1 – Acquisition et traitement des données GPS

Des données sont enregistrées dans un fichier sous forme de chaînes de caractères. Il s'agit d'extraire ces données du fichier et de les préparer à un traitement numérique.

► **Récupération des données brutes**

► **Q1. Analyse du code fourni par le document réponse**

- L'argument de la ⟨*fonction*⟩ `recupererGPGGA` est la ⟨*chaîne de caractères*⟩ `chaine`.
  Cette fonction renvoie la ⟨*liste*⟩ `ligne`.
- On découpe la ⟨*chaîne*⟩ `chaine` en fonction du ⟨*caractère*⟩ `'\n'` pour obtenir la ⟨*liste de chaînes de caractères*⟩ `chaineDecoupe`.

```
chaineDecoupe = chaine.split('\n')
```

- Normalement, cette liste contient la trame GGA et d'autres trames.

```
indGGA = indice_GPGGA(chaineDecoupe)
```

La ⟨*fonction*⟩ `indice_GPGGA`, qui sera définie à la prochaine question, doit retourner l'indice `indGGA` qui situe la trame GGA dans la ⟨*liste*⟩ `chaineDecoupe`.
- Initialement, la ⟨*liste*⟩ `ligne` est vide.

```
ligne = []
if indGGA != len(chaineDecoupe):
```

Si l'indice retourné par la ⟨*fonction*⟩ `indice_GPGGA` est égal à la longueur de la ⟨*liste*⟩ `chaineDecoupe`, cela signifie que la trame GGA est absente de la liste. Dans ce cas, la ⟨*liste*⟩ `ligne` est renvoyée sans être modifiée et donc vide.
- Si l'indice renvoyé par la ⟨*fonction*⟩ `indice_GPGGA` est strictement inférieur à la longueur de la ⟨*liste*⟩ `chaineDecoupe`, cela signifie que la trame GGA est bien présente dans la liste : il s'agit de la ⟨*chaîne de caractères*⟩ `chaineDecoupe[indGGA]`.

```
ligne = chaineDecoupe[indGGA][1:]
ligne = ligne.split(',')
ligne.pop(len(ligne)-2)
```

On supprime le ⟨*caractère*⟩ `'$'` initial de la trame en considérant la ⟨*tranche*⟩ `chaineDecoupe[indGGA][1:]`.

On scinde ensuite la ⟨*chaine de caractères*⟩ `ligne` en fonction du ⟨*caractère*⟩ `','` pour définir une ⟨*liste de chaînes de caractères*⟩ `ligne`.

On élimine enfin l'avant-dernier élément de cette ⟨*liste*⟩ (il s'agit de l'élément dont l'indice est égal à `len(ligne)-2`), qui est toujours égal à la ⟨*chaîne*⟩ vide `''`.

- En résumé, si la trame GGA est présente dans l'⟨*argument*⟩ `chaine`, la fonction renvoie sous la forme d'une liste de chaînes de caractères les données séparées par des virgules dans la trame (à l'exception du caractère initial et de la donnée vide).

Si la trame GGA est absente, la fonction renvoie la liste vide.

► **Q2. Localisation de la trame GGA**

On parcourt la ⟨*liste*⟩ `listeChaine` : tant qu'on n'a pas trouvé l'indicatif de la trame GGA (c'est-à-dire la ⟨*chaîne de caractères*⟩ `'$GPGGA'`) et qu'on n'a pas atteint le terme de la liste (c'est-à-dire tant qu'il subsiste de l'espoir), on incrémente le compteur.

```
def indice_GPGGA(listeChaine):
  i = 0
  n = len(listeChaine)
    while (i<n) and (listeChaine[i][:6]!='$GPGGA'):
      i += 1
  return i
```

Finalement, la ⟨*chaîne de caractères*⟩ `'$GPGGA'` a été trouvée si, et seulement si, la valeur finale de i est strictement inférieure à n (= la longueur de la liste) ; elle est absente si, et seulement si, la valeur finale de i est égale à n.

❧ Au cas où la chaîne cherchée serait absente, le compteur i atteint la valeur n, le booléen (i<n) est alors égal à False et la boucle while s'arrête aussitôt : *le second booléen n'est même pas évalué.*

Heureusement ! S'il était évalué, il se produirait une erreur puisqu'on demanderait à accéder à l'élément d'indice n dans une chaîne de longueur n.

Autrement dit, au cas où la trame cherchée serait absente de l'argument, le code

```
while (listeChaine[i][:6]!='$GPGGA') and (i<n):
```

produirait une erreur (IndexError) *avant* d'évaluer le booléen (i<n).

