teaching sciences for innovation

Proposition de corrigé

Concours : Concours Centrale-Supélec

Année : 2019

Filière : MP - PC - PSI - TSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Correction IPT – CCS 2019
Élasticité d'un brin d'ADN

1 Fonctions utilitaires

Q 1. Calcul d'une moyenne classique.

```
1 def moyenne(X) -> float :
2     somme = 0
3     for i in X :
4         somme += i
5     return(somme/len(X))
```

Commentaire : Puisque `X` est un itérable, on peut applique la méthode `sum` directement.

Q 2. Cette fois-ci, c'est le calcul de la variance qu'il faut coder – en gardant une complexité en $O(n)$:

```
1 def variance(X) -> float :
2     moy = moyenne(X)
3     temp = 0
4     for i in X :
5         temp += i**2
6     var = 1/len(X)*temp - moy**2
7     return(var)
```

Q 3. C'est une question sur le parcours d'un arbre (ici une séquence imbriquée). Il est pratique de traiter ce type de questions par une fonction récursive. De plus, l'aide sur le typage des grandeurs dans la séquence permet d'exprimer facilement le cas de base de cette fonction. Ce qui donne :

```
1 import numbers
2 def somme(M) :
3     if isinstance(M,numbers.Real) : # cas de base
4         return(M)
5     som = 0
6     for i in M : # boucle sur toutes les composantes
7         som += somme(i) # recursion
8     return(som)
```

Commentaire : Il manque dans le sujet l'appel à la bibliothèque `numbers` pour pouvoir utiliser la méthode `Real`.

2 Mesures expérimentales

2.1 Position de la bille

Q 4. Rien de bien particulier, le parcours d'une image par une double boucle `for`, puis un test sur la valeur du pixel par rapport au seuil.

```
1 def seuillage(A:np.ndarray, seuil:int) -> np.ndarray :
2     a,b = A.shape
3     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)
```

Q 5. Il suffit de balayer de nouveau l'image seuillée afin de déterminer le barycentre des positions. Il faut par contre renvoyer un couple (tuple) de deux entiers du pixel le plus proche. C'est la fonction `round` qui permet ceci.

```
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))
```

Commentaire : Je n'ai pas utilisé la fonction `moyenne`, mais bien sûr il est possible de stocker dans un itérable pour réaliser celle-ci, mais la complexité mémoire serait plus grande.

Q 6. Il suffit d'exécuter les différentes tâches dans l'ordre.

```
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)
```

Q 7. En relisant correctement la question, on s'aperçoit qu'il est simplement demandé de calculer la somme des variances. À ne pas oublier de multiplier par t^2, t étant la taille d'un pixel.

```
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)
```

2.2 Allongement du brin d'ADN

Q 8. 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)
```

Commentaire : Dans cette section 2, je me suis affranchi d'une possible erreur de sujet sur les axes x et y proposés. Et sur cette question ma proposition est à prendre avec des pincettes.

Q 9. 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 $O(p^2)$.

3 Modèle du ver

3.1 Calcul des paramètres

Q 10. En `numpy`, le résultat est immédiat puisque la fonction est vectorielle.

```
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))
```

Q 11. Il s'agit de bien comprendre la documentation proposée. Ensuite, il faut penser à extraire du tableau général, les vecteurs `force` et `allongement` et enfin de renvoyer uniquement les coefficients.

```
import scipy

```

```python
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)
```

3.2 Algorithme du minimum local

3.2.1 Implantation d'un algorithme de minimisation 1D

Q 12. La mantisse étant codée sur 52 bits, la manipulation des nombres flottants entraîne systématiquement une erreur relative de $2^{-52} \simeq 1 \times 10^{-16}$ ce qui induit une erreur systématique sur le 17ième chiffre significatif du nombre.En conclusion, on possède donc 16 chiffres significatifs.

Q 13. 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}.

Q 14. De manière très simple :

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

Q 15. En utilisant la fonction précédente,on exprime la dérivée seconde facilement :

```python
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))
```

Q 16. Il s'agit d'utiliser la méthode de Newton sur la dérivée première de Φ afin de déterminer un minium local. La condition de convergence ne se fait pas sur la suite des x_n et plus précisément sur la différence $x_{n+1} - x_n$, mais sur une condition de type $\Phi'(x_n)$ suffisamment proche de zéro. Ensuite rien de particulier, si l'on suppose que l'on obtient bien un minimum.

