Chapitre 4

La bibliothèque SciPy

4.1 Introduction

La bibliothèque **SciPy** (https://www.scipy.org/) rassemble des composants Python à usage scientifique. SciPy est destiné à travailler de concert avec NumPy et contient en particulier des modules destinés . . . :

- . . . à l'**utilisation de constantes physiques** (`scipy.constants`),
- . . . à l'**optimisation** et à la **recherche de zéros** (`scipy.optimize`),
- . . . à l'**intégration numérique** (`scipy.integrate`),
- . . . à l'**algèbre linéaire** (`scipy.linalg`),
- . . . aux **statistiques** et aux **nombres aléatoires** (`scipy.stats`),
- . . . aux **fonctions spéciales** (`scipy.special`) (fonction d'Airy, fonctions de Bessel, intégrales elliptiques, etc.),
- . . . au **traitement du signal** (`scipy.signal`),
- . . . au **traitement des images** (`scipy.ndimage`).

4.2 Dictionnaire des constantes physiques et mathématiques

Le module `scipy.constants` (voir https://docs.scipy.org/) renferme toute une série de **constantes mathématiques et physiques**). Les constantes physiques sont accompagnées de leurs **unités** et en général de l'**incertitude** associée à leur mesure. Il est ainsi par exemple possible d'utiliser directement la valeur d'une constante physique dans un programme :

```
from scipy import constants
print('c = ',constants.c)
print("masse de l'électron = ",constants.m_e)
print("g = ",constants.g)
```

```
c =  299792458.0
masse de l'électron =  9.1093837015e-31
g =  9.80665
```

Il est possible d'obtenir les unités et incertitudes correspondantes en consultant le **dictionnaire** `physical_constants`. En fournissant une clé décrivant la constante recherchée, on obtient un tuple (triple) :

```
from scipy import constants
print(constants.physical_constants["speed of light in vacuum"])
print(constants.physical_constants["electron mass"])
print(constants.physical_constants["standard acceleration of
 ↪  gravity"])
print(type(constants.physical_constants["speed of light in
 ↪  vacuum"]))
print(f'Sur Terre, la constante de la gravité vaut : g =
 ↪  {constants.physical_constants["standard acceleration of
 ↪  gravity"][0]}{constants.physical_constants["standard
 ↪  acceleration of gravity"][1]} .')
```

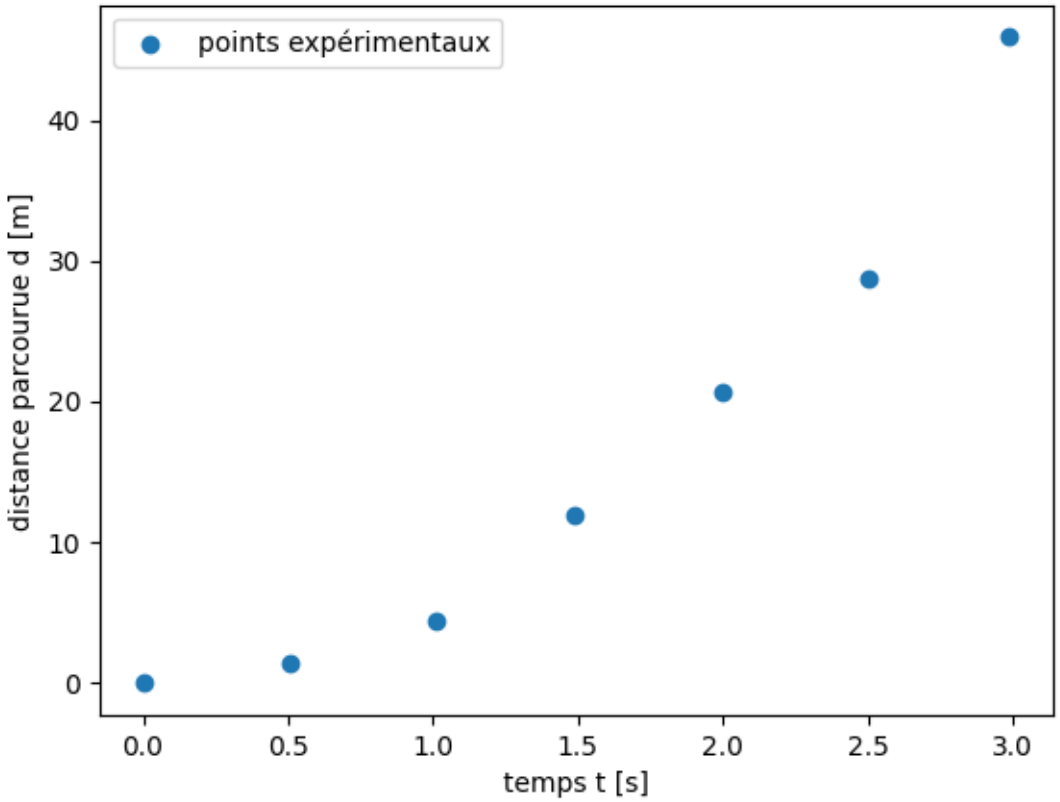
```
(299792458.0, 'm s^-1', 0.0)
(9.1093837015e-31, 'kg', 2.8e-40)
(9.80665, 'm s^-2', 0.0)
<class 'tuple'>
Sur Terre, la constante de la gravité vaut : g = 9.80665m s^-2 .
```