► **Q3.**

On passe le temps nécessaire pour scruter l'énoncé (bien lire les préambules des alinéas **I.1** et **I.3**) et on écrit simplement ce qui suit.

```
listeGPGGABrute = recupererGPGGA(donneesGPS)
```

► Validité de la trame GPGGA

► **Q4.**

L'argument de la ⟨*fonction*⟩ testPresence est la ⟨*liste de chaînes de caractères*⟩ trame. On parcourt cette liste en vérifiant qu'aucune des chaînes qui la composent n'est vide (longueur strictement positive, par exemple).

```
def testPresence(trame):
  non_vide = True
  for s in trame:
    non_vide = non_vide and (len(s)>0)
  return non_vide
```

On rappelle que le ⟨*booléen*⟩ (A and B) est égal à True si, et seulement si, les deux booléens A et B sont égaux à True. De la sorte,

— ou bien le booléen non_vide est toujours égal à True;

— ou bien à partir d'un certain rang il est toujours égal à False.

❧ L'avant-dernière valeur de la trame GGA initiale est toujours vide (par définition), mais on a bien pris soin d'éliminer cette valeur lors de la définition de la ⟨*liste*⟩ listeGPGGABrute (**Q1**).

► **Q5.**

On prend le temps de bien compter sur l'exemple fourni par l'énoncé : la précision est donnée en neuvième position dans la liste (donc l'indice est égal à 8).

```
def testPrecision(trame):
  precision = float(trame[8])
  return (precision<5)
```

► **Q6.**

L'argument de la ⟨*fonction*⟩ testCS est une ⟨*liste de chaînes de caractères*⟩ trame. Ainsi, trame[-1] est le dernier élément de la ⟨*liste*⟩ trame et la ⟨*chaîne de caractères*⟩ trame[-1][1:] est la chaîne de contrôle privée de son caractère initial (le ⟨*caractère*⟩ '*') : la ⟨*chaîne de caractères*⟩ somme est donc la représentation hexadécimale de la somme de contrôle.

La variable somme est donc bien nommée.

❧ Dans la ⟨*liste*⟩ trame, on a déjà éliminé le ⟨*caractère*⟩ '$' initial. En contrôlant la ⟨*boucle*⟩ for avec trame[:-1], on élimine aussi le dernier élément de la liste, c'est-à-dire la chaîne finale qui commence par le ⟨*caractère*⟩ '*'.

❧ Dans la sous-⟨*boucle*⟩ for ch in chaine:, on parcourt chaque élément de la ⟨*liste*⟩ trame[:-1] caractère par caractère.

❧ D'après l'annexe, ord(ch) est l'⟨*entier*⟩ associé au ⟨*caractère*⟩ ch dans la table ASCII.

Toujours d'après l'annexe, $0 \wedge n = n$ pour tout entier n. La valeur finale de l'⟨*entier*⟩ cs est donc bien le résultat de l'opération *ou exclusif* entre les représentations binaires des caractères compris strictement entre '$' et '*'.

La variable cs est donc bien nommée elle aussi (*cs* pour *checksum*).

❧ La ⟨*chaîne*⟩ hex(cs) est la représentation hexadécimale de la somme de contrôle cs. La ⟨*chaîne*⟩ hex(cs)[2:] est donc la même ⟨*chaîne de caractères*⟩, privée du préfixe '0x'.

❧ En théorie, les deux ⟨*chaînes de caractères*⟩ somme et csHex doivent donc être égales.

Par conséquent, le ⟨*booléen*⟩ csHex==somme renvoyé est égal à True si, et seulement si, la somme de contrôle calculée sur les caractères reçus est bien égale à la somme de contrôle calculée par l'émetteur.

❧ Il y a une faute de frappe amusante dans l'énoncé : la question évoque la ⟨*fonction*⟩ testSC au lieu de la ⟨*fonction*⟩ testCS. Si on prend la question posée au pied de la lettre, la valeur retournée est une ⟨*erreur*⟩ NameError, puisqu'on évalue une fonction qui n'est pas définie !

❧ Passons sur la faute de frappe : il s'agit de calculer ici la somme de contrôle sur les caractères situés entre '$' et '*', c'est-à-dire sur les caractères de la ⟨*chaîne*⟩ 'GPG', autrement dit d'évaluer l'expression suivante.

```
ord('G') ^ ord('P') ^ ord('G')
```

Comme l'opération *ou exclusif* est associative et commutative et qu'elle admet 0 pour élément neutre :

$$G \wedge P \wedge G = (G \wedge G) \wedge P = 0 \wedge P = P.$$

D'après l'énoncé, la représentation binaire du ⟨*caractère*⟩ `'P'` est `'0b01010000'` et d'après l'annexe 2, la représentation hexadécimale de cet entier est `'0x50'` (pour ceux qui se sentent mal hors de la base 10, il s'agit de $80 = 65 + (16 - 1)$ - puisque P est la seizième lettre de l'alphabet et que la lettre A est représentée par l'entier 65).

