Introduction à Scipy

October 15, 2023

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

1 Introduction à Scipy

Scipy contient des implémentations de plusieurs algorithmes numérique : * Fonctions spéciales * Intégrales * Equations différentielles * Optimisation * Algebre linéaires * Transformée de Fourier

1.1 Fonctions spéciales

Fonctions qui ne sont pas dans numpy : Bessel, Airy, fonction d'erreur, … (Ce sont des fonctions définies par des intégrales)

Exemple : fonction erreur

$$\mathrm{erf}(x) = \int_0^x \frac{2}{\sqrt{\pi}} e^{-t^2} dt$$

[3]:
```
from scipy.special import erf

erf(1)
```

[3]:
```
0.8427007929497148
```

1.2 Intégrales numériques

1.2.1 Intégrale d'une fonction :

Il existe plusieurs algorithme. Le plus simple : `quad`

[4]:
```
import numpy as np
from scipy.integrate import quad
```

[5]:
```
quad?
```

[6]:
```
def ma_fonction(t):
    return 2/np.sqrt(np.pi)*np.exp(-t**2)

# Renvoie la valeur et une estimation de l'incertitude
```

```
res, err = quad(ma_fonction, 0, 1)
print(res)
print(res - erf(1))
```

```
0.8427007929497149
1.1102230246251565e-16
```

[7]:
```
def ma_fonction(t, sigma):
    return np.exp(-t**2/(2*sigma**2))

quad(ma_fonction, 0, 1, args=(0.45,))
```

[7]:
```
(0.5491762723634688, 6.09708142155739e-15)
```

1.2.2 Remarques

Si on connait la fonction, ne pas en faire un tableau

La fonction quad calcul automatiquement les points pour l'intégrale afin d'atteintre une erreur donnée

La fonction quad peut intégrer sur des bornes infinies (`np.inf`)

[11]:
```
list_of_points = []
def ma_fonction(t):
    list_of_points.append(t)
    return 2/np.sqrt(np.pi)*np.exp(-t**2)
res, err = quad(ma_fonction, 0, np.inf)
print(res)
print("Nombre de points :" , len(list_of_points))
#print(list_of_points)
print("Erreur :", np.abs(res - 1 ))
```

```
1.0
Nombre de points : 135
Erreur : 0.0
```

[12]:
```
print(len(list_of_points))
```

```
135
```

[17]:
```
list_of_points = []
def ma_fonction(t):
    list_of_points.append(t)
    return t/(1+t**2)
res, err = quad(ma_fonction, 0, np.inf)
print(res)
print("Nombre de points :" , len(list_of_points))
#print(list_of_points)
```

```python
print("Erreur :", np.abs(res - 1 ))
```

```
40.99601281916947
Nombre de points : 1485
Erreur : 39.99601281916947
```

```
<ipython-input-17-0b027cf4276a>:5: IntegrationWarning: The maximum number of
subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  res, err = quad(ma_fonction, 0, np.inf)
```

