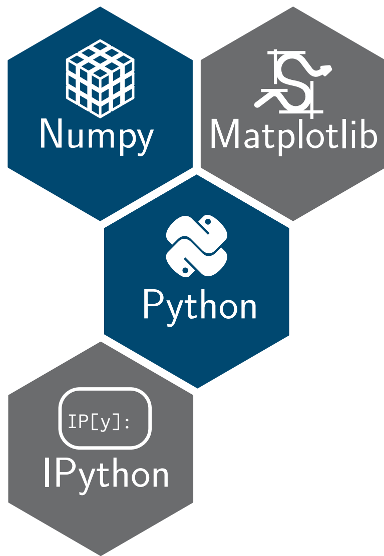


Option « Programmation en Python »

`scipy` : librairie pour la programmation scientifique

scipy ?

- Le module scipy vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- scipy s'appuie sur numpy en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

scipy ?

- Le module scipy vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- scipy s'appuie sur numpy en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

scipy ?

- Le module scipy vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- scipy s'appuie sur numpy en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

- Le module `scipy` vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- `scipy` s'appuie sur `numpy` en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

scipy ?

- Le module scipy vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- scipy s'appuie sur numpy en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

- Le module `scipy` vise à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique
- `scipy` s'appuie sur `numpy` en utilisant les objets de type tableaux et fournit des algorithmes scientifiques optimisés :
  - Algèbre linéaire (résolution d'équations linéaires, valeurs/vecteur propres)
  - Fonctions spéciales (fonction de Bessel, loi de distribution,...)
  - Algorithmes d'interpolation, d'intégration et d'optimisation
  - Traitement du signal et des images (transformée de Fourier, convolution,...)

Installation & importation de scipy

- Installation *via* pip

```
pip install scipy
```

- Convention d'importation : l'ensemble des modules scipy peuvent être importés individuellement

```
In [1]: import scipy.linalg as linalg
```

Installation & importation de scipy

- Installation *via* pip

```
>_ pip install scipy
```

- Convention d'importation : l'ensemble des modules scipy peuvent être importés individuellement

```
In [1]: import scipy.linalg as linalg
```

scipy.linalg : algèbre linéaire

- Le module `scipy.linalg` inclut diverses fonctions dont
  - les opérations matricielles (inversion de matrices, calcul de déterminant)
  - résolution d'équations linéaires $Ax = b$
  - recherche de valeurs/vecteurs propres
  - pivot de Gauss, décomposition en valeurs singulières, ...

```
In [1]: from scipy import linalg

In [2]: A = np.random.rand(2, 2)
In [3]: A
Out[3]:
array([[ 0.38005786,  0.06901768],
       [ 0.86144407,  0.03601743]])

In [4]: linalg.det(A)
Out[4]: -0.045766161972959955

In [5]: linalg.inv(A)*linalg.det(A)
Out[5]:
array([[ 0.03601743, -0.06901768],
       [-0.86144407,  0.38005786]])
```

scipy.linalg : algèbre linéaire

- Le module scipy.linalg inclut diverses fonctions dont
  - les opérations matricielles (inversion de matrices, calcul de déterminant)
  - résolution d'équations linéaires $Ax = b$
  - recherche de valeurs/vecteurs propres
  - pivot de Gauss, décomposition en valeurs singulières, ...

```
In [1]: from scipy import linalg

In [2]: A = np.random.rand(2, 2)
In [3]: A
Out[3]:
array([[ 0.38005786,  0.06901768],
       [ 0.86144407,  0.03601743]])

In [4]: linalg.det(A)
Out[4]: -0.045766161972959955

In [5]: linalg.inv(A)*linalg.det(A)
Out[5]:
array([[ 0.03601743, -0.06901768],
       [-0.86144407,  0.38005786]])
```

- Résolution d'équation linéaire $Ax = b$

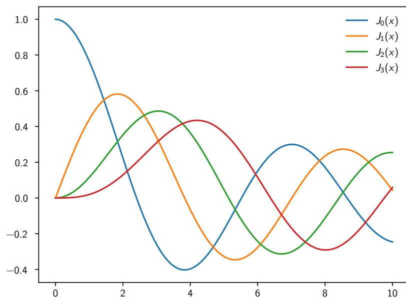
```
In [1]: from scipy import linalg

In [2]: A = np.random.rand(3, 3)
In [3]: b = np.random.rand(3)

In [4]: x = linalg.solve(A, b)

In [5]: x
Out[5]: array([ 0.61826973,  0.09161294, -0.35492909])

In [6]: np.dot(A, x) - b
Out[6]: array([ 0.,  0.,  0.])
```

- Recherche de valeurs/vecteur propres $Av_n = \lambda_n v_n$ où v_n est le $n^{\text{ième}}$ vecteur propre et λ_n la $n^{\text{ième}}$ valeur propre

```
In [1]: from scipy import linalg

In [2]: evals, evecs = linalg.eig(A)

In [3]: evals
Out[3]: array([ 1.89774095+0.j, -0.27128129+0.j,  0.34921006+0.j])

In [4]: evecs
Out[4]:
array([[-0.52832832, -0.7845609 ,  0.06535214],
       [-0.49359384,  0.58672007, -0.51283945],
       [-0.69082147,  0.2005586 ,  0.85599345]])

In [5]: n = 1
In [6]: linalg.norm(np.dot(A, evecs[:,n]) - evals[n]*evecs[:,n])
Out[6]: 5.8191634490868685e-16
```

- Fonctions de Bessel : $x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0$
  - Fonctions de Bessel de première espèce J_n
  - Fonctions de Bessel de seconde espèce Y_n

```
In [1]: from scipy.special import jn, yn

In [2]: x = np.linspace(0, 10, 100)
In [3]: for n in range(4):
   ...:     plt.plot(x, jn(n, x), label=r"$J_%d(x)$" % n)
In [4]: plt.legend()
In [5]: for n in range(4):
   ...:     plt.plot(x, yn(n, x), label=r"$Y_%d(x)$" % n)
In [6]: plt.legend()
```



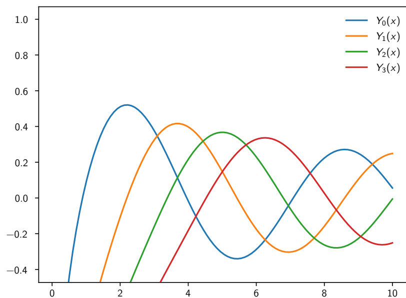
- Pour découvrir l'ensemble des fonctions spéciales offertes par scipy

```
In [7]: from scipy import special
In [8]: special?
```

scipy.special : fonctions spéciales

- Fonctions de Bessel : $x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} + (x^2 - \alpha^2)y = 0$
  - Fonctions de Bessel de première espèce J_n
  - Fonctions de Bessel de seconde espèce Y_n

```
In [1]: from scipy.special import jn, yn

In [2]: x = np.linspace(0, 10, 100)
In [3]: for n in range(4):
   ...:     plt.plot(x, jn(n, x), label=r"$J_%d(x)$" % n)
In [4]: plt.legend()
In [5]: for n in range(4):
   ...:     plt.plot(x, yn(n, x), label=r"$Y_%d(x)$" % n)
In [6]: plt.legend()
```

