```
array([2.        , 1.5       , 1.25      , 1.125     , 1.0625    ,
       1.03125   , 1.015625  , 1.0078125 , 1.00390625, 1.00195312,
       1.00097656, 1.00048828, 1.00024414, 1.00012207, 1.00006104,
       1.00003052, 1.00001526, 1.00000763, 1.00000381, 1.00000191,
       1.00000095, 1.00000048, 1.00000024, 1.00000012, 1.00000006,
       1.00000003, 1.00000001, 1.00000001, 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        ])
```

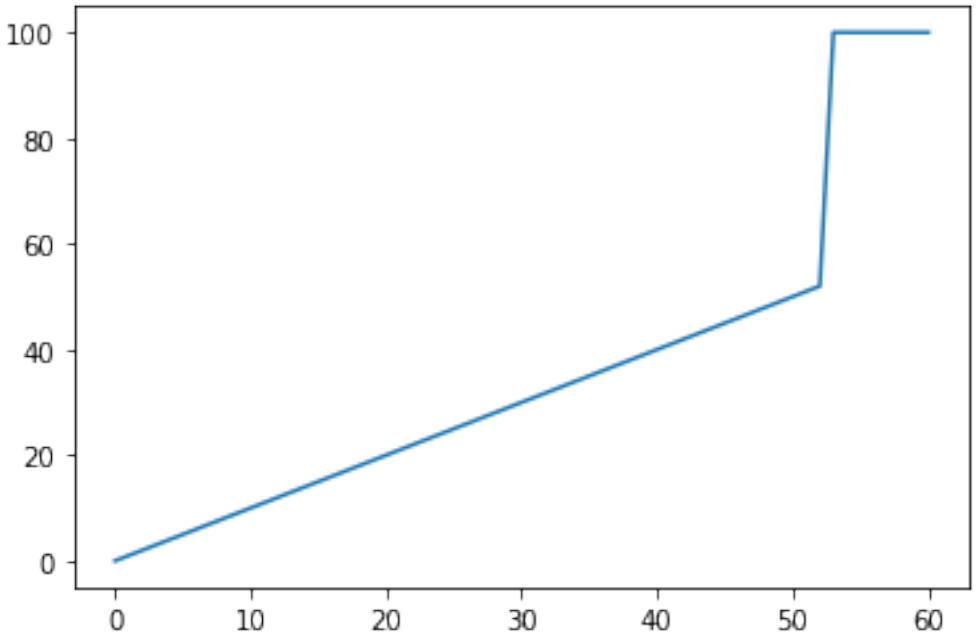
b.

[6]:
```
V = U - 1
V
```

[6]:
```
array([1.00000000e+00, 5.00000000e-01, 2.50000000e-01, 1.25000000e-01,
       6.25000000e-02, 3.12500000e-02, 1.56250000e-02, 7.81250000e-03,
       3.90625000e-03, 1.95312500e-03, 9.76562500e-04, 4.88281250e-04,
       2.44140625e-04, 1.22070312e-04, 6.10351562e-05, 3.05175781e-05,
       1.52587891e-05, 7.62939453e-06, 3.81469727e-06, 1.90734863e-06,
       9.53674316e-07, 4.76837158e-07, 2.38418579e-07, 1.19209290e-07,
       5.96046448e-08, 2.98023224e-08, 1.49011612e-08, 7.45058060e-09,
       3.72529030e-09, 1.86264515e-09, 9.31322575e-10, 4.65661287e-10,
       2.32830644e-10, 1.16415322e-10, 5.82076609e-11, 2.91038305e-11,
       1.45519152e-11, 7.27595761e-12, 3.63797881e-12, 1.81898940e-12,
       9.09494702e-13, 4.54747351e-13, 2.27373675e-13, 1.13686838e-13,
       5.68434189e-14, 2.84217094e-14, 1.42108547e-14, 7.10542736e-15,
       3.55271368e-15, 1.77635684e-15, 8.88178420e-16, 4.44089210e-16,
       2.22044605e-16, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00])
```

c.

