CORRIGE (CCS 2019)
ELASTICITE D'UN BRIN D'ADN

```
# Q1
def moyenne(X) -> float :
    somme = 0
    for i in X :
        somme += i
    return(somme/len(X))

# Q2
def variance(X) -> float :
    moy = moyenne(X)
    temp = 0
    for i in X :
        temp += i**2
    var = 1/len(X)*temp - moy**2
    return(var)

# Q3
import numbers
def somme(M) :
    if isinstance(M,numbers.Real) : # condition d'arrêt
        return(M)
    som = 0
    for i in M : # boucle sur toutes les composantes
        som += somme(i) # recursion
    return(som)


# Q4
def seuillage(A:np.ndarray, seuil:int) -> np.ndarray :
    a,b = A.shape
    B = np.zeros((a,b))
    for i in range(a) :
        for j in range(b) :
            if A[i,j] < seuil :
                B[i,j] = 1
    return(B)

# Q5
def pixel_centre_bille(A:np.ndarray) -> (int, int) :
    a,b = A.shape
    somx, somy, nb = 0, 0, 0
    for i in range(a) :
        for j in range(b) :
            if A[i,j] == 1 :
                somx += i
                somy += j
                nb += 1
    return(round(somx/nb), round(somy/nb))

# on suppose connu def prendre_photo() -> np.ndarray :
# Q6
def positions(n:int, seuil:int) -> [(int,int)] :
    L = []
    for i in range(n) :
        A = prendre_photo()
        B = seuillage(A, seuil)
        coord = pixel_centre_bille(B)
        L.append(coord)
    return(L)
```

```
# Q7
def fluctuations(P:[(int, int)], t:float) -> float :
    var = variance([i[0] for i in P] + [i[1] for i in P])
    vart = t**2*var
    return(vart)
```

 # Q8

La question est très ouverte et pas forcément très claire… Je propose ce que j'en ai compris. Il faut tout d'abord déterminer le centre ; puis déterminer la longueur maximale que l'on va découper pour déterminer la distance `inter_anneau` (fonction `longueur_maxi`). Ensuite, on balaie l'image et pour chaque pixel, il faut déterminer à quel anneau appartient le pixel en cours (fonction `a_quel_anneau`).
Puis on regarde si le pixel est blanc afin d'incrémenter le nombre de pixels blancs dans l'anneau. Enfin en stockant de manière avantageuse dans des tableaux les données, on retourne la proportion de pixels blancs dans chaque anneau.

```
def profil(A:np.ndarray, n:int) :
    def longueur_maxi(xa:int, ya:int, a:int, b:int) :
        ''' longeur maxi entre le centre et un des 4 coins'''
        d1 = sqrt(xa**2 + xb**2)
        d2 = sqrt((a-xa)**2 + xb**2)
        d3 = sqrt(xa**2 + (b-xb)**2)
        d4 = sqrt((a-xa)**2 + (b-xb)**2)
        return(max(np.array([d1, d2, d3, d4])))

    def a_quel_anneau(i:int, j:int, xa:int, ya:int, inter_anneau:float) :
        ''' renvoie l'indice de l'anneau d'appartenance du pixel (i,j)'''
        d = (i-xa)**2 + (j-ya)**2
        ind = int(sqrt(d)/inter_anneau)
        return(int)

    # determination du centre des anneaux
    xa, ya = pixel_centre_bille(A)
    a, b = A.shape
    # distance entre les anneaux
    inter_anneau = longueur_maxi(xa, ya, a, b) / n
    # initialisation des tableaux
    tabBlancs = np.zeros(n)
    tabTotals = np.zeros(n)

    # on balaie l'ensemble de la figure
    for i in range(a) :
        for i in range(b) :
            # A quel anneau le pixel appartient-il ?
            indice = a_quel_anneau(i, j, xa, ya, inter_anneau)
            tabTotals[indice] += 1
            if A[i,j] == 1 :
                tabBlancs[indice] += 1

    # on renvoie la proportion dans chaque anneau des pixels blancs
    return(tabBlancs/tabTotals)
```

 # Q9

En reprenant le code précédent, on a :
— la complexité de pixel_centre_bille en $O(p^2)$
— celle de création d'un tableau en $O(n)$;
— la double boucle a une complexité en $O(p^2)$
En supposant $n << p$, la complexité globale de la fonction est en $O(p^2)$

```
# Q10
global  K_B
K_B = 1.38064852e-23

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

```
# Q11
import scipy #c'est plus simple !
def ajusteWLC(Fz:np.ndarray, T:float) -> (float, float) :

    def fonctionAjustement(z:np.ndarray, Lp:float, T:float) :
        return(force(z, Lp, L0, T))

    tabAll = Fz[:,1]
    tabForce = Fz[:,0]
    popt, pcov = scipy.optimize.curve_fit(fonctionAjustement, tabAll, tabForce)
    return(popt)
```

 # Q12

La mantisse étant codée sur 52 bits, la manipulation des nombres flottants entraîne systématiquement une erreur relative $de\ 2^{-52} \cong\ 1\ \times\ 10^{-16}$ ce qui induit une erreur systématique sur le 17ième chiffre significatif du nombre. En conclusion, on possède donc 16 chiffres significatifs.

 # Q13

Le choix *h* = 1 conduit à considérer l'intervalle $[0, 2x]$ comme voisinage de x. Pour $h\ =\ 1 \times 10^{-16}$, on se trouve très proche de la précision machine, par conséquent, on risque de ne pas voir ce taux d'accroissement, ce qui conduit à une dérivée nulle. À priori une valeur intermédiaire fait l'affaire, on peut opter pour 1×10^{-8}.