Les ⟨*chaînes*⟩ `csHex` et `somme` étant respectivement égales à `'50'` et à `'A0'`, elles sont différentes et la valeur renvoyée est donc le ⟨*booléen*⟩ `False`.

► Récupération des données

► Q7.

D'après la norme GPGGA, la ⟨*chaîne*⟩ `chHoraire` se décompose en trois parties : les deux premiers caractères donnent l'heure (à convertir en secondes) ; les deux suivants donnent les minutes (à convertir en secondes) ; les derniers donnent les secondes (à conserver telles quelles).

Bien entendu, il faut convertir toutes les ⟨*chaînes de caractères*⟩ en ⟨*flottants*⟩ avant d'effectuer les calculs !

```
def convHoraire(chHoraire):
  heures = float(chHoraire[:2])
  minutes = float(chHoraire[2:4])
  secondes = float(chHoraire[4:])
  horaire = secondes + 60*(minutes + 60*heures)
  return horaire
```

► Q8.

On commence par déterminer le signe de l'angle : d'après l'énoncé, les latitudes Nord (repérées par le ⟨*caractère*⟩ `'N'`) et les longitudes Ouest (probablement repérées par le ⟨*caractère*⟩ `'W'`) sont comptées positivement.

Les angles sont données en degrés et minutes (décimales !!!), on demande qu'ils soient convertis en degrés décimaux. Une difficulté bien subtile : les degrés sont représentés avec 2 ou 3 caractères ! (Pourquoi tant de haine ?)

Comme les minutes sont données sont représentees sur 8 caractères, on va scinder la ⟨*chaîne*⟩ `chAngle` en deux : d'un côté, les 8 derniers caractères (les minutes) et de l'autre, ce qui est à gauche des 8 derniers caractères (les degrés).

Comme il y a 60 minutes d'angle dans un degré, il suffit de diviser le nombre de minutes par 60 pour obtenir la valeur décimale de l'angle.

```
def convAngle(chAngle, chCard):
  angle_positif = (chCard=='N') or (chCard=='W')
  if angle_positif:
    signe = 1
  else:
    signe = -1
  minutes = float(chAngle[-8:])
  degres = float(chAngle[:-8])
  angle = signe*(degres + minutes/60)
  return angle
```

► Q9.

Question de synthèse. Il faut compter sur ses doigts pour repérer les positions des différentes ⟨*chaînes de caractères*⟩ dans la ⟨*liste*⟩ `trame`, en se souvenant que les indices commencent à 0.

L'annexe n'est pas claire au sujet de l'altitude (il semble qu'il y ait deux données dans la trame à ce sujet). La lecture du préambule de la question 10 semble indiquer que la valeur de l'altitude est donnée par l'élément d'indice 9 dans la ⟨*liste*⟩ `trame`. (Il faut vraiment *tout* lire — mais comment faire, et faire bien, en si peu de temps ?)

```
def recupDonnees(trame):
  Horaire = convHoraire(trame[1])
  Latitude = convAngle(trame[2], trame[3])
  Longitude = convAngle(trame[4], trame[5])
  Altitude = float(trame[9])
  return [Horaire, Latitude, Longitude, Altitude]
```

► **Sauvegarde d'une activité**

► **Q10.**

Question très classique, avec des subtilités classiques !
D'après l'énoncé,
— l'horaire est codé sur 5 caractères ;
— chacun des deux angles est codé sur 11 caractères au plus (deux ou trois chiffres pour la partie entière, six décimales, le point décimal et le signe éventuel) ;
— l'altitude est codée sur 6 caractères (quatre pour la partie entière, un seul pour la partie décimale [*"précision au décimètre"*] et le point décimal) ;
— la fréquence cardiaque est codée sur 3 caractères ;
— ces cinq valeurs sont séparées par 4 virgules et on ajoute 1 saut de ligne au bout de la chaîne.
On n'a rien oublié : chaque seconde, on enregistre une chaîne de 41 caractères (au maximum).

Chaque caractère est codé sur un octet, donc il faut stocker environ 40 octets par seconde pendant 200 heures, c'est-à-dire $7,2.10^5$ secondes. Cela nous donne 288.10^5 octets, c'est-à-dire un peu moins de 30 Mo.

► **Q11.**

Il faut diminuer d'un tiers environ la taille des données enregistrées, ce qui reviendrait à n'enregistrer que 25 caractères chaque seconde...

