exo2-solution

September 29, 2023

0.1 Exercice 1 : décortiquer un problème pour des publications

** Voir la solution de la question 1 **

Proposition de correction :

1. Interroger la base de données pour avoir accès la liste des publications du laboratoire, la limiter à 300 identifiants que l'on peut mettre dans une liste `list_publications` par exemple.
2. Écrire une fonction qui tire des nombres aléatoires en vérifiant qu'ils n'ont pas déjà été tirés : pour cela,
   - faire une liste `random_numbers` vide
   - tirer un nombre aléatoire compris entre 0 et 299
   - vérifier que le nombre tiré n'est pas déjà dans `random_numbers` (sinon on recommence)

   On peut aussi faire une liste contenant les chiffres compris entre 0 et 299, la mélanger puis ne garder que les 5 premiers éléments pour former `random_numbers`.
3. afficher les éléments de `list_publications` ayant les indices fournis par la liste `random_numbers`

** Voir la solution de la question 2 **

Proposition de correction :

1. Interroger la base de données pour avoir accès la liste des publications que l'on peut mettre dans une liste `list_publications` par exemple.
2. Créer une liste `pair_authors` qui va stocker 3 informations : `author1`, `author2` et `common_publications`
3. Écrire une fonction qui pour chaque élément de la liste `list_publications`:
   - interroge la base pour avoir accès à la publication que l'on peut stocker dans `publication`
   - parcourir la liste des auteurs `list_authors` de `publication`
   - regarder si l'auteur a une de ses affiliations qui est le laboratoire concerné, si c'est le cas, le placer dans une liste `authors_affiliated`
   - une fois tous les auteurs de la publication analysés, s'il y a plus de deux auteurs affiliés : faire une double boucle imbriquée sur deux indices `i1` et `i2` (i1 allant de 0 à la taille de la liste `authors_affiliated`-1) et i2 allant de i1+1 à la taille de la liste `authors_affiliated`-1. Pour chaque paire, regarder si elle existe déjà (en faisant attention sur le fait que l'ordre ne correspond peut-être pas à celui donné dans `pair_authors`) * si les deux auteurs coincident avec un élément de `pair_authors`, alors on ajoute le DOI à la liste `common_publications`.
     - si les deux auteurs ne sont pas dans la liste, alors on les ajoute à `pair_authors`

L'algorithme est ici grossier, il pourrait être intéressant de stocker de manière ordonnée les auteurs

(par exemple ordre alphabétique). Faire un tri alphabétique de `authors_affiliated` pour ensuite simplifier le test d'existance dans la base de donnée.

On pourrait également stocker les listes `authors_affiliated` pour chaque publication d'un côté, la liste des auteurs ayant des co-auteurs de l'autre puis faire l'analyse à la fin pour toutes les paires possibles.

1 Exercice 2

[1]:
```
def count_base_dna(dna, base):
    """
    Counts the number of occurence of `base` (str) in `dna` (str)
    """
    i = 0 # counter
    for char in dna:
        if char == base:
            i += 1
    return i

dna = 'ATGCGGACCTAT'
base = 'C'
n = count_base_dna(dna, base)
# or (new) format string syntax
print('{base} appears {n} times in {dna}'.format(base=base, n=n, dna=dna))
```

```
C appears 3 times in ATGCGGACCTAT
```

2 Exercice 3

Voici un exemple de correction en français

[2]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Voici un exemple de code qui prend la valeur du signal Is mesuré pour une gamme␣
 ↪étalon
dont les concentrations initiales Cs sont connues. Ensuite, on calcule␣
 ↪l'absorbance définie comme :
$$A = - \log\left(\dfrac{I}{I_0}\right) $$ où I0 est l'absorbance du blanc.

Ayant deux points, on extrait ensuite la droite affine passant par ces deux␣
 ↪points
pour avoir la réponse du détecteur.