[ ]:

```python

```

1.2.3 Intégrales d'un tableau de points

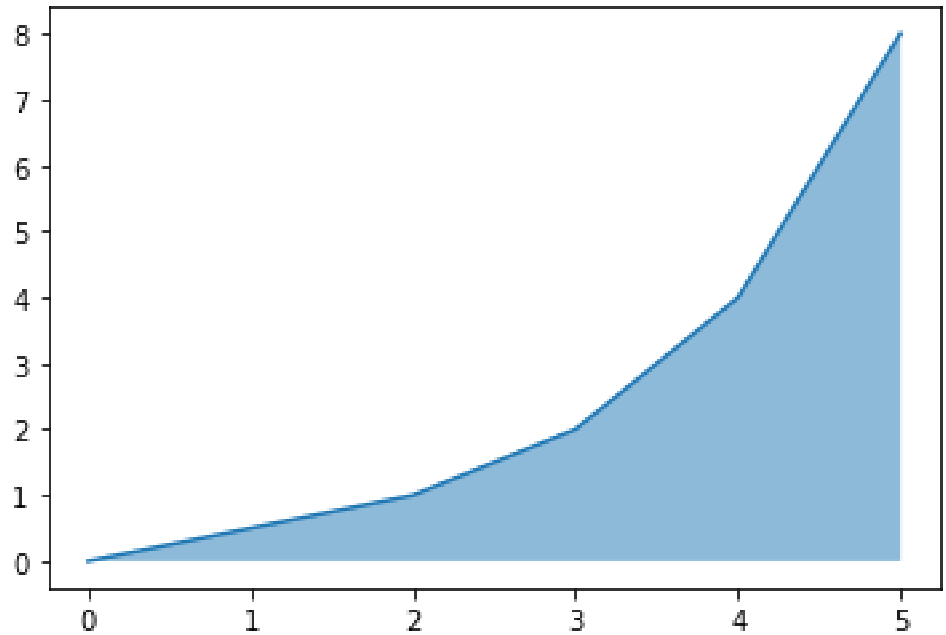
Utiliser la fonction trapz ou simps

[19]:

```python
from scipy.integrate import trapz

data_y = [0, 1, 2, 4, 8]
data_x = [0, 2, 3, 4, 5]

plt.plot(data_x, data_y)
plt.fill_between(data_x, data_y, alpha=.5)

trapz(data_y, data_x)
```

[19]:

```
11.5
```



1.3 Equations différentielles

La librairie `scipy.integrate` contient des fonctions pour résoudre les équations différentielles ordinaires, c'est à dire des équations de la forme:

$$\frac{dy}{dt} = f(t, y)$$

avec conditions initiales (on connait y à l'instant t_0). La variable y peut être un tableau numpy.

On utilise la fonction `solve_ivp` (remplace `ode` ou `odeint`):

`def solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, ...)`

Il existe plusieurs méthodes d'intégration (par défaut Runge-Kutta d'ordre 5(4) qui adapte la taille des pas)

La fonction `solve_ivp` renvoie un objet (dictionnaire) qui le résultat (res.y) mais aussi d'autres informations sur la convergence de l'algorithme.

Exemple :

$$\frac{dy}{dt} = -y$$

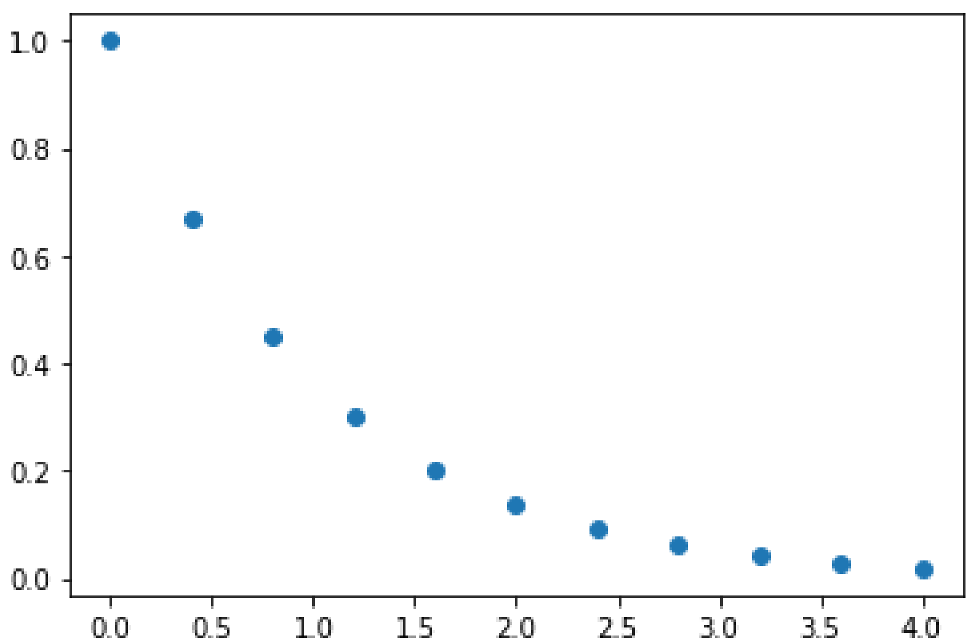
[22]:
```
from scipy.integrate import solve_ivp
# Solve initial value problem

def f(t, y):
    return -y
```

```
res = solve_ivp(f, t_span=[0, 4], y0=[1], t_eval=np.linspace(0, 4, 11),
                rtol=1E-7, atol=1E-7)
res
```