❧ On a pris six chiffres après la virgule pour la latitude et la longitude. Si on n'en avait pris que trois, on n'aurait économisé que 6 caractères en tout au prix d'une perte de précision considérable... (Un millième de degré correspond à environ 20 mètres au sol, vérifiez sur Google Maps !) Ce n'est pas une bonne idée.

Si on accepte de perdre en précision, il serait plus pertinent de n'enregistrer les données que toutes les deux secondes, le gain de place serait sensible (50% d'économie !) et la perte de précision peu importante...

❧ Meilleure idée : on pourrait enregistrer les données numériques au format binaire plutôt que sous forme de chaînes de caractères. Ainsi, 2 octets pour l'horaire, 4 octets pour la latitude et la longitude, 2 octets pour l'altitude et 1 octet pour la pulsation, on tombe à 13 octets par enregistrement (sans doute plus à cause de la structure qui contient les données, mais c'est une vraie piste).

❧ Autre possibilité : les 200 heures d'enregistrement couvrent plus d'une activité (même pour les fadas des ultra-trails). On pourrait imaginer qu'une activité achevée soit stockée sous forme compressée, on pourrait ainsi facilement diviser par deux la place occupée en mémoire.

❧ Si vous avez d'autres suggestions raisonnables, je suis preneur !

2 – Acquisition du rythme cardiaque

► **Application d'un filtre passe-bas**

Il n'est peut-être pas inutile de revenir sur la notion de **filtre passe-bas**.

❧ Un filtre (analogique) est un dispositif modélisé par une équation différentielle reliant un *signal d'entrée* $e(t)$ et un *signal de sortie* $s(t)$. Dans le cas qui nous occupe ici :

$$\frac{ds(t)}{dt} + 2\pi f_c s(t) = 2\pi f_c e(t).$$

Pour comprendre comment l'entrée et la sortie sont liées par cette équation différentielle, on va supposer que le signal d'entrée est sinusoïdal :

$$e(t) = \sin 2\pi f t$$

où $f > 0$ est la fréquence du signal d'entrée.

❧ La solution générale de cette équation différentielle est

$$s(t) = K.e^{-2\pi f_c t} - \frac{f/f_c}{1+(f/f_c)^2} \cdot \cos 2\pi f t + \frac{1}{1+(f/f_c)^2} \cdot \sin 2\pi f t.$$

Le premier terme dépend de la condition initiale mais c'est sans réelle importance : pour $2\pi f_c t > 5$, il est négligeable devant les deux termes suivants, qui donnent la **réponse en régime permanent**

$$s_0(t) = \frac{-f/f_c}{1+(f/f_c)^2} \cdot \cos 2\pi f t + \frac{1}{1+(f/f_c)^2} \cdot \sin 2\pi f t.$$

La fonction s_0 est une fonction périodique, de période f *comme le signal d'entrée*.

Il existe un réel φ (qui dépend de la fréquence f) tel que

$$s_0(t) = A_0 \cdot \sin(2\pi f t + \varphi) \qquad \text{avec} \qquad A_0 = \frac{1}{\sqrt{1+(f/f_c)^2}}.$$

❧ En laissant de côté le **déphasage** φ, on voit que le filtre transforme un signal $e(t)$ d'amplitude 1 en un signal $s(t)$ d'**amplitude** A_0.

Lorsque la fréquence f est basse, c'est-à-dire pour $^f/_{f_c} \ll 1$, l'amplitude A_0 est proche de 1 et l'effet du filtrage se limite à un déphasage.

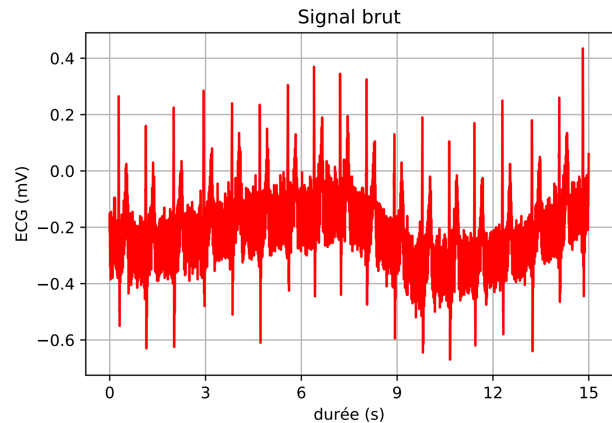
Au contraire, lorsque la fréquence f est assez haute pour que $^f/_{f_c} \gg 1$, l'amplitude A_0 est proche de 0 : le signal d'entrée est pour ainsi dire annulé par le filtrage.

La fréquence f_c sur laquelle repose la discussion précédente est appelée la **fréquence de coupure**. Lorsque la fréquence f est voisine de la fréquence de coupure, le signal est un peu amorti sans être vraiment annulé : pour $f = f_c$, l'amplitude A_0 est égale à $\sqrt{2}/_2$, soit environ 70%.

