Chapitre D4 – Python appliqué à la physique

Ce cours ne se substitue pas à un cours d'informatique mais il rassemble l'essentiel des compétences Python qui apparaissent dans le programme de physique. En particulier, les scripts Python présentés sont minimalistes (ils répondent spécifiquement à la tâche demandée) et ils ne se soucient guère de leur élégance ou leur complexité algorithmique.

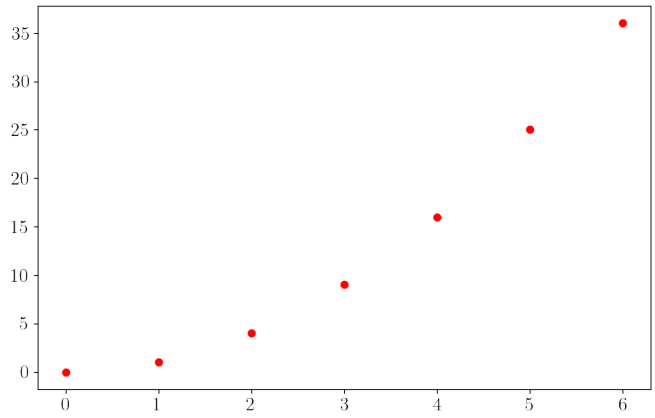
I) Outils graphiques

1) Représentation d'un nuage de points

</> Ligne 10 :

> `plt.plot(x, y, style)` trace `y` en fonction de `x` (`x` et `y` étant de même longueur). L'option `style` permet de contrôler l'apparence du graphe :
> - préciser une couleur (`r` → rouge, `b` → bleu, `g` → vert, `k` → noir) ;
> - préciser l'allure des marqueur (aucun marqueur, `o` → cercle, `.` → point, `+` → plus) ;
> - préciser l'allure des lignes (aucune ligne, `-` → continue, `--` → pointillée).

```
1  import matplotlib.pyplot as plt
2
3  # CRÉATION DES LISTES ------------------------------------------------------------
4
5  x = [0, 1, 2, 3, 4, 5, 6]
6  y = [0, 1, 4, 9, 16, 25, 36]
7
8  # AFFICHAGE GRAPHIQUE ------------------------------------------------------------
9
10 plt.plot(x, y, "ro")
11 plt.show()
```



2) Représentation d'une fonction

Le module `numpy` stocke les données dans un tableau numpy (`array` en anglais). Contrairement aux listes, les opérations élémentaires (`*`, `+`, `/`, `-`, etc) et les fonctions (`cos`, `sin`, `exp`, `log`, etc) peuvent s'appliquer sur des tableaux numpy. Le résultat renvoyé est un tableau numpy de même taille que le tableau utilisé.

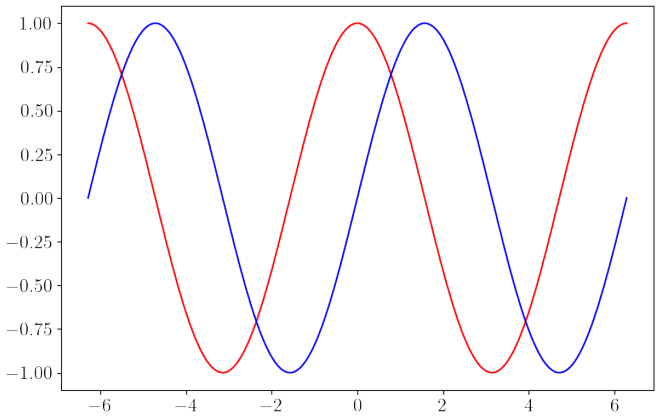
</> Ligne 6 :

> `np.linspace(a, b, N)` crée un tableau numpy contenant `N` valeurs régulièrement espacées et comprises entre `a` (inclus) et `b` (inclus).

</> Ligne 7 :

> `np.cos(u)` crée un tableau numpy de même taille que `u` où le n-ième élément est égal au cosinus du n-ième de `u`.

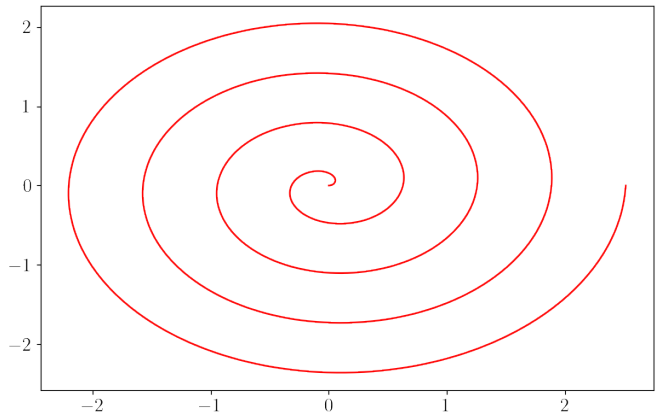
```python
import numpy as np
import matplotlib.pyplot as plt

# CRÉATION DES ARRAY ------------------------------------------------------------

x = np.linspace(-2*np.pi, 2*np.pi, 10000)
y1 = np.cos(x)
y2 = np.sin(x)

# AFFICHAGE GRAPHIQUE -----------------------------------------------------------

plt.plot(x, y1, "r-")
plt.plot(x, y2, "b-")
plt.show()
```