4.3 Ajustement de courbes (de paramètres)

La fonction `scipy.optimize.curve_fit` permet un **ajustement optimal** du (ou des) paramètre(s) d'une fonction que l'on pense susceptible de décrire un ensemble de données expérimentales.
Par exemple, essayons d'ajuster une fonction polynomiale $f(t) = at^b + c$, où a, b et c sont des paramètres réels inconnus, aux données collectées lors de la chute libre d'une masse (voir le fichier `ChuteLibreData.txt` déjà utilisé dans ce cours).
Représentation des points expérimentaux :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
t,d = np.loadtxt('ChuteLibreData.txt', usecols = (1,2), skiprows =
 ↪  4, unpack = True)
plt.xlabel("temps t [s]")
plt.ylabel("distance parcourue d [m]")
plt.scatter(t,d,label='points expérimentaux')
plt.legend(loc='best')
plt.show()
```

La fonction `plt.scatter()` permet de représenter uniquement les points, sans les relier.



Ajustement des paramètres d'une fonction polynomiale aux données expérimentales :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
t,d = np.loadtxt('ChuteLibreData.txt', usecols = (1,2), skiprows =
 ↪  4, unpack = True)
plt.xlabel("temps t [s]")
plt.ylabel("distance parcourue d [m]")
plt.scatter(t,d,label='points expérimentaux')
plt.legend(loc='best')
plt.show()
def fct_fit(x,a,b,c):
    return a * x**b + c
parametres, m_covariance = optimize.curve_fit(fct_fit,t,d)
print('tableau des paramètres : ', parametres)
print('matrice (tableau) de covariance : \n', m_covariance)
```