```python
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)
```

Commentaire : Rien n'est précisé sur $x0$. On suppose que c'est un flottant pris suffisamment proche d'un minimum local de la fonction étudiée.

3.2.2 Implantation d'un algorithme de minimisation 2D

Q 17. 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 :

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

Q 18. Il faut penser à bien dériver la bonne fonction par rapport à la bonne variable en utilisant la fonction `derive`, et renvoyer un `np.ndarray`. J'ai par conséquent utilisé des sous-fonctions définissant correctement `Gx` et `Gy`.

```
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)]))
```

Q 19. Dans un premier temps, il est intéressant de se créer une fonction qui évaluera la matrice jacobienne. Pour cela, on utilise ici 3 sous-fonctions. Puis on met en place la récurrence de l'algorithme de Newton en prenant garde aux conditions de fin, mais aussi aux calculs sur J et $Y = \text{grad}(G, X, h)$.

```
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)
```

Commentaire : Je n'arrive pas à comprendre ce que représente G. En effet au début de la sous-section, il nous présente la fonction E à minimiser par rapport à L_p et L_0. Puis dans le blabla qui suit, on parle d'*adapter la méthode de Newton unidimensionnelle pour rechercher un zéro ... appliquer cette méthode au* ***gradient*** *de* E. Ensuite on a G mais on ne sait pas ce qu'elle représente ... puisqu'on change encore de nom en parlant de `fct_dont_je_veux_le_minimum()`, qui n'est clairement pas le gradient ... Bref, j'ai proposé une version sur la logique du minimum local 1D.

4 Modèle de la chaîne librement jointe

4.1 Modélisation plane

Q 20. Il s'agit de définir une liste (ou tableau) d'angles compris entre $[-\pi, \pi[$. Par conséquent, en utilisant la méthode `random` qui renvoie un nombre compris entre 0 (inclus) et 1 (exclu), il suffit de multiplier par 2 et de soustraire 1 pour avoir un intervalle entre $[-1, 1[$. Ce qui donne :

```
def conformation(n:int) -> list :
    listeAngles = []
    for i in range(n) :
```

```
        listeAngles.append((2*random.random() - 1)*math.pi)
    return(listeAngles)
```

Q 21. Pas de difficulté, il suffit de sommer les projections suivant l'axe horizontal des différents segments :

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

Q 22. Tout d'abord il faut initialiser une nouvelle liste (réaliser ici par une copie), puis on cherche l'indice à partir duquel on peut obtenir k valeurs successives. Il ne faut pas oublier la borne supérieure qui vaut `len(theta)-k` (le plus 1 est présent par le fait que la méthode `randrange` a une borne supérieure exclue). Puis on modifie k éléments :

```
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)
```

Commentaire : L'en-tête de la fonction ne précise pas qu'il faut retourner une nouvelle liste, mais cette précision est inscrite dans la question. Je ne sais donc pas s'il fallait modifier la liste initiale ou en générer une nouvelle.

4.2 Critère de Metropolis Monte Carlo (MMC)

Q 23. Il faut dans un premier temps calculer les énergies ; les comparer puis si nécessaire réaliser un tirage et le comparer à une probabilité donnée ; pour enfin renvoyer la bonne conformation.

```
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)
```

4.3 Implantation de la simulation

Q 24. Il s'agit ici de mettre en musique l'ensemble des fonctions précédentes. Tout d'abord, on créé une liste contenant 500 allongements – pour cela, on génère une première conformation et son allongement associé – puis à partir de celle-ci, on génère 499 autres conformations et allongements associés. Ensuite, la condition de convergence de la boucle `while` porte sur la variance de la liste des allongements et on élimine de la file le premier élément pour le remplacer par une nouvelle conformation. Enfin on renvoie la moyenne des 500 allongements vérifiant le critère de la variance.

```
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))
```