Proposition de corrigé du sujet d'info commune CSE 2019

En plus des importations renseignées dans le sujet, on ajoute

```
import numbers
```

pour pouvoir utiliser le test du type numérique d'une variable comme suggéré dans l'énoncé.

1. Je n'utilise personnellement nulle part cette fonction.

```
def moyenne(X) :
    """
    calcul de la moyenne des valeurs contenues dans X
    """
    somme = 0
    for x in X :
        somme += x
    return somme / len(X)
```

2. Cette fonction a été améliorée par rapport à ce qui était simplement demandé, pour pouvoir réaliser des calculs de variance en parallèle, sur des composantes de ndarray, en question 7.

```
def variance(X) :
    """
    calcul de la variances des valeurs ou des "vecteurs" contenues dans X
    """
    if isinstance(X[0], numbers.Real) :
        # init si liste de nombres
        somme_x, somme_carre_x = 0, 0
    else :
        # on suppose que si ce n'est pas une liste de nombre, alors
        # c'est une liste de ndarray de mêmes dimensions
        somme_x = np.zeros(len(X[0]), dtype=float)
        somme_carre_x = np.zeros(len(X[0]), dtype=float)
    for x in X :
        somme_x += x
        somme_carre_x += x * x
    return somme_carre_x/len(X) - (somme_x/len(X))**2
```

3. Fonction utilisant la récursivité.

```
def somme(M) :
    """
    somme des termes de cette structure, réalisée récursivement
    """
    res = 0
    for x in M :
        if isinstance(x, numbers.Real) :
            res += x
        else :
            res += somme(x)
    return res
```

4.
```
def seuillage(A, seuil) :
    """
    renvoie un tableau du même format que A, seuillé par "seuil", contenant 0 ou 1.
    """
    n, m = A.shape
    Aseuil = np.zeros((n, m), dtype=int)
    for i in range(n) :
        for j in range(m) :
            if A[i, j] < seuil :
                Aseuil[i, j] = 1
    return Aseuil
```

5. Dans cette fonction, on exploite les opérations + et // sur des ndarray, ce qui rend le code plus concis, et lisible

```
def pixel_centre_bille(A) :
    cmpt = 0  # nbre de points pris en compte
    G = np.zeros(2, dtype=int)  # pour le calcul du barycentre
    for i in range(A.shape[0]) :
        for j in range(A.shape[1]) :
            if A[i, j] == 1 :
                # on ne compte que les points blancs
                G += np.array([i, j])
                cmpt += 1
    return G // cmpt
```

6. Cette opération ne nécessite pas vraiment de fonction si on utilise les définitions de listes par compréhension.

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

7. Si l'on note C_i, le ième centre de bille, et G, le barycentre des positions des billes, alors les fluctuations demandées sont

$$< C_iG^2 >=< (x_i - x_G)^2 + (y_i - y_G)^2 >= var(x_i) + var(y_i)$$

Ce double calcul de variance peut être mené en parallèle grâce aux tableau numpy Il suffit donc d'appliquer la fonction `variance` à la liste de centres, que j'ai modifié en conséquence, par rapport à ce qui était demandé, afin de pouvoir l'appliquer à des listes de points.

```
def fluctuation(P, t) :
    """
    un pixel est de longueur T. La variance est homogène au carré
    d'un nbre de pixels, donc on multiplie la variance par t*t
    """
    return somme(variance(P) * t * t)
```

8. Ici, on comprend qu'il faut d'abord définir la taille du cercle le plus à l'extérieur. C'est celui dont le rayon vaut la distance entre le centre de la bille et le coin le plus éloigné de ce dernier. Ce calcul est effectué au début. Il faudra éviter de prendre le point en ce coin en compte

```
def profil(A, n) :
    n_lig, n_col = A.shape
    C = pixel_centre_bille(A)  # centre de la bille
    # -----------------------------------------------------
    # calcul de la dist^2 max entre le centre un coin
    l_coins = ([0, 0], [0, n_col], [n_lig, 0], [n_lig, n_col])
    d2_max_coin = max(somme((C - np.array(coin))**2) for coin in l_coins) + 0.1
    # le +0.1 permet de ne pas avoir un cercle d'indice n pour
    # le pixel du coin le plus éloigné dans la suite
    # -----------------------------------------------------
^^I^^I# liste du nbre de pt blancs/noirs pour les différentes couronnes
    pt_blancs, pt_noirs = np.zeros(n, dtype=float), np.zeros(n, dtype=float)
    for i in range(n_lig) :
        for j in range(n_col) :
            num_cercle = int(n * somme((C-np.array([i, j]))**2) /
                             d2_max_coin)
            if A[i, j] == 1 :
                pt_blancs[num_cercle] += 1
            else :
                pt_noirs[num_cercle] += 1
    return pt_blancs / (pt_blancs + pt_noirs)
```

9. La fonction parcours des objets de deux tailles distinctes : un tableau de taille $p \times p$ et des objets de taille n. Ainsi, la complexité est en $\mathcal{O}(p^2 + n)$, car `pt_blancs / (pt_blancs + pt_noirs)`, ainsi que `np.zeros(n, dtype=float)` est en $\mathcal{O}(n)$.

