

MINES INFORMATIQUE COMMUNE 2018, CORRIGÉ

Merci à Serge Bays pour sa relecture attentive.

Partie I. Stockage interne des données

Q1 Hormis la première ligne qu'on demande d'ignorer, chaque ligne est constituée de 8 caractères (5 chiffres, un signe, un point et le caractère de fin de ligne comme le signale l'énoncé) donc occupe simplement 8 octets. 20 minutes correspondent à 1200 s. Comme il y a deux mesures par seconde, cela correspond à 2400 mesures au total, soit une taille de $8 \times 2400 = 19\,200$ octets = 19,2 ko.

Q2 La campagne de mesure fait état d'un fichier récolté toutes les demi-heures pendant 15 jours. Il y a donc $15 \times 24 \times 2 = 7,2 \cdot 10^2$ fois les informations précédentes collectées, ce qui représente donc une taille totale de 13 824 000 octets, soit 13,8 Mo.

Une carte-mémoire de 1 Go est largement suffisante.

Q3 Ôter un chiffre revient à passer chaque ligne de 8 à seulement 7 octets, d'où une réduction de $1/8 = 12\%$ de la taille du fichier total.

Q4 La lecture d'un fichier texte peut se faire de la manière suivante

```
1 f = open('donnees.txt') # Ouverture du descripteur de fichier
2 L = f.readlines()       # Lecture effective du fichier
3 liste_niveaux = []      # Définition du conteneur
4 for i in range(1,len(L)): # On saute la première ligne de texte
5     liste_niveaux.append(float(L[i])) # et on convertit le reste en flottants
6 f.close()               # Fermeture du descripteur de fichier
```

Bien sûr, on peut aussi demander à Numpy de faire tout le travail à notre place

```
1 import numpy as np # Importation de la bibliothèque
2 # On saute la première ligne et on convertit au vol en liste car np.loadtxt
3 # renvoie normalement un np.array
4 liste_niveaux = list(np.loadtxt('donnees.txt',skiprows=1))
```

Partie II. Analyse « vague par vague »

Q5 À noter que les calculs des H_i font à chaque fois intervenir le maximum précédant Z_i et le minimum qui le suit (l'existence du maximum précédant étant assuré par l'hypothèse de fonction initialement croissante et au-dessus de la moyenne). Pour la figure 2 de l'énoncé, on a alors $H_1 \approx 6 - (-3) = 9$ m, puis $H_2 \approx 7 - (-2) = 9$ m et $H_3 \approx 5 - (-1) = 6$ m. Pour les mesures de périodes, on obtient

$$T_1 \approx 15,5 - 3,5 = 12 \text{ s} \quad \text{et} \quad T_2 \approx 28,5 - 15,5 = 13 \text{ s}$$

Q6 Pour calculer la moyenne, on peut procéder comme suit

```

1 def moyenne(liste_niveaux):
2     return sum(liste_niveaux)/len(liste_niveaux)

```

Ou alors, si on a un doute pour savoir si la fonction `sum` est autorisée, on peut expliciter son effet par une boucle

```

1 def moyenne(liste_niveaux):
2     m = 0
3     for valeur in liste_niveaux:
4         m = m + valeur
5     return m/len(liste_niveaux)

```

Q7 Commençons par implémenter l'intégration trapèze en se souvenant que le pas d'intégration doit être d'une demi-seconde pour correspondre à la période d'échantillonnage de 2 Hz imposée par le système.

```

1 def integrale_precise(liste_niveaux):
2     dt = 0.5      # Pas d'intégration
3     integrale = 0 # Initialisation de l'intégrale
4     for i in range(len(liste_niveaux)-1):
5         trapeze = (liste_niveaux[i] + liste_niveaux[i+1])/2 * dt
6         integrale = integrale + trapeze
7     return integrale

```

Pour faire le lien avec la valeur moyenne, il faut se souvenir qu'une valeur moyenne d'une fonction continue η sur un temps total t_{tot} est définie par

$$\langle \eta(t) \rangle = \frac{1}{t_{\text{tot}}} \int_0^{t_{\text{tot}}} \eta(t) dt$$

```

1 def moyenne_precise(liste_niveaux):
2     dt = 0.5
3     t_tot = len(liste_niveaux) * dt
4     integrale = integrale_precise(liste_niveaux)
5     return integrale / t_tot