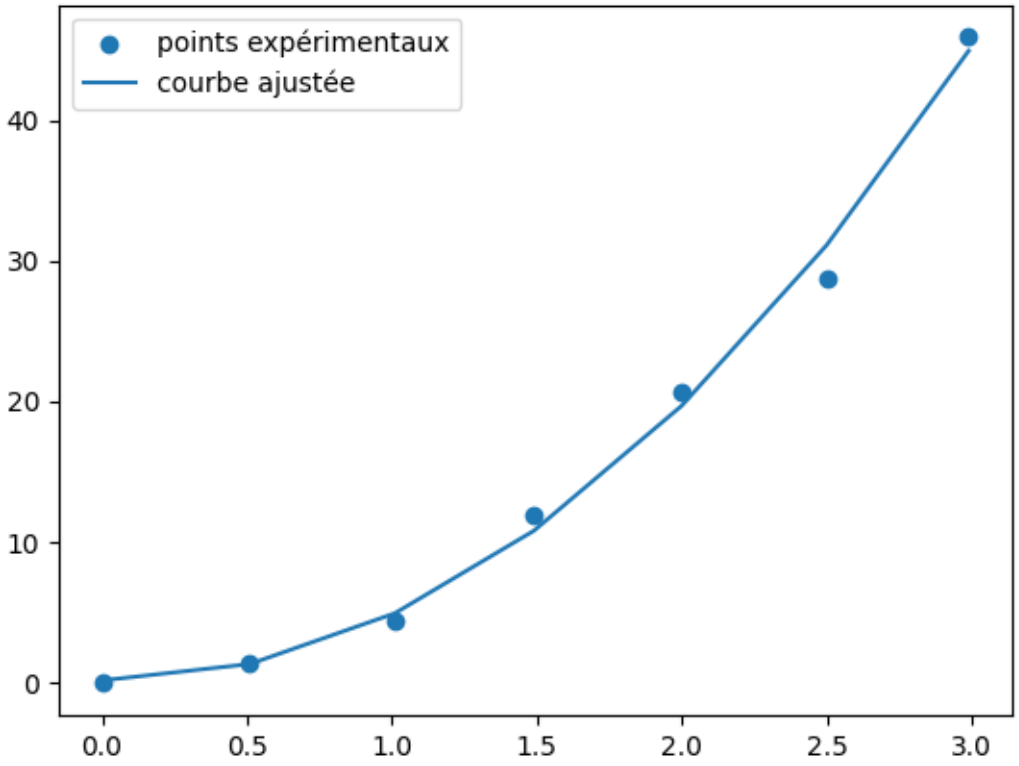
```
tableau des paramètres :  [4.67235125 2.06247522 0.19456514]
matrice (tableau) de covariance :
 [[ 0.92275613 -0.1731547  -0.81855416]
 [-0.1731547   0.03352579  0.13967152]
 [-0.81855416  0.13967152  1.22872764]]
```

Les valeurs estimées des paramètres obtenues grâce à la fonction `scipy.optimize.curve_fit` sont : $a \cong 4.766$, $b \cong 2.020$ et $c \cong 0.095$.

Les valeurs théoriques attendues sont évidemment (chute libre à la surface de la Terre) : $a = \frac{1}{2}g \cong 4.903$, $b = 2$ et $c = 0$.