Code proposé par Martin Vérot sous licence CC-BY-NC-SA
"""
```

```python
import numpy as np

def calc_absorbance(I,I0):
    """
    Calcul de l'absorbance
    - I est l'intensité lumineuse en sortie de l'échantillon (float)
    - I0 est l'intensité lumineuse pour le blanc (float)
    retourne l'absorbance (float)
    """
    return -np.log(I/I0)

def calc_droite(Abs,Cs):
    """
    Calcul de la pente et l'ordonnée à l'origine à partir de deux points
    - Abs : liste des absorbances de longueur 2
    - Cs : liste des concentrations de longueur 2
    retourne la pente et l'ordonnée à l'origine (tuple)
    """
    a = (Abs[1]-Abs[0])/(Cs[1]-Cs[0])
    b = Abs[1]-a*Cs[1]
    return a,b #slope, intercept

if __name__ == "__main__":
    #Intensité lumineuse pour le blanc
    I0 = 986
    #Intensités brutes en sortie du détecteur
    Is = []
    Is.extend([611,281])
    #Liste qui contiendra les absorbances
    Abs = []
    #Valeurs des concentrations exprimée en mmol/L
    Cs = [2,4]

    #Calcul de l'absorbance
    for I in Is:
        Abs.append(calc_absorbance(I,I0))

    #Calcul des paramètres de la droite affine
    droite = calc_droite(Abs,Cs)