[22]:
```
  message: The solver successfully reached the end of the integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  4.000e-01  8.000e-01  1.200e+00  1.600e+00
             2.000e+00  2.400e+00  2.800e+00  3.200e+00  3.600e+00
             4.000e+00]
        y: [[ 1.000e+00  6.703e-01  4.493e-01  3.012e-01  2.019e-01
              1.353e-01  9.072e-02  6.081e-02  4.076e-02  2.732e-02
              1.832e-02]]
      sol: None
 t_events: None
 y_events: None
     nfev: 122
     njev: 0
      nlu: 0
```

[23]:
```
plt.plot(res.t, res.y[0], 'o')
```

[23]:
```
[<matplotlib.lines.Line2D at 0x7f7c7ea417c0>]
```



[24]:
```
res.y[0, -1] - np.exp(-4)
```

[24]: 1.7731305576584866e-08

[15]:
```python
len(res.t)
```

[15]: 11

[16]:
```python
# Utilisation d'un paramètre
def f(t, y, tau):
    return -y/tau

res = solve_ivp(lambda t, y:f(t, y, tau=0.1), t_span=[0, 4], y0=[1], t_eval=np.
 ↪linspace(0, 4, 11))
res
```

[16]:
```
  message: 'The solver successfully reached the end of the integration
interval.'
     nfev: 140
     njev: 0
      nlu: 0
      sol: None
   status: 0
  success: True
        t: array([0. , 0.4, 0.8, 1.2, 1.6, 2. , 2.4, 2.8, 3.2, 3.6, 4. ])
 t_events: None
        y: array([[ 1.00000000e+00,  1.83531441e-02,  3.36575782e-04,
         6.25529975e-06,  1.60535093e-07,  1.43992293e-07,
         9.50371941e-07,  7.83634365e-09, -2.64102615e-07,
        -4.41883282e-07,  2.53752885e-07]])
 y_events: None
```

1.4 Equations différentielles d'ordre élevé

L'astuce consiste à augmenter la dimension de y en rajoutant des fonctions intermédiaires qui sont les dérivées de la fonction initiale.

Par exemple l'équation

$$\frac{d^2y}{dt^2} = \frac{f(y)}{m}$$

devient

$$\frac{d}{dt}\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} y' \\ f(y)/m \end{pmatrix} = F(y, y')$$