3) Représentation d'une courbe paramétrée

Une courbe paramétrée est une courbe où l'abscisse $x(t)$ et l'ordonnée $y(t)$ sont exprimés en fonction d'un paramètre t (en physique, le temps).

```python
import numpy as np
import matplotlib.pyplot as plt

# CRÉATION DES ARRAY ------------------------------------------------------------

t = np.linspace(0, 8*np.pi, 10000)
x = t/10 * np.cos(t)
y = t/10 * np.sin(t)

# AFFICHAGE GRAPHIQUE -----------------------------------------------------------

plt.plot(x, y, "r-")
plt.show()
```



II) Équations algébriques

1) Recherche d'une racine d'une fonction

L'objectif de cette partie est de résoudre une équation de type $g(x) = 0$ avec g une fonction quelconque.

Méthode dichotomique

Nous allons commencer par résoudre cette équation par méthode dichotomique. Les étapes clés de l'algorithme sont les suivantes.

- Trouver un intervalle $I = [a, b]$ où la fonction est strictement monotone et telle que $g(a)$ et $g(b)$ soient de signe opposé. Il existe alors une unique solution x_0 à l'équation $g(x) = 0$ dans I. Notons Δ la largeur de l'intervalle et m le milieu.
- Regarder le signe de $g(m)$. Si $g(a)$ et $g(m)$ sont de signe opposé, alors $x_0 \in [a, m]$. Sinon, $x_0 \in [m, b]$.
- Recommencer l'étape précédente autant que nécessaire. À chaque étape, la largeur Δ de l'intervalle est divisée par 2. Terminer l'algorithme lorsque Δ est inférieure à la précision ε souhaitée.

Application

Prenons l'exemple suivant. Cherchons la solution de l'équation transcendante :

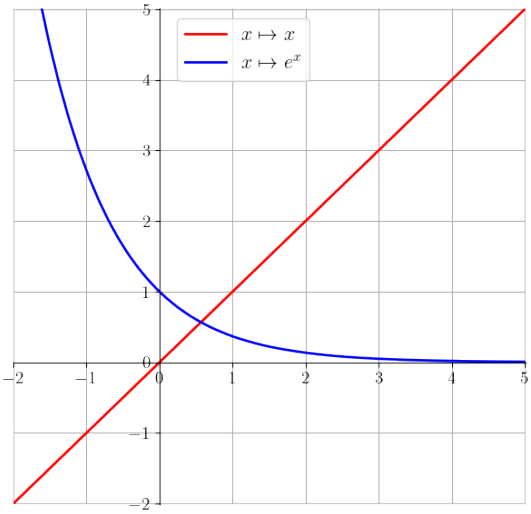
$$x = e^{-x}$$

Cette équation ne possède pas de solution analytique. Nous allons alors déterminer la valeur approchée de la solution avec une précision $\varepsilon = 10^{-9}$.

Pour cela, on cherche par dichotomie le zéro de la fonction :

$$g(x) = e^{-x} - x$$

D'après le graphe ci-contre, cette fonction est strictement monotone et la solution se trouve dan l'intervalle $I = [0, 1]$.




```python
import numpy as np

# FONCTION DICHOTOMIE ---------------------------------------------------------

def dichotmie(g, I, ε):
    while I[1] - I[0] > ε:          # Boucle tant que Δ > ε
        m = (I[0] + I[1]) / 2       # Calcul de m

        if g(I[0]) * g(m) < 0:      # g(a) et g(m) de signe opposé ?
            I = [I[0], m]           # Nouveau I = [a, m]
        else:                       # Sinon
            I = [m, I[1]]           # Nouveau I = [m, b]

    return (I[0] + I[1]) / 2        # On renvoie le m du dernier I de recherche

# EXEMPLE ---------------------------------------------------------------------

def g(x):                           # Définition de la fonction d'intérêt
    return np.exp(-x) - x

x0 = dichotmie(g, [0, 1], 1e-9)     # Appel de l'algorithme dichotomique

print("Solution de x = exp(-x) pour x0 = {:.9f}".format(x0))
```

```
Solution de x = exp(-x) pour x0 = 0.567143291
```

Fonction « `bisect` »

Il existe (évidemment) des fonctions déjà implémentées dans Python qui font le même travail.

</> Ligne 7 :

> `bisect(g, a, b)` renvoie la racine de la fonction `g` dans l'intervalle compris entre `a` et `b`, à condition que `g(a)` et `g(b)` soient de signe opposé.

```
import numpy as np
from scipy.optimize import bisect

def g(x):                       # Définition de la fonction d'intérêt
    return np.exp(-x) - x

x0 = bisect(g, 0, 1)            # Appel de la fonction bisect

print("Solution de x = exp(-x) pour x0 = {:.9f}".format(x0))
```

```
Solution de x = exp(-x) pour x0 = 0.567143290
```

2) Résolution d'une équation matricielle

L'objectif de cette partie est de résoudre un système linéaire de n équations indépendantes à n inconnues.

Considérons le système d'équations ci-dessous et mettons-le sous forme matricielle :

$$\begin{cases} 3x - 2y + z = 2 \\ x + y = -3 \\ -x + 5y - 2z = 0 \end{cases} \Leftrightarrow \underbrace{\begin{pmatrix} 3 & -2 & 1 \\ 1 & 1 & 0 \\ -1 & 5 & -2 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} 2 \\ -3 \\ 0 \end{pmatrix}}_{B} \Leftrightarrow \quad AX = B$$

L'objectif est donc de trouver X (ie. x, y et z) qui satisfait l'équation $AX = B$.

</> Ligne 9 :

> `np.linalg.solve(A, B)` renvoie la solution de l'équation `AX=B`, à condition que `A` soit inversible.

```
import numpy as np

A = np.array([[3, -2, 1],       # Matrice A
              [1, 1, 0],
              [-1, 5, -2]])

B = np.array([2, -3, 0])        # Vecteur B

X = np.linalg.solve(A, B)       # Appel de la fonction solve

print("Solution de AX = B pour X =", ["{:.3f}".format(x) for x in X])
```

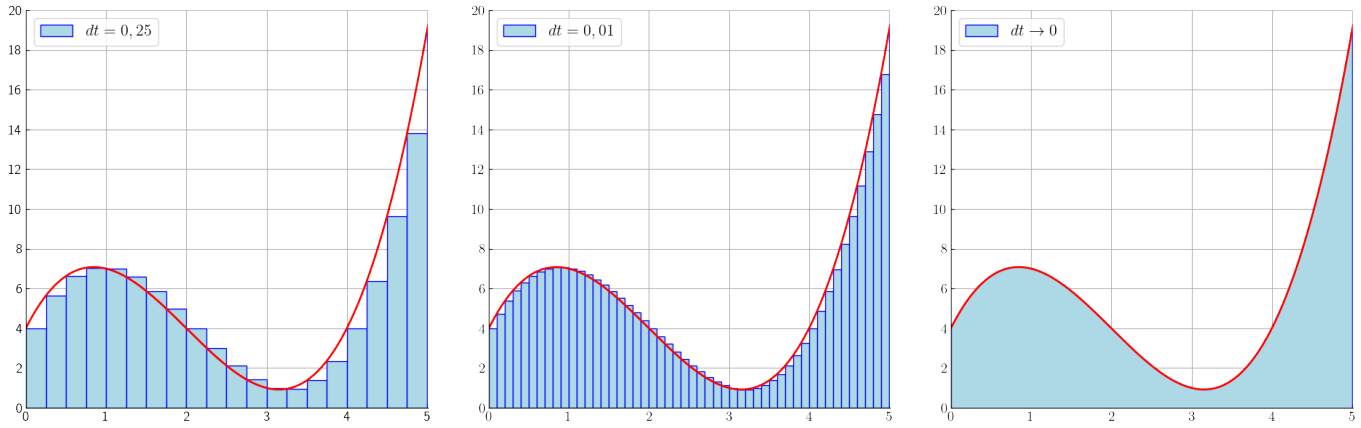
```
Solution de AX = B pour X = ['1.750', '-4.750', '-12.750']
```

III) Calcul infinimésimal

1) Calcul d'une intégrale

L'**intégrale** d'une fonction $f(t)$ entre deux bornes a et b correspond à l'aire sous la courbe entre les bornes a et b. Pour calculer cette aire, on somme (symbole $\int$ qui ressemble à un long « S ») les aires de rectangles de hauteur $f(t)$ et de largeur dt, pour t allant de a à b. L'aire d'un tel rectangle vaut : $d\mathcal{A} = f(t)\,dt$. Dans la limite où $dt \to 0$, la somme devient rigoureusement égale à l'aire sous la courbe.

Exemple avec l'intégrale la fonction $f(t) = t^3 - 6t^2 + 8t + 4$ entre 0 et 5 (illustrations ci-dessous).



</> Ligne 4 :

> `np.arange(a, b, p)` crée un tableau numpy contenant les valeurs comprises entre `a` (inclus) et `b` (exclu), régulièrement espacées du pas `p`.

</> Ligne 8 :

> `range(n)` crée la séquence des entiers de `0` (inclus) à `n` (exclu).

</> Ligne 8 :

> `len(u)` renvoie le nombre d'éléments contenus dans `u`.

```python
import numpy as np

dt = 1e-2                          # Choix du pas de temps dt
t = np.arange(0, 5, dt)            # Array des temps de 0 à 5 avec un pas dt
f = t**3 - 6*t**2 + 8*t + 4        # Fonction d'intérêt

A = 0                              # Initialisation de l'aire
for k in range(len(t) - 1):        # Boucle sur tous les rectangles
    A += f[k] * dt                 # Ajout de l'aire du rectangle n°k

print("Aire sous le courbe : A = {:.3f} pour dt = {:.0e}".format(A, dt))
```

```
Aire sous le courbe : A = 25.987 pour dt = 1e-02
Aire sous le courbe : A = 26.224 pour dt = 1e-03
Aire sous le courbe : A = 26.247 pour dt = 1e-04
Aire sous le courbe : A = 26.250 pour dt = 1e-05
```

Remarque : l'algorithme, dans la limite où $dt \to 0$, est bien en accord avec le calcul exact.

$$\mathcal{A} = \int_0^5 f(t)\,dt = \int_0^5 \Big(t^3 - 6t^2 + 8t + 4\Big)dt = \left[\frac{t^4}{4} - 2t^3 + 4t^2 + 4t\right]_0^5 = 26{,}25$$

2) Calcul d'un nombre dérivé

Le nombre dérivé d'une fonction en un point et donné par la limite du taux d'accroissement de la fonction en ce point.

$$f'(t) = \lim_{dt\to 0}\left(\frac{df}{dt}\right) = \lim_{dt\to 0}\left(\frac{f(t+dt) - f(t)}{dt}\right)$$

Calculons le nombre dérivé de la fonction $f(t) = t^3 - 6t^2 + 8t + 4$ en $t = 2$.

```python
def f(t):                                  # Fonction d'intérêt
    return t**3 - 6*t**2 + 8*t + 4

t = 2                                      # Choix du temps t
dt = 1e-2                                  # Choix du pas de temps dt
D = (f(t+dt) - f(t))/ dt                   # Calcul du nombre dérivé f'(t)

print("Nombre dérivé : f'({:.0f}) = {:.3f} pour dt = {:.0e}".format(t, D, dt))
```

```
Nombre dérivé : f'(2) = -3.000 pour dt = 1e+00
Nombre dérivé : f'(2) = -3.990 pour dt = 1e-01
Nombre dérivé : f'(2) = -4.000 pour dt = 1e-02
```

Remarque : l'algorithme, dans la limite où $dt \to 0$, est bien en accord avec le calcul exact.

$$f'(t) = 3t^2 - 12t + 8 \quad \Rightarrow \quad f'(2) = -4$$

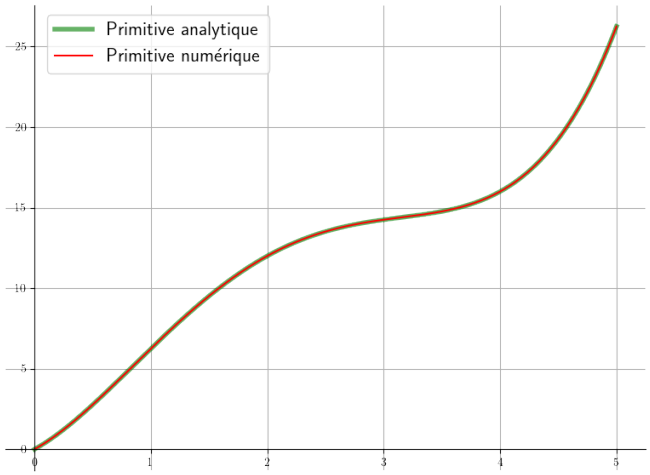
3) Généralisation : primitive et dérivée numérique

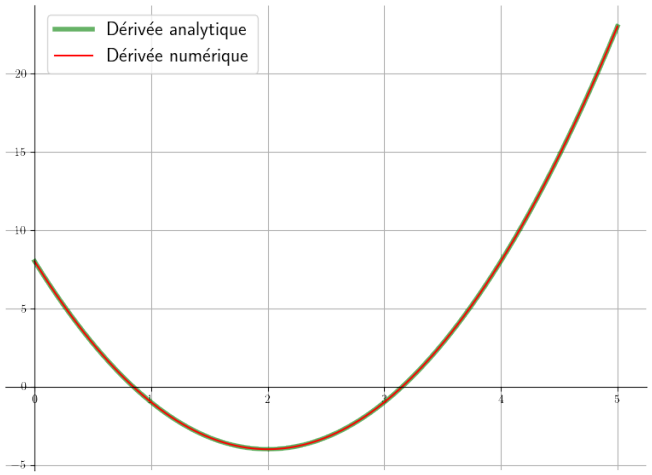
Il est possible d'adapter facilement les algorithmes précédent pour déterminer une primitive et la dérivée d'une fonction. Pour cela, il suffit de boucler les algorithmes pour toutes les valeurs de t.

</> Ligne 11 :

`u.append(x)` ajoute l'élément `x` en dernière position de la liste `u`.

```python
import numpy as np

dt = 1e-5                                   # Choix du pas de temps dt
t = np.arange(0, 5, dt)                     # Array des temps de 0 à 5 avec un pas dt
f = t**3 - 6*t**2 + 8*t + 4                 # Fonction d'intérêt

# CALCUL DE PRIMITIVE ---------------------------------------------------------

A = [0]                                     # Initialisation de l'intégrale
for k in range(len(t) - 1):                 # Boucle sur tous les rectangles
    A.append(A[k] + f[k] * dt)              # Ajout de l'aire du rectangle n°k

F = t**4/4 - 2*t**3 + 4*t**2 + 4*t          # Primitive (formule analytique)

# CALCUL DE DERIVÉE -----------------------------------------------------------

D = []                                      # Initialisation de la dérivée
for k in range(len(t) - 1):                 # Boucle sur toutes les valeurs de t
    D.append((f[k+1] - f[k]) / dt)          # Calcul du nombre dérivé

fp = 3*t**2 - 12*t + 8                      # Dérivée (formule analytique)

# AFFICHAGE GRAPHIQUE ---------------------------------------------------------

""" Les paramètres de l'affichage graphique sont masqués pour plus de concision """
```





On constate de nouveau que si dt est suffisamment petit, alors la primitive et la dérivée calculées numériquement coïncident visuellement avec les fonctions analytiques.

IV) Équations différentielles

1) ED d'odre 1

Soit une ED de la forme suivante, que l'on cherche à résoudre numériquement.

$$\frac{df}{dt} + \frac{f(t)}{\tau} = \frac{g(t)}{\tau} \quad \text{avec :} \quad g(t) \text{ connue}$$

Méthode d'Euler

La méthode d'Euler consiste à faire l'approximation suivante, où τ représente le temps caractéristique de variation de $f(t)$.

$$f'(t) \underset{dt \ll \tau}{\simeq} \frac{f(t+dt) - f(t)}{dt}$$

On peut alors isoler $f(t+dt)$:

$$\frac{df}{dt} + \frac{f(t)}{\tau} = \frac{g(t)}{\tau} \quad \Rightarrow \quad \frac{f(t+dt) - f(t)}{dt} + \frac{f(t)}{\tau} = \frac{g(t)}{\tau} \quad \Rightarrow \quad \boxed{f(t+dt) = f(t) + \Big(g(t) - f(t)\Big)\frac{dt}{\tau}}$$

Grâce à cette relation il est possible, connaissant l'état du système à l'instant t, de déterminer son état à l'instant d'après $(t+dt)$. Connaissant l'état initial $(t=0)$, il est alors possible, de proche en proche, de connaître l'état du système $\forall\ t = n \times dt$ avec $n \in \mathbb{N}$.

Les listes contenant les valeurs de t et f sont construites de manière récurrente, chaque terme étant défini à partir du précédent.

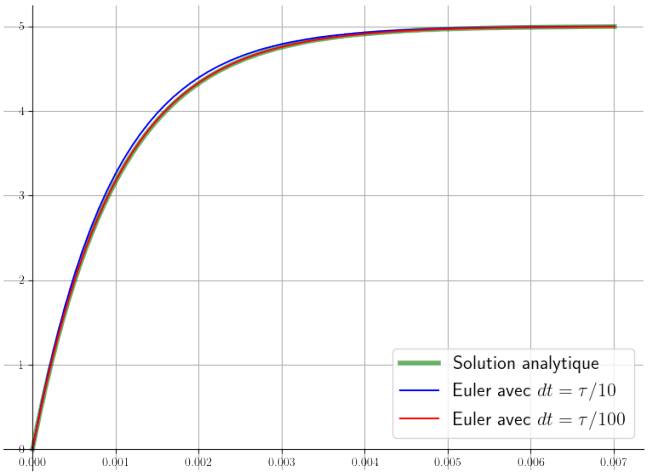
$$\begin{cases} t[n+1] = t[n] + dt & \text{avec :} \quad t[0] = 0 \\ f[n+1] = f[n] + \Big(g[n] - f[n]\Big)\dfrac{dt}{\tau} & \text{avec :} \quad f[0] = f_0 \end{cases}$$

Application

Prenons l'exemple du circuit RC soumis à un échelon de tension. La tension aux bornes du condensateur vérifie l'ED :

$$\frac{du}{dt} + \frac{u(t)}{\tau} = \frac{E}{\tau} \quad \Rightarrow \quad \boxed{u[n+1] = u[n] + \Big(E - u[n]\Big)\frac{dt}{\tau} \quad \text{avec :} \quad \begin{cases} dt \ll \tau \\ u(0) = 0 \end{cases}}$$

```python
# PARAMÈTRES EXPERIMENTAUX ------------------------------------------------------------

E = 5
R = 1e3
C = 1e-6
tau = R*C

# MÉTHODE D'EULER ---------------------------------------------------------------------

dt = tau/100          # Pas de temps dt
t_max = 7*tau         # Temps de fin de simulation
t = [0]               # CI sur le temps
u = [0]               # CI sur la tension

while t[-1] < t_max:                          # Boucle dan que t < t_max
    t.append(t[-1] + dt)                      # Formule de récurence sur t
    u.append(u[-1] + (E-u[-1])*dt/tau)        # Formule de récurence sur u

# AFFICHAGE GRAPHIQUE -----------------------------------------------------------------

""" Les paramètres de l'affichage graphique sont masqués sont plus de concision """
```



On constate que la solution numérique coïncide visuellement avec la solution analytique pour $dt = \tau/100$. Ce n'est pas le cas $dt = \tau/10$.

Fonction « `odeint` »

Il existe (évidemment) des fonctions déjà implémentées dans Python qui font le même travail.

</> Ligne 9 :

> `deriv(y, t)` renvoie la dérivée de `y` en fonction de `y` et `t`. L'argument `t` est à indiquer même si `t` n'apparaît pas explicitement dans la fonction.

</> Ligne 14 :

> `odeint(deriv, CI, t)` renvoie la solution de l'équation différentielle donnée par la fonction `deriv`, avec pour condition initiale `CI`, et évaluée aux temps `t`.
>
> `y` et `CI` doivent être de même dimension (flottant ou liste).

```
1  import numpy as np
2  from scipy.integrate import odeint
3
4  E = 5
5  R = 1e3
6  C = 1e-6
7  tau = R*C
8
9  def deriv(y, t):                    # Fonction qui renvoie y'(t) connaissant y(t)
10     dydt = (E - y)/tau
11     return dydt
12
13 t = np.linspace(0, 7*tau, 1000)     # Array des temps
14 u = odeint(deriv, 0, t)             # Appel de la fonction odeint
```

Je ne joins pas le graphe car la solution numérique coïncide visuellement avec la solution analytique (cf. graphe précédent).

2) ED d'ordre supérieur ou égal à 2

Pour résoudre avec `odeint` une équation différentielle d'ordre n, il faut la transformer en n équations différentielles couplées d'ordre 1. Les fonctions sont alors stockées dans une liste de taille n.

Prenons l'exemple du pendule simple avec frottements fluides. L'angle $\theta(t)$ du pendule est donné par l'équation différentielle :

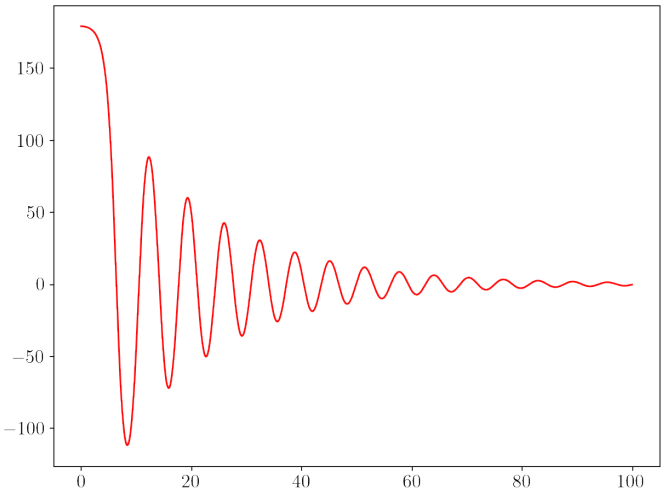
$$\ddot{\theta} + \frac{\omega_0}{Q}\,\dot{\theta} + \omega_0^2\,\sin(\theta) = 0$$

On pose le vecteur Y ci-dessous (où $\omega = \dot{\theta}$) et on détermine l'expression de $\dot{\mathrm{Y}}$ grâce à l'ED :

$$\mathrm{Y} = \begin{bmatrix} \theta \\ \omega \end{bmatrix} \begin{matrix} \leftarrow \mathrm{Y}[0] \\ \leftarrow \mathrm{Y}[1] \end{matrix} \quad \Rightarrow \quad \dot{\mathrm{Y}} = \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega \\ -\frac{\omega_0}{Q}\,\omega - \omega_0^2\,\sin(\theta) \end{bmatrix} = \begin{bmatrix} \mathrm{Y}[1] \\ -\frac{\omega_0}{Q} \times \mathrm{Y}[1] - \omega_0^2\,\sin(\mathrm{Y}[0]) \end{bmatrix}$$

On obtient à ce stade un système de deux équations différentielles couplées d'ordre 1, portant sur les fonctions $\theta(t)$ et $\omega(t)$.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# RÉSOLUTION ODEINT ------------------------------------------------------------

ω0 = 1
Q = 10

def deriv(y, t):
    dydt = [y[1], -ω0/Q * y[1] - ω0**2 * np.sin(y[0])]
    return dydt

t = np.linspace(0, 100, 10000)  # Array des temps
CI = [179*np.pi/180, 0]         # CI : θ(0) = 179° en rad et ω(0) = 0
Y = odeint(deriv, CI, t)        # Y → array 10000 lignes x 2 colonnes

θ = Y[:, 0] * 180/np.pi         # θ en ° → toutes les lignes (:) de la colonne n°0
ω = Y[:, 1]                     # ω → toutes les lignes (:) de la colonne n°1

# AFFICHAGE GRAPHIQUE ----------------------------------------------------------

plt.plot(t, θ, "r-")
plt.show()
```



On observe sur les temps courts (première oscillation) le comportement non linéaire du fait de la présence de $\sin(\theta)$.

Sur les temps longs, les oscillations sont de faible amplitude. Il est alors possible de faire l'approximation $\sin(\theta) \simeq \theta$, et on retombe ainsi sur la solution de l'oscillateur amorti (en régime pseudo-périodique) vu en cours.

V) Traitement de données numériques

1) Variable aléatoire

La génération d'une variable aléatoire est utile en physique pour faire des simulations de Monte-Carlo (cf. partie correspondante du cours sur les incertitudes).

Loi uniforme

Une loi uniforme de valeur moyenne μ et de demi-largeur Δ correspond à une densité de probabilité uniforme dans l'intervalle $[\mu - \Delta, \mu + \Delta]$ et nulle hors de cet intervalle (fonction porte).

</> Ligne 10 :

> `np.random.uniform(a, b, N)` génère un tirage de `N` valeurs aléatoires selon une loi uniforme entre `a` et `b`.

</> Ligne 19 :

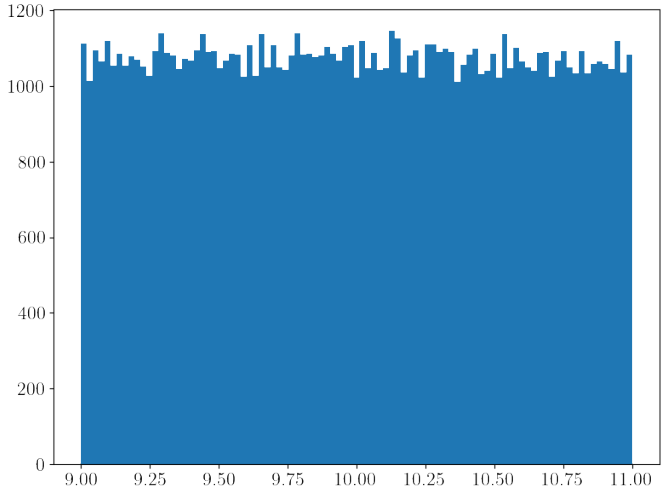
`np.average(u)` calcule la valeur moyenne du tableau numpy `u`.

</> Ligne 20 :

`np.std(u, ddof=1)` calcule l'écart-type du tableau numpy `u`.

```python
import numpy as np
import matplotlib.pyplot as plt

# TIRAGE ALÉATOIRE ------------------------------------------------------------

m = 10                                  # Valeur moyenne
Δ = 1                                   # Demi-largeur
N = 100000                              # Nombre de tirages

z = np.random.uniform(m-Δ, m+Δ, N)      # Tirage uniforme des {z_i}

# ANALYSE STATISTIQUE ---------------------------------------------------------

z_moy = np.average(z)                   # Valeur moyenne des {z_i}
u_z = np.std(z, ddof=1)                 # Écart-type (= incertitude-type) des {z_i}

# AFFICHAGE DES RÉSULTATS -----------------------------------------------------

print("Valeur moyenne : <z> = {:.3f}".format(z_moy))
print("Écart-type : u(z) = {:.3f}".format(u_z))

plt.hist(z, bins="rice")    # Tracé de l'histogramme
plt.show()
```

```
Valeur moyenne : <z> = 10.000
Écart-type : u(z) = 0.577
```



L'écart-type d'une loi uniforme (comme on peut le constater avec ce code), vaut :

$$\sigma = \frac{\Delta}{\sqrt{3}} = 0{,}577 \quad \text{avec :} \quad \Delta = 1$$

Cette loi est utilisée dans le traitement de type B des incertitudes.

Loi normale

Une loi normale de valeur moyenne μ et d'écart-type σ correspond à une densité de probabilité donnée par une courbe de Gauss (ou gaussienne) d'équation :

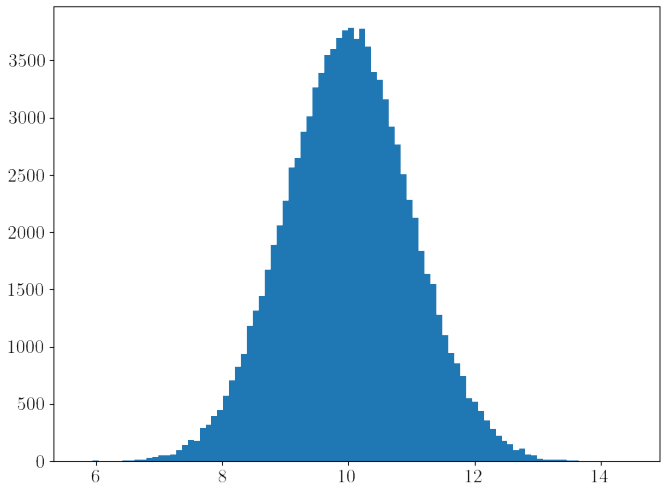
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

</> Ligne 10 :

`np.random.normal(`μ`, `σ`, N)` génère un tirage de `N` valeurs aléatoires selon une loi normale de valeur moyenne μ et d'écart-type σ.