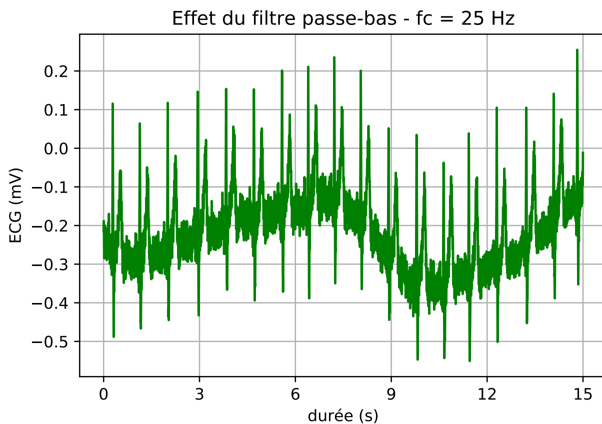
La théorie de Fourier (séries ou intégrales) décrit un signal quelconque comme une superposition de signaux sinusoïdaux. Puisque l'équation différentielle est linéaire, on peut en déduire que le filtrage va conserver les termes de basses fréquences ($f \ll f_c$) et faire disparaître les termes de hautes fréquences ($f \gg f_c$), d'où l'expression **filtre passe-bas**.

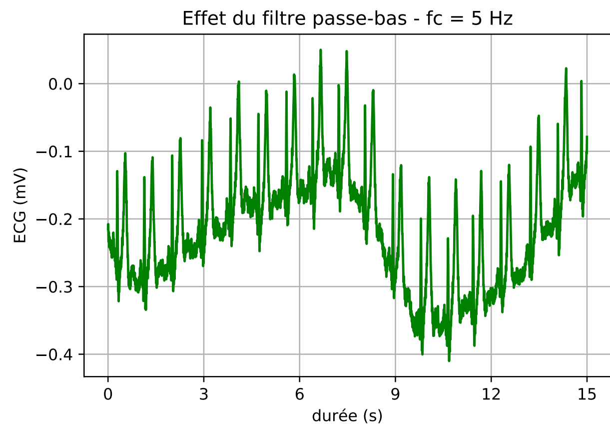
❧ En appliquant un filtre passe-bas, on élimine les perturbations dues aux hautes fréquences - mais on perd également certaines informations du signal qui n'apparaissent qu'aux hautes fréquences et on n'élimine pas les perturbations de basses fréquences... On ne doit donc pas espérer qu'un simple filtre passe-bas transforme un signal expérimental (et donc bruité) en un signal proche du modèle théorique !

Voici un ECG brut (données récupérées sur `https://physionet.org/lightwave/?db=ecgiddb/1.0.0`, il s'agit du premier enregistrement du patient 2).



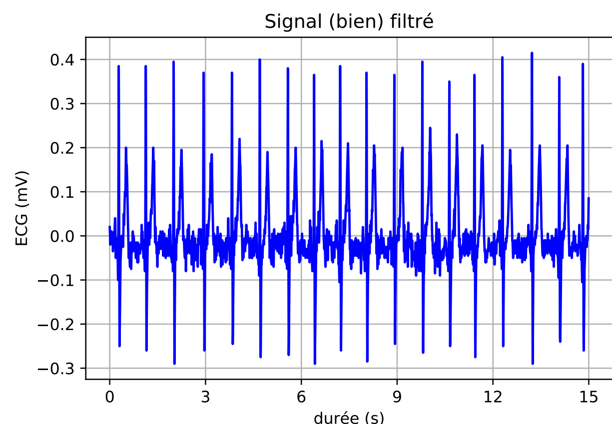
En appliquant à ce signal d'entrée le filtre passe-bas défini en **Q13** pour deux fréquences de coupure distinctes, on récupère les signaux suivants.





Avec la fréquence de coupure la plus basse, on voit bien que, on a éliminé les oscillations de haute fréquence en perdant du même coup certaines caractéristiques de l'ECG (le premier pic, très bref, est devenu moins ample que le second pic, moins bref). On voit aussi qu'on a bien mis en relief les variations de basse fréquence !

Le site `physionet.org` fournit également un signal filtré dans les règles de l'art : les oscillations de basse fréquence ont été éliminées (le signal est maintenant centré sur l'origine) ; les bruits ont été sensiblement réduits lorsque le signal est proche de l'origine et on a conservé l'amplitude des variations très brèves.