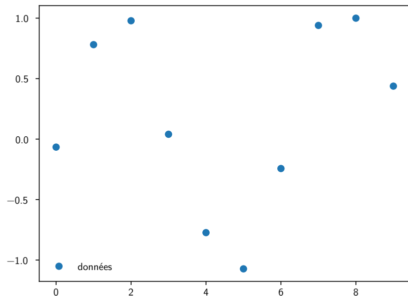


- Pour découvrir l'ensemble des fonctions spéciales offertes par scipy

```
In [7]: from scipy import special
In [8]: special?
```

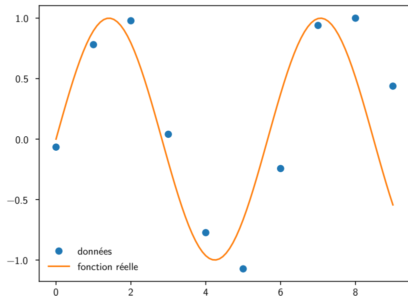
scipy.interpolate : interpolation numérique

```
In [1]: def f(x):
   ...:     return np.sin(x)

In [2]: n = np.arange(0, 10)
In [3]: y_meas = f(n) + 0.1*np.random.randn(n.size)
In [4]: from scipy.interpolate import interp1d
In [5]: linear_interpolation = interp1d(n, y_meas)
In [6]: yinterp1 = linear_interpolation(np.linspace(0, 9, 100))
In [7]: cubic_interpolation = interp1d(n, y_meas, kind="cubic")
In [8]: yinterp2 = cubic_interpolation(np.linspace(0, 9, 100))
```



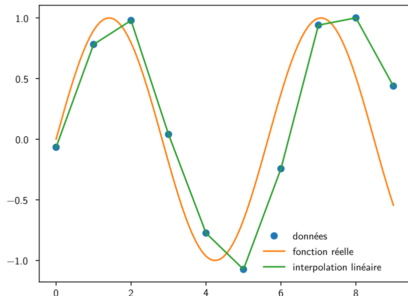
scipy.interpolate : interpolation numérique

```
In [1]: def f(x):
   ...:     return np.sin(x)

In [2]: n = np.arange(0, 10)
In [3]: y_meas = f(n) + 0.1*np.random.randn(n.size)
In [4]: from scipy.interpolate import interp1d
In [5]: linear_interpolation = interp1d(n, y_meas)
In [6]: yinterp1 = linear_interpolation(np.linspace(0, 9, 100))
In [7]: cubic_interpolation = interp1d(n, y_meas, kind="cubic")
In [8]: yinterp2 = cubic_interpolation(np.linspace(0, 9, 100))
```



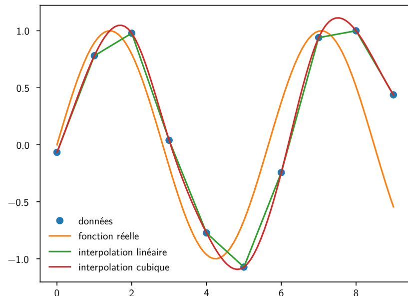
scipy.interpolate : interpolation numérique

```
In [1]: def f(x):
   ...:     return np.sin(x)

In [2]: n = np.arange(0, 10)
In [3]: y_meas = f(n) + 0.1*np.random.randn(n.size)
In [4]: from scipy.interpolate import interp1d
In [5]: linear_interpolation = interp1d(n, y_meas)
In [6]: yinterp1 = linear_interpolation(np.linspace(0, 9, 100))
In [7]: cubic_interpolation = interp1d(n, y_meas, kind="cubic")
In [8]: yinterp2 = cubic_interpolation(np.linspace(0, 9, 100))
```



`scipy.interpolate` : interpolation numérique

```
In [1]: def f(x):
   ...:     return np.sin(x)

In [2]: n = np.arange(0, 10)
In [3]: y_meas = f(n) + 0.1*np.random.randn(n.size)

In [4]: from scipy.interpolate import interp1d
In [5]: linear_interpolation = interp1d(n, y_meas)
In [6]: yinterp1 = linear_interpolation(np.linspace(0, 9, 100))

In [7]: cubic_interpolation = interp1d(n, y_meas, kind="cubic")
In [8]: yinterp2 = cubic_interpolation(np.linspace(0, 9, 100))
```



- L'intégration numérique de $\int_a^b f(x)\mathrm{d}x$ peut se faire *via* le module `scipy.integrate` :
  - **`quad`** calcule une intégrale simple
  - **`dblquad`** calcule une intégrale double
  - **`tplquad`** calcule une intégrale triple
  - **`nquad`** calcule une intégrale à n dimensions
- Exemple $\int_{-\infty}^{+\infty} \exp(-x^2)\mathrm{d}x = \sqrt{\pi}$

```
In [1]: import scipy.integrate as integrate

In [2]: val, abserr = integrate.quad(lambda x : np.exp(-x**2), -np.inf, +np.inf)
In [3]: print("I =", val, "+/-", abserr)
I = 1.7724538509055159 +/- 1.4202636780944923e-08
```

`scipy.integrate` : intégration de fonctions

- L'intégration numérique de $\int_a^b f(x)\mathrm{d}x$ peut se faire *via* le module `scipy.integrate` :
  - **quad** calcule une intégrale simple
  - **dblquad** calcule une intégrale double
  - **tplquad** calcule une intégrale triple
  - **nquad** calcule une intégrale à n dimensions

- Exemple $\int_{-\infty}^{+\infty} \exp(-x^2)\mathrm{d}x = \sqrt{\pi}$

```
In [1]: import scipy.integrate as integrate

In [2]: val, abserr = integrate.quad(lambda x : np.exp(-x**2), -np.inf, +np.inf)
In [3]: print("I =", val, "+/-", abserr)
I = 1.7724538509055159 +/- 1.4202636780944923e-08
```

`scipy.integrate` : intégration de fonctions

- Exemple d'intégration avec passage de paramètre : $I(a, b) = \int_0^1 (ax^2 + b)dx$

```
In [1]: import scipy.integrate as integrate

In [2]: def integrand(x, a, b):
   ...:     return a*x**2+b
In [3]: a = 2
In [4]: b = 1
In [5]: integrate.quad(integrand, 0, 1, args=(a, b))
Out[7]: (1.6666666666666667, 1.8503717077085944e-14)
```