```

Q8 Il s'agit de calculer la moyenne et passer en revue tous les points pour savoir s'il y en a un qui dépasse la moyenne en sens descendant. On sort alors directement de la fonction par le `return`.

```

1 def ind_premier_pzd(liste_niveaux):
2     moyenne = moyenne_precise(liste_niveaux)
3     for i in range(len(liste_niveaux)-1):
4         if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
5             return i
6     return -1 # Cas où on n'a pas réussi à vérifier la condition dans la boucle

```

Q9 Même chose avec le dernier, il suffirait de garder une mémoire et la renvoyer à la fin.

```

1 def ind_dernier_pzd(liste_niveaux):
2     moyenne = moyenne_precise(liste_niveaux)
3     position = -2 # Valeur par défaut si jamais on ne trouve rien
4     for i in range(len(liste_niveaux)-1):
5         if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
6             position = i
7     return position

```

Néanmoins, cette implémentation a deux soucis : on doit nécessairement parcourir toute la liste (ce qui donne une complexité en $\mathcal{O}(N)$ et non $\mathcal{O}(1)$) et le calcul de la moyenne est lui-même déjà en $\mathcal{O}(N)$. Pour avoir une méthode en $\mathcal{O}(1)$ dans le meilleur des cas, on peut utiliser la même technique qu'à la question précédente mais en partant (de manière symétrique) de la fin. Le meilleur des cas est alors que le dernier indice cherché soit l'avant-dernier. Néanmoins, comme dit plus haut, il faut aussi s'assurer que le calcul de la moyenne a déjà été fait auparavant et passé en paramètre à la fonction sinon il imposera sa complexité linéaire en N .

```

1 # Ne pas oublier de donner la moyenne en argument
2 def ind_dernier_pzd(liste_niveaux, moyenne):
3     # On parcourt la boucle avec un pas négatif de -1 pour remonter la liste
4     for i in range(len(liste_niveaux)-2,-1,-1):
5         if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
6             return i # Sortie rapide de la fonction dans le meilleur des cas
7     return -2 # Si où on n'a pas réussi à vérifier la condition dans la boucle

```

Q10 Il s'agit à présent de récupérer tous les indices et les stocker dans une liste. La fonction complétée s'écrit

```

1 def construction_succeesseurs(liste_niveaux):
2     n = len(liste_niveaux)
3     succeesseurs = []
4     m = moyenne(liste_niveaux)
5     for i in range(n-1):
6         if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < m:
7             succeesseurs.append(i+1)
8     return succeesseurs

```

Il faut bien prendre le point en position $i+1$ qui succède juste à la traversée de la moyenne.

Q11 Pour construire la liste des vagues, on va itérer sur les succeesseurs et découper notre liste initiale en conséquence.

```

1 def decompose_vagues(liste_niveaux):
2     succeesseurs = construction_succeesseurs(liste_niveaux)
3     vagues = []
4     for i in range(len(succeesseurs)-1):
5         Zi = succeesseurs[i] # Position de Zi
6         Zip1 = succeesseurs[i+1] # Position de Zi+1
7         vagues.append(liste_niveaux[Zi:Zip1]) # La vague associée
8     return vagues

```

Q12 Maintenant que l'on dispose d'une liste de vagues, il suffit d'appliquer `max` et `min` sur cette liste pour obtenir les crêtes et compter le nombre d'éléments pour estimer la période. Attention tout de même au fait que l'on prend le `max` avant Z_i et le `min` après Z_i et que dans la fonction `decompose_vagues`, on n'ait pas pensé à garder la première demi-vague pourtant nécessaire pour estimer H_1 et T_1 ...

```

1 def proprietes(liste_niveaux):
2     vagues = decompose_vagues(liste_niveaux)
3     dt = 0.5 # Intervalle de temps entre deux points
4     P = [] # La future liste des propriétés que l'on veut obtenir
5     # On traite à part le premier cas
6     i0 = ind_premier_pzd(liste_niveaux)
7     H1 = max(liste_niveaux[:i0]) - min(vagues[0])
8     T1 = len(vagues[0])
9     P.append([H1,T1])
10    # Après, on peut prendre les vagues les unes après les autres.
11    for i in range(len(vagues)-1):
12        Hi = max(vagues[i]) - min(vagues[i+1])
13        Ti = len(vagues[i+1]) * dt
14        P.append([Hi,Ti])
15    return P