```
import numpy as np
import matplotlib.pyplot as plt

# TIRAGE ALÉATOIRE ------------------------------------------------------------

m = 10                                  # Valeur moyenne
σ = 1                                   # Écart-type
N = 100000                              # Nombre de tirages

z = np.random.normal(m, σ, N)           # Tirage des {z_i} loi normale

# ANALYSE STATISTIQUE ---------------------------------------------------------

z_moy = np.average(z)                   # Valeur moyenne des {z_i}
u_z = np.std(z, ddof=1)                 # Écart-type (= incertitude-type) des {z_i}

# AFFICHAGE DES RÉSULTATS -----------------------------------------------------

print("Valeur moyenne : <z> = {:.3f}".format(z_moy))
print("Écart-type : u(z) = {:.3f}".format(u_z))

plt.hist(z, bins="rice")    # Tracé de l'histogramme
plt.show()
```

```
Valeur moyenne : <z> = 10.000
Écart-type : u(z) = 1.000
```



Cette loi est très souvent rencontrée lorsque l'on répète un grand nombre de fois une même expérience (traitement de type A des incertitudes).

2) Régression linéaire

L'objectif est de modéliser un nuage de points expérimentaux par une droite affine.

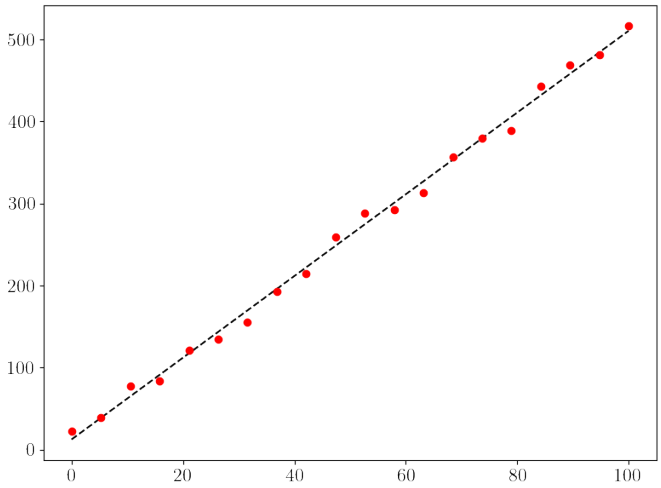
</> Ligne 12 :

`a, b = np.polyfit(x, y, 1)` stocke dans les variables `a` et `b` le résultat de la régression affine $y = ax + b$.

```
import numpy as np
import matplotlib.pyplot as plt

# MESURES EXPÉRIMENTALES FICTIVES ---------------------------------------------

x = np.linspace(0, 100, 20)             # Valeurs de x fictives
y = 5*x + 8                             # Valeurs de y fictives : a = 5 et b = 8
y += np.random.normal(0, 10, 20)        # Ajout d'une variabilité fictive dans y
```

```
9
10 # RÉGRESSION AFFINE ------------------------------------------------------------------
11
12 a, b = np.polyfit(x, y, 1)              # Régression affine y = a*x + b
13
14 # AFFICHAGE DES RÉSULTATS ------------------------------------------------------------
15
16 print("Paramètres de régression : a = {:.3f} et b = {:.3f}".format(a, b))
17
18 plt.plot([min(x), max(x)], [a*min(x)+b, a*max(x)+b], 'k--')  # Droite de régression
19 plt.plot(x, y, "ro")                                        # Données expérimentales
20 plt.show
```

```
1 Paramètres de régression : a = 4.983 et b = 12.399
```



Rappel : ce qu'il faut faire et conclure en TP.

Tracer sur le même graphique les données expérimentales par des points et la droite de régression par un trait.

Les données expérimentales sont « proches » de la droite de régression et « aléatoirement répartis » au-dessus et en-dessous de la droite. Cela valide *qualitativement* le modèle affine.

Il faut tracer les écart-normalisés pour valider *quantitativement* le modèle affine (cf. partie correspondante du cours sur les incertitudes).

Remarque : le paramètre `a` est toujours plus fiable que `b`.