Devoir Surveillé 3 - Corrigé

1 Équation de Bessel

Les fonctions de Bessel, découvertes par le mathématicien suisse Daniel Bernoulli, portent le nom du mathématicien allemand Friedrich Wilhelm Bessel. Bessel développa l'analyse de ces fonctions en 1816 dans le cadre de ses études du mouvement des planètes induit par l'interaction gravitationnelle, généralisant les découvertes antérieures de Bernoulli. Ces fonctions sont des solutions canoniques $y(x)$ de l'équation différentielle de Bessel :

$$t^2\frac{\mathrm{d}^2y}{\mathrm{d}t^2}+t\frac{\mathrm{d}y}{\mathrm{d}t}+(t^2-\alpha^2)y=0,$$

où α est un paramètre réel ou complexe. Le plus souvent, α est un entier naturel (on dit alors que c'est l'ordre de la fonction), ou un demi-entier (la moitié d'un entier). C'est de toute façon une **constante**.

On travaillera avec $t>0$, si bien que l'équation s'écrira :

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2}+\frac{1}{t}\frac{\mathrm{d}y}{\mathrm{d}t}+\frac{t^2-\alpha^2}{t^2}y=0.$$

On s'intéresse ici aux fonctions de Bessel dites *de première espèce*, J_n, qui sont solutions de l'équation de Bessel avec les conditions initiales $y(1)=y_0$ et $y'(1)=0$.

Dans l'écriture des fonctions,on n'oubliera pas d'importer les modules nécessaires.

Question 1. *On pose $z=y'$. Écrire le système d'équations différentielles d'ordre 1 vérifié par y et z.*

Le système est $\begin{cases} y'(t)=z(t) \\ z'(t)=-\dfrac{1}{t}z(t)+\dfrac{\alpha^2-t^2}{t^2}y(t) \end{cases} \iff \begin{pmatrix} y \\ z \end{pmatrix}'(t)=\begin{pmatrix} 0 & 1 \\ \dfrac{\alpha^2-t^2}{t^2} & -\dfrac{1}{t} \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix}.$

Question 2. *On intègre cette équation de manière approchée sur [1,t_f] avec la méthode d'Euler explicite avec n pas de même longueur h. Montrer que l'on alors les relations de récurrences suivantes avec $0\leqslant k\leq n$:*

$$y_{k+1}=y_k+hz_k \text{ et } z_{k+1}=(1-\frac{h}{t_k})z_k+h\frac{\alpha^2-t_k^2}{t_k^2}y_k. \tag{1}$$

Que représentent y_k et z_k ? Préciser également la valeur de h en fonction de t_f et n et les valeurs de k considérées.

La méthode d'Euler explicite au premier ordre consiste à approcher $y'(t_k)$ par $\dfrac{y_{t_{k+1}}-y(t_k)}{t_{k+1}-t_k}$, où $t_k=1+kh$ et $h=\dfrac{t_f-1}{n}$. On note y_k (resp. z_k) la valeur approchée de $y(t_k)$ (resp. $z(t_k)$ obtenue par cette méthode. Ainsi, $k\in[\![0,n]\!]$.

En remplaçant dans le système de deux équations $y'(t_k)$ et $z'(t_k)$ par leurs approximations, on obtient bien les relations de récurrence (1).

Question 3. *Écrire une fonction `baset(t0, tf, n)` qui renvoie une liste `T` contenant les temps t_i pour lesquels on calcule les valeurs approchées de y_i et z_i.*

```
import numpy as np
def baset(tf, n): # -> tableau numpy unidimensionnel
    return np.linspace(1, tf, n+1)
```

```
def baset(tf, n): # -> liste
    return [1+k*h for k in range(n+1)]
```

Question 4. *Écrire une fonction `Bessel(tf, y0, alpha,n)` qui prend en argument les flottants `tf`, `y0`, `z0` et `alpha` qui représentent respectivement les temps finaux, $y(1)$ et $y'(1)$ et la constante α et l'entier `n` qui représente le nombre de pas d'intégration, et qui renvoie `Y` et `Z`, deux listes contenant les valeurs des y_i et z_i pour i allant de 0 à n, à l'aide du schéma d'Euler explicite.*

```
def Bessel(tf, y0, alpha, n):
    Y, Z, h = [y0], [0], (tf-1)/n
    T = baset(tf,n)
    for k in range(n):
        Y.append(Y[k] + h*Z[k])
        Z.append((1-h/T[k])*Z[k] + h*Y[k]*(alpha/T[k]**2 -1))
    return Y, Z
```

