Outils numériques

1 Tableaux numpy

Le module numpy permet de créer et de manipuler facilement des tableaux, également appelés arrays, particulièrement adaptés aux données numériques. Contrairement à une liste, la taille d'un tableau est fixe (on ne peut pas ajouter d'élément) et tous les éléments sont du même type.

```
import numpy as np
```

1.1 Création d'un tableau à 1D

`x=np.array([x1,x2,...])` crée un tableau numpy à 1 dimension à partir d'une liste de valeur.
`np.zeros(N)` crée un tableaux de N flottants égaux à 0.
`np.linspace(a,b,N)` crée un tableaux contenant une subdivision de l'intervalle $[a, b]$ en N valeurs équidistantes (a et b inclus).

1.2 Opérations sur les tableaux

Les opérateurs arithmétiques +, -, *, / permettent de réaliser des opérations termes à termes sur un tableau. Par exemple,

— `x*1e-3` multiplie tous les éléments du tableau `x` par 10^{-3}

— `x+y` additionne terme à terme les éléments des tableaux `x` et `y` de mêmes dimensions.

De plus, toutes les fonctions mathématiques usuelles sont disponibles dans le module numpy et s'appliquent à l'ensemble des termes du tableau passé en argument.
Exemples : `np.sqrt`, `np.log` (logarithme népérien), `np.log10` (logarithme décimal), `np.arctan`

2 Outils graphiques

On utilise le module matplotlib.pyplot.

```
import matplotlib.pyplot as plt
```

2.1 Représentation graphique d'un nuage de point

Pour tracer le graphe de y en fonction de x :

```
x=[x1,x2,...]
y=[y1,y2,...]
plt.close() # ferme une éventuelle fenêtre graphique précédemment ouverte
plt.plot(x,y)
plt.show() # ouvre la fenêtre graphique
```

En option de la fonction `plt.plot`, on peut préciser la couleur ('k' pour black, 'b' pour blue, 'r' pour red...), le style de points (+, x, .) et/ou le style de ligne (-, :, - -, -.).

```
plt.plot(x,y,'k+') # croix noires
plt.plot(x,y,'g:') # pointillés verts
```

Pour légender les axes :

```
plt.xlabel('nom axe x')
plt.ylabel('nom axe y')
```

Pour fixer l'étendue des axes :

```
plt.xlim(xmin,xmax)
plt.ylim(ymin,ymax)
```

Pour légender les courbes, il faut utiliser l'option label de la fonction `plt.plot` :

```
plt.plot(x,y,label='courbe y(x)')
plt.plot(x,z,label='courbe z(x)')
plt.legend() # affiche la légende
```

2.2 Représentation graphique d'une fonction

Pour tracer le graphe de $f : t \mapsto e^{-2t}\cos(4t+1)$ sur l'intervalle $[a,b]$:

```
def f(x):
    return np.exp(-2*x)*np.cos(4*x+1)
t=np.linspace(a,b,1000)
plt.plot(t,f(t))
```

2.3 Régression linéaire

La fonction `np.polyfit(x,y,n)` permet de modéliser un nuage de point par un polynôme de degré n et renvoie les coefficients du polynôme dans un tableau.

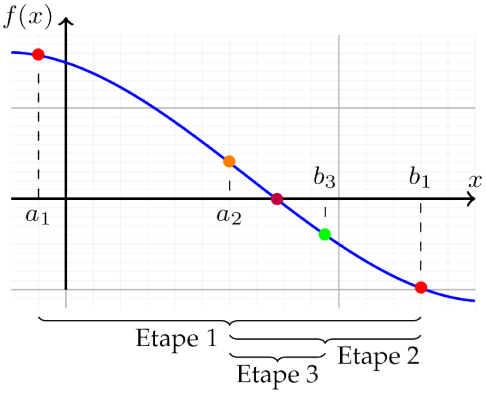
```
x=np.array([x1,x2,...])
y=np.array([y1,y2,...])
a,b=np.polyfit(x,y,1) # a : coefficient directeur, b : ordonnée à l'origine
plt.plot(x,y,'+') # tracé des points
plt.plot(x,a*x+b) # tracé de la droite de régression (en trait plein par défaut)
```

3 Résolution d'une équation algébrique par dichotomie

On cherche une solution x de l'équation $f(x)=0$ (avec f continue) dans un intervalle $[a,b]$ (avec $f(a)f(b)<0$).

