Chapitre 3 : Fonctions, courbes et dichotomie

1. Fonctions

Dans toute cette partie, on importera la bibliothèque numpy :

```
import numpy as np
```

En Python, certaines fonctions sont déjà définies dans le noyau, ou dans les modules additionnels : exp, log, sqrt, …
On peut également créer de nouvelles fonctions.

Syntaxe d'une fonction f :

```
def nomfonction(arguments):
    ...
    ...   Calcul de y
    return y
```

Remarques :
_ une fonction sert à **calculer une valeur** (celle de y), elle ne contient donc **ni print, ni input, ni plot, …**
_ dans les cas simples, on peut trouver aussi :

```
def nomfonction(arguments):
    return expression
```

_ s'il y a plusieurs arguments, ils sont séparés par une virgule. Attention à l'ordre lors de l'appel de la fonction.
_ les arguments d'entrée et y sont des variables locales, qui ne sont définies que dans la fonction. (d'ailleurs, on peut choisir une autre variable à la place de y, bien sûr).
_ appel de la fonction : il suffit d'écrire nomfonction(valeur), où valeur peut être un réel, une variable ou une expression arithmétique qui sera calculée.
_ Attention : nomfonction(valeur) est un nombre (ou une liste, un tableau…), mais pas une instruction !
On peut donc l'afficher, le stocker dans une variable, l'utiliser dans un calcul…
(Penser à np.sqrt(2), np.exp(-4), …)
_ une fonction ne produit un résultat que quand elle est appelée !

Exemples :
1) Ecrire une fonction f qui à un réel x associe $f(x) = e^x - 3x$

```
def f(x):
    return np.exp(x)-3*x
```

2) Ecrire une fonction somme qui à un entier n associe le réel $S_n = \sum_{k=1}^{n} \frac{1}{k}$.

```
def somme(n):
    s=0
    for k in range(1,n+1):
        s=s+1/k
    return s
```

Rappels :
_ Si a, b, h sont des réels, **np.arange(a, b, h)** est le tableau [a, a + h, … a + nh], avec a + nh qui est le dernier strictement inférieur à b.

Si a, b sont des réels et n un entier, **np.linspace(a, b, n+1)** est le tableau [a, a + h, …, b], avec $h = \frac{b-a}{n}$

C'est un tableau qui contient n+1 valeurs réparties régulièrement entre a et b (on "divise" l'intervalle en n intervalles de même longueur)

Evaluation multiple :
Soit X un tableau. Pour déterminer le tableau de toutes les images par une fonction f, deux possibilités :
_ avec les fonctions usuelles :
np.exp(X) np.log(X) np.sqrt(X) X**n … donnent le tableau qui contient toutes les images
_ de manière générale : **Y = [f(t) for t in X]**

Ex :
Soit $f \begin{cases} \mathbb{R} \longrightarrow \mathbb{R} \\ x \longmapsto \frac{x^2}{2} + e^x \end{cases}$
Déterminer le tableau de valeurs [f(0), f(0.1), f(0.2), …, f(1)] (de deux manières différentes)

```
X=np.linspace(0,1,11)
Y=X**2/2+np.exp(X)
```

```
def f(x):
    return x**2/2+np.exp(x)
X=np.linspace(0,1,11)
Y=[f(t) for t in X]
```

2. Graphiques en deux dimensions

Dans toute cette partie, on utilisera la bibliothèque matplotlib.pyplot de la manière suivante :

import matplotlib.pyplot as plt

Un graphique 2D en Python est un ensemble de points reliés entre eux par une ligne brisée.

Pour afficher le graphique, il ne faut oublier de terminer par l'instruction :

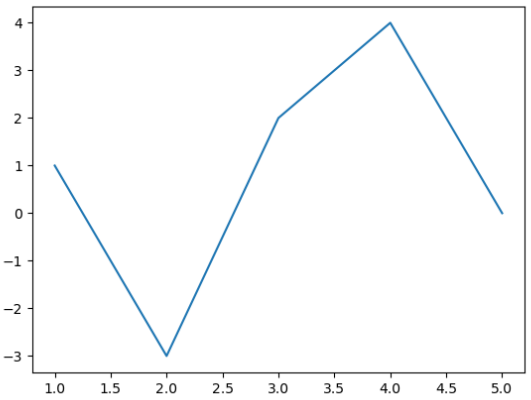
plt.show()

2.1 Ligne brisée définie par n points

Si $X = [x_1, \ldots, x_n]$ et $Y = [y_1, \ldots, y_n]$: **plt.plot(X,Y)** trace la ligne brisée qui joint les points de coordonnées $(x_1, y_1), \ldots, (x_n, y_n)$

Exemple :
Soit A(1, 1) B(2,-3) C(3,2) D(4,4) E(5,0)
Tracer la ligne brisée qui relie les points A, B, C, D, E

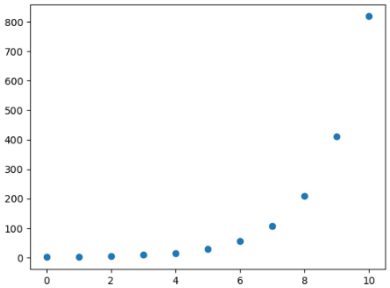
```
X=[1,2,3,4,5]
Y=[1,-3,2,4,0]
plt.plot(X,Y)
plt.show()
```



