DS 3
Informatique
Mesure de houle
Extrait Concours Mines-Ponts 2017
Correction (Corrigé de J.J. Fleck)

Simon Dauguet
simon.dauguet@gmail.com

Mercredi 12 Février 2025

Partie I. Stockage interne des données

1. 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 $1200s$. Comme il y a deux mesures par seconde, cela correspond à 2400 mesures au total, soit une taille de $8 \times 2400 = 19200octets = 19,2ko$.

2. La campagne de mesure fait état d'un fichier récolté toutes les demi-heures pendant 15 jours. Il y a donc $15 \times 24 \times 2 = 7,2 \times 10^2$ fois les informations précédentes collectées, ce qui représente donc une taille totale de $13824000o$, soit $13,8Mo$.

Une carte-mémoire de $1Go$ est largement suffisante.

3. 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.

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

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

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

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

Partie II. Analyse "vague par vague"

5. 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 $H_1 \approx 6 - (-3) = 9m$, puis $H_2 \approx 7 - (-2) = 9m$ et $H_3 \approx 5 - (-1) = 6m$. Pour les mesures de périodes, on obtient

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

6. Pour calculer la moyenne, on peut procéder comme suit

```
def moyenne(liste_niveaux):
    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

```
def moyenne(liste_niveaux):
    m = 0
    for valeur in liste_niveaux:
        m = m + valeur
    return m/len(liste_niveaux)
```

7. 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`.

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

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

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

Néanmoins, cette implémentation a deux soucis : on doit nécessairement parcourir toute la liste (ce qui donne une complexité en $O(N)$ et non $O(1)$) *et* le calcul de la moyenne est lui-même déjà en $O(N)$. Pour avoir une méthode en $O(1)$ 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.

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

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

```
def construction_successeurs(liste_niveaux):
    n = len(liste_niveaux)
    successeurs = []
    m = moyenne(liste_niveaux)
    for i in range(n-1):
        if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < m:
            successeurs.append(i+1)
    return successeurs
```

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

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

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

11. 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

12. 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
```

13. 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.

14. Les calculs de S et K font chacun intervenir une unique boucle sur les valeur 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 façon [1].

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.

Partie IV. Analyse “spectrale”

15. 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

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

Finalement, on obtient $C_N = NC_1 + \alpha\, N \times \log_2(N)$ puisqu'il y a $\log_2(N)$ divisions successives par 2, ce qui donne une complexité globale en $O(N \log_2 N)$.

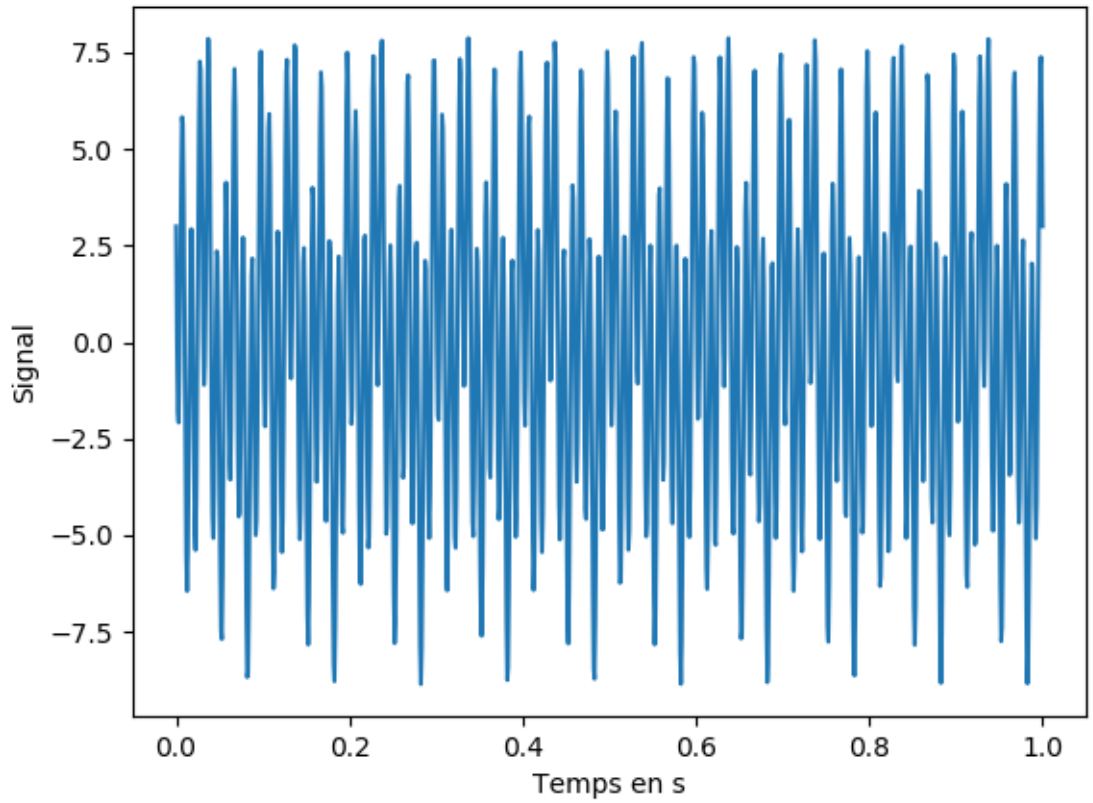
16. Pour écrire cet algorithme sous forme récursive, il faut remarquer que P_k et I_k correspondent respectivement aux transformées de Fourier des éléments d'indices respectifs pairs et impairs de la liste initiale. On peut donc demander le calcul desdites TFD pour calculer chacun des X_k.

