LA PROGRAMMATION PYTHON EN PHYSIQUE

Lycée Saint-Exupéry de Mantes-la-Jolie

2019

En surfant sur le net, à la recherche d'informations sur le nouveau programme de seconde en physique et plus particulièrement le tracé de vecteurs en langage de programmation Python, je suis tombé sur un document d'une journée de formation écrit en LaTeX que j'ai trouvé très réussi visuellement. Pour ceux qui ne connaissent pas LaTeX, c'est une alternative à tous les *WYSIWYG* (What You See Is What You Get) que vous connaissez, les plus connus étant LibreOffice Writer et Microsoft Word. C'est beaucoup plus austère mais tellement plus adapté quand on enseigne les mathématiques ou la physique! Bref, j'ai voulu essayer de reproduire la mise en page du document.
Ensuite, je ne trouvais pas mon bonheur sur la problématique des tracés de vecteurs, seulement quelques réponses assez succinctes dans des forums spécialisés. J'ai donc combiné mon envie de réécrire en LaTeX avec mon besoin de maîtriser le tracé de vecteurs vitesse et accélération. Et si, au final, cela peut aider certains d'entre vous qui vont se mettre à *pythonner*, contraints et forcés, alors ce travail n'aura pas été vain.

Table des matières

Chapitre I

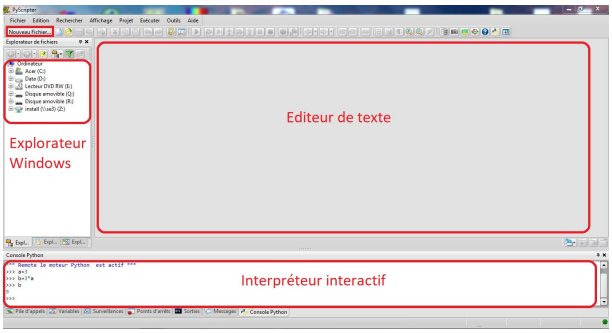
Avant de programmer

1 La génèse

En 1989, un programmeur, **Guido Van Rossum**, travaille à l'Institut national de recherches mathématiques et informatiques des Pays-Bas (CWI), sur un projet de développement d'un système d'exploitation. Afin d'utiliser au mieux ce nouveau système, il perfectionne le langage ABC qui était alors utilisé. Il rend également ce nouveau langage portable, c'est-à-dire utilisable sur plusieurs système d'exploitation. Il appellera sa création Python, en l'honneur des Monty Python dont il appréciait l'humour. Une des particularités de ce langage est la syntaxe par indentation. Ce langage est abordable par des élèves lycéens car il reste intuitif et moins austère que d'autres langages.

2 Faites votre choix

Si vous êtes adepte de l'OS de Microsoft, Windows, il existe de nombreuses distributions. La plus simple dans son installation et dans son utilisation reste, pour moi, Edupython, initiée par Vincent Maille et l'académie d'Amiens. Cette distribution contient toutes les bibliothèques nécessaires pour être utilisé dans l'éducation nationale. Son interface est en français et il y a aussi un module *lycee* spécifique. De plus, cette distribution est portable, elle peut se mettre sur une clé USB pour l'avoir toujours sur soi. Faisons une présentation rapide d'Edupython.



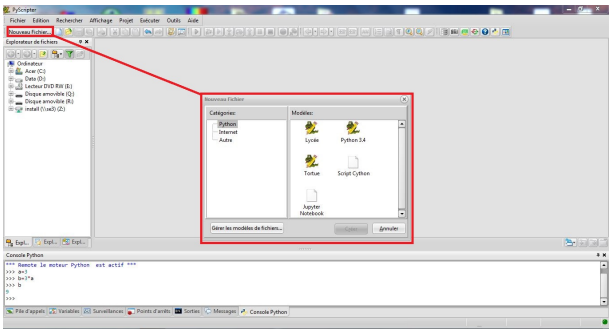
Voici à quoi ressemble Edupython. Vous remarquerez qu'il existe trois zones. Une qui permet de retrouver facilement vos programmes, **l'explorateur Windows**, une autre qui permet de rédiger vos programmes, **l'éditeur de texte** et une dernière, **la console**, qui est un **interpréteur interactif**. C'est-à-dire que cela permet d'avoir une interaction immédiate entre l'utilisateur et l'interpréteur. Son fonctionnement rappelle en quelque sorte celui d'une calculatrice améliorée. On rentre les instructions dans la console, celles-ci sont exécutées par l'interpréteur et le résultat s'affiche sur la ligne suivante. Utilisons-la pour votre baptême de programmeur. Classiquement, on doit écrire « Hello World ! ». Les trois chevrons indiquent que la console attend nos instructions.

```
>>> print("Hello World !")
```

On exécute l'instruction avec la touche « Entrée » . Voici ce que vous devez obtenir :

```
>>> print("Hello World !")
Hello World !
>>>
```

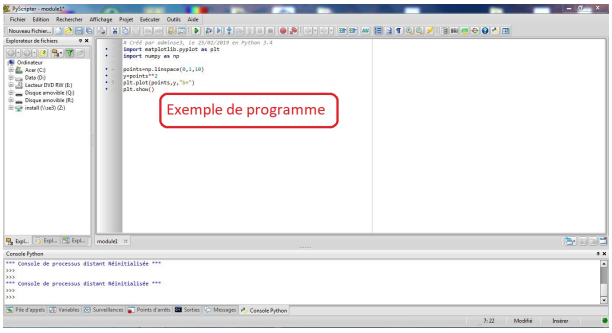
L'interpréteur interactif est pratique mais d'un intérêt limité car on ne peut pas sauvegarder ce que l'on écrit d'où l'utilité de l'éditeur de texte. Ouvrez un nouveau fichier Python.



Lorsque l'on clique sur « *nouveau fichier* », on peut choisir entre 3 sortes de programmes différents, il y a :

- Lycée : si on veut utiliser le module « lycee ».
- Python 3.4 : pour programmer en python 3.4 jusqu'à une prochaine évolution .
- Tortue : pour utiliser le module « tortue ».

Le module Tortue est un module de dessin. Il est intéressant et amusant pour des élèves de seconde en première approche de la programmation pour faire des fractales par exemple. Pour l'instant, on ouvrira seulement un nouveau fichier Python. Voici un exemple de programme que vous serez bientôt capable de faire.



Quand on préfère Linux, comme moi, je conseillerais la suite Anaconda qui intègre Spyder. L'installation se fait facilement. Elle existe aussi pour Windows. Cela reste, au final, une question de goût et de besoin.

Chapitre II

Les notions de base

1 Les variables

Les variables sont l'un des concepts qui se retrouvent dans la totalité des langages de programmation. Une variable est une donnée de votre programme stockée dans la mémoire de l'ordinateur afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs. En Python, pour affecter une valeur à une variable, il suffit d'écrire quelque chose comme :

nom_de_variable = valeur

Une variable doit respecter quelques règles de syntaxe :

1. Le nom de la variable ne peut être composé que de lettres majuscules, minuscules, de chiffres et du symbole *underscore*.
2. Le nom de la variable ne peut pas commencer par un chiffre.
3. Le langage Python est sensible à la casse, AGE est différent de age ou aGe.

⚠ Il existe des noms interdits pour les variables car ils sont utilisés dans le langage Python :
and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, if, import, in, is, global, lambda, None, nonlocal, not, or, pass, return, rise, True, try, while, with, yield.

On peut affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première.

```
1 mon_age=20
2 Mon_age=25
3 somme=mon_age+Mon_age
4 print(mon_age,Mon_age,somme)
```

Le langage Python a des fonctions prédéfinies dont la fonction **print()** qui permet d'afficher une ou plusieurs données.

La fonction **print** permet, ici, d'afficher les valeurs des trois variables utilisées.

On est souvent amené à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'un certain nombre.

Il y a deux moyens pour le faire :

```
1 mon_age=mon_age+3
```

ou plus subtil

```
1 mon_age+=3
```

Notez qu'il existe une méthode simple pour permuter deux variables sans passer par une troisième.

```
1 a,b=5,9 # affectation parallèle
2 c=d=2   # affectation multiple
3 print(a,b,c,d)
4 a,b=b,a
5 print(a,b,c,d)
```

Les variables que nous venons de voir sont des entiers mais ce n'est pas toujours le cas, cela peut tout aussi bien être une chaîne de caractères :

```
1 phrase="Le langage Python mérite d'être connu!"
2 print(phrase)
```

Dans le langage C++, nous devons définir le type de variable avant de l'utiliser (entier, chaîne de caractères...) or, avec Python, cela se fait automatiquement. La fonction **type()** qui retourne le type de données de l'objet indiqué entre les parenthèses est une autre fonction prédéfinie. Essayez :

```
1 a=3
2 print(type(a))
3 # ou
4 b="Hello"
5 print(type(b))
```

Voyons plus en détail les principaux types de données.

2 Principaux types de données

2.1 Les données numériques

2.1.1 Le type integer

Le type **int** ou **integer** concerne les nombres entiers. Le langage Python est capable de traiter des nombres entiers de taille illimitée. Il faut quand même apporter une petite précision. Le cœur d'un ordinateur est constitué d'un processeur qui ne peut que traiter des nombres binaires de taille limitée (32 ou 64 bits sur les ordinateurs récents). Sur un ordinateur n bits, les nombres relatifs qu'il est possible d'encoder sont compris entre -2^{n-1} et $2^{n-1}-1$. On représente un entier relatif x positif ou nul comme l'entier naturel x et un entier relatif x strictement négatif comme l'entier naturel $x+2^n$. Ainsi, les entiers naturels de 0 à $2^{n-1}-1$ servent à représenter les entiers relatifs positifs et les entiers naturels de 2^{n-1} à 2^n-1 servent à représenter les entiers relatifs strictement négatifs. Les opérations sur des nombres compris entre ces deux limites sont donc rapides. Mais lorsque les nombres sont hors de ces limites, les interpréteurs et compilateurs doivent effectuer un travail de codage/décodage afin de ne présenter au processeur que des nombres de n bits au maximum. Le temps de traitement sera alors plus long.

2.1.2 Le type float

C'est ce qu'on appelle le type « nombre à virgule flottante ». Pour qu'une donnée numérique soit du type **float**, il suffit qu'elle contienne un point décimal ou une puissance

de 10. Ce type de variable permet de manipuler de grands nombres positifs ou négatifs. La précision sera au maximum de 12 chiffres significatifs avec les puissances de 10.

```
a=3.14
# ou
b=8e4
print(type(b))
```

Un nombre flottant est représenté, pour l'interpréteur Python, sous la forme $s.m.2^n$ où s est le signe du nombre, n son exposant et m sa mantisse. Le signe est + ou -, l'exposant est un entier relatif et la mantisse est un nombre à virgule en base 2, compris entre 1 inclus et 2 exclu. Par exemple, quand on utilise 64 bits pour représenter un nombre à virgule, on utilise 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse. J'ai choisi de parler de la représentation des flottants, non pas pour vous donner un mal de tête mais pour mettre l'accent sur un détail qui peut être source de problème.

⚠ À savoir

À cause de la base binaire utilisée, il est impossible de représenter exactement la plupart des nombres décimaux, plus précisément tous ceux qui ne s'écrivent pas sous la forme $\frac{k}{2^n}$. Des nombres qui habituellement ne posent pas de problème dans les calculs en mathématique deviennent ainsi une source d'erreurs multiples.

Pour vous en rendre compte, exécutez dans l'interpréteur interactif ce qui suit (** sert à l'exponentiation) :

```
>>>5+2**(-76)-5
```

puis

```
>>>5-5+2**(-76)
```

2.2 Les données alphanumériques

2.2.1 Le type string

Dans la plupart des langages de programmation, pour traiter des caractères alphabétiques comme des mots, des phrases, il existe une structure de données particulière que l'on appelle « chaîne de caractères »(type **string** ou **str**). Pour la définir, on utilise les apostrophes ou les guillemets.

```
mot1='Hello'
#ou
mot2="World"
print(mot1,mot2)
```

On peut avoir besoin d'une chaîne de caractères qui contient apostrophe et guillemets, on utilise alors l'antislash « \ ».

```
mot3="On la surnommait \"casque d'or\"."
mot4='L\'appel de la forêt'
```

Une autre façon pour utiliser des apostrophes et des guillemets dans la chaîne de caractères est de mettre des triples quotes (trois guillemets ou trois apostrophes).

```
mot5="""On la surnommait "casque d'or"."""
mot4='''L'appel de la forêt'''
```

2.2.2 Accès à un caractère

Une chaîne de caractères est considérée comme une séquence. Chaque caractère peut être désigné par sa place dans la chaîne à l'aide d'un index. Pour cela, on utilise le nom de la variable qui contient la chaîne de caractères et on ajoute l'index de la position du caractère qui nous intéresse dans la chaîne, entre crochets.