```

Partie III. Contrôle des données

Q13 Pour obtenir H_{\max} , il suffit de déterminer le maximum des H calculés par la fonction `proprietes` précédente.

```

1 def H_max(liste_niveaux):
2     prop = proprietes(liste_niveaux) # Détermination des propriétés des vagues
3     Hmax = 0 # Initialisation de la mémoire.
4     for Pi in prop: # On itère sur toutes les propriétés.
5         Hi,Ti = Pi # On récupère le couple de propriété pour la vague i.
6         if Hi > Hmax: # Si on trouve mieux que la mémoire,
7             Hmax = Hi # on le mémorise.
8     return Hmax # Renvoi du maximum observé

```

Q14 Au premier appel de la fonction de tri, on doit regarder la liste en totalité. Par conséquent, l'indice `g` (pour « gauche ») doit valoir initialement 0 alors que l'indice `d` (pour « droit ») doit valoir `len(liste)-1`. Pour le choix du pivot, il faut juste se souvenir qu'on cherche à appliquer cette fonction sur une liste de listes de propriétés (dont seule la première nous intéresse). Comme dans la suite du code (ligne 6 par exemple), on voit que pivot correspond à la valeur stockée dans la liste de liste, on peut choisir `pivot=liste[g][0]` par exemple comme pivot.

Q15 Il suffit de rajouter en toute première ligne de la fonction `triRapide` la vérification sur la taille `d-g` restant à vérifier

```

1 def triRapide(liste,g,d):
2     if d - g < 15:
3         triInsertion(liste,g,d)

```

```

4     else:
5         # On ne réécrit pas le reste de la fonction qui est inchangé.
6         # (à l'indentation près)

```

Remarquons que l'énoncé ne parle d'appliquer le `triInsertion` que sur des « sous-listes » de la liste initiale (ce qui fait que le test devrait plutôt avoir lieu aux environs des lignes 17 et 19) mais outre le fait que cela duplique inutilement du code, si on considère que le `triInsertion` est plus efficace que `triRapide` sur des « petites » listes, autant l'appliquer tout de suite sur la liste initiale si celle-ci est déjà suffisamment petite.

Q16 Pour procéder à un tri par insertion, on démarre à l'indice de gauche et on évolue sur la droite en prenant chaque nouvel élément et en décalant au fur et à mesure les éléments dans la liste jusqu'à atteindre sa place dans la partie triée (un peu comme vous écartez vos cartes pour laisser passer la carte que vous voulez rajouter dans votre jeu en partant du côté droit)

```

1  def triInsertion(liste,g,d):
2      for i in range(g+1,d+1):
3          j = i-1          # On regarde le plus à droite du jeu de carte déjà trié
4          tmp = liste[i] # On sort la carte de son jeu
5          # Tant que la carte à placer n'est pas assez grande par rapport à la
6          # carte courante et qu'on n'a pas parcouru toute la main,
7          while j >= g and liste[j][0] > tmp[0]:
8              liste[j+1] = liste[j] # on décale la carte courante vers la droite
9              j = j - 1             # et on passe à la suivante.
10         # Si on est sorti, c'est que sa place est à droite de la carte courante
11         liste[j+1] = tmp
12     return liste

```

À noter qu'on suppose toujours trier ici des listes de listes par rapport au premier élément, ce qui demande de comparer `liste[j][0]` à `tmp[0]` et pas seulement `liste[j]` à `tmp`.

Q17 Dans la fonction `skewness` proposée, on peut remarquer à la ligne 6 que la moyenne est recalculée pour chacune des hauteurs de la liste alors que sa valeur est fixe et pourrait n'être calculée qu'une unique fois. Comme un calcul de moyenne est forcément linéaire en la taille de la liste initiale, le faire pour chacune des n valeurs de hauteurs produit une fonction globalement quadratique en n , ce qui n'est guère optimal ici. Il suffit de rajouter le calcul `Hmoyen = moyenne(liste_hauteurs)` entre les lignes 4 et 5 (juste avant la boucle `for`) et utiliser cette valeur calculée dans la ligne 6 pour s'affranchir de ce souci.

Q18 Les calculs de S et K font chacun intervenir une unique boucle sur les valeurs de la liste (pour calculer la somme). Il ne devrait donc pas y avoir de différence de type de la complexité si les deux fonctions procédant à cette évaluation ont été programmées de la même façon¹.

Partie IV. Base de données relationnelle

Q19 La première requête est une simple sélection sur la localisation associée à une projection sur l'identifiant et le nom du site.