Voir le TD

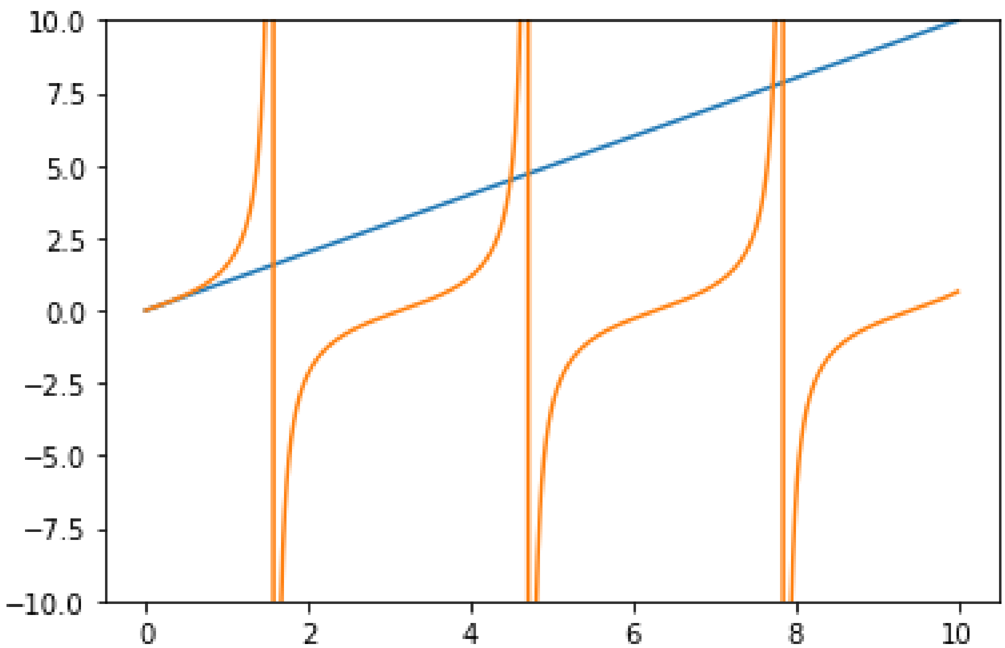
2 Optimisation

- Zeros d'une fonction
- Minimum
- Ajustement d'une courbe

Exemple : * première solution > 0 de $\tan(x) = x$ * Premier minimum de $sinc(x)$

[17]:
```
x = np.linspace(0, 10, 2001)
plt.plot(x, x)
plt.plot(x, np.tan(x))
plt.ylim(-10, 10)
```

[17]: (-10.0, 10.0)



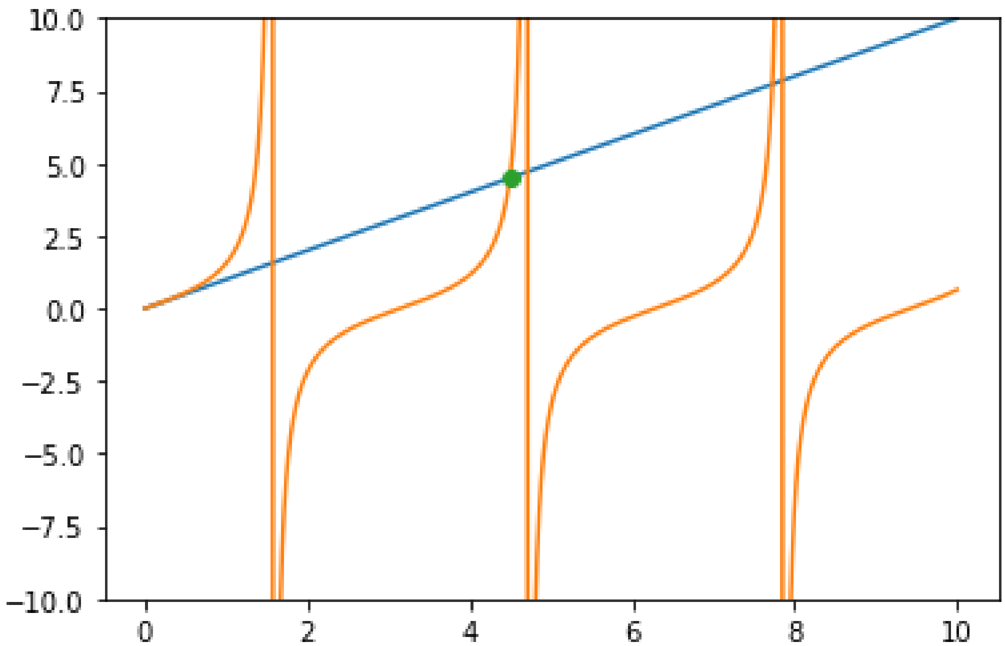
[25]:
```
from scipy.optimize import root_scalar

def f(x):
    return np.tan(x) - x

res = root_scalar(f, bracket=[4, 4.7], method='brentq')
res
```

[25]:
```
      converged: True
           flag: 'converged'
 function_calls: 12
```

```
iterations: 11
      root: 4.493409457909064
```

[28]:
```
x = np.linspace(0, 10, 2001)
plt.plot(x, x)
plt.plot(x, np.tan(x))
plt.ylim(-10, 10)
plt.plot(res.root, [f(res.root)+res.root], 'o')

res.root
```