    print('pente : {}, ordonnée à l\'origine : {}'.format(*droite))
```

```
pente : 0.38837114491892477, ordonnée à l'origine : -0.29818289440680945
```

3 Exercice 4

Le code un minimum fonctionnel qui ne corrige pas les erreurs de conception mais est exécutable et donne un résultat presque correct.

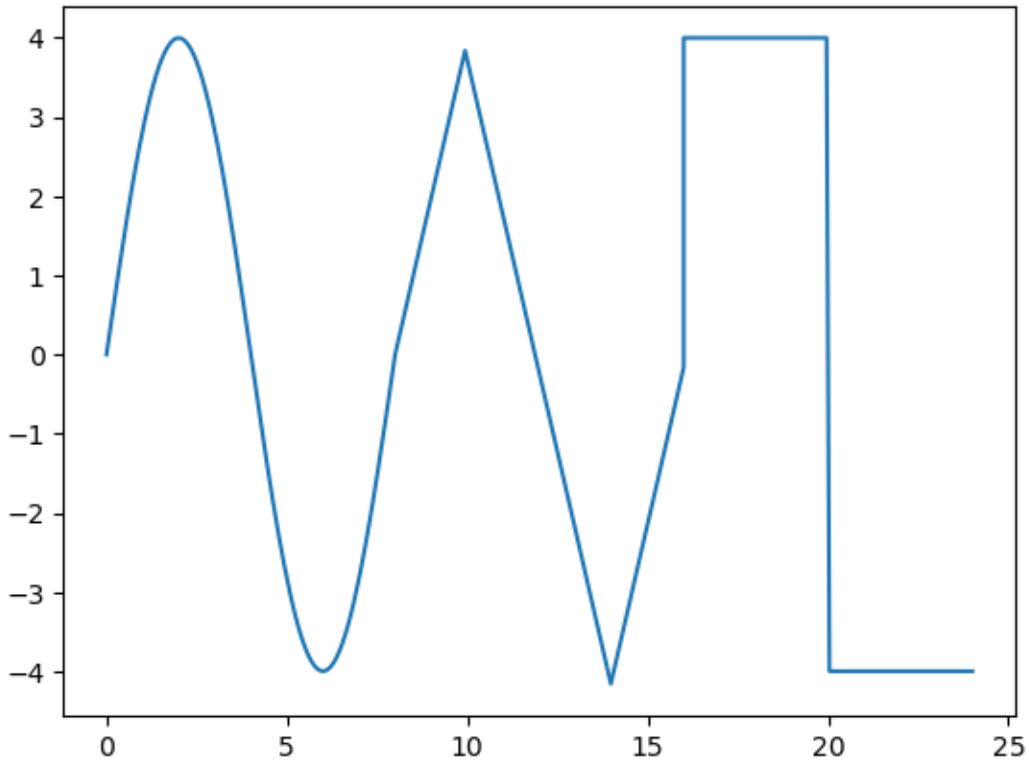
[9]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Création de signaux périodiques

On crée un signal périodique, celui-ci sera ajouté à la suite des listes
`time` et `signal` si jamais ces deux listes sont non vides, sinon on doit␣
 ↪retourner une période.

Le but est ici d'avoir à la suite un signal sinusoïdal suivi d'un signal␣
 ↪triangulaire suivi d'un signal carré.
Le signal final doit être continu (excepté aux discontinuités du signal carré).
"""

#### Bad mauvais import de l'alias, du coup les fonctions dans le namespace␣
 ↪numpy ne fonctionnent pas
#### en général lié à un mauvais copier coller depuis internet
#$$# import numpy
#### Good
import numpy as np #£££


# import de librairie pour avoir un aperçu du résultat,
# PAS de bug dans les deux lignes ci-dessous
import matplotlib as mpl
import matplotlib.pyplot as plt

#$$# def make_sinusoid(amplitude, period=1, steps=100):
def make_sinusoid(amplitude, period=1, steps=100, time = None, signal = None):␣
 ↪#£££
    """
    crée un signal sinusoïdal, retourne deux listes :
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire

    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
```

```
    - time (list) :  liste à laquelle on va ajouter le signal
    """

    #### GOOD : initialisation d'objet mutables au sein de la fonction pour des␣
 ↪arguments optionnels
    if time == None: #£££$
        time = [] #£££$
    if signal == None: #£££$
        signal = [] #£££$

    #### Bad en trop => conflit de nom de variable dans deux namespaces␣
 ↪différents
    #### en déclarant time ici, on créé une variable interne à la fonction, ce␣
 ↪qui fait
    #### que le extend va porter sur la variable interne à la fonction et pas␣
 ↪la variable globale
    #$$# time = list(np.linspace(0,period,steps))

    #création de l'ensemble des abscisses temporelles
    #la fonction linspace va créer steps points régulièrement espacés entre 0␣
 ↪et period
    times = np.linspace(0,period,steps)

    #Calcul de l'amplitude pour chacun des pas de temps
    #### Good
    amplitudes = amplitude * np.sin(times*2*np.pi/period) #£££$
    #### Bad : oubli de la pulsation
    #$$# amplitudes = amplitude * np.sin(times)


    #### Good : fait le shift
    if len(time)>0: #£££$
        times += max(time) #£££$

    time.extend(list(times))
    signal.extend(list(amplitudes))

    #### Bad : pas de valeur retour de la fonction
    return times,amplitudes #£££$


#$$# def make_triangle(amplitude, period=1, steps=100):
def make_triangle(amplitude, period=1, steps=100, time = None, signal = None):␣
 ↪#£££$
    """
    crée un signal triangulaire, retourne deux listes :
```

```
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire

    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
    - time (list) :  liste à laquelle on va ajouter le signal
    """

    if time == None: #£$
        time = [] #£$
    if signal == None: #£$
        signal = [] #£$

    #calcul de la pente associée au signal qui sera l'incrément entre deux pas de temps
    slope = 4*amplitude/steps


    #création de l'ensemble des abscisses temporelles
    #la fonction linspace va créer steps points régulièrement espacés entre 0 et period
    times = np.linspace(0,period,steps)
    #amplitude du signal correspondant à chaque abscisse
    amplitudes = []

    #indice correspondant au pas de temps
    current_step = 1 #£$
    #$$# current_step = 0
    amplitudes.append(0) #£$
    current_amplitude = 0
    """
    on fait un signal triangulaire découpé en 4 morceaux :
    - lors du premier quart de période on incrémente d'une valeur égale à la pente
    - sur les deux quarts de période suivants on soustrait d'une valeur égale à la pente
    - lors du dernier quart de période on incrémente d'une valeur égale à la pente
    """
    while current_step<steps:
        if current_step < steps//4:
            current_amplitude += slope
            amplitudes.append(current_amplitude)
```

```
        elif current_step < 3*steps//4:
            current_amplitude -= slope
            amplitudes.append(current_amplitude)
        else:
            current_amplitude += slope
            amplitudes.append(current_amplitude)
        #### BAD :  non bouclage d'un while => boucle infinie
        current_step+=1 #£££

    if len(time)>0:#£££
        times += max(time)#£££

    time.extend(times)
    signal.extend(list(amplitudes))

    #### Bad : pas de valeur retour de la fonction
    return times,amplitudes #£££


#$$$ def make_square(amplitude, period=1, steps=100):
def make_square(amplitude, period=1, steps=100, time = None, signal = None):␣
 ↪#£££
    """
    crée un signal rectangulaire, retourne deux listes :
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire

    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
    - time (list) :  liste à laquelle on va ajouter le signal
    """
    #création de l'ensemble des abscisses temporelles
    times = np.linspace(0,period,steps)
    amplitudes = []
    for step in range(steps):
        if step < steps//2:
            amplitudes.append(amplitude)
        else:
            amplitudes.append(-amplitude)
    if len(time)>0:#£££
        times += max(time)#£££

    time.extend(list(times))
```

```
    signal.extend(list(amplitudes))

    return times,amplitudes #£££$



if __name__ == "__main__":
    time = []
    signal = []
    pas = 100


    times,amplitudes = make_sinusoid(4, 8,pas, time = time, signal = signal)␣
 ↪#£££$
    #$$# times,amplitudes = make_sinusoid(4, 8,pas)
    times,amplitudes = make_triangle(4, 8,pas, time = time, signal = signal)␣
 ↪#£££$
    #$$# times,amplitudes = make_triangle(4, 8,pas)
    times,amplitudes = make_square(4, 8,pas, time = time, signal = signal) #£££$
    #$$# times,amplitudes = make_square(4, 8,pas)


    #Affichage du signal final, pas de bug dans les deux lignes ci-dessous
    plt.plot(time,signal)
    plt.show()
```