- Exemple d'intégrale multiple

$$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n} \mathrm{d}t\mathrm{d}x = \frac{1}{n}$$

```
In [1]: def I(n):
   ...:     from scipy.integrate import dblquad
   ...:     return dblquad(lambda t, x: np.exp(-x*t)/t**n, 0, np.inf, lambda x: 1, lambda x: np.inf)

In [2]: I(4)[0], I(4)[0]-1/4
Out[2]: (0.2500000000043577, 4.357680882804971e-12)

In [3]: I(100)[0], I(100)[0]-1/100
Out[3]: (0.010000000000118046, 1.1804619781674575e-13)
In [4]: def I(n):
   ...:     from scipy.integrate import nquad
   ...:     return nquad(lambda t, x: np.exp(-x*t)/t**n, [[1, np.inf], [0, np.inf]])
```

- Exemple d'intégrale multiple

$$I_n = \int_0^{\infty} \int_1^{\infty} \frac{e^{-xt}}{t^n} \mathrm{d}t\mathrm{d}x = \frac{1}{n}$$

```
In [1]: def I(n):
   ...:     from scipy.integrate import dblquad
   ...:     return dblquad(lambda t, x: np.exp(-x*t)/t**n, 0, np.inf, lambda x: 1, lambda x: np.inf)

In [2]: I(4)[0], I(4)[0]-1/4
Out[2]: (0.2500000000043577, 4.357680882804971e-12)

In [3]: I(100)[0], I(100)[0]-1/100
Out[3]: (0.010000000000118046, 1.1804619781674575e-13)

In [4]: def I(n):
   ...:     from scipy.integrate import nquad
   ...:     return nquad(lambda t, x: np.exp(-x*t)/t**n, [[1, np.inf], [0, np.inf]])
```

scipy.integrate : résolution d'équations différentielles ordinaires

- scipy fournit l'interface **odeint** pour résoudre les EDO en plus de l'interface ode, plus complète mais plus subtile
- Une équation différentielle ordinaire peut s'écrire sous la forme $y' = f(y, t)$ où $y = [y_1(t), y_2(t), \ldots, y_n(t)]$ et f **est une fonction fournissant les dérivées des fonctions** $y_i(t)$
- La résolution *via* la fonction odeint implique la connaissance de f et des conditions initiales $y(0)$

```
y_t = odeint(f, y_0, t)
```

où t est un vecteur numpy correspondant à l'échantillonnage en temps et y_t contient, en chaque temps t, une colonne pour chaque solution $y_i(t)$

`scipy.integrate` : résolution d'équations différentielles ordinaires

- `scipy` fournit l'interface **`odeint`** pour résoudre les EDO en plus de l'interface `ode`, plus complète mais plus subtile
- Une équation différentielle ordinaire peut s'écrire sous la forme $y' = f(y, t)$ où $y = [y_1(t), y_2(t), \ldots, y_n(t)]$ et f **est une fonction fournissant les dérivées des fonctions** $y_i(t)$
- La résolution *via* la fonction `odeint` implique la connaissance de f et des conditions initiales $y(0)$

```
y_t = odeint(f, y_0, t)
```

où `t` est un vecteur `numpy` correspondant à l'échantillonnage en temps et `y_t` contient, en chaque temps `t`, une colonne pour chaque solution $y_i(t)$

`scipy.integrate` : résolution d'équations différentielles ordinaires

- `scipy` fournit l'interface **`odeint`** pour résoudre les EDO en plus de l'interface `ode`, plus complète mais plus subtile
- Une équation différentielle ordinaire peut s'écrire sous la forme $y' = f(y, t)$ où $y = [y_1(t), y_2(t), \ldots, y_n(t)]$ et f **est une fonction fournissant les dérivées des fonctions $y_i(t)$**
- La résolution *via* la fonction `odeint` implique la connaissance de f et des conditions initiales $y(0)$

```
y_t = odeint(f, y_0, t)
```

où `t` est un vecteur `numpy` correspondant à l'échantillonnage en temps et `y_t` contient, en chaque temps `t`, une colonne pour chaque solution $y_i(t)$

`scipy.integrate` : résolution d'équations différentielles ordinaires

☞ Mouvement du double pendule

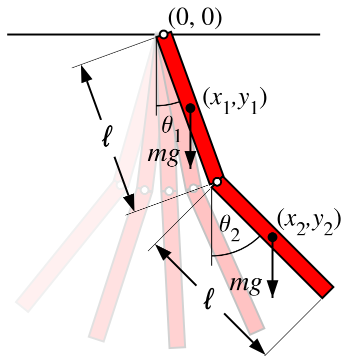
$$\dot{\theta}_1 = \frac{6}{m\ell^2} \times \frac{2p_{\theta_1} - 3\cos(\theta_1 - \theta_2)p_{\theta_2}}{16 - 9\cos^2(\theta_1 - \theta_2)}$$

$$\dot{\theta}_2 = \frac{6}{m\ell^2} \times \frac{8p_{\theta_2} - 3\cos(\theta_1 - \theta_2)p_{\theta_1}}{16 - 9\cos^2(\theta_1 - \theta_2)}$$

$$\dot{p}_{\theta_1} = -\frac{1}{2}m\ell^2\left[\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) + 3\frac{g}{\ell}\sin\theta_1\right]$$

$$\dot{p}_{\theta_2} = -\frac{1}{2}m\ell^2\left[-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) + \frac{g}{\ell}\sin\theta_2\right]$$

où p_{θ_i} sont les impulsions des barycentres (x_1, y_1) et (x_2, y_2).
On pose $y = [\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}]$



`scipy.integrate` : résolution d'équations différentielles ordinaires

Mouvement du double pendule

$$\dot{y}_1 = \frac{6}{m\ell^2} \times \frac{2y_3 - 3\cos(y_1 - y_2)y_4}{16 - 9\cos^2(y_1 - y_2)}$$

$$\dot{y}_2 = \frac{6}{m\ell^2} \times \frac{8y_4 - 3\cos(y_1 - y_2)y_3}{16 - 9\cos^2(y_1 - y_2)}$$

$$\dot{y}_3 = -\frac{1}{2}m\ell^2 \left[\dot{y}_1\dot{y}_2 \sin(y_1 - y_2) + 3\frac{g}{\ell}\sin y_1\right]$$

$$\dot{y}_4 = -\frac{1}{2}m\ell^2 \left[-\dot{y}_1\dot{y}_2 \sin(y_1 - y_2) + \frac{g}{\ell}\sin y_2\right]$$

```
In [1]: def dy(y, t):
   ...:     g, l, m = 9.82, 0.5, 0.1
   ...:     y1, y2, y3, y4 = y[0], y[1], y[2], y[3]
   ...:
   ...:     dy1 = 6.0/m/l**2*(2*y3 - 3*np.cos(y1-y2)*y4)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy2 = 6.0/m/l**2*(8*y4 - 3*np.cos(y1-y2)*y3)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy3 = -0.5*m*l**2*(+dy1*dy2*np.sin(y1-y2) + 3*(g/l)*np.sin(y1))
   ...:     dy4 = -0.5*m*l**2*(-dy1*dy2*np.sin(y1-y2) + 1*(g/l)*np.sin(y2))
   ...:
   ...:     return [dy1, dy2, dy3, dy4]
```

