TABLEAUX, TRACES DE COURBES, DICHOTOMIE, PROCESSUS ALEATOIRES (Monte Carlo)

I – RAPPELS ET COMPLEMENTS SUR LES TABLEAUX ET LA BIBLIOTHEQUES NUMPY

En sciences, nous sommes très souvent conduits à utiliser des tableaux de données numériques (issues de calculs ou d'expériences) et il est commode de pouvoir « visualiser » ces données sous forme de courbes (l'homme est un animal visuel).
Ainsi nous allons introduire les objets de type tableaux, `array`, afin de pouvoir stocker facilement les données, comme des couples de points, pour pouvoir ensuite tracer des courbes.

L'objet de type `array` peut être considéré comme une variante de l'objet `list` mais avec les différences et caractéristiques suivantes :

✔ Tous les éléments d'un tableau doivent être du même type, de préférence des nombres entiers (`integer`), des réels (`real`) ou des complexes (`complex`) afin d'améliorer l'efficacité du stockage des données et du calcul numérique.
✔ Le nombre d'éléments du tableau doit être connu avant de créer le tableau.
✔ Les tableaux ne font pas partie de la distribution standard de Python. Il faut donc faire appel à une bibliothèque spécifique (package en anglais) appelée **`Numerical Python`** souvent abrégée par **`numpy`**.
✔ Avec numpy, une large gamme d'opérations mathématiques est directement disponible sur les tableaux, ainsi il n'est plus nécessaire de faire des boucles sur les éléments du tableau, c'est « automatique » comme nous allons le voir sur des exemples. On parle de ***vectorisation***.
✔ Un tableau avec un seul indice est appelé un vecteur (`vector`). Il est bien sûr possible de manipuler des tableaux à plusieurs indices.

Voici quelques commandes importantes sur les tableaux :

```
>>> import numpy as np
```
Permet d'importer la bibliothèque numpy. np est ***l'alias*** standard pour préfixer les commandes dans numpy.

```
>>> a=np.array(r)
```
Convertit la liste `r` en un tableau (assigné à la variable `a`).

```
>>> a=np.zeros(n)
```
Crée un tableau de n éléments tous égaux à zéro.

```
>>> a=np.zeros_like(c)
```
Crée un tableau d'éléments tous égaux à zéro, de type identique à celui du tableau c (entier, réel etc…) et de même longueur que ce dernier.

```
>>> a=np.linspace(p,q,n)
```
Crée un tableau de n éléments uniformément distribués sur l'intervalle $[p,q]$. Un élément i du tableau est accessible par la commande $a[i]$ (comme pour les listes).

```
>>> scalaire=np.vdot(u,v)
```
Calcule le produit scalaire entre deux vecteurs. `scalaire` est un `real`. En anglais, le produit scalaire se nomme « dot product ».

`>>> vectoriel=np.cross(u,v)`
Calcule le produit vectoriel entre deux vecteurs. `vectoriel` sera un tableau (vecteur) de taille 3 si u et v sont de taille 3 (vecteurs usuels de l'espace Euclidien). En anglais, le produit vectoriel se nomme « cross product ».

Voici un exemple d'application de ***vectorisation*** avec numpy.

```
import numpy as np

def f(t):
    return t**2*np.exp(-t**2)

t = np.linspace(0, 3, 51)
y = np.zeros(len(t))
y = f(t)

print(t)
print(y)
```

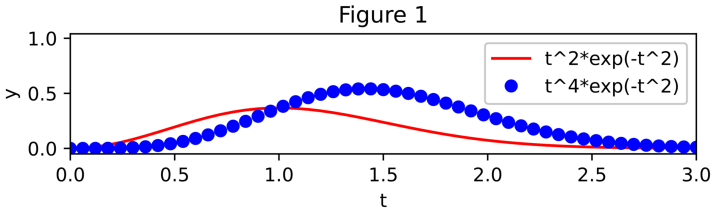
```
[0.   0.06 0.12 0.18 0.24 0.3  0.36 0.42 0.48 0.54 0.6  0.66 0.72 0.78
 0.84 0.9  0.96 1.02 1.08 1.14 1.2  1.26 1.32 1.38 1.44 1.5  1.56 1.62
 1.68 1.74 1.8  1.86 1.92 1.98 2.04 2.1  2.16 2.22 2.28 2.34 2.4  2.46
 2.52 2.58 2.64 2.7  2.76 2.82 2.88 2.94 3.  ]
[0.         0.00358706 0.01419413 0.03136706 0.05437598 0.08225381
 0.1138467  0.14787305 0.18298733 0.21784483 0.25116348 0.28177937
 0.30869297 0.33110401 0.34843389 0.36033503 0.36668798 0.36758719
 0.36331759 0.35432402 0.34117597 0.32453024 0.30509364 0.28358794
 0.26071856 0.23714826 0.21347654 0.190225   0.16782854 0.14663192
 0.12689102 0.10877791 0.09238879 0.07775392 0.06484857 0.05360434
 0.04392008 0.03567213 0.02872323 0.02293025 0.0181504  0.01424603
 0.01108811 0.00855858 0.00655163 0.00497417 0.00374573 0.00279779
 0.00207287 0.00152345 0.00111069]
```

