

## Centrale 2019 : Corrigé

## I. Fonctions utilitaires

**Question 1.** On n'utilise pas la fonction `sum` dans le doute, mais après tout rien n'interdit de l'utiliser...

```
def moyenne(X):
    s=0
    for x in X:
        s+=x
    return s/len(X)
```

**Question 2.** Avec  $n$  le nombre d'éléments de la liste, attention à ne pas faire  $n$  appels à la fonction précédente, sous peine d'avoir une complexité  $O(n^2)$  (pénalisée!)

```
def variance(X):
    m=moyenne(X)
    s=0
    for x in X:
        s+=(x-m)**2
    return s/len(X)
```

Autre version :

```
def variance(X):
    return moyenne([x**2 for x in X]) - moyenne(X)**2
```

**Question 3.** Cette fonction doit être récursive! Bien que  $M$  soit supposée être une liste, le plus simple est de prendre comme cas de base celui où c'est un nombre.

```
def somme(M):
    if isinstance(M, numbers.Real):
        return M
    else:
        s=0
        for y in M:
            s+=somme(y)
        return s
```

Remarque : on peut s'en sortir sans récursivité avec des moyens alambiqués, mais ce n'est pas conseillé. Ceci dit c'est un bon exercice. Un exemple :

```
def somme(M):
    L=[]
    b=True
    while b:
        b=False
        for x in M:
            if isinstance(x, numbers.Real):
                L.append(x)
            else:
                for y in x:
                    L.append(y)
                b=True
        M, L=L, []
    return sum(M)
```

## II. Mesures expérimentales

### II.A. Position de la bille

**Question 4.** Attention à ne pas mélanger 0 et 1...

```
def seuillage(A, s):
    B=np.copy(A)
    for i in range(len(A)):
        for j in range(len(A[0])):
            if A[i][j]<s:
                B[i][j]=1
            else:
                B[i][j]=0
    return B
```

**Question 5.** Les pixels blancs correspondent à un 1 dans l'image seuillée.

```
def pixel_centre_bille(A):
    sx, sy=0,0
    n,m=A.shape
    for i in range(n):
        for j in range(m):
            if A[i][j]==1:
                sx+=i
                sy+=j
    return round(sx/n), round(sy/m)
```

**Question 6.** Il suffit d'appliquer les trois fonctions successivement. Rappel au cas où, `prendre_photo` est la fonction, `prendre_photo()` est un appel à cette fonction (qui ne prend pas d'argument).

```
def positions(n, seuil):
    return [pixel_centre_bille(seuillage(prendre_photo(), seuil)) for _ in range(n)]
```

**Question 7.** En notant  $(x_i)$  et  $(y_i)$  les différentes abscisses et ordonnées, ainsi que  $\bar{x}$  et  $\bar{y}$  les moyennes respectives, ce qu'on veut calculer est la moyenne des  $(x_i - \bar{x})^2 + (y_i - \bar{y})^2$  (distances quadratiques entre les  $(x_i, y_i)$  et la position  $(\bar{x}, \bar{y})$ ). Ce n'est autre que la somme des variances des  $(x_i)$  additionnée à celle des  $(y_i)$ . À ceci s'ajoute le facteur  $t^2$ .

```
def fluctuations(P,t):
    X, Y = [c[0] for c in P], [c[1] for c in P]
    return (variance(X)+variance(Y))*t**2
```

### II.B. Allongement du brin d'ADN

**Question 8.** Cette question est assez technique, expliquez un minimum ce que vous faites, d'autant que l'énoncé n'est pas très clair.

```
def distance(p, q):
    return ((p[0]-q[0])**2+(p[1]-q[1])**2)**0.5

def profil(A,n):
    m,p=A.shape
    C=pixel_centre_bille(A)
    dmax=max([distance(C,p) for p in [(0,0), (0,p-1), (m-1,0), (m-1,p-1)]] #distance max à un pixel
    r=dmax/n #les cercles sont de rayon r,2r,3r,4r,...,nr
    total=[0]*n #pour compter les pixels
    blancs=[0]*n #pour les blancs
    for i in range(m):
        for j in range(p):
            d=distance(C, (i, j))
            total[min(n-1,int(d/r))]+=1
            if A[i][j]==1:
                blancs[min(n-1,int(d/r))]+=1
    return [blancs[i]/total[i] for i in range(n)]
```

Principe :

- on a écrit une fonction distance, qui permet d'obtenir la distance entre deux pixels (c'est également la distance entre les centres des pixels!);
- on convient que le  $n$ -ème cercle doit passer par le pixel le plus éloigné, comme sur l'image. On calcule cette distance  $d_{\max}$  à partir des 4 coins; le rayon du plus petit cercle est  $r = d_{\max}/n$ .
- pour tous les pixels de l'image, on repère à quel « couronne » appartient chacun. Le numéro de la couronne est  $\lfloor d/r \rfloor$ , avec  $d$  la distance du pixel courant au centre de la bille. Remarque : dans le cas  $d = d_{\max}$ , on convient que le numéro est  $n - 1$ , d'où la modification du code pour éviter un dépassement d'indice;
- on convient que « la proportion de pixels blancs dans chaque anneau » signifie le rapport entre le nombre de pixels blancs de l'anneau sur le nombre total de pixels de l'anneau, mais ce n'est pas très clair.