⚠ À savoir

On commence à compter à partir de 0.

```
TS="abc"
print(TS[1],TS[0],TS[2])
```

2.2.3 Opérations élémentaires

2.2.3.1 La concaténation :

On peut assembler plusieurs chaînes de caractères pour en former de plus grandes. C'est la **concaténation**. Pour se faire, on utilise l'opérateur mathématique +.

```
TS="Je passe"
TL="mon bac."
Terminal=TS+TL
print(Terminal)
```

2.2.3.2 Extraction, transformation :

Voici quelques possibilités à tester.

```
mot="Ha"
mots=mot*10
Mot="Saperlipopette"
part=Mot[2:9] # début inclus fin exclue (slicing)
print(mot,mots,Mot,part)
```

On peut également transformer la chaîne de caractères en majuscule ou en minuscule. Mais pour cela, il faudra mettre en place une syntaxe un peu particulière car on doit considérer la chaîne de caractères comme un objet et utiliser une méthode disponible pour cet objet. Une méthode est une fonction qui est attachée à un objet précis.

```
mot="ForMidaBle"
# Méthodes lower() et upper() appliquées sur l'
    objet mot
min=mot.lower()
maj=mot.upper()
print(mot, min, maj)
```

2.2.3.3 Rechercher, remplacer :

Voici d'autres possibilités qui pourront s'avérer utiles selon les circonstances.

```
phrase="Cette phrase va me servir d'exercice."
nb=phrase.count("e")
nb2=phrase.count("se")
ToF="se" in phrase
ou=phrase.find("va")
phrase2=phrase.replace("se","za")
print(phrase, "\n",nb,"\n",nb2,"\n",ToF,"\n",ou,"\n
    ",phrase2)
```

2.2.3.4 La fonction len() :

Cette fonction est très utile et très utilisée dans ce document, notamment dans les boucles **for** que nous aborderons un peu plus loin. Elle permet de déterminer la longueur de l'élément entre parenthèses.

```
variable="Einstein"
longueur=len(variable)
print(longueur)
```

2.2.3.5 La conversion :

Si la chaîne de caractères représente un nombre, on peut convertir cette chaîne en entier ou en flottant. Essayez :

```
variable="128"
somme=variable+12
```

On ne peut pas additionner une chaîne de caractères qui ressemble à un entier avec un entier. Mais on peut faire :

```
variable="128"
entier=int(variable)
somme=entier+12
print(somme)
```

ou avec un nombre de type **float** :

```
variable="128.68"
flottant=float(variable)
somme=flottant+12
print(somme)
```

Les fonctions **int()** et **float()** transforment une chaîne de caractères en entier ou flottant. On peut aussi faire la conversion inverse avec la fonction **str()**.

```
entier=128
ch=str(entier) # conversion inverse
```

2.2.4 Formatage

Nous allons voir, ici, plusieurs façons d'afficher différents types de variables.

2.2.4.1 La plus simple :

C'est celle que l'on voit beaucoup dans le code des programmeurs débutants.

```
nombre=128
aliments="carottes"
print("J'ai planté",nombre,aliments,'dans mon
    jardin.')
```

2.2.4.2 Avec % :

Voici la première vraie méthode de formatage :

```
nombre=128
aliments="carottes"
print("J'ai planté %d %s dans mon jardin." %(nombre
    ,aliments))
```

%s indique que la variable est de type *str*, %d *int*, %f *float*. Pour les float, la syntaxe peut être différente si besoin puisque l'on peut tronquer la valeur comme ci-dessous :

```
nombre=128
aliments="carottes"
autres="choux"
print("J'ai planté %f %s et %.2f %s dans mon jardin
    ." %(nombre,aliments,nombre,autres))
```

On doit rappeler les variables à chaque fois que l'on en a besoin et respecter l'ordre.

2.2.4.3 Avec format :

Cette méthode est à privilégier si vous n'avez pas encore une version supérieure à Python 3.6.

```
nombre=128
aliments="carottes"
autres="choux"
print("J'ai planté {0} {1} et {0} {2} dans mon
    jardin.".format(nombre,aliments,autres))
```

On peut aussi tronquer les flottants avec :

```
pi=3.14159
print("Pi vaut {0:2f} ou {0:3f} ou mieux {0}.".
    format(pi))
```

2.2.4.4 Avec f-strings :

Depuis Python 3.6. Cela ressemble à *format* en plus simple et cela permet d'utiliser en plus des fonctions à l'intérieur. Notez le **f** au début de la chaîne.

```
prix=12.56
fruit="pommes"
print(f"Les {fruit.upper()} sont à {prix:.1f} € le
    kilo.")
```

2.3 Booléen

Les booléens sont un type un peu particulier dont les valeurs sont particulièrement simples car il n'y en a que deux : **True** ou **False**. C'est le résultat d'expressions mathématiques logiques.

2.3.1 La négation

C'est le **NON** logique qui s'écrit en Python avec le mot-clé **not**. Ainsi l'expression ***not*** B a la valeur *True* si B a la valeur *False*.

2.3.2 La conjonction

C'est le **ET** logique qui s'écrit en Python avec le mot-clé **and**. L'expression B1 **and** B2 renvoie la valeur *True* si et seulement si B1 et B2 ont pour valeur *True*.

2.3.3 La disjonction

C'est le **OU** logique qui s'écrit en Python avec le mot-clé **or**. L'expression B1 **or** B2 renvoie la valeur *False* si et seulement si B1 et B2 ont pour valeur *False*.

2.3.4 Table de vérité

B1	B2	**not** B1	B1 **and** B2	B1 **or** B2
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

2.4 Les listes

2.4.1 Généralités

Une liste est une collection d'éléments séparés par des virgules et encadrés par des crochets. Les éléments qui composent une liste peuvent être de types différents. Comme pour les chaînes de caractères, on peut accéder à un élément précis de la liste en connaissant son index. En effet, l'ordre dans une liste est fixe. La fonction **len()** est également applicable aux listes. On ajoutera que l'on peut enlever un élément d'une liste avec la fonction **del()**. Il faut noter que les listes sont des séquences modifiables. On peut remplacer un élément par un autre.

Devinez ce que donne le programme suivant :

```
ma_liste=["bonjour",38,"Hello",12.5)
longueur=len(ma_liste)
print(longueur,ma_liste[0])
ma_liste[2]="Great"
print(ma_liste)
del(ma_liste[1])
longueur=len(ma_liste)
print(longueur,ma_liste[2])
```

2.4.2 Des méthodes associées.

On peut également ajouter un élément à une liste. Mais pour cela, il faudra utiliser une méthode disponible pour les objets de type **list**. Ici, contrairement aux méthodes sur les chaînes de caractères, on applique la méthode sur l'objet sans la stocker dans une variable. Un exemple sera plus parlant :

```
"""méthode append() appliquée sur l'objet semaine
    avec comme argument "dimanche"."""
semaine=["lundi","mardi","mercredi","jeudi","
    vendredi","samedi"]
semaine.append("dimanche") # pas de var =
print(semaine)
```

La méthode **append** ajoute l'argument en fin de liste. Si on préfère un rang précis, il faudra utiliser la méthode **insert**. Si on veut supprimer la première occurrence d'un élément dans une liste, il faut utiliser la méthode **remove**. Pour supprimer l'élément qui se trouve à un certain rang d'une liste, on utilise la méthode **pop**. La méthode **reverse** permet d'inverser l'ordre d'une liste. Quel sera le résultat du programme suivant ?

```
L=["a",0,1,2,"banane"]
L.insert(2,"pomme") # L'index en premier argument
L.insert(1,4)
L.remove(2) # L'élément en argument
L.pop(3) # L'index en argument
L.reverse() #
print(L)
```

Sans oublier, la méthode **sort** qui permet de trier une liste.

```
L=[18,12,9,7,130,15,9]
L.sort()
print(L)
```

2.4.3 Slicing

Le slicing ou, si vous préférez, la « découpe en tranche », permet de modifier une liste en travaillant par tranche. Elle consiste à indiquer entre crochets les indices correspondant au début et à la fin de la tranche qui nous intéresse. On retrouve les mêmes effets que les méthodes **del** et **append**.

```
L=["autruche","girafe","lion","ours"]
L[2:2]=["panda"]
L[3:3]=["rat","chien"]
print(L)
L[1:3]=[]
print(L)
L[3:]=["radis","banane"] # signifie de 3 jusqu'à
    la fin
print(L)
```

2.4.4 Liste de nombres

En utilisant la fonction **range**(n) qui génère par défaut une séquence de nombres entiers successifs de 0 à n-1 de type **range** et la fonction **list()** qui convertit une séquence en liste, une liste est vite créée. Attardons-nous un peu sur ce nouveau type d'objet car nous le retrouverons plus tard.

```
variable=range(10)
print(type(variable))
print(variable)
print(variable[2])
print(6 in variable)
print(15 in variable)
```

Range est bien un type de variable à part entière. On peut mettre entre parenthèses, un, deux ou trois arguments, **range**(***start***,***stop***,***pas***). L'avantage du type **range** est que l'on crée un objet qui occupera toujours une petite quantité de mémoire quelle que soit sa taille. Seuls les paramètres sont mémorisés.

La fonction **list()** transforme cet objet en liste.

```
L1=list(range(10,18,2))
L2=list(range(10,-10,-2))
print(L1)
print(L2)
```

2.4.5 La copie de liste

On peut vouloir copier une liste pour ensuite faire des opérations dessus sans toucher à l'original. Intuitivement, on ferait :

```
L1=[0,1,2,3,4,5]
L2=L1
print(L1,L2)
```

On pourrait croire jusqu'ici que tout va bien mais ....

```
L1=[0,1,2,3,4,5]
L2=L1
print(L1,L2)
L1.append(6)
print(L1,L2)
```

L1 et L2 renvoie à la même adresse mémoire, toute modification sur L1 se répercute sur L2 et réciproquement. Pour créer une nouvelle liste L2 indépendante, on peut faire comme cela :

```
L1=[0,1,2,3,4,5]
L2=list(L1)
print(L1,L2)
L1.append(6)
print(L1,L2)
```

2.4.6 Avec des chaînes de caractères

Deux méthodes associées aux chaînes de caractères vont être très utiles lorsque l'on manipule les fichiers par exemple. J'en parle ici parce que cela fait aussi intervenir la notion de liste. En effet, il faut en argument un objet qui doit être itérable, comme une liste ou un tuple (une liste non modifiable) et qui ne doit contenir que des chaînes de caractères. La première méthode est **join** qui renvoie une chaîne de caractères. La deuxième méthode, **split**, est l'inverse en quelque sorte.

Un exemple est toujours plus parlant, notamment pour la syntaxe. Regardez le résultat de ceci :

```
L=["un","deux","trois"]
var="et".join(L)
print(var)
autre="ou".join(L)
print(autre)
Liste=autre.split("e")
# e est notre séparateur d'éléments pour créer la
    liste
print(Liste)
```

3 Les opérateurs spécifiques

Il faut savoir que les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. On ajoute l'existence de l'opérateur de division entière « // », celle de l'opérateur « ** » pour l'exponentiation ou celle de l'opérateur modulo « % » qui donne le reste de la division entière. Un exemple permettra de bien comprendre les différents opérateurs.

```
a,b,c,d=3,8,27,84
e=a+b
f=c-b
g=a*b
h=c/a
i=d/b
j=b**a
k=d//b
l=d%b
print(e,f,g,h,i,j,k,l)
```

Il existe également des opérateurs de comparaison, utiles pour écrire des conditions.

```
x==y # x est égal à y, différent de l'affectation
x!=y # x est différent de y
x>y # x supérieur à y
x<y # x inférieur à y
x>=y # x supérieur ou égal à y
x<=y # x inférieur ou égal à y
```

4 La structure conditionnelle

Cette structure très usitée possèdent des mots-clés particuliers qui permettent de l'identifier immédiatement : **if, elif, else**. Cela se traduit respectivement par si, sinon si, sinon .

Elle se présente sous la forme d'un bloc dans lequel on remarque une indentation des instructions.

```
instruction 1
if condition A: # Remarquez les :
  instruction 2 # indentation
  instruction 3 # indentation
else: # pas de condition mais :
  instruction 4 # indentation
instruction 5
```

Après avoir exécuter l'instruction 1, si la condition A est vraie, l'instruction 2 puis l'instruction 3 sont exécutées et enfin on passe à l'instruction 5. Par contre, si la condition A est fausse, on exécute l'instruction 4 puis on passe à l'instruction 5. Les instructions 2, 3, 4 contenues dans **if** et dans **else** ne peuvent être exécutées successivement. La partie **else** est facultative. On peut aussi écrire **if not** à la place de **if** dans le programme ci-dessus. Les instructions 2 et 3 seront alors exécutées quand la condition A est fausse. On pourra aussi avoir besoin de l'instruction **elif**, que l'on peut traduire par **sinon si**, pour introduire une nouvelle condition. On peut ajouter autant de **elif** que l'on souhaite mais l'instruction **else** ne pourra figurer qu'une fois, clôturant le bloc de la condition.

