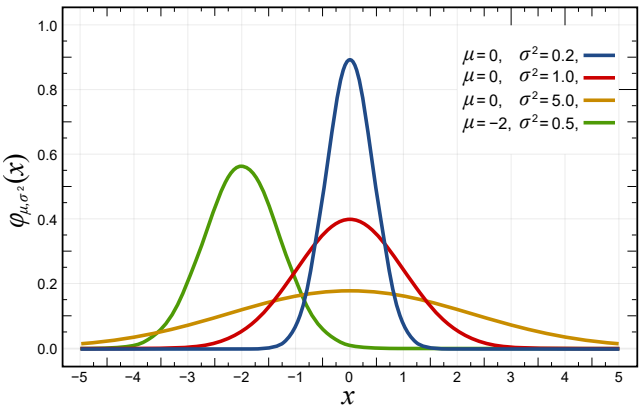
Quelques modules bien pratiques



Objectifs

A la fin de la séquence d'enseignement les élèves doivent :

- être en mesure d'importer des éléments ou tous les éléments d'une bibliothèque
- être en mesure de charger une bibliothèque et d'en utiliser les fonctions via un alias
- pouvoir utiliser les principaux outils des bibliothèques `random`, `numpy`, `scipy` et `matplotlib` pour résoudre des problèmes courants.

Table des matières

1 Bibliothèques couramment utilisées et leurs alias

Python est un langage de programmation partagé par une très grande communauté d'utilisateurs. Ainsi, de nombreux modules (ou bibliothèques) existent et offrent des outils performants dans des domaines très divers à travers des fonctions et des méthodes optimisées. Une page (https://www.lfd.uci.edu/ gohlke/pythonlibs/) recense une partie des ces modules, dans différentes versions de **Python** .

Certaines bibliothèques sont pré-installée avec **Python** (ex : math, random). D'autres nécessites une installation à la main en utilisant par exemple la commande `pip install module`.

Pyzo installe directement les modules les plus courants. Avec **Idle** , mieux vaut se reporter ici, sur mon site.

Nous utiliserons principalement les bibliothèques **math** (fonction mathématiques), **random** (tirage au sort), **numpy** (algèbre linéaire), **scipy** (calcul numérique) et **matplotlib** (pour tracer des courbes).

Rappel plusieurs méthodes pour appeler les méthodes et les fonctions d'une bibliothèque (y compris la votre) :

- `from math import *` : on importe la totalité de la bibliothèque mathématique
- `from math import sqrt, sin, pi` : on importe uniquement les fonctions souhaitées
- `import math as m` : on écrira `2 * m.sqrt(3)` (resp. `m.sin(pi / 4)`) pour calculer $2.\sqrt{3}$ (resp. $\sin(\pi/4)$).

J'utiliserai principalement les alias `m`, `rd`, `np`, `sciop` (pour scipy.optimize), `scint` (pour scipy.integrate) et `plt`.

2 Random - tirages aléatoires

Objectif : tirer au sort.

Très souvent, pour tester des programmes, il convient de créer des tests aléatoires. Le module **random**, qu'on chargera avec l'alias `rd`, permet de faire des tirages aléatoires avec différentes répartitions.

Citons principalement :

- `rd.random()` retourne un réel $\in [0;1[$ avec une distribution linéaire.
- `rd.randint(a, b)` retourne un entier $\in [\![a;b]\!]$ avec une distribution linéaire.
- `rd.gauss(mu,sigma)` retourne un réel $\in [0;1[$ avec une distribution gaussienne suivant la loi normale : $f(x) = \dfrac{1}{\sigma.\sqrt{2.\pi}} e^{-\frac{1}{2}.\left(\frac{x-\mu}{\sigma}\right)^2}$

3 Numpy - vers l'algèbre linéaire

3.1 Tableaux de `floats`

3.1.1 Répartitions linéaires ou logarithmiques

Objectif : obtenir un tableau avec des valeurs réparties de façons linéaire ou logarithmique.

- `np.linspace(deb, fin, nbp)` avec `deb` la première valeur, `fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.linspace(10, 50, 5)
array([10., 20., 30., 40., 50.])
```

```
>>> np.linspace(0.1, 1.1, 5)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.arange(deb, fin, pas)` avec `deb` la première valeur, `fin` la dernière valeur (exclues) et `pas` le pas entre deux éléments.

```
>>> np.arange(10, 60, 10)
array([10, 20, 30, 40, 50])
```

```
>>> np.arange(0.1, 1.2, 0.25)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.logspace(deb, fin, nbp)` avec `10**deb` la première valeur, `10**fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.logspace(-1, 2, 4)
array([  0.1,   1. ,  10. , 100. ])
```

```
>>> np.logspace(0, 1, 3)
array([ 1.        ,  3.16227766, 10.        ])
```

3.1.2 Vectorisation

Contrairement aux listes, il est possible d'appliquer des opérations directement à un tableau **numpy**.

```
>>> T = np.linspace(0, np.pi/2, 7)
>>> np.cos(T)
array([1.00000000e+00, 9.65925826e-01, 8.66025404e-01, 7.07106781e-01,
       5.00000000e-01, 2.58819045e-01, 6.12323400e-17])
```

3.2 Charger un fichier de points

Le module `numpy` propose la méthode `loadtxt` pour lire un fichier de points. Dans un fichier `.csv`, les colonnes sont séparées par des `;`. Ainsi `donnees = np.loadtxt('donnees.csv', delimiter = ';')` permet d'obtenir un tableau `donnees` avec les valeurs du fichier `donnees.csv`.

Si on connait le nombre de colonnes, il est possible d'associer une variable à chaque colonne avec l'argument optionnel `unpack`. Si les colonnes sont séparées par des tabulations, il faut utiliser `\t` comme valeur de `delimiter`.

```
t, x, v = np.loadtxt('donnees.txt', delimiter = '\t', unpack = True)
```

3.3 Effectuer une régression linéaire

Lorsqu'on veut récupérer le coefficient directeur a et l'ordonnée à l'origine b d'une droite passant *au plus près* d'un ensemble de points, on peut utiliser la fonction `polyfit`.

Comme `polyfit` ne s'appelle pas *régression linéaire*, on comprend l'intérêt du `1` dans l'argument. On peut donc faire plus puissant !

```
>>> X = np.linspace(-2, 4, 1000)
>>> Y = [3*x + 1 for x in X]
>>> a, b = np.polyfit(X, Y, 1)
>>> print(a, b)
3.0 1.0
```

3.4 Algèbre linéaire - système de Cramer

Objectif : Résoudre un système linéaire du type A.X = B où A est une matrice inversible.

Pour faire de l'algèbre linéaire avec **Python** , le plus simple peut être d'utiliser le module `numpy.linalg` contenant les méthodes `det(A)` pour le calcul du déterminant de `A`, `inv(A)` pour le calcul de la matrice inverse de `A` et `solve(A,B)` pour obtenir la solution du problème linéaire A.X = B (où X est l'inconnue).

La classe `array` (tableaux **numpy**) possède beaucoup de méthodes notamment `T` pour obtenir la transposée d'un tableau et `dot(B)` pour multiplier un tableau à droite par `B`.

```
A = np.array([[2, 1, -1], [6, -1, -2], [-6, -2, 3]])
## Vérification du déterminant
print(nalg.det(A))
X = np.array([2, 3, 7]).T
## Produit matriciel de A par B
B = A.dot(X)
```

```
## Résolution par calcul
## de la matrice inverse
inv_A = nalg.inv(A)
X_sol_1 = inv_A.dot(B)
## Résolution avec solve
X_sol_2 = nalg.solve(A,B)
```

4 Matplotlib - module graphique

Pour tracer des courbes sous **Python** , plusieurs bibliothèques existent. L'une des plus pratiques et utilisées est **matplotlib**. Cependant, on chargera un des ses sous modules `matplotlib.pyplot` sous l'alias `plt`.

```
import matplotlib.pyplot as plt
```

Remarque : nous utiliserons aussi `plt` pour le traitement des images mais cela fera l'objet d'un autre cours. Patience !

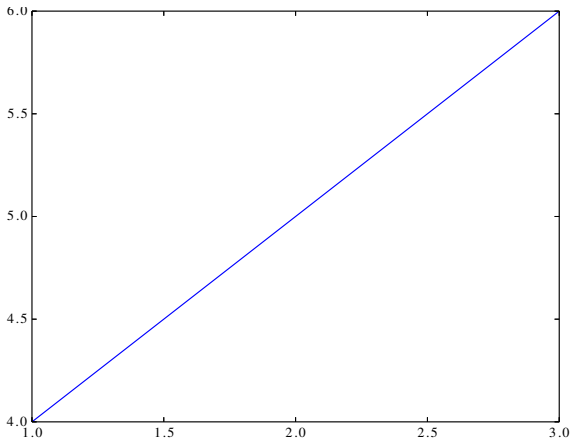
4.1 Tracer une courbe

Pour tracer des courbes, on utilise l'instruction `plot`. Pour afficher les figures, on utilise l'instruction `show()` .

