# Python pour la physique-chimie

Emmanuel Loyer

Été 2022

# Sommaire

**Introduction** **5**

**1 Utilisation de base de python** **7**
I Utilisation de Pyzo . . . 7
II Manipulation des types les plus utiles . . . 8
1 Variables et affectation . . . 8
2 Entiers (relatifs) et flottants . . . 8
3 Chaînes de caractères . . . 9
4 Listes . . . 10
5 Booléens . . . 10
III Structures algorithmiques . . . 11
1 Boucle for (boucle inconditionnelle) . . . 11
2 Boucle while (boucle conditionnelle) . . . 12
3 Structure conditionnelle . . . 13
4 Fonction . . . 14

**2 Commandes python et algorithmes pour les méthodes numériques** **15**
I Préliminaire : autres type utiles . . . 15
1 Tableau numpy . . . 15
2 Nombres complexes . . . 16
II Représentations graphiques . . . 17
1 Pour démarrer . . . 17
2 Améliorer son graphique . . . 19
3 Vecteurs et champs de vecteurs . . . 20
4 Histogrammes . . . 23
5 Pour aller plus loin . . . 24
III Résolution approchée d'équations différentielles . . . 24
1 Résolution d'une équation différentielle du premier ordre par la méthode d'Euler . . . 24
2 Résolution d'une équation différentielle du premier ordre avec la commande odeint . . . 27
3 Systèmes différentiels d'ordre 1 . . . 28
4 Vectorisation d'un équation différentielle d'ordre supérieur ou égal à 2 . . . 29
IV Équations algébriques . . . 31
1 Dichotomie . . . 31
2 Utilisation de bisect . . . 32
V Régression linéaire et autres ajustements . . . 32
1 Régression linéaire . . . 33
2 Autres ajustements . . . 33
VI Intégration, dérivation . . . 34
1 Calcul approché d'une intégrale : méthode des rectangles . . . 35
2 Calcul approché du nombre dérivé d'une fonction en un point . . . 36
VII Variables aléatoires et statistiques . . . 38
1 Variables aléatoires à densité . . . 38
2 Estimateurs de la moyenne et de l'écart-type . . . 39
3 Simulation d'une loi normale en Python . . . 40
4 Simulation d'une loi uniforme en Python . . . 41

**A Bonus** **45**
I Lire et écrire dans un fichier . . . . . 45
1 Écrire dans un fichier . . . . . 45
2 Lire dans un fichier . . . . . 48
3 Exemple : récupération des données exportées depuis LatisPro . . . . . 50
4 Autre exemple : exploitation de données exportées depuis Avimeca . . . . . 53
II Animations en python . . . . . 55
1 Faire une animation en python . . . . . 56
2 Exporter des images en python et fabriquer une animation avec ffmpeg . . . . . 56

# Introduction

Ce document a pour vocation d'accompagner le travail sur les capacités numériques en deuxième année de CPGE. Il est organisé pour le moment en deux chapitres :

- le premier chapitre rassemble les notions de python vues dans l'enseignement secondaire (et revues en cours d'informatique en début de première année) qui sont utiles pour la physique-chimie en CPGE,
- le deuxième chapitre présente les commandes python et les algorithmes cités dans l'annexe 3 des programmes de physique-chimie de CPGE ; notons que la quai-totalité de ces notions relèvent du programme de première année[1].

Une annexe contient les bonus : des méthodes et commandes python qui peuvent s'avérer bien utiles aussi bien en physique-chimie que pour les TIPE, mais qui ne font pas partie des notions exigibles des programmes de CPGE.

[1] Le programme de deuxième année introduit très peu de notions nouvelles, les capacités numériques de deuxième année réinvestissent en majorité les notions vues en première année dans des contextes différents.

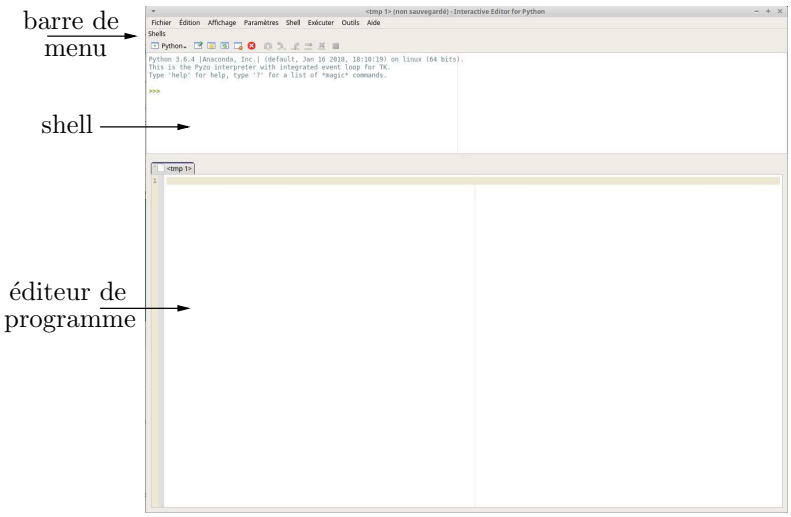
# Chapitre 1

# Utilisation de base de python

Ce chapitre contient des rappels de notions vues dans l'enseignement secondaire et première année (dans le cadre de l'enseignement d'Informatique de Tronc Commun). Il s'agit de rappeler les propriétés des principaux types de variables ainsi que des principales structures algorithmiques.

## I Utilisation de Pyzo

La programmation en python s'effectue en général à l'aide d'un Environnement de Développement Intégré (IDE en anglais) ; sur les ordinateurs du lycée, c'est pyzo qui est installé. Rappelons à quoi il ressemble :




On distingue deux fenêtres particulièrement utiles, en plus de la barre de menu :

- le shell : on peut y exécuter directement les commandes une par une, c'est également ici que s'affichent les messages d'erreur,
- l'éditeur de programmes : permet de saisir facilement un programme comportant plusieurs lignes de code.

On y trouve également un explorateur de fichiers qui n'est pas forcément utile. Quand un programme est écrit dans l'éditeur, plusieurs possibilités s'offrent à nous pour l'exécuter :

- aller dans le menu « Exécuter » et choisir « Exécuter la cellule »,
- utiliser le raccourci clavier "ctrl"+e (qui signifie appuyer simultanément sur la touche "ctrl" et la touche "e").
- appuyer sur F5.

On peut exécuter seulement une partie du programme ; pour cela, sélectionner la partie du code désirée (en utilisant le clic gauche de la souris) puis

- aller dans le menu « Exécuter » et choisir « Exécuter la sélection »,
- faire un clic droit, et choisir « Exécuter la sélection »,
- utiliser le raccourci clavier "alt"+"entrée".

Si on efface accidentellement une ou plusieurs lignes dans l'éditeur "ctrl"+"z" annule la dernière action.

Quelques fonctionnalités pratiques :

- pour enregistrer son programme, aller dans le menu « Fichier » puis sélectionner « Enregistrer » ou « Enregistrer sous... », ou faire un clic droit sur l'onglet qui se situe juste au-dessus de l'éditeur de programme, ou le raccourci clavier "ctrl"+"s",
- pour exporter le code en pdf, aller dans le menu « Fichier » puis "Exporter au format PDF",
- pour tuer le shell (en cas de plantage), cliquer sur la croix blanche sur fond rouge dans la barre de menu ; il faut alors relancer un shell en allant dans le menu « Shell » puis « Création du shell »,
- dans l'éditeur de programme, on peut commenter une ligne en la commençant par un croisillon # ; on peut également sélectionner plusieurs lignes et faire un clic droit puis « Commenter » (« Décommenter » réalise l'opération inverse),
- commencer une ligne par deux croisillons.

Il est fortement recommandé de commenter son code : cela en facilite la lisibilité pour les autres, mais aussi pour soi-même (quand on reprend un code écrit précédemment).

# II Manipulation des types les plus utiles

On rappelle ici les différents types de variables qui seront utilisés par la suite.

## 1 Variables et affectation

Rappelons que l'affectation s'effectue par exemple avec la commande du type x=2 ; le symbole = n'a pas ici le même sens qu'en mathématiques[1], il réalise une affectation : la valeur 2 est rangée dans la variable étiquetée x. On peut afficher la valeur stockée dans la variable x par la commande print(x)[2]. Python possède quelques raccourcis utiles comme l'affectation multiple : x,y=3,4 qui est équivalent à x=3 et y=4.

## 2 Entiers (relatifs) et flottants

Après l'affectation x=2, la variable x est de type **entier**, comme on peut s'en convaincre en tapant print(type(x)) dans le shell. Notons que nous n'avons pas déclaré que la variable x serait de type entier avant l'affection : en python, le typage est **implicite**. Voici les opérations possibles sur les entiers :

| opération | addition | soustraction | multiplication | division | exponentiation | division entière | reste |
|---|---|---|---|---|---|---|---|
| symbole | + | - | * | / | ** | // | % |

Les règles de priorité usuelles s'appliquent.

Les nombres à virgule sont représentés à l'aide du type **flottant** : par exemple x,y=1.2,-2e-3. Les opérations sur les flottants sont les mêmes que pour les entiers. Les fonctions mathématiques plus évoluées (comme les fonctions trigonométriques par exemple) ne sont pas présentes nativement dans python ; nous verrons plus loin comment résoudre ce problème.

Bien qu'un entier et un flottant correspondent à deux types différents, les opérations entre un entier et un flottant sont possibles : on peut par exemple aditionner un entier et un flottant (le résultat est un flottant).

Terminons par une remarque *a priori* anodine en exécutant le code suivant

```
1 x=2
2 print(type(x))
3 x=4.0
4 print(type(x))
```

qui donne

```
<class 'int'>
<class 'float'>
```

L'affectation de la ligne 3 « écrase » celle de la ligne 1 ; cela occasionne le changement de type de la variable de x. On dit que le typage est **dynamique**.

[1] en particulier il n'est pas symétrique.

[2] attention à ne pas confondre avec la commande print("x") qui affiche la chaîne de caractère réduite à la lettre x.

## 3 Chaînes de caractères

Rappelons que la variable définie par x="bonjour" est du type **chaîne de caractère**. La commande len(x) renvoie la longueur de la chaîne de caractère x, c'est-à-dire le nombre de caractères qu'elle comporte ; les caractères de x sont numérotés de 0 à len(x)-1. On accède au caractère en position $i$ par la commande x[i] ; on accède aux caractères aux positions comprises entre $i$ (inclus) et $j$ (exclu) par la commande x[i :j].

À titre d'exemple si l'on exécute le programme suivant

```
x="bonjour"
print(x[0], x[1], x[6])
print(x[-1],x[-2],x[-7])
print(x[1:4])
```

on obtient

```
b o r
r u b
onj
```

Notons que la commande x[-i] est un raccourci pour x[len(x)-i].

Le symbole + réalise la concaténation, et * permet de répéter une concaténation (dans le même sens que la multiplication est une répétition d'additions), comme on peut s'en convaincre sur les programme suivant

```
x,y="bonjour","bonsoir"
print(x+y, 2*x, y*3)
```

dont l'exécution conduit à

```
bonjourbonsoir bonjourbonjour bonsoirbonsoirbonsoir
```

Pouvez-vous prévoir le résultat des commandes suivantes ?

```
x,y="12","34"
print(x+y, 2*x, y*3)
```

Notons que plusieurs délimiteurs peuvent être utilisés pour une chaîne de caractères : les affectations x='bonjour', x="bonjour" et x=''' '''bonjour''' ''' sont équivalentes. Cela peut s'avérer utile si l'on veut fabriquer une chaîne de caractères contenant des guillemets. Pour cela, on peut aussi utiliser les caractères spéciaux ; par exemple, le code :

```
print('Elle a dit \'il pleut\'')
print("il a répondu \"il ne fait pas beau\"")
```

conduit à l'affichage

```
Elle a dit 'il pleut'
il a répondu "il ne fait pas beau"
```

Autre caractère spécial bien utile : \n qui correspond à un saut de ligne ; le code

```
essai="bonjour\nbonsoir"
print(essai)
print(len(essai))
```

conduit à l'affichage

```
bonjour
bonsoir
15
```

Le décompte des caractères de la chaîne essai montre que le caractère spécial \n compte pour un caractère.

Il est possible de convertir un entier en chaîne de caractères avec la commande str, la conversion d'une chaîne de caractère en entier s'effectuant avec la commande **int** ; le code suivant

```
x,y=2,"3"
print(type(x),type(y))
a,b=str(x),int(y)
print(a,type(a))
print(b,type(b))
```

conduit à

```
<class 'int'> <class 'str'>
2 <class 'str'>
3 <class 'int'>
```

ce qui montre que la variable x est de type entier et la variable y est de type chaîne de caractères ; ensuite, a est de type chaîne de caractères alors que b est entier. On fait de même avec des flottants, la conversion d'une chaîne de caractère en flottant s'effectuant avec la commande **float**. Ces conversions seront utiles par exemple lors de la lecture[3] ou l'écriture[4] de données dans un fichier.

## 4 Listes

Une liste permet de ranger des éléments : on peut construire des listes d'entiers, des listes de flottants, des listes de chaînes de caractères ou des listes regroupant des objets de type différents :

```
L1=[1,2,3,4,5,6]
L2=[3.00e8,8.85e-12,1.26e-6,6.67e-11,6.63e-34,8.314]
L3=['Alice','Bob','Charles','Denise','Etienne','Fannie','Gaston']
L4=[1,3.0,'bonjour']
```

On accède aux éléments d'une liste de la même façon qu'aux éléments d'une chaîne de caractères ; de même, la concaténation fonctionne avec les listes comme avec les chaînes de caractères :

```
print(L1[3],L2[-1])
print(len(L2))
print(L3[1:5])
print(L1+L2,L1*2)
```

Contrairement à une chaîne de caractères, une liste est « mutable » : on peut la modifier. Par exemple, on peut modifier un élément d'une liste, ou ajouter un élément dans une liste grâce à la méthode append ; on peut retirer le dernier élément grâce à la méthode pop.

Notons enfin qu'une liste peut contenir des listes : on obtient alors une liste de listes.

## 5 Booléens

On utilise rarement les booléens explicitement pour faire de la chimie ou de la physique, mais on les utilise implicitement lorsque l'on réalise un test. Un booléen est une variable qui ne peut prendre que deux valeurs, « vrai » ou « faux », True ou False en python. On peut effectuer les opérations logiques habituelles sur les booléens : and, or, not...

```
X,Y=True,False
print(type(X))
print(X and Y, X or Y, not X)
```

[3] Lecture des données issues d'un capteur par exemple.

[4] Écriture de données dans un fichier en vue d'une utilisation ultérieure, éventuellement par un autre logiciel (tableur, LaTeX).

ce qui conduit à

```
<class 'bool'>
False True False
```

Le résultat d'un test est un booléen ; réalisons un test d'égalité et la comparaison de deux entiers :

```
a,b=1,2
print(a==b,a<b,a>b,a<=b,a>=b)
print(a!=b)
```

```
False True False True False
True
```

Le dernier test réalisé montre que le contenu des variables a et b sont différents. Notons une source d'erreur très classique : le test d'égalité est noté ==, le symbole = est réservé à l'affectation.

# III Structures algorithmiques

## 1 Boucle for (boucle inconditionnelle)

Rappelons que la boucle for sert à répéter une instruction :

```
for k in range(1,5):
    print('bonjour')
#
for k in range(1,5):
    print(k)
#
for k in range(1,5):
    print(2, '*', k, '=', 2*k)
```

On obtient

```
Bonjour
bonjour
bonjour
bonjour
1
2
3
4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

La syntaxe k in range(m,n) signifie que k varie de m inclus à n **exclu** : $k \in [\![ m ; n-1 ]\!]$. Notons le rôle de l'indentation : tout ce qui est indenté est dans la boucle, ce qui n'est pas indenté est exécuté après la boucle. Ainsi les deux boucles de l'exemple sont exécutées successivement. Avec pyzo, les deux points à la fin de la ligne de la commande for produisent automatiquement une indentation.

La commande range peut prendre un troisième argument, qui correspond au pas :

```
for k in range(1,11,3):
    print(k)
```

```
1
4
7
10
```

Notons enfin que la syntaxe range(n) est un raccourci pour range(0,n).

Pour afficher les éléments d'une liste, deux syntaxes peuvent être utilisées :

```
1 L=[1,2,"a",3.8,[1,2]]
2 #
3 for i in range(len(L)):
4     print(L[i])
5 #
6 for x in L:
7     print(x)
```

Ces deux synthaxes produisent le même affichage.

Plusieurs boucles for peuvent être imbriquées

```
1 for i in range(1,11):
2     for j in range(1,11):
3         print(i,'*',j,'=',i*j)
4     print('\n')
```

Que fait ce code ? À quel moment est exéctuée la commande de la ligne 4 ?

## 2 Boucle while (boucle conditionnelle)

La boucle while permet de répéter une instruction (ou un bloc d'instruction) tant qu'une condition est satisfaite : il n'est pas nécessaire de connaître *a priori* le nombre de répétitions (contrairement au cas de la boucle for). Illustrons cette notion avec le problème suivant : on souhaite afficher tous les entiers dont le carré est inférieur strictement à $n$ (où $n$ est un entier donné).

```
1 n=10
2 i=1
3 while i**2<n:
4     print(i)
5     i=i+1
```

On obtient :

```
1
2
3
```

La condition d'arrêt de la boucle while est donnée par un test, c'est-à-dire un booléen. On pourrait écrire

```
1 n=10
2 i=1
3 test=i**2<n
4 while test:
5     print(i)
6     i=i+1
7     test=i**2<n
```

mais cette syntaxe est plus lourde.

La boucle while est un peu plus délicate à manipuler que la boucle for : il faut s'assurer que la boucle s'arrête. Par exemple, la boucle suivante ne s'arrête pas car le test est toujours vrai (on a oublié d'incrémenter i) :

```
n=10
i=1
while i**2<n:
    print(i)
```

## 3 Structure conditionnelle

La structure conditionnelle la plus simple s'écrit

```
Si condition
  instruction
```

Donnons un premier exemple

```
n= #donner une valeur pour faire fonctionner le programme
if n==0:
    print("n est nul")
```

Notons immédiatement le rôle de l'indentation.

Compliquons un peu avec la structure

```
Si condition
  instruction
Sinon
  instruction
```

et un exemple

```
n= #donner une valeur pour faire fonctionner le programme
if n==0:
    print("n est nul")
else:
    print("n est non nul")
```

Encore un peu plus compliqué

```
Si condition
  instruction
Sinon si
  instruction
Sinon
  instruction
```

et un exemple

```
n= #donner une valeur pour faire fonctionner le programme
if n==0:
    print("n est nul")
elif n>0:
    print("n est strictement positif")
else:
    print("n est strictement négatif")
```

La structure conditionnelle la plus générale comporte un nombre quelconque de Sinon si.

Notons que chaque instruction peut correspondre à un bloc d'instructions ; la fin du bloc d'instructions est repéré par la fin de l'indentation. L'indentation joue donc un rôle essentiel, comme c'est le cas pour les boucles.

## 4 Fonction

Nous limitons cette introduction aux fonctions « à retour de valeur »[5] ; ces structures python reproduisent le comportement d'une fonction au sens des mathématiques. Ainsi, la fonction « mathématique »

$$f : \begin{cases} \mathbb{R} \longrightarrow \mathbb{R} \\ x \longmapsto 2x \end{cases}$$

est programmée en python de la façon suivante :

```
def f(x):
    return 2*x
```

Contrairement à l'usage en mathématiques, il n'est pas précisé l'ensemble de départ et l'ensemble d'arrivée ; on ne précise même pas le type de l'argument (ici x) et de la valeur renvoyée[6] (ici 2 x). Testons

```
print(f(2))
print(f(2.5))
print(f("bonjour"))
```

On obtient

```
4
5.0
bonjourbonjour
```

La fonction renvoie donc un résultat pour un argument de type entier, flottant ou chaîne de caractères. Comment interpréter le dernier résultat ?

On peut composer des fonctions :

```
def g(x):
    return x**2
#
def h(x):
    return f(g(x))
```

Vérifier le fonctionnement de $g$ et $h$. Exercice : écrire une fonction signe qui, pour tout flottant, renvoie 0 si le flottant est nul, 1 s'il est positif, $-1$ s'il est négatif.

Notons qu'un fonction peut prendre plusieurs arguments. Un argument peut-être un entier, un flottant, une chaîne de caractères, une liste etc... Il en est de même pour le résultat renvoyé par la fonction. Exemple : que fait la fonction suivante, dont l'argument est une liste d'entiers ou de flottants ?

```
def toto(L):
    m=L[0]
    for i in range(1,len(L)):
        if L[i]>m:
            m=L[i]
    return m
```

Dans l'exemple précédent, m et i sont des variables locales : elles sont créées lors de l'appel de la fonction mais n'existent plus une fois que la fonction a renvoyé son résultat. Essayer print(m) ou print(i).

Inversement, dans le code suivant, C est une variable globale.

```
C=5
def toto2(x):
    return C*x
```

[5] et laissons de côté les fonctions dites « à effet de bord ».
[6] Il est recommandé de commenter le code afin de donner ces informations.

# Chapitre 2

# Commandes python et algorithmes pour les méthodes numériques

Dans ce chapitre, on introduit les commandes python ainsi que les méthodes numériques citées par les programmes de physique-chimie. Le lien avec le contexte physique ou chimique n'est pas abordé (ou très succintement) à ce stade.

## I Préliminaire : autres type utiles

En plus des types entiers, flottants, booléens, liste et chaînes de caractères vus dans le chapitre précédent (et rencontrés en ITC), nous introduisons ici deux nouveaux types particulièrement utiles en physique-chimie.

### 1 Tableau numpy

La bibliothèque numpy comporte de nombreuses fonctions très utiles pour les mathématiques et la physique-chimie. On l'importe de la façon suivante

```
import numpy as np
```

où np sera l'alias par lequel on appellera la bibliothèque. Cette bibliothèque comporte un nouveau type d'objet, que nous appellerons « tableau numpy », qui peuvent servir à représenter des vecteurs ou des matrices. Ainsi, un tableau unidimensionnel (on parle souvent de « vecteur ») est construit de la façon suivante à partir d'une liste

```
L1,L2=[1,2,3],[4,5,6]
X1,X2=np.array(L1),np.array(L2)
```

où la synthaxe np.array signifie que l'on utilise la commande array issue de la bibliothèque numpy (identifiée par son alias np). Pour distinguer le comportement des listes et des tableaux numpy, exécutons les instructions suivantes :

```
print(L1+L2)
print(X1+X2)
```

On obtient

```
[1, 2, 3, 4, 5, 6]
[5 7 9]
```

On constate que le symbole + effectue la concaténation pour des listes, alors qu'il réalise l'addition au sens des vecteurs (c'est-à-dire terme à terme) pour les tableaux numpy ; c'est pourquoi on appelle (abusivement) ces derniers « vecteurs ». Essayons de même

```
print(2*L1,2*X1)
print(2.5*X1)
```

Il vient

```
[1, 2, 3, 1, 2, 3] [2 4 6]
[2.5 5.  7.5]
```

« Multiplier » une liste par 2 revient à la concaténer avec elle-même, alors que « multiplier » un tableau numpy par deux revient à multiplier chacun de ces éléments par 2 (ce qui correspond à la multiplication d'un vecteur par un scalaire) ; on peut de même « multiplier » un tableau numpy par un flottant. En revanche, si l'on tente de « multiplier » une liste par un flottant

```
print(2.5*L1)
```

on obtient un message d'erreur.

Notons également qu'on accède aux éléments d'un « vecteur » comme aux éléments d'une liste comme on peut s'en convaincre en tapant print(X1[0], X1[-1],X1[0 :2]) ; par ailleurs, un tableau numpy est mutable (on peut modifier la valeur d'un ou plusieurs de ses éléments).

La bibliothèque numpy comporte également les fonctions mathématiques usuelles : exp, sin, cos, log, sqrt... Ces fonctions peuvent agir sur des tableaux numpy

```
X3=np.array([0,1,2])
print(np.exp(X3))
```

qui donne

```
[1.          2.71828183 7.3890561 ]
```

L'exponentielle d'un tableau numpy unidimensionnel est le tableau des exponentielles des différents éléments. Nous verrons l'intérêt de cette synthaxe pour tracer des courbes dans la partie suivante.

Un tableau numpy peut également servir à représenter une matrice :

```
L1=[[1,2,3],[4,5,6]]
L2=[[1,3,5],[2,4,6]]
X=np.array(L1)
Y=np.array(L2)
print(X)
print(X+Y)
```

```
[[1 2 3]
 [4 5 6]]
[[ 2  5  8]
 [ 6  9 12]]
```

L'addition correspond bien à celle des matrices. De nombreuses opérations sur les matrices sont disponibles dans la bibliothèque numpy (produit, transposition, valeurs propres...) ; ces opérations relèvent de l'algèbre linéaire, elles peuvent être utiles en physique ou en chimie (niveau post-CPGE). Nous rencontrons par ailleurs ces tableaux bidimensionnels lors de la résolution numérique d'équations différentielles. On peut extraire une « ligne » ou une « colonne » d'un tableau numpy bidimensionnel :

```
print(X[0,:],X[:,1])
```

```
[1 2 3] [2 5]
```

Dans les deux cas, on obtient un tableau numpy unidimensionnel.

## 2 Nombres complexes

Python comporte un type dédié aux nombres complexes ; ce type n'apparaît pas dans les programmes de CPGE, mais on signale son existence car il peut être très utile. On définit les nombres complexes de la façon suivante :

```
X=1.1+2.3j
Y=-3.2+4.5j
a,b=2,3
Z=a+b*1j
print(type(X))
print(Z)
```

```
<class 'complex'>
(2+3j)
```

Les opérations usuelles sont implémentées :

```
print(X+Y,X-Y) # addition, soustraction
print(X*Y,X/Y) # multiplication, division
print(X.real,X.imag) # partie réelle, partie imaginaire
print(abs(X)) # module
print(np.exp(X)) # exponentielle
```

```
(-2.1+6.8j) (4.300000000000001-2.2j)
(-13.870000000000001-2.4099999999999993j) (0.22400787143325673-0.4037389307969826j)
1.1 2.3
2.5495097567963922
(-2.0016037856990585+2.240222262301069j)
```

On peut notamment utiliser ce type complexe pour manipuler des fonctions de transfert en électrocinétique ou des fonctions d'onde en physique quantique.

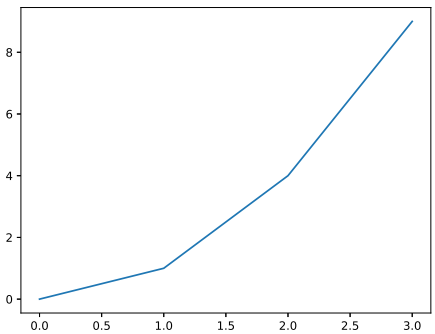
# II Représentations graphiques

La plupart des simulations numériques se terminent par une visualisation graphique des résultats. Le programme de première année mentionne l'utilisation des « fonctions de base de la bibliothèque matplotlib » (sans préciser lesquelles).

## 1 Pour démarrer

Commençons par un exemple minimaliste :

```
import matplotlib.pyplot as plt
#
X=[0,1,2,3]
Y=[0,1,4,9]
#
plt.plot(X,Y)
plt.show()
```



Dans le programme, la première ligne correspond à l'importation de la bibliothèque matplotlib.pyplot, les lignes 3 et 4 contiennent les données qui vont servir à tracer le graphique : une liste pour les valeurs des abscisses, et une liste pour les valeurs des ordonnées (ces deux listes doivent avoir le même nombre d'éléments). Les deux dernières lignes commandent le calcul du tracé (ligne 6) puis son affichage (ligne 7).
On observe le comportement suivant (par défaut) :

- les points sont reliés par des segments,
- la courbe est tracée en bleu.

Nous verrons plus loin comment modifier ce comportement. On remarque par ailleurs des icônes en bas de la fenêtre graphique ; en passant sa souris sur une icône, sa fonction s'affiche. Signalons en particulier la loupe (qui permet de zoomer dans le graphique) et la disquette[1] (qui permet d'exporter le graphique).

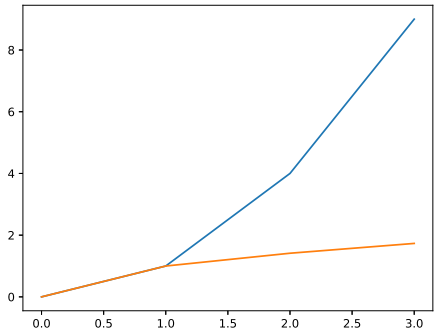
Fermer la fenêtre graphique et exécuter le programme suivant, qui fait la même chose que le programme précédent, mais utilise des tableaux numpy plutôt que des listes :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 #
4 X=np.array([0,1,2,3])
5 Y=X**2
6 plt.plot(X,Y)
7 plt.show()
```

Si l'on a oublié de fermer la fenêtre graphique avant de tracer le nouveau graphe, celui-ci se superpose au précédent dans la même fenêtre. Afin d'éviter se désagrément, on commence chaque nouveau graphe par la commande plt.figure().

On peut tracer plusieurs courbes sur le même graphe :

```
1 X=np.array([0,1,2,3])
2 Y1=X**2
3 Y2=np.sqrt(X)
4 #
5 plt.figure()
6 plt.plot(X,Y1)
7 plt.plot(X,Y2)
8 plt.show()
```




On constate que la deuxième courbe est tracée par dessus par première ; elle est tracée en orange.

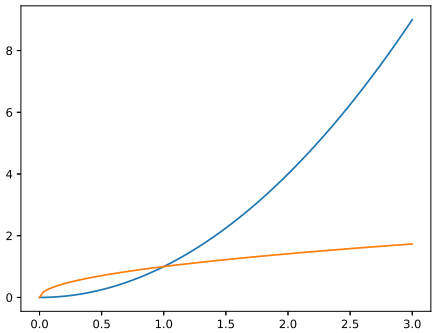
Terminons par un exemple illustrant le rôle de la commande linspace de la bibliothèque numpy

```
1 X=np.linspace(0,3,100)
2 Y1=X**2
3 Y2=np.sqrt(X)
```

[1] Savez-vous ce qu'est une disquette ?

```
4  #
5  plt.figure()
6  plt.plot(X,Y1)
7  plt.plot(X,Y2)
8  plt.show()
```

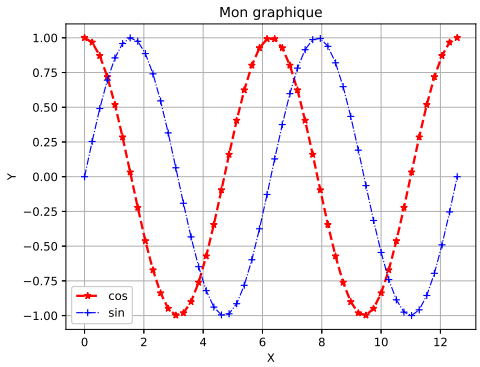
La commande de la ligne 1 fabrique un tableau numpy unidimensionnel (un « vecteur ») comportant 100 flottants régulièrement espacés compris entre 0 et 3.




## 2 Améliorer son graphique

Afin d'illustrer les possibilités offertes par matplotlib.pyplot, exploitons l'exemple suivant :

```
1  X=np.linspace(0,4*np.pi,50)
2  Y1=np.cos(X)
3  Y2=np.sin(X)
4  #
5  plt.figure()
6  plt.plot(X,Y1,color='red',marker='*',linestyle='--',linewidth=2,label='cos')
7  plt.plot(X,Y2,color='blue',marker='+',linestyle='-.',linewidth=1,label='sin')
8  plt.grid()
9  plt.title('Mon graphique')
10 plt.xlabel('X')
11 plt.ylabel('Y')
12 plt.legend()
13 plt.show()
```




Décortiquons cet exemple, en commançant par les lignes 6 et 7. Nous voyons que plusieurs paramètres optionnels sont passés à la commande plot :

– color='red' permet de spécifier la couleur ; on peut écrire plus simplement color ='r',
– marker='*' permet de préciser le marqueur utilisé pour repérer les points (ici une étoile),
– linestyle='- -' permet de préciser le type de ligne (ici des tirets),
– linewidth=2 permet d'imposer l'épaisseur de la ligne,
– label='cos' permet d'associer une « étiquette » à une courbe (son utilité sera vue plus loin).

Voici quelques valeurs possibles pour la couleur, le style et le marqueur :

| | |
|---|---|
| b | blue |
| c | cyan |
| g | vert |
| k | noir |
| m | magenta |
| r | rouge |
| y | jaune |

| | |
|---|---|
| - | trait continu |
| – | tirets |
| -. | points-tirets |
| : | pointillés |
| None | pas de trait |

| | |
|---|---|
| . | point |
| , | pixel |
| + | + |
| x | × |
| * | étoile |
| None | pas de marqueur |

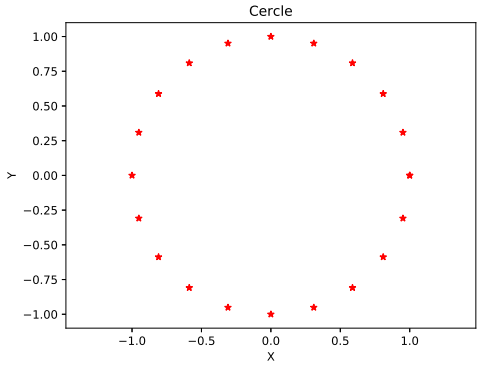
Décrivons maintenant le comportement des autres commandes :

– grid() affiche un quadrillage,
– title, xlabel et ylabel permettent d'afficher respectivement le titre, la légende en abscisse et la légende en ordonnée,
– legend() utilise les « étiquettes » associées à chaque courbe pour identifier les courbes.

Notons que l'on peut imposer des bornes en abscisse et en ordonnée en spécifiant plt.xlim(0,15) et plt.ylim(-1,1) ou de façon équivalente plt.axis([0,15,-1,1]). On impose par ailleurs un repère orthonormé avec plt.axis('equal').

Terminons avec un exemple illustrant un tracé de courbe paramétrée :

```
theta=np.linspace(0,2*np.pi,21)
X=np.cos(theta)
Y=np.sin(theta)
#
plt.figure()
plt.plot(X,Y,color='red',marker='*',linestyle='None')
plt.title('Cercle')
plt.xlabel('X')
plt.ylabel('Y')
plt.axis('equal')
plt.show()
```



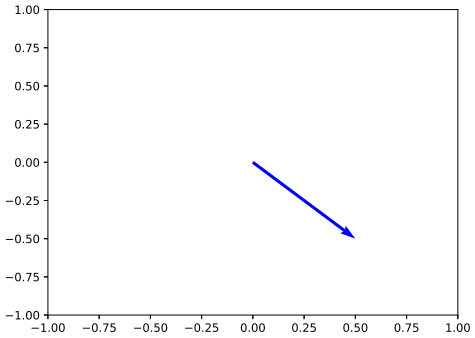

Que se passe-t-il si on n'utilise pas la commande plt.axis('equal') ?

## 3 Vecteurs et champs de vecteurs

### a. Représentation d'un vecteur

On peut représenter un vecteur à l'aide de la commande quiver. Donnons un premier exemple, qui permet de tracer un vecteur de coordonnées $(v_x, v_y)$ au point de coordonnées $(x, y)$ :

```
x,y=0,0
vx,vy=0.5,-0.5
plt.quiver(x,y,vx,vy,scale_units='xy',angles='xy',scale=1,color='blue')
plt.axis([-1,1,-1,1])
plt.show()
```




Les deux premiers arguments de la commande quiver sont les coordonnées du point où l'on trace le vecteur, viennent ensuite les composantes du vecteurs ; le paramètre scale permet de modifier la taille du vecteur (essayer scale=2 par exemple).

### b. Représentation d'un champ de vecteur

On peut représenter un champ de vecteur en parcourant les points d'un domaine donné de l'espace, et en traçant en chacun de ces points le champ correspondant.

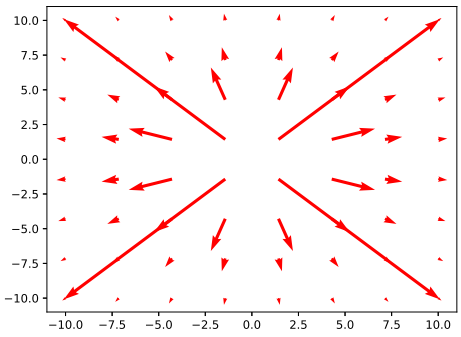
Représentons le champ de vecteur suivant[2]

$$\overrightarrow{\mathrm{E}} = \frac{\overrightarrow{e_r}}{r^2} = \frac{\overrightarrow{r}}{r^3} = \frac{x\,\overrightarrow{e_x} + y\,\overrightarrow{e_y} + z\,\overrightarrow{e_z}}{(x^2+y^2+z^2)^{3/2}}$$

On le représente de la façon suivante dans le plan $z = 0$ :

```
X=np.linspace(-10,10,8) # vecteur contenant les valeurs des abscisses
Y=np.linspace(-10,10,8) # vecteur contenant les valeurs des ordonnées
#
def champEx(x,y): # fonction calculant la composante x du champ
    return x/(x**2+y**2)**1.5
def champEy(x,y): # fonction calculant la composante y du champ
    return y/(x**2+y**2)**1.5
#
plt.figure()
for x in X: # on parcourt toutes les abscisses
    for y in Y: # on parcourt toutes ordonnées
        Ex=champEx(x,y)
        Ey=champEy(x,y)
        plt.quiver(x,y,Ex,Ey,scale_units='xy',angles='xy',scale=0.02,color='red')
plt.show()
```

[2] On reconnaît, à une constante multiplicative près, le champ électrostatique créé par une particule ponctuelle chargée placée à l'origine du repère.




Les lignes 1 et 2 fabriquent les tableaux unidimensionnels contenant les valeurs des abscisses et des ordonnées des points où l'on va représenter le champ. Les boucles imbriquées (lignes 5 et 6) permettent de parcourir tous les points où l'on va représenter le champ ; en chacun de ces points, on calcule les composantes du champ (lignes 12 et 13) et on affiche le vecteur (ligne 14) grâce aux fonctions définies aux lignes 4 à 7 (il suffit de modifier ces fonctions si on veut changer le champ de vecteur que l'on représente).

Donnons une méthode alternative pour réaliser le même tracé ; elle utilise la commance meshgrid dont le fonctionnement est décrit plus bas.

```
1  X=np.linspace(-10,10,8) # vecteur contenant les valeurs des abscisses
2  Y=np.linspace(-10,10,8) # vecteur contenant les valeurs des ordonnées
3  XX,YY=np.meshgrid(X,Y) # obtentions de deux tableaux bidimensionnels
4  #
5  def champEx(x,y): # fonction calculant la composante x du champ
6      return x/(x**2+y**2)**1.5
7  def champEy(x,y): # fonction calculant la composante y du champ
8      return y/(x**2+y**2)**1.5
9  #
10 EX=champEx(XX,YY) # EX et EY sont des tableaux bidimensionnels
11 EY=champEy(XX,YY)
12 #
13 plt.figure()
14 plt.quiver(XX,YY,EX,EY,scale_units='xy',angles='xy',scale=0.02,color='red')
15 plt.show()
```

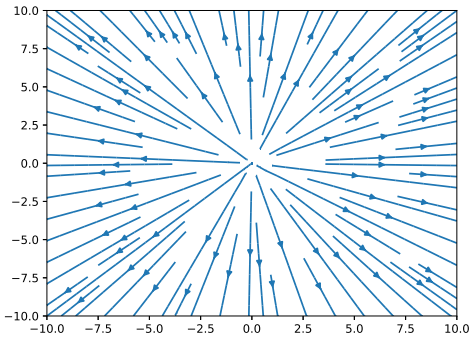
Le résultat est identique à celui produit par le code précédent. Cependant, son fonctionnement mérite d'être détaillé. On constate que les deux premières lignes sont identiques au code précédent : elles fabriquent les tableaux (unidimensionnels) contenant respectivement les valeurs des abscisses et les valeurs des ordonnées des points où l'on va calculer le champ. La ligne 3 fabrique deux tableaux bidimensionnels (XX et YY) qui contiennent respectivement les abscisses et les ordonnées de l'ensemble des points où l'on va calculer le champ. Les fonctions définies aux lignes 5 à 8 (identiques au code précédent) permettent le calcul des composantes du vecteur champ au point dont les coordonnées sont passées en argument ; elles fonctionnent non seulement avec des flottants mais aussi avec les tableaux bidimensionnels : on calcule ainsi aux lignes 10 et 11 les composantes du champ en chacun des points étudié sans passer par une boucle. De même, la commande quiver accepte comme argument des tableaux numpy bidimensionnels.

### c. Lignes de champs

Un tracé de lignes de champ s'appuie sur une grille réalisée avec la commande meshgrid comme dans l'exemple précédent. La commande streamplot réalise le tracé des lignes de champ ; le code suivant se distingue des précédents par le nombre de point tracés (lignes 1 et 2) et par la modification de la ligne 14 :

```
1  X=np.linspace(-10,10,50)
2  Y=np.linspace(-10,10,50)
3  XX,YY=np.meshgrid(X,Y)
4  #
```

```
5  def champEx(x,y):
6      return x/(x**2+y**2)**1.5
7  def champEy(x,y):
8      return y/(x**2+y**2)**1.5
9  #
10 EX=champEx(XX,YY)
11 EY=champEy(XX,YY)
12 #
13 plt.figure()
14 plt.streamplot(XX,YY,EX,EY)
15 plt.show()
```

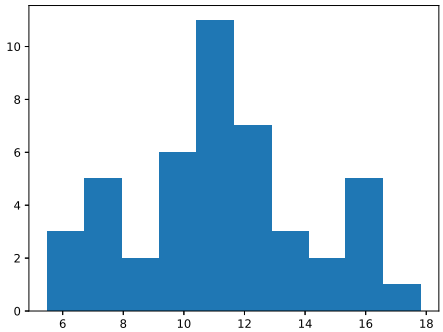



Notons que le tracé de lignes de champ apparaît explicitement dans le programme de deuxième année.

## 4 Histogrammes

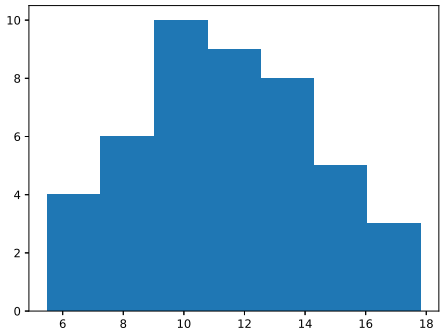
On souhaite représenter l'histogramme d'un ensemble de données, par exemple les notes du dernier devoir surveillé (ou des données expérimentales correspondant à la répétition d'une même mesure). Pour plus de lisibilités, les notes sont d'abord entrées dans trois listes (lignes 1 à 3) qui sont ensuite concaténées :

```
1  L1=[12.7,16.4,13.4,15.9,5.5,14.9,6.8,9.4,9.3,11.1,13.2,11.3,8.5,11.2,7.4]
2  L2=[14.6,12.6,7.9,11.8,12.7,15.5,11.6,10.7,13.2,6.4,12.7,10.0,17.8,8.8,10.6]
3  L3=[10.2,6.2,10.6,12.3,10.3,15.5,7.3,10.6,11.3,9.7,12.6,11.3,7.9,11.4,16.1]
4  L=L1+L2+L3
5  #
6  plt.figure()
7  plt.hist(L)
8  plt.show()
```



Le tracé par défaut a conduit à regrouper les notes en 10 classes ; on peut imposer le nombre de classes grâce au paramètre bins de la commande hist :

```
plt.figure()
plt.hist(L,bins=7)
plt.show()
```




L'aide en ligne de la commande hist précise comment le paramètre bins permet de personnaliser le choix des classes.

### 5 Pour aller plus loin

La documentation en ligne de matplotlib, disponible à l'adresse matplotlib.org répondra à toutes vos questions. Allez y faire un tour pour voir l'étendue des possibilités de cette bibliothèque.

## III Résolution approchée d'équations différentielles

### 1 Résolution d'une équation différentielle du premier ordre par la méthode d'Euler

On souhaite résoudre une équation différentielle du type

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t) = \mathrm{F}(x(t), t) \qquad \text{avec} \qquad x(0) = x_0$$

sur l'intervalle $[t_{\min}, t_{\max}]$ avec la condition initiale $x(t_{\min}) = x_0$. Ceci correspond au « problème de Cauchy » en mathématiques.

#### a. Principe de la méthode

La première étape consiste à discrétiser l'intervalle de temps : on introduit ainsi une suite $t_k, k \in [\![ 0 \,;\, \mathrm{N} ]\!]$ telle que $t_0 = t_{\min}$ et $t_{\mathrm{N}} = t_{\max}$. On peut par exemple choisir

$$t_k = t_{\min} + k\,\frac{t_{\max} - t_{\min}}{\mathrm{N}}$$

qui vérifie bien $t_0 = t_{\min}$ et $t_{\mathrm{N}} = t_{\max}$.

On cherche ensuite à déterminer une valeur approchée de $x(t)$ aux instants $t_k$ et on pose $x_k = x(t_k)$. On écrit

$$x_{k+1} - x_k = x(t_{k+1}) - x(t_k) = \int_{t_k}^{t_{k+1}} x'(t)\,\mathrm{d}t = \int_{t_k}^{t_{k+1}} \mathrm{F}(x(t), t)\,\mathrm{d}t$$

La méthode d'Euler explicite consiste à approcher $\mathrm{F}(x(t), t)$ sur l'intervalle $[t_k ; t_{k+1}]$ par $\mathrm{F}(x(t_k), t_k)$ :

$$\mathrm{F}(x(t), t) \simeq \mathrm{F}(x(t_k), t_k) = \mathrm{F}(x_k, t_k) \qquad \text{donc} \qquad \int_{t_k}^{t_{k+1}} \mathrm{F}(x(t), t)\,\mathrm{d}t \simeq \mathrm{F}(x_k, t_k) \times (t_{k+1} - t_k)$$

Ainsi $$x_{k+1} - x_k \simeq \mathrm{F}(x_k, t_k) \times (t_{k+1} - t_k)$$

et donc $$x_{k+1} \simeq x_k + \mathrm{F}(x_k, t_k) \times (t_{k+1} - t_k)$$

On obtient ainsi une relation de récurrence ; le calcul des $x_k$ pour $k \in [\![ 0 \,;\, \mathrm{N} ]\!]$ est possible grâce à la condition initiale qui donne $x_0$.

### b. Programmation

On peut programmer la méthode d'Euler de la façon suivante, où F est la fonction caractérisant l'équation différentielle, x0 la condition initiale et lt la liste des valeurs de $t$ :

```
1  def euler(F,x0,lt):
2      x=x0 # initialisation
3      L=[x] # on range la première valeur dans une liste
4      for i in range(len(lt)-1): # attention, on connaît la première valeur de x
5          x=x+F(x,lt[i])*(lt[i+1]-lt[i]) # on calcule les valeurs successives de x
6          L.append(x) # on range la valeur de x dans la liste
7      return np.array(L) # on renvoie un tableau numpy
```

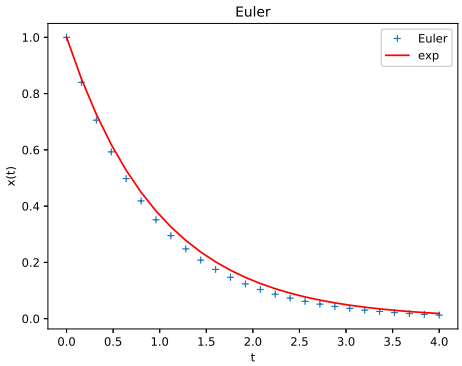
Notons que la commande de la ligne 5 est la traduction de la rélation de récurrence écrite plus haut.

Testons notre programme sur la résolution de l'équation différentielle $x'(t) + x(t) = 0$ sur l'intervalle $[0, 4]$ avec la condition initale $x(0) = 1$. Dans ce cas, on écrit

$$x'(t) = \mathrm{F}(x(t), t) \qquad \text{avec} \qquad \mathrm{F}(x(t), t) = -x(t)$$

On calcule et on trace la solution obtenue par la méthode d'Euler ; on superpose avec la courbe représentative de la solution exacte (une exponentielle) :

```
1  def F(x,t):
2      return -x
3  #
4  tmin,tmax,N=0,4,25 # intervalle de travail et nombre de points
5  x0=1 # condition initiale
6  T=np.linspace(tmin,tmax,N+1)
7  #
8  X=euler(F,x0,T)
9  #
10 plt.figure()
11 plt.plot(T,X,linestyle='None',marker='+',label='Euler')
12 plt.plot(T,np.exp(-T),'r',label='exp')
13 plt.xlabel('t')
14 plt.ylabel('x(t)')
15 plt.title('Euler')
16 plt.legend()
17 plt.show()
```

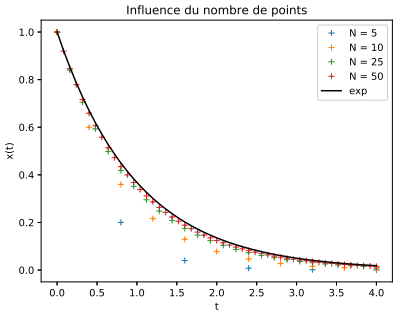



Afin d'observer l'influence du nombre de points, exécutons le programme suivant :

```
1  LN=[5,10,25,50]
2  plt.figure()
```

```
for n in LN:
    T=np.linspace(tmin,tmax,n+1)
    X=euler(F,x0,T)
    plt.plot(T,X,linestyle='None',marker='+',label='N = '+str(n))
plt.plot(T,np.exp(-T),'k',label='exp')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.title('Influence du nombre de points')
plt.legend()
plt.show()
```

Notons l'utilisation de la concaténation pour fabriquer les « étiquettes » associées à chaque courbe.




On observe que l'augmentation de N conduit à une solution approchée qui s'approche de la solution exacte.

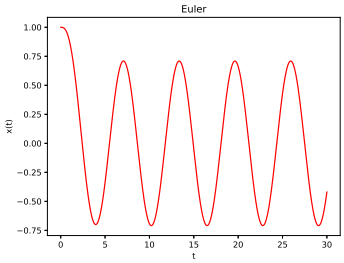
Terminons par un dernier exemple, où l'équation différentielle fait apparaître une dépendance temporelle explicite :

$$x'(t) = -x(t) + \cos(t)$$

Il suffit de modifier la définition de la fonction F ainsi que l'intervalle d'étude :

```
def F(x,t):
    return -x+np.cos(t)
#
tmin,tmax,N=0,30,2500 # intervalle de travail et nombre de points
x0=1 # condition initiale
T=np.linspace(tmin,tmax,N+1)
#
X=euler(F,x0,T)
#
plt.figure()
plt.plot(T,X,'r')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.title('Euler')
plt.show()
```

Quel système physique est décrit par cette équation différentielle ?



### c. Limites

De nombreuses équations différentielles ne sont pas résolues efficacement par la méthode d'Euler : cette la méthode, qui a le bon goût d'être très simple, n'est pas assez précise[3]. Il nous faut donc utiliser une méthode plus sophistiquée que la méthode d'Euler si l'on veut obtenir des résultats de meilleure qualité.

## 2 Résolution d'une équation différentielle du premier ordre avec la commande odeint

### a. Principe

La méthode d'Euler repose sur l'approximation

$$\int_{t_k}^{t_{k+1}} \mathrm{F}(x(t),t)\,\mathrm{d}t \simeq \int_{t_k}^{t_{k+1}} \mathrm{F}(x(t_k),t_k)\,\mathrm{d}t = \mathrm{F}(x_k,t_k)\times(t_{k+1}-t_k)$$

On cherche donc une approximation plus précise. Par exemple, la méthode de Runge-Kutta subdivise l'intervalle $[t_k,t_{k+1}]$ pour arriver à une approximation plus précise. De nombreuses approches sont possibles, nous n'entrons pas dans les détails.

### b. La commande odeint

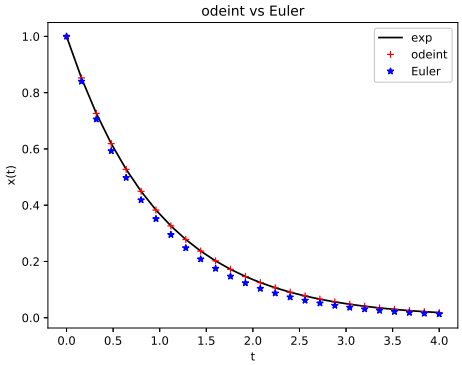
Parmi les méthodes numériques de résolution des équations différentielles, l'une est implémantée en python au travers de la fonction odeint ; c'est elle qui figure au programme. Nous ne nous préoccuperons pas de la méthode mathématique sous-jacente[4], nous nous contentons de l'utiliser.

Sa synthaxe est identique à la fonction euler introduite précédemment ; utilisons-la pour résoudre la même équation que précédemment (il suffit de remplacer euler par odeint dans les programmes précédents, sous réserve de l'avoir importée) :

```
from scipy.integrate import odeint
#
def F(x,t):
    return -x
#
tmin,tmax,N=0,4,25 # intervalle de travail et nombre de points
x0=1 # condition initiale
T=np.linspace(tmin,tmax,N+1)
#
Xeuler=euler(F,x0,T)
Xodeint=odeint(F,x0,T)
#
plt.figure()
plt.plot(T,np.exp(-T),color='k',label='exp')
plt.plot(T,Xodeint,linestyle='None',color='r',marker='+',label='odeint')
plt.plot(T,Xeuler,linestyle='None',color='b',marker='*',label='Euler')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.title('odeint vs Euler')
plt.legend()
plt.show()
```

Cette méthode est plus efficace que la méthode d'Euler, comme le montre le graphe suivant :

[3] Nous donnerons un exemple plus loin.
[4] Il s'agit d'une méthode à pas variable, appelée « lsoda ».




## 3 Systèmes différentiels d'ordre 1

On s'intéresse au système différentiel d'ordre 1 suivant où les fonctions $x$ et $y$ vérifient les équations différentielles couplées suivantes :

$$\begin{cases} x'(t) = x(t)(1-y(t)) \\ y'(t) = y(t)(x(t)-1/2) \end{cases}$$

ainsi que les conditions initiales $x(0) = y(0) = 1/2$. Ce système d'équations différentielles appelé « système proie-prédateur de Lotka-Volterra », il a été introduit dans le contexte de l'étude de la dynamique des populations animales.

Introduisons le « vecteur » $\mathrm{X}(t)$ défini par

$$\mathrm{X}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

Le système d'équations différentielles s'écrit

$$\frac{\mathrm{dX}}{\mathrm{d}t}(t) = \mathrm{F}(\mathrm{X}(t), t) \qquad \text{avec} \qquad \mathrm{F}\left(\begin{pmatrix} u \\ v \end{pmatrix}, t\right) = \begin{pmatrix} u(1-v) \\ v(u-1/2) \end{pmatrix}$$

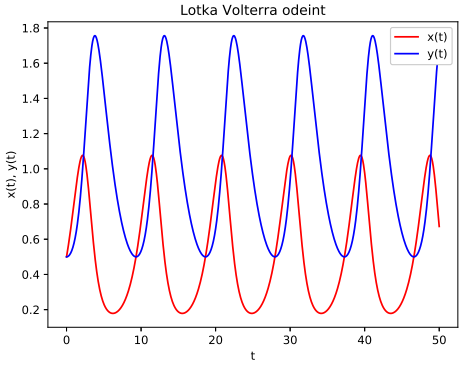
et la condition initiale s'écrit

$$\mathrm{X}(0) = \mathrm{X}_0 \qquad \text{avec} \qquad \mathrm{X}_0 = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$$

Grâce à l'utilisation des tableaux numpy, la résolution du système différentiel avec odeint est identique à celle des équations différentielles vues dans la partie précédente :

```
def F(X,t):
    x,y=X[0],X[1] # on récupère les composantes du vecteur X
    return np.array([x*(1-y),y*(x-0.5)])
#
tmin,tmax,N=0,50,1000 # intervalle de travail et nombre de points
X0=np.array([0.5,0.5]) # condition initiale
T=np.linspace(tmin,tmax,N+1)
#
LKodeint=odeint(F,X0,T) # tableau numpy comportant 2 colonnes
xodeint=LKodeint[:,0] # les valeurs de x sont dans la première colonne
yodeint=LKodeint[:,1] # les valeurs de y sont dans la deuxième colonne
#
plt.figure()
plt.plot(T,xodeint,color='r',label='x(t)')
plt.plot(T,yodeint,color='b',label='y(t)')
plt.xlabel('t')
plt.ylabel('x(t), y(t)')
plt.title('Lotka Volterra odeint')
plt.legend()
plt.show()
```

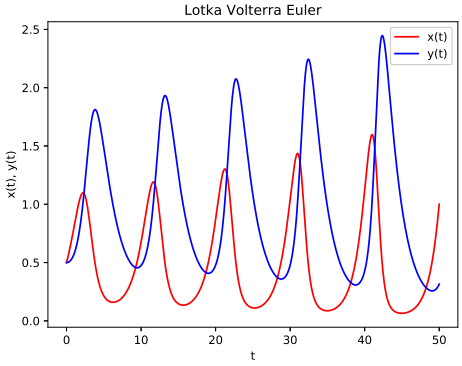
La variable LKodeint renvoyée par la commande odeint est maintenant un tableau bidimensionnel qui peut être vu comme une matrice dont la première colonne contient les valeurs de $x$ la deuxième colonne les valeurs de $y$.




On constate que les solutions semblent périodiques ; c'est l'une des caractéristiques de ce système différentiel.

Notons que l'on peut faire la résolution de la même façon par la méthode d'Euler ; la façon dont nous avons programmé la fonction euler est compatible avec des tableaux numpy

```
LKeuler=euler(F,X0,T) # tableau numpy comportant 2 colonnes
xeuler=LKeuler[:,0] # les valeurs de x sont dans la première colonne
yeuler=LKeuler[:,1] # les valeurs de y sont dans la deuxième colonne
#
plt.figure()
plt.plot(T,xeuler,color='r',label='x(t)')
plt.plot(T,yeuler,color='b',label='y(t)')
plt.xlabel('t')
plt.ylabel('x(t), y(t)')
plt.title('Lotka Volterra Euler')
plt.legend()
plt.show()
```




Les solutions semblent ici diverger, ce qui montre à nouveau les limites de la méthode d'Euler.

Nous avons illustré la résolution d'un système de deux équations différentielles couplées ; la démarche peut en fait s'appliquer à un nombre quelconque d'équations différentielles.

## 4 Vectorisation d'un équation différentielle d'ordre supérieur ou égal à 2

Intéressons-nous à une équation différentiellle d'ordre 2 en commençant par l'exemple de l'oscillateur harmonique (adimensionné) dont l'équation s'écrit

$$x''(t) + x(t) = 0$$

Choisissons les conditions initiales suivantes

$$x(0) = 1 \qquad \text{et} \qquad x'(0) = 0$$

Cette équation différentielle se vectorise de la façon suivante :

$$\frac{\mathrm{dX}}{\mathrm{d}t} = \mathrm{F}(\mathrm{X}(t), t) \qquad \text{avec} \qquad \mathrm{X} = \begin{pmatrix} y \\ v \end{pmatrix} \qquad \text{et} \qquad \mathrm{F}\left(\begin{pmatrix} y \\ v \end{pmatrix}, t\right) = \begin{pmatrix} v \\ -y \end{pmatrix}$$

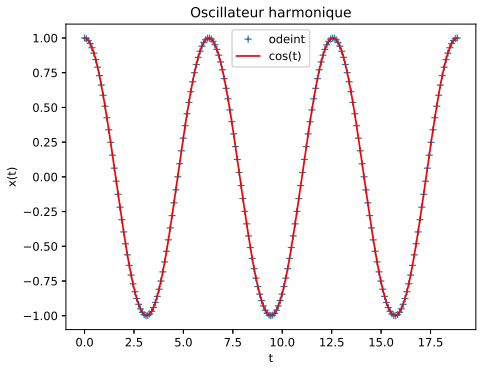
et la condition initiale s'écrit

$$\mathrm{X}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

On est donc passé d'une équation différentielle d'ordre 2 à un système différentiel d'ordre 1 et de dimension 2. On peut ainsi la résoudre avec odeint.

```python
def F(X,t):
    x,v=X # on récupère les composantes de X
    return np.array([v,-x])
tmin,tmax,N=0,6*np.pi,200
X0=np.array([1,0]) # condition initiale y0=1 et v0=0
T=np.linspace(tmin,tmax,N+1)
X=odeint(F,X0,T)
x=X[:,0] # les valeurs de la position sont dans la première colonne du tableau
#
plt.figure()
plt.plot(T,x,linestyle='None',marker='+',label='odeint')
plt.plot(T,np.cos(T),'r',label='cos(t)')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.title('Oscillateur harmonique')
plt.legend()
plt.show()
```

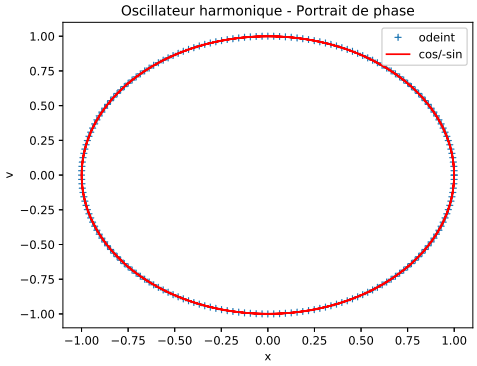
On superpose la solution donnée par odeint et la solution exacte $x(t) = \cos(t)$.



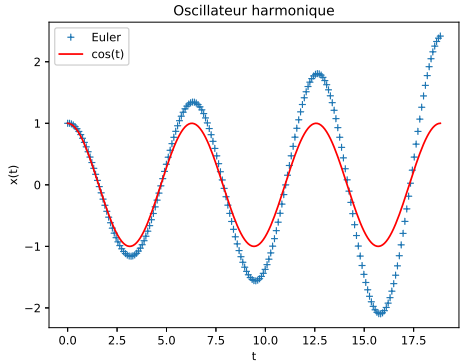

On trace ensuite le portrait de phase :

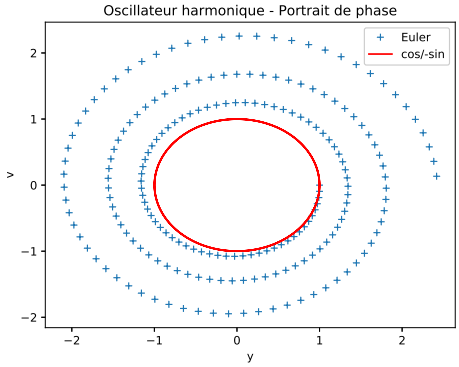
```python
v=X[:,1] # les valeurs de la "vitesse" sont dans la deuxième colonne
plt.figure()
plt.plot(x,v,linestyle='None',marker='+',label='odeint')
plt.plot(np.cos(T),-np.sin(T),'r',label='cos/-sin')
plt.xlabel('y')
plt.ylabel('v')
plt.title('Oscillateur harmonique - Portrait de phase')
plt.legend()
plt.show()
```

où l'on a posé $v(t) = x'(t)$ (appelé abusivement « vitesse » par analogie avec la mécanique) ; la solution exacte s'écrit $v(t) = -\sin(t)$.




Le tracé par la méthode d'Euler est laissé en exercice[5] On obtient les courbes suivantes :




On constate encore une fois l'inefficacité de la méthode d'Euler qui conduit à une solution divergente.

# IV Équations algébriques

Dans cette partie, on cherche à résoudre des équations de la forme $f(x) = 0$. On se limite au cas où la recherche est effectuée sur un intervalle $[a, b]$ où la fonction $f$

- est continue et strictement monotone,
- vérifie $f(a)f(b) < 0$.

Ces conditions assure que $f$ s'annule une et une seule fois sur $]a; b[$.