Voici un exemple complet de la structure possible :

```
instruction 1
if condition 1: # Remarquez les :
  instruction 2 # indentation
  instruction 3 # indentation
elif condition 2:
  instruction 4
elif condition 3:
  instruction 5
else: # pas de condition mais :
  instruction 6 # indentation
instruction 7
```

5 La boucle

5.1 While

Il est souvent nécessaire de répéter plusieurs fois une même séquence d'instructions jusqu'à ce qu'une certaine condition soit satisfaite. Pour se faire, on utilise une boucle. Le mot-clé **while** permet de réaliser cela.

Vous trouverez, ci-dessous, un exemple complet de la structure d'une boucle.

```
instruction 1
while condition A: # se termine par :
  instruction 2 # indentation
  instruction 3 # indentation
instruction 4
```

Dans le programme ci-dessus, l'instruction 1 est exécutée puis les instructions 2 et 3 sont répétées tant que la condition A est vraie. Quand la condition A devient fausse, on passe à l'instruction 4. Si la condition A est fausse dès le départ, les instructions 2 et 3 ne sont jamais exécutées. La ligne du bloc qui commence par **while** se termine par « **:** » et les lignes suivantes sont indentées.

Si on veut exécuter les instructions 2 et 3 quand la condition est fausse, on doit ajouter **not** après while dans l'algorithme ci-dessus et si la condition est vraie alors c'est l'instruction 4 qui sera seulement exécutée.

Il se peut, parfois, que l'on veuille une boucle infinie, on utilise alors **while True :**. Une telle boucle est notamment utilisée quand on veut créer des fenêtres graphiques avec le module *Tkinter* par exemple. Attention, une boucle infinie est considérée comme un bug si aucune condition de sortie n'est prévue puisque l'on doit, alors, forcer l'arrêt du programme.

```
instruction 1
while True : # se termine par :
  instruction 2 # indentation
  instruction 3 # indentation
instruction 4
```

Dans ce cas, l'instruction 4 ne sera jamais exécutée si les instructions 2 et 3 ne permettent pas de sortir de la boucle.

5.2 For ... in ...

Une autre façon de faire une boucle est d'utiliser les mots-clés **for** et **in** quand on doit parcourir une séquence (chaîne de caractères ou liste). Cette structure de boucle évite de devoir définir et incrémenter un compteur. Voici des exemples pour mieux comprendre la syntaxe. Les possibilités sont nombreuses. Prenez un peu de temps pour vous attarder sur *end*, *sep* et *enumerate* :

Exemple 1 : Le parcours d'une chaîne de caractères

```
nom="Merlin"
print(nom)
for car in nom:
    print(car +"-",end="*") # * remplace un saut de
        ligne
print() #Pour changer de ligne
for car in nom:
  print(car ,"*",sep="-") # séparateur particulier
```

Exemple 2 : Le parcours d'une liste

```
liste=["arrosoir","seau","pelle","plantoir"]
for element in liste:
    print("le nombre de lettres de : ",element," est
        de :",len(element))
```

Exemple 3 : La table de 9

```
for i in range(11):
  print (i," *9= ",9*i)
```

Exemple 4 : carré des 100 premiers nombres impairs

```
for i in range(1,100,2):
  print (i," * ",i," = ",i*i)
```

Exemple 5 : L'intérêt de **enumerate**

```
Liste=["Bonjour",9,9.5,True,range(10),[1,False],(1,
    "x")]
for index,element in enumerate(Liste):
  print ("L'élément {0} est du type {1}".format(
      index,type(element))
```

6 Fonction input()

La fonction *input()* fait partie des fonctions prédéfinies comme la fonction *print()*. De façon courante, un programme peut nécessiter l'intervention de l'utilisateur pour entrer un paramètre par exemple. La fonction *input()* permet cela en provoquant une interruption dans l'exécution du programme pour que l'utilisateur puisse rentrer une valeur au clavier qui pourra être affectée à une variable. Une fois la valeur rentrée et après avoir validé avec la touche *Enter*, l'exécution du programme se poursuit.

```
prenom=input("Quel est votre prénom ?")
print("Bonjour, ",prenom,". Je suis ravi de vous
    rencontrer!")
print("Bonjour, {0}. Je suis ravi de vous
    rencontrer".format(prenom)) #formatage des str
print("Bonjour, %s. Je suis ravi aussi"%prenom) #
    ancien formatage
```

⚠ À savoir

La fonction *input()* renvoie toujours une chaîne de caractères. Si un entier est attendu en réponse ou autre chose, il faut alors faire une conversion de typage.

7 Applications

Les exercices qui suivent font appel à toutes les notions vues auparavant. Il n'existe pas qu'une seule solution. c'est ce qui fait le charme de la programmation.

7.1 Échauffement

7.1.1. Afficher les 20 premiers termes de la multiplications par 21.

7.1.2. Afficher une suite de 20 termes qui débute par le nombre que vous entrez avec la fonction *input* et où le nombre suivant est le triple du précédent.

7.1.3. Un jour, un roi fut sauvé des eaux par un vieux sage. Pour le récompenser, il prit la décision d'exaucer le vœu du vieil homme s'il trouvait sa demande raisonnable. Le sage prit un grain de riz et le posa sur la première case d'un échiquier. Il souhaita que son seigneur double ce nombre à chaque nouvelle case (2 grains sur la deuxième case, 4 sur la troisième ...) et il précise qu'il se contentera de tous les grains de riz que cela représentera. Le roi accepte. Afficher le nombre de grains de riz sur chaque case, ainsi que le nombre total de grains de riz en mole.

7.1.4. Écrire un programme qui demande à l'utilisateur de rentrer une année en entrée et en sortie le programme « dit »si cette année est bissextile ou non. Une année bissextile comporte 366 jours au lieu de 365 jours. Une année quelconque est bissextile si elle est divisible par 4 mais pas par 100 sauf si elle est divisible par 400. Par exemple, 1800 n'est pas bissextile alors que 2000 l'était. À vous de jouer !

7.1.5. Écrire un programme qui calcule les 1000 premiers nombres premiers. Un nombre premier est un nombre plus grand que 1 qui a pour uniques diviseurs, 1 et lui-même. Ce programme devra, par ailleurs, ressortir les nombres premiers jumeaux. Deux nombres premiers sont jumeaux si leur différence est égal à 2 (5 et 7 sont deux nombres premiers jumeaux).

7.2 Piste d'un CD

Le Compact Disc est un objet numérique. L'information y est stockée sous forme de séquences de *bits* qui valent 0 ou 1. À la surface d'un CD, il y a un sillon sur lequel se succèdent des plats (*lands*) et des creux (*pits*). On cherche à calculer la longueur totale de la piste en faisant l'approximation qu'il s'agit d'une succession de cercles. Dans la console, il s'affichera une phrase du genre : *Pour être lu complètement un CD doit tourner .....fois, la piste mesure environ ..... mètres.*

> La Program Area est la zone contenant des données. Elle commence à partir d'un rayon de 25 mm et s'étend jusqu'à 58 mm. La largeur d'un pit ou d'un land est de 0.6 μm, le pas de la spirale est de 1,59 μm.

7.3 Suite de Syracuse

Écrire un programme qui calcule les termes de la suite de Syracuse :

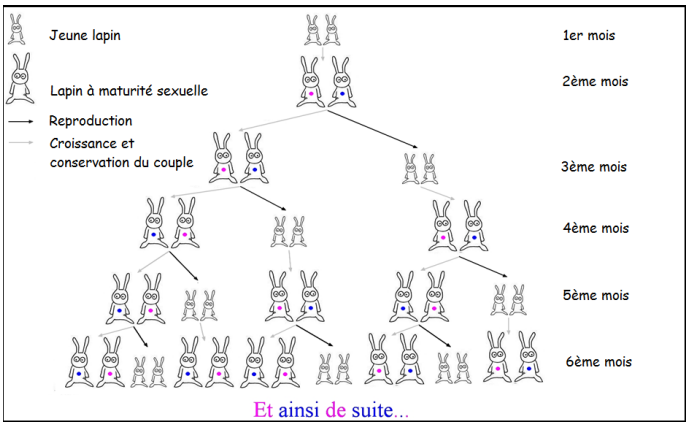
$u_0 = a$ où a est un entier quelconque et :

$$\begin{cases} u_{n+1} = \dfrac{1}{2}\, u_n & \text{si } u_n \text{ est pair} \\ u_{n+1} = 3\, u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Quel que soit le nombre de départ a, la suite se termine à 1. On appelle **vol** d'un nombre a le nombre de termes utilisés avant de retourner 1 et **hauteur de vol** la valeur maximale atteinte. Stockez dans des listes les longueurs et hauteurs de vol pour chaque valeur de a entier $\leqslant 1000$ et affichez les valeurs pour lesquels soit la hauteur de vol est la plus grande soit le vol le plus long.

7.4 Fibonacci

Un homme a un couple de lapins enfermés dans un enclos et il voudrait suivre l'évolution du nombre de couples de sa colonie de lapins, mois par mois, pendant 5 ans.



7.5 Spécial Prof

Écrire un programme qui demande à l'utilisateur de saisir au clavier des notes d'élèves au clavier jusqu'à ce que l'utilisateur tape « fin ». Avec les notes saisies, construire une liste, afficher le nombre de notes entrées, la note la plus basse, la plus élevée ainsi que la moyenne. Pour une liste L, Python comprend **min(L)**, **max(L)**, **sum(L)** mais essayez de vous en passer.

7.6 Enigme

Trouver un nombre de 4 chiffres qui soit le carré d'un nombre entier et tel que, en ajoutant un à chaque chiffre (ex : 1258 $\hookrightarrow$ 2369), on obtienne le carré d'un autre entier.

7.7 Suite de Conway

Programmer les 20 premiers termes de la suite de Conway (voir sur Wikipédia).

7.8 Codage

Écrire un programme qui intercale un mot dans une phrase. Si on intercale « python » dans « Hello World ! » cela donne « Hpeyltlhoo nwpoyrtlhdo!n ». On affiche ensuite la chaîne de caractères en l'inversant.

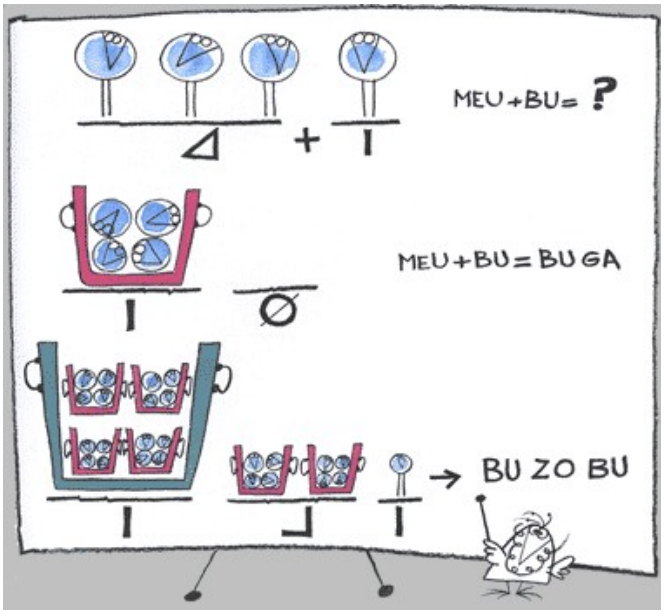
7.9 Fraude à la sécurité sociale

Faire un programme qui teste une liste de numéros de sécurité sociale afin de voir si ceux-ci sont valides ou non. Pour cela, il faut savoir que le nom officiel est Numéro d'Inscription au Répertoire des Personnes Physiques (NIRPP ou **NIR**). Il est construit à partir de l'état civil transmis par les mairies (sexe (1 chiffre : 1 ou 2), année (2 chiffres) et mois de naissance (2 chiffres), département (2 chiffres) et commune de naissance (3 chiffres), numéro d'ordre du registre d'état civil (3 chiffres)). C'est un numéro « signifiant », composé de 13 chiffres, suivi d'une clé de contrôle de 2 chiffres (complément à 97 du nombre formé par les 13 premiers chiffres du NIR modulo 97).Voici les numéros à mettre dans votre liste :

2550814168025 12
1760664543295 50
5632436397578 0
1770976451032 10
2821275114123 14

7.10 Résolution de problème

Compter jusqu'à 100 en langue Shadok. Pour vous aider, voici l'explication du professeur Shadoko :



Je me souviens qu'il avait dit :« Quand on a MEU shadoks, et qu'on en ajoute BU, il n'y a plus de place. On les met alors dans une poubelle. Il y a donc BU poubelle et GA shadok à côté ».