Les « courbes » tracées sont en fait des segments de droites dont les extrémités sont définies par les éléments successifs de deux listes (par défaut, la liste des abscisses et celle des indices de la liste des ordonnées).

```
x = [1,2,3]
y = [4,5,6]
plt.plot(x,y)
plt.show()
```

ce programme donne :



Remarque : ajouter l'option `block=False` dans `show()` permet de ne pas avoir à fermer la figure pour continuer le programme quand on utilise **Idle** .

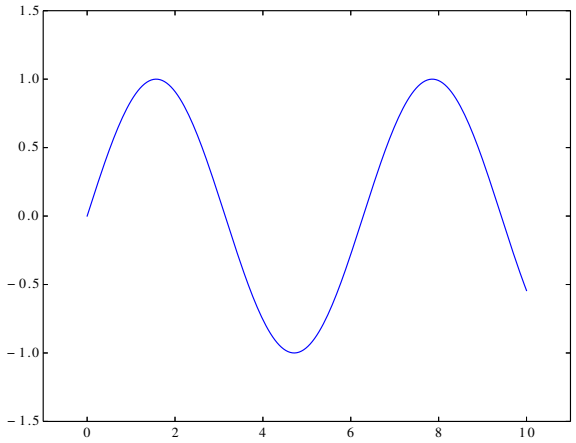
L'effet secondaire, c'est que la figure suivante se construit sur la figure précédente. Il est possible d'effacer celle d'avant avec `plt.clf()` ou de numéroter les figures avec `plt.figure(n)`. L'avantage, c'est qu'on peut écrire sur une figure, puis une autre, puis revenir à la première. Pour avoir toutes les figures sans se poser de questions, on peut mettre `nfig=0` en début de programme et à chaque nouvelle figure, écrire `nfig+=1` puis `plt.figure(nfig)`.

Lorsqu'on utilise **Pyzo** , il faut bien penser à nettoyer la figure quand on ré-exécute le programme car elle demeure contrairement à ce qui se passe avec **Idle** .

4.2 Définition des axes

Pour définir les domaines des axes, on peut utiliser les instructions `xlim(xmin,xmax)` et `ylim(ylim,ymax)`.

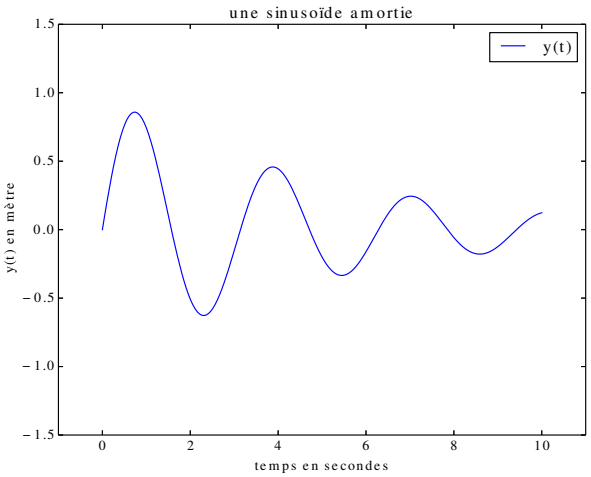
```
t = np.linspace(0, 10, 200)
y = [m.sin(i) for i in t ]
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5)
plt.plot(t, y)
plt.show(block = False)
```



4.3 Mettre les légendes

On utilise la fonction `title()` pour rajouter un titre, la fonction `legend()` pour rajouter une légende (il faut obligatoirement utiliser `label` pour que la légende soit correctement écrite), et les instructions `xlabel()` et `ylabel()` pour rajouter un titre aux axes.

```
plt.title("une sinusoide amortie")
plt.xlabel("temps en secondes")
plt.ylabel(" y(t) en metre")
plt.plot(t, y, label = "y(t)")
plt.legend()
```



4.4 Modifier le style des courbes

Il est possible de changer la couleur des courbes, le style des courbes, le symbole des courbes (ou marqueur, *marker*), et la largeur des courbes.

Chaîne	Effet
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker

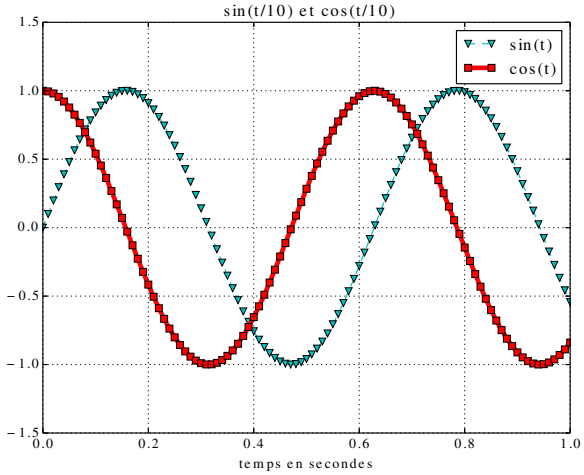
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker

3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker

+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

Chaîne	b	g	r	c	m	y	k	w
Couleur	bleu	vert	rouge	cyan	magenta	jaune	noir	blanc

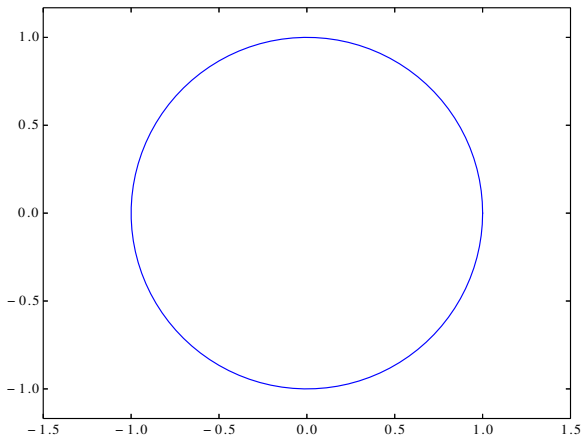
```
t, x , y = [], [], []
for i in range(101):
    t.append(i/100)
    x.append(m.sin(t[i]*10))
    y.append(m.cos(t[i]*10))
plt.ylim(-1.5,1.5)
plt.plot(t, x , 'c--v' ,label="sin(t)")
plt.plot(t, y, 'r-s', label="cos(t)", linewidth=4)
plt.legend()
plt.xlabel("temps en secondes")
plt.title(" sin(t/10) et cos(t/10)")
plt.grid()#permet d'ajouter une grille
```



4.5 Isométrie

Pour *contraindre* les axes à avoir la même graduation en abscisse et en ordonnée, on place l'instruction `axis('equal')` avant l'insertion des bornes.

```
n, x, y = 100, [], []
t=[i*2*m.pi/(n-1) for i in range(n)]
for i in range(n):
    x.append(m.cos(t[i]))
    y.append(m.sin(t[i]))
plt.axis('equal')
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.plot(x, y)
```



4.6 Échelle logarithmique

Il n'est pas rare en science d'avoir besoin d'échelles logarithmique. En physique et en SI, on utilise les diagrammes de Bode qui fonctionnent avec une échelle semi-logarithmique. Pour tracer une diagramme semi-logarithmique en x on utilise `semilogx()` et pour un tracé log-log on utilise `loglog()`.

```
plt.semilogy(np.logspace(-2, 4, 1000))
plt.grid('on', which='both') #permet de tracer le quadrillage 'fin'
plt.show()
```

4.7 Tracer plusieurs figures dans une même fenêtre

On utilise `subplot()` pour afficher plusieurs figures dans une même fenêtre.

- le premier argument de la fonction est le nombre de lignes de figures désiré
- le deuxième, le nombre de colonnes
- le troisième argument, donne le numéro de la figure

Python remplit alors les figures ligne par ligne comme on remplirait les cases d'un tableau.

Ainsi l'exemple précédent donne :

```
plt.subplot(2,1,1)
plt.plot(np.logspace(-2, 4, 1000), label = 'echelle lineaire')
plt.grid('on', which = 'both')
plt.xlabel("numero du points")
plt.legend()

plt.subplot(2, 1, 2)
plt.semilogy(np.logspace(-2, 4, 1000), label = 'echelle logarithmique en y')
plt.grid('on', which = 'both')
plt.xlabel("numero du points")
plt.legend()

plt.suptitle("Echelles logarithmique et lineaires")
plt.show()
```

5 Module Scipy

5.1 scipy.optimize

Pour résoudre un problème du type $f(x) = 0$, le plus simple est d'utiliser le module `scipy.optimize` avec la méthode `newton`. Ainsi, `sciop.newton(f, x0)` permet de déterminer un zéro de la fonction `f` en partant de `x0`. Il n'est pas nécessaire de donner la dérivée `fp` de `f` mais il est possible de le faire `sciop.newton(f, x0, fp)`.

```
def f(x):
    return x**2/2 - 1

def fp(x):
    return x

sol = sciop.newton(f, 3, fp)
```