Représentation des points expérimentaux et de la courbe ajustée :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
t,d = np.loadtxt('ChuteLibreData.txt', usecols = (1,2), skiprows =
 ↪  4, unpack = True)
plt.xlabel("temps t [s]")
plt.ylabel("distance parcourue d [m]")
plt.scatter(t,d,label='points expérimentaux')
plt.legend(loc='best')
plt.show()
def fct_fit(x,a,b,c):
    return a * x**b + c
parametres, m_covariance = optimize.curve_fit(fct_fit,t,d)
print('tableau des paramètres : ', parametres)
print('matrice (tableau) de covariance : \n', m_covariance)
plt.scatter(t,d,label='points expérimentaux')
plt.plot(t, fct_fit(t, parametres[0], parametres[1],
 ↪  parametres[2]), label='courbe ajustée')
plt.legend(loc='best')
plt.show()
```



4.4 Recherche de zéros

4.4.1 La méthode de la bissection

SciPy fournit une fonction `scipy.optimize.bisect` qui implémente la méthode de la bissection et demande **trois arguments obligatoires** :

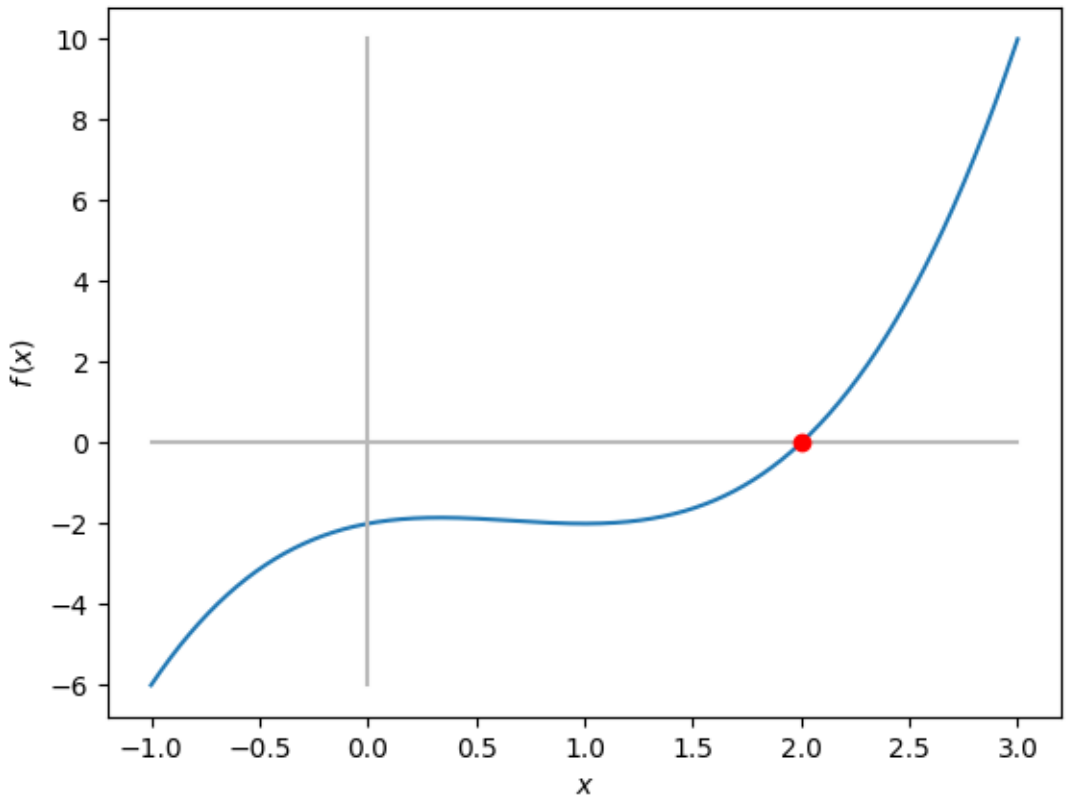
- la fonction $f(x)$,
- une limite inférieure a,
- une limite supérieure b.

Un paramètre optionnel `xtol` (par défaut `2e-12`) précise l'**erreur maximale tolérée**.
Un autre paramètre optionnel, `maxiter`, permet d'**interrompre la méthode après un certain nombre d'itérations** (par défaut, `100`).
Il est primordial que $f(a)$ et $f(b)$ n'aient pas le même signe (**condition de Bolzano**).

```
from scipy import optimize
def f(x):
    return (1+x**2)*(x-2)
x = optimize.bisect(f,-0.5,2.5,xtol=1e-3,maxiter=100)
print("Le zéro trouvé par la méthode est x =", x)
print("L'erreur commise est donnée par 2-x =", 2-x)
import numpy as np
```

```
import matplotlib.pyplot as plt
x_vec = np.linspace(-1,3,1000)
y_vec = f(x_vec)
plt.xlabel('$x$')
plt.ylabel('$f\,(x)$')
plt.plot(x_vec,y_vec)
plt.plot([x_vec[0],x_vec[-1]], [0,0], c='0.72')
plt.plot([0,0], [y_vec.min(),y_vec.max()], c='0.72')
plt.plot(x,f(x),'ro')
plt.show()
```