`scipy.integrate` : résolution d'équations différentielles ordinaires

Mouvement du double pendule

$$
\begin{aligned}
\dot{y}_1 &= \frac{6}{m\ell^2} \times \frac{2y_3 - 3\cos(y_1 - y_2)y_4}{16 - 9\cos^2(y_1 - y_2)} \\
\dot{y}_2 &= \frac{6}{m\ell^2} \times \frac{8y_4 - 3\cos(y_1 - y_2)y_3}{16 - 9\cos^2(y_1 - y_2)} \\
\dot{y}_3 &= -\frac{1}{2}m\ell^2 \left[\dot{y}_1\dot{y}_2\sin(y_1 - y_2) + 3\frac{g}{\ell}\sin y_1\right] \\
\dot{y}_4 &= -\frac{1}{2}m\ell^2 \left[-\dot{y}_1\dot{y}_2\sin(y_1 - y_2) + \frac{g}{\ell}\sin y_2\right]
\end{aligned}
$$

```
In [1]: def dy(y, t):
   ...:     g, l, m = 9.82, 0.5, 0.1
   ...:     y1, y2, y3, y4 = y[0], y[1], y[2], y[3]
   ...:
   ...:     dy1 = 6.0/m/l**2*(2*y3 - 3*np.cos(y1-y2)*y4)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy2 = 6.0/m/l**2*(8*y4 - 3*np.cos(y1-y2)*y3)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy3 = -0.5*m*l**2*(+dy1*dy2*np.sin(y1-y2) + 3*(g/l)*np.sin(y1))
   ...:     dy4 = -0.5*m*l**2*(-dy1*dy2*np.sin(y1-y2) + 1*(g/l)*np.sin(y2))
   ...:
   ...:     return [dy1, dy2, dy3, dy4]
```

scipy.integrate : résolution d'équations différentielles ordinaires

Mouvement du double pendule

```
In [1]: g, l, m = 9.82, 0.5, 0.1
In [2]: def dy(y, t):
   ...:     y1, y2, y3, y4 = y[0], y[1], y[2], y[3]
   ...:
   ...:     dy1 = 6.0/m/l**2*(2*y3 - 3*np.cos(y1-y2)*y4)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy2 = 6.0/m/l**2*(8*y4 - 3*np.cos(y1-y2)*y3)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy3 = -0.5*m*l**2*(+dy1*dy2*np.sin(y1-y2) + 3*(g/l)*np.sin(y1))
   ...:     dy4 = -0.5*m*l**2*(-dy1*dy2*np.sin(y1-y2) + 1*(g/l)*np.sin(y2))
   ...:
   ...:     return [dy1, dy2, dy3, dy4]
In [3]: # Conditions initiales
In [4]: y0 = [np.pi/4, np.pi/2, 0, 0]

In [5]: # Échantilonnage du temps
In [6]: t = np.linspace(0, 10, 250)

In [7]: # Résolution des équations différentielles
In [8]: from scipy.integrate import odeint
In [9]: y = odeint(dy, y0, t)
```

scipy.integrate : résolution d'équations différentielles ordinaires

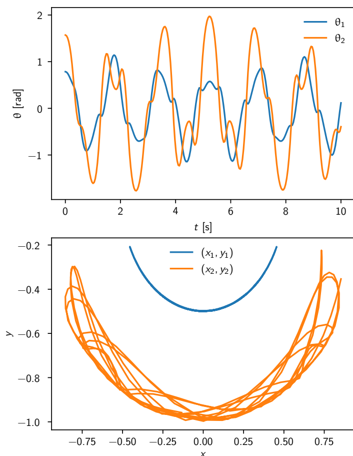
Mouvement du double pendule

```
In [1]: g, l, m = 9.82, 0.5, 0.1
In [2]: def dy(y, t):
   ...:     y1, y2, y3, y4 = y[0], y[1], y[2], y[3]
   ...:
   ...:     dy1 = 6.0/m/l**2*(2*y3 - 3*np.cos(y1-y2)*y4)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy2 = 6.0/m/l**2*(8*y4 - 3*np.cos(y1-y2)*y3)/(16 - 9*np.cos(y1-y2)**2)
   ...:     dy3 = -0.5*m*l**2*(+dy1*dy2*np.sin(y1-y2) + 3*(g/l)*np.sin(y1))
   ...:     dy4 = -0.5*m*l**2*(-dy1*dy2*np.sin(y1-y2) + 1*(g/l)*np.sin(y2))
   ...:
   ...:     return [dy1, dy2, dy3, dy4]
In [3]: # Conditions initiales
In [4]: y0 = [np.pi/4, np.pi/2, 0, 0]

In [5]: # Échantilonnage du temps
In [6]: t = np.linspace(0, 10, 250)

In [7]: # Résolution des équations différentielles
In [8]: from scipy.integrate import odeint
In [9]: y = odeint(dy, y0, t)
```

scipy.integrate : résolution d'équations différentielles ordinaires

Mouvement du double pendule

```
In [10]: t1, t2 = y[:, 0], y[:, 1]
In [11]: fig, ax = plt.subplots(2, 1, figsize=(5,7))
In [12]: ax[0].plot(t, t1, label=r"$\theta_1$")
In [13]: ax[0].plot(t, t2, label=r"$\theta_2$")
In [14]: ax[0].set(xlabel=r"$t$ [s]",
                   ylabel=r"$\theta$ [rad]")
In [15]: ax[0].legend()

In [16]: x1, y1 = l*np.sin(t1), -l*np.cos(t1)
In [17]: x2, y2 = x1 + l*np.sin(t2), y1 - l*np.cos(t2)
In [18]: ax[1].plot(x1, y1, label=r"$(x_1, y_1)$")
In [19]: ax[1].plot(x2, y2, label=r"$(x_2, y_2)$")
In [20]: ax[1].set(xlabel=r"$x$", ylabel=r"$y$")
In [21]: ax[1].legend()
```



scipy.integrate : résolution d'équations différentielles ordinaires

Mouvement du double pendule

```
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False,
                     xlim=(-1, 1), ylim=(-1.2, 0.2))
ax.grid()

pendule, = ax.plot([], [], "ok-", lw=2)
mvt1, = ax.plot([], [], c="C0")
mvt2, = ax.plot([], [], c="C1")
text = ax.text(0.05, 0.9, "", transform=ax.transAxes)

def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    pendule.set_data(thisx, thisy)
    mvt1.set_data(x1[:i], y1[:i])
    mvt2.set_data(x2[:i], y2[:i])
    text.set_text("temps = %.1f s" % (i*0.04))
    return pendule, mvt1, mvt2, text

ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
                              interval=25, blit=True)
ani.save("double_pendulum.mp4", fps=15)
```