## 1 Dichotomie

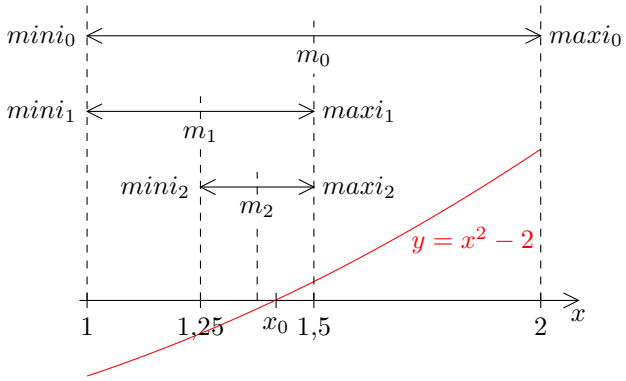
La méthode de dichotomie permet de déterminer la solution[6] d'une équation $f(x) = 0$ sur un intervalle $[a; b]$ tel que $f(a)$ et $f(b)$ sont de signe différent. Le principe de la recherche dichotomique est de déterminer si la racine $x_0$ appartient à l'intervalle $[a; (a + b)/2]$ ou à l'intervalle $[(a + b)/2; b]$, puis de répéter l'opération. À chaque étape, la largeur de l'intervalle contenant $x_0$ est divisée par 2 ; après $n$ étapes, on obtient un intervalle de largeur $(b - a)/2^n$.

Formalisons cette méthode, en notant $\varepsilon$ la précision souhaitée.

- initialisation : mini=a, maxi=b
- tant que maxi-mini $>$ 2 epsilon répéter m = (maxi+mini)/2
  - si f(mini) x f(m) $>$ 0, alors mini = m,
  - sinon maxi = m.

[5] Il suffit en fait de remplacer odeint par euler dans le code précédent.
[6] supposée unique.

On illustre cette méthode à la résolution de l'équation $x^2 - 2 = 0$ sur l'intervalle $[1; 2]$ à l'aide du schéma suivant :




Le code suivant réalise la recherche dichotomique

```
def dichotomie(f,a,b,epsilon):
    mini,maxi=a,b
    while maxi-mini>2*epsilon:
        m=(maxi+mini)/2
        if f(mini)*f(m)>0:
            mini=m
        else:
            maxi=m
    return (mini+maxi)/2
#
def ftest(x):
    return x**2-2
#
print(dichotomie(ftest,1,2,0.001))
```

On obtient

```
1.4150390625
```

qui constitue une approximation de $\sqrt{2}$.

### 2 Utilisation de bisect

La fonction bisect de la bibliothèque scipy.optimize recherche la solution d'une équation $f(x) = 0$ sur un intervalle $[a; b]$ où $f(a)$ et $f(b)$ sont de signe différent. Voici un exemple d'utilisation (même situation que pour la dichotomie) :

```
from scipy.optimize import bisect
#
def ftest(x):
    return x**2-2
#
print(bisect(ftest,1,2))
```

Il vient

```
1.4142135623715149
```

Notons que l'utilisateur ne spécifie pas la précision de la résolution (contrairement à la dichotomie).

## V Régression linéaire et autres ajustements

Les séances de TP conduisent à des situations où il s'agit d'exploiter des données expérimentales pour déterminer des paramètres liés à une modélisation. Le cas le plus courant[7] est celui où la modélisation est une loi affine : la méthode

[7] Le seul qui figure au programme

d'ajustement est la régression linéaire.

## 1 Régression linéaire

Il existe plusieurs façons de faire des régressions linéaires en python. Le programme impose l'utilisation de la fonction polyfit de la bibliothèqe numpy ; elle permet de faire un ajustement polynomial de degré quelconque. Nous utiliserons uniquement des polynômes de degré 1, ce qui correspond au cas de la régression linéaire[8]. On cherche donc à ajuster des données $\{(x_i, y_i\}_{1\leqslant i\leqslant \mathrm{N}}$ avec une loi affine $y = a\,x + b$.

Voici la syntaxe :

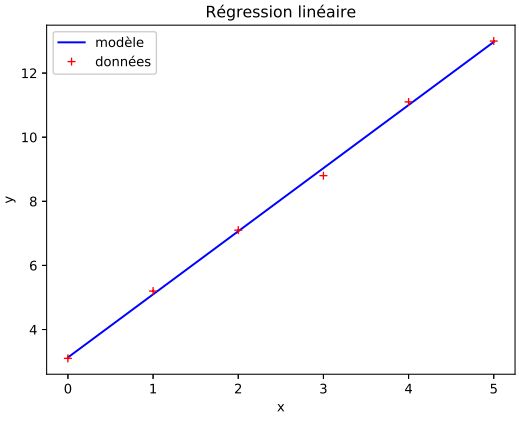
```
x=np.array([0,1,2,3,4,5]) #valeurs de x
y=np.array([3.1,5.2,7.1,8.8,11.1,13.0]) # valeurs de y
#
a_est,b_est=np.polyfit(x,y,1) # pente et ordonnée à l'origine estimées
```

Les deux premiers arguments de polyfit sont les listes des valeurs de $x$ et de $y$, le troisième argument est le degré du polynôme utilisé pour l'ajustement (ici 1). La fonction polyfit renvoie un tableau numpy contenant les coefficients du polynôme (en commençant par le terme de plus haut degré, ici la pente).

On obtient respectivement 1.968571428571428 et 3.1285714285714294 pour $a$ et $b$, sachant que les données ont été fabriquées avec $a = 2$ et $b = 3$ (un bruit gaussien a été ajouté sur les ordonnées, de valeur moyenne nulle et d'écart-type 0,2).

Superposons les données et le modèle

```
plt.figure()
plt.plot(x,a_est*x+b_est,color="blue",label="modèle")
plt.plot(x,y,linestyle='None',marker='+',color='r',label="données")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Régression linéaire')
plt.show()
```




## 2 Autres ajustements

D'autres ajustements peuvent être réalisés en se ramenant à une loi affine ; c'est par exemple le cas pour une loi de puissance

$$y = a\,x^b$$

pour laquelle il suffit d'écrire $\ln y = a + b \ln x$ ; on ajuste le modèle en effectuant le régression linéaire de $\ln y$ en fonction de $\ln x$. De même, pour une loi exponentielle

[8] Notons qu'on devrait parler de régression affine car il s'agit d'ajuster des données à un modèle affine et non pas linéaire.

$$y = a\mathrm{e}^{b\,x}$$

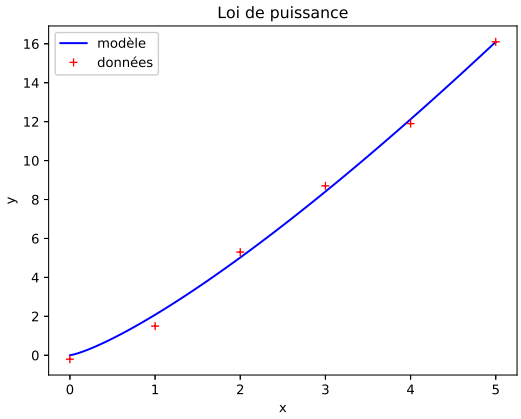
pour laquelle on écrit $\ln y = \ln a + b\,x$ ; on ajuste le modèle en effectuant la régression linéaire de $\ln(y)$ en fonction de $x$.

Cette façon de faire est mathématiquement incorrecte, bien que ce soit la seule qui soit compatible avec les outils du programme. On peut cependant imaginer aller plus loin, en utilisant la fonction curve_fit de la bibliothèque scipy.optimize. Elle permet en effet de faire tout type d'ajustement. Nous en donnons un exemple d'utilisation à titre purement indicatif.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
#
x=np.array([0,1,2,3,4,5]) #valeurs de x
y=np.array([-0.2,1.5,5.3,8.7,11.9,16.1]) # valeurs de y
#
def f(x,a,b): # modèle utilisé pour l'ajustement
    return a*x**b
#
a_est,b_est=curve_fit(f,x,y)[0] # attention à l'ordre des paramètres
```

On obtient respectivement 2.0797179930991336 et 1.2715432626417429 pour $a$ et $b$, sachant que les données ont été fabriquées avec $a = 2$ et $b = 1.3$ (un bruit gaussien a été ajouté sur les ordonnées, de valeur moyenne nulle et d'écart-type 0,2). Représentons les données et le modèle :

```
xt=np.linspace(x[0],x[-1],100)
plt.figure()
plt.plot(xt,a_est*xt**b_est,color="blue",label="modèle")
plt.plot(x,y,linestyle='None',marker='+',color='r',label="données")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Loi de puissance')
plt.show()
```




# VI Intégration, dérivation

Les notions introduites dans cette partis correspondent au troisième item de l'annexe 3 du programme de première année. Ces notions ne sont cependant reliées à aucune des capacités numériques du programme.

# 1 Calcul approché d'une intégrale : méthode des rectangles

## a. Intégration d'une fonction sur un segment

On cherche à calculer l'intégrale d'une fonction $f$ sur un intervalle $[a;b]$ :

$$\mathrm{I} = \int_a^b f(x)\,\mathrm{d}x$$

Pour cela, on subdivise l'intervalle $[a;b]$ en $n$ intervalles du type $[x_i;x_{i+1}]$ avec $i \in [\![\,0\,;\,n-1\,]\!]$ où

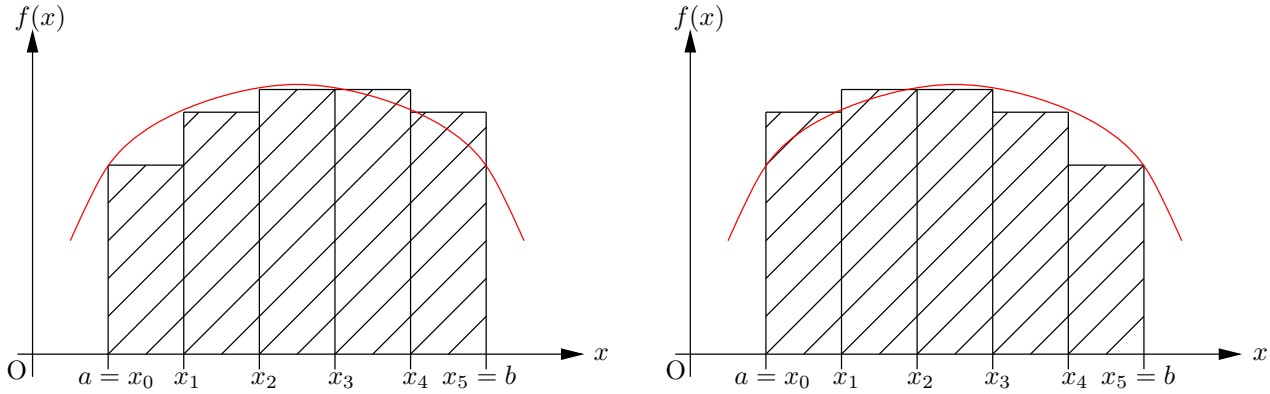
$$x_i = a + \frac{b-a}{n}\,i$$

Sur chacun des intervalle $[x_i;x_{i+1}]$, la fonction $f$ est approchée par une fonction constante. On approche ainsi $f$ sur $[a;b]$ par une fonction en escalier.

Deux approximations sont possibles pour $f$ sur l'intervalle $[x_i;x_{i+1}]$ :

$$f(x) \simeq f(x_i) \qquad \text{ou} \qquad f(x) \simeq f(x_{i+1})$$

On parle respectivement de méthode des rectangles à gauche et de méthode des rectangles à droite[9]. Illustrons ces deux méthodes par un schéma pour $n = 5$.




Ces deux méthodes aboutissent aux valeurs approchées suivantes de I :

$$\mathrm{I} \simeq \left(\frac{b-a}{n}\right)\sum_{i=0}^{n-1} f(x_i) \qquad \text{ou} \qquad \mathrm{I} \simeq \left(\frac{b-a}{n}\right)\sum_{i=1}^{n} f(x_i)$$

Elles se programment simplement en python :

```
def rect_g(f,a,b,n):
    h=(b-a)/n
    x,I=a,0
    for i in range(0,n):
        I=I+f(x)
        x=x+h
    return I*h
#
def rect_d(f,a,b,n):
    h=(b-a)/n
    x,I=a+h,0
    for i in range(0,n):
        I=I+f(x)
        x=x+h
    return I*h
#
def ftest(x):
```

[9] On pourrait également considérer la méthode des rectangles au point milieu, qui consiste à approcher $f(x)$ sur $[x_i;x_{i+1}]$ par la valeur prise par $f$ au milieu de l'intervalle ; cela revient à la méthode des trapèzes, où l'on approche $f$ sur $[x_i;x_{i+1}]$ par une fonction affine qui prend les mêmes valeurs en $x_i$ et $x_{i+1}$ que $f$.

```
    return x**2
#
print(rect_g(ftest,1,2,500),rect_d(ftest,1,2,500))
```

On obtient

```
2.330334 2.3363340000000004
```

qui consituent d'assez bonnes approximation du résultat exact (qui vaut 7/3).

Notons que la fonction quad de la bibliothèque scipy.integrate fait mieux, en renvoyant non seulement une valeur approchée de l'intégrale mais également une évaluation de l'erreur :

```
from scipy.integrate import quad
#
print(quad(ftest,1,2))
```

```
(2.3333333333333335, 2.590520390792032e-14)
```

La connaissance de cette fonction ne fait pas partie des exigences du programme.

### b. Intégration de données expérimentales

En physique, un capteur couplé à un un ordinateur renvoie des données que nous supposerons, une fois mise en forme, sous la forme de deux listes : une liste des valeurs du temps, et une liste des valeurs de la grandeur mesurée par le capteur. Par exemple, si le capteur est un accéléromètre, la grandeur mesurée est une composante de l'accélération. Dans ce cas, si l'on souhaite déterminer la vitesse, il faut intégrer l'accélération.

En notant $\{t_i\}_{1\leqslant i\leqslant n}$ est différents instants où on été réalisées les mesures, et $\{a_i\}_{1\leqslant i\leqslant n}$ les accélérations mesurées à ces instants, on peut écrire :

$$v_i = v(t_i) = v_0 + \int_0^{t_i} a(t)\,\mathrm{d}t \simeq v_0 + \sum_{j=1}^{i-1} a_j\,(t_{j+1} - t_j) \qquad \text{ou} \qquad v_i = v(t_i) = v_0 + \int_0^{t_i} a(t)\,\mathrm{d}t \simeq v_0 + \sum_{j=1}^{i-1} a_{j+1}\,(t_{j+1} - t_j)$$

selon que l'on utilise la méthode des rectangles à gauche ou à droite. On en déduit le code suivant :

```
def int_g(T,L):
    I=[0]
    for i in range(len(L)-1):
        I.append(I[-1]+L[i]*(T[i+1]-T[i]))
    return I
#
def int_d(T,L):
    I=[0]
    for i in range(len(L)-1):
        I.append(I[-1]+L[i+1]*(T[i+1]-T[i]))
    return I
```

où T et L sont respectivement les listes des valeurs du temps et de la grandeur que l'on veut intégrer. On a supposé la valeur initiale de la vitesse nulle.

## 2 Calcul approché du nombre dérivé d'une fonction en un point

Par définition, le nombre dérivé de $f$ en $x_0$ est la limite (si elle existe)

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}$$

Cette limite ne peut bien-sûr pas être traduite directement sur le plan numérique.

### a. Dérivation d'une fonction

On peut approcher la dérivée par le taux d'accroissement

$$f'(x_0) \simeq \frac{f(x_0+h)-f(x_0)}{h} \qquad \text{ou} \qquad f'(x_0) \simeq \frac{f(x_0)-f(x_0-h)}{h}$$

sous réserve que $h$ soit suffisamment petit. Ces deux expressions correspondent respectivement à la dérivée à droite et à la dérivée à gauche (si $h$ est positif). On peut également faire la moyenne de ces deux expressions pour obtenir

$$f'(x_0) \simeq \frac{f(x_0+h)-f(x_0-h)}{2h}$$

On les programme de la façon suivante :

```
def drv_d(f,x0,h):
    return (f(x0+h)-f(x0))/h
#
def drv_g(f,x0,h):
    return (f(x0)-f(x0-h))/h
#
def drv_m(f,x0,h):
    return (drv_g(f,x0,h)+drv_d(f,x0,h))/2
```

Effectuons un essai

```
def ftest(x):
    return x**2
#
print(drv_d(ftest,1,0.01),drv_g(ftest,1,0.01),drv_m(ftest,1,0.01))
```

ce qui donne

```
2.0100000000000007 1.9900000000000029 2.0000000000000018
```

alors que la valeur attendue est 2.

### b. Dérivation de données expérimentales

Ce dernier point est abordé à titre indicatif, car il ne figure pas au programme ; il peut cependant être utile en TIPE. Par exemple, si une expérience de mécanique est filmée, un logiciel de pointage permet de déterminer la position d'un mobile. Le traitement en python des données permet de déterminer la vitesse par dérivation. On suppose que les données sont récupérées sous la forme de deux listes[10] (de même longueur).

Pour la programmation, il faut être vigilant au fait que

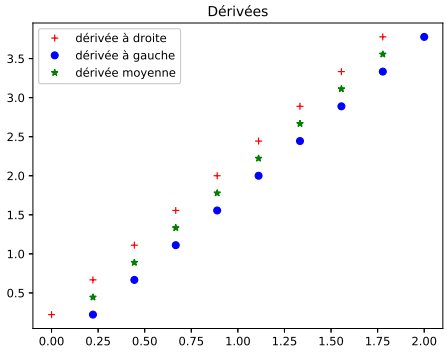
- la dérivée à droite ne peut pas être calculée sur le dernier point,
- la dérivée à gauche ne peut pas être calculée sur le premier point,
- la dérivée moyenne ne peut pas être calculée sur le premier et le dernier point.

Il faut par ailleurs faire attention aux indices de la liste représentant le temps lors de la représentation graphique.

```
import numpy as np
import matplotlib.pyplot as plt
#
T=np.linspace(0,2,10)
L=T**2
#
def drv_d(T,L):
    d=[]
    for i in range(len(T)-1):
```

[10] ou deux tableaux numpy unidimensionnels.

```
        d.append((L[i+1]-L[i])/(T[i+1]-T[i]))
    return d
#
def drv_g(T,L):
    d=[]
    for i in range(1,len(T)):
        d.append((L[i]-L[i-1])/(T[i]-T[i-1]))
    return d
#
def drv_m(T,L):
    d=[]
    for i in range(1,len(T)-1):
        d.append((L[i+1]-L[i-1])/(T[i+1]-T[i-1]))
    return d
#
plt.figure()
plt.plot(T,L,'b+',label='position')
plt.show()
#
plt.figure()
plt.plot(T[:-1],drv_d(T,L),'+r',label="dérivée à droite")
plt.plot(T[1:],drv_g(T,L),'ob',label="dérivée à gauche")
plt.plot(T[1:-1],drv_m(T,L),'*g',label="dérivée moyenne")
plt.title('Dérivées')
plt.legend()
plt.show()
```




Notons enfin que la dérivation (qui correspond à un filtrage passe-haut) de données bruitées (le bruit étant associées aux hautes fréquences) pose problème : on amplifie le bruit. Il est recommandé de lisser des données expérimentales avant de procéder à la dérivation.

# VII Variables aléatoires et statistiques

Les variables aléatoires sont particulièrement utiles dans le traitement des incertitudes en physique-chimie. Avant d'aborder leur traitement en python, on commence par quelques notions mathématiques.

## 1 Variables aléatoires à densité

Mathématiquement, on dit qu'une variable aléatoire X (réelle) est une « variable aléatoire à densité » s'il existe une fonction $f$ positive et intégrable appelée « fonction de densité » ou « densité de probabilité » telle que la probabilité de l'événement $\mathrm{X} \in [x_1 ; x_2]$ s'écrive

$$\mathrm{P}(x_1 \leqslant \mathrm{X} \leqslant x_2) = \int_{x_1}^{x_2} f(x)\,\mathrm{d}x$$

La densité de probabilité vérifie

$$\int_{-\infty}^{+\infty} f(x)\,\mathrm{d}x = 1$$

L'espérance de la variable aléatoire X, notée $\mathbb{E}(\mathrm{X})$, est définie par

$$\mathbb{E}(\mathrm{X}) = \int_{-\infty}^{+\infty} x\, f(x)\,\mathrm{d}x$$

Elle correspond à la « moyenne » des valeurs prises par la variable aléatoire X. La variance et l'écart-type renseignent sur la dispersion de $f$ autour de $\mathbb{E}(\mathrm{X})$ :

$$\mathrm{V}(\mathrm{X}) = \mathbb{E}(\mathrm{X} - \mathbb{E}(\mathrm{X})) = \mathbb{E}(\mathrm{X}^2) - (\mathbb{E}(\mathrm{X}))^2 \qquad \text{et} \qquad \sigma(\mathrm{X}) = \sqrt{\mathrm{V}(\mathrm{X})}$$

## 2 Estimateurs de la moyenne et de l'écart-type

En notant $\mathrm{X}_i$ les différentes valeurs obtenues lors d'une sucession de tirages aléatoires de la variable aléatoire X, on obtient un estimateur (non-biaisé) de l'espérance de X en calculant la moyenne arithmétique des $\mathrm{X}_i$ :

$$\overline{\mathrm{X}_i} = \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \mathrm{X}_i$$

la fonction np.mean le fait automatiquement :

```
Xi=[1,2,3,4,5] # résultats (fictifs) d'une sucession de tirages aléatoires
moy=np.mean(Xi)
print(moy)
```

donne 3.0 comme attendu. Notons que la np.mean peut agir sur une liste ou un tableau numpy.

On peut espérer obtenir un bon estimateur de l'écart-type de la variable aléatoire X en calculant l'écart-type de la série $\{\mathrm{X}_i\}_{1\leqslant i\leqslant \mathrm{N}}$ défini par

$$\sigma(\{\mathrm{X}_i\}_{1\leqslant i\leqslant \mathrm{N}}) = \sqrt{\frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} (\mathrm{X}_i - \overline{\mathrm{X}_i})^2}$$

Le calcul de l'écart-type est réalisé par la commande numpy.std :

```
ecart_type=np.std(Xi)
print(ecart_type)
```

On obtient

```
1.4142135623730951
```

En fait, l'écart-type de la série $\{\mathrm{X}_i\}_{1\leqslant i\leqslant \mathrm{N}}$ n'est pas un bon estimateur de l'écart-type de la variable aléatoire X, car il est biaisé. Un estimateur non biaisé de l'écart-type de la variable aléatoire X est donné par

$$\sigma'(\{\mathrm{X}_i\}_{1\leqslant i\leqslant \mathrm{N}}) = \sqrt{\frac{1}{\mathrm{N}-1} \sum_{i=1}^{\mathrm{N}} (\mathrm{X}_i - \overline{\mathrm{X}_i})^2}$$

La fonction std de numpy peut faire le travail

```
ecart_type2=np.std(Xi,ddof=1)
print(ecart_type2)
```

On obtient

```
1.5811388300841898
```

Su l'exemple choisi, $\mathrm{N} = 5$ et la différence entre les deux estimateurs de l'écart-type de la variable X est significatif. Cette différence devient d'autant plus faible que N est grand.

## 3 Simulation d'une loi normale en Python

Rappelons d'abord que la **loi normale** d'espérance $\mu$ et d'écart-type $\sigma$ est la loi de probabilité définie par la densité de probabilité gaussienne

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

Lorsqu'une variable aléatoire X suit une loi normale d'espérance $\mu$ et d'écart-type $\sigma$, on note en général

$$\mathrm{X} \sim \mathcal{N}(\mu, \sigma^2)$$

Simulons un tirage aléatoire selon une loi normale d'espérance 10 et d'écart-type 10 à l'aide de la commande normal de la bibliothèque numpy.random :

```
import numpy.random as rd
mu,sigma=10,2
print(rd.normal(mu,sigma))
```

Le résultat est obtenu est bien évidemment différent à chaque tirage. On peut effectuer plusieurs tirages simultanément en ajoutant un paramètre à la commande normal, et on obtient alors un tableau numpy unidimensionnel :

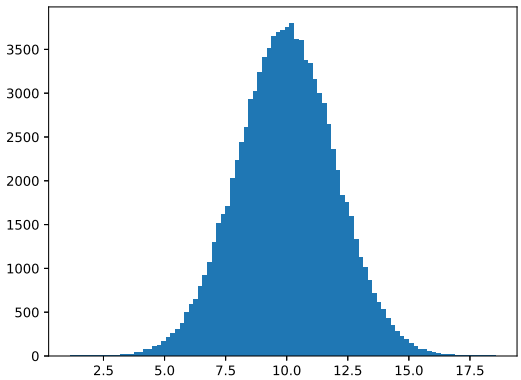
```
mu,sigma=10,2
N=5
Xi=rd.normal(mu,sigma,N)
print(Xi)
print(type(Xi))
```

On obtient par exemple :

```
[12.37959546 15.15341346 11.35762124  5.16725754  8.14610727]
<class 'numpy.ndarray'>
```

Effectuons maintenant un grand nombre de tirages aléatoires, et traçons l'histogramme correspondant :

```
mu,sigma=10,2
N=100000
Xi=rd.normal(mu,sigma,N)
#
plt.figure()
plt.hist(Xi,bins='rice')
plt.show()
```




On reconnaît bien l'allure d'une gaussienne. Notons que le paramètre bins='rice' est d'un usage assez courant pour déterminer le nombre de classes automatiquement.

Estimons l'espérance et l'écart-type de loi de probabilité à partie de la série de tirages :

```
moy,ecart_type=np.mean(Xi),np.std(Xi,ddof=1)
print('Espérance', moy)
print('Ecart-type', ecart_type)
```

On obtient

```
Espérance 9.9850903478506
Ecart-type 1.9990217473254668
```

Ces estimations sont cohérentes avec les paramètres utilisés (respectivement 10 et 2) ; l'estimations est d'autant meilleure que N est élevé.

Enfin, si on veut réaliser en une seule fois des tirages associés à des espérances et des écarts-types différents, il suffit de ranger les espérances et les écarts-types dans des tableaux numpy (qui doivent avoir la même taille) :

```
Lmu=np.array([5,10,15,20,25])
Lsigma=np.array([0.5,1,1.5,2,2.5])
print(rd.normal(Lmu,Lsigma))
```

On obtient

```
[ 5.23475611  9.80721962 15.51329309 21.64373647 22.55206238]
```

Le résultat est un tableau numpy ; la première valeur correspond à un tirage aléatoire réalisé à partir d'une loi normale d'espérance 5 et d'écart-type 0,5, la deuxième valeur à un tirage aléatoire réalisé à partir d'une loi normale d'espérance 10 et d'acart-type 1 etc.

On peut enfin utiliser le raccourci suivant si tous les écarts-types sont identiques :

```
Lmu=np.array([5,10,15,20,25])
sigma=1
print(rd.normal(Lmu,sigma))
```

## 4 Simulation d'une loi uniforme en Python

La loi uniforme n'est pas au programme de physique-chimie bien qu'elle puisse s'avérer utile. Nous la décrivons, ainsi que son utilisation en python, à titre indicatif.

On dit qu'une variable aléatoire à densité suit une loi de probabilité **uniforme** sur un intervalle $[a;b]$ si sa densité de probabilité est nulle en dehors de $[a;b]$ et constante sur $[a;b]$. On en déduit

$$f(x)=\frac{1}{b-a} \quad \text{si} \quad x\in[a;b] \qquad \text{et} \qquad f(x)=0 \quad \text{sinon}$$

Son espérance et son écart-type valent respectivement

$$\mathbb{E}(\mathrm{X})=\frac{a+b}{2} \qquad \text{et} \qquad \sigma(\mathrm{X})=\frac{b-a}{2\sqrt{3}}$$

Notons que l'espérance de la variable aléatoire correspond au milieu de l'inervalle $[a;b]$, conformément à l'intuition. Quant à l'écart-type, il correspond à la demi-largeur de l'intervalle $[a;b]$ divisée par $\sqrt{3}$ (c'est l'origine du facteur $\sqrt{3}$ que l'on retrouve dans les évaluations d'incertitudes de type B).

On peut simuler une loi uniforme en python avec la commande uniform de la bibliothèque numpy.random ; ainsi, un tirage aléatoire suivant une loi de probabilité uniforme sur l'intervalle $[1;2]$ s'obtient de la façon suivante :

```
a,b=1,2
print(rd.uniform(a,b))
```

Pour réaliser N tirages aléatoires, la synthaxe est calquée sur celle de la loi normale

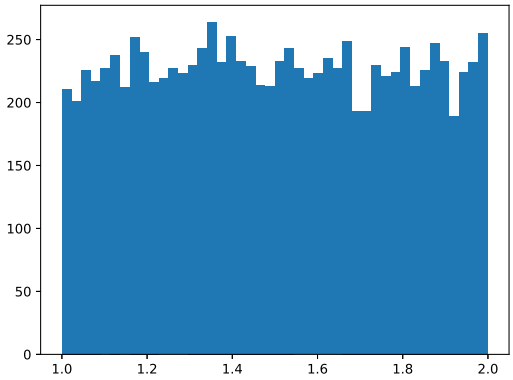
```
a,b=1,2
N=10
#
X=rd.uniform(a,b,N)
print(X)
```

Par exemple, on peut obtenir

```
[1.29657548 1.68398738 1.34672775 1.64470407 1.79006865 1.62936552
 1.6176828  1.69803053 1.71734602 1.42110821]
```

Traçons maintenant un histogramme d'un grand nombre de tirages :

```
a,b=1,2
N=10000
#
Xi=rd.uniform(a,b,N)
#
plt.figure()
plt.hist(Xi,bins='rice')
plt.show()
```




Estimons maintenant l'espérance et l'écart-type de la loi uniforme

```
moy,ecart_type=np.mean(Xi),np.std(Xi,ddof=1)
print('Espérance estimée', moy)
print('Ecart-type estimé', ecart_type)
```

```
Espérance estimée 1.5003009892937629
Ecart-type estimé 0.2873339566344716
```

et comparons aux valeurs attendues

```
m,e_t=(a+b)/2,(b-a)/2/np.sqrt(3)
print('Espérance', m)
print('Ecart-type', e_t)
```

```
Espérance 1.5
Ecart-type 0.2886751345948129
```

Plus N augmente, meilleure est l'estimation.

Si l'on veut effectuer simultanément un tirage aléatoire suivant une loi uniforme dans plusieurs intervalles $[a_1, b_1]$, $[a_2, b_2]$ etc., il suffit de ranger les valeurs des $a_i$ dans un tableau numpy unidimensionnel, puis de faire la même chose avec les valeurs des $b_i$, et enfin d'appeler la fonction uniform avec ces deux tableaux comme paramètres :

```
a=np.array([1, 4, 8, 12])
b=np.array([2,5,9,13])
#
Xi=rd.uniform(a,b)
print(Xi)
```

```
[ 1.8359202   4.01714411  8.64757891 12.11302644]
```

Ce fonctionnement est identique à celui de la loi normale.

# Annexe A

# Bonus

Sont regroupés dans cette annexe diverses méthodes qui peuvent être utiles dans différents domaines de la physique-chimie ainsi qu'en TIPE. Ces notions ne font pas parties de capacités exigibles des programmes de physique-chimie.

## I Lire et écrire dans un fichier

La lecture et l'écriture dans un fichier font parties des notions du programme d'informatique de première année. Les commandes décrites dans cette partie ont donc pour la plupart été vues en TP d'informatique. En plus du fonctionnement de ces commandes, des applications sont proposées en lien avec la physique.

Avant d'entrer dans les détails, citons deux exemples où l'écriture et la lecture de données dans un fichier peut d'avérer utile

- Considérons un programme python réalisant une simulation numérique ; plutôt que de relancer la simulation numérique à chaque fois qu'on veut exploiter les données issues de la simulation, il est préférable de sauvegarder les résultats numériques dans un fichier ; cela permet par exemple une utilisation des données par un autre programme. C'est d'autant plus pertinent si la simulation nécessite un temps d'exécution relativement long. C'est aussi nécessaire si l'on veut inclure les résultats dans un document LaTeXen réalisant les courbes avec PGF/TikZ plutôt qu'en incluant les images réalisées en python.
- Il peut être nécessaire de traiter en python des données issues d'un capteur (carte d'acquisition, microcontroleur) ; si l'on ne contrôle pas directement le dispositif d'acquisition en python, il faut alors de lire en python des données qui ont été stockées dans un fichier écrit par le dispositif d'acquisition[1].

Les deux opérations, lecture dans un fichier et écriture dans un fichier, sont complémentaires.

### 1 Écrire dans un fichier

#### a. Pour commencer

La sythaxe permettant d'écrire dans un fichier est la suivante :

```
1 monfichier=open('fichier.txt','w')
2 monfichier.write("Bonjour")
3 monfichier.close()
```

Expliquons en détail le fonctionnement de ce bref programme :

- à la ligne 1, un objet python appelé monfichier est créé, il est associé au fichier fichier.txt qui est créé sur le disque dur de l'ordinateur et qui apparaît dans le répertoire personnel de l'utilisateur ; le paramètre 'w' signifie que le fichier est ouvert en écriture (write),
- à la ligne 2, on trouve la commande d'écriture dans le fichier,
- à la ligne 3, on termine l'opération ; c'est seulement au terme de l'exécution de cette ligne que le texte est effectivement écrit dans le fichier sur le disque dur de l'ordinateur.

[1] On réalise par exemple une acquisition avec une carte d'acquisition SYSAM-SP5 pilotée par LatisPro, on exporte les données dans un fichier depuis LatisPro, et on les traite ensuite en python.

Afin d'illustrer le fonctionnement de ces commandes, exécuter seulement les deux premières lignes : un fichier apparaît dans le répertoire de l'utilisateur (vérifier avec le gestionnaire de fichiers) mais il est vide. Exécuter la troisième ligne : le fichier contient effectivement le mot Bonjour.

**Attention ! ! !** Sur les ordinateurs du lycée sous Windows, python peut écrire (et lire) des fichiers dans le répertoire utilisateur, mais l'explorateur de fichier ne peut pas y accéder ! ! ! ! Il faut utiliser le lecteur P, de la façon suivante :

```
1 monfichier=open('P:\\fichier.txt','w')
2 monfichier.write("Bonjour")
3 monfichier.close()
```

Notons l'utilisation du double backslash : dans une chaîne de caractère, le backslash sert à désigner un caractère de contrôle. Par exemple \n désigne un retour à la ligne ; si le nom du fichier commence par un n, la commande open('P :\nono.txt','w') conduira à un message d'erreur. On peut s'en protéger en doublant le backslash, qui est alors lui-même interpréter comme le caractère codant le backslash, ce qui résout le problème. Par précaution, on peut doubler systématiquement le backslash, ce qui évite de se demander comment sera interpréter l'enchaînement du backslash et de la première lettre du nom du fichier.

De plus, sur certaines machines, il semble qu'il soit interdit d'accéder directement à P ; dans ce cas, on écrirt dans le répertoire Documents :

```
1 monfichier=open('P:\Documents\\fichier.txt','w')
2 monfichier.write("Bonjour")
3 monfichier.close()
```

Ce problème est spécifique aux machines sous Windows, tout marche sans problème sous linux.

Allons plus loin, et exécutons le programme suivant :

```
1 monfichier=open('fichier.txt','w')
2 monfichier.write("Sous le pont Mirabeau coule la Seine")
3 monfichier.write("Et nos amours")
4 monfichier.close()
```

On peut écrire autant de lignes qu'on le souhaite avec autant de commandes write. Ouvrir le fichier avec un éditeur de texte : le mot Bonjour a disparu. En effet, en ouvrant le fichier en mode écriture ('w'), on écrase ce que le fichier contenait auparavant. Si on veut ajouter des lignes, on utilise la commande synthaxe suivante :

```
1 monfichier=open('fichier.txt','a')
2 monfichier.write("Faut-il qu'il m'en souvienne\nLa joie venait toujours apres la peine")
3 monfichier.close()
```

L'ouverture du fichier avec l'argument 'a' (comme ajouter) permet d'ajouter des caractères au fichier. Notons également le rôle du caractère spécial \n. Comment peut-on modifier la ligne 2 pour obtenir un retour à la ligne entre le deuxième et le troisième vers ?

Deux remarques :

- les commandes d'écritures (write) sont toujours « encadrées » par une commande open et par une commande close.
- attention aux caractères accentués qui peuvent poser des problèmes d'encodage.

### b. Écrire des données numériques

Écrivons maintenant des données numériques :

```
1 import numpy as np
2 #
3 x=np.linspace(0,2,5)
```

```
4 y=x**2
5 #
6 mesdonnees=open('data.txt','w')
7 for i in range(len(x)):
8     mesdonnees.write(str(x[i])+' '+str(y[i])+'\n')
9 mesdonnees.close()
```

Le fichier data.txt contient

```
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
```

Chaque ligne contient une valeur de $x$ et une valeur de $y$ (qui est le carré de la valeur de $x$ correspondante). Notons la structure de la ligne 8 : la commande write prend comme argument une chaîne de caractères, il faut donc convertir les valeurs numériques (de type flottant) en chaînes de caractères ; on utilise de plus la concaténation, on insère une espace et on n'oublie pas le caractère de retour à la ligne à la fin.

### c. Écrire dans un fichier situé dans un autre répertoire que le répertoire utilisateur

Supposons que le répertoire utilisateur contienne un répertoire travail ; pour écrire dans le répertoire travail, on ouvrira sous linux le fichier de la façon suivante

```
1 monfichier=open('travail/fichier.txt','w')
```

On peut également donner le chemin absolu :

```
1 monfichier=open('/home/username/travail/fichier.txt','w')
```

où username doit être remplacé par le nom d'utilisateur de l'utilisateur.

L'équivalent sous Windows est

```
1 monfichier=open('travail\fichier.txt','w')
```

On peut également donner le chemin absolu :

```
1 monfichier=open('C:\Utilisateurs\username\travail\fichier.txt','w')
```

On peut se faciliter la vie en créant une variable contenant le chemin d'accès au répertoire, ce qui est pratique s'il est nécessaire d'ouvrir plusieurs fois un même fichier ou plusieurs fichiers dans un même répertoire

```
1 repert='/home/username/travail/' # exemple sous linux
2 monfichier=open(repert+'fichier.txt','w')
```

### d. La bibliothèque os

Il n'est pas toujours évident de manipuler les chemins d'accès aux différents répertoires, surtout quand on se retrouve sur un ordinateur dont le système d'exploitation n'est pas celui qu'on connaît le mieux. Par ailleurs, il faut *a priori* modifier le programme python à chaque fois que l'on change d'ordinateur. Dans ce cas, la bibliothèque os est un grand secours : elle permet de nombreuses manipulations sur le système de fichier indépendamment du système d'exploitation.

Ainsi, on peut par exemple obtenir le nom du répertoire courant avec la commande

```
import os
rep=os.getcwd()
print(rep)
```

*A priori*, celui-ci correspond au répertoire utilisateur. On peut créer un répertoire avec la commande os.mkdir, puis se déplacer dans ce répertoire avec la commande os.chdir

```
os.mkdir('travail')
os.chdir('travail')
```

Attention, il ne doit pas exister de répertoire dénommé travail avant de créer ce répertoire. Afin de vérifier que nous sommes dans le bon répertoire, il suffit de taper dans le shell os.getcwd().

Le programme suivant créé un répertoire dénommé travail2 dans le répertoire courant de l'utilisateur et y crée un fichier essai.txt :

```
import os
os.mkdir('travail2')
os.chdir('travail2')
monfichier=open('essai.txt','w')
monfichier.write('Bonjour\n')
monfichier.close()
```

Le chemin d'accès au fichier peut-être obtenu de la façon suviante :

```
rep=os.getcwd()
acces_fichier=rep+'essai.txt'
print(rep)
print(acces_fichier)
```

Le code précédant fonctionne quelque soit le système d'exploitation ! Pour une première introduction à la bibliothèque os, on peut consulter le lien suivant : https://pythonforge.com/module-os-systeme-dexploitation/

## 2 Lire dans un fichier

### a. Localiser le fichier que l'on veut lire

Comme dans le cas de l'écriture dans un fichier, il faudra indiquer le chemin le chemin d'accès du fichier que l'on veut lire. Sous Windows, si on n'utilise pas la bibliothèque os, on peut procéder de la façon suivante :

- utiliser l'explorateur de fichier pour trouver le fichier que l'on souhaite ouvrir,
- faire un clic droit sur le fichier, et demander à voir ses propriétés,
- le chemin (absolu) est indiqué, il suffit de le copier-coller dans la commande d'ouverture du programme python (en faisant attention à l'interprétation des backslash).

### b. Principe de la lecture séquentielle

En python, la lecture dans un fichier est séquentielle : on lit à partir du début du fichier, toujours en avançant, caractère par caractère, sans sauter de caractère. Contrairement à un livre, on ne peut ni revenir en arrière, ni sauter un passage qui ne nous intéresse pas : on lit forcément « vers l'avant ». On peut en revanche fermer un fichier que l'on a commencé à lire : en l'ouvrant à nouveau, on repart du début (il n'existe pas de « marque-page »).

Supposons que l'on dispose du fichier poeme.txt qui contient

```
Sous le pont Mirabeau coule la Seine
Et nos amours
Faut-il qu'il m'en souvienne
La joie venait toujours apres la peine
```

Exercice préliminaire : écrire un programme python qui crée ce fichier.

```
apollinaire=open('poeme.txt','w') # attention au repertoire de travail
apollinaire.write("Sous le Pont Mirabeau coule la Seine\nEt nos amours\n")
apollinaire.write("Faut-il qu'il m'en souvienne\n")
apollinaire.write("La joie venait toujours apres la peine")
apollinaire.close()
```

Nous allons maintenant lire ce fichier ; pour cela, il faut ouvrir le fichier en lecture (avec le paramètre 'r' comme « read ») :

```
guillaume=open('poeme.txt','r') # ouverture du fichier en mode lecture
x=guillaume.read() # lecture et affectation
guillaume.close() # on ferme le fichier
print(type(x)) # affichage du type de x
print(x) # affichage du contenu de x
```

Dans le shell, on obtient

```
<class 'str'>
Sous le Pont Mirabeau coule la Seine
Et nos amours
Faut-il qu'il m'en souvienne
La joie venait toujours apres la peine
```

Ce qui montre que la commande de lecture renvoie une chaîne de caractères.

Testons le code suivant

```
alcools=open('poeme.txt','r')
x=alcools.read(20)
y=alcools.read(25)
z=alcools.read()
alcools.close()
print(x)
print(y)
print(z)
```

Dans le shell, on obtient

```
Sous le Pont Mirabea
u coule la Seine
Et nos a
mours
Faut-il qu'il m'en souvienne
La joie venait toujours apres la peine
```

La commande de lecture de la ligne 2 lit les 20 premiers caractères ; la commande de la ligne 3 lit les 25 caractères suivants ; enfin, la commande de la ligne 4 lit jusqu'à la fin du fichier. Vérifier que le nombre de caractères lus est correct (le retour à la ligne \n compte pour un caractère) et que la lecture est bien séquentielle.

### c. Lecture par lignes

Il peut être plus pratique de lire le fichier par ligne ; observons le comportant du code suivant :

```
apo=open('poeme.txt','r')
z=apo.readline()
zz=apo.read(5)
zzz=apo.readline()
```

```
apo.close()
print(z)
print(zz)
print(zzz)
```

qui donne dans le shell

```
Sous le Pont Mirabeau coule la Seine

Et no
s amours
```

La commande de la ligne 2 lit la première ligne (y compris le caractère de retour à la ligne), puis la commande la ligne 3 lit 5 caractères ; enfin, la commande de la ligne 4 lit jusqu'à la fin de la ligne. On écrit parfois abusivement que la commande readline() lit une ligne ; il est plus correct d'écrire que la commande readline() lit depuis la position courante jusqu'à rencontrer un caractère de retour à la ligne.

Enfin, testons

```
apollinaire=open('poeme.txt','r')
x=apollinaire.readlines()
apollinaire.close()
print(x)
```

On obtient

```
['Sous le Pont Mirabeau coule la Seine\n', 'Et nos amours\n', "Faut-il qu'il m'en souvienne\n",
'La joie venait toujours apres la peine']
```

La commande readlines() renvoie donc une liste de chaînes de caractères, chaque chaîne de caractère correspondant à une ligne du fichier. On vérifie le type des objets que l'on manipule

```
print(type(x))
print(type(x[0]))
```

```
<class 'list'>
<class 'str'>
```

## 3 Exemple : récupération des données exportées depuis LatisPro

LatisPro permet d'exporter des données dans un fichier texte ; c'est ce qui a été fait suite à l'étude de la caractéristique d'une diode. Le fichier diode2.txt contient les données exportées ; lors de l'exportation, les paramètres par défaut ont été choisis (avec une virgule comme séparateur décimal et un point virgule comme séparateur de données). Observons les premières lignes du fichier :

```
Temps;EA1;Temps;i_D
-0,0005;-1,51139439642429;-0,0005;-3,83754493668675E-6
-0,0004998;-1,51139439642429;-0,0004998;-3,83754493668675E-6
-0,0004996;-1,51139439642429;-0,0004996;-3,83754493668675E-6
-0,0004994;-1,51636936049908;-0,0004994;1,13741913810372E-6
-0,0004992;-1,51636936049908;-0,0004992;1,13741913810372E-6
```

Sur chaque ligne, on a une valeur du temps, de la tension aux bornes de la diode (la voie EA1 correspond à la tension aux bornes de la diode), à nouveau une valeur du temps et l'intensité du courant traversant la diode (grandeur calculée par LatisPro à partir de la tension aux bornes d'une résistance placée en série avec la diode).

Nous allons illustrer le traitement à effectuer à partir de la deuxième ligne du fichier (la première ne contient pas de donnée).

```
donnees=open('diode2.txt','r')
donnees.readline() # lecture de la première ligne, pas d'affectation
x=donnees.readlines() # lecture et affectation des autres lignes (données)
donnees.close() # fermeture du fichier
y=x[0]
print(y)
```

```
'-0,0005;-1,51139439642429;-0,0005;-3,83754493668675E-6\n'
```

On se débarasse du retour à la ligne en modifiant la ligne 4 en utilisant le tranchage

```
donnees=open('diode2.txt','r')
donnees.readline() # lecture de la première ligne, pas d'affectation
x=donnees.readlines() # lecture et affectation des autres lignes (données)
donnees.close() # fermeture du fichier
y=x[0][:-1]
print(y)
```

```
-0,0005;-1,51139439642429;-0,0005;-3,83754493668675E-6
```

Changeons le séparateur décimal en remplaçant les virgules par des points[2] :

```
yy=y.replace(',','.')
print(yy)
```

```
-0.0005;-1.51139439642429;-0.0005;-3.83754493668675E-6
```

Découpons ensuite la chaîne de caractères :

```
yyy=yy.split(';')
print(yyy)
```

```
['-0.0005', '-1.51139439642429', '-0.0005', '-3.83754493668675E-6']
```

On obtient ainsi une liste de chaînes de caractères ; il suffit d'en récupérer les différents éléments et de les convertir en flotant ; par exemple

```
z=float(yyy[-1])
print(z)
```

```
-3.83754493668675e-6
```

Il ne reste plus qu'à ranger cette valeur dans une liste qui contiendra toutes les valeurs successsives de l'intensité du courant.

Voici finalement le code permettant d'obtenir trois listes contenant respectivement les valeurs du temps, de la tension au bornes de la diode et de l'intensité qui la traverse :

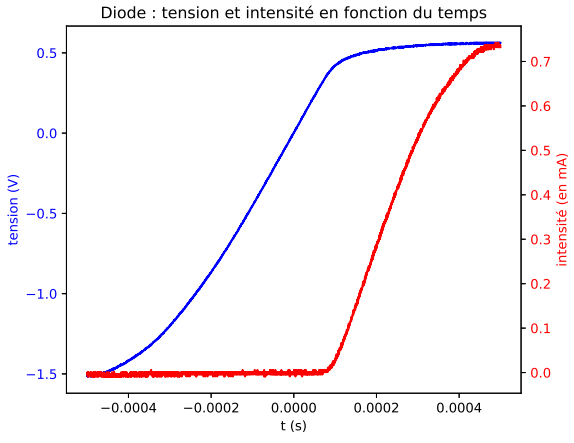
```
donnees=open('diode2.txt','r')
donnees.readline() # lecture de la première ligne, pas d'affectation
x=donnees.readlines() # lecture et affectation des autres lignes (données)
donnees.close() # fermeture du fichier
#
Lt,Lu,Li=[],[],[] # création de listes vides pour stocker les données
```

[2] On aurait pu s'épargner cette étape lors de l'exportation : LatisPro permet à l'utilisateur de choisir le séparateur décimal.

```
#
for y in x: # on parcourt toutes les lignes du fichier
    yy=y[:-1].replace(',','.').split(';') # on effectue les opérations précédentes en une ligne
    Lt.append(float(yy[0])) # on range la première valeur dans la liste des temps
    Lu.append(float(yy[1])) # on range la deuxième valeur dans la liste des tensions
    Li.append(float(yy[-1])) # on range la dernière valeur dans la liste des intensités
lt=np.array(Lt) # on transforme les listes...
lu=np.array(Lu) # ...en tableaux numpy...
li=np.array(Li) # ...(facultatif)...
li=li*1000 # on convertit l'intensité en mA
```

Il ne reste plus qu'à faire les tracés : un premier tracé avec la tension et l'intensité en fonction du temps, avec deux axes différents pour les ordonnées[3] :

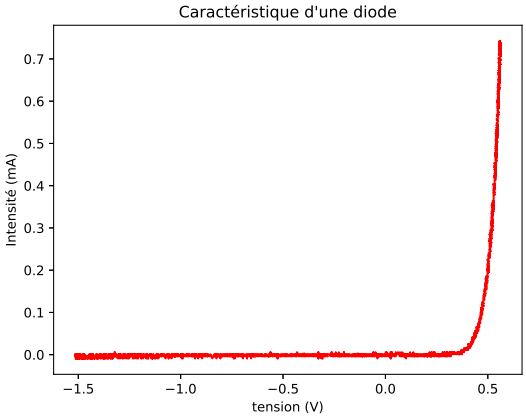
```
fig, ax1 = plt.subplots()
ax1.plot(lt, lu, 'b-')
ax1.set_xlabel('t (s)')
ax1.set_ylabel('tension (V)', color='b')
for tl in ax1.get_yticklabels():
    tl.set_color('b')
#
ax2 = ax1.twinx()
ax2.plot(lt, li, 'r-')
ax2.set_ylabel('intensité (en mA)', color='r')
for tl in ax2.get_yticklabels():
    tl.set_color('r')
fig.tight_layout(pad=1.6) #pour ne pas rogner le titre
#
plt.title("Diode : tension et intensité en fonction du temps")
plt.show()
```




Puis la caractéristique :

```
plt.figure()
plt.plot(lu,li,'r')
plt.xlabel('tension (V)')
plt.ylabel('Intensité (mA)')
plt.title("Caractéristique d'une diode")
plt.show()
```

[3] La synthaxe est peu compréhensible à la première utilisation !




## 4 Autre exemple : exploitation de données exportées depuis Avimeca

Le fichier roue.txt contient le résultat du pointage du mouvement d'une roue. Observons les premières lignes du fichier[4] :

```
roue
t;x;y
(s);(m);(m)
0,125;1,18E+0;1,75E+0
0,167;9,65E-1;1,75E+0
0,208;7,86E-1;1,70E+0
0,250;6,08E-1;1,61E+0
```

Les trois premières lignes du fichier ne contiennent pas de données. Pour les lignes suivantes, il faudra (comme dans l'exemple précédent) :

- ne pas prendre en compte le caractère de contrôle de retour à la ligne (\n, qui n'est pas visible),
- remplacer les virgules par des points (séparateur décimal),
- couper chaque ligne au niveau des points-virgules,
- convertir la chaîne de caractère obtenue en flottant.

Le code suivant conduit au résultat :

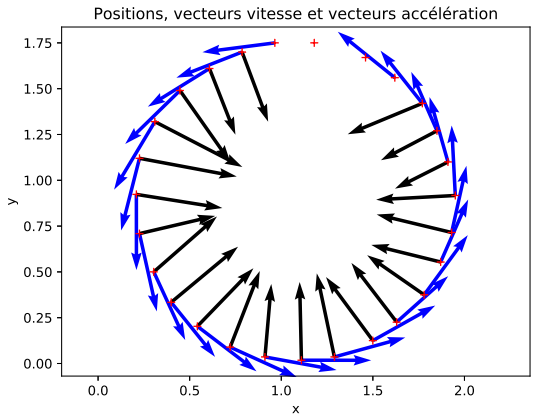
```
import os
import numpy as np
import matplotlib.pyplot as plt
#
os.chdir('') # à compléter avec le chemin d'accès au répertoire de la vidéo
#
## Lecture du fichier
donnees=open('roue.txt','r')
for i in range(3):
    donnees.readline() # on lit les 3 premières lignes sans les affecter
data=donnees.readlines() # on lit les lignes suivantes
donnees.close()
#
## Récupération des données
Lt,Lx,Ly=[],[],[] # listes vides qui contiendront t, x et y
for d in data: # on parcourt les données
    L=d[:-1].replace(',','.').split(';') # récupération sous forme d'une liste
    Lt.append(float(L[0])) # on stocke le temps dans Lt
    Lx.append(float(L[1])) # on stocke la valeur de x dans Lx
```

[4] fourni par Mme Cochennec.

```
    Ly.append(float(L[2])) # on stocke la valeur de y dans Ly
N=len(Lt)
#
## Calcul de la vitesse (moyenne de la dérivée à droite  et de la dérivée à gauche)
Lvx,Lvy=[0]*N,[0]*N # on travaille avec des listes ayant toutes la même longueur
for i in range(1,N-1):
    Lvx[i]=((Lx[i+1]-Lx[i])/(Lt[i+1]-Lt[i])+(Lx[i]-Lx[i-1])/(Lt[i]-Lt[i-1]))/2
    Lvy[i]=((Ly[i+1]-Ly[i])/(Lt[i+1]-Lt[i])+(Ly[i]-Ly[i-1])/(Lt[i]-Lt[i-1]))/2
#
## Calcul de l'accélération
Lax,Lay=[0]*N,[0]*N
for i in range(2,N-2):
    Lax[i]=((Lvx[i+1]-Lvx[i])/(Lt[i+1]-Lt[i])+(Lvx[i]-Lvx[i-1])/(Lt[i]-Lt[i-1]))/2
    Lay[i]=((Lvy[i+1]-Lvy[i])/(Lt[i+1]-Lt[i])+(Lvy[i]-Lvy[i-1])/(Lt[i]-Lt[i-1]))/2
#
## Tracés
plt.figure()
plt.plot(Lx,Ly,'r+',linestyle='None')
for i in range(1,N-1):
    plt.quiver(Lx[i],Ly[i],Lvx[i],Lvy[i],scale_units='xy',angles='xy',scale=12,color='blue')
for i in range(2,N-2):
    plt.quiver(Lx[i],Ly[i],Lax[i],Lay[i],scale_units='xy',angles='xy',scale=60,color='black')
plt.title('Positions, vecteurs vitesse et vecteurs accélération')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.show()
```

Notons qu'on ne peut pas calculer la vitesse sur le premier et le dernier point. De même, on ne peut pas calculer l'accélération sur les deux premiers points et les deux derniers points.

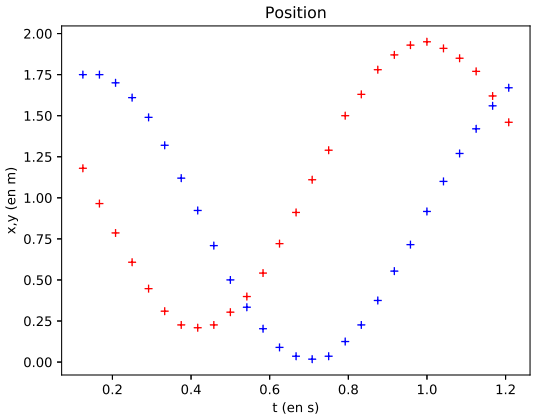
Le pointage conduit à des incertitudes pour les positions successives ; le calcul de la vitesse puis celui de l'accélération va amplifier ces incertitudes[5] : on constate en particulier que le vecteur accélération n'est pas parfaitement centripète.

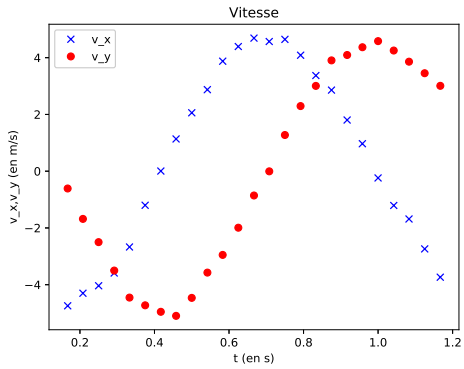


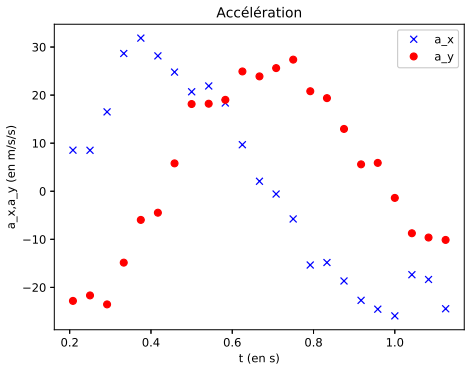

```
## Tracés complémentaires
plt.figure()
plt.plot(Lt,Lx,'r+',linestyle='None')
plt.plot(Lt,Ly,'b+',linestyle='None')
plt.xlabel('t (en s)')
plt.ylabel('x,y (en m)')
```

[5] Dériver revient à appliquer un filtre passe-haut qui amplifie le bruit.

```
plt.title('Position')
plt.show()
#
plt.figure()
plt.plot(Lt[1:-1], Lvx[1:-1], 'bx', linestyle='None',label='v_x')
plt.plot(Lt[1:-1], Lvy[1:-1], 'ro', linestyle='None',label='v_y')
plt.legend()
plt.xlabel('t (en s)')
plt.ylabel('v_x,v_y (en m/s)')
plt.title('Vitesse')
plt.show()
#
plt.figure()
plt.plot(Lt[2:-2], Lax[2:-2], 'bx', linestyle='None',label='a_x')
plt.plot(Lt[2:-2], Lay[2:-2], 'ro', linestyle='None',label='a_y')
plt.legend()
plt.xlabel('t (en s)')
plt.ylabel('a_x,a_y (en m/s/s)')
plt.title('Accélération')
plt.show()
```




# II Animations en python

Réaliser une animation permet de visualiser des phénomènes physiques ou chimiques qui dépendent du temps. On peut s'en passer : en mécanique, on peut par exemple reprtésenter la trajectoire d'un point matériel qui correspond à l'ensemble des positions successives d'un système ; mais la seule connaissance de la trajectoire ne permet pas d'appréhender l'évolution temporelle (vitesse constante ou pas, par exemple). C'est la raison pour laquelle réaliser une animation peut s'avérer très démonstratif.

## 1 Faire une animation en python

Pour représenter l'évolution d'une grandeur physique correspondant à une onde progressive sinusoïdale

$$y(x,t) = y_0 \cos(\omega\, t - k\, x)$$

on peut utiliser le code suivant[6] :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib import animation
4  #
5  T,lbd=1,1 # période et longeur d'onde
6  k = 2*np.pi/lbd # pulsation spatiale
7  w = 2*np.pi/T # pulsation
8  dt = 0.01 # durée entre deux images
9  #
10 xmin,xmax,Nx = 0,3,100 # valeurs de x (min, max, nb de points)
11 x = np.linspace(xmin, xmax, Nx)
12 #
13 fig=plt.figure() # initialisation de la figure
14 line, = plt.plot([],[])
15 plt.xlim(xmin, xmax)
16 plt.ylim(-1,1)
17 #
18 def animate(i):
19     t = i * dt
20     y = np.cos(k*x - w*t)
21     line.set_data(x, y)
22     return line,
23 #
24 ani = animation.FuncAnimation(fig, animate, frames=100, interval=20, repeat=True)
25 #
26 plt.show()
```

On reconnaît notamment l'expression de la grandeur physique qui se propage à la ligne 20. On peut bien-sûr améliorer l'animation obtenue en ajoutant un titre etc. Si l'on souhaite modifier la forme de l'onde, il suffit de modifier la ligne 20 (ainsi que les paramètres des lignes 5 à 8).

## 2 Exporter des images en python et fabriquer une animation avec ffmpeg

Une autre stratégie pour réaliser une animation est de réaliser l'opération en deux étapes :

- en python : fabriquer les images qui constitueront l'animation et les exporter,
- avec un outil adapté (par exemple ffmpeg), assembler les images pour produire une vidéo.

Cette méthode présente plusieurs intérêts :

- les calculs nécessaires à la réalisation de l'animation ne sont effectués qu'une fois et pas à chaque visionnage,
- lors de la production de la vidéo à partir des images, on peut fixer librement de nombreux paramètres (choix du codec, nombre d'images par seconde...),
- une fois que la vidéo a été réalisée, on peut la lire avec n'importe quel lecteur (vlc etc.), même si python n'est pas installé sur l'ordinateur,
- on peut utiliser toutes les fonctionnalités du lecteur vidéo (ralenti, arrêt sur image, lecture en boucle...).

Voici le programme python

[6] La synthaxe est un peu cryptique en première lecture.

```python
import numpy as np
import matplotlib.pyplot as plt
#
# Caractéristique de l'onde
#
def f(x,t): # onde dont on veut représenter la propagation
    return np.cos(k*x - w*t)
#
T,lbd=1,1 # période et longeur d'onde
k = 2*np.pi/lbd # pulsation spatiale
w = 2*np.pi/T # pulsation
#
# Paramètres spaciaux
#
xmin,xmax,Nx = 0,3,100 # valeurs de x (min, max, nb de points)
x = np.linspace(xmin, xmax, Nx)
#
# Paramètres temporels
#
tmin,tmax,Nt=0,3,200 # Nt correspond au nombre d'images
Deltat=(tmax-tmin)/Nt # pas de temps
#
# Paramètres de sauvegarde des images
#
chemin='MesImages/' # repertoire ou sont stokées les images
fichier='image_' # nom generique de l'image
#
#  Fabrication et exportation des images
#
plt.figure() # initialisation de la figure
for i in range(Nt): # boucle fabriquant sucessivement les différentes images
    t=tmin+i*Deltat # calcul du temps
    plt.xlim(xmin, xmax) # cosmétique de l'image
    plt.ylim(-1,1)
    plt.title("Propagation d'une onde progressive sinusoïdale")
    plt.xlabel("x")
    plt.ylabel("amplitude de l'onde")
    y=[] # on créé une liste vide pour stocker les valeurs de y
    for xv in x: # on parcours toutes les abscisses
        y.append(f(xv,t)) # on range dans la liste y l'amplitude de l'onde en (x,t)
    y=np.array(y) # on transforme la liste en tableau numpy (facultatif)
    plt.plot(x,y) # on fabrique l'image correspondant à l'onde à un instant donné
    plt.savefig(chemin+fichier+'0'*(len(str(Nt))-len(str(i+1)))+str(i+1)) #sauvegarde de l'image
    # numerotation 001, 002, 003...
    plt.clf() # on efface l'image avant de recommencer à l'instant suivant
```

**Attention !** Il faut avoir créé préalablement le répertoire MesImages (au bon endroit) ; il est recommandé de donner le chemin de façon absolue pour éviter tout problème. Le programme n'affiche rien, les images sont crées dans le répertoire MesImages.

Il reste à « assembler » les images pour fabriquer la vidéo ; on peut réaliser cette opération avec ffmpeg, en tapant dans un terminal

```
ffmpeg -f image2 -i image_%03d.png -r 24 -vcodec mpeg4 -b 15000k mavideo.mp4
```

Expliquons le fonctionnement de cette commande :

- ffmpeg est le nom de la commande,
- le paramètre -f image2 signifie que la vidéo sera fabriquée à partir d'un ensemble d'images,

- le paramètre -i image_%03d.png indique que les fichiers images sont nommés image_001.png, image_002.png etc ; en particulier, %03d précise que les images sont numérotées avec 3 chiffres,
- le paramètre -r 24 précise que l'encodage se fait à 24 images par seconde ,
- le paramètre -vcodec mpeg4 précise le choix du codec d'encodage de l'image,
- le paramètre -b 15000k est lié à la qualité (et donc à la taille) du fichier vidéo obtenu (il s'agit du bitrate ou débit binaire ; plus il est élevé, meilleure est la qualité et plus volumineux est le fichier vidéo),
- mavideo.mp4 est le nom donnée à la vidéo.