```
Le zéro trouvé par la méthode est x = 1.999755859375
L'erreur commise est donnée par 2-x = 0.000244140625
```



4.4.2 La fonction `newton`

SciPy fournit la fonction `scipy.optimize.newton` qui implémente la **méthode de Newton** (ou la **méthode de la sécante**, ou la **méthode de Halley**) et demande **deux arguments obligatoires** :

- la fonction $f(x)$;
- un x_0 proche du zéro cherché.

La dérivée de f peut être fournie de manière optionnelle (argument `fprime`). En son absence, la méthode de la sécante est utilisée. On retrouve les paramètres optionnels liés à la tolérance, `tol` (par défaut, `1.48e-08`), et au nombre maximum d'itérations, `maxiter` (par défaut, `50`).
Lorsque les première et deuxième dérivées sont fournies (via `fprime` et `fprime2`), la méthode utilisée est celle de Halley. Remarquons que nous n'avons pas discuté cette méthode dans le chapitre 3. Elle a été proposée par l'astronome Edmond Halley et généralise la méthode de Newton. Sa convergence est cubique.
Exemple d'utilisation avec la fonction $f(x) = (1 + x^2)(x - 2)$:
a) Méthode de la sécante (la dérivée n'est pas fournie à la fonction `newton`)

```
from scipy import optimize
def f(x):
    return (1+x**2)*(x-2)
x = optimize.newton(f,1.5,full_output=True)
print('Méthode de la sécante : \n',x)
```

```
Méthode de la sécante :
 (1.9999999999999991,       converged: True
           flag: 'converged'
 function_calls: 9
     iterations: 8
           root: 1.9999999999999991)
```

b) Méthode de Newton (la première dérivée est fournie à la fonction `newton`)

```
from scipy import optimize
def f(x):
    return (1+x**2)*(x-2)
def f_prime(x):
    return 3*x**2-4*x+1
x = optimize.newton(f,1.5,fprime=f_prime,full_output=True)
print('Méthode de Newton : \n',x)
```

```
Méthode de Newton :
 (2.0,       converged: True
           flag: 'converged'
 function_calls: 12
     iterations: 6
           root: 2.0)
```

c) Méthode de Halley (les deux premières dérivées sont fournies à la fonction `newton`)

```
from scipy import optimize
def f(x):
    return (1+x**2)*(x-2)
def f_prime(x):
    return 3*x**2-4*x+1
def f_seconde(x):
    return 6*x-4
x = optimize.newton(f,1.5,fprime=f_prime,fprime2=f_seconde,
 ↪  full_output=True)
print('Méthode de Halley : \n',x)
```