▶ **Q12.**

En discrétisant l'équation intégrale (2) avec la méthode des trapèzes, on obtient

$$s_{k+1} = s_k + 2\pi f_c \cdot [(t+T_e) - t] \cdot \frac{(e_{k+1} - s_{k+1}) + (e_k - s_k)}{2} = s_k + \pi f_c T_e \cdot (e_{k+1} + e_k - s_k - s_{k+1})$$

En posant

$$G = \pi f_c T_e = \pi \frac{f_c}{f_e},$$

on en déduit que $(1+G)s_{k+1} = G(e_{k+1} + e_k) + (1-G)s_k$, c'est-à-dire

$$s_{k+1} = \frac{G}{1+G}\,(e_{k+1} + e_k) + \frac{1-G}{1+G}\,s_k.$$

▶ **Q13.**

L'énoncé se borne à donner une relation de récurrence, sans indiquer comment la valeur initiale est choisie, ni donner le domaine dans lequel l'indice k varie...

Nous partirons du principe que $s_0 = e_0$ (le filtrage n'a pas d'effet sur la valeur initiale) et que la longueur du tableau retourné est égale à la longueur du tableau fourni en argument (ce qui permettra de les comparer graphiquement sans difficulté).

Première version avec des listes.

```
def filtrage(entree, G):
  s = entree[0]
  sortie = [s]
  for k in range(1, len(entree)):
    s = ((1-G)*s+G*(entree[k]+entree[k-1]))/(1+G)
    sortie.append(s)
  return np.array(sortie)
```

Deuxième version avec des tableaux numpy (chacun ses goûts).

```
def filtrage(entree, G):
  N = len(entree)
  sortie = np.zeros(N)
  sortie[0] = entree[0]
  for k in range(1, N):
    sortie[k] = ((1-G)*sortie[k-1] + G*(entree[k]+entree[k-1]))/(1+G)
  return sortie
```

▶ **Première méthode : récupération des pics**

▶ **Q14.**

Lors de la i-ème itération, les variables `P0`, `P1` et `P2` contiennent respectivement les valeurs de P_{i-1}, P_i et P_{i+1}. La boucle `while` se poursuit tant qu'on n'a pas détecté un pic, c'est-à-dire tant que

$$P_{i-1} > P_i \quad \text{ou} \quad P_{i+1} > P_i$$

ou qu'on détecte des maxima trop faibles :

$$P_{i-1} < \text{seuil} \quad \text{ou} \quad P_{i+1} < \text{seuil}.$$

(Inutile de vérifier si on a bien $P_i \geqslant$ seuil !)

Normalement, au cours d'un intervalle de 3 secondes, un tel pic est observé, mais il n'est pas mauvais de vérifier que l'indice $(i+2)$ reste strictement inférieur à la longueur de la $\langle$*liste*$\rangle$ `extSignal`.

```
def detectionPics(extSignal, seuil):
  P0, P1, P2 = extSignal[:3]
  i = 1
  while i<len(extSignal)-2 and (P0<seuil or P2<seuil or P1<P0 or P1<P2):
    P0, P1 = P1, P2
    P2 = extSignal[i+2]
    i += 1
  return i
```

► **Q15.**

❧ La durée T_e est l'intervalle de temps entre deux mesures consécutives, dont $1/T_e$ est le nombre de mesures effectuées chaque seconde et $3/T_e$ est le nombre de mesure effectuées pendant l'intervalle de 3 secondes (durée pendant laquelle on est sûr de détecter un pic, selon l'énoncé).

L'⟨*entier*⟩ `nbPoints` est, comme son nom l'indique, le nombre de points se trouvant dans la "fenêtre" étudiée.

❧ L'indice de début est l'⟨*entier*⟩ `deb` et on se restreint aux indices strictement inférieurs à l'⟨*entier*⟩ `deb + nbPoints`.

La condition sur la ⟨*boucle*⟩ `while` indique qu'on parcourt la totalité de la ⟨*liste*⟩ `signal`.

❧ On utilise la ⟨*fonction*⟩ `detectionPics` pour détecter l'indice où est atteint le premier pic rencontré. Cet indice est alors placé dans la ⟨*liste*⟩ `pics` et est pris comme nouvelle valeur de l'indice ⟨*entier*⟩ `deb` (pour localiser le pics suivant).

❧ À la sortie de la ⟨*boucle*⟩ `while`, on a localisé tous les pics. On calcule la différence entre l'indice du premier pic : `pics[0]` et l'indice du dernier pic : `pics[-1]`. On la multiplie par la durée d'échantillonnage T_e pour obtenir la durée qui sépare le premier pic du dernier pic.

L'⟨*entier*⟩ `len[pics]` donne le nombre de pics localisés, donc la différence `len[pics]-1` donne le nombre d'intervalles entre pics consécutifs et le quotient `periode_moy` donne la durée moyenne (en secondes) qui sépare deux pics consécutifs.