`scipy.integrate` : résolution d'équations différentielles ordinaires

Oscillateur harmonique amorti

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + 2\zeta\omega_0 \frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

- On pose $p = \frac{\mathrm{d}x}{\mathrm{d}t}$

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

```
In [1]: def dy(y, t, zeta, w0):
   ...:     x, p = y[0], y[1]
   ...:
   ...:     dx = p
   ...:     dp = -2 * zeta * w0 * p - w0**2 * x
   ...:     return [dx, dp]
In [2]: y0 = [1.0, 0.0]
In [3]: t = np.linspace(0, 10, 1000)
In [4]: w0 = 2*np.pi*1.0
In [5]: from scipy.integrate import odeint
In [6]: y1 = odeint(dy, y0, t, args=(0.0, w0))
In [7]: y2 = odeint(dy, y0, t, args=(0.2, w0))
In [8]: y3 = odeint(dy, y0, t, args=(1.0, w0))
In [9]: y4 = odeint(dy, y0, t, args=(5.0, w0))
```

scipy.integrate : résolution d'équations différentielles ordinaires

Oscillateur harmonique amorti

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + 2\zeta\omega_0\frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

- On pose $p = \frac{\mathrm{d}x}{\mathrm{d}t}$

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$
$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

```
In [1]: def dy(y, t, zeta, w0):
   ...:     x, p = y[0], y[1]
   ...:
   ...:     dx = p
   ...:     dp = -2 * zeta * w0 * p - w0**2 * x
   ...:     return [dx, dp]
In [2]: y0 = [1.0, 0.0]
In [3]: t = np.linspace(0, 10, 1000)
In [4]: w0 = 2*np.pi*1.0
In [5]: from scipy.integrate import odeint
In [6]: y1 = odeint(dy, y0, t, args=(0.0, w0))
In [7]: y2 = odeint(dy, y0, t, args=(0.2, w0))
In [8]: y3 = odeint(dy, y0, t, args=(1.0, w0))
In [9]: y4 = odeint(dy, y0, t, args=(5.0, w0))
```

scipy.integrate : résolution d'équations différentielles ordinaires

Oscillateur harmonique amorti

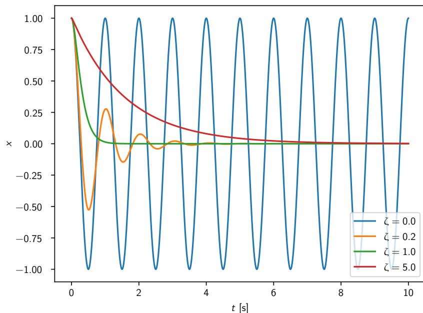
$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + 2\zeta\omega_0 \frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

- On pose $p = \frac{\mathrm{d}x}{\mathrm{d}t}$

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$

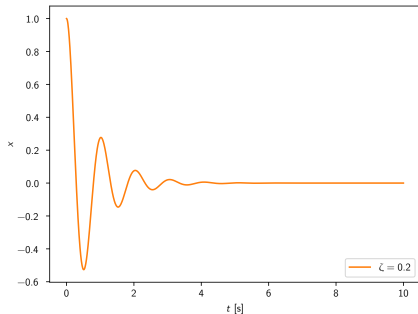
$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

```
In [1]: def dy(y, t, zeta, w0):
   ...:     x, p = y[0], y[1]
   ...:
   ...:     dx = p
   ...:     dp = -2 * zeta * w0 * p - w0**2 * x
   ...:     return [dx, dp]
In [2]: y0 = [1.0, 0.0]
In [3]: t = np.linspace(0, 10, 1000)
In [4]: w0 = 2*np.pi*1.0
In [5]: from scipy.integrate import odeint
In [6]: y1 = odeint(dy, y0, t, args=(0.0, w0))
In [7]: y2 = odeint(dy, y0, t, args=(0.2, w0))
In [8]: y3 = odeint(dy, y0, t, args=(1.0, w0))
In [9]: y4 = odeint(dy, y0, t, args=(5.0, w0))
```



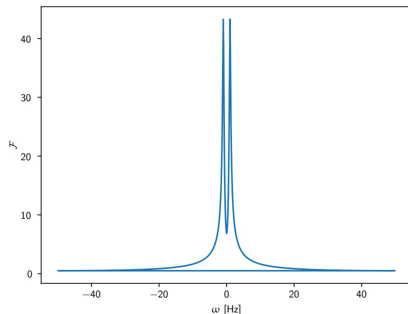
scipy.fftpack : transformations de Fourier

```
In [1]: from scipy.fftpack import fft, fftfreq

In [2]: F = fft(y2[:, 0])
In [3]: w = fftfreq(t.size, t[1]-t[0])
In [4]: plt.plot(w, np.abs(F))
In [5]: mask = w > 0
In [6]: plt.plot(w[mask], np.abs(F[mask]))
```