Question 5. *On prend $t_f = 30$, $y0 = 2$, $\alpha = 1$. Écrire la suite d'instructions qui vous permet d'obtenir la courbe représentative de y en fonction de t. On choisira une valeur de n plausible.*

```
import matplotlib.pyplot as plt
T = baset(30, 1000)
Y, Z = Bessel(30, 2, 1, 1000) #Attention au type renvoyé par la fonction Bessel
plt.figure()
plt.title("$Y$ en fonction de $t$")
plt.plot(T, Y)
plt.show()
```

Question 6. *Avec les mêmes valeurs que précédemment donner les commandes permettant de tracer le portrait de phase : il s'agit de la courbe paramétrée définie par les points $(y(t), y'(t))$.*

```
plt.figure()
plt.title("Portrait de phase")
plt.plot(Y, Z)
plt.show()
```

2 Algèbre linéaire et numpy

Cet exercice sera traité en utilisant le type `array` du module `numpy`

Question 7. *Écrire une fonction `transpose` qui prend en argument une matrice M (pas nécessairement carrée) et renvoie la matrice transposée M^T.*

```
def transpose(M): # boucle
    n, p = np.shape(M)
    N = np.zeros((p,n))
    for i in range(p):
        for j in range(n):
            N[i,j] = M[j,i]
    return M
```

```
def transpose(M): # compréhension
    n, p = M.shape
    return np.array([[M[i,j] for j in range(p)] for i in range(n)])
```

Question 8. *Écrire une fonction `test` qui prend en argument un couple de matrices (A, M), renvoie un message d'erreur si A et M ne sont pas des matrices carrées de même taille et la matrice $AM + MA^T$ sinon.*

```
def test(A, M):
    chaipe = A.shape
    assert M.shape == chaipe and chaipe[0] == chaipe[1]
    return np.dot(A, M) + np.dot(M, transpose(A))
```

Question 9. *Écrire une fonction* `base_canonique` *d'argument un entier naturel n et qui renvoie une liste dont les éléments sont, dans l'ordre usuel, les éléments de la base canonique de $\mathcal{M}_n(\mathbb{R})$.*

```
def base_canonique(n):
    L = []
    for i in range(n):
        for j in range(n):
            M = np.zeros((n,n))
            M[i,j] = 1
            L.append(M)
    return L
```

Question 10. *Écrire une suite d'instructions définissant la matrice* $A = \begin{pmatrix} 1 & 2 & -3 \\ 4 & 2 & 0 \\ 5 & 6 & 1 \end{pmatrix}$ *et générant la matrice de l'application $M \longmapsto AM + MA^T$ dans la base canonique de $\mathcal{M}_n(\mathbb{R})$.*

`A = np.array([1, 2, 3, 4, 2, 0, 5, 6, 1]).reshape(3,3)` ou bien
`A = np.array([[1, 2, -3], [4, 2, 0], [5, 5, 1]])`

```
bc = base_canonique(3)
col = [test(A,M) for M in bc]
Mat = transpose(np.array([c.reshape(9) for c in col]))
```

3 Traitement d'images

Une image en couleur acquise par une caméra est constituée de trois couches (RGB) (pour *red, green, blue*).

Les données de l'image sont stockées dans un tableau à trois dimensions : la première dimension correspond à la couleur exprimant, dans l'ordre, la quantité de rouge, de vert et de bleu, ces quantités étant codées par un entier naturel de l'intervalle $[\![0, 255]\!]$. La deuxième dimension correspond à la coordonnée selon x et la troisième à la coordonnée selon y. Ainsi, les dimensions du tableau sont : $3 \times m \times n$, où, ici, $m = 800$ et $n = 600$.

Question 11. *Donner la quantité de mémoire nécessaire en octets pour stocker le tableau représentant l'image émise par la caméra en justifiant le codage retenu pour un pixel d'une couche.*

Chaque pixel est codé par un triplet d'entiers non signés de l'intervalle $[\![0, 255]\!] = [\![0, 2^8 - 1]\!]$. On choisit donc le format `uint8`, qui permet de coder chacune des trois couleurs sur un octet.

Pour une image, il faut donc $800 \times 600 \times 3 = 1440000 = 1,44$ Mo.

Question 12. *Dans le cadre du traitement des images acquises, la première étape est de convertir l'image en couleurs en niveaux de gris. La méthode consiste à rechercher le maximum et le minimum pour chaque pixel sur les trois couches, puis à faire la moyenne de ces maxima et minima et ainsi obtenir la valeur du nouveau pixel en niveaux de gris.*

On note `imagecolor` *le tableau représentant une image en couleur.*

Écrire une fonction `grayscale(imagecolor)` *qui renvoie une image en niveaux de gris (qui sera notée* `image` *dans l'algorithme principal), sous forme de tableau à deux dimensions de taille $m \times n$ contenant des valeurs entières comprises entre 0 et 255, en suivant l'algorithme décrit ci-dessus. Il est possible d'utiliser les fonctions* `min` *et* `max` *de Python.*

```
def grayscale(imagecolor):
    image = np.zeros((800, 600)).astype('uint8')
    for i in range(800):
        for j in range(600):
            couleur = imagecolor[0, i, j], imagecolor[1, i, j], imagecolor[2, i, j]
            m, M = min(couleur), max(couleur)
            image[i, j] = (m+M)/2
    return image
```