10.
```
def force(z, Lp, L0, T) :
    return K_B * T / Lp / 4 * (1/(1-z/L0)**2 - 1 + 4*z/L0)
```

11. Il fallait faire attention ici au fait que les valeurs de `F` sont sur la première colonne, alors que les `z` sont sur la deuxième. D'autre part, la température n'est pas un paramètre ajustable, mais un paramètre fixe, ce qui nécessite, par exemple, de redéfinir la fonction à passer en argument de `curve_fit`, pour réduire les dépendances.

```
def ajusteWLC(Fz, T) :
    import scipy.optimize
    popt, pcov = scipy.optimize.curve_fit(lambda z, Lp, L0 : force(z, Lp, L0, T), Fz[ :, 1], Fz[ :, 0])
    return popt
```

12. les 52 bits de mantisses permettent de définir les 52 puissances négatives de 2 de cette mantisse. La précision relative du nombre est de $2^{-52} > 1 \cdot 10^{-16}$, donc le nombre de chiffres significatifs décimaux est de 16.

13. h est ici le pas relatif de l'accroissement. Alors $h = 1$ ou $h = 1 \cdot 10^{-16}$ ne conviennent pas, car dans le premier l'expression approchée de la dérivée n'est plus valable, et dans le deuxième, on passe sous la précision relative des flottants décrits. On peut proposer $h = 1 \cdot 10^{-15}$ par exemple au plus petit.

14.
```
def derive(phi, x, h) :
    return (phi(x*(1+h)) - phi(x*(1-h))) / (2*h*x)
```

15. Cette fonction peut être définie à partir de la précédente ou bien directement. J'ai choisi la première option.

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

16. Utilisation de la méthode de Newton pour trouver le minimum. le critère de convergence se fait sur la valeur de la dérivée.

```
def min_local(phi, x0, h) :
    dphi = derive(phi, x0, h)
    x = x0
    while abs(dphi) > 1e-7 :
        x -= phi(x)/dphi
        dphi = derive(phi, x, h)
    return x - phi(x)/dphi
```

17. Les deux premiers plans ont pour équations :

    - $z = g_x(x_0,y_0) + \frac{\partial g_x}{\partial x}(x_0,y_0)(x - x_0) + \frac{\partial g_x}{\partial y}(x_0,y_0)(y - y_0)$
    - $z = g_y(x_0,y_0) + \frac{\partial g_y}{\partial x}(x_0,y_0)(x - x_0) + \frac{\partial g_y}{\partial y}(x_0,y_0)(y - y_0)$

    Donc le point de coordonnées $(x_1,y_1,0)$ vérifie :

$$\begin{cases} -g_x(x_0,y_0) &= \frac{\partial g_x}{\partial x}(x_0,y_0)(x_1 - x_0) + \frac{\partial g_x}{\partial y}(x_0,y_0)(y_1 - y_0) \\ -g_y(x_0,y_0) &= \frac{\partial g_y}{\partial x}(x_0,y_0)(x_1 - x_0) + \frac{\partial g_y}{\partial y}(x_0,y_0)(y_1 - y_0) \end{cases}$$

    Ce qui peut se mettre sous forme matricielle :

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

    D'où

$$\mathrm{J}(x_0,y_0) = \begin{pmatrix} \frac{\partial g_x}{\partial x}(x_0,y_0) & \frac{\partial g_x}{\partial y}(x_0,y_0) \\ \frac{\partial g_y}{\partial x}(x_0,y_0) & \frac{\partial g_y}{\partial y}(x_0,y_0) \end{pmatrix}$$

18. Dans toute la suite il faudra prendre en compte le fait que les ndarray construits à partir d'une liste sont de type ligne et non colonne.

```
def grad(G, X, h) :
    # construction du gradient de G au point X
    return np.array([derive(lambda x : G(x, X[1]), X[0], h),
                     derive(lambda y : G(X[0], y), X[1], h)])
```