Chapitre III

Modules/fonctions

1 Les modules

Python offre de très nombreuses bibliothèques de fonctions prédéfinies pour réaliser des tâches courantes. On a déjà vu les fonctions *input* et *print*. D'autres fonctions sont regroupées dans ce qui s'appelle des modules. Contrairement aux fonctions *print* et *input* qui sont accessibles directement, pour pouvoir utiliser une fonction d'un module, il est nécessaire d'importer la fonction explicitement. Par exemple, pour utiliser la fonction cos (cosinus) du module **math** :

```
1 from math import cos
2 print(cos(2.0))
```

La ligne **from math import cos** indique que la fonction cos est importée du module math. Si on souhaite importer toutes les fonctions du module *math*, on peut écrire :

```
1 from math import *
```

Cette façon d'importer toutes les fonctions est pratique quand on utilise l'interpréteur interactif, mais peu recommandée dans les programmes, car elle nuit aux performances et à la lisibilité. La méthode recommandée est la suivante :

```
1 import math
```

import math importe le module math, mais les fonctions du module ne sont pas accessibles directement. Il faut alors utiliser le préfixe **math.** pour désigner la fonction. Cette méthode permet d'avoir des modules différents qui contiennent chacun une fonction du même nom. Par exemple les modules scientifiques *numpy* et *math* contiennent tous deux une fonction racine carrée *sqrt* mais celle du module math est un peu moins puissante. Essayez :

```
1 import math
2 print(math.sqrt([4,9,16]))
```

et

```
1 import numpy as np #je renomme en plus numpy
2 print(np.sqrt([4,9,16]))
```

C'est pourquoi, pour les équations de fonctions dans les graphiques (dans une partie que nous verrons plus tard) le module *numpy* (ou le module *scipy*) est recommandé. Mais le module *math* contient les principales fonctions mathématiques utiles pour les cas généraux. Un aperçu est donné ci-contre.

Il est à noter que les angles doivent être donnés en radians. Il existe des descriptifs de toutes les fonctions disponibles pour un module donné sur internet. Il faut cependant s'armer de courage pour les lire car ceux-ci peuvent être composés d'un millier de pages.

module math	mathématiques
math.**factorial**(n)	$n!$
math.**fabs**(x)	$\|x\|$
math.**exp**(x)	e^x
math.**log**(x)	$\ln(x)$
math.**log10**(x)	$\log(x)$
math.**pow**(x,y)	x^y
math.**sqrt**(x)	$\sqrt{x}$
math.**sin**(x)	$\sin(x)$
math.**cos**(x)	$\cos(x)$
math.**tan**(x)	$\tan(x)$
math.**asin**(x)	$\arcsin(x)$
math.**acos**(x)	$\arccos(x)$
math.**atan**(x)	$\arctan(x)$
math.**pi**	π
math.**e**	e
math.**floor**(x)	premier entier $\leqslant x$
math.**ceil**(x)	premier entier $\geqslant x$

Une autre bibliothèque fréquemment utilisée est la bibliothèque **random**. Elle permet de générer des nombres aléatoires avec notamment la fonction *random()* pour avoir un nombre flottant compris ente 0 et 1, la fonction *randrange(n)* pour avoir un nombre entier aléatoire compris entre 0 et n, la fonction *randint(a,b)* pour avoir un entier au hasard entre a et b.

Application directe

Créer une liste de 100 nombres aléatoires, compris entre 0 et 100. En sortie, on affichera cette liste sans aucun doublon.

2 La fonction

2.1 Utilité

Lorsque l'on écrit des programmes, nous sommes souvent amenés à répéter plusieurs fois une même séquence d'instructions. C'est fastidieux, surtout que des erreurs ne manquent pas de se produire. Pour nous faciliter la vie, le langage dispose d'une syntaxe qui permet de regrouper une séquence et de la désigner par un mot unique, une fonction. Quand le programme commence à avoir de nombreuses lignes de code, il devient aussi plus difficile pour un tiers de le comprendre. Mettre des fonctions qui exécutent des tâches particulières en tête du programme permet de le rendre plus compréhensible. Une fonction est définie par un nom, par ses arguments qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel, et éventuellement une valeur de retour communiquée au programme par la fonction en fin d'exécution. L'ajout d'un commentaire est toujours un plus pour la clarté des choses.

Admettons que je veuille récapituler une journée de championnat de Quidditch avec une feuille de score de la forme :

L'équipe de Appleby bat l'équipe de Ballycastle.

L'équipe de Caerphilly bat l'équipe de Chudley.

L'équipe de Falmounth bat l'équipe de Holyhead.

L'équipe de Kenmare bat l'équipe de Montrose.

L'équipe de Puddlemere bat l'équipe de Tutshill.

La solution la plus « primaire » serait de faire une méthode lourde, écrire 10 lignes commençant par **print**. On peut également utiliser une boucle **for** qui parcourt une liste avec des paires de noms d'équipe.

```
1 print("L'équipe de Appleby.....")
2 print("--------------------------")
3 print( etc etc etc ....
4 #ou
5 for (i,j) in ([("Appleby","Ballycastle"),("
      Caerphilly","Chudley")]):
6   print("L'équipe {} bat l'équipe {}.".format(i,j))
7   print("-----------------------------------")
```

La façon la plus élégante de procéder, dans un cas général, reste de créer une fonction que l'on pourra appeler *score* par exemple. Voyons comment faire.

2.2 Structure

Une fonction est introduite par le mot-clé **def**, suivi du **nom de la fonction**, de ses paramètres entre **parenthèses**, de **deux points**, puis d'un bloc d'instructions. Ce bloc d'instructions s'appelle le corps de la fonction. Il doit être **indenté** et la fin de ce bloc marque la fin de la définition de fonction.

```
1 def score(equipe1,equipe2):
2     """Cette fonction donne l'équipe gagnante d'une
          rencontre de Quidditch"""
3     print("L'équipe de {} bat l'équipe de {}.".
          format(equipe1,equipe2))
4     tiret=50*'-'
5     print(tiret)
6
7
8 #début du programme
9 score("Appleby","Ballycastle")
```

L'instruction de la ligne 9, *score*(...), qui est dans le corps du programme appelle la fonction *score*. Organiser un programme à l'aide de fonctions permet d'éviter les répétitions de lignes de code. Cela rend les programmes plus clairs et plus faciles à lire : pour comprendre le programme, il n'est pas nécessaire de savoir comment la fonction *score* est programmée, il suffit de savoir ce qu'elle fait. Enfin, cela permet d'organiser l'écriture du programme. On peut décider d'écrire la fonction *score* un jour et le programme principal le lendemain. On peut aussi organiser une équipe de manière à ce qu'un programmeur écrive la fonction *score* et un autre le programme principal.

2.3 Return

On vient de voir une fonction qui ne retournait aucune valeur, elle se contentait d'écrire. On peut cependant avoir besoin d'une fonction qui effectue un calcul et qui nous renvoie le résultat de ce calcul (une fonction qui a deux paramètres u et v, abscisse et ordonnée d'un vecteur $\vec{V}$ et qui nous renvoie la norme de ce vecteur). On utilisera alors le mot-clé **return**.

```
1  import numpy as np
2
3  def norme(u,v):
4      """Cette fonction renvoie la norme d'un vecteur
           d'abscisse u et d'ordonnée v"""
5      norme=np.sqrt(u**2+v**2)
6      return norme
7
8  abs=3
9  ord=4
10 norme=norme(abs,ord)
11 print("La valeur de la norme du vecteur({0},{1})
       est {2}".format(abs,ord,norme))
```

⚠ À savoir

Il faut savoir que le mot-clé **return** peut se trouver au milieu d'une fonction et non pas à la fin. Cela pour effet d'interrompre la fonction dès qu'il est exécuté. Et pour être précis, une fonction avec un **return** est une vraie fonction. Quand il n'y en a pas, cela s'appelle une procédure.

Exemple :

```
1 def cube(w):
2   return w*w*w
3   print("ceci ne sera jamais écrit")
4
5 x=cube(9)
6 print(x)
```

2.4 Règles usuelles

Quand on conçoit une fonction, il est préférable de lui donner un nom explicite, car elle est susceptible d'apparaître à de nombreux endroits dans le programme. (On n'imagine pas que les fonctions de la bibliothèque Python s'appellent f1 , f2 , etc...). En revanche, les noms des arguments formels peuvent être courts, car leur portée, et donc leur signification, est limitée au corps de la fonction.

Il convient par ailleurs de documenter convenablement les fonctions, en spécifiant les hypothèses faites sur les arguments formels, leurs relations avec le résultat renvoyé, mais aussi les effets de la fonction (affichage, etc.) le cas échéant. Python propose un mécanisme pour associer une documentation à toute fonction, sous la forme d'une chaîne de caractères entre triple quote, placée au début du corps de la fonction.

⚠ À savoir

La documentation d'une fonction f peut être affichée en écrivant help(f).

```
def cube(w):
  """Renvoie la valeur de w élevée au cube"""
  return w*w*w

help(cube)
```

Application directe

Écrire une fonction qui prend en arguments deux entiers $n \geqslant 0$ et $d > 0$ et qui renvoie un couple formé du quotient et du reste de la division euclidienne de n par d. Le quotient et le reste seront calculés par soustractions successives. Une documentation sera comprise dans la fonction.

3 Variable locale et variable globale

Lorsque nous initialisons des variables au début d'un programme, celles-ci sont dites des variables globales, elles sont donc accessibles à l'intérieur d'une fonction.

```
def inutile():
  print(x)

x=5
inutile()
```

Ici, dans la fonction, la variable x est inconnue, l'interpréteur va donc voir dans le programme si la variable existe. Par contre, quand une variable est définie à l'intérieur d'une fonction celle-ci n'existe que pour cette fonction.

```
def inutile():
  x=1
  print(x)

x=5
inutile()
```

La variable x est une variable locale puisqu'elle est définie dans la fonction, c'est elle qui sert dans le *print()*. Il y a une variable locale x et une variable globale x. Aucun conflit. Deux entités x coexistent.

Maintenant si on fait :

```
def inutile():
  print(x)
  x=1

x=5
inutile()
```

On voit qu'il y a un problème, la variable locale x est définie mais après son appel dans la fonction *print()*. La variable locale est prioritaire par rapport à la variable globale du même nom. Pour illustrer cette notion d'indépendance, essayez :

```
def inutile():
  x=1

x=5
print(x)
inutile()
print(x)
```

Quand on crée une variable dans une fonction, on n'a pas à se préoccuper de savoir si ce nom a été utilisé dans le programme principal. Par contre, il se peut que nous ayons besoin que notre fonction modifie la variable x du programme principal.

Premier essai :

```
def inutile(a):
    x=a+3
    print(x)

x=5
inutile(x)
print(x)
```

La variable globale x est bien envoyée en argument dans la fonction *inutile()*. J'obtiens juste que la variable locale x prend la valeur 8. Mais la variable globale x reste à 5. **C'est un échec**.

Deuxième essai : utilisation de **return**

```
def inutile(a):
    x=a+3
    print(x)
    return x

x=5
x=inutile(x)
print(x)
```

On a bien modifié la variable globale x avec une fonction. Cette méthode est à privilégier mais il existe une solution plus simple qui peut aussi être source d'erreurs difficilement identifiables.

Troisième essai : **global**

```
def inutile(a):
    global x
    x=x+a
    print(x)

x=5
inutile(x)
print(x)
```

⚠ À savoir

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les constantes du problème. En pratique, on ne devrait pas recourir souvent à la construction **global** de Python. Comme pour les fonctions, il est préférable de donner aux variables globales des noms longs et explicites, ce qui les distinguera de fait des variables locales qui portent habituellement des noms courts (comme les paramètres formels).

Application directe

Que donne en sortie le programme suivant :

```
def fct1():
    global x
    x=x+8
    x=12
def fct2():
  x=3

x=5
fct1()
fct2()
print(x)
```

4 Ordre d'évaluation

L'ordre dans lequel les arguments d'une fonction sont évalués a une influence sur le résultat. Vérifiez que **Python évalue les arguments d'une fonction de la gauche vers la droite**.

```
n = 1
def fct1(x):
    global n
    n = n+5
    return x + n
def somme(x, y):
    return x + y
x=somme(fct1(1),n)
print(x)
```

Montrer qu'une évaluation de la droite par la gauche des arguments aurait donné un résultat différent.

5 Récursivité

En programmation, rien n'interdit d'appeler une fonction f à l'intérieur de la fonction f elle-même. La fonction f est dite alors **récursive**.