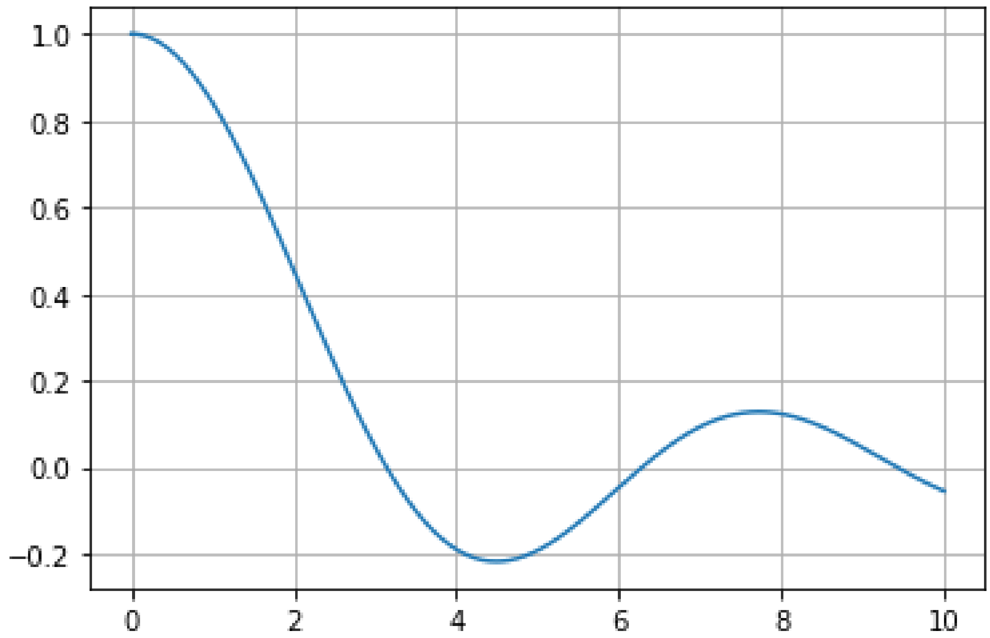
[28]:
```
4.493409457909064
```



2.0.1 Minimisation

[29]:
```
def french_sinc(x):
    return np.sinc(x/np.pi)
```

[30]:
```
plt.plot(x, french_sinc(x))
plt.grid()
```



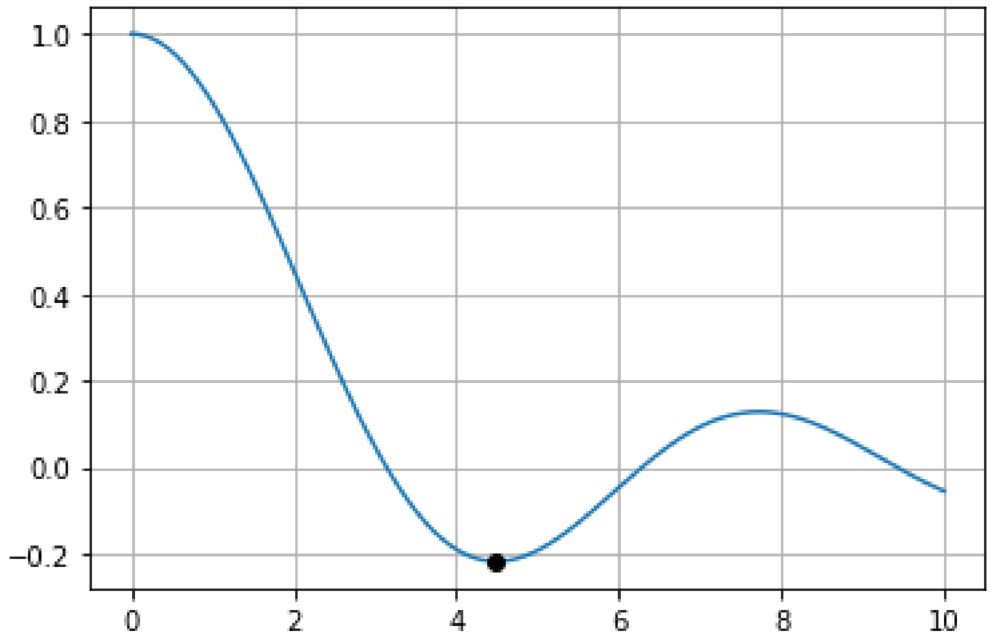
[34]:
```
from scipy.optimize import minimize_scalar
minimize_scalar?
```