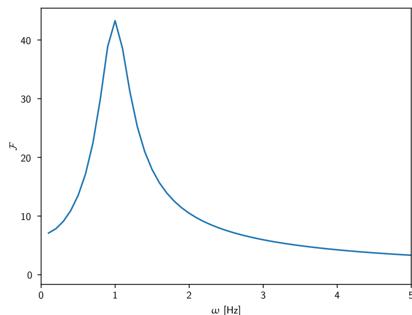
scipy.fftpack : transformations de Fourier

```
In [1]: from scipy.fftpack import fft, fftfreq

In [2]: F = fft(y2[:, 0])
In [3]: w = fftfreq(t.size, t[1]-t[0])
In [4]: plt.plot(w, np.abs(F))
In [5]: mask = w > 0
In [6]: plt.plot(w[mask], np.abs(F[mask]))
```



scipy.fftpack : transformations de Fourier

```
In [1]: from scipy.fftpack import fft, fftfreq

In [2]: F = fft(y2[:, 0])
In [3]: w = fftfreq(t.size, t[1]-t[0])
In [4]: plt.plot(w, np.abs(F))
In [5]: mask = w > 0
In [6]: plt.plot(w[mask], np.abs(F[mask]))
```



scipy.ndimage : traitement d'images

```
In [1]: import scipy.ndimage as ndimage

In [2]: img = ndimage.imread("../data/puzo_patrick.png")
In [3]: plt.imshow(img)
In [4]: plt.axis("off")
In [5]: img_flou = ndimage.gaussian_filter(img, sigma=10)
In [6]: fig, ax = plt.subplots(2,2)
In [7]: ax[1, 0].imshow(img[:,:,0], cmap=plt.cm.Reds)
In [8]: ax[0, 1].imshow(img[:,:,1], cmap=plt.cm.Greens)
In [9]: ax[1, 1].imshow(img[:,:,2], cmap=plt.cm.Blues)
```

```
In [1]: import scipy.ndimage as ndimage

In [2]: img = ndimage.imread("../data/puzo_patrick.png")
In [3]: plt.imshow(img)
In [4]: plt.axis("off")
In [5]: img_flou = ndimage.gaussian_filter(img, sigma=10)
In [6]: fig, ax = plt.subplots(2,2)
In [7]: ax[1, 0].imshow(img[:,:,0], cmap=plt.cm.Reds)
In [8]: ax[0, 1].imshow(img[:,:,1], cmap=plt.cm.Greens)
In [9]: ax[1, 1].imshow(img[:,:,2], cmap=plt.cm.Blues)
```

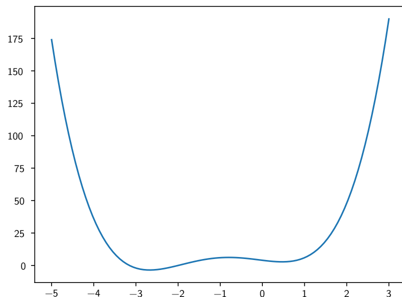
scipy.ndimage : traitement d'images

```
In [1]: import scipy.ndimage as ndimage

In [2]: img = ndimage.imread("../data/puzo_patrick.png")
In [3]: plt.imshow(img)
In [4]: plt.axis("off")
In [5]: img_flou = ndimage.gaussian_filter(img, sigma=10)
In [6]: fig, ax = plt.subplots(2,2)
In [7]: ax[1, 0].imshow(img[:,:,0], cmap=plt.cm.Reds)
In [8]: ax[0, 1].imshow(img[:,:,1], cmap=plt.cm.Greens)
In [9]: ax[1, 1].imshow(img[:,:,2], cmap=plt.cm.Blues)
```

scipy.optimize : recherche d'*extrema* d'une fonction

- L'objectif de l'optimisation est de trouver les *minima* (ou *maxima*) d'une fonction
- Domaine d'étude très actif en mathématiques/informatique notamment pour les problèmes multi-variables

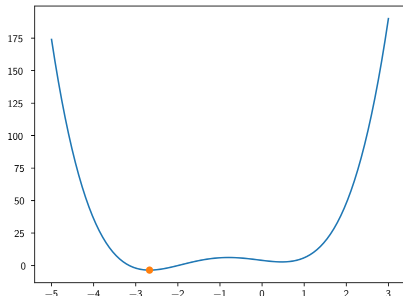
```
In [1]: def f(x):
   ...:     return 4*x**3 + (x-2)**2 + x**4
In [2]: from scipy.optimize import fmin
In [3]: fmin(f, x0=-2)
Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 15
         Function evaluations: 30
Out[3]: array([-2.67294922])
In [4]: fmin(f, x0=0)
Optimization terminated successfully.
         Current function value: 2.804988
         Iterations: 23
         Function evaluations: 46
Out[4]: array([ 0.469625])
```



`scipy.optimize` : recherche d'*extrema* d'une fonction

- L'objectif de l'optimisation est de trouver les *minima* (ou *maxima*) d'une fonction
- Domaine d'étude très actif en mathématiques/informatique notamment pour les problèmes multi-variables

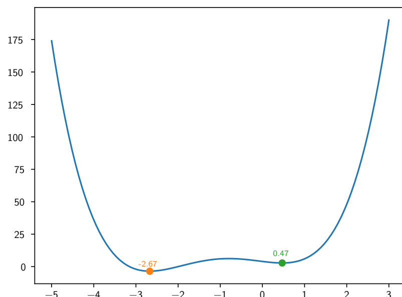
```
In [1]: def f(x):
   ...:     return 4*x**3 + (x-2)**2 + x**4

In [2]: from scipy.optimize import fmin
In [3]: fmin(f, x0=-2)
Optimization terminated successfully.
     Current function value: -3.506641
     Iterations: 15
     Function evaluations: 30
Out[3]: array([-2.67294922])
In [4]: fmin(f, x0=0)
Optimization terminated successfully.
     Current function value: 2.804988
     Iterations: 23
     Function evaluations: 46
Out[4]: array([ 0.469625])
```



`scipy.optimize` : recherche d'*extrema* d'une fonction

- L'objectif de l'optimisation est de trouver les *minima* (ou *maxima*) d'une fonction
- Domaine d'étude très actif en mathématiques/informatique notamment pour les problèmes multi-variables

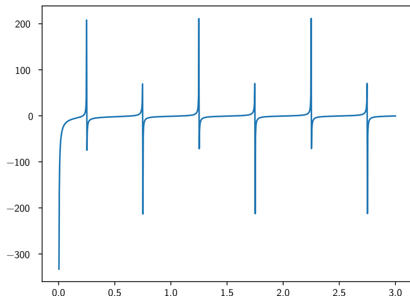
```
In [1]: def f(x):
   ...:     return 4*x**3 + (x-2)**2 + x**4

In [2]: from scipy.optimize import fmin
In [3]: fmin(f, x0=-2)
Optimization terminated successfully.
     Current function value: -3.506641
     Iterations: 15
     Function evaluations: 30
Out[3]: array([-2.67294922])

In [4]: fmin(f, x0=0)
Optimization terminated successfully.
     Current function value: 2.804988
     Iterations: 23
     Function evaluations: 46
Out[4]: array([ 0.469625])
```



`scipy.optimize` : recherche des racines d'une fonction

$$f(x_0) = tan(2\pi x_0) - \frac{1}{x_0} = 0$$

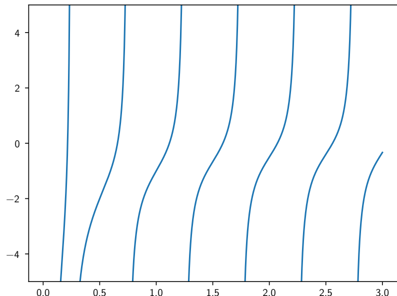
```
In [1]: def f(x):
   ...:     return np.tan(2*np.pi*x) - 1/x
In [2]: x = np.linspace(0, 3, 1000)
In [3]: plt.plot(x, f(x))
In [4]: y[abs(y) > 50] = np.nan
In [5]: plt.ylim(-5, +5)
In [6]: from scipy.optimize import fsolve
In [7]: fsolve(f, x0=0.1)
Out[7]: array([ 0.21612385])
In [8]: fsolve(f, x0=np.arange(0.1, 3, 0.5))
Out[8]:
array([ 0.21612385,  0.6574377 ,  1.116265  ,
        1.58938086,  2.071577  ,  2.55928414])
```



scipy.optimize : recherche des racines d'une fonction

$$f(x_0) = tan(2\pi x_0) - \frac{1}{x_0} = 0$$

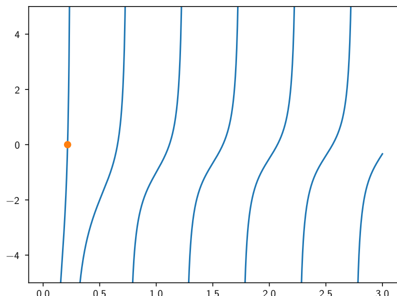
```
In [1]: def f(x):
   ...:     return np.tan(2*np.pi*x) - 1/x
In [2]: x = np.linspace(0, 3, 1000)
In [3]: plt.plot(x, f(x))

In [4]: y[abs(y) > 50] = np.nan
In [5]: plt.ylim(-5, +5)

In [6]: from scipy.optimize import fsolve
In [7]: fsolve(f, x0=0.1)
Out[7]: array([ 0.21612385])

In [8]: fsolve(f, x0=np.arange(0.1, 3, 0.5))
Out[8]:
array([ 0.21612385,  0.6574377 ,  1.116265  ,
        1.58938086,  2.071577  ,  2.55928414])
```



scipy.optimize : recherche des racines d'une fonction

$$f(x_0) = tan(2\pi x_0) - \frac{1}{x_0} = 0$$

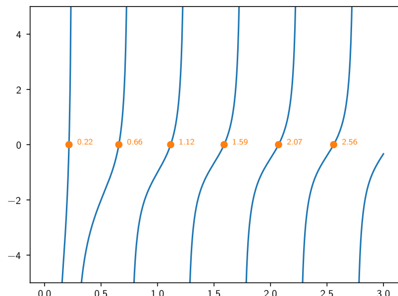
```
In [1]: def f(x):
   ...:     return np.tan(2*np.pi*x) - 1/x
In [2]: x = np.linspace(0, 3, 1000)
In [3]: plt.plot(x, f(x))

In [4]: y[abs(y) > 50] = np.nan
In [5]: plt.ylim(-5, +5)

In [6]: from scipy.optimize import fsolve
In [7]: fsolve(f, x0=0.1)
Out[7]: array([ 0.21612385])

In [8]: fsolve(f, x0=np.arange(0.1, 3, 0.5))
Out[8]:
array([ 0.21612385,  0.6574377 ,  1.116265  ,
        1.58938086,  2.071577  ,  2.55928414])
```



`scipy.optimize` : recherche des racines d'une fonction

$$f(x_0) = tan(2\pi x_0) - \frac{1}{x_0} = 0$$

```
In [1]: def f(x):
   ...:     return np.tan(2*np.pi*x) - 1/x
In [2]: x = np.linspace(0, 3, 1000)
In [3]: plt.plot(x, f(x))

In [4]: y[abs(y) > 50] = np.nan
In [5]: plt.ylim(-5, +5)

In [6]: from scipy.optimize import fsolve
In [7]: fsolve(f, x0=0.1)
Out[7]: array([ 0.21612385])

In [8]: fsolve(f, x0=np.arange(0.1, 3, 0.5))
Out[8]:
array([ 0.21612385,  0.6574377 ,  1.116265  ,
        1.58938086,  2.071577  ,  2.55928414])
```



`scipy.optimize` : ajustement d'un modèle/fonction à des données

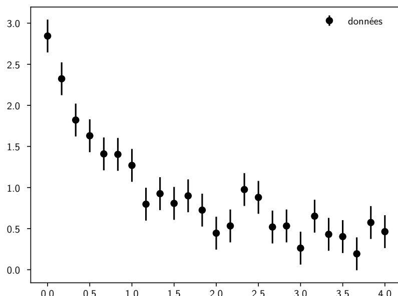
- L'ajustement consiste **à minimiser** une quantité caractérisant le niveau d'accord entre données expérimentales et modèle "théorique"
- Exemple de fonction à minimiser

$$\chi^2(p_0, \ldots, p_n) = \sum_i^N \frac{\left(y_i^{\text{data}} - y^{\text{model}}(x_i|p_0, \ldots, p_n)\right)^2}{\sigma^2_{y_i^{\text{data}}}}$$

où $p_0, \ldots, p_n$ sont les n paramètres du modèle.

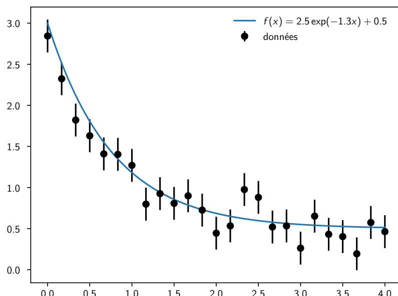
scipy.optimize : ajustement d'un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])
In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