19. On va définir une fonction renvoyant J^{-1}, tel que définie précédemment.

```
def inv_J(G, X, h) :
    # définition de la matrice inverse de J
    # grad(gx(X))
    gradgx = grad(lambda x0, y0 : derive(lambda x : G(x, y0), x0, h), X, h)
    # grad(gy(X))
    gradgy = grad(lambda x0, y0 : derive(lambda y : G(x0, y), y0, h), X, h)
    return np.linalg.inv(np.array([gradgx, gradgy]))
```

```
def min_local_2D(G, X0, h) :
    """
    comme les "vecteurs" sont des vecteurs ligne et non colonne,
    il conviendra de faire le produit avec la transposé de J^-1,
    X_{n+1} = X_n - grad(G) . (J^-1)^t
    X = X0
    """
    X = X0
    gradG = grad(G, X, h)
    while abs(gradG[0]) > 1e-7 or abs(gradG[1]) > 1e-7 :
        # X_{n+1} = X_n - grad(G) . (J^-1)^t
        X -= np.dot(gradG, np.transpose(inv_J(G, X)))
        gradG = grad(G, X, h)
    return X - np.dot(gradG, np.transpose(inv_J(G, X)))
```

20.
```
def conformation(n) :
    return [(1 - 2*random.random()) * math.pi for i in range(n)]
```

21. Ici, utilisation du cos vectorisé de numpy.

```
def allongement(theta, l) :
    return l * somme(np.cos(theta))
```

22. Il faut faire attention au fait que si l'indice aléatoire est >k, alors il faut continuer les modifications d'angles à partir du début (d'où le modulo).

```
def nouvelle_conformation(theta, k) :
    n = len(theta)
    i0 = random.randrange(0, n)
    for i in range(k) :
        theta[(i0+i) % n] = (1 - 2*random.random()) * math.pi
```

23. Dans l'énoncé, il est dit que la fonction renvoie une conformation, bien que cela ne soit pas indiqué dans l'en-tête de la fonction…

```
def selection_conformation(thetaA, thetaB, F, l, T) :
    EA = - allongement(thetaA, l) * F
    EB = - allongement(thetaB, l) * F
    if EB < EA or random.random() < math.exp((EA - EB)/K_B/T) :
        # cas de conservation de la nouvelle config
        return thetaB
    else :
        return thetaA
```

Personnellement, j'aurais choisi de ne passer que la conformation actuelle theta en argument, et de réaliser la nouvelle conformation ainsi que la sélection dans cette fonction. La conformation sélectionnée aurait ainsi été donnée à theta, et inutile de renvoyer quoi que ce soit. La suite aurait été grandement simplifié. Mais ce n'est pas ce qui est demandé…

```
def selection_conformation_alt(theta, k, F, l, T) :
    theta_copy = theta.copy()
    nouvelle_conformation(theta, k) :
    EA = - allongement(theta_copy, l) * F
    EB = - allongement(theta, l) * F
    if EB >= EA and random.random() > math.exp((EA - EB)/K_B/T) :
        # cas de non conservation de la nouvelle config
        theta = theta_copy
```

24. J'ai fais le choix de remplir la file puis de l'entretenir à 500 dans une seule boucle pour éviter de la redondance de code. Cela génère quelques tests en plus, mais rien qui n'augmente la complexité asymptotique.

```
def monte_carlo(F, n, l, T, k, epsilon) :
    N = 500
    thetaB = conformation(n)
    z = allongement(thetaB, l)
    somme_z, somme_z_carre = z, z*z  # pour le calcul de la variance
    ma_file = [z]  # init de la file des conformations
```

```
    var, z0 = 0, 0
    # on crée la file des 500 premières conformations, puis les suivantes
    while len(ma_file) < N or var > epsilon :
        thetaA = thetaB
        thetaB = selection_conformation(thetaA, nouvelle_conformation(thetaA, k), F, l, T)
        z = allongement(thetaB, l)
        # calcul de la variance, en ajoutant la dernière conformation
        # et supprimant le premier (si 500 conform.) : on évite de tout recalculer !
        if len(ma_file) > N :
            z0 = ma_file.pop(0)
        somme_z += z - z0
        somme_z_carre += z * z - z0*z0
        var = somme_z_carre/N - somme_z * somme_z / N / N
    return somme_z / N
```

Une fonction `monte_carlo_alt` utilise la version de fonction `selection_conformation_alt`.

```
def monte_carlo_alt(F, n, l, T, k, epsilon) :
    N = 500
    theta = conformation(n)
    z = allongement(theta, l)
    somme_z, somme_z_carre = z, z*z  # pour le calcul de la variance
    ma_file = [z]  # init de la file des conformations
    var, z0 = 0, 0
    # on crée la file des 500 premières conformations, puis les suivantes
    while len(ma_file) < N or var > epsilon :
        selection_conformation(theta, k, F, l, T)
        z = allongement(theta, l)
        # calcul de la variance, en ajoutant la dernière conformation
        # et supprimant le premier (si 500 conform.) : on évite de tout recalculer !
        if len(ma_file) > N :
            z0 = ma_file.pop(0)
        somme_z += z - z0
        somme_z_carre += z * z - z0*z0
        var = somme_z_carre/N - somme_z * somme_z / N / N
    return somme_z / N
```