❧ La valeur renvoyée est donc le nombre moyen de pics observés pendant une durée de 60 secondes, c'est-à-dire la pulsation cardiaque (en bpm).

REMARQUE.— J'ai donné tous les détails qui me paraissaient utiles pour bien comprendre ce qui se passe dans ce code. Il n'est absolument pas nécessaire d'entrer à ce point dans les détails sur sa copie, on peut se contenter de donner la signification des quantités calculées sans se justifier.

► **Seconde méthode : transformée de Fourier discrète**

► **Q16.**

Si elle ne rappelle pas les opérations qu'on peut effectuer sur les tableaux grâce au module `numpy`, l'annexe mentionne tout de même que c'est *pratique*. C'est pourquoi je n'hésite pas à donner un code très compact, comme le permet `numpy`.

```
def TFD(signal, Te):
  N = len(signal)
  n_sur_N = np.arange(0, 1, 1/N)
  frequences = n_sur_N/Te
  e = np.exp(-2j*np.pi*n_sur_N)
  S = [ np.abs(np.sum(signal*e**k)) for k in range(N) ]
  return list(frequences), S
```

Quelques explications...

La ⟨*fonction*⟩ `np.arange(0, 1, 1/N` renvoie des réels compris entre 0 (inclus) et 1 (exclu) séparés par la valeur $1/N$, c'est-à-dire la famille $(n/N)_{0\leqslant n<N}$.

D'après l'énoncé, la famille des fréquences prises en compte est

$$\left(\frac{k}{N}\cdot f_e\right)_{0\leqslant k<N} = \left(\frac{n}{N}\cdot\frac{1}{T_e}\right)_{0\leqslant n<N}$$

(puisque les indices sont muets). Cette famille est donc représentée par le ⟨*tableau*⟩ `n_sur_N/Te`.

Le ⟨*tableau*⟩ `e` représente la famille $\big(\exp(-2i\pi\frac{n}{N})\big)_{0\leqslant n<N}$ et le produit `signal*e**k` représente la famille

$$\left(s_n\cdot\exp\frac{-2i\pi kn}{N}\right)_{0\leqslant n<N}.$$

Il reste à calculer la somme de ces complexes, puis à calculer le module de cette somme et ce, pour tout entier $0\leqslant k<N$.

► **Q17.**

Par définition,

$$S_0 = \sum_{n=0}^{N} s_n.$$

D'après la Figure (1) de l'énoncé, la valeur moyenne du signal étudié est de l'ordre de 300 (unité indéterminée). Or le signal dure environ 6 secondes et comme la période d'échantillonnage $T_e = 2.10^{-2}$ s, on a $N\approx 6\times 50 = 300$ et donc $S_0\approx 9.10^4$, ce qui est cohérent avec l'ordre de grandeur observé sur la Figure (2).

► **Q18.**

Il s'agit essentiellement de convertir les pulsations extrêmes (30 et 220 bpm) en fréquences (exprimées en s^{-1}), puis en indices entiers.

À la pulsation ω (en bpm), on associe la fréquence $f=\omega/60$ (en s^{-1}) et, d'après l'énoncé, cette fréquence correspond f à l'entier

$$k = N\cdot\frac{f}{f_e} = N\cdot f\cdot T_e = N\cdot\omega\cdot\frac{T_e}{60}.$$

Il est essentiel d'utiliser la ⟨*fonction*⟩ `int` pour récupérer des entiers !

On effectue alors les mêmes calculs que précédemment, mais seulement pour les entiers k compris entre les deux valeurs extrêmes ainsi déterminées.

```
def TFD_modifiee(signal, Te):
  puls_min, puls_max = 30, 220
  N = len(signal)
  k_min, k_max = int(N*puls_min*Te/60), int(N*puls_max*Te/60)+1
  n_sur_N = np.arange(0, 1, 1/N)
  frequences = (n_sur_N/Te)[k_min:k_max]
  e = np.exp(-2j*np.pi*n_sur_N)
  S = [ np.abs(np.sum(signal*e**k)) for k in range(k_min, k_max) ]
  return list(frequences), S
```

► **Q19.**

On calcule la fréquence cardiaque moyenne sur un intervalle du signal qui compte 600 points (la différence entre l'indice `indFin` et l'indice `indDepart`).

La fréquence d'échantillonnage f_e (qui est bien égale à l'inverse de la durée T_e) indique qu'on mesure 50 valeurs par seconde.

La durée Δt est donc $\frac{600}{50} = 12$ secondes. Cette durée semble suffisante pour calculer la fréquence cardiaque avec une précision convenable : elle couvre environ au moins dix périodes sans que sa durée risque de lisser abusivement des variations de fréquence.