Remarque : en utilisant des tableaux Numpy à la place de listes, on aurait pu écrire `return blancs/total` à la fin.

**Question 9.** La complexité est  $O(p^2 + n)$  ( $= O(p^2)$  si  $n < p$ , ce qui paraît logique).

### III. Modèle du ver

#### III.A. Calcul des paramètres

**Question 10.** Puisqu'on travaille avec des tableaux Numpy, toutes les fonctions sont vectorielles :

```
def force(z, Lp, L0, T):
    return K_B*T/Lp*(1/4/(1-z/L0)**2-1/4+z/L0)
```

Bien sûr, on pouvait écrire :

```
def force(z, Lp, L0, T):
    return np.array([K_B*T/Lp*(1/4/(1-x/L0)**2-1/4+x/L0) for x in z])
```

**Question 11.** Il est nécessaire de définir localement la fonction à optimiser.

```
def ajusteWLC(Fz,T):
    def fT(z, Lp, L0):
        return force(z, Lp, L0, T)
    return scipy.optimize.curve_fit(fT, Fz[:, 1], Fz[:, 0])[0] #[0] pour le premier élément
```

#### III.B. Algorithme du minimum local

##### III.B.1) Implantation d'un paramètre de minimisation 1.D

**Question 12.** Il y a 52 bits de mantisse. le plus petit flottant strictement supérieur à 1 est  $1+2^{-52}$ , et  $2^{-52} \simeq 2.5 \times 10^{-16}$  puisque  $2^{10} \simeq 1000$ . Le nombre de chiffres significatifs décimaux est 15.

**Question 13.**  $h = 10^{-16}$  est inadapté car trop petit ( $1 + 10^{-16}$  est arrondi à 1!)  $h = 1$  est à l'inverse trop grand (par exemple pour  $\phi : x \mapsto \sin(\pi x)$ , on a  $\phi'(1) = -\pi$  mais la formule donne 0!). Une valeur correcte est plus petite que 1 (pour raison mathématiques), mais pas trop proche de  $10^{-16}$  (limite de la précision de la représentation). On peut prendre  $h = 10^{-7}$  par exemple<sup>1</sup>.

**Question 14.**

```
def derive(phi, x, h):
    return (phi(x*(1+h))-phi(x*(1-h)))/(2*x*h)
```

**Question 15.** On peut appliquer à la main la formule précédente à  $\varphi'$ , où définir une fonction locale :

```
def derive_seconde(phi, x, h):
    phi_prime=lambda y:derive(phi, y, h)
    return derive(phi_prime, x, h)
```

1. On peut calculer un  $h$  optimal faisant un compromis entre l'approximation mathématique et la précision informatique, mais ce n'était pas demandé ici!

**Question 16.** On rappelle que la méthode de Newton pour trouver un zéro de  $f$  consiste à répéter l'itération  $x \mapsto x - f(x)/f'(x)$ . Ici, un minimum local de  $\varphi$  correspond à un zéro de  $\varphi'$ . Notons qu'on a ni l'assurance de la convergence de la méthode, ni le fait que l'on trouve bien un minimum local en cas de convergence.

```
def min_local(phi, x0, h):
    x=x0
    while abs(derive(phi, x, h))>=10**-7:
        x-=derive_seconde(phi, x, h) / derive(phi,x,h)
    return x
```

### III.B.1) Implantation d'un paramètre de minimisation 2.D

**Question 17.** Un vecteur normal à la surface d'équation  $z - g_x(x, y) = 0$  au point  $(x_0, y_0, g_x(x_0, y_0))$  est donné par le gradient  $(-\frac{\partial g_x}{\partial x}(x_0, y_0), -\frac{\partial g_x}{\partial y}(x_0, y_0), 1)$ . Par suite, le point  $(x_1, y_1, 0)$  appartient au plan tangent si et seulement si  $(x_1 - x_0, y_1 - y_0, -g_x(x_0, y_0))$  est orthogonal au vecteur normal, donc si et seulement si

$$(x_1 - x_0) \frac{\partial g_x}{\partial x}(x_0, y_0) + (y_1 - y_0) \frac{\partial g_x}{\partial y}(x_0, y_0) + g_x(x_0, y_0) = 0$$