1. On calcule le milieu de l'intervalle $[a,b]$:
$$m = \frac{a+b}{2}$$
2. Il y a deux cas possibles.
   — Si $f(a)$ et $f(m)$ sont de signes opposés, alors x se trouve à gauche de m, c'est-à-dire $x \in [a,m]$. On réaffecte alors $b \leftarrow m$.
   — Sinon, x se trouve à droite de m, c'est-à-dire $x \in [m,b]$. On réaffecte alors $a \leftarrow m$.
3. On recommence à l'étape 1 avec le nouvel intervalle $[a,b]$.
4. On arrête la recherche lorsque $b-a<\varepsilon$ où ε est la précision souhaitée.

```
def dicho(f:function,a:float,b:float,eps:float)->float:
    "renvoie la solution de f(x)=0 sur [a,b] à eps près"
    while b-a>eps:
        m=(a+b)/2
        if f(a)*f(m)<=0:
            b=m
        else:
            a=m
    return m
```



On peut également utiliser la fonction `bisect(f,a,b)` du module scipy.optimize.

4 Dérivation - Intégration

4.1 Calcul approché d'une dérivée

Si on dispose des valeurs (y_i) d'une fonction $y(t)$ aux points (t_i) équidistants, on peut approcher la dérivée en un point par :

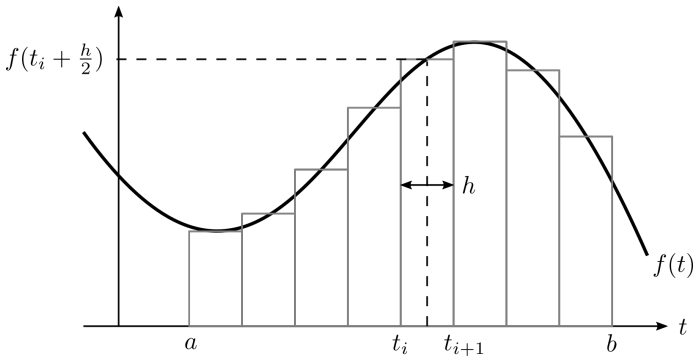
$$y'(t_i) = \frac{\mathrm{d}y}{\mathrm{d}t}(t_i) \simeq \frac{y_{i+1}-y_{i-1}}{t_{i+1}-t_{i-1}}$$

```
t=[t0,t1,...]
y=[y0,y1,...]
dydt=[(y[i+1]-y[i-1])/(t[i+1]-t[i-1]) for i in range(1,len(t)-1)]
# Attention la liste dydt contient 2 valeurs en moins que les listes t et y
```

4.2 Calcul approché d'une intégrale par la méthode des rectangles

On souhaite calculer l'intégrale $\int_a^b f(t)\mathrm{d}t$.
On pose $(t_i)_{i\in[\![0,N]\!]}$ une subdivision de l'intervalle $[a,b]$ en $N+1$ valeurs : $t_i = a+ih$, où $h = \frac{b-a}{N}$.
On peut alors approcher l'intégrale par la somme des aires des N rectangles de largeur h et de hauteur $f(t_i+\frac{h}{2})$:

$$\int_a^b f(t)\mathrm{d}t \simeq \sum_{i=0}^{N-1} f\left(t_i+\tfrac{h}{2}\right)h$$




```
def rectangles(f:function,a:float,b:float,N:int)->float:
    "renvoie la valeur approchée de l'integrale de f entre a et b"
    h=(b-a)/N
    I=0
    for i in range(N):
        I+=f(a+i*h+h/2)
    return I*h
```