Considérons la suite (u_n) qui calcule une approximation de $\sqrt{3}$:

$$\begin{cases} u_0 = 2 \\ u_1 = \dfrac{1}{2}\left(u_{n\text{-}1} + \dfrac{3}{u_{n\text{-}1}}\right) \end{cases}$$

Cette suite peut être calculée à l'aide d'une fonction qui suit directement la définition ci-dessus :

```
import numpy as np
def racine(n):
  if n == 0:
    return 2.
  else:
    return 0.5 * (racine(n-1) + 3 / racine(n-1))

print(racine(4))
print(np.sqrt(3))
```

Application directe

Écrire un programme qui donne les termes de la suite de Fibonacci en utilisant la récursivité :

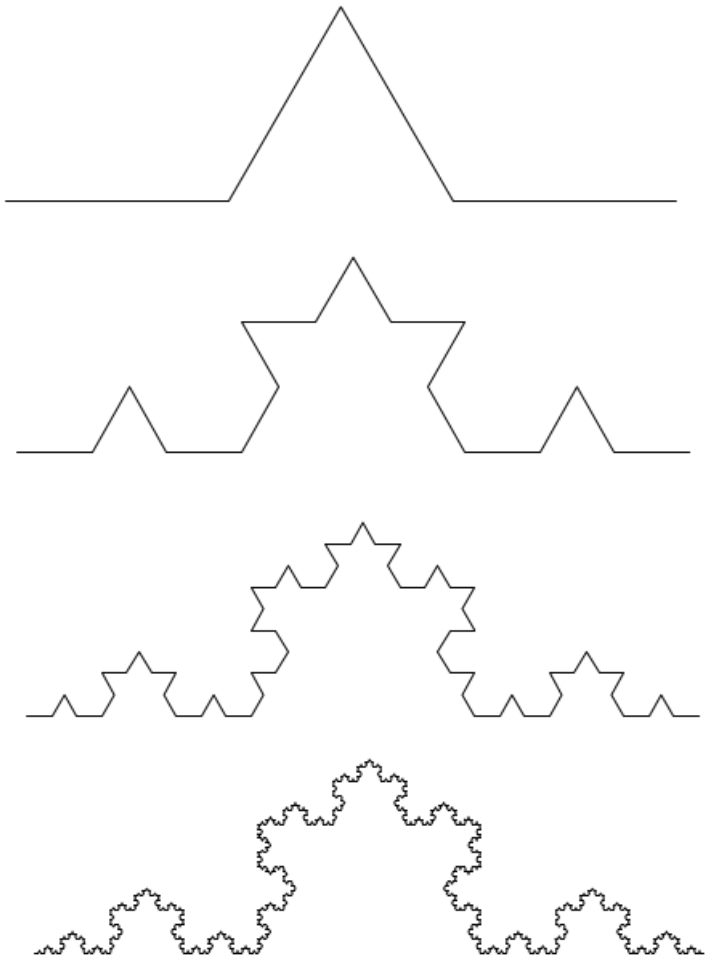
$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_n = u_{n\text{-}1} + u_{n\text{-}2} \text{ si } n \geqslant 2 \end{cases}$$

Que se passe-t-il si on augmente n, par exemple $n = 50$? Comparez avec une méthode non-récursive.

La performance doit toujours être votre priorité. Par conséquent, la récursivité n'est pas toujours la meilleure manière de faire. Par contre, c'est une notion essentielle quand on veut dessiner des fractales en utilisant le module de dessin **Turtle** de Python.

La plus connue des fractales est le flocon de Koch. On part d'un triangle, donc trois côtés. Chaque côté est divisé en trois segments et on remplace le segment du milieu par deux autres segments qui forment un triangle équilatéral et ainsi de suite....

Je donne une version Python du flocon de Koch avec les fichiers joints pour ceux qui veulent essayer.



Chapitre IV

Les fichiers

On va commencer ici à aborder des notions essentielles pour le professeur de physique. Par exemple, dans les nouveaux programmes de seconde, on demande explicitement dans le B.O que l'élève maîtrise des capacités numériques spécifiques telles que *la représentation des positions successives d'un système modélisé par un point lors d'une évolution unidimensionnelle ou bidimensionnelle à l'aide d'un langage de programmation* et *la représentation des vecteurs vitesse d'un système modélisé par un point lors d'un mouvement à l'aide d'un langage de programmation.*

Ce que l'on va faire ici marchera dans tous les cas, mais nous verrons une méthode beaucoup beaucoup plus simple ensuite si les données ont été exporté d'une façon facilement exploitable.

1 Ouverture et fermeture

L'ouverture d'un fichier se fait avec cette ligne :

```
variable=open("chemin_complet","mode en ouverture")
```

Dès que l'on ouvre un fichier ainsi, il faut écrire sa fermeture, cela évitera un oubli possible.

```
variable.close()
```

mode	effet
'r'	mode lecture seule
'w'	mode écriture, fichier crée sinon écrasé
'a'	fichier crée sinon données ajoutées en fin de fichier

La première chose que l'on va faire sera de créer un fichier qui n'existe pas encore, comme cela il sera placé dans le même dossier que notre programme Python.

```
objet=open('objet.txt','w')
objet.close()
```

Quand vous exécutez ces deux lignes, un fichier texte apparaît. Si on l'ouvre avec un éditeur de texte, on constate qu'il est vide. Maintenant on va le remplir en l'ouvrant en mode « a », on veut ajouter des données et non écraser ce qui existe même si il n'y a rien pour l'instant.

```
objet=open('objet.txt','a')
objet.write('Lampe')
objet.close()
```

Quand on va voir ce que contient notre fichier avec un éditeur de texte, on remarque que le mot « Lampe » est bien écrit. Si on veut s'éviter la peine d'écrire cette dernière ligne avec le *close*, on peut adopter une autre syntaxe :

```
with open('objet.txt','a') as objet:
  objet.write('Table')
```

Et là, avec l'éditeur de texte, on voit que le nouveau mot est bien ajouté mais qu'il est collé au premier. On va ajouter avec l'éditeur une deuxième ligne dans le fichier « objet.txt » qui contient un seul mot : **Chaise**. On enregistre et on referme le fichier. On passe à la lecture.

2 Lecture et écriture

On ouvre et on lit le fichier avec Python très facilement :

```
with open('objet.txt','r') as text: # 'r' pour
    lecture
  print(text.readlines())
```

La méthode **readlines** appliquée à l'objet que j'ai nommé **text** cette fois, permet de lire tout le fichier. Dans la console, on peut lire quelque chose comme :

```
['LampeTable\n', 'Chaise']
```

On obtient une liste dont chaque élément est une chaîne de caractères. On remarque dans l'élément zéro, un \n. C'est un caractère invisible pour un éditeur de texte qui représente un saut de ligne. Donc en Python, si on veut écrire une nouvelle ligne dans un fichier, nous devons taper :

```
with open('objet.txt','a') as objet:
  objet.write('\nArmoire')
```

Il existe également une autre méthode de lecture qui s'appelle **readline**. C'est une méthode pas à pas, une seule ligne est lue à chaque fois.

```
with open('objet.txt','r') as text: # 'r' pour
    lecture
  print(text.readline())
```

Si on veut lire les trois lignes :

```
with open('objet.txt','r') as text: # 'r' pour
    lecture
  for i in range(3):
    print('Sur la ligne {} on peut lire {}'.format(i
        ,text.readline()))
```

On remarque que chaque ligne est considérée comme une chaîne de caractère, le \n n'est plus affiché mais il est exécuté car dans la console il y a un espace d'une ligne entre chaque ligne.

Application directe

Écrire un programme qui écrit dans un fichier le résultat de 10 000 lancers de 3 dés ainsi que la somme, du style :
Lancer ;dé 1 ;dé 2,dé 3 ;somme
1 ;5 ;3 ;3 ;11
2 ;2 ;6 ;1 ;9
3 ;1 ;1 ;1 ;3

3 Traitement

Dans le chapitre suivant, on abordera les graphiques. Pour tracer un graphique, il nous faut, au minimum, deux listes de points, une pour les abscisses, une autre pour les ordonnées. Nous allons partir d'un exemple réel avec un fichier *.txt* qui rassemble les données obtenues lors d'un pointage vidéo avec Latis-Pro. Cet exemple est formateur car il y a une image manquante sur la vidéo pointée donc une erreur dans une ligne « corrompue » dans le fichier .txt.

```
Temps;Mouvement X;Temps;Mouvement Y
0;-0.591983083570837;0;2.1124390916613
0.04;-0.783212282798566;0.04;2.39192792130183
0.08;-0.989151420428427;0.08;2.59786705893169
0.12;-1.19509055805829;0.12;2.80380619656155
0.16;-1.40102969568815;0.16;2.99503539578928
0.2;-1.60696883331801;0.2;3.14213477981061
0.24;-1.81290797094787;0.24;3.30394410223407
0.28;-1.98942723177347;0.28;3.43633354785327
0.32;-2.21007630780546;0.32;3.5393031166682
0.36;-2.40130550703319;0.36;3.64227268548313
0.4;-2.62195458306518;0.4;3.74524225429806
0.44;-2.81318378229291;0.44;3.80408200790659
0.48;-3.00441298152064;0.48;3.84821182311299
0.52;-3.2103521191505;0.52;3.87763169991726
0.56;-3.40158131837823;0.56;3.90705157672152
0.6;-3.59281051760596;0.6;3.92176151512366
0.64;-3.79874965523582;0.64;3.92176151512366
0.68;-3.98997885446355;0.68;3.89234163831939
0.72;-4.18120805369128;0.72;3.86292176151512
0.76;-4.38714719132114;0.76;3.81879194630872
0.8;-4.593086328951;0.8;3.73053231589593
0.84;-4.76960558977659;0.84;3.65698262388526
0.88;-4.97554472740645;0.88;3.55401305507033
0.92;-5.16677392663418;0.92;3.46575342465753
NAN;NAN;NAN;NAN
1;-5.53452238668751;1;3.21568447182127
```

Voici une façon de faire :

```
import numpy as np
liste=["0","1","2","3","4","5","6","7","8","9",".",
    ";","-"]
erreur=[]

with open('fichier_pointvirgule.txt',"r") as fich:
    texte=fich.readlines()#on utilise le parcours de
        l'objet fich
for i in range(len(texte)) :
    texte[i]=texte[i].replace("\n","")
    for j in range(len(texte[i])):
        if not texte[i][j] in liste:
            if not i in erreur:
                erreur.append(i)

#on parcourt la liste erreur à l'envers pour que
    les index ne changent pas
for k in erreur[::-1]:
    texte.pop(k)
for i in range(len(texte)):
    texte[i]=texte[i].split(';')
for i in range(len(texte)) :
    for j in range(len(texte[i])):
        texte[i][j]=eval(texte[i][j])

#on transforme la liste en tableau numpy plus
    pratique
tableau=np.array(texte)
print(type(tableau))
#on extrait la premiere colonne avec :
temps=tableau[::1,0]
X=tableau[::1,1]
Y=tableau[::1,3]
```

Si on introduit des fonctions (dont deux pour sauvegarder notre fichier sans erreur de deux manières différentes), cela devient :

```
import numpy as np

def ouverture():
  """On ouvre le fichier qui nous intéresse"""
    with open('fichier_pointvirgule.txt',"r") as
        fich:
        texte=fich.readlines()#on utilise le
            parcours de l'objet fich
        return texte

def erreur(texte):
    erreur=[]
    liste=["0","1","2","3","4","5","6","7","8","9","
        .",";","-"]
    for i in range(len(texte)) :
        texte[i]=texte[i].replace("\n","")

        for j in range(len(texte[i])):
            if not texte[i][j] in liste:

                if not i in erreur:
                    erreur.append(i)
    return erreur

def elimination(texte,erreur):
    """fonction qui élimine les lignes erronnées,on
        parcourt la liste erreur à l'envers pour
        que les index ne changent pas"""
    for k in erreur[::-1]:
        texte.pop(k)
    return texte

def creation_liste(texte):
    for i in range(len(texte)):
        texte[i]=texte[i].split(';')
    for i in range(len(texte)) :
        for j in range(len(texte[i])):
            texte[i][j]=eval(texte[i][j])
    return texte

def creation_tableau(texte):
  """on transforme la liste en tableau numpy plus
      pratique"""
    tab=np.array(texte)
    return tab
```

```
40
41
42  def sauvegarde(texte):
43      """Sauvegarde à partir de la liste créee"""
44      with open('pointage.txt','w') as fich:
45        fich.write('Temps;Mouvement X;Temps;Mouvement
              Y\n')
46          for i in range (len(texte)):
47              fich.write(str(texte[i])+'\n')
48
49
50  def sauvegarde2(texte):
51      """"On sauvegarde notre fichier expurgé des
            erreurs à partir de la liste de liste"""
52      with open('pointage2.txt',"w") as fich:
53          for i in range (len(texte)):
54              for j in range(len(texte[i])):
55                  fich.write(str(texte[i][j])+';')
56              fich.write('\n')
57  ##########début du programme############
58  premier_traitement=ouverture()
59  lignes_erronnees=erreur(premier_traitement)
60  deuxieme_traitement=elimination(premier_traitement,
        lignes_erronnees)
61  sauvegarde(deuxieme_traitement)
62  liste=creation_liste(deuxieme_traitement)
63
64  sauvegarde2(liste)
65  tableau=creation_tableau(liste)
66
67  #on extrait la premiere colonne avec :
68  temps=tableau[::1,0]
69  X=tableau[::1,1]
70  Y=tableau[::1,3]
```

Ici, on pourrait enlever la ligne avec un éditeur de texte car il y a peu de lignes de donnée cependant plus tard nous représenterons la hauteur d'eau au marégraphe de Dieppe. La hauteur est mesurée toutes les dix minutes du 01 septembre 2018 au 31 décembre 2018 soit plus de 17 000 points donc si une erreur existe, il sera difficile de l'identifier. Cependant, partir du principe que nos fichiers ne sont pas corrompus est source de simplification. En outre, dans la littérature « pythonnesque », on ne fait pas grand cas d'une autre façon de faire. Or, comme c'est ma préférée, je vais l'évoquer.