❧ Ligne 69 : La ⟨*fonction*⟩ `TFD` nous donne d'une part le ⟨*tableau*⟩ `freq` des fréquences et d'autre part le ⟨*tableau*⟩ `Sk` des amplitudes.

Dans un premier temps, on calcule la plus grande valeur du ⟨*tableau*⟩ `Sk` (avec la ⟨*fonction*⟩ `max`), puis on cherche un indice $i_{depart} \leqslant i_0 < i_{fin}$ tel que le maximum de `Sk` soit égal à `Sk[i0]`.

On prend alors la fréquence `freq[i0]` (valeur en s^{-1}), qu'on multiplie par 60 pour obtenir une pulsation en bpm. Cette pulsation est alors enregistrée dans la ⟨*liste*⟩ `HR`.

REMARQUE.— Ce qui précède n'a de sens que si le ⟨*tableau*⟩ `Sk` présente un maximum net, atteint une seule fois ou presque (il ne serait pas gênant que ce maximum fût atteint pour plusieurs fréquences relativement proches, ce serait vraiment gênant s'il était atteint pour des fréquences plutôt éloignées).

❧ Lignes 70 et 71 : Pour estimer la fréquence cardiaque toutes les secondes, il faut décaler la partie étudiée du signal d'une seconde. La fréquence d'échantillonnage f_e nous donne le nombre de valeurs mesurées chaque seconde : il faut donc décaler les deux indices `indDepart` et `indFin` de la valeur de `fe`.

```
indDepart += fe
indFin += fe
```

REMARQUE.— Une faute d'énoncé : le second argument de la ⟨*fonction*⟩ `TFD` devrait être la durée T_e et non pas la fréquence f_e (ligne 68).

► **Q20.**

Le ⟨*tableau*⟩ `hann` contient les valeurs de

$$\frac{1 - \cos \frac{2\pi t}{T_{max}}}{2} \qquad \text{pour} \qquad 0 \leqslant t \leqslant T_{max}.$$

Ces valeurs sont donc positives, nulles pour $t = 0$ et pour $t = T_{max}$: seul le graphe en haut à droite possède ces propriétés. En outre, on constate que le maximum est égal à 1 et atteint au milieu de l'intervalle, pour $t = T_{max}/2$.

► **Q21.**

D'après l'énoncé, il s'agit de remplacer la fenêtre rectangulaire par la fenêtre de Hann *avant* d'appliquer la TFD sur le signal.

```
while (indFin<nbPoints):
  partSignal = SignalFiltre[indDepart:indFin]     # fenêtre rectangulaire
  h = Hann(partSignal, fe)                        # fenêtre de Hann
  freq, Sk = TFD(h, Te)                           # TFD
  HR.append(60*freq[Sk.index(max(Sk))])
  indDepart += fe
  indFin += fe
```

3 – Partage des activités

► **Q22.**

Les identifiants des activités (resp. des membres) sont les valeurs de l'attribut `Ida` (resp. `Idm`). Ces deux attributs se trouvent dans la table `activite`.

Le code SQL est donc sans mystère.

```
SELECT Ida
FROM activite
WHERE Idm=1
```

► **Q23.**

Le quotient `Distance/Temps` donne la vitesse moyenne lors de l'activité réalisée, mais cette vitesse est exprimée en mètres par seconde et on attend un résultat en kilomètres par heure : il faut convertir !

Par ailleurs, il est judicieux de donner un alias à ce nouvel attribut, afin que le résultat soit plus lisible.

```
SELECT DATE, Distance, 3.6*Distance/Temps AS Vitesse_Moyenne
FROM activite
WHERE Type="course" AND Idm=1
```

❧ Comme on le voit sur le code précédent, **`DATE`** est un mot réservé en SQL. Ce n'est donc pas judicieux de l'utiliser pour identifier un attribut...

► **Q24.**

Baptisons *Alice* le membre dont l'identifiant est 1.

Dans la table `amis`, Alice peut apparaître aussi bien dans la première colonne que dans la deuxième colonne. Les amis d'Alice sont extraits de la première colonne (première sous-requête) et de la deuxième colonne (deuxième sous-requête).

Dans les deux cas, le résultat de la sous-requête est une table contenant un seul attribut, nommé `idam1` (pour identifiant des amis de 1).

En réunissant (avec l'opérateur **`UNION`**) les tables produites par ces deux sous-requêtes, on crée une table, judicieusement renommée `amis1`, dont l'unique attribut est `idam1`.

On réalise alors une jointure entre la table `activite` et la table `amis1` : en se restreignant aux membres qui sont des amis d'Alice (clause `ON activite.idm=amis1.idam1`), on dresse la table contenant les identifiants des activités de type "marche".