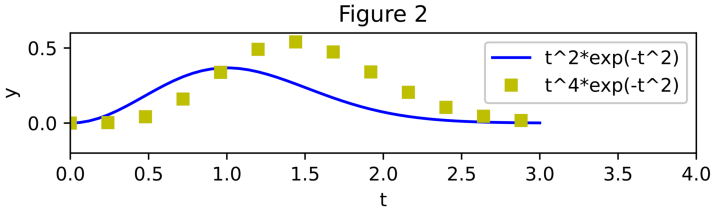
y et t sont des tableaux. Attention, il ne faut pas utiliser la fonction exp de la bibliothèque math mais il faut appeler `np.exp` c'est-à-dire la fonction exp de numpy.

II – TRACES DE COURBES, BIBLIOTHEQUES MATPLOTLIB.PYLAB

Avec l'utilisation des tableaux, il est assez facile de tracer des courbes sous Python. Afin de voir les commandes essentielles, voici un exemple:

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

def f1(t):
    return t**2*np.exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = np.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plt.subplot(3, 1, 1)
plt.plot(t, y1, 'r-', t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Figure 1')

plt.subplot(3, 1, 3)
t3 = t[::4]
y3 = f2(t3)
plt.plot(t, y1, 'b-', t3, y3, 'ys')
plt.xlabel('t')
plt.ylabel('y')
plt.axis([0, 4, -0.2, 0.6])
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Figure 2')

plt.savefig('tmp2.eps')
plt.show()
```

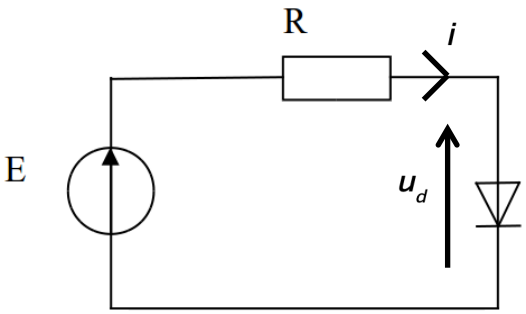




`plt` est ***l'alias*** standard pour préfixer les commande `matplotlib`.

III – APPLICATIONS SUR UN CIRCUIT ELECTRIQUE

1) Tracé des caractéristiques



On reprend le circuit suivant étudié expérimentalement en TP avec :

✔ $E = 2$ V et $R = 500\ \Omega$.

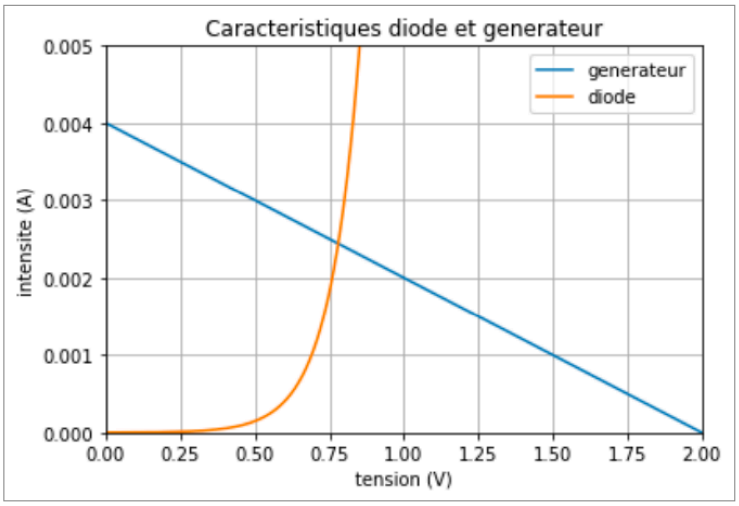
✔ caractéristique de la diode :

$i(u_d) = 10^{-6}\left(e^{\frac{u_d}{0,1}} - 1\right)$, i en A et u_d en V.

✔ le point de fonctionnement doit vérifier: $\underbrace{E - Ri}_{u_g} = u_d$.