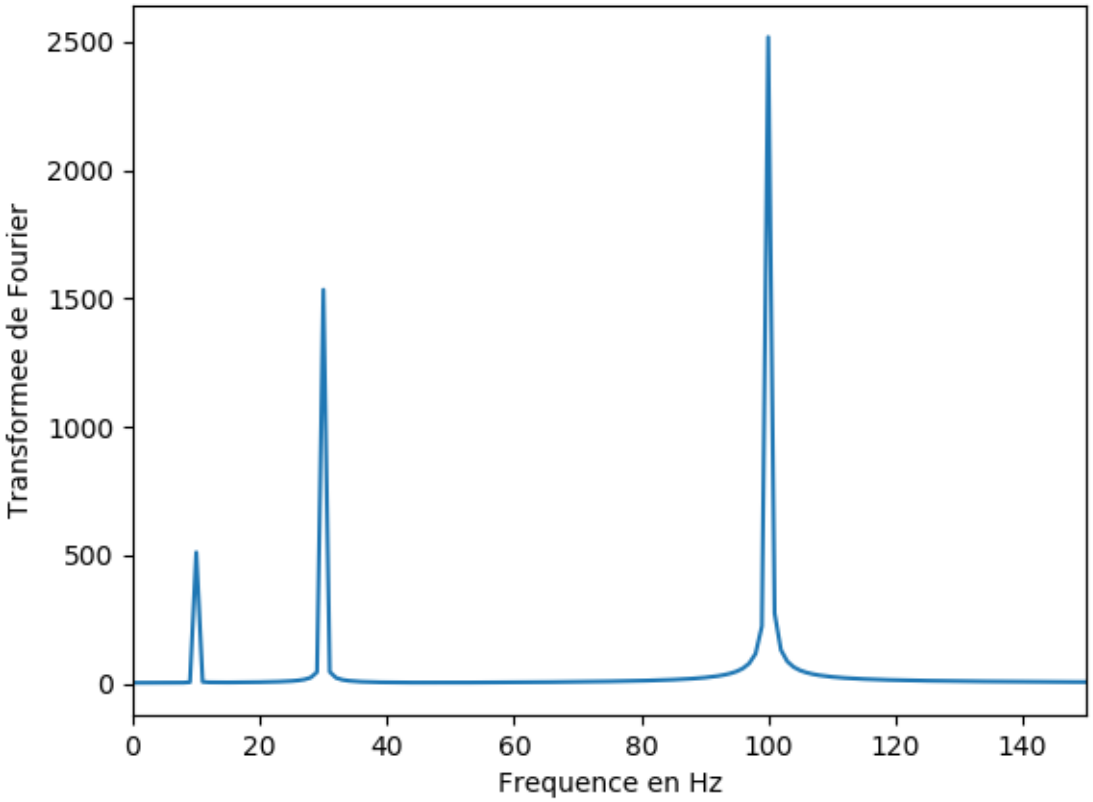
```
def TFD(x):
    N = len(x)       # Raccourci
    if N == 1:       # On traite le cas particulier d'un élément unique
        return x     # Dans ce cas, la TFD ne change rien
    # Préparation des coefficients pairs et impairs
    liste_pairs   = [x[i] for i in range(0,N,2)]
    liste_impairs = [x[i] for i in range(1,N,2)]
    # Transformées de Fourier pour les coefficients pairs et impairs
    P = TFD(liste_pairs)
    I = TFD(liste_impairs)
    X = [0]*N  # Préparation du réceptacle
    w = np.exp(-2*np.pi*1j/N)
    for k in range(N//2):
        X[k]      = P[k] + w**k * I[k]
        X[k+N//2] = P[k] - w**k * I[k]
    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”.

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0,1,2**10)
f1 = 10
f2 = 30
f3 = 100
x = np.sin(2*np.pi*f1*t) + 3 * np.cos(2*np.pi*f2*t) - 5 * np.sin(2*np.pi*f3*t)

plt.plot(t,x)
plt.xlabel('Temps en s')
plt.ylabel('Signal')
plt.savefig('PNG/signal.png')
plt.clf()

FFT = TFD(x)
plt.plot(np.abs(FFT[:len(x)//2]))
plt.xlabel('Frequence en Hz')
plt.ylabel('Transformee de Fourier')
plt.xlim(0,150)
plt.savefig('PNG/TFD.png')
```

Ça a l'air de bien marcher !


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Signal
0.0
0.2
0.4
0.6
0.8
1.0
Temps en s


2500
2000
1500
1000
500
0
Transformee de Fourier
0
20
40
60
80
100
120
140
Frequence en Hz