```
Méthode de Halley :
 (2.0,       converged: True
           flag: 'converged'
 function_calls: 13
     iterations: 4
           root: 2.0)
```

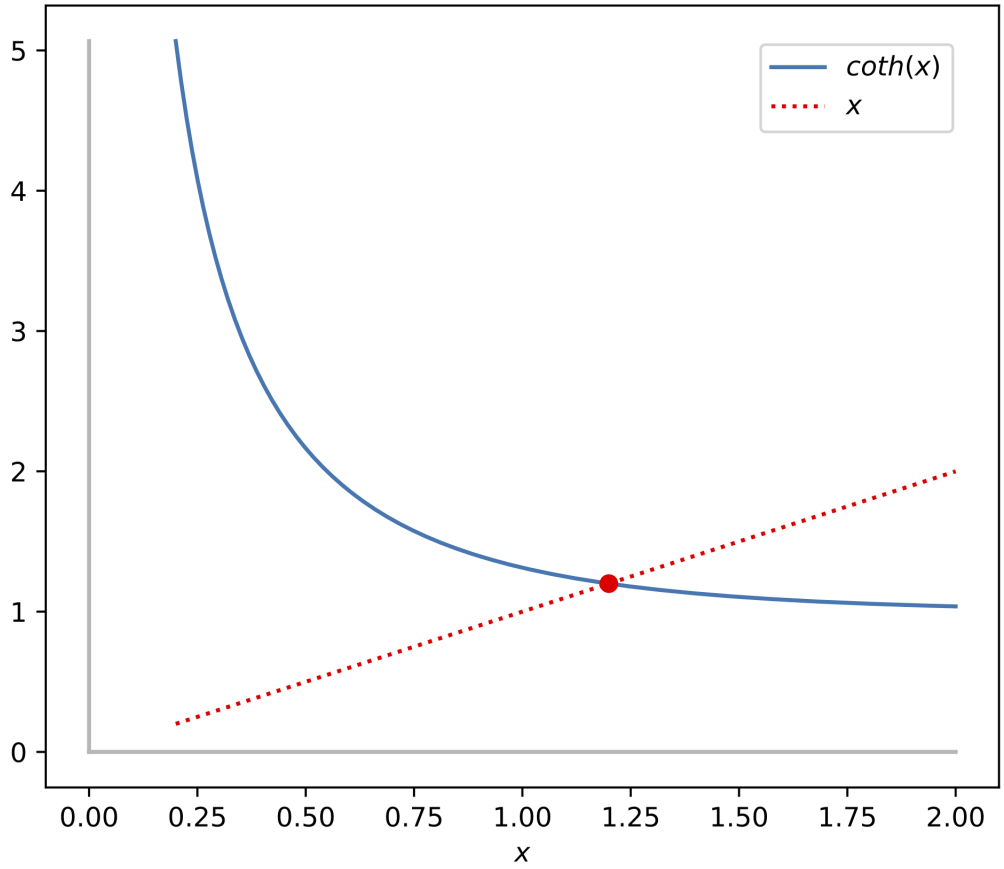
4.4.3 La fonction `fsolve`

SciPy permet également de faire appel à un algorithme a priori plus efficace pour trouver un zéro d'une fonction à l'aide de la fonction `scipy.optimize.fsolve`. La méthode ne nécessite qu'un point de départ (proche du point cherché). Elle ne fournit pas de garantie de convergence. Si la dérivée de la fonction n'est pas fournie en argument, elle est estimée.

Recherchons par exemple la solution de l'équation $\coth(x) = x$ pour $x > 0$:

```
import numpy as np
from scipy import optimize
def f(x):
    return np.cosh(x)/np.sinh(x) - x
zero = optimize.fsolve(f,1.0)
print('x = ', zero[0], ' et f(x) = ', f(zero[0]))
```

```
x =  1.1996786402577135  et f(x) =  2.90878432451791e-14
```



Remarquons que la fonction `fsolve` permet de résoudre des problèmes multidimensionnels, c'est-à-dire des problèmes dans lesquels plusieurs inconnues scalaires sont à déterminer (la **dimension** d'un problème est donnée par le nombre de variables scalaires dont on cherche à déterminer les valeurs). Ainsi, l'exemple suivant correspond à un système bidimensionnel (intersection d'une ellipse et d'une parabole) :

$$\begin{cases} \dfrac{x^2}{8} + \dfrac{y^2}{2} = 1 & \text{(ellipse)} \\ y = \dfrac{x^2}{4} & \text{(parabole)} \end{cases}$$

```
import numpy as np
from scipy import optimize
def fct(valeurs):
    x = valeurs[0]
    y = valeurs[1]
    SE = np.zeros(2, float) # SE : Système d'Equations (ndarray)
    SE[0] = pow(x,2)/8 + pow(y,2)/2 - 1
    SE[1] = y - pow(x,2)/4
    return SE
valeurs_0 = np.array([0.5,0.5])
solution = optimize.fsolve(fct,valeurs_0)
print('x = ',solution[0],' et  y = ',solution[1])
```

```
1 x =  2.0000000000000018  et  y =  1.0000000000000007
```