Remarques :
_ on peut ajouter un titre : **plt.title(' ')**
_ on peut choisir le cadre : **plt.xlim(a, b) plt.ylim(a, b)**
_ on peut placer les points, sans les relier par une ligne brisée, par : **plt.scatter(X, Y)** ou **plt.plot(X, Y,'*')**
(On parle alors de nuage de points)

Exemple :
Soit (u_n) la suite définie par : $\begin{cases} u_0 = 2 \\ \forall\ n \in \mathbb{N},\ u_{n+1} = 2u_n - \ln(u_n) \end{cases}$
Représenter les valeurs de $(u_0, \ldots, u_{10})$.

```
u=2 ;U=[2]
for n in range(1,11):
   u=2*u-np.log(u)
   U.append(u)
N=range(0,11)
plt.plot(N,U,'*')
plt.show
```



2.2 Courbe d'une fonction

Pour tracer la courbe d'une fonction sur un intervalle, il faut choisir une subdivision assez fine de l'intervalle.

Ex : On veut découper l'intervalle [0;2] en 200 subdivisions de longueur 0.01 :

```
X=np.linspace(0,2,201)
```

Pour représenter la courbe d'une fonction f sur un intervalle, Python affiche la ligne brisée correspondant aux images des points de la subdivision. Plus la subdivision est fine, plus la ligne brisée peut être assimilée à la courbe de f.

A retenir :

```
def f(x):          #définition de la fonction
   return …
X = np.linspace(a, b, n)   #subdivision  (ou à l'aide de np.arange)
Y=[f(t) for t in X]    #images de la subdivision
plt.plot(X,Y)
plt.show()
```

Ex : Afficher la courbe de la fonction $f(x) = x^2$ sur [-1;1] (pas : 0.01)

```
def f(x): #définition de la fonction
        return x**2
X = np.linspace(-1, 1, 201) #subdivision
Y=[f(t) for t in X] #images de la subdivision
plt.plot(X,Y)
plt.show()
```

3. Résolution d'équation par dichotomie

Cadre :
Soit f une fonction continue et strictement monotone sur un intervalle [a ; b]. Soit $\lambda \in \mathbb{R}$, fixé.
On suppose qu'on a montré (par le théorème de la bijection) que l'équation $f(x) = \lambda$ admet une unique solution α sur [a ;b].
Soit $\varepsilon > 0$. On cherche à déterminer une valeur approchée de α à ε près.

Remarque :
Soit $x \in$ [a ;b].
Si f est croissante sur [a ;b] : $\begin{cases} \text{Si } f(x) > \lambda \text{ alors : } x > \alpha \\ \text{Si } f(x) < \lambda \text{ alors : } x < \alpha \end{cases}$
Si f est croissante sur [a ;b] : $\begin{cases} \text{Si } f(x) > \lambda \text{ alors : } x < \alpha \\ \text{Si } f(x) < \lambda \text{ alors : } x > \alpha \end{cases}$

Principe :
On construit deux suites a_n et b_n telles que $\forall\, n \in \mathbb{N}$, $\alpha \in [a_n ; b_n]$.

Méthode :
_ Initialisations : $a_0 = a$ et $b_0 = b$
_ Récurrence : Quand a_n et b_n sont calculés :
on calcule le milieu $c_n = \dfrac{a_n + b_n}{2}$ de l'intervalle $[a_n ; b_n]$.
(Si f est croissante) Si $f(c_n) > \lambda$, alors on pose $\begin{cases} a_{n+1} = \\ b_{n+1} = \end{cases}$
sinon on pose $\begin{cases} a_{n+1} = \\ b_{n+1} = \end{cases}$
(inverser si f est décroissante)
Ainsi $\alpha \in [a_{n+1}, b_{n+1}]$.
_ On s'arrête quand $b_n - a_n \leq \varepsilon$.
a_n et b_n sont alors des valeurs approchées de α à ε près.

Remarques :
_ $b_n - a_n$ est une suite géométrique de raison $\dfrac{1}{2}$, donc $\forall\, n \in \mathbb{N}$, $b_n - a_n = \dfrac{b_0 - a_0}{2^n}$
_ $a_{n+1} = a_n$ ou $b_{n+1} = b_n$ n'a pas besoin de s'écrire en informatique, puisque la valeur reste inchangée.

Algorithme à retenir :

```
a=... ; b=….;     # mettre les bornes de l'intervalle
while b – a > epsilon :
      c = (a + b)/2;
      if f(c) > lambda :
            b=c      # si f est croissante, inverser si f décroissante
      else :
            a=c
print('Une valeur approchée de alpha est ',a);  # ou b à la place de a
```

Exemple
Soit f(x) = x.ln(x) sur]0 ; + ∞[. On peut montrer que f est croissante sur [1 ;2], et que l'équation f(x) = 1 admet une unique solution α sur [1 ;2].
Ecrire un programme qui détermine une valeur approchée de α à 10^{-4} près.

```
a=1 ; b=2;
while b-a > 10**(-4) :
        c=(a + b)/2;
        if c*np.log(c) > 1 :
                b=c
        else :
                a=c
print('Une valeur approchée de alpha est ',a)
```

Remarque :
Cas particulier de l'équation f(x) = 0 :
On peut regrouper les cas où f est croissante et où f est décroissante de la manière suivante :
$\alpha \in [a_n; c_n]$ si et seulement si f change de signe entre a_n et c_n, c'est-à-dire si et seulement si $f(a_n) \times f(c_n) \leq 0$.
On peut donc écrire également :

```
if f(a) × f(c) < 0  :                  # Valable pour f croissante ou décroissante
        b=c
else :
        a = c;
```