4 numpy.loadtxt

Les données que nous utilisons sont disposées de façon régulière. Une colonne pour le temps, une pour des abscisses des points, une pour des ordonnées des points, le tout séparé par « ; », « , » ou une tabulation. Ouvrons le fichier de données du pointage vidéo expurgé « pointage.txt » pour extraire les valeurs.

```
1  import numpy as np # le module numpy est renommé np
2  Temps,X,Y=np.loadtxt('pointage.txt',delimiter=';',
       unpack=True,usecols=(0,1,3),skiprows=1)
```

Temps,X et Y sont directement extraits du fichier et exploitables, ce ne sont pas des listes mais des tableaux d'une seule dimension. *delimiter* permet d'indiquer comment les colonnes sont séparées, *unpack=True* indique que l'on veut les séparer, *usecols* permet de choisir la séquence de colonnes que l'on souhaite utiliser, *skiprows* permet d'éliminer un certains nombres de lignes d'en-tête. Il existe aussi *dtype* qui peut être utile pour préciser le type de variable de chaque colonne car c'est le type *float* qui est par défaut. Vous pourrez revenir à cette partie quand j'utiliserai *dtype* pour convertir une colonne où le temps sera à convertir sur l'exemple de la marée dieppoise.

Application directe

Le traitement du fichier « pointage.txt » donnera une représentation correcte des vecteurs vitesse mais pas des vecteurs accélération. Aussi votre mission, si vous l'acceptez, sera de créer un fichier de données d'une chute libre à partir des équations horaires. Sauvegardez-le sous le nom de « mon_fichier.txt », nous l'utiliserons pour *étudier la relation approchée entre la variation du vecteur vitesse d'un système modélisé par un point matériel entre deux instants voisins et la somme des forces appliquées sur celui-ci* (programme première spécialité).

Chapitre V

Les graphiques

1 Les bases

Il est possible avec Python de créer un graphique. Le module **matplotlib** a tous les outils graphiques utiles pour les tracés de fonctions. Le module **numpy** possède quant à lui tout un arsenal de fonctions prédéfinies très utiles. On veut, par exemple, tracer la fonction f définie par $f(x) = x$.

```
import matplotlib.pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np
x=np.arange(10) #crée un tableau de 10 valeurs équiréparties commençant à 0.
y=x
print(x) #facultatif
print(y) #facultatif
plt.plot(x,y) # Trace la fonction mais ne montre pas le graphique.
# on peut écrire également plt.plot(x,x)
plt.show() # Affiche le graphe.
```

Cela peut paraître curieux d'avoir une fonction qui trace la fonction et une autre qui affiche le graphe mais c'est utile quand on a plusieurs courbes lourdes. On les charge toutes tranquillement en mémoire puis on les affiche toutes avec **plt.show()**.

Si tout se passe correctement, vous devriez voir votre graphe avec un axe horizontal, un axe vertical et la courbe en bleu. S'il y a plusieurs courbes, chacune aura une couleur différente.

À noter qu'il existe une autre façon de faire, qui est plus simple mais que j'utilise peu, non par préférence mais par habitude. *Spyder* aime moins aussi, même si tout se passe bien à l'exécution, il y a beaucoup de signaux ⚠ qui apparaissent. Je vous la livre quand même, vous ferez votre choix vous-même.

```
from pylab import *

x=arange(10) #crée un tableau de 10 valeurs équiréparties commençant à 0.
y=x
plot(x,y) # Trace la fonction mais ne montre pas le graphique.
show() # Affiche le graphe.
```

Attardons-nous sur **numpy.arange(start,stop,step)**. Cela permet de générer un tableau de valeurs qui inclut la valeur en *start* mais qui exclut la valeur en *stop*. L'écart entre deux valeurs étant fixé par *step*. Cela fonctionne très bien avec un *step* entier. Sinon on préférera **numpy.linspace(start,stop,num)** qui génère un tableau de *num* valeurs qui commence à *start* et qui finit à *stop*.

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

x=np.arange(1,11,1) # 10 valeurs de 1 à 10
y=np.linspace(1,100,10)
print(x,y)
plt.plot(x,y)
plt.show()
```

2 Les options de plot

Entre les parenthèses du mot-clé **plot** nous avons seulement indiqué les deux paramètres essentiels, x et y. Or, on peut ajouter de nombreuses options au *plot*. Vous allez voir que tout est paramétrable.

L'écriture est la suivante :
plt.plot(abscisse, ordonnée, "couleur $style_{point}$ $style_{trait}$", *paramètres*).

couleur : première lettre anglaise de la couleur(b,g,y,c,r,m,w ou « k »pour le noir).
$style_{point}$: liste ci-dessous.
$style_{trait}$: ' :' pointillé, '-' ligne continue, '-.'point tiret, '- -' ligne de tirets, ',' pixel, 'v' triangle.
paramètres : cela peut être l'épaisseur du trait (linewidth=flottant) et/ou une légende (label="ce qui doit être écrit"). Il existe aussi **markersize** ou ms, **markeredgecolor** ou mec, **markerfacecolor** ou mfc... Tout est listé sur **https ://matplotlib.org**. Il faut essayer, échouer, réessayer, chercher mais au final il y a toujours une solution pour ce que vous voulez entreprendre.

Regardez ce que signifie markeredgecolor ou markerfacecolor :

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

x=np.linspace(1,100,30)
plt.plot(x,np.sqrt(x),"b+:",markeredgecolor='r')
plt.show()
```

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

x=np.linspace(1,100,30)
plt.plot(x,np.log(x),"bo:",mec='g',ms=15,mfc='c',mew=5)
plt.show()
```

⚠ À savoir

En promenant la souris sur le graphique, les coordonnées du pointeur sont inscrites en bas de la fenêtre.

paramètre	marqueur
x	une croix
X	une croix pleine
+	un plus
P	un plus plein
.	un pixel
o	un rond
1	tri_ bas
2	tri_ haut
3	tri_ gauche
4	tri_ droite
^	triangle pointe en haut
v	triangle pointe en bas
>	triangle pointe droite
<	triangle pointe gauche
p	pentagone
h	hexagone
H	autre hexagone
*****	étoile
d	carreau
D	carreau plus gros
|	barre verticale
_	barre horizontale

⚠ Si vous utilisez une légende avec *label*, rien ne sera affiché sans écrire une ligne qui appelle la légende. Si on appelle le module *pyplot* avec le raccourci *plt*, il faudra ajouter *plt.legend()* :

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

x=np.linspace(1,100,30)
plt.plot(x,np.log(x),"b+:",label=r'$y=log(x)$')
plt.legend()
plt.show()
```

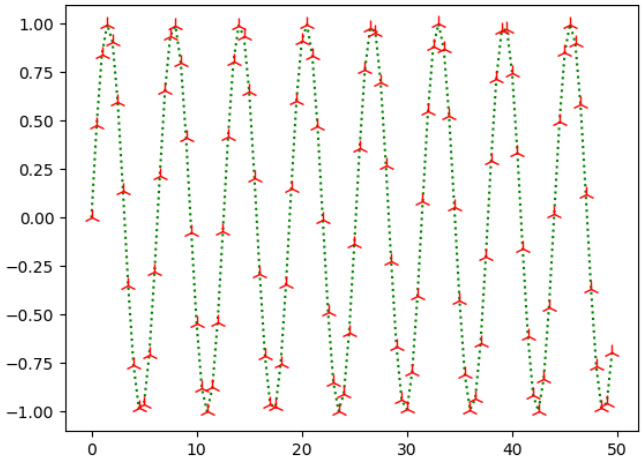
On verra, un peu plus loin, comment faire autrement mais dans un premier temps, faisons simple. Entre les parenthèses de *legend*, on peut aussi mettre des paramètres. Voici quelques paramètres possibles , la liste est non-exhaustive et consultable toujours sur internet sur le site cité plus haut :

paramètre	marqueur
0,1...10	au mieux, haut droite ...centre
fontsize=	taille
fancybox=True or False	boîte arrondie ou pas
Shadow=True or False	ombre sous la boîte
facecolor=	couleur du fond de la boîte
edgecolor=	couleur du contour
title=	titre de la boîte
title_fontsize=	taille du titre
ncol=	nombre de colonne occupée

Application directe

Mofifier le programme pour afficher un graphe comme celui ci-dessous.

```
import matplotlib.pyplot as ... # Le module pyplot est renommé plt
import numpy as ... # Le module numpy est renommé np
x=np.arange(.....) # Crée un tableau de valeurs de x=0 à 50 avec deltax=0.50.
y=..... # On trace sinus x
plt.plot(x,y) # Trace la fonction comme le montre l'image ci-dessous
..... # Affiche le graphe.
```



3 Les méthodes

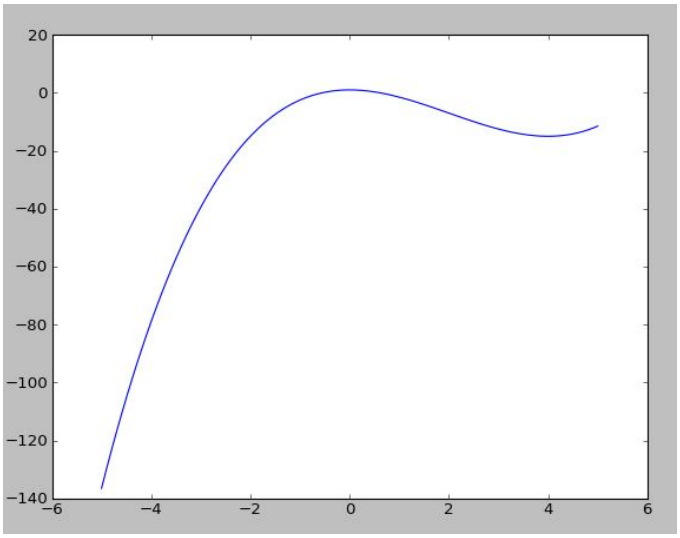
Nous allons ici voir comment faire de beaux graphiques en utilisant quelques lignes de code particulières. Il n'y a pas de meilleures façons d'apprendre qu'en essayant par vous-même. On veut, par exemple, tracer la fonction $f(x)$ telle que :

$$\begin{cases} x \in [-5;5] \\ f(x) = \dfrac{1}{2}x^3 + 3x^2 + 1 \end{cases}$$

```
import matplotlib.pyplot as plt # On renomme le tout "plt".
import numpy as np # Le module numpy est renommé np .
x=np.linspace(-5,5,100) # la courbe est tracée à partir de 100 points sur [-5;5].
plt.plot(x,x**3/2-3*x**2+1,label='f(x)')
plt.legend()
plt.show()
```

Application directe

Ci-contre, le résultat le plus basique, à vous de l'enrichir avec les possibilités listées dans le tableau qui suit.



Instructions	Significations
np.linspace(a,b,N)	Crée une matrice ligne de N valeurs régulièrement espacées.
plt.plot(x,f(x))	Trace la fonction f de variable x.
plt.show()	Affiche le graphique.
plt.clf()	Efface la fenêtre graphique.
plt.axis([xmin,xmax, ymin,ymax])	Spécifie la fenêtre de représentation.
plt.savefig()	Sauvegarde le graphique en format png par défaut.
plt.title(*texte*)	Ajoute le titre *texte* au graphique.
plt.xlabel(*texte*)	Affiche *texte* sur l'axe des abscisses.
plt.ylabel(*texte*)	Affiche *texte* sur l'axe des ordonnées.
plt.text(x,y,*texte*)	Affiche *texte* à la position (x, y).
plt.grid(True/False)	Permet d'avoir une grille ou pas.
plt.legend()	Permet d'afficher la légende.