Rappel il n'est pas nécessaire de déclarer les fonctions avant d'utiliser `newton`. La fonction `lambda` permet de faire lors de l'appel.

```
>>> sciop.newton(lambda x: x**2/2 - 1, 3)
1.4142135623730971
```

5.2 scipy.integrate

5.2.1 Intégrer une fonction ou une liste de points

Objectif : obtenir une approximation de $\int_a^b f(x).dx$

Pour obtenir une approximation d'une intégrale simple, double ou triple d'une fonction, le module `scipy.intergrate` propose les fonctions `quad`, `dblquad` et `tplquad`.

Ainsi `quad(f, a, b)` renvoie un tuple contenant une approximation de $\int_a^b f(x).dx$ et une estimation de l'erreur commise.

```
>>> scint.quad(lambda x : x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

Utiliser `cumtrapz` du module `scipy.integrate` permet d'intégrer une liste de valeurs, par exemple pour passer des mesures d'accélération à la vitesse en chaque point de mesure.

Ainsi `scipy.intergrate.cumtrapz(val, x=t, initial=i0)` permet d'obtenir l'évolution de l'aire sous la courbe formée par les éléments de `val`, avec `t`, la liste de leurs abscisses et `i0` la valeur initiale.

Remarque : la fonction ne semble pas tenir compte de la valeur initiale. Le troisième exemple serait peut-être la

solution à adopter pour obtenir le résultat recherché :

```
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=3)
array([3. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0)
array([0. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0) +3
array([3. , 5.5, 9. ])
```

5.2.2 Intégrer une équation différentielle

Objectif : Résoudre une équation différentielle du type $\vec{Y}' = \vec{F}(\vec{Y}, t)$.

Le module `scipy.integrate` propose la fonction `odeint` telle que `odeint(F, Y0, T)` permet d'obtenir une estimation de la solution, pour chaque valeur de `T`, de l'équation différentielle $Y' = F(Y,t)$ en partant de `T[0]` avec comme valeur de l'état initial `Y0`. L'état du système peut contenir N variables avec $N \in \mathbb{N}^*$.

En ramenant l'équation différentielle scalaire du second ordre (1), à une équation différentielle vectorielle du premier ordre (2), le problème de Cauchy initial (3) peut être traité avec `odeint`.

$$\begin{cases} \frac{1}{\omega_0^2}.y''(t) + \frac{2.\xi}{\omega_0}.y'(t) + y(t) = \mathrm{K}.(u(t-t_0) - u(t-t_1)) \\ y(t_0) = y_0 \text{ et } y'(t_0) = v_0 \end{cases} \quad (1)$$

avec $u(t)$ l'échelon unitaire, nul si t est négatif, égale à 1 sinon.

$$\begin{cases} \frac{\mathrm{d}y(t)}{\mathrm{d}t} &= v(t) \\ \frac{\mathrm{d}v(t)}{\mathrm{d}t} &= (\mathrm{K}.(u(t-t_0) - u(t-t_1)) - y(t)).\omega_0^2 - 2.\xi.\omega_0.v(t) \\ y(t_0) &= y_0 \\ v(t_0) &= v_0 \end{cases} \quad (2)$$

$$\begin{cases} \frac{\mathrm{dY}(t)}{\mathrm{d}t} &= \mathrm{F}(\mathrm{Y}, t) \\ \mathrm{Y}(t_0) &= [y_0, v_0] \end{cases} \quad (3)$$

avec $\mathrm{Y}(t) = [y(t), v(t)]$

```
def sys_amort(Y, t):
    if t < t0 or t > t1:
        u = 0
    else:
        u = 1
    return [Y[1], (K*u - Y[0]) * w0**2 - 2*xi*w0*Y[1]]
```

- fonction associé au problème de Cauchy
- définition des paramètres et simulations
- présentation des résultats

```
K, xi, w0, t0, t1, y0, v0 = 3, 0.25, 20, 0, 1, 0, 0
T = np.linspace(t0-1, t1+1, 200)
sim = scint.odeint(sys_amort, [y0, v0], T)
```

```
plt.figure(0)
plt.title(r'Déplacement $y(t)$')
plt.plot(T, sim[:,0])
plt.figure(1)
plt.title(r'Vitesse $v(t)$')
plt.plot(T, sim[:,1])
```