De même,  $(x_1, y_1, 0)$  appartient au plan tangent à la surface  $z - g_y(x, y) = 0$  au point  $(x_0, y_0, g_y(x_0, y_0))$  si et seulement si

$$(x_1 - x_0) \frac{\partial g_y}{\partial x}(x_0, y_0) + (y_1 - y_0) \frac{\partial g_y}{\partial y}(x_0, y_0) + g_y(x_0, y_0) = 0$$

Ainsi le point  $(x_1, y_1, 0)$  à l'intersection des trois plans vérifie la relation

$$\begin{pmatrix} -g_x(x_0, y_0) \\ -g_y(x_0, y_0) \end{pmatrix} = J(x_0, y_0) \begin{pmatrix} x_1 - x_0 \\ y_1 - y_0 \end{pmatrix} \text{ avec } J(x_0, y_0) = \begin{pmatrix} \frac{\partial g_x}{\partial x}(x_0, y_0) & \frac{\partial g_x}{\partial y}(x_0, y_0) \\ \frac{\partial g_y}{\partial x}(x_0, y_0) & \frac{\partial g_y}{\partial y}(x_0, y_0) \end{pmatrix}$$

**Question 18.** Par exemple :

```
def grad(G,X,h):
    x0, y0=X
    return np.array([derive(lambda x:G(x,y0), x0, h), derive(lambda y:G(x0,y), y0, h)])
```

**Question 19.** Par exemple avec une fonction annexe calculant la matrice jacobienne.

```
def Jac(G,X,h):
    x0,y0=X
    gxx=derive_seconde(lambda x:G(x,y0), x0, h)
    gyy=derive_seconde(lambda y:G(x0,y), y0, h)
    gxy = derive(lambda x:derive(lambda y:G(x,y), y0, h), x0, h)
    return np.array([[gxx, gxy], [gxy, gyy]])

def min_local_2D(G,X0,h):
    X=X0
    gr=grad(G,X,h)
    gx, gy=gr
    while abs(gx)>=10**-7 or abs(gy)>=10**-7:
        J=Jac(G,X,h)
        X=X-np.dot(np.linalg.inv(J), gr)
        gr = grad(G,X,h)
        gx, gy = gr
    return X
```

## IV. Modèle de la chaîne librement jointe

### IV.A. Modélisation plane

**Question 20.** Avec une liste :

```
def angle():
    """ angle aleatoire de [-pi,pi[ """
    return -math.pi + 2*math.pi*random.random()

def conformation(n):
    return [angle() for _ in range(n)]
```

(Remarque : vu le dessin, il faudrait plutôt prendre l'angle entre  $-\pi/2$  et  $\pi/2$ , mais on respecte l'énoncé).

**Question 21.** Il suffit de sommer les cosinus, et multiplier par le facteur  $\ell$ .

```
def allongement(t,l):
    return sum([math.cos(a) for a in t])*l
```

**Question 22.** On s'arrange pour qu'il y ait bien au moins  $k$  valeurs à modifier. Il faut faire une copie car on veut une nouvelle conformation.

```
def nouvelle_conformation(t, k):
    t=t[:] #copie de la liste.
    n=len(t)
    i=random.randrange(0,n-k+1)
    for j in range(i,i+k):
        t[j]=angle()
    return t
```

## IV.B. Critère de Métropolis Monte Carlo (MMC)

**Question 23.**

```
def selection_conformation(tA, tB, F, l, T):
    EA=-allongement(tA,l)*F
    EB=-allongement(tB,l)*F
    if EB<EA or random.random() <= math.exp((EA-EB)/(k_B*T)):
        return tB
    else:
        return tA
```

## IV.C. Implantation de la simulation

**Question 24.** On procède en deux étapes :

- calcul des 500 premières conformations et de leurs allongements, stockés dans une file ;
- évolution de la file jusqu'à ce que la variance soit faible.

```
def monte_carlo(F,n,l, T, k, eps):
    C=conformation(n)
    FA = [allongement(C,l)] #File des allongements
    for i in range(499): #on calcule au moins 500 conformations
        C=selection_conformation(C,nouvelle_conformation(C,k), F, l, T)
        FA.append(allongement(C,l))
    while variance(FA)>=eps:
        C=selection_conformation(C,nouvelle_conformation(C,k), F, l, T)
        FA.append(allongement(C,l))
        FA.pop(0)
    return moyenne(FA)
```

Remarque : l'opération `pop(0)` sur une liste de taille  $n$  est en  $O(n)$ . Même si ici la file ne présente que 500 éléments, il est assez maladroit d'implémenter une file ainsi !