[7]:
```
X = np.arange(61)
Y = -np.log(U - 1 + 2**(-100)) / np.log(2)
plt.plot(X,Y)
plt.show()
```



d. L'ordinateur fait des erreurs machine car sa précision n'est pas assez bonne. C'est le même défaut que l'on observe quand on demande à l'ordinateur de calculer 0.1 + 0.2, il renvoie 0.30000000000000004. Cela vient du fait qu'il représente les nombres à virgule (que l'ordinateur appelle flottants) par 64 bits donc 1bit représente le signe, 11 représentent l'exposant et 52 représentent le nombre (la mantisse) qu'il faudra multiplier par 2^e où e est l'exposant. En effet, ces flottants représentent un nombre fini de réels et donc dans certains cas, l'ordinateur fait des erreurs d'arrondi, de plus il existe un réel très petit en dessous duquel l'ordinateur ne peut pas le distinguer par rapport à zéro (à peu près 10^{-16} pour la représentation en 64 bits).

Exercice 3. Quotient de Rayleigh d'une matrice hermitienne.

a.

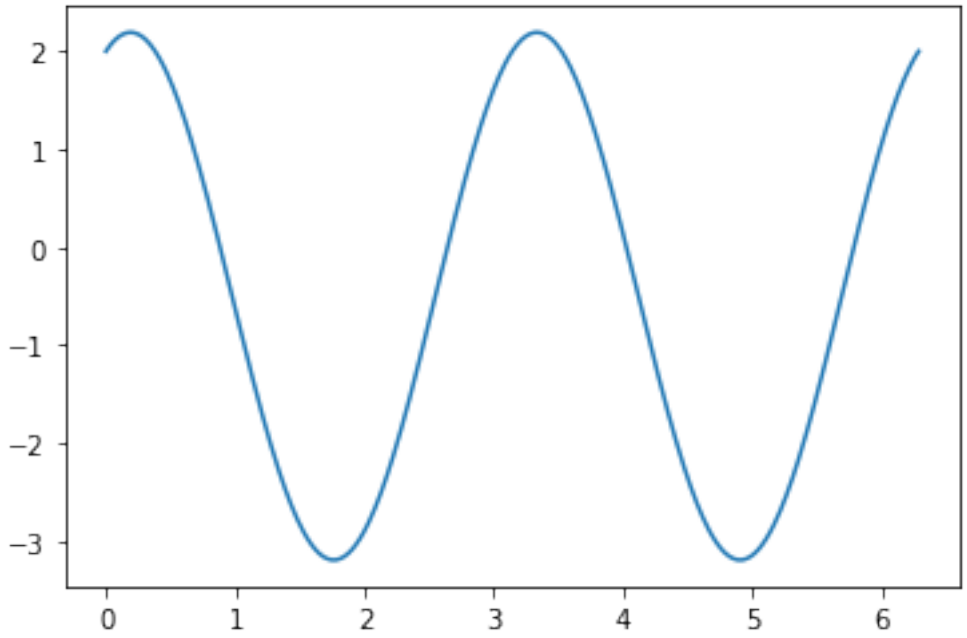
[8]:
```
def Rayleigh(A, x): #A et x sont de type np.array
    num = np.vdot(A.dot(x), x)
    den = np.vdot(x, x)
    return num / den
```

b.

[9]:
```
A = np.array([[2, 1], [1, -3]])
B = np.array([[2, 1j],[-1j, -3]])
valpropA, vectpropA  = np.linalg.eig(A)
valpropB, vectpropB = np.linalg.eig(B)
```

c.