5 Équations différentielles

5.1 Résolution d'une équation différentielle d'ordre 1 par la méthode d'Euler

On cherche la solution $y(t)$ sur un intervalle $[a,b]$, d'une équation différentielle d'ordre 1 avec la condition initiale : $y(a) = y_0$.
Il faut mettre l'équation différentielle sous la forme :

$$y' = f(y,t)$$

On pose $(t_i)_{i\in[\![0,N]\!]}$ une subdivision de l'intervalle $[a,b]$ en $N+1$ valeurs : $t_i = a+ih$, où $h = \frac{b-a}{N}$. On fait alors l'approximation :

$$y'(t_i) = \frac{\mathrm{d}y}{\mathrm{d}t} \simeq \frac{y(t_{i+1})-y(t_i)}{h}$$

c'est-à-dire

$$\begin{aligned} y(t_{i+1}) &\simeq y(t_i) + y'(t_i)h \\ &\simeq y(t_i) + f(y(t_i),t_i)h \end{aligned}$$

Partant de $y(t_0) = y_0$, on peut ainsi calculer tous les $y(t_i)$ successivement.

```
def Euler(f:function,y0:float,a:float,b:float,N:int)->list:
    "renvoie la liste des y(ti)"
    h=(b-a)/N
    y=[y0]
    for i in range(N):
        ti=a+i*h
        y.append(y[i]+f(y[i],ti)*h)
    return y
```

5.2 Résolution d'un système de n équations différentielles d'ordre 1 avec `odeint`

Un système de n équations différentielles d'ordre 1 peut se mettre sous la forme :

$$\begin{cases} y_0' = f_0(y_0,y_1,y_{n-1},t) \\ y_1' = f_1(y_0,y_1,y_{n-1},t) \\ \vdots \\ y_{n-1}' = f_{n-1}(y_0,y_1,y_{n-1},t) \end{cases}$$

c'est-à-dire

$$Y' = f(Y,t) \qquad \text{avec} \qquad Y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Les conditions initiales s'écrivent :

$$\begin{cases} y_0(t_0) = y_{00} \\ y_1(t_0) = y_{10} \\ \vdots \end{cases} \qquad \text{soit} \qquad Y(t_0) = Y_0 = \begin{pmatrix} y_{00} \\ y_{10} \\ \vdots \end{pmatrix}$$

La fonction `odeint(f,Y0,t)` du module scipy.integrate permet de résoudre ce type de système et renvoie un tableau à deux dimensions de taille `len(t),len(Y0)` contenant les composantes de Y à chaque instant du tableau `t`.

```
from scipy.integrate import odeint

def f(Y:list,t:float)->list:
    "renvoie la liste Y'(t)"
    ...
    return [dy0dt,dy1dt,...]
t=np.linspace(a,b,N) # valeurs de t auxquelles on souhaite évaluer Y
Y0=[y00,y10,...] # liste des conditions initiales

sol=odeint(f,Y0,t) # renvoie un tableau à deux dimensions de taille (N,n)

y0=sol[:,0] # tableau contenant y0 à chaque instant
y1=sol[:,1]
...
plt.plot(t,y0) # trace la courbe y0(t)
plt.plot(t,y1)
...
```

5.3 Résolution d'une équation différentielle d'ordre n

On considère une équation différentielle d'ordre n, de la forme :

$$y^{(n)} = g\left(y, y', \ldots, y^{(n-1)}, t\right)$$

Pour résoudre cette équation différentielle, il faut la vectoriser, c'est-à-dire la réécrire comme un système de n équations différentielles d'ordre 1, à n inconnues $y, y_1, y_2, ..., y_{n-1}$:

$$\begin{cases} y' = y_1 \\ {y_1}' = y_2 \\ {y_2}' = y_3 \\ \vdots \\ {y_{n-1}}' = g\left(y, y_1, \ldots, y_{n-1}, t\right) \end{cases}$$

soit

$$Y' = f(Y,t) \qquad \text{avec} \qquad Y = \begin{pmatrix} y \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} y \\ y' \\ \vdots \\ y^{(n-1)} \end{pmatrix}$$

On peut alors résoudre le système avec la fonction `odeint(f,Y0,t)`.