Vous noterez qu'en ajoutant un r devant la chaîne de caractères, on peut afficher des formules mathématiques pour peu que l'on connaisse un minimum la syntaxe LaTeX. On est plus dans le domaine de l'esthétisme qu'autre chose! Voici un exemple de ce que l'on peut ajouter :

```
plt.text(0,-80,r'$y=\frac{1}{2}x^3-3x^2+1$')
plt.xlabel='abscisse'
plt.ylabel='ordonnée'
```

4 Plusieurs courbes sur un graphe

Voici comment créer une figure qui accueillera le graphe, les courbes, les axes et un titre.

```
from matplotlib import pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np

x=np.arange(10) # on crée un tableau de 10 valeurs équirépartie commençant à 0.
plt.plot(x,x,'r',label='y=x') # Je crée une courbe d'équation y=x.
plt.plot(x,x*x,'k',label='x²') #y=x*x
plt.xlabel('axe des abscisses')
plt.ylabel('axe des ordonnées')
plt.title('Graphique de ouf')
plt.legend()
plt.show() # Affiche le graphe.
```

Il est intéressant d'avoir une légende des différents tracés. C'est possible avec la fonction **plt.legend()** qui s'utilise ainsi :

plt.legend(loc= 'position de la légende'(,autres options possibles)

On peut mettre comme position : 'best'(ou 0), 'upper right'(ou 1),'lower left'(ou 4) etc...

Il se peut que l'on veuille mettre la légende hors du cadre, on ajoute alors des options à *legend*.

```
from matplotlib import pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np
x=np.arange(10) # on crée un tableau de 10 valeurs équirépartie commençant à 0.
plt.figure("Légende hors du graphe") # On crée un graphe 1
plt.plot(x,x,'r',label='y=x')
plt.xlabel('axe des x')
plt.ylabel('axe des y')
plt.legend(loc=2,bbox_to_anchor=(1.01,1), borderaxespad=0.)
plt.tight_layout() #utile pour que la légende ne soit pas coupée
plt.show()
```

5 Multiplot

Il est souvent utile de pouvoir afficher plusieurs graphes pour les comparer. Il y a deux possibilités, soit on fait une fenêtre avec deux graphes ou plus, soit deux fenêtres ou plus avec un graphe soit un mixte des deux.

5.1 Deux fenêtres contenant une figure

```
from matplotlib import pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np
x=np.arange(10) # on crée un tableau de 10 valeurs équirépartie commençant à 0.
plt.figure('Exemple 1') # On crée une première fenêtre appelée exemple 1
plt.plot(x,x,'r')
plt.figure("Exemple 2") # une deuxième
plt.plot(x,x*x,'g+--')
plt.show()
```

5.2 Une fenêtre contenant deux figures

Cette partie est importante car elle est également valable quand on ne fait qu'un seul graphe sur une fenêtre graphique. C'est la façon la plus rigoureuse de procéder et celle que je vais adopter le plus souvent. Il y a deux possibilités, soit on utilise **subplot** soit **subplots**. Un « **s** »en plus et tout change.

5.2.1 subplot

```
from matplotlib import pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np

x=np.arange(10) # on crée un tableau de 10 valeurs équirépartie commençant à 0.
fig=plt.figure("découpage de fenêtre",figsize=(12,6),facecolor='w')#taille en pouce et couleur du fond
graphe1=fig.add_subplot(121) #graphe 1
graphe1.plot(x,x,'r')
graphe2=fig.add_subplot(122) #graphe 2
graphe2.plot(x,x*x,'g')
plt.show()
```

La méthode .add_subplot(xyz) subdivise la fenêtre en **x** ligne(s) et **y** colonne(s) soit $(x \times y)$ cases. Chaque case est numérotée, **z** est le numéro de la case où je veux mettre un graphique. La numérotation se fait **de gauche à droite puis de haut en bas, en commençant par 1**.

5.2.2 subplots

```
from matplotlib import pyplot as plt # le module pyplot est renommé plt
import numpy as np # le module numpy est renommé np

x=np.arange(10) # on crée un tableau de 10 valeurs équirépartie commençant à 0.
fenetre,(graphe1,graphe2)=plt.subplots(2,1,figsize=(12,6))
graphe1.plot(x,x,'r')
graphe2.plot(x,x*x,'g')
fenetre.show()
```

C'est un exemple simple. Vous en verrez de nombreux autres ensuite puisque j'utiliserai les deux méthodes (*subplots* et *subplot*).

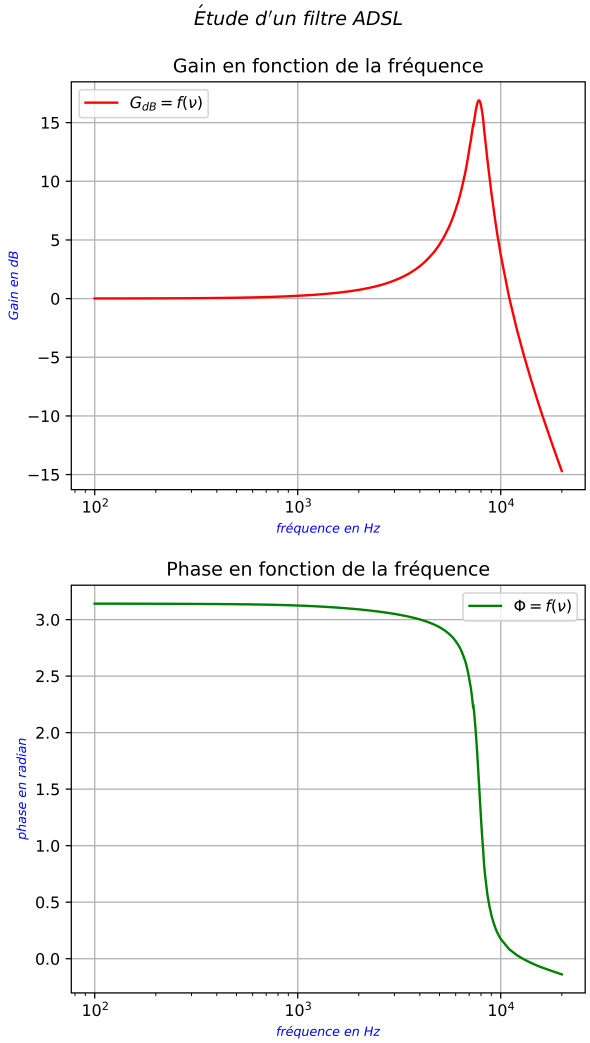
6 Le plot logarithmique

En classe préparatoire, on trace des fonctions de transfert et, pour cela, on doit utiliser du papier semi-logarithmique. Avec Python, si on veut faire un graphe logarithmique ou semi-logarithmique, trois fonctions de plot logarithmique sont à notre disposition :

- semilogx : axe des ordonnées linéaire et axe des abscisses logarithmique.
- semilogy : axe des abscisses linéaire et axe des ordonnées logarithmique.
- loglog : axes des ordonnées et des abscisses logarithmiques.

Application directe

À partir du fichier *FiltrADSL.txt*, représenter la phase et le gain en fonction de la fréquence. C'est un vrai fichier issu de mesures. Par contre, l'exportation très capricieuse rend l'exploitation un peu plus compliquée, je n'ai malheureusement pas pu faire mieux. Je ne vais pas trop m'en plaindre non plus, c'est plutôt formateur. Il faut commencer par retravailler ce fichier pour qu'il soit exploitable. Un programme d'une quinzaine de lignes suffira pour tout remettre dans l'ordre. Ensuite, faire le programme qui traitera les données.



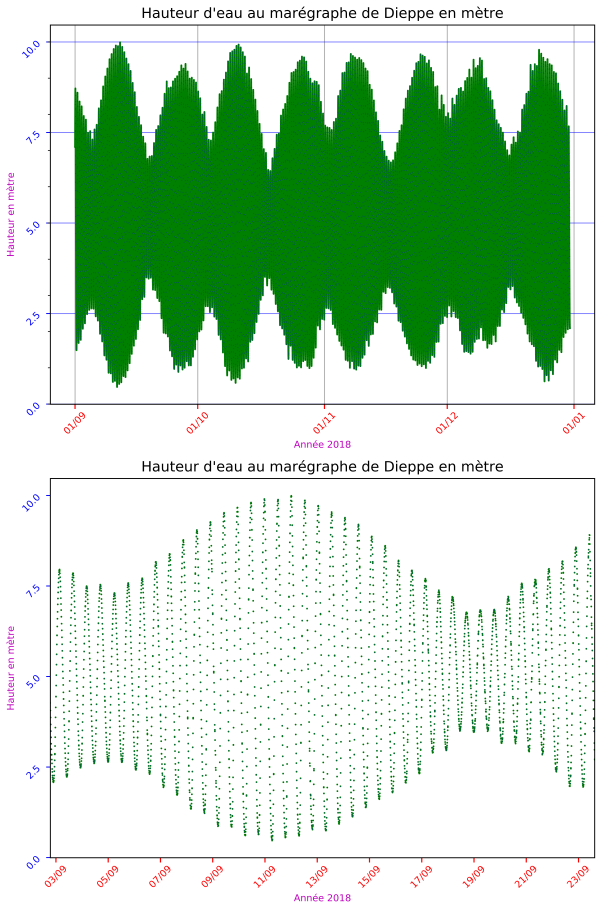
Ceci est le diagramme de Bode quand la ligne est ouverte en sortie, à vide. On pourrait refaire la même étude avec une ligne fermée, cela revient à mettre une résistance de 600 Ω en sortie.

7 Marée dieppoise

Le traitement des données en lui-même est court. Le plus long est le côté esthétique. On peut enlever l'appel à la fonction *embellissement* pour le constater. Je vous laisse le copier, l'exécuter, l'analyser :

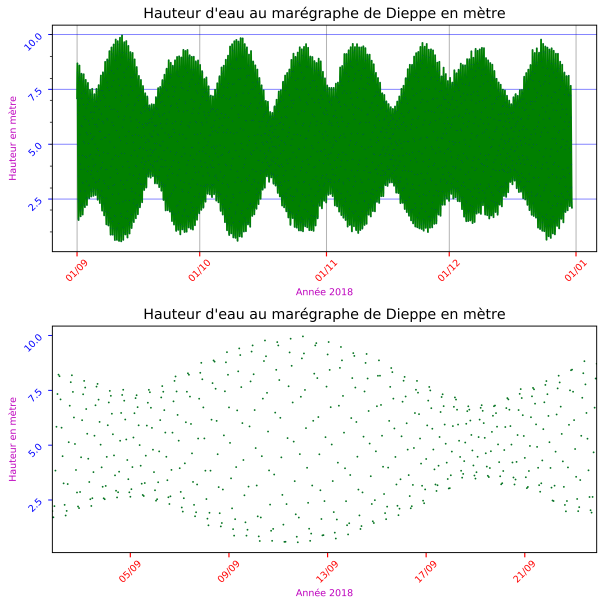
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime
from matplotlib.ticker import MultipleLocator

def embellissement():
    #juste de la déco à partir d'ici
    graphe.tick_params(axis='x',labelsize=8,
        labelrotation=45,colors='r',pad=1,length=4,
        width=1)
    graphe.tick_params(axis='y',labelsize=8,
        labelrotation=45,colors='b',grid_color='b',
         grid_alpha=0.5)
    graphe.set_title('Hauteur d\'eau au marégraphe
        de Dieppe en mètre',{'fontsize':12})
    graphe.set_xlabel('Année 2018',{'fontsize':8,'
        color':'m'})
    graphe.set_ylabel('Hauteur en mètre',{'fontsize'
        :8,'color':'m'})
    graphe.xaxis.set_major_locator(mdates.
        AutoDateLocator(minticks=3))
    graphe.yaxis.set_major_locator(plt.MaxNLocator
        (5))
    graphe.yaxis.set_minor_locator(MultipleLocator
        (1))
    graphe.xaxis.set_major_formatter(mdates.
        DateFormatter("%d/%m"))
    graphe.yaxis.grid(True, which='major') #minor or
         both
    graphe.xaxis.grid(True)

    graphe2.tick_params(axis='x',labelsize=8,
        labelrotation=45,colors='r',pad=1,length=4,
        width=1)
    graphe2.tick_params(axis='y',labelsize=8,
        labelrotation=45,colors='b',grid_color='b',
         grid_alpha=1)
    graphe2.set_title('Hauteur d\'eau au marégraphe
        de Dieppe en mètre',{'fontsize':12})
    graphe2.set_xlabel('Année 2018',{'fontsize':8,'
        color':'m'})
    graphe2.set_ylabel('Hauteur en mètre',{'fontsize
        ':8,'color':'m'})
    graphe2.set_xlim(temps[300],temps[2500])
    graphe2.xaxis.set_major_locator(mdates.
        AutoDateLocator(minticks=3))
    graphe2.yaxis.set_major_locator(plt.MaxNLocator
        (5))
    graphe2.xaxis.set_major_formatter(mdates.
        DateFormatter("%d/%m"))


temps=[]
dt=np.dtype('U19,f')


date,hauteur=np.loadtxt('mareeoctnovdec2018.txt',
     dtype=dt,delimiter=";",skiprows=14,usecols
     =[0,1],unpack=True)