[10]:
```
Theta = np.linspace(0, 2*pi, 400)
Y = [Rayleigh(A, np.array([np.cos(t), np.sin(t)])) for t in Theta]
plt.plot(Theta, Y)
plt.show()
```



[11]:
```
valpropA
```

[11]:
```
array([ 2.1925824, -3.1925824])
```

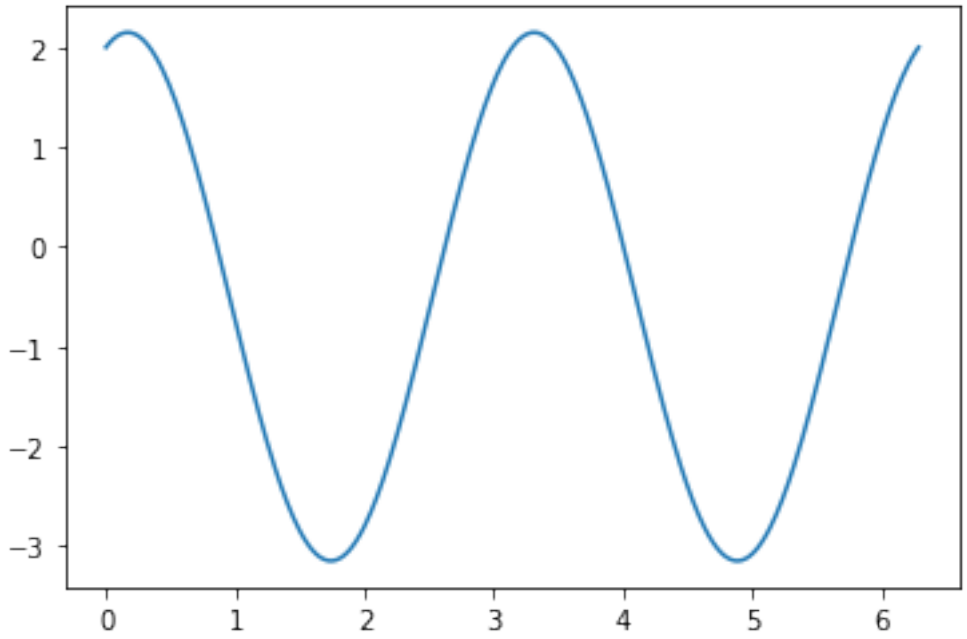
La courbe oscille entre les deux valeurs propres. En effet, on a une propriété qui dit que

$$\lambda_{min} \leqslant R_A(x) \leqslant \lambda_{max}$$

Ainsi, comme $x(\theta)$ parcourt tous les vecteurs de $\mathbb{R}^2$ de norme 1, on peut observer les deux valeurs propres directement sur le graphique en prenant les bornes inférieures et supérieures de la courbe.

d.

[12]:
```
phi = 0.5
Theta = np.linspace(0, 2*pi, 400)
Y = [(Rayleigh(A, np.array([cos(t) * e**(1j*phi), sin(t)]))).real for t in␣
 ↪Theta]
plt.plot(Theta, Y)
plt.show()
```



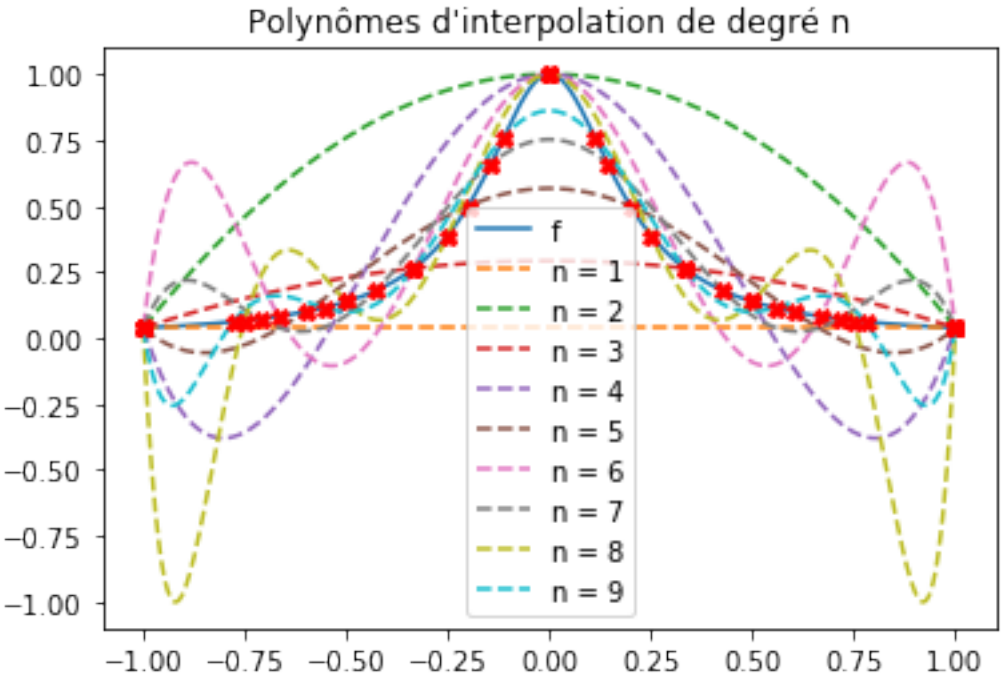
On remarque la même chose qu'à la question c. Ici le vecteur $z(\theta, \phi)$ parcourt tous les vecteurs de $\mathbb{C}^2$ de norme 1.