scipy.optimize : ajustement d'un modèle/fonction à des données

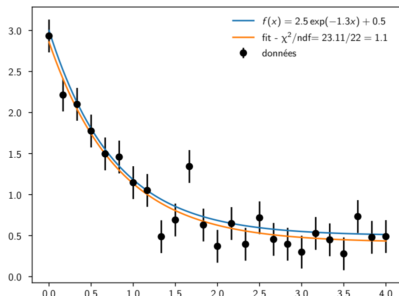
```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))

In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])

In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



scipy.optimize : ajustement d'un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
```

```
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])

In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



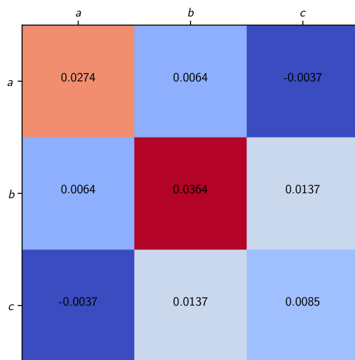
`scipy.optimize` : ajustement d'un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])
In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```

$$\begin{pmatrix} \sigma^2_{p_0} & \sigma_{p_0 p_1} & \cdots & \sigma_{p_0 p_n} \\ \sigma_{p_1 p_0} & \sigma^2_{p_1} & \cdots & \sigma_{p_1 p_n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p_n p_0} & \cdots & \cdots & \sigma^2_{p_n} \end{pmatrix}$$

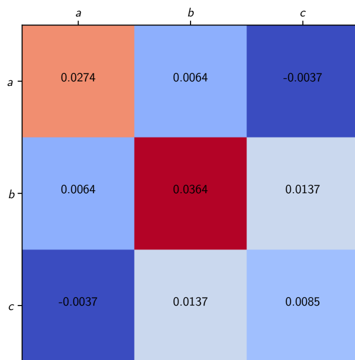
scipy.optimize : ajustement d'un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])
In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



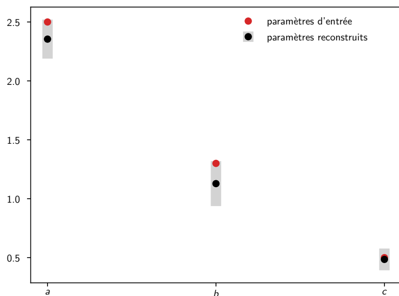
scipy.optimize : ajustement d’un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])
In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



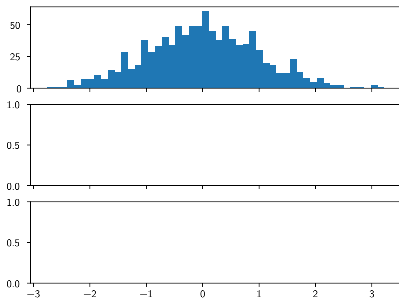
scipy.optimize : ajustement d'un modèle/fonction à des données

```
In [1]: def f(x, a, b, c):
   ...:     return a*np.exp(-b*x) + c

In [2]: a, b, c = 2.5, 1.3, 0.5
In [3]: xdata = np.linspace(0, 4, 25)
In [4]: dy = 0.2
In [5]: ydata = f(xdata, a, b, c) +  dy*np.random.randn(xdata.size)
In [6]: from scipy.optimize import curve_fit
In [7]: popt, pcov = curve_fit(f, xdata, ydata,
                               sigma=np.full_like(ydata, dy))
In [8]: popt
Out[8]: array([ 2.28680731,  1.21827861,  0.45424157])

In [9]: x = np.linspace(0, 4, 100)
In[10]: plt.plot(x, f(x, *popt))
In [11]: pcov
Out[11]:
array([[ 0.01681475,  0.00513406, -0.00182363],
       [ 0.00513406,  0.0254771 ,  0.00788938],
       [-0.00182363,  0.00788938,  0.00433422]])
In [12]: np.sqrt(np.diag(pcov))
Out[12]: array([ 0.16549342,  0.190719  ,  0.09236422])
```



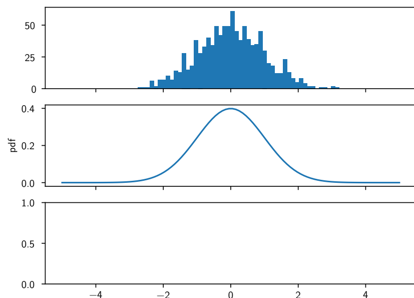
scipy.stats : distributions, fonctions & tests statistiques

```
In [1]: from scipy import stats

In [2]: normal = stats.norm()
In [3]: ax[0].hist(normal.rvs(1000), bins=50)
In [4]: x = np.linspace(-5, 5, 100)
In [5]: ax[1].plot(x, normal.pdf(x))

In [6]: ax[2].plot(x, normal.cdf(x))

In [7]: normal.mean(), normal.std(), normal.var()
Out[7]: (0.0, 1.0, 1.0)

In [8]: t_statistic, p_value = stats.ttest_ind(normal.rvs(1000),
                                               normal.rvs(1000))
In [9]: t_statistic, p_value
Out[9]: (0.026897392679505635, 0.97854425922146115)
```



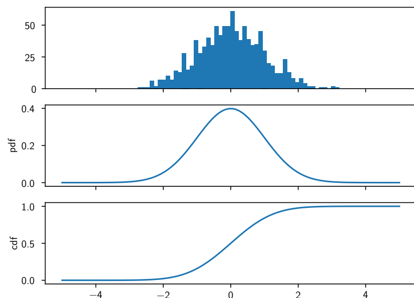
scipy.stats : distributions, fonctions & tests statistiques

```
In [1]: from scipy import stats

In [2]: normal = stats.norm()
In [3]: ax[0].hist(normal.rvs(1000), bins=50)
In [4]: x = np.linspace(-5, 5, 100)
In [5]: ax[1].plot(x, normal.pdf(x))
In [6]: ax[2].plot(x, normal.cdf(x))

In [7]: normal.mean(), normal.std(), normal.var()
Out[7]: (0.0, 1.0, 1.0)

In [8]: t_statistic, p_value = stats.ttest_ind(normal.rvs(1000),
                                               normal.rvs(1000))
In [9]: t_statistic, p_value
Out[9]: (0.026897392679505635, 0.97854425922146115)
```



scipy.stats : distributions, fonctions & tests statistiques

```
In [1]: from scipy import stats

In [2]: normal = stats.norm()
In [3]: ax[0].hist(normal.rvs(1000), bins=50)
In [4]: x = np.linspace(-5, 5, 100)
In [5]: ax[1].plot(x, normal.pdf(x))
In [6]: ax[2].plot(x, normal.cdf(x))
In [7]: normal.mean(), normal.std(), normal.var()
Out[7]: (0.0, 1.0, 1.0)

In [8]: t_statistic, p_value = stats.ttest_ind(normal.rvs(1000),
                                               normal.rvs(1000))
In [9]: t_statistic, p_value
Out[9]: (0.026897392679505635, 0.97854425922146115)
```



scipy.stats : distributions, fonctions & tests statistiques

```
In [1]: from scipy import stats

In [2]: normal = stats.norm()
In [3]: ax[0].hist(normal.rvs(1000), bins=50)
In [4]: x = np.linspace(-5, 5, 100)
In [5]: ax[1].plot(x, normal.pdf(x))
In [6]: ax[2].plot(x, normal.cdf(x))
In [7]: normal.mean(), normal.std(), normal.var()
Out[7]: (0.0, 1.0, 1.0)

In [8]: t_statistic, p_value = stats.ttest_ind(normal.rvs(1000),
                                               normal.rvs(1000))
In [9]: t_statistic, p_value
Out[9]: (0.026897392679505635, 0.97854425922146115)
```