Ici, il faut mieux : * commenter les deux fonctions triangles et rectangle pour commencer à déboguer la fonction la plus simple * ensuite décommenter la fonction rectangle * ensuite décommenter la fonction triangle qui est la plus complexe

Liste des bugs : ##### bugs qui rendent le code non exécutable * import de numpy sans faire l'aliasing sur "np" : erreur de nommage

bugs `triviaux` faciles à corriger

- il manque une ligne qui incrémente le compteur dans le while : la boucle est infinie.

mauvaises pratiques

- la fonction n'a pas de valeur de retour, ce qui rend plus difficile le contrôle et le débogage
- pour les fonctions : on ne passe pas explicitement time et signal en paramètre de la fonction alors que ce sont des objets mutables : les fonctions vont les modifier sans signe extérieur ! Génère un bug dans la fonction pour la sinusoïde
- le code pour décaler les abscisses en temps est répété trois fois, il faut mieux faire une fonction dédiée pour pouvoir le déboguer une unique fois et pouvoir le faire évoluer

bugs `conceptuels` liés au fait de faire des erreurs de réflexion sur les opérations à mener

- pour la fonction rectangle : la fonction est discontinue donc il faut gérer les cas particulier à la demi période et aux bords

- il manque un décalage de temps pour que les abscisses temporelles soit décalées
- pour la fonction sinusoidales il manque le facteur 2*np.pi/period

bug difficile à corriger

- pour la fonction triangle, le résultat n'est correct que si le nombre de pas est un multiple de 5, il faut corriger en calculant explicitement la valeur de la fonction plutôt que de fonctionner par incrément. En règle général fonctionner par incrément sur le résultat précédent est moins bien que de faire un calcul direct lorsque cela est possible : cela peut propager des erreurs.

bug plus technique et difficilement visible

- Pour les trois fonctions : il ne faut pas créer de liste vide lors de l'initialisation de la fonction car sinon on a un objet mutable qui sera toujours le même lors de l'exécution de la fonction

[5]:
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Création de signaux périodiques



On crée un signal périodique, celui-ci sera ajouté à la suite des listes
`time` et `signal` si jamais ces deux listes sont non vides, sinon on doit␣
 ↪retourner une période.

Le but est ici d'avoir à la suite un signal triangulaire suivi d'un signal␣
 ↪carré suivi d'une sinusoïde.
Le signal final doit être continu (excepté aux dsicontinuité du signal carré).
"""

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import math

def shift_time(liste_orig,times):
    """
    décale la liste times pour ajouter l'élément le plus grand de liste_orig
    """
    if len(list(liste_orig))>0:
        times_shifted = times + max(liste_orig)
        return times_shifted
    else:
        return times

def triangle(time,period,amplitude):
```

```
    """
    Calcule l'amplitude d'un signal triangulaire au temps `time`
    pour un signal triangulaire de période et d'amplitude donnée"""
    if time < period/4:
        return amplitude*time/(period/4)
    if time < 3/4*period:
        return amplitude - 2*amplitude*(time-period/4)/(period/2)
    else:
        return -amplitude+amplitude*(time-3/4*period)/(period/4)

def make_triangle(amplitude, period=1, steps=100, time = None, signal = None):
    """
    crée un signal triangulaire, retourne deux listes :
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire


    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
    - time (list) :  liste à laquelle on va ajouter le signal
    """
    if time == None:
        time = []
    if signal == None:
        signal = []

    #création de l'ensemble des abscisses temporelles
    times = np.linspace(0,period,steps)
    #amplitude du signal correspondant à chaque abscisse
    amplitudes = []


    """
    #calcul de la pente associée au signal qui sera l'incrément entre deux pas␣
 ↪de temps
    slope = 4*amplitude/(steps-1)
    #indice correspondant au pas de temps
    current_step = 0
    current_amplitude = 0
    on fait un signal triangulaire découpé en 4 morceaux :
    - lors du premier quart de période on fait un signal croissant
    - décroissance sur la demi-période suivante
    - croissance sur le dernier quart de période
```

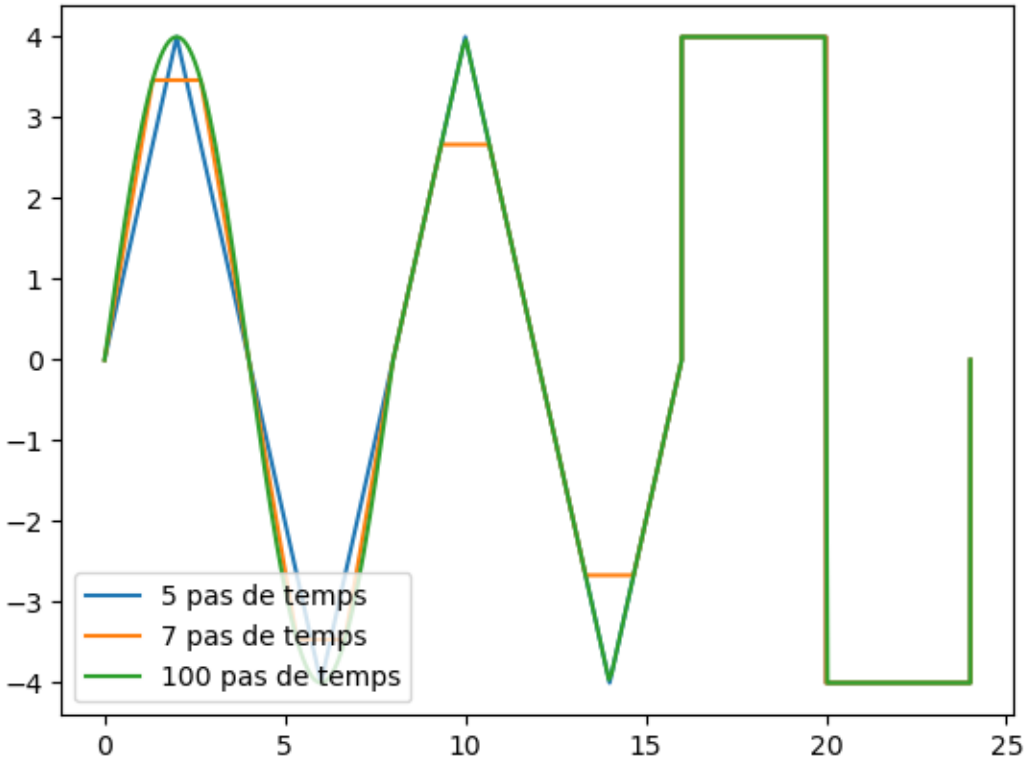
```
    """

    """
    while current_step<steps-1:
        if current_step < steps//4:
            current_amplitude += slope
            amplitudes.append(current_amplitude)
        elif current_step < 3*steps//4:
            current_amplitude -= slope
            amplitudes.append(current_amplitude)
        else:
            current_amplitude += slope
            amplitudes.append(current_amplitude)
        current_step += 1
    """
    for i,t in enumerate(times):
        amplitudes.append(triangle(t,period,amplitude))
    """"correction de l'abscisse temporelle pour ajouter un décalage temporel
    correspondant au maximum de la liste `time` fournie"""
    times = shift_time(time,times)

    time.extend(list(times))
    signal.extend(list(amplitudes))
    return times,amplitudes

def make_square(amplitude, period=1, steps=100, time = None, signal = None):
    """
    crée un signal rectangulaire, retourne deux listes :
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire

    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
    - time (list) :  liste à laquelle on va ajouter le signal
    """
    if time == None:
        time = []
    if signal == None:
        signal = []

    #création de l'ensemble des abscisses temporelles
    if steps % 2 == 1:
        times = np.linspace(0,period/2,steps//2+1)
```

```
        times2 = np.linspace(period/2,period,steps//2+1)
        times = np.concatenate((times,times2,[period]))
    else:
        print('Attention, le nombre de pas de temps devrait être impair')
        times = np.linspace(0,period,steps)
        times = np.concatenate(([0],times,[period]))
    amplitudes = []

    if steps % 2 == 0:
        amplitudes.append(0)
        for step in range(steps):
            if step < steps//2:
                amplitudes.append(amplitude)
            else:
                amplitudes.append(-amplitude)
        amplitudes.append(0)
    else:
        for step in range(steps):
            if step < steps//2:
                amplitudes.append(amplitude)
            elif step == steps//2:
                amplitudes.append(amplitude)
                amplitudes.append(-amplitude)
            else:
                amplitudes.append(-amplitude)
        amplitudes.append(0)

    """correction de l'abscisse temporelle pour ajouter un décalage temporel
    correspondant au maximum de la liste `time` fournie"""
    times = shift_time(time,times)

    time.extend(list(times))
    signal.extend(list(amplitudes))
    return times,amplitudes

def make_sinusoid(amplitude, period=1, steps=100, time = None, signal = None):
    """
    crée un signal sinusoïdal, retourne deux listes :
    - une liste avec les abscisses temporelles,
    - l'autre avec le signal
    ces deux listes sont ajoutées aux listes `time` et `signal`,
    avec un décalage de l'abscisse temporelle pour le temps si nécessaire

    Paramètres :
    - amplitude (float) : amplitude du signal
    - period (float) : période du signal
    - steps (int) : nombre de pas pour l'échantillonage de la fonction
```

```
    - time (list) :  liste à laquelle on va ajouter le signal
    """
    if time == None:
        time = []
    if signal == None:
        signal = []


    #création de l'ensemble des abscisses temporelles
    times = np.linspace(0,period,steps)
    amplitudes = amplitude * np.sin(times*2*np.pi/period)

    """"correction de l'abscisse temporelle pour ajouter un décalage temporel
    correspondant au maximum de la liste `time` fournie"""
    times = shift_time(time,times)

    time.extend(list(times))
    signal.extend(list(amplitudes))
    return times,amplitudes


if __name__ == "__main__":
    #Abscisse temporelle et ordonnée du signal
    time5 = []
    signal5 = []

    times,amplitudes = make_sinusoid(4, 8,5, time = time5, signal = signal5)
    times,amplitudes = make_triangle(4, 8,5, time = time5, signal = signal5)
    times,amplitudes = make_square(4, 8,5, time = time5, signal = signal5)

    time100 = []
    signal100 = []
    times,amplitudes = make_sinusoid(4, 8,100, time = time100, signal =␣
 ↪signal100)
    times,amplitudes = make_triangle(4, 8,100, time = time100, signal =␣
 ↪signal100)
    times,amplitudes = make_square(4, 8,100, time = time100, signal = signal100)

    time7 = []
    signal7 = []
    times,amplitudes = make_sinusoid(4, 8,7, time = time7, signal = signal7)
    times,amplitudes = make_triangle(4, 8,7, time = time7, signal = signal7)
    times,amplitudes = make_square(4, 8,7, time = time7, signal = signal7)

    #Affichage du signal final
    plt.plot(time5,signal5,label='5 pas de temps')
    plt.plot(time7,signal7,label='7 pas de temps')
```

```
    plt.plot(time100,signal100,label='100 pas de temps')
    plt.legend(loc='lower left')
    plt.show()
```

Attention, le nombre de pas de temps devrait être impair



[ ]:

[ ]: