Python pour scientifique

Pierre Cladé

sept. 28, 2022

Table des matières

CHAPITRE 1

Prise en main

1.1 Qu'est ce que Python

L'objectif d'un langage informatique est de décrire un ensemble d'instructions que l'on souhaite voir exécuter par un ordinateur. Le langage que comprend l'ordinateur (l'assembleur) est un langage trop primitif pour être simple et rapide à utiliser. C'est pour cela qu'il existe des langages dits évolués.

Parmi ces langages on peut distinguer les langages compilés (par exemple, le C, le Fortran, le C++, …) et les langages interprétés (basic, python, matlab, scilab, …). Dans le premier cas, le programme est entièrement traduit (compilé) pour être exécuté directement par le processeur (on peut prendre l'analogie d'un livre traduit entièrement dans une autre langue). Les langages interprétés sont traduits au fur et à mesure par un interpréteur (pour continuer l'analogie, il s'agit d'un traducteur qui traduit les phrases au fur et à mesure). Les langages interprétés ont la réputations d'être plus lent, cependant, il sont bien plus facile à utiliser : on peut faire une modification sans avoir à recompiler, ou simplement voir le résultat de chaque instruction au fur et à mesure de l'exécution.

Le langage Python est un langage interprété. Il a été créé en 1995 avec pour objectif d'être

un langage informatique général (par opposition aux langages spécialisés pour effectuer un type de tache bien précis). Python fait parti des langages informatiques les plus utilisés au monde. Python est devenu au cours des années un langage interprété de premier choix. Il est présent dans beaucoup de projet open source (par exemple LibreOffice).

1.2 Installer Python

1.2.1 L'interpréteur Python

Il existe plusieurs interpréteurs pour Python. Citons principalement CPython, qui est l'interpréteur de référence écrit en C, JPython (écrit en Java), pypy (un interpréteur plus rapide, partiellement écrit en Python !), IronPython (écrit en C#). Il existe aussi des interpréteurs Python pour programmer des micro-processeur (python-on-chip, pymite, MicroPython <https://micropython.org/>). Ces derniers ne peuvent interpréter qu'un partie du langage Python. Nous utiliserons CPython.

Python est un langage qui évolue au cours des années. La définition du langage est faite de facto par l'interpréteur standard CPython et on assimile la version de langage à celle de cette interpréteur. La principale version de python utiliser est la version 3.

1.2.2 La distribution Python

Il existe plusieurs distributions de Python qui permettent d'installer facilement l'interpréteur CPython déjà compilé ainsi que les librairies supplémentaires. Dans ce cours, nous utiliserons la distribution Anaconda <https://www.anaconda.com/distribution/> qui offre plusieurs avantages : elle est disponible sous Windows, Mac et Linux. Elle offre aussi la possibilité d'avoir une installation indépendante du système (ce qui est particulièrement utile sous Linux où Python est souvent installé par défaut). En installant Anaconda, la plupart des logiciels permettant de travailler avec Python seront installés (notons IPython, jupyter notebook, spyder..). De même les librairies scientifiques de bases seront aussi installées.

Nous laissons le lecteur suivre en ligne les instructions d'installation propre à son système.

1.3 Utilisation de Python

Il existe plusieurs façons d'exécuter un code Python. On peut lancer un interpréteur dans une console, ce qui permet d'exécuter les instructions ligne par ligne. On peut aussi écrire le script dans un ficher que l'on exécute. Enfin, il existe des solutions intermédiaires avec lesquels on utilise à la foi un fichier et une console.

1.3.1 Console Python

La console Python se lance simplement à partir du programme `python`. Sous windows, on peut ouvrir un terminal depuis le menu `Anaconda`.

La console Python peut être utilisée comme une simple calculatrice :

```
>>> 1+13
14
```

On peut aussi y entrer des structures plus complexes directement :

```
>>> l = ['alpha', 'beta']
>>> for elm in l:
...     print(elm)
...
alpha
beta
```

1.3.2 Console IPython

La console IPython est beaucoup plus pratique à utiliser que la console Python. Elle remplace de fait la console standard. La page <https://ipython.org/ipython-doc/2/interactive/tutorial.html> indique les principales fonctions de IPython. Notons :

— Une console qui permet d'accéder au commandes précédentes (en utilisant la flèche du haut). Il est possible de limiter la recherche en commençant à taper les premiers caractères de la lignes que l'on veut retrouver

— La complétion automatique à l'aide de la touche de tabulation.

```
In [1]: une_variable_dont_le_nom_est_long = 3.2
```

On pourra ensuite simplement taper quelques lettres (par exemple `une_va`) puis la touche de tabulation. Si plusieurs choix sont possibles, ils s'afficheront. Ceci est valable pour les noms de variables (donc aussi pour les fonctions), mais aussi les attributs ou méthodes des objets.

— Les commandes magiques. Sera utilisé particulièrement la commande `%run` qui permet d'exécuter un fichier.

1.3.3 Utilisation d'un fichier

Dès que l'on veut exécuter plus que quelques instructions, il est préférable d'utiliser un fichier. Les fichiers contenant des instructions python ont comme extension `.py`. Écrivons à l'aide d'un éditeur de texte standard les quelques instructions suivantes :

```
nom = 'Pierre'
print('Bonjour', nom, '!')
```

Ce fichier peut être exécuté directement à partir d'un shell à l'aide de l'instruction `python nom_de_fichier.py`. Normalement, ce programme devrait afficher :

```
Bonjour Pierre !
```

1.3.4 Spyder

Spyder offre un environnement de travail pour Python qui se veut un clone de celui de Matlab. Il est donc adapté à l'utilisation pour les scientifiques. En ouvrant ce programme, on a d'un côté un éditeur de texte et de l'autre une fenêtre d'aide et une console IPython.

Exemple : lancer spyder et ouvrir le fichier créé précédemment. Exécuter le contenu de ce fichier (à partir des menus ou directement avec la touche F5).

1.3.5 Jupyter Notebook

Le jupyter notebook offre un environnement dans lequel on peut afficher à la foi les instructions (regroupés dans dans des cellules) et les résultats de ces instructions au fur et a mesure de leur exécutions. Il est par exemple possible d'afficher des graphiques. Cet environnement est particulièrement intéressant lorsque l'on veut pouvoir présenter directement des résultats (par exemple pour un TP, ou une simulation).

Jupyter se présente sous la forme d'un serveur http auquel on accède avec un navigateur web. Sous linux, lancer simplement le notebook à partir d'un terminal à l'aide de la commande

```
jupyter notebook
```

Normalement, le navigateur par défaut devrait s'ouvrir.

Sous windows, il existe un lien depuis le menu `anaconda`.

Pour tout ce qui est petit exercices ou traitement de données, nous conseillons fortement d'utiliser des Notebook Jupyter, ceci l'offre l'avantage de présenter le code et les résultats sequentiellement sur une même page que l'on peut alors facilement partager.

1.4 Exemple de code

Voici un premier exemple de code Python. L'objectif de ce code est de calculer la valeur de e^x. Nous allons utiliser le développement limité suivant :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Pour cela, nous allons calculer la somme jusqu'à une valeur $n_{\max}$ de n telle que $x^n/n! < \epsilon$. La variable ϵ déterminera la précision du calcul (plus ϵ est petit, meilleure est la précision).

Le code Python permettant de faire ce calcul est le suivant :

```
x = 3.14
epsilon = 1E-6
resultat = 0
n = 1
terme = 1 # Valeur initiale de terme de la boucle
while terme>epsilon :
    resultat = resultat + terme
    terme  = terme * x/n
    n = n + 1
print(resultat)
```

Ce code ne devrait pas poser de problème au lecteur ayant déjà eu des cours de programmation. Une particularité du langage Python est présente : dans la boucle `while`, le bloc d'instruction qui est répété est déterminé par l'indentation (espaces au début de chaque

ligne). Python est donc différent de la pluspart des langages qui utilisent une structure du type `begin-end` ou bien des accolades pour déterminer le bloc d'instruction qui est répété. En python un bloc d'instruction est repéré par les `:` et un ensemble de lignes indenté identiquement.

Pour effectuer l'indentation d'une ligne ou d'un bloc de ligne, le plus simple est d'utiliser la touche de tabulation (et shift + tabulation) pour supprimer l'indentation.

Exercice

Essayez d'exécuter ce code en utilisant les différentes méthodes proposées ci dessus :

— En recopiant le script dans un fichier `exponentielle.py` et en l'exécutant avec la commande `python exponentielle.py`

— En utilisant ipython

— En utilisant spyder

— En utilisant un notebook jupyter.

Pour créer une fonction, on utilise l'instruction `def`, ce qui donne :

```
def exp(x, epsilon=1e-6): # epsilon vaut par défaut 1e-6
    """ Renvoie e a la puissance x """
    resultat = 0
    n = 1
    terme = 1 # Valeur initiale du terme de la boucle
    while terme>epsilon :
        resultat = resultat + terme
        terme = terme * x/n
        n = n + 1
    return resultat
```

Exercice

Depuis un éditeur (par exemple spyder), modifiez le fichier pour en faire une fonction. Exécutez le fichier et la fonction.

CHAPITRE 2

Les bases de Python

2.1 Les types de données en Python

Il s'agit d'une introduction basique sur les types de données offerts par python. Cette introduction peut servir d'aide mémoire pour les utilisations les plus courantes, sachant que l'aide officiel de python est beaucoup plus complète.

2.1.1 Les nombres

C'est le type le plus simple. Il existe trois types de nombre :

— Les entiers (type `int`). Par exemple `a = 5`, en binaire `a = 0b1001`, en hexadécimal `a = 0x23`.

— Les nombres à virgules flottante (type `float`). Par exemple `a = 1234.234` ou `a = 6.2E-34`. Les nombres sont enregistré en double précision [1] (64 bits). La précision relative est de 52 bits, soit environ 10^{-16}. :

1. http://fr.wikipedia.org/wiki/IEEE_754

```
>>> a = 3.14
>>> a == a + 1E-15
False
>>> a == a + 1E-16
True
```

— Les nombres complexes (type `complex`). Il sont enregistrés sous la forme de deux nombres à virgule flottante. Par exemple `a = 1 + 3J`.

Les opérations sur les nombres sont les suivantes :

— somme : +

— produit : *

— différence ou négation : –

— division : /

— division entière : //

— modulo : % (par exemple `7%2`)

— puissance : ** (par exemple `2**10`)

Pour les nombres complexes, on peut facilement accéder à la partie réelle et imaginaire de la façon suivante : :

```
>>> a = 1 + 3J
>>> a.imag
3.0
>>> a.real
1.0
>>> print("Norme ", sqrt(a.real**2 + a.imag**2))
Norme  3.1622776601683795
```

2.1.2 Les booléens et comparaison

Il existe deux valeurs : `True` et `False` (attention à la casse). Les opérations sont par ordre de priorité : `not`, `and` et `or`. Les comparaisons se font à l'aide des symboles <, <=, ==, > et >=. Pour savoir si deux valeurs sont différentes, on utilise `!=`.