[32]:
```
res = minimize_scalar(french_sinc, [4, 4.71])
res
```

[32]:
```
 message:
          Optimization terminated successfully;
          The returned value satisfies the termination criteria
          (using xtol = 1.48e-08 )
 success: True
     fun: -0.21723362821122166
       x: 4.4934094607238
     nit: 9
    nfev: 12
```

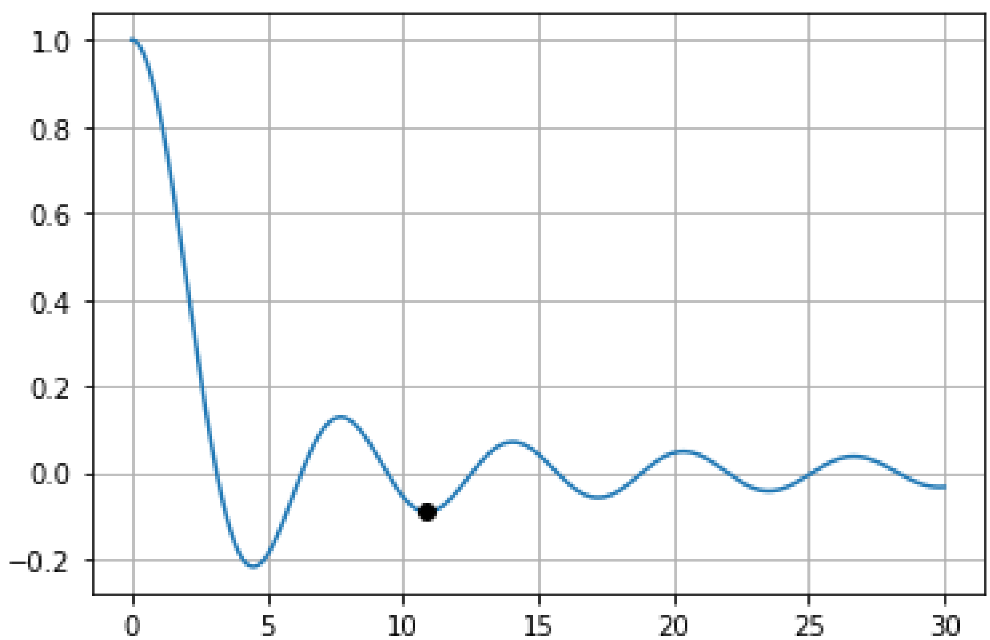
[33]:
```
plt.plot(x, french_sinc(x))
plt.grid()
plt.plot(res.x, res.fun, 'ko')
```

[33]: [<matplotlib.lines.Line2D at 0x7f7c7e8a81f0>]



[25]:
```
x = np.linspace(0, 30, 2001)
plt.plot(x, french_sinc(x))
plt.grid()
res = minimize_scalar(french_sinc, [10, 205])
plt.plot(res.x, res.fun, 'ko')
```

[25]: [<matplotlib.lines.Line2D at 0x7f8a26f8ba60>]



3 Algèbre linéaire

numpy.linalg et scipy.linalg (plus de fonction dans scipy)

- Matrice : np.matrix (produit matriciel)
- Inverse de matrice
- Diagonalisation/valeurs propres/vecteurs propres

Exemple: valeurs propres de

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

Tracer les vp en fonction de δ pour $\Omega = 1$

$$\begin{bmatrix} \delta & \frac{\Omega}{2} & 0 \\ \frac{\Omega}{2} & 0 & \frac{\Omega}{2} \\ 0 & \frac{\Omega}{2} & -\delta \end{bmatrix}$$

[38]:
```
import numpy as np

a = np.array([[0, 1], [1, 1]])
a*a
a@a

a = np.matrix([[0, 1], [1, 1]])
a*a
```

[38]:
```
matrix([[1, 1],
        [1, 2]])
```

[39]:
```
H = np.matrix([[1, 1, 0], [1, 0, 1], [0, 1, -1]])
H
```

[39]:
```
matrix([[ 1,  1,  0],
        [ 1,  0,  1],
        [ 0,  1, -1]])
```

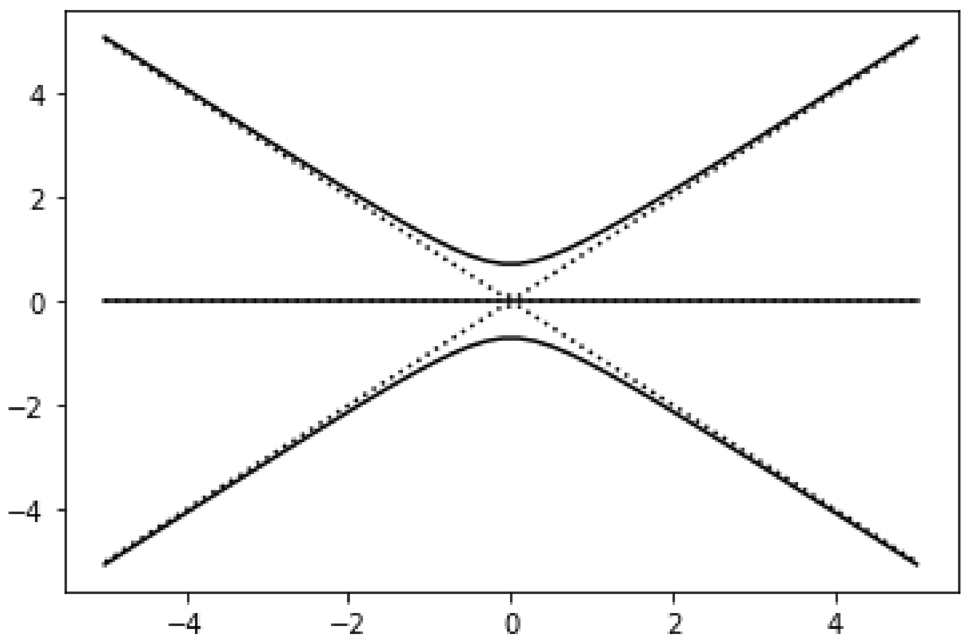
[40]:
```
from scipy.linalg import eigh # Matrice hermicienne

eigh(H) # Renvoie les valeurs propres et vecteurs propres
```

[40]:
```
(array([-1.73205081,  0.        ,  1.73205081]),
 array([[-0.21132487, -0.57735027,  0.78867513],
        [ 0.57735027,  0.57735027,  0.57735027],
        [-0.78867513,  0.57735027,  0.21132487]]))
```

[41]:
```
def trois_niveaux(delta, omega):
    H = np.matrix([[delta, omega/2, 0], [omega/2, 0, omega/2], [0, omega/2,␣
↪-delta]])
    return eigh(H)[0]

all_delta = np.linspace(-5, 5)
sans_couplage = np.array([trois_niveaux(delta, omega=0) for delta in all_delta])
avec_couplage = np.array([trois_niveaux(delta, omega=1) for delta in all_delta])
```

[32]:
```
plt.plot(all_delta, sans_couplage, 'k:')
plt.plot(all_delta, avec_couplage, 'k-')
```

[32]:
```
[<matplotlib.lines.Line2D at 0x7f8a271d8c70>,
 <matplotlib.lines.Line2D at 0x7f8a271d8e20>,
 <matplotlib.lines.Line2D at 0x7f8a271d8cd0>]
```



[ ]: