Capacités numériques exigibles en Physique et en Chimie

En prépa, tu as des capacités numériques spécifiques à la Physique et à la Chimie à maîtriser. Elle peuvent toutes être mises en oeuvre à l'aide du langage de programmation Python. J'espère que tu me connais déjà un peu ! Malheureusement, aucune heure n'est prévue à l'emploi du temps pour faire cela sur machine et en petit groupe. Alors, les profs ils montrent parfois des trucs à toute la classe mais si tu ne bidouilles pas toi même chez toi çà ne sert à rien. Pas de panique ! Je suis là pour t'aider. Il faut d'abord que tu installes un logiciel sur ton ordi perso permettant d'écrire, lire et exécuter des scripts .py.
Moi j'aime bien Spyder ! Tu vas voir c'est bien expliqué !

Dans ce document, tu trouveras de nombreux exemples qui présentent les instructions Python de base dont tu auras besoin. Les script repérés par un 💻 *sont déjà écrits. Ils sont disponibles sur le site de la classe et dans l'espace de partage sur le réseau du Lycée. Il faut les sauvegarder, les exécuter, tenter des modifications, en observer les effets...*

Bref... il faut les manipuler pour assimiler mon langage ! Je te conseille de t'y mettre assez vite mais tu n'es pas obligé de tout maîtriser dès le début de l'année. Seuls les deux dernières parties sont un peu difficiles.

Conserve ce document à portée de main, il te servira en TP, pour des exercices ou un DM...

Sommaire

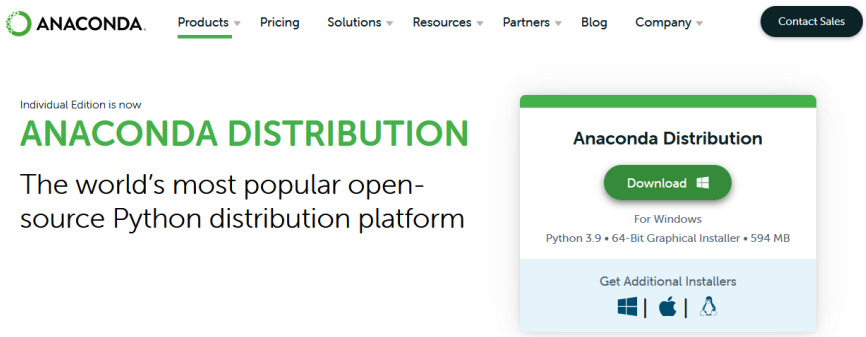
PCSI

Installer Spyder chez soi

De nombreux logiciel permettent d'exploiter le langage Python : IDLE, Pyzo, Spyder…
Nous retenons ici **Spyder** car il a le mérite de contenir toutes les bibliothèques nécessaires (et beaucoup d'autres, ce qui le rend un peu lourd sur le disque, comptez un peu moins de 6 Go). Par ailleurs il détecte de nombreuses erreurs de synthaxe avant exécution... c'est un vrai gain de temps : pas besoin d'attendre que çà plante pour corriger ☺!

Procédure à suivre :

- ✓ Se rendre sur https://www.anaconda.com/products/distribution#windows



- ✓ Double cliquer, dans le dossier « Téléchargements » sur le fichier

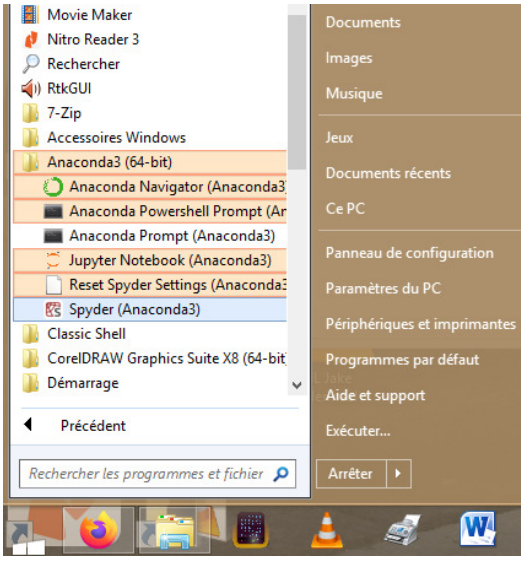
Anaconda3-2022.05-Windows-x86_64.exe

- ✓ Cliquer sur l'enchaînement habituel : « Next », « I agree », cocher « All users (admin) », « Install », « Next », « Next », « Finish ».

Après quelques minutes, Python 3 est installé avec les bibliothèques.

- ✓ Lancer Spyder en cherchant dans le « menu démarrer » le dossier « Anaconda3 » puis « Spyder ».

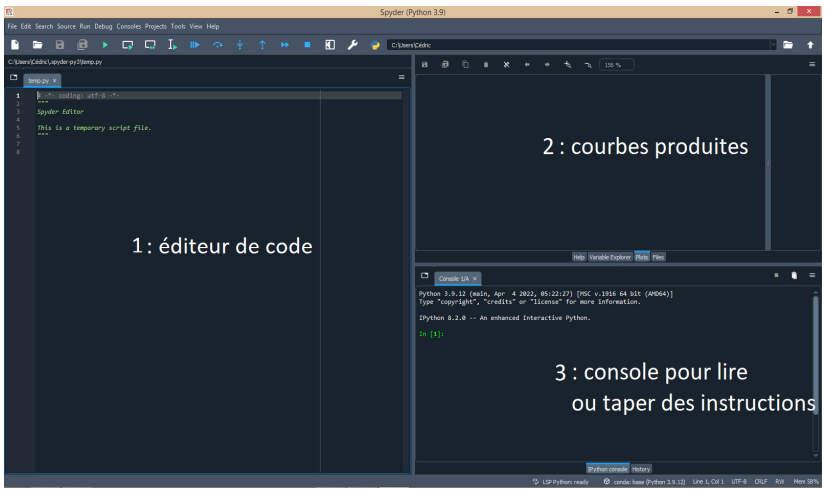
- ✓ Installer un raccourci sur le bureau pour les utilisations futures



Remarque :

Les utilisateurs de MAC peuvent suivre une procédure similaire en cliquant sur la pomme « Get additional Installers » puis en sélectionnant le premier « 64-Bit Graphical Installer (591 MB) ».

✓ Une fois Spyder lancé, voici l'allure de l'environnement de travail : nous utiliserons le même en TP.

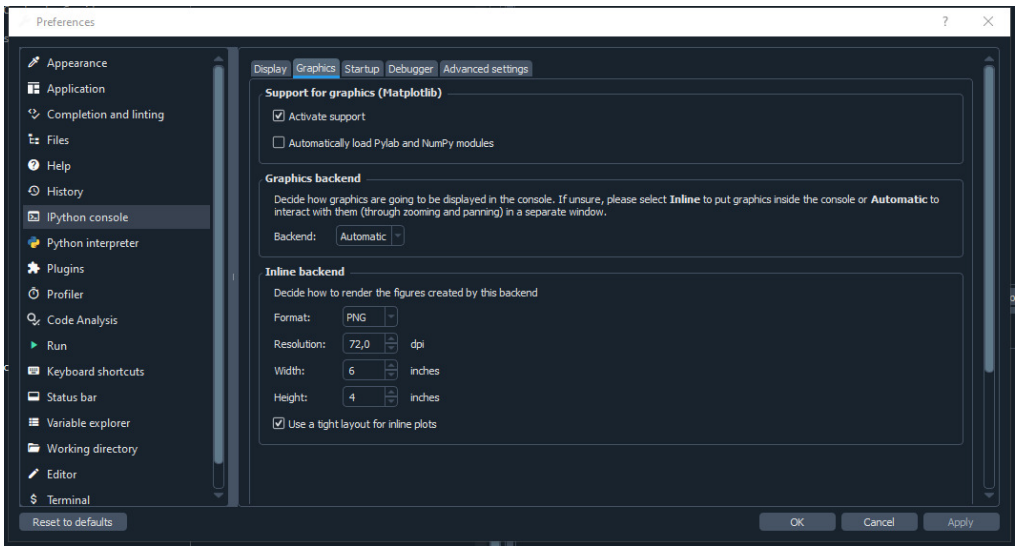


Le code sera tapé en 1, puis en appuyant sur F5 (Run en haut de la fenêtre), les effets seront visibles en 3.

Ces effets peuvent être par exemple d'afficher certaines valeurs ou phrases : instruction « print( ) » dans l'éditeur.

Si les instructions « plot » ou « hist » sont utilisées dans l'éditeur, ce qui est fréquent en physique/chimie, la fenêtre 2 présentera la ou les courbes tracées après l'appui sur F5 (courbes expérimentales, tracés de fonctions, histogrammes et représentations graphiques diverses....).

Si on veut par exemple repérer des points particuliers avec des curseurs, il ne faut pas tracer les courbes dans la console mais dans des fenêtres en suivant cette procédure :
Aller dans le menu Tools puis les onglets Preferences, IPython console, Graphics, Graphics backend et cocher Automatic au lieu de inline :



Extraits du programme officiel pour la filière PCSI (Chimie)

Annexe 3 : outils numériques

La prise en compte de capacités de codage en langage Python incluant l'utilisation de fonctions extraites de diverses bibliothèques dans la formation des étudiant·es vise à une meilleure appréhension des principes mis en œuvre par les différents logiciels de traitement des données dont l'utilisation est par ailleurs toujours recommandée et à mobiliser ces capacités dans un contexte concret, celui de la physique-chimie. Cette formation par le codage permet également de développer des capacités utiles à la physique-chimie comme le raisonnement, la logique ou la décomposition d'un problème complexe en étapes plus simples.
Le tableau ci-dessous explicite ces outils ainsi que les capacités exigibles en fin de première année. Il sera complété dans le programme de seconde année.

Outils numériques	Capacités exigibles
1. Outils graphiques	
Représentation graphique d'un nuage de points.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour représenter un nuage de points et rendre le graphe exploitable (présence d'une légende, choix des échelles…).
Représentation graphique d'une fonction.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer la courbe représentative d'une fonction et rendre le graphe exploitable (présence d'une légende, choix des échelles…).
2. Équations algébriques	
Résolution d'une équation algébrique ou d'une équation transcendante : méthode dichotomique, méthode de Newton.	Déterminer, en s'appuyant sur une représentation graphique, un intervalle adapté à la recherche numérique d'une racine par la méthode dichotomique ou par la méthode de Newton. Mettre en œuvre la méthode dichotomique ou la méthode de Newton afin de résoudre une équation avec une précision donnée. Utiliser les fonctions **bisect** ou **newton** de la bibliothèque **scipy.optimize** (leurs spécifications étant fournies).
Systèmes linéaires de n équations indépendantes à n inconnues.	Définir les matrices A et B adaptées à la représentation matricielle AX = B du système à résoudre. Utiliser la fonction **solve** de la bibliothèque **numpy.linalg** (sa spécification étant fournie).
3. Intégration – Dérivation	
Calcul approché du nombre dérivé d'une fonction en un point.	Utiliser un schéma numérique centré ou décentré pour déterminer une valeur approchée du nombre dérivé d'une fonction en un point.
4. Equations différentielles	
Équations différentielles d'ordre 1.	Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1 ou un système d'équations différentielles.
5. Statistiques	
Régression linéaire.	Utiliser la fonction **polyfit** de la bibliothèque numpy (sa spécification étant fournie) pour exploiter des données. Utiliser la fonction **random.normal** de la bibliothèque numpy (sa spécification étant fournie) pour simuler un processus aléatoire.

Extraits du programme officiel pour la filière PCSI (Physique)

Domaines numériques	Capacités exigibles
1. Outils graphiques	
Représentation graphique d'un nuage de points.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour représenter un nuage de points.
Représentation graphique d'une fonction.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer la courbe représentative d'une fonction.
Courbes planes paramétrées.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer une courbe plane paramétrée.
2. Équations algébriques	
Résolution d'une équation algébrique ou d'une équation transcendante : méthode dichotomique.	Déterminer, en s'appuyant sur une représentation graphique, un intervalle adapté à la recherche numérique d'une racine par une méthode dichotomique. Mettre en œuvre une méthode dichotomique afin de résoudre une équation avec une précision donnée. Utiliser la fonction **bisect** de la bibliothèque **scipy.optimize** (sa spécification étant fournie).
3. Intégration – Dérivation	
Calcul approché d'une intégrale sur un segment par la méthode des rectangles.	Mettre en œuvre la méthode des rectangles pour calculer une valeur approchée d'une intégrale sur un segment.
Calcul approché du nombre dérivé d'une fonction en un point.	Utiliser un schéma numérique pour déterminer une valeur approchée du nombre dérivé d'une fonction en un point.
4. Equations différentielles	
Equations différentielles d'ordre 1.	Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1.
Equations différentielles d'ordre supérieur ou égal à 2	Transformer une équation différentielle d'ordre n en un système différentiel de n équations d'ordre 1. Utiliser la fonction **odeint** de la bibliothèque **scipy.integrate** (sa spécification étant fournie).
5. Probabilité - statistiques	
Variable aléatoire.	Utiliser les fonctions de base des bibliothèques **random** et/ou **numpy** (leurs spécifications étant fournies) pour réaliser des tirages d'une variable aléatoire. Utiliser la fonction **hist** de la bibliothèque **matplotlib.pyplot** (sa spécification étant fournie)
	pour représenter les résultats d'un ensemble de tirages d'une variable aléatoire. Déterminer la moyenne et l'écart-type d'un ensemble de tirages d'une variable aléatoire.
Régression linéaire.	Utiliser la fonction **polyfit** de la bibliothèque **numpy** (sa spécification étant fournie) pour exploiter des données. Utiliser la fonction **random.normal** de la bibliothèque **numpy** (sa spécification étant fournie) pour simuler un processus aléatoire.

1) Représenter graphiquement un nuage de points, une fonction ou une courbe paramétrée

Les bibliothèques et modules nécessaires sont `numpy` pour la gestion des tableaux et `matplotlib.pyplot` pour les représentations graphiques. On les importe en créant des alias :

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

1.1) Nuage de points

Un expérimentateur mesure l'indice de réfraction n d'un prisme pour différentes longueurs d'onde λ de la lumière.

λ (nm)	404,7	435,8	480,0	546,1	578,1	615,0
n	1,7761	1,7652	1,7475	1,7358	1,7294	1,7229

Chaque paire (λ_i, n_i) peut être représentée graphiquement par un point dans le plan muni de deux axes orthogonaux. On obtient ainsi un "nuage de points". La première étape consiste à placer les valeurs dans des "tableaux numpy". Cela peut se faire à la main... attention : les virgules deviennent les séparateurs entre les différentes valeurs.

```
Lambda=np.array([404.7,435.8,480.0,546.1,578.1,615])
n=np.array([1.7761,1.7652,1.7475,1.7358,1.7294,1.7229])
```

Ou bien, si les données sont issues d'un tableur généré par le logiciel dédié à une carte d'acquisition (Latis-Pro) ou une application sur Smartphone (Phyphox), il est possible de demander à Python de lire les valeurs dans un fichier .csv Ce fichier doit être placé dans le même répertoire que le script .py.

```
import csv
with open("prisme.csv") as fichier :
        reader = csv.reader(fichier, delimiter = ",")
        entete = next(reader)
        entete = next(reader)
        données = np.array(list(reader)).astype("float")
Lambda = données [:,0]
n = données [:,1]
```

La commande pour représenter les valeurs de λ en abscisse et celles de n en ordonnée est :

```
plt.plot(Lambda,n,'+')
```

Il faut absolument préciser que l'on veut représenter chaque point par un +... sans cela on obtient une ligne brisée !

Si la couleur ou le style déplaisent, on peut en changer :

```
plt.plot(Lambda,n,'o',color='red')
```

Enfin il faut penser à la communication : nommer les axes et proposer un titre pertinent est le minimum requis !

```
plt.xlabel(r"$\lambda(nm)$")
plt.ylabel("n")
plt.title("Dispersion de la lumière par un prisme")
```

💻 `prisme.py`

Note importante :
Pour le traitement de données, l'utilisation des objets de type `numpy.ndarray` doit être privilégiée : il est alors très commode d'effectuer des opérations "élément par élément". Ceci n'est pas le cas avec des objets de type `list`.

💻 `liste VS tableau numpy.py`

1.2) Fonction d'une variable

On souhaite tracer la courbe représentative de la fonction : $f : x \to \ln(\sqrt{x^2-1}) + \sin(x)$.
Notons qu'elle n'est pas définie entre -1 et +1.

Il faut définir cette fonction (noter l'indentation et l'utilisation des fonctions prédéfinies de la bibliothèque `numpy`) :

```
def mafonction(x):
        return np.log(np.sqrt(x**2-1))+np.sin(x)
```

Il faut ensuite échantillonner cette fonction assez finement pour avoir une sensation de continu (N assez grand) :

```
xmin = -12
xmax = 12
N = 1000
X = np.linspace(xmin,xmax,N)
Y = mafonction(X)
```

Il suffit alors de la représenter comme un nuage de N points que l'on ne souhaite pas matérialiser individuellement :

```
plt.plot(X,Y)
```

tracé de fonction.py

1.3) Courbe plane paramétrée

Soit un point matériel se déplaçant au cours du temps t (le paramètre !) dans le plan cartésien selon :

$$x(t) = 2\cos(t)$$
$$y(t) = \sin(3t)$$

On souhaite tracer l'allure de sa trajectoire.

Pour cela, on utilise le fait qu'une fonction Python peut renvoyer plusieurs éléments :

```
def mafonction(t):
    return 2*np.cos(t),np.sin(3*t)
```

Il faut veiller à les mettre dans le bon ordre lors de l'appel de la fonction :

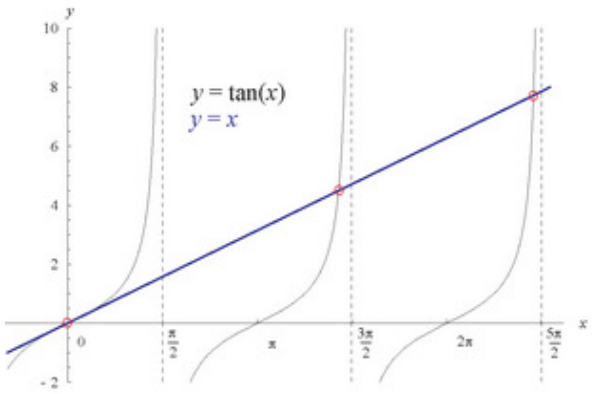
```
N = 1000
t = np.linspace(0,6,N)
X,Y = mafonction(t)
plt.plot(X,Y)
```

La trajectoire qui s'affiche est une courbe de Lissajou.

Lissajou.py

2) Résoudre une équation algébrique ou transcendante

En physique-chimie, on peut être amené à résoudre une équation à une inconnue x qui n'a pas de solution analytique simple.



Prenons l'exemple de l'équation $\tan x = x$.
$x = 0$ est une solution évidente mais une analyse graphique montre qu'il en existe beaucoup d'autres.

Cherchons à déterminer avec une précision fixée disons 10^{-4}, la solution légèrement inférieure à $\frac{3\pi}{2} \approx 4{,}7$

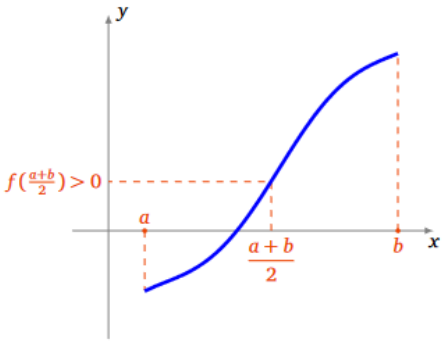
Soit la fonction $f : x \to x - \tan x$

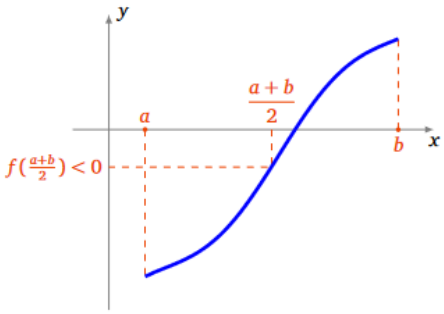
Résoudre le problème revient à chercher "le zéro" de f (c'est à dire la valeur de x qui annule f) voisin de 4,7...

2.1) Méthode dichotomique

On se donne un intervalle $[a,b]$ sur lequel la fonction f est monotone et contenant un zéro de f. On évalue ensuite le signe de f au milieu de cet intervalle. **Si** il est différent de celui de $f(a)$, c'est que le zéro est à gauche du milieu. **Sinon** c'est qu'il est à droite. On a divisé ainsi par 2 la largeur de l'intervalle de recherche et on réitère le procédé **tant que** cette largeur est supérieure à la précision souhaitée. **Si , Sinon** et **tant que** : dans la plupart des langages de programmation, cela se traduit respectivement par : `if` , `else` and `while` !

💻 `dichotomie.py`



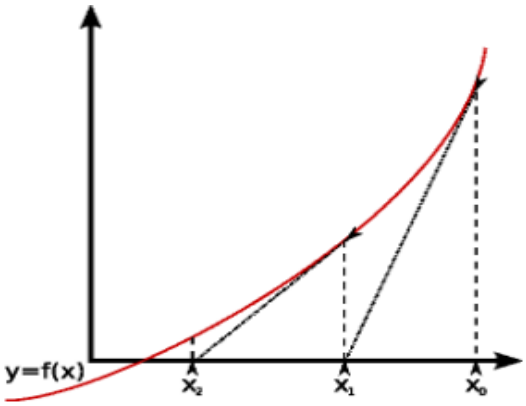


2.2) Méthode de Newton

Soit x_0 voisin d'un zéro de f. La tangente en x_0 à la courbe représentative de f est d'équation : $y = f(x_0) + f'(x_0)(x - x_0)$.

Elle coupe l'axe des abscisses en : $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

De même, la tangente à la courbe en x_1 coupe l'axe des abscisses en : $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$, etc. On construit ainsi, par récurrence, une suite $\{x_n\}$ qui , pour des fonctions f "sympathiques", converge vers le zéro que l'on cherche. On effectue cette construction **tant que** l'écart entre deux termes successifs est supérieur à la précision souhaitée... (d'où l'utilisation d'une boucle `while`).



💻 `Newton.py`

3) Résoudre un système linéaire de n équations à n inconnues

Soit le système linéaire suivant à résoudre (il s'agit de déterminer les valeurs des 3 inconnues) : $\begin{cases} 2x+3y-2z=8 \\ x-4z=1 \\ 2x-y-6z=4 \end{cases}$

Ceci peut être fait à la main (méthode du pivot de Gauss par exemple) ou à l'aide de la fonction `solve` de la bibliothèque `numpy.linalg`. Il faut passer à une écriture matricielle du système :

$$\begin{bmatrix} 2 & 3 & -2 \\ 1 & 0 & -4 \\ 2 & -1 & -6 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \\ 4 \end{bmatrix}$$

coefficient matrix A — variable matrix X — constant matrix B

```
import numpy as np
from numpy.linalg import solve
A = np.array([[2,3,-2],[1,0,-4],[2,-1,-6]])
B = np.array([8,1,4])
X = solve(A,B)
print(X)
```

Que dire de plus ?

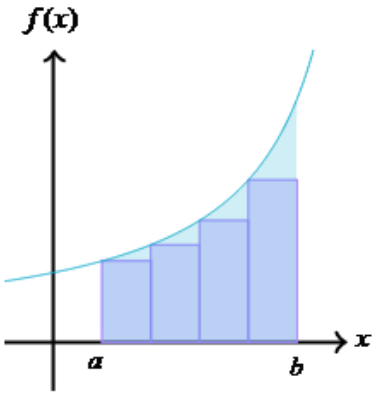
💻 système linéaire.py

4) Estimer la valeur numérique d'une intégrale

On souhaite déterminer la valeur numérique d'une intégrale : $I=\int_a^b f(x)dx$. Ceci n'est pas toujours commode, en particulier lorsque l'on ne connaît pas de primitive de f. On peut néanmoins en déterminer une valeur approchée par la méthode des rectangles qui consiste à prendre N valeurs régulièrement espacées de x dans $[a,b]$.

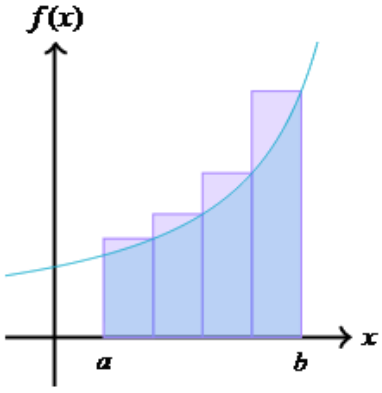
On les note ainsi : $x_n = a + n\dfrac{b-a}{N-1}$. La première valeur est $x_0 = a$ et la dernière valeur est $x_{N-1} = b$.

I est représentée graphiquement par l'aire sous la courbe représentative de f. On peut approcher I par une somme de $N-1$ rectangles chacun de largeur : $h = \dfrac{b-a}{N-1}$ appelée le pas. Deux approches sont possibles : ci dessous pour N = 5.



Rectangles à droites : $I > \sum_{n=0}^{N-2} f(x_n)\dfrac{b-a}{N-1}$

On ignore la dernière valeur : $f(x_{N-1}) = f(b)$



Rectangles à gauches : $I < \sum_{n=1}^{N-1} f(x_n)\dfrac{b-a}{N-1}$

On ignore la première valeur : $f(x_0) = f(a)$

On constate qu'en mettant en oeuvre successivement les deux approches, on obtient un encadrement de I qui sera d'autant plus resserré que N est grand (ou h petit). Ceci reste vrai pour une fonction f non monotone et/ou qui change de signe dans l'intervalle $[a,b]$. Le sens des inégalités peut néanmoins changer.

💻 integrale.py

5) Procéder à des tirages d'une variable aléatoire

Le langage Python permet de mettre en oeuvre des générateurs de nombres aléatoires.
Il est nécessaire pour cela d'importer le module `random` de la bibliothèque `numpy` :

```
from numpy import random
```

Les instructions suivantes permettent par exemple de simuler 25 tirages d'un dé à 6 faces (à priori non pipé !) :

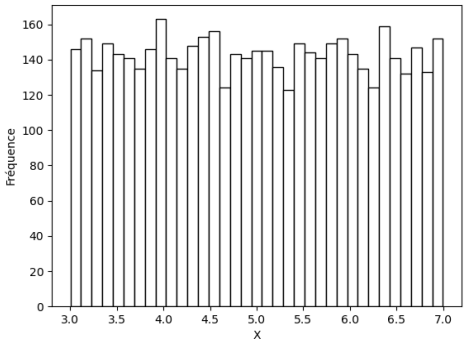
```
N = 25
tirage = random.randint(1,7,N)
print(tirage)
```

Cela génère un objet de type `numpy.ndarray` : `[4 6 3 5 6 6 6 4 4 2 6 5 5 3 2 4 5 2 1 5 1 2 6 4 3]`

Cet exemple correspond à N tirages d'une **variable aléatoire discrète**. En TP, pour simuler la variabilité du résultat d'un mesurage, la grandeur physique mesurée est modélisée par une **variable aléatoire X continue**. Deux lois de distribution sont couramment utilisées : la **loi uniforme** et la **loi normale.** On peut afficher l'histogramme du tirage réalisé : `plt.hist()`, les valeurs "expérimentales" de la moyenne : `np.mean()` et de l'écart-type : `np.std()`.

- distribution uniforme sur l'intervalle $[a\ ;b]$:

```
a,b = 3,7
N = 5000
X = random.uniform(a,b,N)
plt.xlabel("X")
plt.ylabel("Fréquence")
plt.hist(X,bins = 'rice')
print("moyenne : {:.3f}  écart-type :
{:.3f}".format(np.mean(X),np.std(X,ddof = 1)))
```

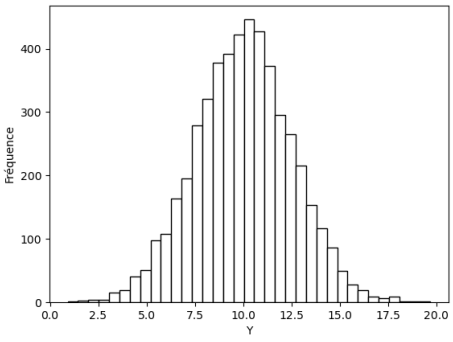


Ces lignes produisent l'histogramme ci-contre et affichent :

```
moyenne : 4.993  écart-type : 1.164
```

- distribution normale (ou gaussienne) d'espérance mathématique μ et d'écart-type σ

```
mu,sigma = 10,2.5
N = 5000
Y = random.normal(mu,sigma,N)
plt.xlabel("Y") ; plt.ylabel("Fréquence")
plt.hist(Y,bins = 'rice')
print("moyenne : {:.3f}  écart-type :
{:.3f}".format(np.mean(Y),np.std(Y,ddof = 1)))
```



Ces lignes produisent l'histogramme ci-contre et affichent :

```
moyenne : 10.017  écart-type : 2.512
```

Il s'agit de la distribution typique des notes obtenues par 5000 candidats à une épreuve de concours bien calibrée.
"bins = rice" permet d'optimiser le nombre de classes.
On peut choisir une autre valeur : "bins =15" par exemple.

💻 `variable aléatoire.py`

Application : si l'on tire au hasard deux nombres x et y dans l'intervalle [0;1] , il est géométriquement clair que la probabilité pour que le complexe z = x + jy ait un module inférieur à 1 est de π / 4. Nous pouvons ainsi estimer la valeur de π par simulation de Monte-Carlo. Le lecteur est invité à modifier le nombre N de tirages et d'observer…

💻 `pi.py`

6) Effectuer une régression linéaire

Augustin Louis Cauchy (1789-1847)

Reprenons les données expérimentales du paragraphe **1)**.

On souhaite voir si la loi empirique proposée par Cauchy au XIX^{e} siècle permet de les décrire convenablement.

Elle s'exprime ainsi : $n = \frac{A}{\lambda^2} + B$ avec A et B des constantes positives.

On reconnaît là l'équation d'une droite à condition de porter n en ordonnées et la variable adéquate $\frac{1}{\lambda^2}$ en abscisses.

Effectuer une régression linéaire consiste à rechercher l'équation de la droite (= courbe représentative d'un **polynôme de degré 1**) qui s'accorde le mieux avec les données expérimentales. On utilise pour cela la fonction `polyfit` de la bibliothèque `numpy` (**to fit** = s'adapter). Cette fonction utilise la méthode dite des moindres carrés (hors programme) pour effectuer cet ajustement. Elle renvoie un tableau numpy contenant deux nombres : le coefficient directeur A (= la pente) et l'ordonnée à l'origine B de la droite. Il suffit alors de superposer cette droite aux points expérimentaux afin de **juger visuellement de pertinence du modèle** affine.

Reprenons les données et créons la variable à reporter en abscisses. Avec les tableaux numpy, c'est immédiat :

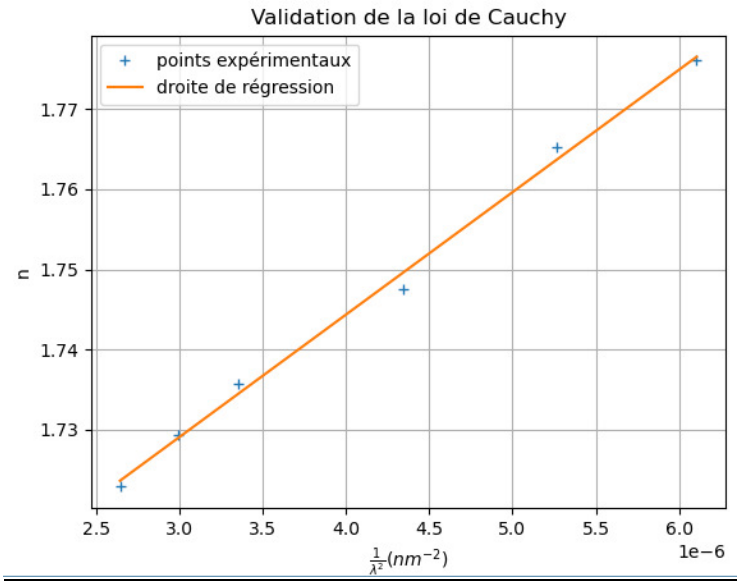
```
Lambda = np.array([404.7,435.8,480.0,546.1,578.1,615])
n = np.array([1.7761,1.7652,1.7475,1.7358,1.7294,1.7229])
X = 1/Lambda**2
```

Stockons dans A et B, les résultats de la régression linéaire :

```
A,B = np.polyfit(X,n,1)
```

Superposons sur un même graphe, les points expérimentaux et la droite de régression :

```
plt.plot(X,n,'+')
plt.plot(X,A*X+B)
```



Les points expérimentaux sont grossièrement alignés et se répartissent aléatoirement de part et d'autre de la droite de régression. Ceci est un premier argument en faveur de la loi de Cauchy. Elle semble être apte à décrire les mesures.

`régression linéaire.py`

Cependant, il faut noter que ce traitement des données ne tient absolument pas compte des incertitudes de mesures, en particulier sur l'indice de réfraction. Ces incertitudes impliquent que A et B ne sont eux mêmes déterminés que de façon incertaine. Pour évaluer les incertitudes-type sur ces paramètres du modèle, il faut effectuer un grand nombre de régressions sur des données simulées et générées aléatoirement (méthode de Monte-Carlo). Cette méthode sera présentée en cours d'année.

`régression linéaire + Monte Carlo.py`

Facultatif : la méthode des moindres carrés

Comment la fonction `polyfit` préprogrammée dans `numpy` détermine-t-elle au juste les coefficients de la régression ?

Soient N points expérimentaux $\{x_i ; y_i\}$ grossièrement alignés et une fonction affine de x de coefficients (a,b).

La quantité : $E(a,b)=\sum_{i=1}^{N}(y_i-(ax_i+b))^2$ est une bonne mesure de « l'écart » entre la droite représentative de cette fonction affine et les points expérimentaux.

Ajuster cette droite au sens des moindres carrés consiste à rechercher les valeurs des coefficients (a,b) qui minimisent cet écart. Ce sont donc celles qui satisfont à la double condition : $\frac{\partial E}{\partial b}=0$ (1) ; $\frac{\partial E}{\partial a}=0$ (2)

La condition (1) s'écrit : $\frac{\partial E}{\partial b}=-2\sum_{i=1}^{N}(y_i-(ax_i+b))=0$

$$\sum_{i=1}^{N}y_i=a\sum_{i=1}^{N}x_i+Nb$$

$$\frac{1}{N}\sum_{i=1}^{N}y_i=a\frac{1}{N}\sum_{i=1}^{N}x_i+b$$

En introduisant la notation $\langle\ \rangle$ signifiant « moyenne arithmétique de », on a ainsi : $\langle y_i\rangle=a\langle x_i\rangle+b$
Ce résultat indique géométriquement que la droite de régression passe par le centre de gravité G du nuage de points !

La condition (2) s'écrit : $\frac{\partial E}{\partial a}=-2\sum_{i=1}^{N}x_i(y_i-(ax_i+b))=0$

$$\sum_{i=1}^{N}x_iy_i=a\sum_{i=1}^{N}x_i^2+b\sum_{i=1}^{N}x_i$$

$\langle x_iy_i\rangle=a\langle x_i^2\rangle+(\langle y_i\rangle-a\langle x_i\rangle)\langle x_i\rangle$ après division par N et réemploi de (1)

$$\langle x_iy_i\rangle-\langle x_i\rangle\langle y_i\rangle=a\langle x_i^2\rangle-a\langle x_i\rangle\langle x_i\rangle$$

On retient donc : $\boxed{a=\frac{\langle x_iy_i\rangle-\langle x_i\rangle\langle y_i\rangle}{\langle x_i^2\rangle-\langle x_i\rangle^2}}$ et $\boxed{b=\langle y_i\rangle-a\langle x_i\rangle}$

Le lecteur curieux se convaincra en comparant le résultat du code précédent à celui fourni par celui-ci :

`régression linéaire sans polyfit.py`

Remarque : le calcule explicite des « meilleurs » coefficients d'un modèle (meilleurs au sens des moindres carrés) que nous venons de présenter n'est possible que dans le cas affine. On ne peut pas établir de telles expressions analytiques pour une modélisation polynômiale de degré supérieur à 1 et à fortiori pour tout autre type de modélisation. La recherche des « meilleurs » coefficients est alors effectuée par tâtonnements comme c'est le cas dans l'exemple qui suit.

Facultatif : ajustement de points expérimentaux par autre chose qu'un polynôme

Ce qui suit ne relève pas des capacités numériques exigibles mais peut être utile pour exploiter des données en TIPE.

Si l'on souhaite ajuster les paramètres d'un modèle non polynomial afin qu'il décrive au mieux des données expérimentales, on peut utiliser la fonction `curve_fit` de la bibliothèque `numpy`.

La fonction modèle choisie doit être définie par l'utilisateur ; par exemple, si on étudie la charge d'un condensateur :

```
def fonction(t,E,tau):
    u=E*(1-np.exp(-t/tau))
    return u
```

fonction de t dans laquelle, il y a deux paramètres à ajuster : E (valeur asymptotique) et tau (durée caractéristique).

Cette fonction doit être passée en premier argument lors de l'appel de `curve_fit`

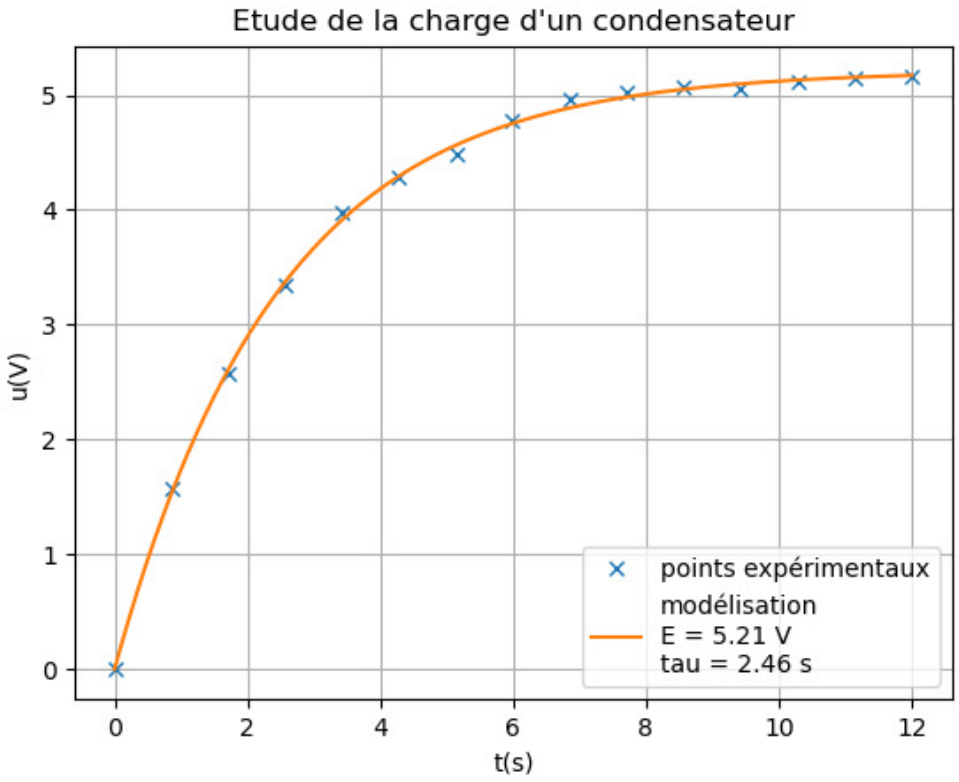
```
best_values,covar = curve_fit(fonction,texp,uexp,p0=initial_values)
```

`initial_values` est un tableau numpy dans lequel on a rentré des valeurs initiales (point trop fantaisistes !) pour les paramètres à ajuster. On récupère les valeurs ajustées « au sens des moindres carrés » dans le tableau `best_values`

On peut aussi récupérer un tableau de covariances (`covar`) mais ceci n'intéressera pas le taupin moyen.

Il suffit alors de tracer la fonction avec ces paramètres par-dessus les points expérimentaux pour juger (au moins qualitativement) de la pertinence du modèle choisi.

`Charge d'un condensateur.py`



7) Mettre en oeuvre la méthode d'Euler explicite

7.1) Exemple choisi

Considérons un système dont l'évolution temporelle est commandée par une **équation différentielle d'ordre 1** : un échantillon radioactif par exemple. Le nombre $R(t)$ de noyaux radioactifs présents à l'instant t évolue en effet selon :

$$\boxed{\frac{dR}{dt} = -\lambda R(t)} \qquad (1)$$

où λ désigne la constante radioactive indépendante du temps et caractéristique du radionucléide étudié.

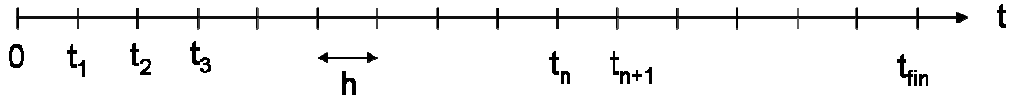
Résoudre numériquement (1) ne se justifie guère car, comme cette équation est linéaire, on en connaît très bien la

solution analytique (en notant $R_0 = R(0)$) : $\boxed{R(t) = R_0 e^{-\lambda t}}$ (2)

Appuyons nous malgré tout sur (1) pour illustrer la méthode d'Euler explicite.
Une comparaison avec (2) permettra de juger de la qualité du résultat obtenu.

7.2) Principe de la méthode

La méthode d'Euler explicite est la plus simple des méthodes "aux différences finies" pour résoudre numériquement une équation différentielle. Ces méthodes procèdent à un découpage de la variable temporelle comme ceci :



On remplace la variable t continue par une variable discrète t_n. En notant h le pas de temps : $t_n = nh$. (3)
Connaissant la valeur initiale $R(0)$, il s'agit de **calculer de proche en proche des approximations** R_n des valeurs prises par la fonction $R(t)$ inconnue aux dates t_n ultérieures : $R_n \approx R(t_n)$ (4)
La qualité de ces approximations dépend fortement de la méthode employée et du pas de temps h choisi.

7.3) Schéma numérique

Après multiplication par dt, (1) s'écrit : $dR = -\lambda R(t)dt$

Intégrons chaque membre entre t_n et t_{n+1} : $\int_{t_n}^{t_{n+1}} dR = \int_{t_n}^{t_{n+1}} -\lambda R(t)dt$

Ce qui donne : $R(t_{n+1}) = R(t_n) + \int_{t_n}^{t_{n+1}} -\lambda R(t)dt$ (5)

Ainsi, si l'on dispose d'une approximation de $R(t)$ à la date t_n, (5) permet d'en déduire une autre à la date t_{n+1}.
Dans la méthode d'Euler explicite, la valeur de l'intégrale est estimée avec la méthode du "rectangle à droite" (voir **4)**).

On remplace ainsi (5) par la relation approchée : $\boxed{R_{n+1} \approx R_n - \lambda R_n h}$ (6)

Cette dernière relation porte le nom de **schéma numérique d'Euler** associé à l'équation différentielle (1).
C'est ce schéma que l'on peut programmer afin de déterminer chaque valeur approchée R_n par récurrence.

Remarque : le schéma numérique (6) peut aussi être obtenu en se souvenant que la dérivée de $R(t)$ à l'instant t est par définition la limite quand $h \to 0$ du taux d'accroissement.

Ainsi, si h est assez petit, on a l'approximation : $$\frac{dR}{dt}(t) \approx \frac{R(t+h)-R(t)}{h} \qquad (7)$$

Ce qui peut aussi s'écrire : $$R(t+h) \approx R(t) + h\frac{dR}{dt}(t) \qquad (8)$$

La relation (8) porte le nom de développement limité à l'ordre 1 de la fonction $R(t)$ à la date t.

En reprenant (1) (3) et (4), ce développement devient bien : $\boxed{R_{n+1} \approx R_n - \lambda R_n h}$

7.4) Mise en oeuvre

Carbone14 Euler.py

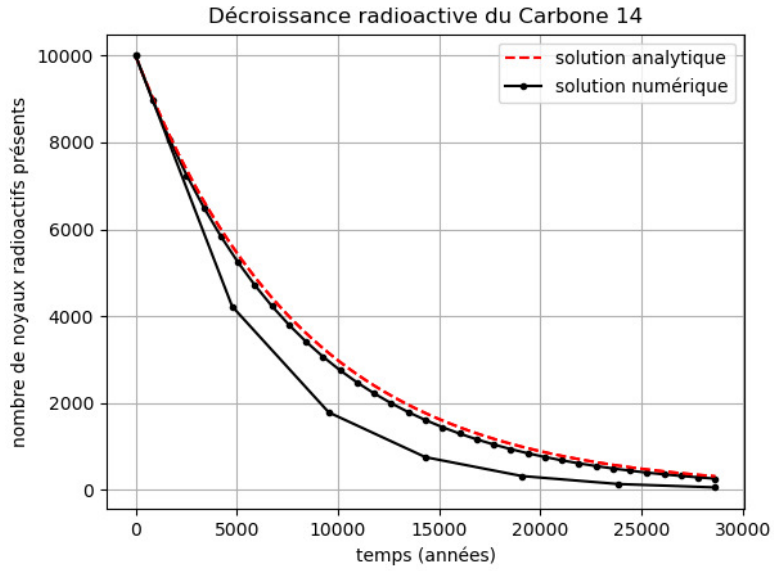
Lire attentivement les commentaires associés au code proposé ici. La nouveauté réside dans l'emploi d'une boucle `for` plus adaptée qu'une boucle `while` car on définit à l'avance le nombre d'itérations. Attention : pour N points souhaités, R_0 inclus, le schéma d'Euler n'est itéré que N-1 fois : `for n in range(N-1):`

```
R[n+1] = R[n]-Lambda*R[n]*h
```

7.5) Discussion

On a représenté ci-dessous les résultats de deux simulations numériques différentes.
La première a été réalisée avec N = 7 points seulement : elle apparaît sous la forme d'une ligne brisée peu conforme à la réalité physique et assez éloignée de la solution analytique.

Leonhard Euler (1707-1783)



Pour la seconde, on a choisi N = 35 points et la durée de la résolution est la même (à ne pas confondre avec le temps de calcul nécessaire !). Le pas de temps h a ainsi été divisé par 5 et la solution numérique proposée apparaît beaucoup plus lisse et proche de la solution analytique.

Retenons que la qualité d'une résolution numérique augmente lorsque le pas de temps h diminue.
Ceci se fait au détriment du temps de calcul qui augmente également.

Ce comportement est assez intuitif et, en pratique, un optimum doit être trouvé.
A cette échelle, on constate qu'à partir de N = 100 points, il est difficile de distinguer la solution numérique (approchée) de la solution analytique (exacte) : il n'est donc pas nécessaire de diminuer davantage le pas de temps.

7.6) Pertinence de la méthode

La méthode d'Euler explicite ne s'applique qu'aux équations différentielles d'ordre 1 et son emploi ne se justifie que si l'on n'en connaît pas de solution analytique (ou que l'on ne sait pas la trouver ☺) : présence d'un terme non linéaire ou d'un second membre non usuel typiquement. Si cette méthode est la plus simple, c'est aussi la moins performante des méthodes aux différences finies. Il en existe bien d'autres : Euler implicite, Heun, Crank-Nicolson, Runge-Kutta, etc... Ces méthodes hors programme se distinguent par leur façon plus ou moins judicieuse selon le contexte d'estimer à chaque pas la valeur de l'intégrale présente dans la relation (5). On dit aussi : "d'en réaliser la quadrature".

7.7) Comparaison avec la solution numérique obtenue à l'aide d'odeint

Une méthode plus performante est mise en oeuvre par la fonction `odeint` à importer du module `scipy.integrate`.

Pour l'utiliser, il faut définir l'équation différentielle à résoudre à l'aide d'une fonction Python :

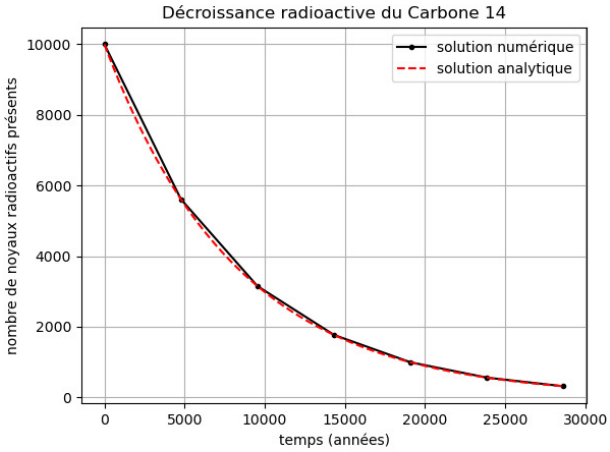
```
def C14 (R,t):
    Rp = -Lambda*R
    return Rp
```

Cette fonction retourne la dérivée de $R(t)$ par rapport à t... soit $\dot{R}(t)$ qui se lit "Rpoint" et s'abrège en "Rp".
L'appel de la fonction `odeint` requiert trois arguments à donner dans cet ordre :

- la fonction C14 précédente
- la condition initiale notée R0
- un tableau numpy noté t contenant les N dates pour lesquelles on souhaite obtenir une approximation de $R(t)$.

Elle renvoie un tableau à N éléments noté Rsol. Il contient la solution numérique souhaitée.

```
Rsol = odeint(C14,R0,t)
```



On constate que même avec N = 7 (un pas de temps assez élevé donc) la solution numérique proposée est très satisfaisante : il faut "zoomer" très fortement sur ce graphe pour y déceler un écart avec la solution analytique !

💻 `Carbone14 odeint.py`

8) Utiliser la fonction odeint pour résoudre numériquement un système différentiel

8.1) Exemple choisi

Considérons l'équation différentielle d'ordre 2 chère aux physiciens : $\boxed{\ddot{x} + \omega_0^2 x(t) = 0}$ (1)

Il s'agit de l'équation de **l'oscillateur harmonique (OH) non amorti** : $x(t)$ peut désigner par exemple la position d'une masse accrochée à l'extrémité d'un ressort et oscillant sans frottements.
$\ddot{x}$ désigne la dérivée seconde par rapport au temps de $x(t)$ et ω_0 est la pulsation propre de l'oscillateur.
Résoudre numériquement (1) ne se justifie guère car, comme cette équation est linéaire, on en connaît très bien la

solution analytique (en notant $x_0 = x(0)$ et en lâchant sans vitesse initiale) : $x(t) = x_0 \cos(\omega_0 t)$ (2)
On va le faire néanmoins car (2) fournit là encore d'un élément de comparaison.

8.2) Transformation en un système différentiel d'ordre 1

Remplaçons (1) par un système équivalent de deux équations différentielles d'ordre 1 : $\dfrac{dx}{dt} = \dot{x}(t)$ (3)

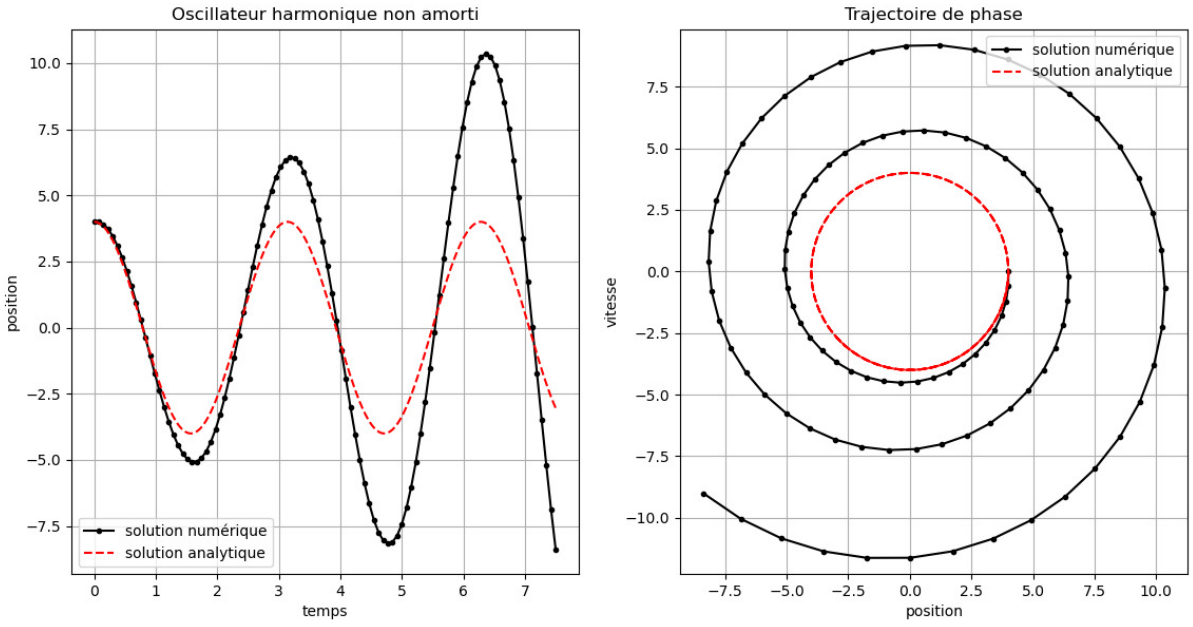
$$\frac{d\dot{x}}{dt} = -\omega_0^2 x(t) \quad (4)$$

Résoudre (1) numériquement peut se faire en résolvant (3) et (4) en parallèle : comme c'est d'ordre 1, c'est à priori possible en adoptant la méthode d'Euler explicite mais cela conduit à une solution peu satisfaisante.

8.3) "Echec" de la méthode d'Euler explicite

En reprenant la démarche vue en **7.3)**, on construit le schéma numérique : $x_{n+1} = x_n + \dot{x}_n h$ (5)

💻 `OH Euler.py` $\dot{x}_{n+1} = \dot{x}_n - \omega_0^2 x_n h$ (6)



Malgré un nombre de points assez conséquent : N = 100, on constate que la solution numérique n'est pas conforme à la réalité physique : l'amplitude des oscillations augmente. La trajectoire de phase obtenue illustre le défaut majeur de la méthode d'Euler explicite : elle ne conserve pas l'énergie mécanique.

Remarque pour les curieux : On améliore sensiblement le résultat en remplaçant x_n par x_{n+1} dans (6)... c'est de l'Euler implicite pour résoudre (4) qui remplace la méthode du rectangle à droite par du rectangle à gauche...

8.4) Utilisation d'odeint

Les équations (3) et (4) peuvent être résolues par une méthode aux différences finies plus performante dont le schéma numérique est déjà programmé dans la fonction `odeint` à importer depuis le module `scipy.integrate`. Cette fonction est un outil puissant pour résoudre n'importe quel système comportant n équations différentielles d'ordre 1 couplées. Son utilisation est cependant délicate : voyons comment çà marche avec notre exemple simple pour lequel n = 2.

L'appel de la fonction `odeint` requiert trois arguments à donner dans cet ordre :

- une fonction Python notée OH (à écrire soi même) définissant le système différentiel à résoudre
- une liste notée W0 contenant les n conditions initiales
- un tableau numpy noté t contenant les N dates pour lesquelles on souhaite obtenir une approximation de la valeur prise par chacune des n fonctions inconnues.

Cette fonction renvoie un tableau (N lignes et n colonnes) noté Wsol. Il contient la solution numérique souhaitée.

```
Wsol = odeint(OH,W0,t)
```

Dans notre exemple, le système différentiel à résoudre comporte deux équations : n = 2. La première colonne de Wsol contient la solution numérique x_n associée à la fonction $x(t)$. La deuxième colonne contient la solution numérique $\dot{x}_n$ associée à la fonction $\dot{x}(t)$. On peut extraire ces solutions ainsi :

```
x = Wsol[:,0]
xp = Wsol[:,1]
```

Comment faut-il coder la fonction OH passée en argument de `odeint` pour qu'il en soit ainsi ?

L'appel de cette fonction requiert deux arguments notés W et t. A partir des n valeurs prises à l'instant t par les n fonctions inconnues placées **dans un certain ordre*** dans le tableau W, la fonction OH doit renvoyer un tableau contenant **dans le même ordre*** les valeurs des n dérivées premières à l'instant t de chacune de ces n fonctions.
* L'ordre en question est celui choisi dans W0. Dans notre exemple, n = 2 et l'on convient arbitrairement que le premier élément de W noté W[0] correspond à $x(t)$ et que le deuxième noté W[1] correspond à $\dot{x}(t)$. .

```
def OH (W,t):
    Wp = np.zeros(2)
    Wp[0] = W[1]
    Wp[1] = -omega0**2*W[0]
    return Wp
```

est ainsi la traduction en langage Python de :

$$\frac{dx}{dt} = \dot{x}(t) \qquad (3)$$

$$\frac{d\dot{x}}{dt} = -\omega_0^2 x(t) \qquad (4)$$

OH odeint.py

Vous êtes invités à lire ce script et à l'exécuter. On obtient une solution numérique bien plus satisfaisante qu'avec la méthode d'Euler explicite pour le même nombre de points (N = 100). Notons que cette efficacité est obtenue au prix d'un schéma numérique assez complexe qui est resté masqué pour l'utilisateur. Une bien belle boîte noire en somme !