Les opération `and` et `or` effectuent en fait un test conditionnel. L'instruction `A and B` est interprété comme `B if not A else A`, de même `A or B` équivaut à `A if A else B`.

Avertissement : Les symboles `&` et `|` sont des opérateurs binaires. Ils réalisent les opérations and et or sur les entiers bit par bit en binaire (par exemple `6 & 5` donne `4`). Il ne faut pas les utiliser pour les opérations sur des booléens.

2.1.3 Les chaînes de caractères (string)

Création d'une chaîne de caractères

Il existe trois façons de créer une chaîne de caractère : avec des `'`, des `"` et des `"""`. Ces caractères servent à délimiter les début et la fin du texte de la chaîne de caractère. Les guillemets simples `'` et doubles `"` sont équivalents. On pourra choisir l'un ou l'autre. Il sera cependant judicieux, si une chaîne de caractère doit contenir un de ces guillemets, d'utiliser l'autre pour le début et la fin de la chaîne. Les trois guillemets sont eux utilisés lorsque l'on veut qu'une chaîne de caractère soit sur plusieurs lignes.

Voici quelques exemples :

```
>>> s = 'Pierre'
>>> s = "Aujourd'hui" #Rq : s = 'Aujourd'hui' ne va pas␣
↪fonctionner
>>> s = """Aujourd'hui, le petit enfant a dit:
...  "Faisons le clown!" """
```

Caractères spéciaux

Les **caractères spéciaux** sont les caractères qui ne sont pas affichables et en tant que tel. Par exemple, il existe un caractère pour le retour à la ligne. Il est possible d'utiliser ce caractère dans une chaîne en utilisant `\n`. L'antislash sert ici de caractère d'échappement pour indiquer que l'on va entrer un caractère spécial. La lettre `n` indique ici qu'il s'agit d'un retour à la ligne.

Voici un exemple :

```
>>> s = "Un\nDeux"
>>> print(s)
Un
Deux
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> len(s) # \n compte pour un caractère
7
```

L'antislash sert aussi à insérer un guillemet dans une chaîne : `'Aujourd\'hui'`. Si on veut mettre un antislash dans un chaîne, il faut le précéder d'un autre antislash « \def ». Si on ne souhaite pas que Python interprète ces caractères spéciaux, il est possible de précéder la déclaration de la chaîne d'un `r` :

```
>>> a = r"\theta"
>>> print(a)
\theta
```

Manipulation des chaînes de caractères

Tout comme une liste, il est possible d'accéder à chaque caractère d'une chaîne ou à une partie d'une chaîne.

```
>>> s = "Pierre"
>>> s[0]
'P'
>>> s[1:3]
'ie'
```

La longueur de la chaîne s'obtient avec la fonction `len`. On peut aussi faire une boucle `for` sur chacun des éléments de la chaîne.

Cependant, il n'est pas possible de modifier une chaîne de caractères (l'opération `s[0]='p'` échoue).

La **concaténation** est l'opération qui consiste à créer une nouvelle chaîne en mettant à la suite deux chaînes de caractères. Elle se fait à l'aide du signe +. Par exemple :

```
>>> s1 = 'Un'
>>> s2 = 'Deux'
>>> s1+s2
'UnDeux'
```

Une autre opération est importante, il s'agit du **formatage** d'une chaîne de caractère. Cette opération consiste à insérer un élément variable dans une chaîne. Elle est souvent utilisée lorsque l'on veut afficher proprement un résultat. Voici une exemple :

```
>>> heure = 15
>>> minute = 30
>>> "Il est {0}h{1}".format(heure, minute)
'Il est 15h30'
```

Pour insérer un élément ou plusieurs éléments variables dans une chaîne de caractère, on crée d'abord cette chaîne en mettant à la place des ces éléments une accolade avec un numéro d'ordre `{i}`. En appliquant la méthode `format` sur cette chaîne, les accolades seront remplacées par le ième argument.

Il est possible de passer l'argument par nom dans ce cas la clé est le nom de l'argument.

```
>>> "Il est {heure}h{minute}".format(heure=heure, minute=minute)
'Il est 15h30'
```

Depuis la version 3.6 de Python, il est possible de demander à Python d'utiliser automatiquement les variables locales à l'aide du préfix `f`.

```
>>> f"Il est {heure}h{minute}"
'Il est 15h30'
```

Il est aussi possible de demander d'utiliser un attribut d'un objet :

```
>>> z = 1 + 2J
>>> print(f'Re(z) is {z.real}')
Re(z) is 1.0
```

En utilisant le formatage de chaîne de caractère, il est possible de spécifier en détail comment ce nombre doit s'afficher. Par exemple, si il s'agit d'un nombre à virgule flottante, combien de décimales faut-il afficher, faut il utiliser la notation scientifique, etc. Pour cela, on rajoute à l'intérieur des accolades un code particulier. Ce code est précédé du signe " :".

Voici quelques exemples :

```
>>> from math import pi
>>> '{0:.5f}'.format(pi)
'3.14159'
>>> c = 299792458. # Vitesse de la lumière en m/s
>>> 'c = {0:.3e} m/s'.format(c)
'c = 2.998e+08 m/s'
```

Le "f" indique que l'on veut une notation a virgule fixe, le "e" une notation scientifique. Le chiffre que l'on indique après le "." donne le nombre de chiffre après la virgule que l'on

souhaite.

Note : L'aide en ligne de Python fournit d'autres exemples et des détails.

Il existe aussi une façon plus élémentaire de formater des chaîne de caractères avec Python et qui est obsolète (mais que l'on peut rencontrer). Pour formater le nombre `pi`, cette méthode écrira dans ce cas `'%.6f'%pi`.

2.1.4 Les listes (list)

— On peut créer et remplir une liste de plusieurs manières :

```
>>> l = [1, 2, 3, 4]
>>> l = [] # liste vide
>>> l.append(3)
>>> l
[3]
>>> l.append(4)
>>> l
[3, 4]
>>> l.insert(0,3.24+1j)
>>> l
[(3.24+1j), 3, 4]
>>> len(l) # Longueur de la liste
3
```

— On peut changer un élément de la liste :

```
>>> l[2] = 23
```

— On peut créer une nouvelle liste à partir d'une liste à l'aide de la commande `list` :

```
>>> l = [2, 3]
>>> m = list(l)
>>> l.append(45)
>>> l==m
False
```

— La commande `range(n)` peut être utilisée pour créer la liste `[0,1,2, .., n-2, n-1]`. Cette liste commence à `0` et contient `n` nombres (elle s'arrête donc à `n-1`).

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Techniquement, depuis la version 3 de Python, range ne renvoie plus une liste, mais un générateur. Ce générateur produit la suite de chiffre qui est ensuite utilisée par le constructeur de la liste.

— Une liste peut contenir des éléments de types différents.

— Souvent on a besoin de créer une liste dans une boucle. Par exemple la liste des `n` premiers nombres au carré peut se calculer de la façon suivante

```
l = []
for i in range(n):
    l.append(i**2)
```

Il existe cependant une façon plus directe de faire, en utilisant les `list comprehension`

```
l = [i**2 for i in range(n)]
```

Cette syntaxe se lit directement en français : calcule la liste des i au carré pour i allant de 0 à n. Il est aussi possible de rajouter une condition (pour i allant de 0 à n si i est pair)

l = [i**2 for i in range(n) if i%2==0]

— Si on souhaite appliquer une fonction à tous les éléments d'une liste pour recréer une liste, il est possible d'utiliser la fonction `map`

```
def mafonction(i):
    if i%2==0:
        return i**2
    else:
        return i**2-1

map(mafonction, range(5)) # [0,0,4,8,16]
```

— On peut parcourir une liste de plusieurs façons

```
l = [1, 3, 5, "Pierre"]
for elm in l:
    print(elm)

for i,elm in enumerate(l):
    print(elm, " est l'element numero ",i," de la liste)
```

Avec les listes, nous avons donc introduit la structure de contrôle `for`. Celle ci fonctionne sur tout type de séquence (par exemple les chaînes de caractères). Par cette syntaxe, elle est assez différente de ce qui existe dans d'autres langage informatique.

Note : Nous avons vu qu'il était possible de rajouter des éléments dans une liste. Pour cela, nous avons utilisé la syntaxe `l.append(elm)`. Il s'agit ici de programmation orientée objet. Il est utile de savoir reconnaître la syntaxe : la liste est suivi du nom de la fonction que l'on applique (on appelle cette fonction une *méthode*) avec ses arguments. Ici, donc, `append` rajoute `elm` à la liste `l`.

Notons de plus, que l'objet `l` est modifié. La variable `l` désigne toujours la même liste, mais cette liste est modifiée (si on ajoute une page à un classeur, il s'agit toujours du même classeur). Ce comportement est différent de celui que nous avons vu dans la paragraphe précédent avec la méthode `format` sur les chaînes de caractères. En effet dans ce cas, on crée un nouvelle chaîne de caractère qui est renvoyé par la méthode - la chaîne initiale n'étant pas modifiée. La méthode `append` appliquée à une liste, modifie la liste, mais ne renvoie rien.

— Pour trier une liste, on peut soit utiliser la méthode `.sort` qui *modifie* la liste soit la fonction `sorted` qui renvoie une nouvelle liste triée. Il est possible d'ajouter comme argument optionnel une fonction qui donne l'ordre. Par défaut, python utilise la fonction cmp (ordre croissant pour les nombre et alphabétique pour le chaîne)

```
l = ['Pour', 'trier', 'une', 'liste', 'on', 'peut']
print(sorted(l))

def compare(a,b):
    u""" Ordre alphabétique inversé sans tenir compte
    de la casse"""
    return cmp(b.lower(), a.lower())

sorted(l, compare)
```

— Fonction `zip` : lorsque l'on veut parcourir deux listes en même temps, il est possible d'utiliser la fonction zip qui crée alors une liste de n-uplets à partir de `n` listes. :

```
>>> liste_nom = ['Martin', 'Lefevre', 'Dubois', 'Durand']
>>> liste_prenom = ['Emma', 'Nathan', 'Lola', 'Lucas']
```

```
>>> for nom, prenom in zip(liste_nom, liste_prenom):
...     print(prenom, nom)
Emma Martin
Nathan Lefevre
Lola Dubois
Lucas Durand
```

2.1.5 Les n-uplets (tuple)

Les n-uplets sont utilisés lorsque l'on veut regrouper des valeurs ensembles. On utilise une liste lorsque l'on a une longue séquence (dont la longueur est souvent variable) et un n-uplet pour regrouper quelques éléments. Par exemple :

```
>>> personne = ('Jean', 'Dupont', '13 juillet 1973', 38)
>>> print('Nom :',personne[1])
Nom : Dupont
```

Les n-uplets sont utilisés lorsqu'une fonction renvoie plusieurs éléments :

```
>>> def fonction(x):
...     return x**2,x**3
>>> a = fonction(4)
>>> a
(16, 64)
>>> a,b = fonction(4)
```

2.1.6 Les dictionnaires (dictionnary)

Contrairement aux listes ou aux n-uplets qui sont indexés par des nombres (`l[2]`), un dictionnaire est indexé par une clé. En général, la clé est une chaîne de caractère ou un nombre. On peut reprendre l'exemple précédent :

```
>>> personne = {"Prenom":"Jean", "Nom":"Dupont",
...   "date_naissance":"13 juillet 1973", "Age":38}
>>> print("Nom :",personne['Nom'])
Nom : Dupont
```

Tout comme pour une liste, on peut ajouter ou enlever des éléments à un dictionnaire :

```
>>> personne['telephone'] = "02 99 79 24 58"
>>> del personne['Age']
```

Il est possible de parcourir et accéder à un dictionnaire de plusieurs façons :

```
>>> releve_note = {'Jacques':15, 'Jean':16, 'Pierre':14}

# Par clef
>>> for nom in sorted(releve_note.keys()):
...     print('La note de {0} est {1}'.format(nom, releve_
↪note[nom]))
La note de Jacques est 15
La note de Jean est 16
La note de Pierre est 14

# Par valeurs
>>> notes = releve_note.values()
>>> print("Moyenne", sum(notes)/len(notes))
Moyenne 15.0
```

```
# Par clef et valeur
>>> for cle, val in releve_note.items():
...     print('La note de {0} est {1}'.format(cle, val))
La note de Pierre est 14
La note de Jean est 16
La note de Jacques est 15
```

La clé d'un dictionnaire doit être un objet non modifiable, c'est à dire un nombre, une chaîne de caractère ou un tuple contenant uniquement des objets non modifiables.

2.1.7 Les ensembles (set)

Utilisé pour représenter un ensemble non-ordonné d'éléments qui sont tous différents. Par exemple :

```
>>> a = set([1,2,3])
>>> b = set([3,5,6])
```

```
>>> c = a | b # union
>>> d = a & b # intersection
```

Un exemple d'application :

```
>>> mdp = raw_input('Entrez un mot de passe contenant \
...                         au moins un signe de ponctuation')
>>> ponctuation = set("?,.;:!")
>>> if (ponctuation & mdp == set()):
...     print("Le mot de passe ne contient pas de signe de␣
↪ponctuation")
```

Les ensembles sont très pratiques lorsque l'on veut supprimer des doublons.

Exercice

Récupérez un texte quelconque en chinois (par exemple https://www.gutenberg.org/files/24051/24051-0.txt). Combien d'idéogrammes chinois différents sont utilisés dans ce texte ?

2.2 Expressions en Python

Dans un langage de programmation, on distingue les expressions des commandes. Les expressions vont être évaluer par l'interpréteur pour renvoyer un objet.

Nous n'allons pas faire une description exhaustive [2] de toutes les possibilités.

— Les parenthèses peuvent avoir plusieurs sens :

```
>>> sin(1 + 2) # appel d'une fonction
0.1411200080598672
>>> (1 + 2J)*3 # parenthèse logique
(3+6j)
>>> (1, 2J)*3 # n-uplet
(1, 2j, 1, 2j, 1, 2j)
```

— Nous vous rappelons la syntaxe dite de *list comprehension* :

```
>>> [i**2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i//3 for i in range(10) if i%2==0]
[0, 0, 1, 2, 2]
```

2. https://docs.python.org/3/reference/expressions.html

De même il existe des *comprehension* pour les dictionnaires et les ensembles :

```
>>> s = {chr(97+i) for i in range(10) if i%2==0} # ensemble
>>> s == {'i', 'c', 'g', 'e', 'a'}
True
>>> d = {chr(97+i):i for i in range(10) if i%2==0} #␣
↪dictionnaire
>>> d == {'e': 4, 'c': 2, 'g': 6, 'i': 8, 'a': 0}
True
```

— Il est possible d'utiliser une condition dans une expression sous la forme intuitive *renvoie A si B sinon C*. Ce type d'expression permet d'éviter de passer par des variables intermédiaires.

```
sqrt(x) if x>=0 else sqrt(-x)*1J
print("Bonjour" if lang=='fr' else 'Hello')
```

2.3 Créer des fonctions

2.3.1 Arguments d'une fonction

Voici un exemple général de la définition d'une fonction :

```
>>> def f(a, b, c=1, d=2, *args, **kwd):
...     print(a, b, c, d)
...     print(args)
...     print(kwd)
```

Cette fonction possède deux arguments obligatoires, deux arguments optionnels. Les variables args et kwd vont contenir les arguments supplémentaires (sans et avec mots-clé - keyword).

Lorsque l'on appelle une fonction, les arguments peuvent être passés anonymement (par ordre) ou avec un nom (*keyword argument*, `nom=valeur`). Il faut mettre d'abord les arguments anonymes puis les autres. Il n'y a pas de lien entre le fait qu'un argument ait une valeur par défaut et le fait qu'il soit utilisé avec son nom. Lorsque les arguments sont passés avec leur nom, l'ordre est indifférent :

```
>>> f(1, 2, 4)
1 2 4 2
()
{}
>>> f(b=2 , a=2)
2 2 1 2
()
{}
```

Les arguments en trop sont envoyés dans args ou kwd :

```
>>> f(1, 2, 3, 4, 5, 6)
1 2 3 4
(5, 6)
{}
```

```
>>> f(1, 2, 3, e=4)
1 2 3 2
()
{'e': 4}
```

```
>>> f(1, 2, 3, 4, 5, 6, e=7)
1 2 3 4
(5, 6)
{'e': 7}
```

Enfin, il est possible de séparer un groupe d'arguments à partir d'un itérable (list, tuple, ..) (séparation anonyme) ou à partir d'un dictionnaire (séparation avec mots-clés) :

```
>>> liste = list(range(1,3))
>>> dct = {'d':3, 'e':4}
>>> f(0, *liste, **dct)
... # equivaut à f(0, 1, 2, d=3, e=4)
0 1 2 3
()
{'e': 4}
```

On remarquera que les variables `args` et `kwd` à l'intérieur de la fonction `f` sont différentes de celles que l'on a séparées (liste et dct dans cet exemple).

Il est de toute façon possible de séparer plusieurs listes ou dictionnaires :

```
>>> print(*range(3), *range(3))
0 1 2 0 1 2
```

Quelques remarques :

— Il ne faut par hésiter à utiliser des arguments par défaut (et c'est mieux que des variables globales)

— Lorsque l'on appelle une fonction, il ne faut pas hésiter à nommer les arguments, même si c'est inutile et que c'est plus long à taper. Comparez

```
>>> scope.configure_channel(1, 0.01, 0.03, 50)
>>> scope.configure_channel(channel_name=1, scale=0.01,␣
↪offset=0.03, impedance=50)
```

2.3.2 Variable locale/globale

Les variables que l'on crée dans une fonction sont locales, c'est à dire indépendante d'une variable extérieure à la fonction et qui porte le même nom.

```
>>> def f():
...     x = 2
>>> x = 3
>>> f()
>>> x
3
```

A l'intérieur d'une fonction, une variable est soit locale soit globale (elle ne peut pas changer en cours).

2.4 Les boucles

Il existe des boucle for et des boucle while.

Pour sortir d'une boucle on peut utiliser l'instruction `break`, pour passer à l'itération suivante l'instruction `continue`. Si une boucle se finit normalement (sans break), il est alors possible d'exécuter un bloc d'instruction dans un `else`. Voici un examole :

```
def affiche_si_premier(n):
    i=2
    while i**2<=n:
        if n%i==0:
            print("{} n'est pas premier".format(n))
            break
        i = i+1
    else:
        print('{} est premier'.format(n))
```

Remarquons que en Python il est possible de quitter une fonction à n'importe quel moment à l'aide de l'instruction return. Lorsque dans une boucle on connaît le résultat de la fonction, il est alors préférable de quitter celle ci immédiatement :

```
def est_premier(n):
    i=2
    while i**2<=n:
        if n%i==0:
            return False
        i = i+1
    return True
```

On peut parcourir une liste directement, sans passer par les indices :

```
panier = ['carottes', 'courgettes', 'tomates']
quantite = [1, 3, 2]

for legume in panier :
    print(legume)

# Si on souhaite l'indice:
for i, legumes in enumerate(panier):
    print(f'{i}: {legume}')

# Parcourir deux listes en même temps
for poids, legumes in zip(quantite, panier):
    print(f'{poids}kg de {legume}')
```

```
carottes
courgettes
tomates
```

(suite sur la page suivante)

(suite de la page précédente)

```
0: tomates
1: tomates
2: tomates
1kg de tomates
3kg de tomates
2kg de tomates
```

CHAPITRE 3

Tableaux numpy

numpy est la librairie qui permet de manipuler de larges tableaux de données. Elle offre plusieurs avantages par rapport aux listes : elle est beaucoup plus rapide et surtout permet de faire des calculs sans utiliser de boucles for. Il est indispensable de savoir manipuler correctement les tableaux numpy pour d'une part gagner du temps lors de l'exécution du programme, mais surtout gagner du temps lors de l'écriture du programme.

Contrairement aux listes, la taille et le type de donnée est fixé à la création d'un tableau numpy, ce qui fait que la mémoire est immédiatement allouée au tableau.

Voici quelques exemples :

```
import numpy as np

a = np.arange(10)
print(a**2)
print(np.sin(a))
```

```
[ 0  1  4  9 16 25 36 49 64 81]
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.
↪95892427
```

(suite sur la page suivante)

(suite de la page précédente)

```
-0.2794155    0.6569866    0.98935825   0.41211849]
```

Avertissement : Le module numpy contient toutes les fonctions mathématiques qui sont dans le module math. Il n'est pas possible d'utiliser une fonction du module math avec numpy.

```
import math
math.sin(np.arange(10))
```

```
---------------------------------------------------------------------
↪--------------
TypeError                                 Traceback (most␣
↪recent call last)
<ipython-input-2-8d2dc11828a2> in <module>
      1 import math
----> 2 math.sin(np.arange(10))

TypeError: only size-1 arrays can be converted to Python␣
↪scalars
```

3.1 Création d'un tableau

Il existe plusieurs fonctions pour créer un tableau.

— Création d'un tableau à partir d'une liste :

```
a = np.array([1, 2, 4])
print(a.dtype)

# data type is calculated automatically
a = np.array([1.2, 2, 4])
print(a.dtype) # all numbers are float

# data type can be forced
a = np.array([1, 2, 4], dtype=float)
print(a.dtype)
```

```
int64
float64
float64
```

— Création d'un tableau uniforme

```
N = 10
a = np.zeros(N)
b = np.ones(N)
```

— Créer un range

```
N = 10
a = np.arange(N)
print('Valeur calculée', a.sum())
print('Valeur théorique', (N*(N+1))/2)
```

```
Valeur calculée 45
Valeur théorique 55.0
```

— Répartition uniforme de points :

```
N = 300
x = np.linspace(0, 2*np.pi, num=N)
```

Avertissement : La fonction linspace inclus le premier et le dernier points. Dans le cas ci-dessus, la distance entre 2 points est donnée par $2\pi/(N-1)$

```
x = np.linspace(0, 1, num=10)
print('ATTENTION\n'*5)
print(x)
x = np.linspace(0, 1, num=10, endpoint=False)
print(x)
```

```
ATTENTION
ATTENTION
ATTENTION
ATTENTION
ATTENTION

```

(suite sur la page suivante)

(suite de la page précédente)

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.
↪55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

— Répartition sur une échelle logarithmique :

```
x = np.logspace(0, 2)
print(x.min())
print(x.max())
```

```
1.0
100.0
```

— Distributions aléatoires :

```
data = np.random.rand(N)
data = np.random.randint(10, size=N)
data = np.random.normal(loc=.3, scale=2.0, size=N)
```

Exemple : On utiliser fait la somme de trois dés, quelle est approximativement la probabilité de trouver 8 ?

```
M = 100000
dice1 = np.random.randint(1, 7, size=M)
dice2 = np.random.randint(1, 7, size=M)
dice3 = np.random.randint(1, 7, size=M)

result = dice1 + dice2 + dice3
print(np.mean(result==8))
```

```
0.09752
```

— Lire et écrire dans un fichier :

Il existe deux type d'enregistrement : l'enregistrement sous forme d'un fichier texte et celui sous forme binaire. Dans un fichier texte, le tableau doit être de dimension un ou deux. Il est écrit ligne par ligne sous forme de nombre décimaux. Dans un fichier binaire, c'est la mémoire de l'ordinateur qui est recopier dans le fichier. Les fichiers textes ont l'avantage d'être lisible par un humain et d'être compatible avec beaucoup de logiciel, cependant l'écriture et surtout la lecture du fichier prend beaucoup de temps. Les fichiers binaire seront beaucoup plus rapide.

Pour enregistrer au format texte :

```
filename = 'test.dat'
a = np.array([1, 2, 4])
np.savetxt(filename, a)
with open(filename) as f:
    print(f.read())
# Numbers are converted to float
```

```
1.00000000000000000e+00
2.00000000000000000e+00
4.00000000000000000e+00

```

Il est possible de lire des fichiers au format “csv” tels que ceux exportés par un tableur. Voici un exemple.

```
# Création du fichier
csv_content = """# Tension; courant
1; 2.3
2; 4.5
3; 7.0"""
filename = 'test.csv'
with open(filename, 'w') as f:
    f.write(csv_content)

data = np.loadtxt(filename, delimiter=';')
print(data[:,1])
```

```
[2.3 4.5 7. ]
```

Dans le cas présent, il est aussi possible de lire chaque colonne dans une variable directement :

```
# Utilisation de l'argument unpack
tension, courant = np.loadtxt(filename, delimiter=';',␣
↪unpack=True)
print(tension/courant)
```

```
[0.43478261 0.44444444 0.42857143]
```

Pour les fichiers binaires, on utilise la fonction `np.load` et `np.save`. Le tableau sera strictement identique après relecture.

```
filename='test.npy'
a = np.array([1, 2, 4])
np.save(filename, a)
new_a = np.load(filename)
print(new_a)
# a est toujours un tableau d'entier
```

```
[1 2 4]
```

3.2 Fonctions vectorisées

Les fonctions vectorisées sont des fonctions qui calculent automatiquement sur des tableaux, point par point. La plus part des opérateurs de python fonctionne sur des tableaux.

— Opérateurs mathématiques. On peut tout utiliser (comme les puissances ou modulos…).

```
def arctan(x, N_max=100):
    """ Calcule arctan par le développement limité"""
    n = np.arange(N_max)
    return np.sum((-1)**n * x**(2*n+1)/(2*n+1))

print(arctan(1), np.arctan(1))
```

```
0.782898225889638 0.7853981633974483
```

— Opérateurs logiques :

```
a = np.array([1, 2, 5, 3, 5])
b = np.array([0, 2, 3, 7, 5])
print(a==b)
print(a>b)
print( (a>b) | (a<b) )
print(a!=b)
print(~(a==b))
print((a>b) & ((a%2)==1)) # nb impaires
```

```
[False  True False False  True]
[ True False  True False False]
[ True False  True  True False]
```

(suite sur la page suivante)

(suite de la page précédente)

```
[ True False  True  True False]
[ True False  True  True False]
[ True False  True False False]
```

Avertissement : Ça n'a pas de sens d'utiliser les fonctions `and` ou `or`. Rappelons que ce sont des `if` implicites. Et dire, par exemple, « Si mon tableau est positif » n'a pas de sens. Seules les phrases du type : « si tous les élements sont positifs » ou « si au moins un élément est positif » ont un sens. C'est ce que le message d'erreur suivant dit :

```
if (a>b):
   print(a)
```

```
---------------------------------------------------------------
↪---------------
ValueError                                Traceback (most␣
↪recent call last)
<ipython-input-4-bdbbc477d411> in <module>
----> 1 if (a>b):
      2    print(a)

ValueError: The truth value of an array with more than one␣
↪element is ambiguous. Use a.any() or a.all()
```

— Fonctions mathématiques : Les fonctions de bases sont dans le module `numpy`. Les fonctions spéciales peuvent se retrouver dans le module `scipy.special`
Exemple : calcul de l'"éclairement d'une tache d'Airy [3]

```
from scipy.special import jv
def eclairement_airy(theta, d=1E-4, lamb=632E-9):
    """ Eclairement d'un disque """
    x = np.sin(theta)*d/lamb
    return (2*jv(1, np.pi*x)/(np.pi*x))**2

print(eclairement_airy(theta=2E-3))
```

```
0.7769307179570368
```

3. https://fr.wikipedia.org/wiki/Tache_d%27Airy

— Fonctions définie par l'utilisateur. Il n'y a rien de magique : une fonction sera vectorisée, si sont contenu l'est.

```
def position(x0, v0, t, g=9.81):
    """ Calcule la position d'un point après un temps de␣
↪chute """
    return x0 + v0*t + g*t**2/2

N = 10
sigma_x = 100E-6
sigma_v = 10E-3
t = 100E-3
x0 = np.random.normal(sigma_x, size=(N,3))
v0 = np.random.normal(sigma_v, size=(N,3))
xf = position(x0, v0, t)
```

Par contre, le code suivant va créer une erreur :

```
def valeur_absolue(x):
    if x>=0:
        return x
    else:
        return -x

print(valeur_absolue(-1))
print(valeur_absolue(np.array([-1, 1])))
```

```
1
```

```
---------------------------------------------------------------
↪-------------
ValueError                                Traceback (most␣
↪recent call last)
<ipython-input-7-9d3584a1f876> in <module>
      6
      7 print(valeur_absolue(-1))
----> 8 print(valeur_absolue(np.array([-1, 1])))

<ipython-input-7-9d3584a1f876> in valeur_absolue(x)
      1 def valeur_absolue(x):
----> 2     if x>=0:
      3         return x
```

(suite sur la page suivante)

(suite de la page précédente)

```
      4     else:
      5         return -x

ValueError: The truth value of an array with more than one␣
↪element is ambiguous. Use a.any() or a.all()
```

Si une fonction fonctionne avec des nombres scalaires, il est possible de la vectoriser automatiquement grâce à la fonction `np.vectorize`

```
valeur_absolue_vec = np.vectorize(valeur_absolue)
print(valeur_absolue_vec(np.array([-1, 1])))
```

```
[1 1]
```

Cette fonction va automatiser la création de la boucle `for` pour l'utilisateur. Elle sera lente comparée au fonction vectorisée native de numpy. On peut le voir en utilisant la commande "magic" de IPython `%timeit`

```
x = np.random.normal(size=100000)

%timeit -n 10 np.abs(x)
%timeit -n 10 valeur_absolue_vec(x)
```

```
36.5 µs ± 15.3 µs per loop (mean ± std. dev. of 7 runs, 10␣
↪loops each)
```

```
12.3 ms ± 623 µs per loop (mean ± std. dev. of 7 runs, 10␣
↪loops each)
```

Il est possible d'utiliser une décorateur pour créer directement une fonction vectorisée :

```
@np.vectorize
def my_abs(x):
    if x>=0:
        return x
    else:
        return -x
```

3.3 Indexer un tableau

Comment récupérer une partie d'un tableau (ou modifier une partie d'un tableau).

Récupérer les indices correspondants à un range :

```
a = np.random.rand(10)

print(a)
print(a[1])
print(a[-2])
print(a[1:4])
print(a[1:4:2])
```

```
[0.57282812 0.89578413 0.91332692 0.90384574 0.24068243 0.
↪11350129
 0.73715458 0.27632517 0.19398931 0.2609887 ]
0.8957841301214312
0.19398930877340892
[0.89578413 0.91332692 0.90384574]
[0.89578413 0.90384574]
```

On rappelle que l'on commence toujours à l'indice `0` et que un `range(n1, n2)` contient `n2-n1` éléments (le dernier est donc `n2-1`). Les indices négatifs sont pris modulo la longueur du tableau (donc `-2` est pareil que `len(a) - 2`.

Il est aussi possible de filtrer des données, par exemple pour récupérer des données selon un critère.

```
a = np.arange(4)

mask = [False, True, True, False]
print(a[mask])
```

```
[1 2]
```

Souvent le mask provient d'un tableau de booléen calculé automatiquement.

```
a = np.random.normal(size=10000)
mask = (a>=0)

print(a[mask].mean()) # Moyenne des tirages positifs
```

```
0.7916986492813732
```

Voici par exemple une façon de calculer la valeur absolue qui est vectorisée :

```
def valeur_absolue(x):
    result = np.empty_like(x)
    result[x<=0] = -x
    result[x>0] = x
    return result
```

Pour les tableau 2D, on peut récupérer toute la colonne (ou ligne) en mettant simplement un `:` :

```
b = np.random.rand(3, 2)
print(b)
print(b[1, 0])
print(b[:,0]) # première colone
print(b[:,1])
```

```
[[0.18615233 0.47605384]
 [0.28514889 0.07586353]
 [0.94335752 0.19762579]]
0.28514889075726646
[0.18615233 0.28514889 0.94335752]
[0.47605384 0.07586353 0.19762579]
```

3.4 Quelques opérations courantes

Il existe beaucoup de fonction numpy permettant de faire simplement des opérations sur les tableaux :

Voici quelques exemples. Parfois les fonctions numpy existent sous forme d'une méthode (`objet.method`).

```
a = np.random.normal(size=10000)

print(np.max(a))
print(a.max()) # Sous forme d'une méthode
```

(suite sur la page suivante)

(suite de la page précédente)

```
np.sum(a)
np.mean(a)
np.var(a)
np.std(a)
np.min(a)
np.max(a)
```

```
4.0351219469370365
4.0351219469370365
```

```
4.0351219469370365
```

Lorsque l'on a un tableau 2D (ou plus), il est possible d'éxecuter l'opération ligne par ligne (ou colonne par colonne).

```
n_eleves = 10
n_exams = 4

# Tableau 2D avec les notes
# Les notes sont aléatoires !
notes = np.random.rand(n_eleves, n_exams)*20

# On fait la moyenne sur les élèves (axis=0)
moyenne_exams = np.mean(notes, axis=0)
print('Moyenne de chaque examen')
print(moyenne_exams)

moyenne_eleves = np.mean(notes, axis=1)
print('Moyenne de chaque élève')
print(moyenne_eleves)
```

```
Moyenne de chaque examen
[ 9.81395276 11.1400401  10.49420259  9.04915144]
Moyenne de chaque élève
[ 7.79374555 11.47850198 11.67461786  7.40069001  8.65054475  9.
↪93937896
 13.09532836  9.81052539 10.24845065 11.15158371]
```

Opération de tri :

```
a = np.random.rand(10)
b = np.sort(a)
print(b)
```

```
[0.01156812 0.08512015 0.2807883  0.3859941  0.41174778 0.
↪45989627
 0.55805187 0.66253723 0.7195286  0.72064735]
```

Il est parfois utile de connaître l'indice du maximum ou minimum ou de connaître l'ordre du tri. Cela s'obtient avec les fonctions `argmax`, `argmin` ou `argsort`.

```
i_max = moyenne_eleves.argmax()

print(f'Le meilleur élève est le numéro {i_max}')
```

```
Le meilleur élève est le numéro 6
```

Autres fonctions utiles :

```
N = 1000
a = np.random.normal(size=N)
# Différence entre deux éléments
np.diff(a)

# Marche aléatoire
b = np.cumsum(a)
```

CHAPITRE 4

Tracer des graphiques

Nous utiliserons la librairie matplotlib. La fonctions sont dans le module `matplotlib.pyplot` qu'il est courant d'importer sous le nom de `plt`.

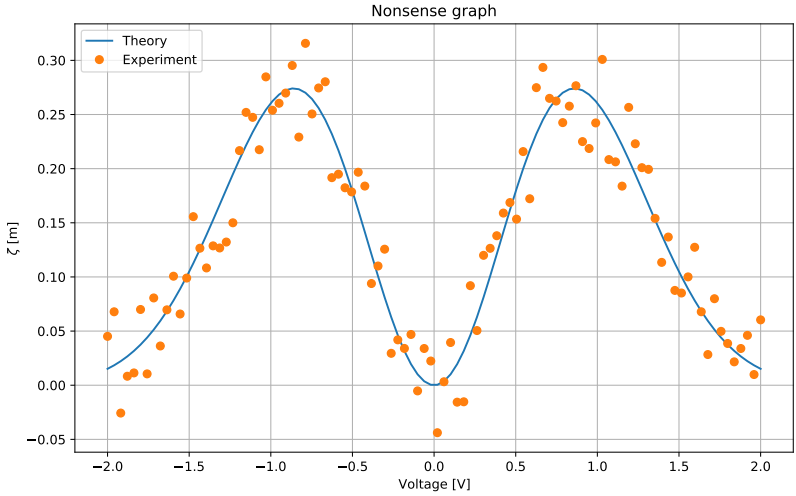
Le plus simple est d'étudier des exemples.

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

plt.figure(figsize=(10, 6))
X = np.linspace(-2,2, 100)
Y = np.sin(X)**2*np.exp(-X**2)
Y_noise = Y + .1*(np.random.rand(len(X))-0.5)

plt.plot(X,Y, label=u"Theory")
plt.plot(X,Y_noise,'o', label=u"Experiment")
plt.xlabel(r'Voltage [V]')
plt.ylabel(r'$\zeta$ [m]')
plt.title("Nonsense graph")
plt.legend(loc='upper left')
```

(suite sur la page suivante)

(suite de la page précédente)

```
plt.grid(True)

plt.savefig('mafigure.pdf')
```



Il existe deux syntaxes pour matplotlib, la syntaxe ci-dessus à base de fonctions (syntaxe historiquement utilisée par beaucoup de personnes) et une syntaxe utilisant sur des objets. L'idée est d'utiliser des méthodes des objets figure ainsi que des graphiques (appelé `axes`, il peut y avoir plusieurs `axes`, à ne pas confondre avec `axis` qui sont les abscisses et ordonnées).

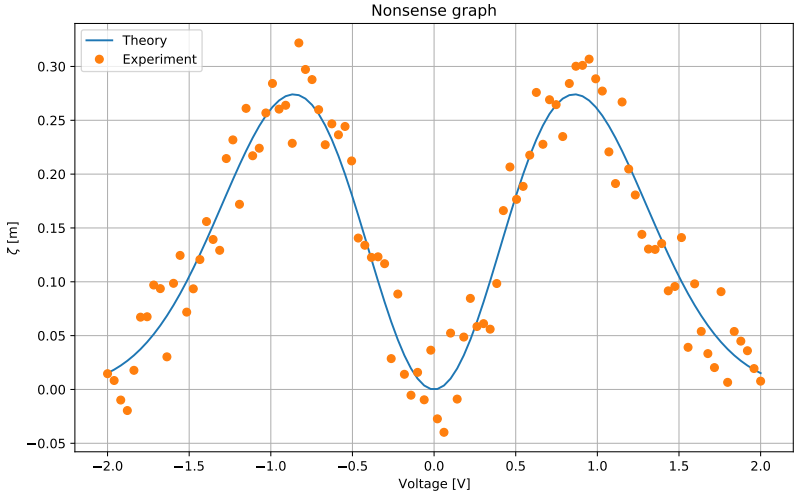
Voici le même exemple en orienté objet :

```
from matplotlib.pyplot import figure
import numpy as np

fig = figure(figsize=(10, 6))
ax = fig.subplots(1, 1) # Création d'un graphique
X = np.linspace(-2,2, 100)
Y = np.sin(X)**2*np.exp(-X**2)
Y_noise = Y + .1*(np.random.rand(len(X))-0.5)
```

(suite sur la page suivante)

(suite de la page précédente)

```
ax.plot(X,Y, label=u"Theory")
ax.plot(X,Y_noise,'o', label=u"Experiment")
ax.set_xlabel(r'Voltage [V]')
ax.set_ylabel(r'$\zeta$ [m]')
ax.set_title("Nonsense graph")
ax.legend(loc='upper left')
ax.grid(True)
```



Le syntaxe orientée objet est plus simple lorsque l'on veut mettre plusieurs graphs dans une même figure.

Voici un exemple du diagramme de Bode de la fonction de transfert suivante :

$$H(\omega) = \frac{\omega_0^2}{\omega_0^2 - \omega^2 + 2j\zeta\omega\omega_0}$$

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

# même pour une formule simple, il faut définir une fonction
```

(suite sur la page suivante)

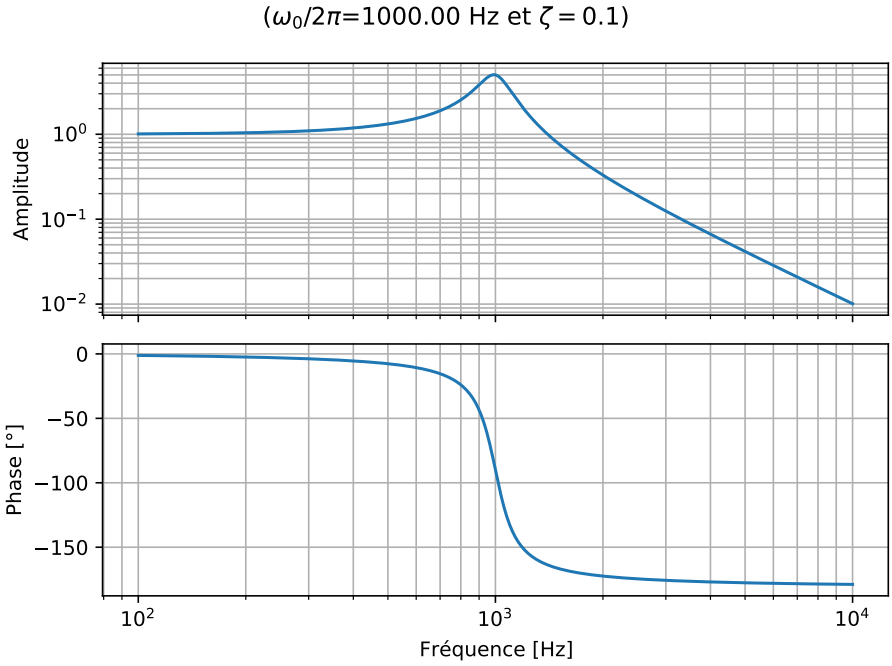
(suite de la page précédente)

```
def H(omega, omega_0, zeta):
    return omega_0**2/(omega_0**2-omega**2 +␣
↪2J*omega*zeta*omega_0)

omega_0 = 2*np.pi*1000
zeta = 0.1

Tomega = 2*np.pi*np.logspace(2,4, 1001)
amplitude = H(Tomega, omega_0, zeta)

fig = plt.figure()
ax1, ax2 = fig.subplots(2, 1, sharex=True)

fig.suptitle(f'($\omega_0/2\pi$={omega_0/(2*np.pi):.2f} Hz et $\
↪zeta={zeta}$)')
ax1.grid(True, which='both')
ax1.loglog(Tomega/(2*np.pi), np.abs(amplitude))
ax1.set_ylabel('Amplitude')

phase_in_deg = np.unwrap(np.angle(amplitude))/(2*np.pi)*360

ax2.semilogx(Tomega/(2*np.pi), phase_in_deg)
ax2.set_ylabel(u'Phase [°]')
ax2.set_xlabel(u'Fréquence [Hz]')
ax2.grid(True, which='both')

fig.tight_layout()
```

Graph avec des légendes : distribution de Fermi-Dirac

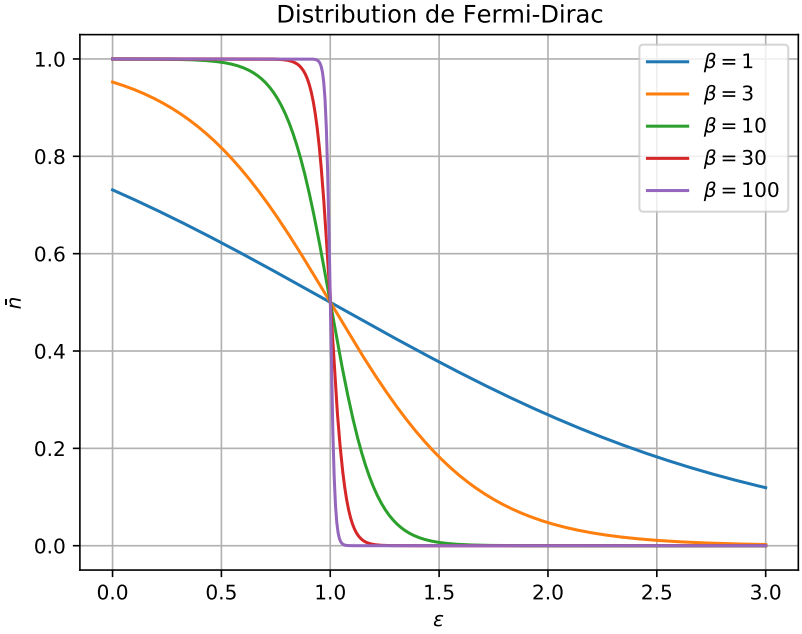
```
import numpy as np
import matplotlib.pyplot as plt

def fermi_dirac(epsilon, mu, beta):
    return 1/(np.exp(beta*(epsilon - mu))+1)

list_beta = [1, 3, 10, 30, 100]
mu = 1

x = np.linspace(0, 3, num=1000)
```

(suite sur la page suivante)


($\omega_0/2\pi$=1000.00 Hz et $\zeta = 0.1$)
Amplitude
10^0
10^{-1}
10^{-2}
Phase [°]
0
−50
−100
−150
10^2
10^3
10^4
Fréquence [Hz]

(suite de la page précédente)

```
fig = plt.figure()
ax = fig.subplots(1, 1)

for beta in list_beta:
    ax.plot(x, fermi_dirac(x, mu=mu, beta=beta),
            label=rf'$\beta={beta}$')

ax.set_xlabel(r'$\epsilon$')
ax.set_ylabel(r'$\bar n$')
ax.set_title('Distribution de Fermi-Dirac')
ax.grid()
ax.legend()
```



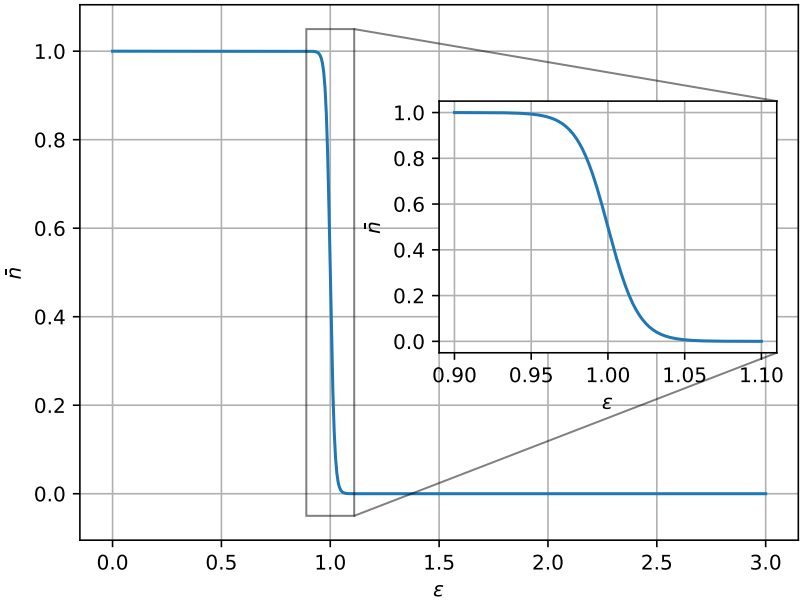
Remarques : pour les chaînes de caractère, il est possible d'utiliser des formules latex en utilisant des `$`. Il faut alors faire attention aux `\` : en effet il est possible qu'ils soient interprété comme des caractères spéciaux (par exemple `\n` est un retour à la ligne). Pour éviter ceci, on utilise des chaîne brutes (raw string), préfixées par un `r`.

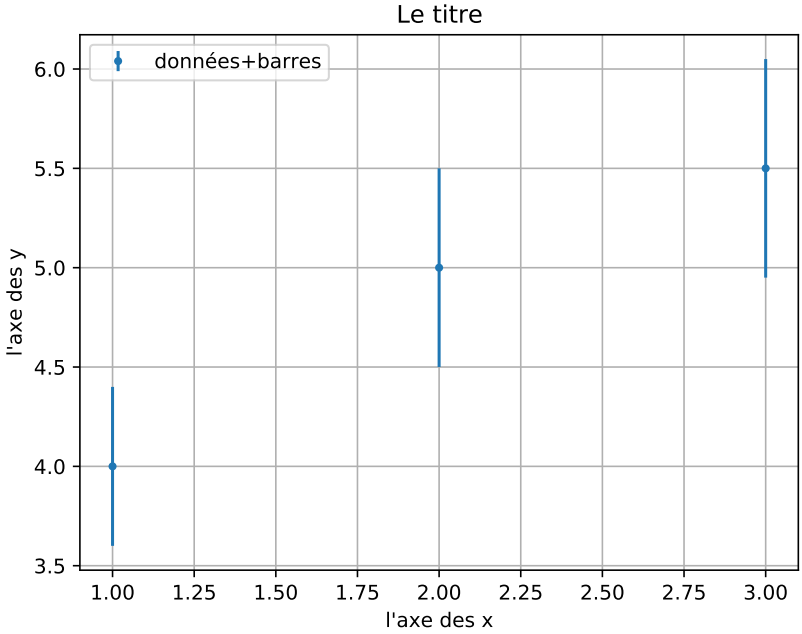
Figure avec un inset : distribution de Fermi-Dirac

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

def fermi_dirac(epsilon, mu, beta):
    return 1/(np.exp(beta*(epsilon - mu))+1)

x = np.linspace(0, 3, num=1000)
x_zoom = np.linspace(0.9, 1.1, num=1000)

fig = plt.figure()
ax = fig.subplots(1, 1)

ax.plot(x, fermi_dirac(x, mu=1, beta=100))

axins = ax.inset_axes([0.5, 0.35, 0.47, 0.47])
axins.plot(x_zoom, fermi_dirac(x_zoom, mu=1, beta=100))
ax.indicate_inset_zoom(axins, edgecolor="black")

for a in [ax, axins]:
    a.set_xlabel(r'$\epsilon$')
    a.set_ylabel(r'$\bar n$')
    a.grid()
```

Barres d'erreur :

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

x = np.array([1, 2, 3])
y = np.array([4, 5, 5.5])
erreurs_y = 0.1 * y

plt.errorbar(x, y, erreurs_y, fmt='.', label="données+barres")

plt.xlabel("l'axe des x")
plt.ylabel("l'axe des y")
plt.legend(loc=2)
plt.grid()
plt.title("Le titre")
```


1.0
0.8
0.6
0.4
0.2
0.0
$\bar{n}$
0.0
0.5
1.0
1.5
2.0
2.5
3.0
ε
0.90
0.95
1.00
1.05
1.10


Le titre
données+barres
6.0
5.5
5.0
4.5
4.0
3.5
l'axe des y
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
l'axe des x

CHAPITRE 5

Calculer avec numpy et scipy

Les librairies numpy et scipy contiennent tous les algorithmes usuels des calculs numériques. Nous allons en voir quelques uns.

5.1 Algèbre linéaire

numpy.linalg et scipy.linalg (plus de fonction dans scipy)

— Matrice : np.matrix (produit matriciel)

— Inverse de matrice

— Diagonalisation/valeurs propres/vecteurs propres

Avertissement : Le produit de deux tableaux np.array n'est pas le produit matriciel. Pour faire le produit matriciel, il faut utiliser l'opérateur "*" et des objets de type `np.matrix` ou utiliser l'opérateur `@` (introduit dans Python 3.5).

Exemple : Calculer les valeurs propres de :

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

```
import numpy as np
from scipy.linalg import eigh # Matrice hermicienne

H = np.matrix([[1, 1, 0], [1, 0, 1], [0, 1, -1]])

w, v = eigh(H)
print(w)
print(v)
print(H@v[:,1])
```

```
[-1.73205081  0.          1.73205081]
[[-0.21132487  0.57735027  0.78867513]
 [ 0.57735027 -0.57735027  0.57735027]
 [-0.78867513 -0.57735027  0.21132487]]
[[ 1.11022302e-16 -1.44328993e-15  1.55431223e-15]]
```

Tracer les valeurs propres en fonction de δ pour $\Omega = 1$ de l'Hamiltonien suivant :

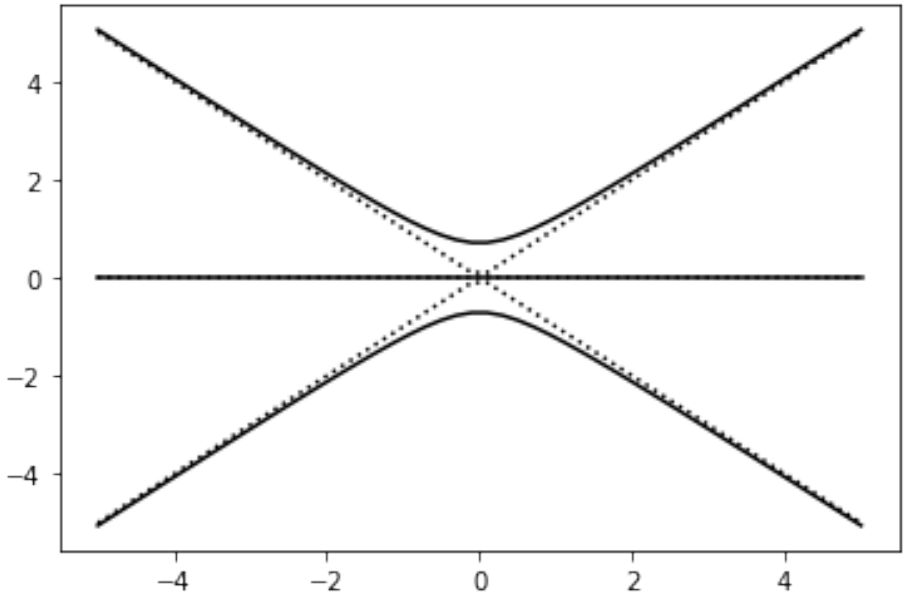
$$\begin{bmatrix} \delta & \frac{\Omega}{2} & 0 \\ \frac{\Omega}{2} & 0 & \frac{\Omega}{2} \\ 0 & \frac{\Omega}{2} & -\delta \end{bmatrix}$$

```
import matplotlib.pyplot as plt

def trois_niveaux(delta, omega):
    H = np.matrix([[delta, omega/2, 0], [omega/2, 0, omega/2],␣
↪[0, omega/2, -delta]])
    return eigh(H)[0]

all_delta = np.linspace(-5, 5)
sans_couplage = np.array([trois_niveaux(delta, omega=0) for␣
↪delta in all_delta])
avec_couplage = np.array([trois_niveaux(delta, omega=1) for␣
↪delta in all_delta])
```

(suite sur la page suivante)

(suite de la page précédente)

```
plt.plot(all_delta, sans_couplage, 'k:')
plt.plot(all_delta, avec_couplage, 'k-');
```



5.2 Optimisation

5.2.1 Minimum d'une fonction

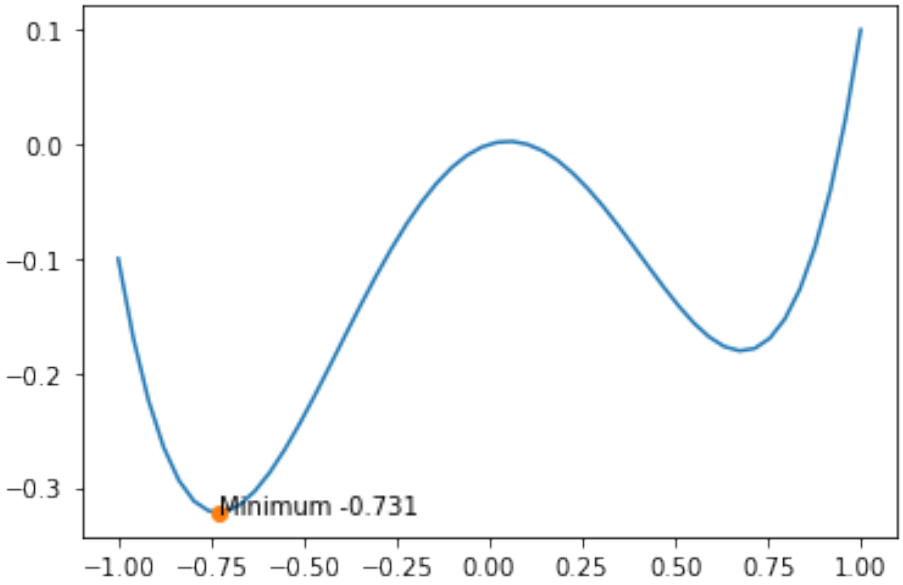
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fmin

def f(x):
    return 0.1*x + x**4 - x**2

x = np.linspace(-1, 1)
plt.plot(x, f(x))
```

(suite sur la page suivante)

(suite de la page précédente)

```
x_min = fmin(f, -0.75)
plt.plot(x_min, f(x_min), 'o')
plt.text(x_min, f(x_min), f'Minimum {x_min[0]:.3f}');
```

```
Optimization terminated successfully.
         Current function value: -0.321919
         Iterations: 11
         Function evaluations: 22
```



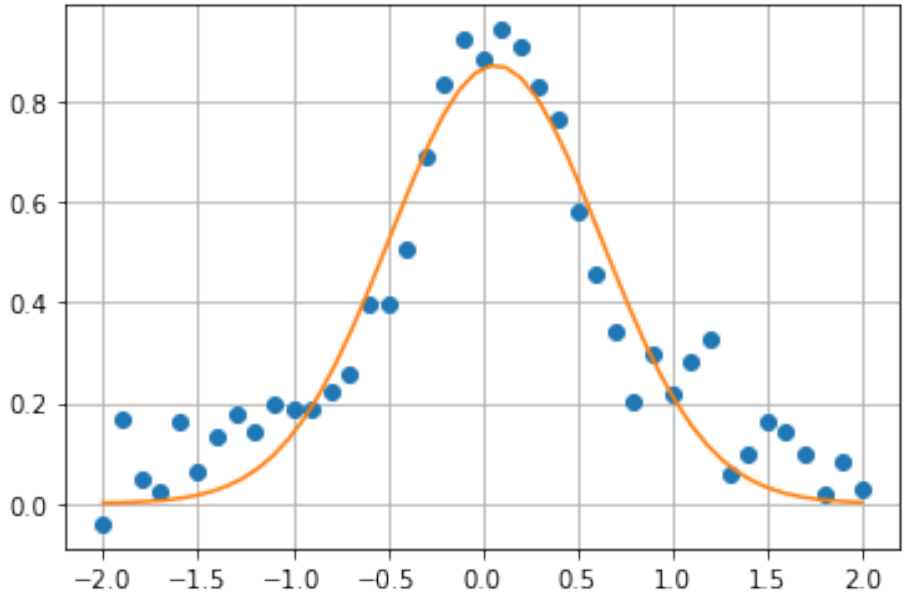
5.2.2 Ajustement des moindre carrés

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def fit_function(x, amplitude, largeur, centre):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return amplitude*np.exp(-(x-centre)**2/(2*largeur**2))

X = np.linspace(-2, 2, 41)
data = 1/(1+(X-0.07)**2/.5**2) +  .05*np.random.
↪normal(size=len(X))

plt.plot(X, data, 'o')

x_plot = np.linspace(-2, 2)
# Uncomment to see the curve with inital parameters
#plot(x_plot, fit_function(x_plot, amplitude=1,
#                          largeur=.5,
#                          centre=0))
init_param = [1, .5, 0]

popt, pcov = curve_fit(fit_function, X, data, init_param)
plt.plot(x_plot, fit_function(x_plot, *popt))
plt.grid()
```



5.3 Intégration

On peut utiliser la fonction `quad`

Exemple : calculer :

$$\int_0^1 \frac{1}{1+x^2} \mathrm{d}x$$

```
from scipy.integrate import quad

def f(x):
    return 1/(1+x**2)

res, _ = quad(f, 0, 1)
print(res)
```

```
0.7853981633974484
```

Avertissement : Si on connaît la fonction, il ne faut pas en faire un tableau. La fonction quad calcule automatiquement les points de l'intégrale afin d'atteindre une erreur donnée. De plus, la fonction quad peut intégrer sur des bornes infinies (`np.inf`)

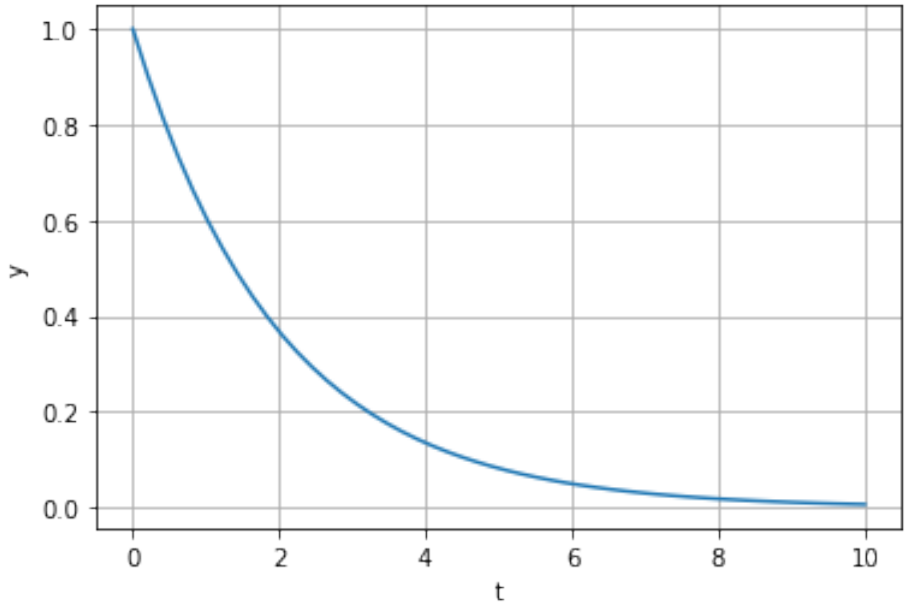
5.4 Equations différentielles

On utilise la fonction `solve_ivp` (initial value problem). Elle remplace les fonctions `ode` ou `odeint`

5.4.1 Equations du premier ordre

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{y}{\tau}$$

```
from scipy.integrate import solve_ivp
import numpy as np

tau = 2

def f(t, y):
    return -y/tau

y0 = 1
res = solve_ivp(f, t_span=[0, 10], y0=[1], t_eval=np.linspace(0,
↪ 10))

plt.plot(res.t, res.y[0,:])
plt.grid()
plt.xlabel('t')
plt.ylabel('y');
```



5.4.2 Équations différentielles d'ordre élevé

L'astuce consiste à augmenter la dimension de y en rajoutant des fonctions intermédiaires qui sont les dérivées de la fonction initiale.

Par exemple l'équation

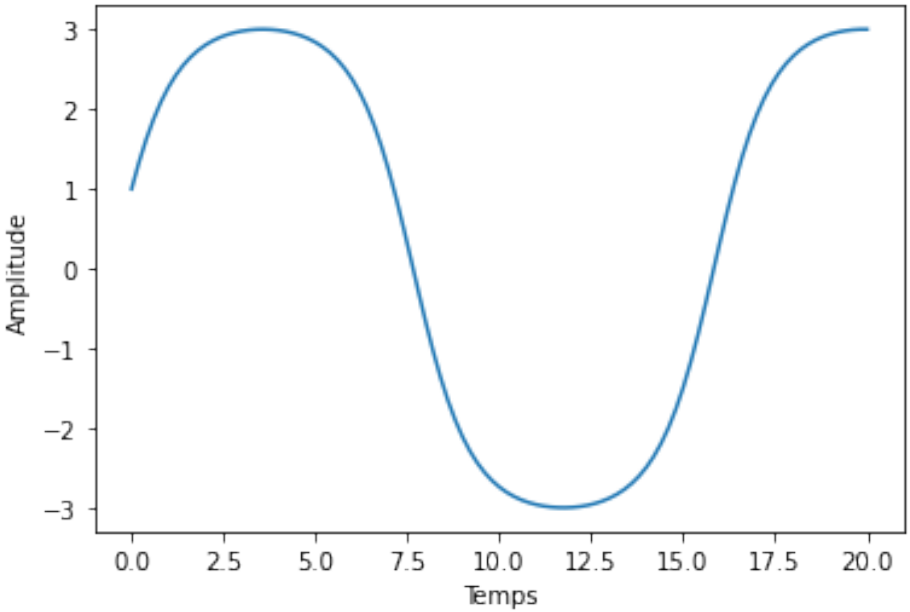
$$\frac{d^2y}{dt^2} = \frac{f(y)}{m}$$

devient

$$\frac{d}{dt}\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} y' \\ f(y)/m \end{pmatrix} = F(y, y')$$

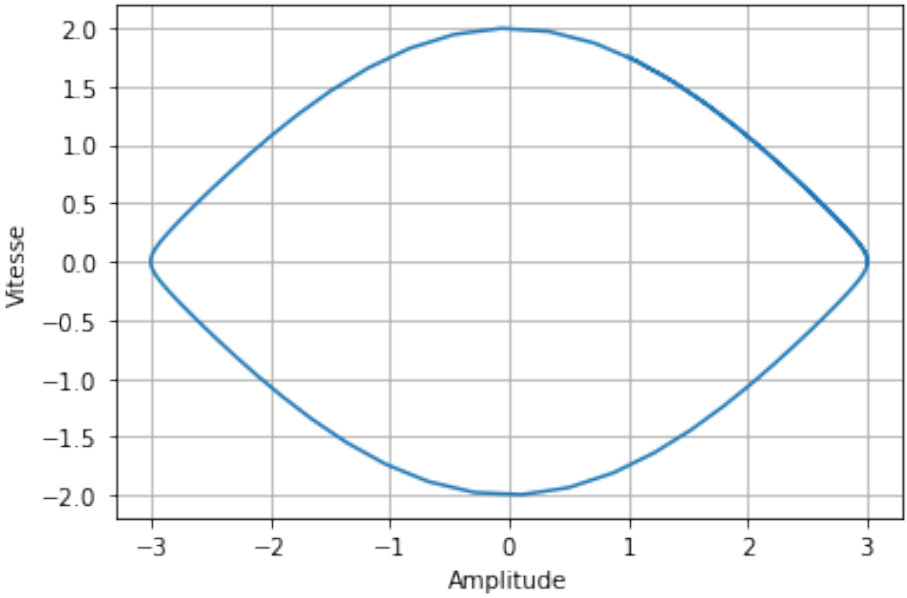
Voici comment résoudre l'équation d'un pendule $\frac{d^2\theta}{dt^2} = -sin(\theta)$:

```
def f(t, Y):
    y, v = Y
    a = -np.sin(y)
    return [v, a]

y0 = [1, 1.75]

res = solve_ivp(f, t_span=[0, 20], y0=y0, t_eval=np.linspace(0,␣
↪20, 101), rtol=1E-7, atol=1E-7)

plt.plot(res.t, res.y[0])
plt.xlabel('Temps')
plt.ylabel('Amplitude');
```



On peut aussi regarder la solution dans l'espace des phases :

```
plt.plot(res.y[0,:], res.y[1,:])
plt.grid()
plt.xlabel('Amplitude')
plt.ylabel('Vitesse');
```



5.5 Transformée de Fourrier

Le module numpy implément la transformée de Fourier numérique par l'algorithme de FFT. Pour cela, il faut utiliser les fonctions `numpy.fft.fft` et `numpy.fft.ifft` pour avoir la transformée de Fourier inverse. Lorsque l'on effectue un transformée de Fourier, l'axe des fréquences peut être obtenu à l'aide de la fonction `numpy.fft.fftfreq`.

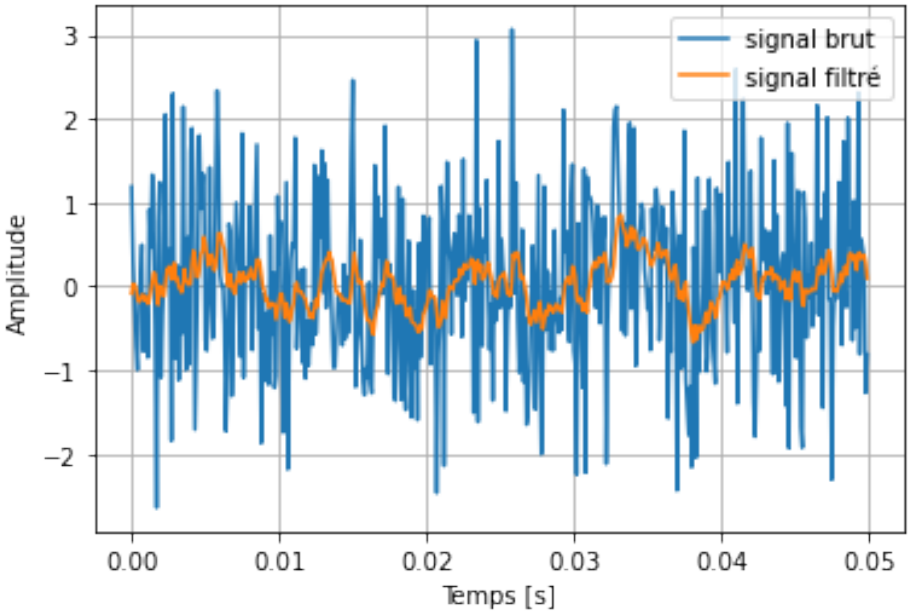
Voici un exemple de filtre passe bas utilisant la FFT

```
import numpy as np

dt = 1E-4
N = 50000
fc = 200
t = np.arange(N)*dt

# Simulation d'un signal
freq = 100
```

(suite sur la page suivante)

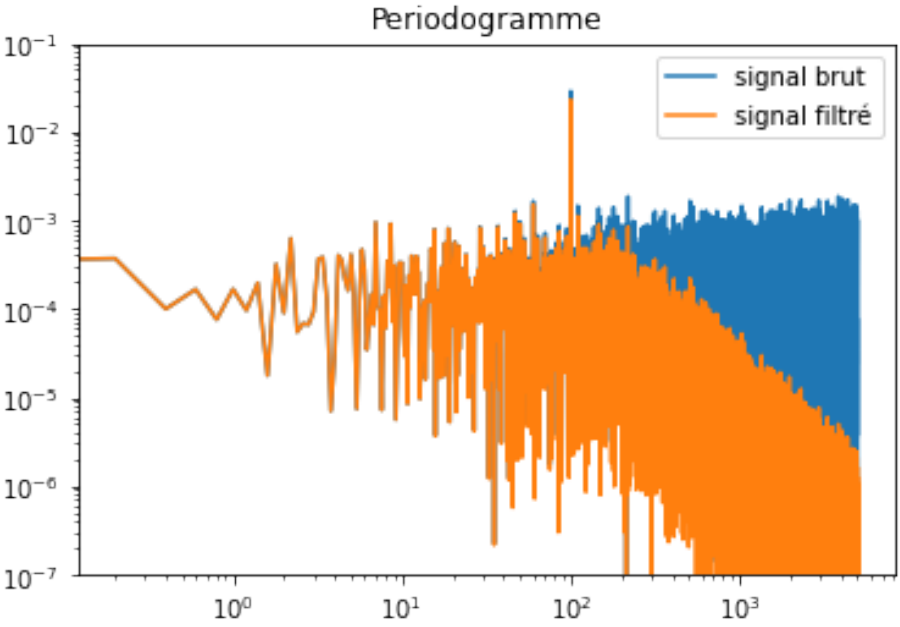
(suite de la page précédente)

```
data = np.sin(2*np.pi*freq*t)*0.1 + np.random.normal(size=N)

# Réalisation d'un filtre dans l'espace de Fourier
data_tilde = np.fft.fft(data)
freq = np.fft.fftfreq(N, d=dt)
H = 1/(1+ 1J*freq/fc) # Fonction de transfert
data_tilde_filtre = data_tilde*H
data_filtre = np.real(np.fft.ifft(data_tilde_filtre))

from matplotlib.pyplot import figure

fig = figure()
ax = fig.subplots(1, 1)

ax.plot(t[:500], data[:500], label='signal brut')
ax.plot(t[:500], data_filtre[:500], label='signal filtré')
ax.set_xlabel('Temps [s]')
ax.set_ylabel('Amplitude')
ax.legend()
ax.grid()
```



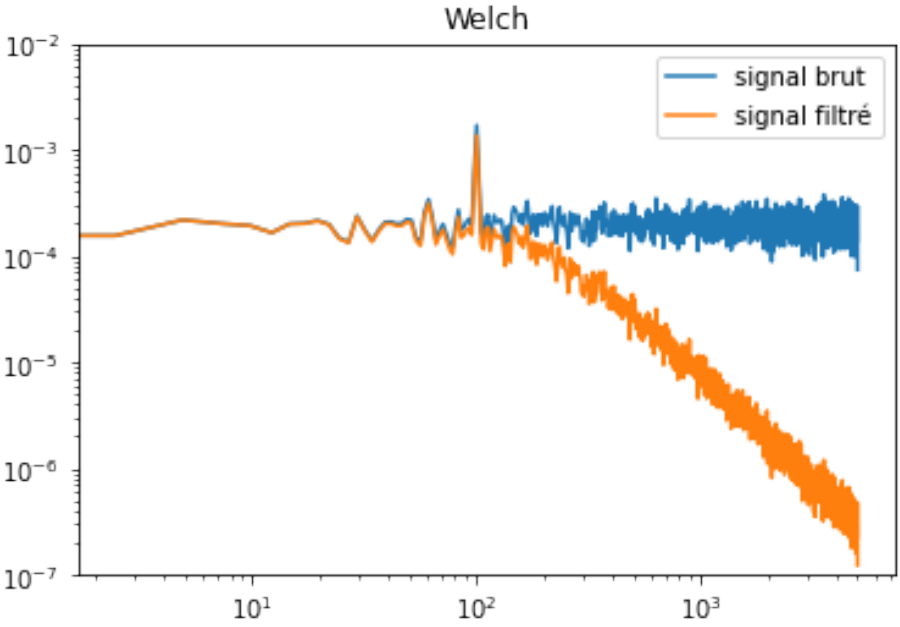
Souvent on a besoin d'évaluer la densité spectrale de puissance d'un signal. Il est possible pour cela d'utiliser la méthode du periodogramme avec la fonction `scipy.signal.periodogram`.

```
# On reprend le signal ci dessus
import scipy.signal
f, psd = scipy.signal.periodogram(data, fs=1/dt)
f, psd_filtre = scipy.signal.periodogram(data_filtre, fs=1/dt)

fig = figure()
ax = fig.subplots(1, 1)

ax.loglog(f, psd, label='signal brut')
ax.loglog(f, psd_filtre, label='signal filtré')
ax.set_title('Periodogramme')
ax.legend()
ax.set_ylim(1E-7, 1E-1);
```



Il est aussi possible d'utiliser la méthode `welch` qui moyenne des periodogrammes pris sur des durées plus petites.

```
# On reprend le signal ci dessus
import scipy.signal
f, psd = scipy.signal.welch(data, fs=1/dt, nperseg=2**12)
f, psd_filtre = scipy.signal.welch(data_filtre, fs=1/dt,␣
↪nperseg=2**12)

fig = figure()
ax = fig.subplots(1, 1)

ax.loglog(f, psd, label='signal brut')
ax.loglog(f, psd_filtre, label='signal filtré')
ax.set_title('Welch')
ax.legend()
ax.set_ylim(1E-7, 1E-2);
```


Welch
signal brut
signal filtré
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{1}
10^{2}
10^{3}