Structure d'un programme

1- Importation des bibliothèques

2 – Définition de toutes les constantes de l'énoncé : T, Ve, V0 et les u associées
ainsi dans la suite du programme : aucune valeur numérique !!! Il faut appeler les constantes par leur nom
Par exemple :
on n'écrit pas : vv=np.random.uniform(9.1,10.1) mais vv=np.random.uniform(V0-u_V0,V0+u_V0)

3- Donner une instruction : calcul de C... une régression linéaire… calcul monté carlo...

- Avec un array, calcul simple

```
C = 5*Cper/(2*V0)*Veq          # génère directement un array avec C
```

- Avec une liste, on fait une boucle

```
C=[ ] # on initialise une liste vide, en dehors de la boucle bien sûr puis on fait la boucle
for i in range(len(Veq)): #len(Veq) correspond au nombre d'éléments dans la liste Veq
    C.append(5*Cper/(2*V0)*Veq[i]) #on ajoute à la liste C le terme Ci calculé à chaque itération
```

4- Demander au programme d'afficher quelque chose : 1 graphe ou une valeur...

- Pour un graphe, il faut le titre du graphe et les titres des axes :

```
plt.plot(t,C,'b*')
plt.title('C = f(t)')
plt.xlabel('t (s)')
plt.ylabel('C (mol/L)')
plt.grid() #parce que c'est plus joli
plt.show()
```

- Pour afficher un résultat : commande print

```
print("l'incertitude type est :", u_LNC)
print("k= {:.0f} s^-1")
print("k={:.2e} s^-1 ".format(p[0])) # permet de contrôler les CS : ici 2 chiffres après la virgule et écriture scientifique
```

```
k= 0.19797036925192596 s^-1
k=1.98e-01 s^-1
```

Sous-programme ou technique à savoir faire :

1- Dichotomie :

```
def dichoto(min,max,ff,precision):
    while (max-min) > precision:
        m=(max+min)/2
    if ff(min)*ff(m) < 0 :
        max=m
    else :
        min=m
    return m
```

2- Méthode d'Euler pour représenter la solution d'une équation différentielle $\frac{d\xi}{dt} = f(\xi)$

L'idée est de construire la courbe solution point par point :
On définit une fonction $d\xi = f(\xi) * dt$ lié à la loi de vitesse (dans le cas d'un ordre 1 : $d\xi = k \times (C_0 - \xi)dt$)
On initialise la valeur de ξ : $\xi_0 = 0$
Ensuite $\xi_1 = \xi_0 + d\xi_0$, puis $\xi_2 = \xi_1 + d\xi_1$ … et $\xi_{i+1} = \xi_i + d\xi_i$
On construit la liste de ξ par itération, donc à l'aide d'une boucle, de même pour la liste de t
Cette méthode n'est valable que si le pas dt est faible
Enfin on trace $\xi = f(t)$

```
def d_ksi(T,ksi,Dt):
    return k(T)*(C0-ksi)*Dt

t_euler=[t0]
ksi_euler=[ksi0]
for i in range(N) :
    t_euler.append(t_euler[i]+Dt)
    ksi_euler.append(ksi_euler[i]+d_ksi(T0,ksi_euler[i],Dt))
```

2

3- Calcul monté-Carlo

L'idée : générer des listes de N valeurs autours des variables fournies avec la ½ étendue proposée cela revient à simuler N expériences

Centrale 2024 :
Par une analyse de sa procédure, l'expérimentateur est amené à considérer les données suivantes :
— précision des solutions fournies par les techniciens : 1 % ;
— indication sur la burette graduée : ±0,03 mL ;
— demi-étendue associée aux volumes équivalents estimée à ±0,05 mL ;
— la variabilité des différentes grandeurs est simulée à l'aide d'une loi uniforme

$$m_{\text{ibuprofène}} = M_{\text{ibuprofène}} \cdot C_2 \cdot (V_{\text{E2}} - V_{\text{E1}})$$

```
#Importation des bibliothèques

import numpy as np                  #pour faire des calculs
import numpy.random as rd           #sous-module de numpy pour générer des nombres aléatoires
import matplotlib.pyplot as plt     #pour tracer divers graphiques

#Données du problème et résultats expérimentaux supposées définis

#Simulation Monte-Carlo

N = 100000                            #nombre de simulations

# ** Partie à compléter par le candidat **

u_m = np.std(m_sim, ddof=1)
print('u_m =',u_m,'g')
```

 # 2- données de l'énoncé (variables)
C2 = 0.100 *# mol/L*
VE1 = 8.7 *# mL*
VE2 = 20.5 *# mL*
pC2= 0.01
DC2 = pC2 ∗ C2 *# mol/L*
tb = 0.03 *# mL*
DVE = 0.05 *# mL*
M = 206 . 27 *# g/ mol*

#3- calcul : Simulation Monte-Carlo
N = 100000 *# nombre de simulations*
C2_MC = C2 + rd.uniform(-DC2 , DC2 , N) ou C2sim = C2 + DC2*rd.uniform(-1, 1 , N)
VE2moinsVE1_MC = VE2 + rd.uniform(-DVE, DVE, N) + rd.uniform(-tb , tb , N) - VE1 + rd.uniform(-DVE, DVE, N) + rd.uniform(-tb , tb , N)
m_sim = M ∗ C2_MC ∗ VE2moinsVE1_MC ∗ 1 E-3 *# en g*
u_m = np.std(m_sim, ddof = 1)
print('u_m = ', u_m, ' g')