```
from math import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import bisect
```

```
#caracteristique de la diode
def diode(i):
    ud=0.1*np.log((i/10**-6)+1)
    return ud

#caracteristique du generateur
def generateur(i):
    R=500
    E=2
    ug=E-R*i
    return ug

nbr_points=1000
i=np.linspace(0.0,5*10**-3,nbr_points)

u_generateur=np.zeros(len(i))
u_diode=np.zeros(len(i))

u_diode=diode(i)
u_generateur=generateur(i)

plt.plot(u_generateur,i)
plt.plot(u_diode,i)
plt.grid()
plt.axis([0, 2, min(i), max(i)])
plt.title('Caracteristiques diode et generateur')
plt.xlabel('tension (V)')
plt.ylabel('intensite (A)')
plt.legend(['generateur','diode'])
plt.show()
```



2) Calcul du point de fonctionnement par dichotomie et par utilisation de la fonction `bisect`

On cherche à déterminer le courant de fonctionnement du circuit précédent qui, graphiquement, est proche de 25 mA. Pour cela, il faut résoudre $u_g(i) - u_d(i) = 0$. Dans la plupart des situations, on ne peut résoudre de façon analytique une telle équation. On peut alors procéder de façon approchée par un calcul numérique. Pour cela, nous allons utiliser l'algorithme bien connu de ***dichotomie*** (cf. cours d'informatique).

```
def difference(i):
    return generateur(i)-diode(i)


#Algorithme de dichotomie
#-----------------------------------------
#On teste la condition f(a)*f(b)<=0
#si non danger d'une boucle infinie,
#"assert" renvoie un message d'erreur.
#On calcule le milieu du segment
#puis si le zéro est à gauche de ce milieu
#alors: déplacer la borne de droite du segment
#sinon: c'est que le zéro est à droite du milieu
#donc: déplacer la borne de gauche à la place
#si abs(a-b>=eps, on peut renvoyer a, b ou m)

def dichotomie(f,a,b,eps):
    assert f(a)*f(b)<=0
    while abs(b-a)>eps:
        m=(a+b)/2
        if f(a)*f(m)<=0:
           b=m
        else:
            a=m
    return m
```

```
Point de fonctionnement (fonction dichotomie) : 0.002431640625 A
Point de fonctionnement (fonction bisect) : 0.0024399698560591786 A
```

On constate que la méthode par dichotomie donne un résultat proche de celui de la méthode `bisect`. Si la fonction est monotone, la solution obtenue par dichotomie est unique. Cependant, on ne sait à priori rien du nombre de solutions. L'algorithme permettra alors d'approcher l'une des solutions, mais on ne contrôle pas laquelle.

IV – SIMULATION D'UN PROCESSUS ALEATOIRE, METHODE DE « MONTE CARLO »

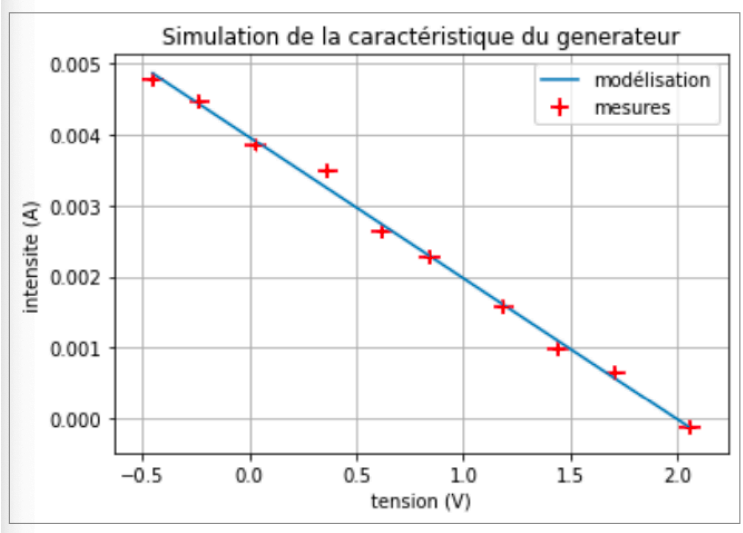
1) Simulation d'une mesure et tracé de barres d'erreurs

On souhaite ***simuler numériquement*** l'expérience du tracé de la caractéristique du générateur. L'erreur commise sur chaque mesure peut alors se modéliser par une variable aléatoire. On utilise la plus souvent ***la loi normale centrée (de moyenne nulle) et d'écart-type égal à l'incertitude***. Pour simuler un jeu de mesures possibles, il s'agit donc d'ajouter cette erreur aux valeurs théoriques.

✔ On prendra pour incertitude $ux = 0{,}1$ V pour les mesures de la tension et $uy = 0{,}1$ mA pour la mesure de l'intensité. De plus dans `matplotlib`, on trouve une fonction `errorbar` qui traces des barres d'erreur.

✔ Dans le module numpy, on trouve une fonction de régression polynômiale `np.polyfit`. En se limitant au degré 1, ***on peut donc modéliser un ensemble de valeurs par une loi affine***. Pour s'assurer que la régression linéaire est valide, il convient de superposer le modèle aux incertitudes.

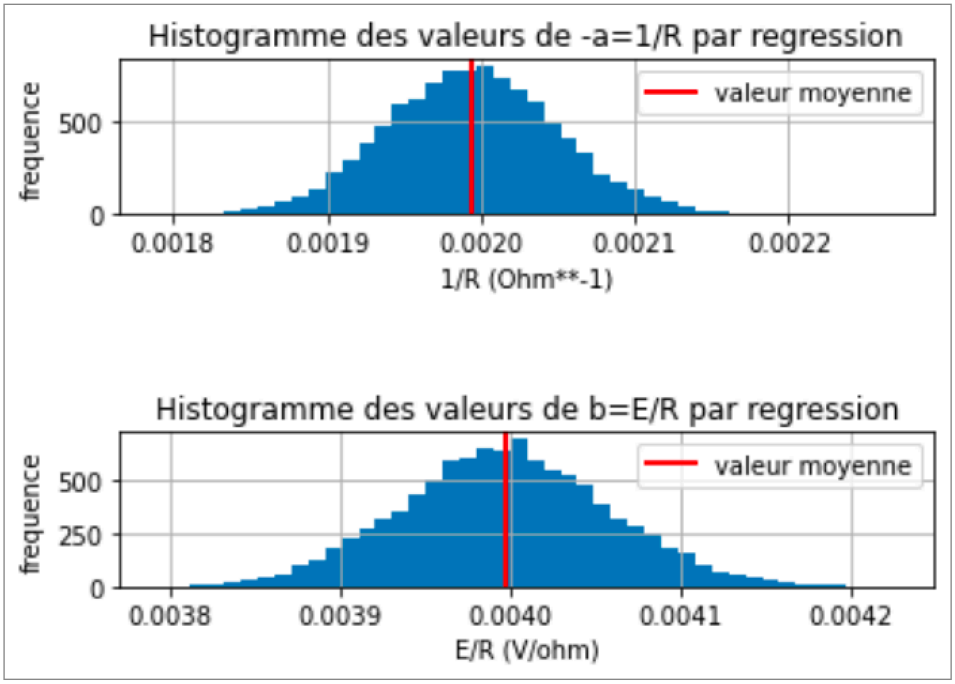
```
from math import *
import numpy as np
import matplotlib.pyplot as plt

def generateur(i,E,R):
    ug=E-R*i
    return ug

def simuler_mesure (n,imin,imax,f,ux,uy):
    i=np.linspace(imin,imax,n)
    u=np.zeros(n)
    for j in range(n):
        u[j]=generateur(i[j],E,R)
    # On ajoute les incertitudes (distribution aleatoire normale) sur i
    y=i+np.random.normal(0,uy,n)
    # On ajoute les incertitudes (distribution aleatoire normale) sur u
    x=u+np.random.normal(0,ux,n)
    return(x,y)

ux,uy=0.05,0.0001 # choix des incertitudes
n=10 # nombre de mesures simulees
imax=5*10**-3
imin=0

# parametres du generateur
E=2
R=500

x,y=simuler_mesure(n,imin,imax,generateur,ux,uy)
# regression lineaire avec le fonction polyfit de numpy
a,b=np.polyfit(x,y,1)
print('a=',a,'b=',b)

xmodele=np.array([min(x),max(x)])
ymodele=a*xmodele+b


plt.errorbar(x,y,xerr=ux,yerr=uy,fmt='r+')
plt.plot(xmodele,ymodele)
plt.title('Simulation de la caractéristique du generateur')
plt.xlabel('tension (V)')
plt.ylabel('intensite (A)')
plt.legend(['modélisation','mesures'])
plt.grid()
plt.show()
print('Paramètres du modèle simulé y=a*x+b:')
print('a(=-1/R)=',a,'b(=E/R)=',b,)
print('valeurs théoriques:')
print('-1/R=',-1/R,'E/R=',E/R,)
```




```
Paramètres du modèle simulé y=a*x+b:
a(=-1/R)= -0.0020033434070864717 b(=E/R)= 0.004005574134843877
valeurs théoriques:
-1/R= -0.002 E/R= 0.004
```

2) Incertitude sur les paramètres du modèle : méthode de « Monte Carlo »

Afin de déterminer les incertitudes sur la pente $a\left(=-1/R\right)$ et l'ordonnée à l'origine $b\left(=E/R\right)$ du modèle, on peut simuler un grand nombre d'expériences, donc de valeurs de a et de b et en déterminer les écart-types. On répète un grand nombre de fois (ici 1000) l'expérience de simulation de 10 mesures de la caractéristique du générateur. On aura 1000 valeurs de a et de b. On pourra donc faire une étude statistique. On vérifiera que l'incertitude type décroit comme $1/\sqrt{N}$.



```
from math import *
import numpy as np
import matplotlib.pyplot as plt


def generateur(i,E,R):
    u=E-R*i
    return u


def exploiter_mesures (ux,uy,imax,imin,nb_mesures,nb_experiences):
    ta=np.zeros(nb_experiences)
    tb=np.zeros(nb_experiences)
    # boucle for sur le nombre d'experiences
    for k in range(nb_experiences):
         i=np.linspace(imin,imax,nb_mesures)
         u=np.zeros(nb_mesures)
         # boucle for sur le nombre de points de mesures simulees
         for j in range(nb_mesures):
             u[j]=generateur(i[j],E,R)
             y=i+np.random.normal(0,uy,nb_mesures)
             x=u+np.random.normal(0,ux,nb_mesures)
         ta[k],tb[k]=np.polyfit(x,y,1)
    ecart_type_a,ecart_type_b=np.std(ta,ddof=1),np.std(tb,ddof=1)
    ua=ecart_type_a/sqrt(nb_experiences)
    ub=ecart_type_b/sqrt(nb_experiences)
    ma,mb=np.mean(ta),np.mean(tb)
    return(ma,mb,ua,ub,ta,tb)


ux,uy=0.05,0.0001 # choix des incertitudes
nb_mesures=10 # nombre de mesures simulees
imax=5*10**-3
imin=0


# parametres du generateur
E=2
R=500

#nombre d'experiences
nb_experiences=10000

ma,mb,ua,ub,ta,tb=exploiter_mesures (ux,uy,imax,imin,nb_mesures,nb_experiences)

plt.subplot(3, 1, 1)
plt.grid()
plt.title('Histogramme des valeurs de -a=1/R par regression')
plt.xlabel('1/R (Ohm**-1)')
plt.ylabel('frequence')
plt.hist(-ta, bins='rice')
plt.axvline(-ma, color='r', linestyle='solid', linewidth=2)
plt.legend(['valeur moyenne',])

plt.subplot(3, 1, 3)
plt.grid()
plt.title('Histogramme des valeurs de b=E/R par regression')
plt.xlabel('E/R (V/ohm)')
plt.ylabel('frequence')
plt.hist(tb, bins='rice')
plt.axvline(mb, color='r', linestyle='solid', linewidth=2)
plt.legend(['valeur moyenne',])

plt.show

moyenne_a=np.mean(ta)
ecart_type_a=np.std(ta,ddof=1)
incertitude_type_a=ecart_type_a/(nb_experiences**(1/2))

moyenne_b=np.mean(tb)
ecart_type_b=np.std(tb,ddof=1)
incertitude_type_b=ecart_type_b/(nb_experiences**(1/2))

print('moyenne de a=',moyenne_a,)
print('incertitude type sur a=',incertitude_type_a,)

print('moyenne de b=',moyenne_b,)
print('incertitude type sur b=',incertitude_type_b,)
```

```
moyenne de a= -0.0019944505493780748
incertitude type sur a= 5.613726230184231e-07
moyenne de b= 0.003996621750032682
incertitude type sur b= 6.100113789786018e-07
```