1. C'est-à-dire qu'il ne s'agit pas de calculer S en appliquant la correction de la questions Q17, mais ne pas faire de même pour K .

```
SELECT idBouee,nomSite FROM Bouee WHERE localisation = 'Mediterranee'
```

Pour la deuxième, il s'agit de vérifier que l'identifiant de la bouée n'est pas dans la table `Tempete` (soit que son nombre d'occurrences dans la table est nul), ce qui demande un aliasing pour ne pas confondre le `idBouee` voulu.

```
SELECT idBouee AS id FROM Bouee
WHERE (SELECT COUNT(*) FROM Tempete WHERE idBouee=id) = 0
```

Une autre possibilité revient à utiliser la notion d'appartenance (ou plutôt de non appartenance)

```
SELECT idBouee AS id FROM Bouee
WHERE idBouee NOT IN (SELECT idBouee FROM Tempete)
```

On peut aussi faire une différence de deux ensembles (avec l'instruction `MINUS` ou `EXCEPT` selon le GDBD utilisé)

```
SELECT idBouee AS id FROM Bouee
EXCEPT
SELECT idBouee FROM Tempete
```

La dernière requête va demander à faire à la fois une jointure (à l'aide de `JOIN`) et un regroupement par régions (à l'aide d'un `GROUP BY`).

```
SELECT nomSite, MAX(Hmax)
FROM Bouee JOIN Tempete ON Bouee.idBouee=Tempete.idBouee
GROUP BY nomSite
```

Partie V. Analyse « spectrale »

Q20 S'il suffit, pour calculer une TFD sur N éléments de calculer deux TFD sur $N/2$ éléments et faire les N calculs nécessaires à repasser des P_k et I_k aux X_k , alors si on note C_N le coût d'une TFD sur N éléments, on devrait avoir

$$C_N = 2 \times C_{N/2} + \alpha N = 2 \left(2 C_{N/4} + \alpha \frac{N}{2} \right) + \alpha N = 4 C_{N/4} + 2\alpha N = 8 C_{N/8} + 3\alpha N = \dots$$

Finalement, on obtient $C_N = N C_1 + \alpha N \times \log_2(N)$ puisqu'il y a $\log_2(N)$ divisions successives par 2, ce qui donne une complexité globale en $\mathcal{O}(N \log_2 N)$.

Q21 Pour écrire cet algorithme sous forme récursive, il faut remarquer que P_k et I_k correspondent respectivement aux transformées de Fourier des éléments d'indices respectifs pairs et impairs de la liste initiale. On peut donc demander le calcul desdites TFD pour calculer chacun des X_k .

```
1 def TFD(x):
2     N = len(x)           # Raccourci
3     if N == 1:         # On traite le cas particulier d'un élément unique
4         return x       # Dans ce cas, la TFD ne change rien
5     # Préparation des coefficients pairs et impairs
6     liste_pairs = [x[i] for i in range(0,N,2)]
7     liste_impairs = [x[i] for i in range(1,N,2)]
```

```
8     # Transformées de Fourier pour les coefficients pairs et impairs
9     P = TFD(liste_pairs)
10    I = TFD(liste_impairs)
11    X = [0]*N # Préparation du réceptacle
12    w = np.exp(-2*np.pi*1j/N)
13    for k in range(N//2):
14        X[k] = P[k] + w**k * I[k]
15        X[k+N//2] = P[k] - w**k * I[k]
16    return X
```

Pour vérifier si tout cela fonctionne bien, on peut regarder ce qu'il advient de la superposition de plusieurs signaux de différentes fréquences fabriqués « à la main ».

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  t = np.linspace(0,1,2**10)
5  f1 = 10
6  f2 = 30
7  f3 = 100
8  x = np.sin(2*np.pi*f1*t) + 3 * np.cos(2*np.pi*f2*t) - 5 * np.sin(2*np.pi*f3*t)
9
10 plt.plot(t,x)
11 plt.xlabel('Temps en s')
12 plt.ylabel('Signal')
13 plt.savefig('PNG/signal.png')
14 plt.clf()
15
16 FFT = TFD(x)
17 plt.plot(np.abs(FFT[:len(x)//2]))
18 plt.xlabel('Frequence en Hz')
19 plt.ylabel('Transformee de Fourier')
20 plt.xlim(0,150)
21 plt.savefig('PNG/TFD.png')
```

Ça a l'air de bien marcher !