Exercice 4. Interpolation polynomiale.

a.

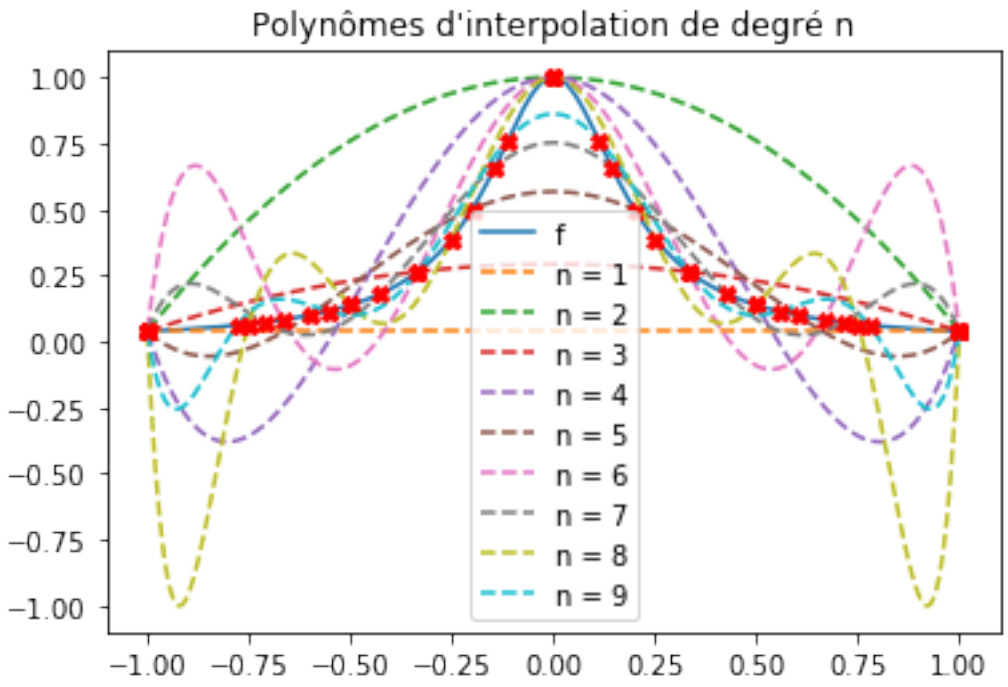
[13]:
```
def f(x): #définition de la fonction qu'on cherche à approcher
    y = 1 / (1 + 25 * x**2)
    return y


def diffdiv(x, y): #création de la matrice qui contient les taux␣
 ↪d'accroissement généralisé qui contient en case (i,j) (pour 0 i,j n avec␣
 ↪i+j n) l'élément f[x_i, ..., x_{i+j}] ainsi sur la première ligne on obtient␣
 ↪les f[x_0,...,x_j] qui vont servir à calculer les polynômes de Lagrange
    n = len(x) - 1
    D = np.zeros((n+1, n+1))
    D[:,0] = y
    for j in range (1, n+1):
        for i in range (n-j+1):
            D[i,j] = (D[i+1,j-1] - D[i,j-1]) / (x[i+j] - x[i])
    return D

def fastexp(x, D, t): # méthode de Horner pour calculer le polynôme en la␣
 ↪valeur t en utilisant les f[x_0, ..., x_j] sur la première ligne de D et le␣
 ↪vecteur x = [x_0,...,x_n]
```

```
    n = len(x) - 1
    y = D[0, n] * np.ones(len(t))
    for k in range (n-1, -1, -1):
        y = (t-x[k]) * y + D[0, k]
    return y


x = np.linspace(-1, 1, 400)
y = f(x)
plt.plot(x, y, label = "f")

for n in range (1, 10):
    X = np.linspace(-1, 1, n+1) #les points [x_0,...,x_n]
    Y = f(X) #les points [f(x_0), ..., f(x_n)]
    D = diffdiv(X, Y)
    y = fastexp(X, D, x) #le polynome de Lagrange
    plt.plot(x, y, '--', label = "n = " + str(n))
    plt.plot(X, Y, 'Xr')
plt.legend(loc = "best")
plt.title("Polynômes d'interpolation de degré n")
plt.show()
```