```
#  Q14
def derive(phi, x:float, h:float) -> float :
    return((phi(x*(1+h)) - phi(x*(1-h))) / (2*x*h))

# Q15
def derive_seconde(phi, x:float, h:float) -> float:
    dmoins = derive(phi, x*(1-h), h)
    dplus = derive(phi, x*(1+h), h)
    return((dplus - dmoins) / (2*x*h))


# Q16
def min_local(phi, x0:float, h:float) -> float :
    # Methode de Newton sur la derivee de phi
    x = x0
    d = derive(phi, x, h)
    while abs(d) >= 1e-7 :
        dd = derive_seconde(phi, x, h)
        x -= d/dd
        d = derive(phi, x, h)
    return(x)
```

 # Q17

Il suffit d'écrire que $g_x(x,y) =\ g_y(x,y) = 0$ / On obtient directement la matrice jacobienne $J(x_0, y_0)$ qui vaut :

$$\begin{bmatrix} \frac{\partial g_x}{\partial x}(x, y_0) & \frac{\partial g_x}{\partial y}(x_0, y) \\ \frac{\partial g_y}{\partial x}(x, y_0) & \frac{\partial g_y}{\partial y}(x_0, y) \end{bmatrix}$$

```
# Q18
def grad(G, X:np.ndarray, h:float) -> np.ndarray :
    x0, y0 = X
    def Gx(x:float) -> float :
        return(G((x, y0)))
    def Gy(y:float) -> float :
        return(G((x0, y)))
    return(np.array([derive(Gx, x0, h), derive(Gy, y0, h)]))
```

```
# Q19
def min_local_2D(G, X0:np.ndarray, h:float) -> np.ndarray :
    ''' G represente la fonction a minimiser  '''

# Calcul de la matrice jacobienne
    def gx(X:np.ndarray) -> float :
        x, y = X
        return((G((x*(1+h), y)) - G((x*(1-h), y))) / (2*x*h))
    def gy(X:np.ndarray) -> float :
        x, y = X
        return((G((x, y*(1+h))) - G((x, y*(1-h)))) / (2*y*h))
    def jacobienne(X:np.ndarray) -> np.ndarray :
        return np.array([grad(gx, X, h), grad(gy, X, h)])

    # Recurrence
    X = X0
    Y = grad(G, X, h)
    while abs(Y[0]) > 1e-7 or abs(Y[1]) > 1e-7 :
        J = jacobienne(Y)
        Jinv = np.linalg.inv(J)
        X -= np.dot(Jinv, Y)
        Y = grad(G, X, h)
    return(X)


# Q20
def conformation(n:int) -> [float] :
    listeAngles = []
    for i in range(n) :
        listeAngles.append((2*random.random() - 1)*math.pi)
    return(listeAngles)

# Q21
def allongement(theta:[float], l:float) -> float :
    longueur = 0
    for a in theta :
        longueur += l*math.cos(a)
    return(longueur)


# Q22
def nouvelle_conformation(theta:[float], k:int) -> [float]:
    listeNouvelle = theta[:] # copie de liste de niveau 1
    indice = random.randrange(0, len(theta)-k+1) # indice a partir duquel ...
    for i in range(k) :
        listeNouvelle[indice + i] = math.pi * (2*random.random() - 1) # modification
    return(listeNouvelle)


# Q23
def selection_conformation(thetaA:[float], thetaB:[float], F:float, l:float, T:float)
-> [float]:
    E1 = - allongement(thetaA, l) * F
    E2 = - allongement(thetaB, l) * F
    if E2 < E1 :
        return(thetaB)
    else :
        p = math.exp((E1 - E2)/(K_B * T))
        if random.random() < p :
            return(thetaB)
        else:
            return(thetaA)
```

```
*# Q24
def monte_carlo(F:float, n:int, l:float, T:float, k:int, epsilon:float) -> float :

    # Remplir la file avec les 500 premieres conformations
    confEnCours = conformation(n) # la conformation courante
    listeAll = [allongement(confEnCours, l)] # file des 500 derniers allongements

    for i in range(499):
        nouvConf = nouvelle_conformation(confEnCours, k)
        confEnCours = selection_conformation(confEnCours, nouvConf, F, l, T)
        listeAll.append(allongement(confEnCours, l))

    # Continuer les simulations jusqu'a la convergence
    while variance(listeAll) > epsilon :
        nouvConf = nouvelle_conformation(confEnCours, k)
        confEnCours = selection_conformation(confEnCours, nc, F, l, T)
        listeAll.pop(0)
        listeAll.append(allongement(confEnCours, l))

    # Moyenne renvoyee
    return(moyenne(listeAll))
```