Il faut bien déclarer le tableau comme de type `uint8` et non comme un `np.zeros((800, 600))`, qui sera automatiquement de type `float64` car, en ce cas, l'affectation `image[i, j] = np.uint8((m+M)/2)` ne sera pas respectée. Un tableau numpy ayant nécessairement tous ses éléments de même type, Python respectera le type flottant initial, les affectations étant individuelles.

Question 13. *On souhaite, à des fins de traitement en temps réel d'images acquises par la caméra, extraire des pixels de l'image initiale.*

- *numH* > 1 *est le nombre de points souhaités horizontalement.*
- *numV* > 1 *est le nombre de points souhaités verticalement.*

Les points doivent être équirépartis sur la zone décrite par la fenêtre.

Un point est défini par ses coordonnées x et y, indices respectifs de la colonne et de la ligne, en prenant comme origine le coin supérieur gauche de l'image, bord inclus.

Une fenêtre (rectangulaire) est décrite par un quadruplet (Px, Py, L, H), où le point situé en haut à gauche de la fenêtre a pour coordonnées (Px, Py), L désignant la largeur de la fenêtre et H sa hauteur.

Écrire une fonction `coordonnees_pts(fenetre, numH, numV)` *qui renvoie une liste des points (représentés par leurs coordonnées) sélectionnés.*

Le parcours des points se fera de haut en bas puis de la gauche vers la droite. On ne traitera pas les cas pour lesquels `numH` *ou* `numV` *sont égaux à 1.*

On fera en sorte que les points situés sur les bords gauche et haut de l'image soient toujours sélectionnés.

```
def coordonnees_pts(fenetre, numH, numV):
    L = []
    Px, Py = fenetre[0], fenetre[1]
    pasH, pasV = fenetre[2]//numH, fenetre[3]//numV
    for i in range(numH):
        for j in range(numV):
            L.append((Px + i*pasH, Py + j*pasV))
    return L
```

```
def coordonnees_pts(fenetre, numH, numV):
    L = [0] * (numH*numV)
    Px, Py = fenetre[0], fenetre[1]
    pasH, pasV = fenetre[2]//numH, fenetre[3]//numV
    for i in range(numH):
        for j in range(numV):
            L[numV*i + j] = Px + i*pasH, Py + j*pasV))
    return L
```

4 Génomique

Les chaînes d'ADN sont représentées par des chaînes de caractères formées à partir des quatre lettres A, C, G, et T, chaque lettre codant un nucléotide. Un codon est une suite de trois nucléotides consécutifs. Par exemple, la chaîne "ACGGTG" contient les quatre codons "ACG", "CGG", "GGT" et "GTG". Deux codons sont situés sur la même phase s'ils sont séparés uniquement par d'autres codons, c'est-à-dire si le nombre de lettres les séparant est un multiple de trois. Dans la mini-chaîne ci-dessus, seuls "ACG" et "GTG" sont situées sur la même phase.

Question 14. *Écrire une fonction* `correction_ADN` *prenant un argument et vérifiant 1) que l'argument est bien une chaîne de caractères, 2) que cette chaîne ne contient pas d'autres lettres que les quatre lettres présentes dans l'ADN. Cette fonction retournera un booléen.*

```
def correction_ADN(adn):
    if type(adn) != str:
        return False
    alphabet = set("ACGT")
    for x in adn:
        if not x in alphabet:
            return False
    return True
```

Question 15. *Écrire une fonction* `codons` *prenant en argument une chaîne d'ADN et retournant la liste des codons présents dans la chaîne, sans les répéter s'ils sont présents plusieurs fois.*

```
def codons(ADN):
    L = []
    n = len(ADN)
    for i in range(n-2):
        w = ADN[i:i+3]
        if not w in L:
            L.append(w)
    return L
```

```
def codons2(ADN):
    return set([ADN[i:i+3] for i in range(len(ADN)-2)])
```

Question 16. *Écrire une fonction* `codons_phase` *prenant en argument une chaîne d'ADN et retournant un tuple formé des trois listes des codons rangés par phase.*

```
def codons_phase(ADN):
    L = [[], [], []]
    q, r = divmod(len(ADN), 3)
    for i in range(3):
        L[i] = [ADN[3*k+i:3*(k+1)+i] for k in range(q-1)]
    for j in range(r+1):
        L[j].append(ADN[3*(q-1)+j:3*q+j])
    return tuple(M for M in L)
```