b.

[14]:
```
def f(x):
    y = 1 / (1 + 25 * x**2)
    return y



def pol(val, polynome): #évaluation du polynôme en la valeur val par la méthode␣
 ↪de Hörner
    if len(polynome) == 0:
        return 0
    else:
        somme = polynome[-1]
        for i in range (len(polynome)-2, -1, -1):
            somme = somme*val + polynome[i]
    return somme



x = np.linspace(-1, 1, 400)
y = f(x)
plt.plot(x, y, label = "f")#tracé de la fonction f


for n in range (1,10):
    X = np.linspace(-1, 1, n+1)
    Y = f(X)
    V = np.ones((n+1, n+1)) #matrice remplie de 1
    for j in range (1, n+1):
        V[:,j] = X * V[:, j-1] #construction de la matrice de VanDerMonde
    coeff = np.linalg.solve(V, Y) #résolution du système pour trouver les␣
 ↪coefficients du polynôme d'interpolation de Lagrange
    x = np.linspace(-1, 1, 400)
    y = pol(x, coeff)
    plt.plot(x, y, '--', label = "n = " + str(n))#tracé du polynôme
    plt.plot(X, Y, 'Xr')#tracé des points qui ont servis à trouver le polynôme
plt.legend(loc = "best")
plt.title("Polynômes d'interpolation de degré n")
plt.show()
```



c. On obtient le même résultat que dans la fiche mémo. On a calculé de deux façons différentes les polynômes de Lagrange qui sont uniques puisque par $n+1$ points il passe un unique polynôme de degré au plus n.

d. On utilise la base de Lagrange qui décrit le polynôme de Lagrange de degré n passant par les points $(x_0, y_0), \cdots, (x_n, y_n)$ par

$$L(X) = \sum_{j=0}^{n} y_j \prod_{i=0, i\neq j}^{n} \frac{X - x_i}{x_j - x_i}.$$

[15]:
```
def f(x):
    y = 1/(1+25*x**2)
    return y


def Lagrange(val, X, Y):#calcul direct du polynôme de lagrange au point val
    somme = 0
    for j in range (len(X)):
        produit = 1
        for i in range(len(X)):
            if i != j:
                produit *= (val - X[i])/(X[j] - X[i])
        somme += produit * Y[j]
    return somme
```

```
x = np.linspace(-1,1,400)
y = f(x)
plt.plot(x,y, label = "f")#tracé de la fonction f

for n in range (1,10):
    X = np.linspace(-1,1,n+1)
    Y = f(X)
    x = np.linspace(-1,1,400)
    y = Lagrange(x, X, Y)
    plt.plot(x,y,'--',label = "n = "+str(n))#tracé du polynôme
    plt.plot(X,Y,'Xr')#tracé des points qui ont servis à trouver le polynôme
plt.legend(loc="best")
plt.title("Polynômes d'interpolation de degré n")
plt.show()
```