Python pour la physique

Ce cours contient une série de scripts Python illustrant l'ensemble des notions à maîtriser pour le cours de sciences physiques de MPSI.

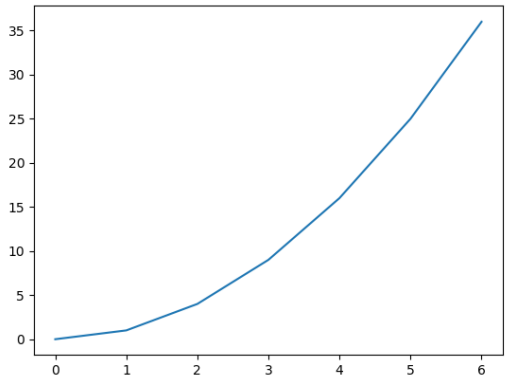
I - Les graphiques

I.1 - Nuage de points

Pour tracer un graphique, il faut impérativement avoir deux listes de même longueur.

```
1  import matplotlib.pyplot as plt  # Module pour tracer les graphiques
2
3  # Création des deux listes de même longueur
4  x = [0, 1, 2, 3, 4, 5, 6]
5  y = [0, 1, 4, 9, 16, 25, 36]
6
7  fig, ax = plt.subplots()      # Création d'une fenêtre vide pour afficher le graphique
8  ax.plot(x, y)                 # Tracé de la courbe : y en fonction de x
```

Le résultat affiché par ce code est le suivant. Ce n'est pas très joli, mais c'est un début !



Dans Python, tout est paramétrable. Il faut seulement lui indiquer quoi faire.

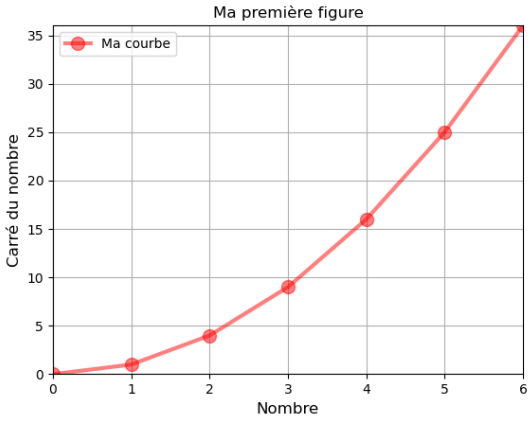
- Personnalisation de la courbe dans la fonction « plot() »

Nom	**Valeur**	**Explication**
c	{'r', 'b', 'k', 'g', 'yellow', …}	Couleur de la courbe (ou *color*)
ls	{'-', '--', '-.', ':', '', …}	Style de la courbe (ou *linestyle*)
lw	Float	Épaisseur de la courbe (ou *linewidth*)
marker	{'o', '.', 's', '+', 'x', …}	Style des points
ms	Float	Épaisseur des points (ou *markersize*)
alpha	$[0, 1]$	Transparence de la courbe
label	String	Nom de la courbe à indiquer dans la légende

- Ajout des noms des axes, du titre, de la légende.

```
1  import matplotlib.pyplot as plt
2
3  x = [0, 1, 2, 3, 4, 5, 6]
4  y = [0, 1, 4, 9, 16, 25, 36]
5
```

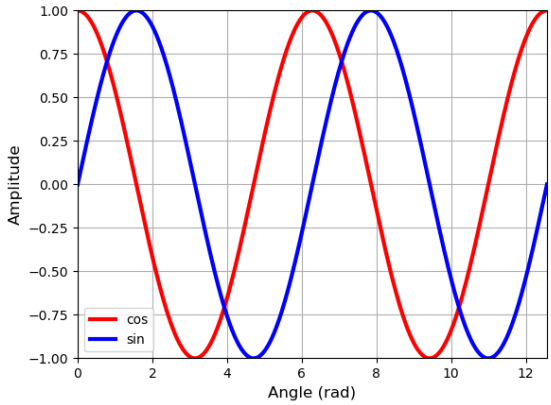
```
fig, ax = plt.subplots()
ax.plot(x, y, c='r', ls='-', lw=3, marker='o', ms=10, alpha=0.5, label='Ma courbe')

ax.set_title("Ma première figure", fontsize=12)      # Titre de la figure (il est possible de préciser la taille de la police)
ax.set_xlabel("Nombre", fontsize=12)                 # Titre de l'axe x
ax.set_ylabel("Carré du nombre", fontsize=12)        # Titre de l'axe y
ax.set_xlim(0, 6)                                    # Valeurs maximales de l'axe des x
ax.set_ylim(0, 36)                                   # Valeurs maximales de l'axe des y
ax.legend()                                          # Ajoute une légende
ax.grid(True)                                        # Ajoute une grille de fond
```



I.2 - Fonction

Le module « numpy » apporte l'ensemble des fonctions usuelles et stocke les données dans des « arrays » (tableaux multidimensionnels), ce qui permet d'effectuer des calculs mathématiques de haut niveau.

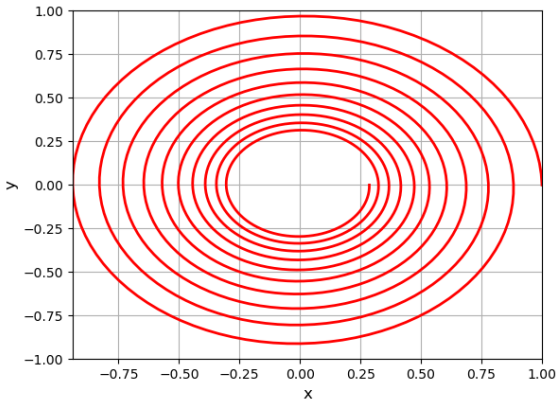
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 1000) # Création d'un array de 1000 points équitablement espacés entre 0 et 4π
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots()

ax.plot(x, y1, c='r', ls='-', lw=3, label='cos')
ax.plot(x, y2, c='b', ls='-', lw=3, label='sin')

ax.set_xlabel("Angle (rad)", fontsize=12)
ax.set_ylabel("Amplitude", fontsize=12)
ax.set_xlim(x[0], x[-1]) # x[0] est la première valeur de l'array x, et x[-1] est la dernière valeur
ax.set_ylim(-1, 1)
ax.legend()
ax.grid(True)
```



I.3 - Courbe paramétrée

Une courbe paramétrée est une courbe où l'abscisse et l'ordonnée sont exprimés en fonction du temps : $x(t)$ et $y(t)$.

```
import matplotlib.pyplot as plt
import numpy as np
plt.close('all') # Permet de fermer toutes les anciennes fenêtres

τ = 8              # Constante de temps de l'amortissement
T = 1              # Période de la rotation
ω = 2*np.pi/T      # Pulsation

t = np.linspace(0, 10, 1000)
x = np.exp(-t/τ) * np.cos(ω*t)
y = np.exp(-t/τ) * np.sin(ω*t)

fig, ax = plt.subplots()
ax.plot(x, y, c='r', ls='-', lw=2)

ax.set_xlabel("x", fontsize=12)
ax.set_ylabel("y", fontsize=12)
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
ax.grid(True)
```



I.4 - Régression linéaire

Dans cette partie, on cherche à visualiser si deux variables x et y sont reliées par une fonction affine, c'est-à-dire par la relation : $y \simeq a\,x + b$, avec a et b des constantes à déterminer. Pour cela, on va réaliser une régression affine, que l'on appelle plus communément (par abus de langage) une régression linéaire.

```
import matplotlib.pyplot as plt
import numpy as np
plt.close('all') # Permet de fermer toutes les anciennes fenêtres

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = np.array([3, 4.2, 4.9, 6.2, 6.8, 8, 8.8, 10.3, 10.8, 12, 13.1])
# A l'aide de np.array(), les listes sont directement converties en array

a, b = np.polyfit(x, y, 1) # Régression linéaire (= polynomiale d'ordre 1) : y = a x + b

# Affichage des paramètres dans la console
print('Résultat de la régression linéaire :')
print('a =', a)
print('b =', b)

fig, ax = plt.subplots()

```

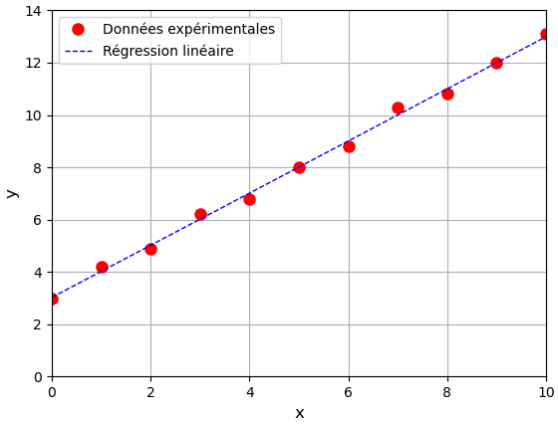
```
ax.plot(x, y, c='r', ls='', marker='o', ms=8, label='Données expérimentales')
ax.plot(x, a*x+b, c='b', ls='--', lw=1, label='Régression linéaire')

ax.set_xlabel("x", fontsize=12)
ax.set_ylabel("y", fontsize=12)
ax.set_xlim(0, 10)
ax.set_ylim(0, 14)
ax.legend()
ax.grid(True)
```

Il se passe deux choses. Premièrement, le graphique ci-contre s'affiche.

Deuxièmement, il apparait dans la console :

```
Résultat de la régression linéaire :
a = 0.9963636363636362
b = 3.0272727272727247
```

Cela signifie que la « meilleure » droite pour modéliser le nuage de point possède une pente $a \simeq 1{,}0$ et une ordonnée à l'origine $b \simeq 3{,}0$.

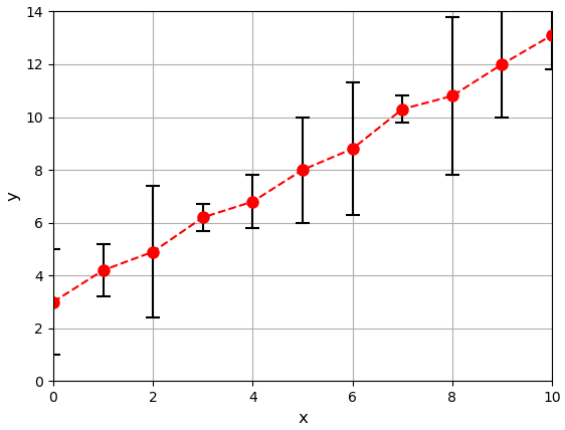


I.5 - Barres d'erreur

Afin d'ajouter des barres d'erreurs sur un graphique, il faut remplacer la fonction « plot » par la fonction « errorbar ». Une série de nouveaux paramètres permettent de personnaliser ces barres d'erreur.

Nom	**Valeur**	**Explication**
xerr	Liste ou Array	Barres d'erreur selon x
yerr	Liste ou Array	Barres d'erreur selon y
ecolor	{'r', 'b', 'k', 'g', 'yellow', …}	Couleur des barres d'erreur
elinewidth	Float	Épaisseur des barres d'erreur
capsize	Float	Taille des casquettes (barres perpendiculaires aux extrémités des barres d'erreur)
capthick	Float	Épaisseur des casquettes

```
import matplotlib.pyplot as plt
import numpy as np
plt.close('all')

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = np.array([3, 4.2, 4.9, 6.2, 6.8, 8, 8.8, 10.3, 10.8, 12, 13.1])

uy = np.array([2, 1, 2.5, 0.5, 1, 2, 2.5, 0.5, 3, 2, 1.3]) # Incertitude-type sur y

fig, ax = plt.subplots()

ax.errorbar(x, y, yerr=uy, c='r', ls='--', marker='o', ms=8, ecolor='k', elinewidth=1.5, capsize=5, capthick=1.5)

ax.set_xlabel("x", fontsize=12)
ax.set_ylabel("y", fontsize=12)
ax.set_xlim(0, 10)
ax.set_ylim(0, 14)
ax.grid(True)
```



II - Équations différentielles

II.1 - Méthode d'Euler explicite

La méthode d'Euler explicite consiste à remplacer une dérivée $y'(t)$ par la forme approchée suivante :

$$y'(t) \simeq \frac{y(t+dt) - y(t)}{dt}$$

L'égalité est stricte lorsque $dt \to 0$.

Prenons l'exemple de la charge d'un condensateur. L'équation différentielle vérifiée par la tension aux bornes du condensateur vaut :

$$\frac{du_c}{dt} + \frac{u_c}{\tau} = \frac{E}{\tau}$$

Avec la méthode d'Euler, cette équation devient :

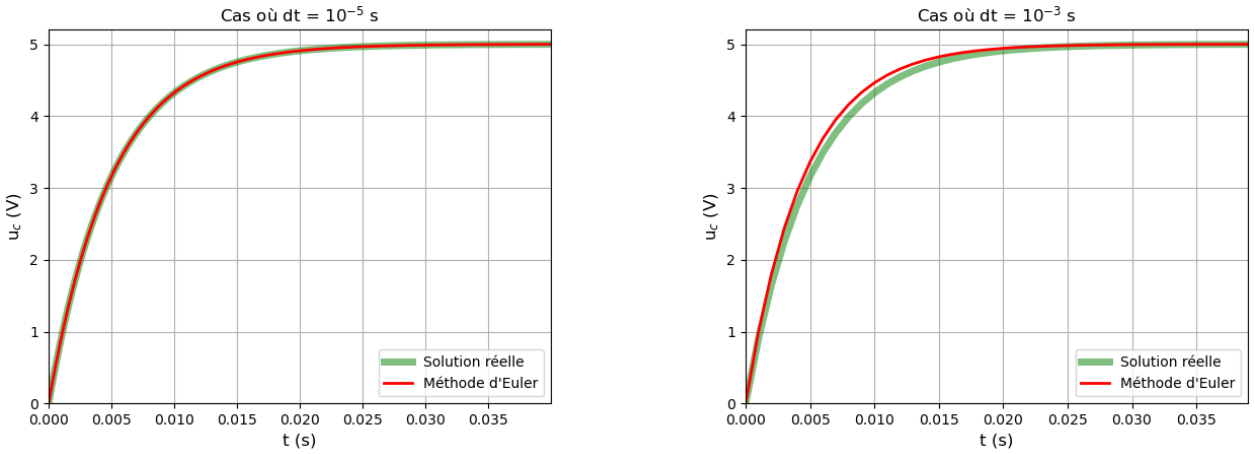
$$\frac{u_c(t+dt) - u_c(t)}{dt} + \frac{u_c(t)}{\tau} = \frac{E}{\tau} \quad \Rightarrow \quad \boxed{u_c(t+dt) = u_c(t) + \frac{E - u_c(t)}{\tau} dt}$$

La connaissance de $u_c(t)$ permet donc de connaître la valeur de $u_c(t+dt)$. Connaissant de plus la condition initiale $u_c(0)$, on peut ainsi, de proche en proche, connaître $u_c(t)$ pour tout t.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  plt.close('all')
4
5  # Paramètres ------------------------------------------------------------
6  E = 5              # Tension d'alimentation (V)
7  R = 1e3            # Résistance (Ω)
8  C = 5e-6           # Capacité (F)
9
10 # Méthode d'Euler ------------------------------------------------------------
11 N = 10000                          # Nombre de points à tracer
12 t_max = 8*R*C                      # Temps maximal
13 t = np.linspace(0, t_max, N)       # Array de taille N allant de 0 à t_max
14 dt = t[1] – t[0]                   # Pas de temps
15
16 uc = np.zeros(N)           # Création d'un array vide (rempli de 0) de longueur N
17 uc[0] = 0                  # Condition initiale
18
19 # On boucle entre i = 0 et i = N-1 afin de remplir l'array uc
20 for i in range(N-1):
21     uc[i+1] = uc[i] + (E-uc[i])/τ * dt
22
23 # Solution exacte ------------------------------------------------------------
24 sol = E * (1 - np.exp(-t/τ))
25
```

```
# Graphique ----------------------------------------------------------------
fig, ax = plt.subplots()

ax.plot(t, sol, c='g', ls='-', lw=5, alpha=0.5, label="Solution réelle")
ax.plot(t, uc, c='r', ls='-', lw=2, label="Méthode d'Euler")

ax.set_title("Cas où dt = 10$^{-5}$ s", fontsize=12)
ax.set_xlabel("t (s)", fontsize=12)
ax.set_ylabel("u$_c$ (V)", fontsize=12)
ax.set_xlim(0, t[-1])
ax.set_ylim(0, 5.2)
ax.legend()
ax.grid(True)
```

La méthode d'Euler est sensible au choix du pas de temps. Il faut choisir dt très petit devant le temps caractéristique de variation de la fonction (ici, $dt \ll \tau = 5 \cdot 10^{-3}$ s). Avec $dt = 10^{-5}$ s, on obtient la figure de gauche : la méthode d'Euler coïncide avec la solution réelle. Avec $dt = 10^{-3}$ s, on obtient la figure de droite : la méthode d'Euler ne coïncide pas avec la solution réelle.



II.2 - Ordre 1 avec « odeint »

En pratique, la méthode d'Euler est peu utilisée car gourmande en ressource et peu précise. On préfèrera utiliser la fonction « odeint » du module « scipy.integrate ».

Cette fonction a besoin de connaître :

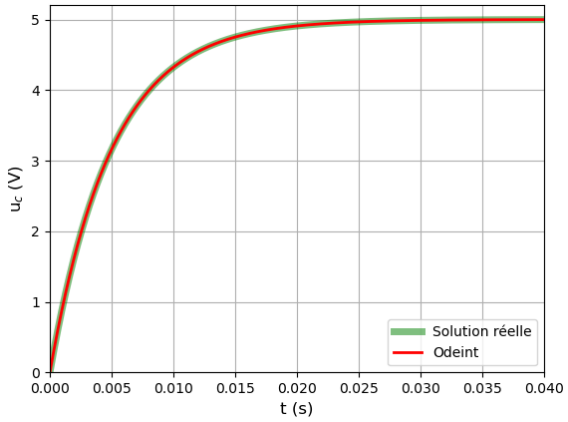
- l'expression de la dérivée première donnée par l'équation différentielle ;
- la condition initiale ;
- l'array des abscisses (en physique, c'est souvent le temps).

Reprenons l'exemple de la charge du condensateur. L'équation différentielle donne l'expression de la dérivée de la tension aux bornes du condensateur :

$$\frac{du_c}{dt} = \frac{E - u_c}{\tau}$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint # On importe uniquement la fonction odeint du module scipy.integrate
plt.close('all')

# Paramètres ----------------------------------------------------------------
E = 5                # Tension d'alimentation (V)
τ = 5e-3             # Constante de temps (s) : 5 ms

# Résolution de l'ED --------------------------------------------------------
```

```python
# Fonction qui renvoie la dérivée dy/dt donnée par l'équation différentielle
def deriv(y, t):
    dydt = (E - y)/τ
    return dydt

t = np.linspace(0, 8*τ, 1000)          # Création d'un array de 1000 points équitablement espacés entre 0 et 8τ
u0 = 0                                 # Condition initiale (V)
uc = odeint(deriv, u0, t)              # Résolution de l'équation différentielle par la fonction odeint

# Solution exacte ------------------------------------------------------------
sol = E * (1 - np.exp(-t/τ))

# Graphique ------------------------------------------------------------------
fig, ax = plt.subplots()

ax.plot(t, sol, c='g', ls='-', lw=5, alpha=0.5, label="Solution réelle")
ax.plot(t, uc, c='r', ls='-', lw=2, label="Odeint")

ax.set_xlabel("t (s)", fontsize=12)
ax.set_ylabel("u$_c$ (V)", fontsize=12)
ax.set_xlim(0, t[-1])
ax.set_ylim(0, 5.2)
ax.legend()
ax.grid(True)
```



II.3 - Ordre 2 avec « odeint »

La fonction « odeint » ne sait résoudre que des équations différentielles d'ordre 1. Heureusement, il est possible de transformer une équation différentielle d'ordre N en N équations différentielles d'ordre 1. C'est cette astuce que nous allons utiliser pour résoudre une équation différentielle d'ordre 2.

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

$$\ddot{\theta} + \omega_0^2 \sin(\theta) = 0$$

On pose le vecteur Y :

$$Y = \begin{pmatrix} \theta(t) \\ \omega(t) \end{pmatrix}$$

On détermine l'expression de dY/dt à l'aide de l'équation différentielle.

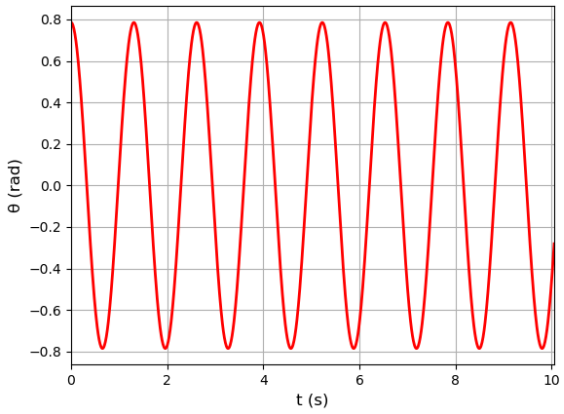
$$\frac{dY}{dt} = \begin{pmatrix} \omega(t) \\ -\omega_0^2 \sin(\theta(t)) \end{pmatrix}$$

On obtient bien au final un système de deux équations différentielles (couplées) d'ordre 1.

Dans Python, nous exprimons la dérivée dY/dt en fonction de Y :

$$\frac{dY}{dt} = \begin{pmatrix} Y[1] \\ -\omega_0^2 \sin(Y[0]) \end{pmatrix}$$

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
plt.close('all')

# Paramètres ----------------------------------------------------------
ω0 = 5              # Pulsation propre (rad/s)
T = 2*np.pi/ω0    # Période

# Résolution de l'ED --------------------------------------------------
def deriv(y, t):
    dy =[y[1], -ω0**2*np.sin(y[0])]
    return dy

t = np.linspace(0, 8*T, 1000)
CI = [np.pi/4, 0]                          # Conditions initiales : θ(t=0) = π/4 rad et ω(t=0) = 0 rad/s
θ, ω = odeint(deriv, CI, t)

# Graphique -----------------------------------------------------------
fig, ax = plt.subplots()

ax.plot(t, θ, c='r', ls='-', lw=2)

ax.set_xlabel("t (s)", fontsize=12)
ax.set_ylabel("θ (rad)", fontsize=12)
ax.set_xlim(0, t[-1])
ax.grid(True)
```



III - Analyse statistique

III.1 - Incertitude de type B

Lorsqu'une mesure ne présente pas de variabilité ou que seule une unique mesure peut être réalisée, l'expérimentateur doit estimer la plus petite plage dans laquelle il est certain de trouver la valeur recherchée. On note $\overline{x}$ la valeur centrale de cette plage et Δ sa demi-largeur. Autrement dit, l'expérimentateur est certain de trouver la valeur recherchée dans l'intervalle $\overline{x} \pm \Delta$.

Dans ce cas, on a vu (cours D2) que l'incertitude-type sur la mesure vaut :

$$u(\overline{x}) = \frac{\Delta}{\sqrt{3}}$$

On considère que l'expérimentateur aurait pu choisir **au hasard** n'importe quelle valeur dans l'intervalle $\overline{x} \pm \Delta$. Nous allons donc simuler avec python une nous une série de N titrages $\{x_i\}$, en choisissant à chaque fois x_i de manière aléatoire dans l'intervalle $\overline{x} \pm \Delta$, puis nous allons réaliser un traitement d'incertitudes de type A sur cette série de mesure.

Ce type de script, présentant un **tirage aléatoire**, s'appelle une **simulation de Monte-Carlo**.

Application :

La mesure d'une masse ne présente aucune variabilité, la balance affiche toujours :

$$m = 500{,}0 \text{ g}$$

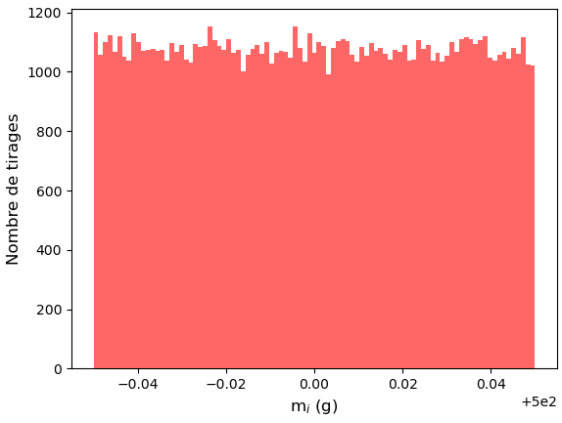
On pose $\boxed{\overline{m} = 500{,}0 \text{ g}}$, et on estime $\boxed{\Delta = 0{,}05 \text{ g}}$.

```python
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rd  # Module nécessaire pour réaliser des tirages aléatoires
plt.close('all')

# Paramètres ----------------------------------------------------------------
m = 500            # Masse en (g)
Δ = 0.05           # (g)

# Tirages aléatoires --------------------------------------------------------
N = 100000                                  # Nombre de tirages à réaliser
m_sim = m + rd.uniform(-Δ, Δ, N)            # Tirage uniforme entre -Δ et Δ

# Analyse statistique des résultats de la simulation MC ---------------------
m_moy = np.average(m_sim)       # Calcul de la valeur moyenne de m_sim
u_m = np.std(m_sim, ddof=1)     # Ecart-type (= incertitude-type) de m_sim

# Affichage des paramètres dans la console
print("Analyse statistique de la distribution :")
print('Moyenne =', m_moy)
print('Ecart-type =', u_m)

# Graphique -----------------------------------------------------------------
fig, ax = plt.subplots()

ax.hist(m_sim, color='r', alpha=0.6, bins='rice')  # Tracé de l'histogramme
# L'argument bins='rice' permet de gérer automatiquement la largeur des rectangles

ax.set_xlabel("m$_i$ (g)", fontsize=12)
ax.set_ylabel("Nombre de tirages", fontsize=12)
```

Sur le graphique, on retrouve bien une distribution qui s'apparente à une loi uniforme.

Dans la console, on obtient :

```
Analyse statistique de la distribution :
Moyenne = 499.9998777987481
Ecart-type = 0.02888238177730027
```



On retrouve donc bien les résultats attendus par la théorie : $\overline{m} = 500{,}0$ g et $u(\overline{m}) = \frac{\Delta}{\sqrt{3}} = 0{,}029$ g.

La précision est d'autant meilleure que le nombre de tirage N est grand.

III.2 - Composition des incertitudes

Dans cette partie, nous allons vérifier par une simulation de Monte-Carlo la cohérence des formules de propagation des incertitudes.

On pose :

$$m = 500{,}00 \pm 0{,}029 \text{ g} \qquad \text{et} \qquad P = 4{,}900 \pm 0{,}058 \text{ N}$$

D'après les formules de propagation :

$$\overline{g} = \frac{\overline{P}}{\overline{m}} = \frac{4{,}900}{0{,}5000} = 9{,}80\ \mathrm{m \cdot s^{-2}} \qquad \text{et} \qquad u(\overline{g}) = \overline{g}\sqrt{\left(\frac{u(\overline{P})}{\overline{P}}\right)^2 + \left(\frac{u(\overline{m})}{\overline{m}}\right)^2} = 0{,}12\ \mathrm{m \cdot s^{-2}}$$

Nous allons retrouver ces résultats à l'aide de titrages aléatoires.

Voici les grandes lignes du code.

- Convertir les incertitudes-type $u(\overline{m})$ et $u(\overline{P})$ en intervalle $\Delta(\overline{m})$ et $\Delta(\overline{P})$ à l'aide de la formule : $u = \Delta/\sqrt{3}$.
- Effectuer une série de titrage aléatoire de m_i dans l'intervalle $\overline{m} \pm \Delta(\overline{m})$ et de P_i dans l'intervalle $\overline{P} \pm \Delta(\overline{P})$.
- Calculer, pour chaque tirage, la valeur de $g_i = P_i/m_i$.
- Déterminer la valeur moyenne $\overline{g}$ et l'écart-type $u(\overline{g})$ de la distribution $\{g_i\}$.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import numpy.random as rd
4  plt.close('all')
5
6  # Paramètres ------------------------------------------------------------
7  m = 500 * 1e-3
8  u_m = 0.029 * 1e-3
9  Δ_m = u_m * np.sqrt(3)
10
11 P = 4.900
12 u_P = 0.058
13 Δ_P = u_P * np.sqrt(3)
14
15 # Tirages aléatoires ---------------------------------------------------
16 N = 100000                                  # Nombre de tirages à réaliser
17 m_sim = m + rd.uniform(-Δ_m, Δ_m, N)       # Tirage uniforme de m
18 P_sim = P + rd.uniform(-Δ_P, Δ_P, N)       # Tirage uniforme de P
19
20 g_sim = P_sim / m_sim                       # Calcul de {gi}
21
22 # Analyse statistique des résultats de la simulation MC ----------------
23 g_moy = np.average(g_sim)          # Calcul de la valeur moyenne de g_sim
24 g_m = np.std(g_sim, ddof=1)        # Ecart-type (= incertitude-type) de g_sim
25
26 # Affichage des paramètres dans la console
27 print("Analyse statistique de la distribution :")
28 print('Moyenne =', g_moy)
29 print('Ecart-type =', g_m)
```

Dans la console, on obtient :

```
Analyse statistique de la distribution :
Moyenne = 9.800439653142298
Ecart-type = 0.11623497063331419
```

Cela est parfaitement cohérent avec la formule théorique. De nouveau, la précision est d'autant meilleure que le nombre de tirage N est grand.

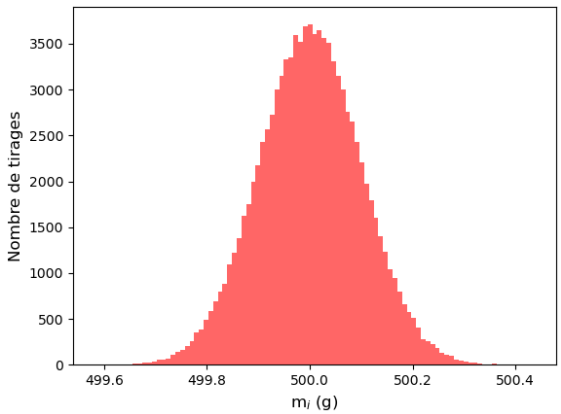
III.3 - Tirage non uniforme

Pour effectuer un tirage non uniforme, on utilise souvent en physique la **loi de probabilité normale** :

$$f(x) \propto \exp\left(\left(\frac{x - \overline{x}}{\sigma_x}\right)^2\right)$$

Il s'agit d'une courbe en cloche, de moyenne $\overline{x}$ et de d'écart-type σ_x.

```python
import matplotlib.pyplot as plt
import numpy.random as rd
plt.close('all')

# Paramètres ----------------------------------------------------------------
m = 500             # Moyenne
σ_m = 0.1           # Ecart-type

# Tirages aléatoire ---------------------------------------------------------
N = 100000                          # Nombre de tirages à réaliser
m_sim = rd.normal(m, σ_m, N)        # Tirage selon la loi normale

# Graphique -----------------------------------------------------------------
fig, ax = plt.subplots()

ax.hist(m_sim, color='r', alpha=0.6, bins='rice')

ax.set_xlabel("m$_i$ (g)", fontsize=12)
ax.set_ylabel("Nombre de tirages", fontsize=12)
```



IV - Algèbre linéaire

IV.1 - Intégration

Intégrer une fonction $f(x)$ entre a et b consiste à déterminer l'aire (algébrique) sous la courbe dans l'intervalle $[a\,;b]$.

Une méthode numérique d'intégration consiste à :

- Découper l'intervalle $[a\,;b]$ en N intervalles de largeur : $x_{i+1} - x_i$;
- Approximer l'aire sous la fonction dans chaque intervalle $[x_i\,;x_{i+1}]$ à l'aire du rectangle de largeur δx et de hauteur :

$$\frac{f(x_i) + f(x_{i+1})}{2}$$

Cette approximation est d'autant meilleure que $N \to \infty$.

Application :

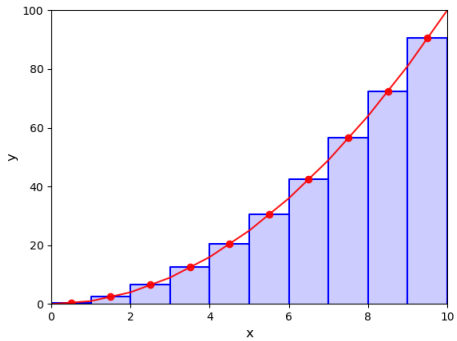
Déterminons l'aire sous la courbe $f(x) = x^2$ dans l'intervalle $[a\,;b]$. On s'attend à obtenir :

$$\mathcal{A} = \int_0^{10} x^2\,dx = \left[\frac{x^3}{3}\right]_0^{10} = \frac{1000}{3} \simeq 333{,}33$$

Résultats de la simulation :

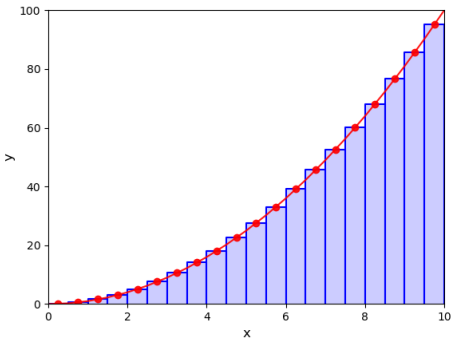
```python
import numpy as np

# Paramètres ----------------------------------------------------------------
```

```
N = 10000          # Nombre d'intervalles
N += 1             # Nombre de points

x = np.linspace(0, 10, N)
y = x**2

# Intégration ----------------------------------------------------------------
A = 0                                   # Aire des rectangles (qui va être incrémentée)
for i in range(N-1):
    A += (x[i+1]-x[i]) * (y[i]+y[i+1])/2          # Ajout de l'aire d'un rectangle

print('Aire =', A)
```



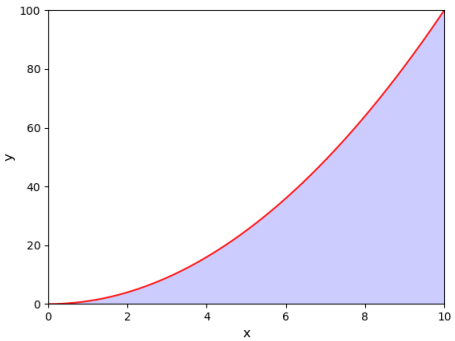
Cas où $N = 10$

$\mathcal{A} = 304{,}55$



Cas où $N = 20$

$\mathcal{A} = 317{,}86$



Cas où $N = 10^4$

$\mathcal{A} = 333{,}30$

IV.2 - Dérivation

Soit une fonction $f(x)$ échantillonnée en un ensemble de point $\{x_i\}$. La dérivée $f'(x_i)$ peut être approximée par la relation :

$$f'(x_i) \simeq \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

L'approximation est d'autant meilleure que $(x_{i+1} - x_i) \to 0$.

À l'aide de Numpy, la dérivée se calcule en une ligne grâce à la commande :

$$dfdx = (f[1:] - f[:-1])/(x[1:] - x[:-1])$$

En effet, $[1:]$ signifie « tous les indices de 1 jusqu'à (:) dernier et $[:-1]$ signifie « tous les indices du premier jusqu'à l'avant dernier (-1). $dfdx$ est donc un array de taille $N-1$ qui contient les termes :

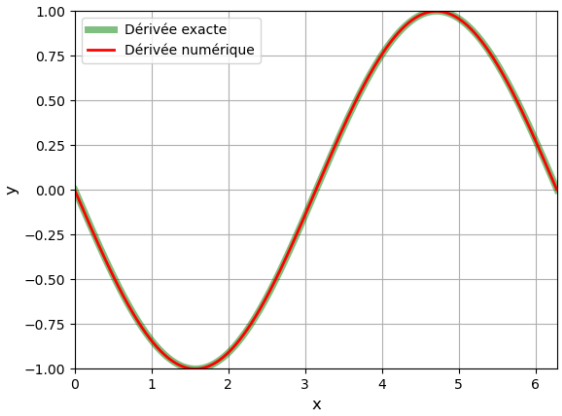
$$dfdx = \left[\frac{f[1]-f[0]}{x[1]-x[0]} \quad \frac{f[2]-f[1]}{x[2]-x[1]} \quad \dots \quad \frac{f[N-1]-f[N-2]}{x[N-1]-x[N-2]}\right]$$

<u>Remarque</u> :

Cette méthode ne permet pas d'estimer la dérivée au dernier point de la liste puisque $f[N]$ n'est, par définition, pas dans la liste. En conséquence, elle réduit de 1 la taille de la liste. C'est un détaille à bien prendre en compte dans le script, notamment lors du tracé des fonctions.

```
import matplotlib.pyplot as plt
import numpy as np
plt.close('all')

# Paramètres ----------------------------------------------------------------
N = 1000 # Nombre de points d'échantillonnage
x = np.linspace(0, 2*np.pi, N)
y = np.cos(x)

# Dérivation ----------------------------------------------------------------
dydx = (y[1:]-y[ :-1])/(x[1:]-x[:-1])
```

```python
dydx_exacte = -np.sin(x)  # Calcul de la dérivée exacte pour comparaison

# Graphique ----------------------------------------------------------------
fig, ax = plt.subplots()

ax.plot(x, dydx_exacte, c='g', ls='-', lw=5, alpha=0.5, label="Dérivée exacte")
ax.plot(x[:-1], dydx, c='r', ls='-', lw=2, label="Dérivée numérique") # x[:-1] permet d'exclure le dernier point de la liste

ax.set_xlabel("x", fontsize=12)
ax.set_ylabel("y", fontsize=12)
ax.set_xlim(x[0], x[-1])
ax.set_ylim(-1, 1)
ax.legend()
ax.grid(True)
```



V - Résolution d'une équation

Dans cette partie, nous allons apprendre à résoudre numériquement une équation. Par exemple, intéressons-nous aux points d'intersection des deux droites :

$$g_1(x) = 2x^2 - 4x - 11 \qquad \text{et} \qquad g_2(x) = x^2 - 4x + 5$$

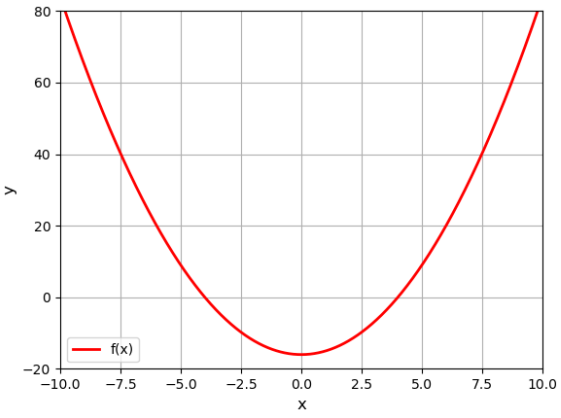
Il s'agit donc de trouver les solutions de l'équation :

$$2x^2 - 4x - 11 = x^2 - 4x + 5 \quad \Rightarrow \quad x^2 - 16 = 0$$

Posons $f(x) = x^2 - 16$, la fonction dont on cherche les racines.

La première étape consiste à tracer $f(x)$ afin de déterminer un intervalle dans lequel chercher une unique racine.

On constate graphiquement qu'il existe deux racines. Intéressons-nous à la racine positive. Grace au graphique, nous savons qu'elle est comprise dans l'intervalle $[0\,;10]$.



V.1 - Méthode dichotomique

La méthode dichotomique consiste à répéter la division d'un intervalle I en deux parties égales, puis à sélectionner le sous-intervalle dans lequel existe la racine de la fonction.

Notons $\mathrm{I} = [a\,;b]$ l'intervalle de recherche, Δ sa largeur et m son milieu.

$$\Delta = b - a \qquad \text{et} \qquad m = \frac{a+b}{2}$$

L'algorithme consiste à créer une boucle, dans laquelle on distingue 3 cas.

- Si $f(m) = 0$, alors m est la racine recherchée et il est possible de sortir de la boucle.
- Si $f(a)$ et $f(m)$ sont de même signe, alors $f(x)$ n'est pas passée par 0 dans l'intervalle $[a\,;m]$. La racine se trouve donc dans l'intervalle $[m\,;b]$.
- Si $f(a)$ et $f(m)$ sont de signe opposé, la racine se trouve dans l'intervalle $[a\,;m]$.

Ainsi, à chaque étape, on divise par 2 la longueur de l'intervalle de recherche. On sort de la boucle lorsque la largeur de l'intervalle Δ est inférieure à une valeur ε arbitraire, qui définie la précision du résultat.

```
# Définition de l'équation dont on cherche la racine : f(x) = 0
def f(x):
    return x**2 - 16

# Algorithme de dichotomie
ε = 1e-9
I = [0, 10]
while I[1] – I[0] > ε:
    m = (I[0]+ I[1])/2          # Milieu de l'intervalle
    if f(m) == 0:               # Cas où f(m) = 0
        I = [m, m]
    elif f(I[0]) * f(m) > 0:    # Cas où f(I[0]) et f(m) sont de même signe
        I = [m, I[1]]
    else:                       # Cas où f(I[0]) et f(m) sont de signe opposé
        I = [I[0], m]

print('Racine =', m)
```

Dans la console, on obtient :

```
Racine = 3.9999999999417923
```

On retrouve bien la racine $x = 4$ avec une précision $\varepsilon = 10^{-9}$.

V.2 - Fonction « bisect »

La fonction « bisect » du module « scipy.optimize » contient un algorithme de recherche de racine plus performant que l'algorithme précédent.

```
from scipy.optimize import bisect

# Définition de l'équation dont on cherche la racine : f(x) = 0
def f(x):
    return x**2 - 16

racine = bisect(f, 0, 10) # On cherche une racine de f entre 0 et 10

print('Racine =', racine)
```

Dans la console, on obtient :

```
Racine = 3.9999999999997726
```