for i in range (len(date)):
    item=date[i]
    temps.append(datetime.strptime(item,"%d/%m/%Y %H
        :%M:%S"))


fenetre,(graphe,graphe2)=plt.subplots(2,1,figsize
    =(12,6))
fenetre.canvas.set_window_title("La marée dieppoise
    ")
graphe.plot_date(temps,hauteur,'g.-',markersize=1,
    markerfacecolor='blue') #axe qui contient des
    dates
graphe2.plot_date(temps,hauteur,'g.',markersize=1,
    markerfacecolor='blue') #axe qui contient des
    dates

embellissement()

fenetre.tight_layout(pad=1,w_pad=5,rect
    =[0.05,0.05,0.95,0.95])
fenetre.show()
fenetre.savefig('MareeDieppe.png',dpi=1000)
```

Voici ce que vous obtenez en exécutant le programme :



Application directe

Quand vous ouvrez le fichier de données, vous remarquez que la dernière colonne montre que les données proviennent de deux sources différentes. À vous d'adapter le programme pour ne visualiser que les hauteurs issues de la source 4. Vous obtiendrez quelque chose de significativement différent pour le deuxième graphe. La fréquence d'échantillonnage commence a être limite.



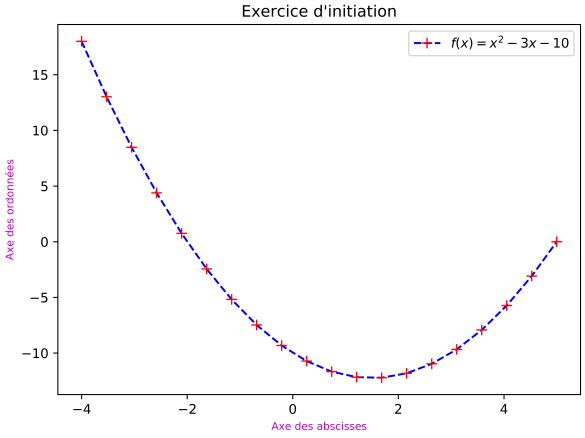
8 Applications

8.1 Reproduire

Tracer dans une fenêtre graphique la courbe représentative $\mathcal{C}$ de la fonction f définie telle que :

$$\begin{cases} \forall x \in [-4;8] \\ f(x) = x^2 - 3x - 10 \end{cases}$$

On affichera une grille sur le graphe, un titre, une légende. La courbe est bleue, en pointillé et les points sont des « + » rouges comme ci-après :



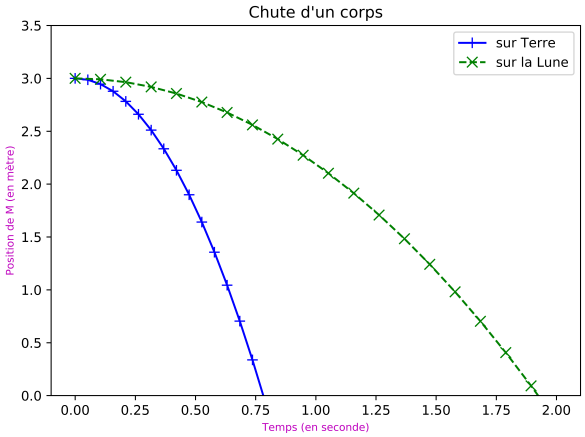
8.2 De la physique

Un corps, matérialisé par un point M de masse m, est lâché sans vitesse initiale d'une hauteur $h > 0$. On étudie le mouvement de M dans le référentiel $\mathcal{R}(0, x, y, z, t)$, supposé galiléen. L'axe $(0y)$ est définit verticalement par rapport au sol et dans le sens ascendant. Le point M n'est soumis qu'à la force de pesanteur. On note g la norme de l'accélération de pesanteur et on suppose $\vec{g}$ constant.

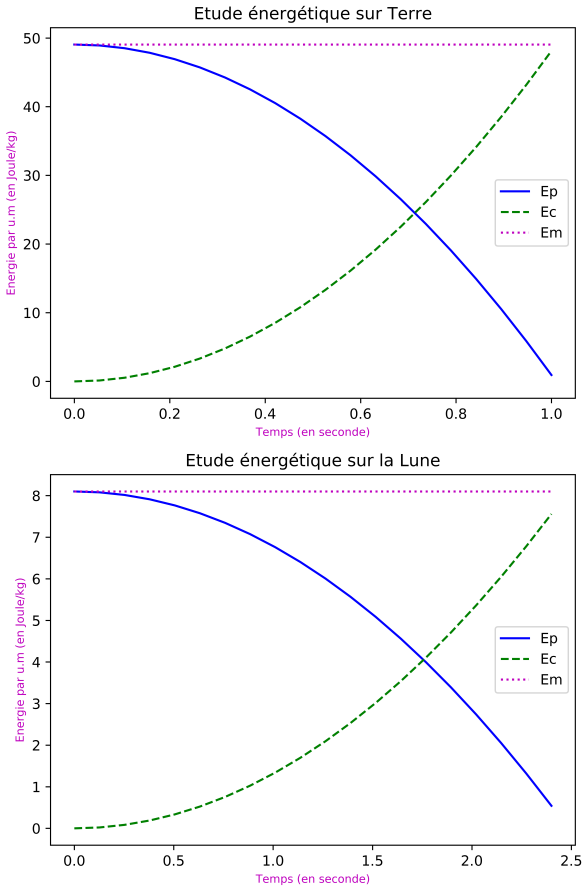
La deuxième loi de Newton s'écrit :

$$\sum \vec{F}_{ext} = m.\vec{a}_{G/\mathcal{R}}$$

• À vous d'obtenir un graphique de ce genre.



• Tracer à présent l'étude énergétique par unité de masse (énergie, potentielle, cinétique et mécanique) sur une même fenêtre divisée en deux. Sur la partie du haut, l'étude d'un corps sur Terre et sur la partie du bas, l'étude d'un corps sur la Lune.



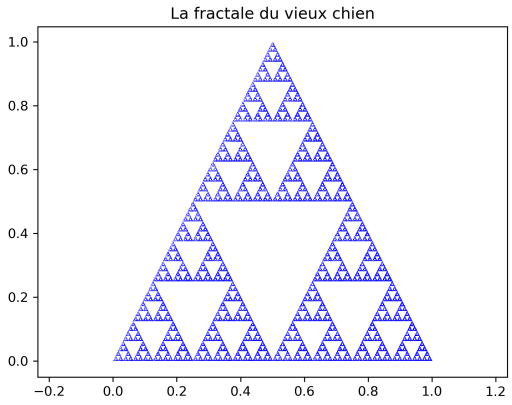
8.3 La fractale du vieux chien

Voici l'énoncé :

Trois prisonniers, enchaînés en des points éloignés de la cour d'un pénitencier, préparent un plan pour s'évader. À mi-chemin des deux premiers se trouve un chien de garde, tenant dans sa gueule le trousseau de clés. Chacun des prisonniers agite alors un bel os pour attirer le chien, qui choisit au hasard de se diriger lentement vers l'un d'entre eux. Mais, arrivé à mi-chemin, fatigué, le vieux chien s'arrête, creuse un trou et se couche. Les prisonniers agitent alors de plus belle leurs appâts : le chien se lève et avance à nouveau vers l'un des prisonniers au hasard. Arrivé à la moitié du chemin, il s'arrête, fait un nouveau trou et se couche... L'opération se répète un très grand nombre de fois.

Si chaque trou est représenté par un point, essayez de construire le graphe qui simule cette situation. Les trois prisonniers peuvent être aux coordonnées suivantes : $A(0;0), B(0.5;1)$ *et* $C(1;0)$ et le chien au milieu du segment $[AB]$

Avec 100 000 itérations, on obtient :



8.4 Cinétique chimique de réactions successives

On entend par réactions successives des réactions dont les produits de la première sont les réactifs de la seconde. Une telle suite de réactions met donc en jeu des molécules appelées intermédiaires réactionnels, dont une caractéristique est, le plus souvent, de ne pas pouvoir être isolés, du moins facilement. L'adjectif « successives » ne doit évidemment pas être pris dans le sens où les réactions auraient lieu l'une après l'autre.

L'exemple le plus simple est constitué de deux réactions élémentaires monomoléculaires successives :

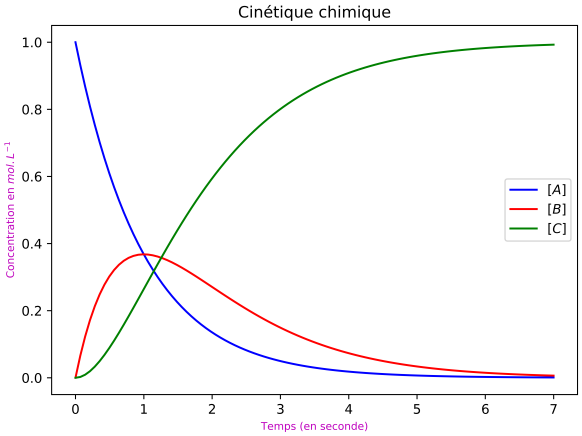
$$\begin{array}{lll} A & \longrightarrow & B \quad k_1 \\ B & \longrightarrow & C \quad k_2 \end{array}$$

Ces réactions sont d'ordre 1 : leurs vitesses sont proportionnelles à la concentration du réactif ou des réactifs. Les équations cinétiques s'écrivent donc :

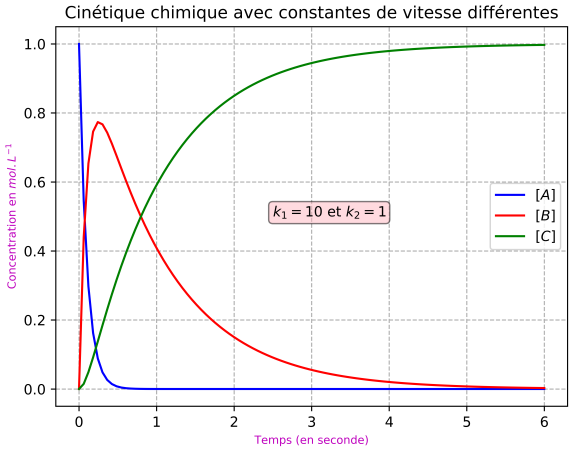
$$\begin{cases} \dfrac{d[A]}{dt} = -k_1.[A] \\ \dfrac{d[B]}{dt} = k_1.[A] - k_2.[B] \\ \dfrac{d[C]}{dt} = k_2.[B] \end{cases}$$

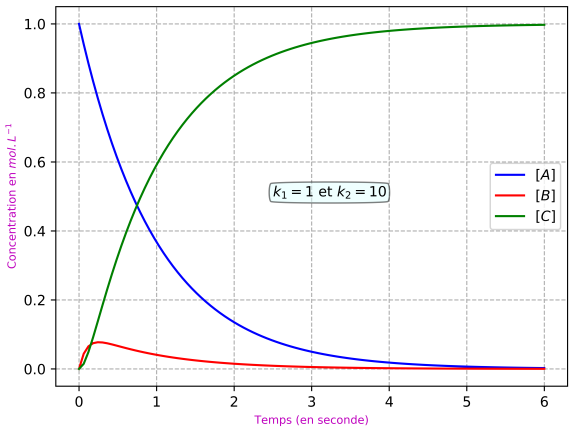
On veut représenter les différentes concentrations au cours du temps. On fixe les concentrations initiales telles que $[A]_0 = 1, [B]_0 = 0, [C]_0 = 0$ et $(k_1, k_2) = (1, 1)$ pour commencer.

Une possibilité :



Ensuite sur un autre graphe, représenter la même chose mais pour $(k_1, k_2) = (1, 10)$ et $(k_1, k_2) = (10, 1)$

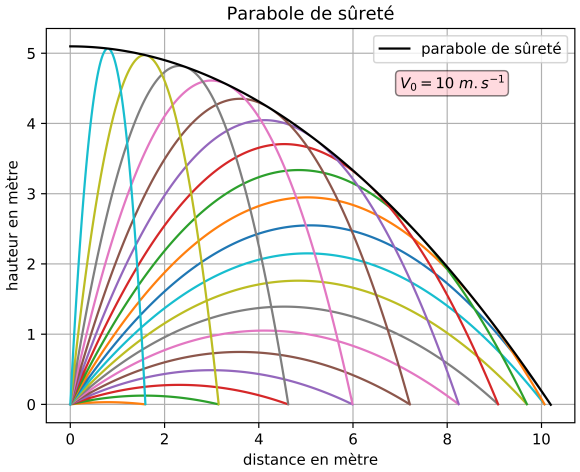




8.5 Parabole de sûreté

On jette un projectile M à partir du point O(0;0) avec une vitesse v_0 constante, mais faisant un angle variable avec l'horizontale. Si le projectile est soumis seulement à la gravitation, la trajectoire est connue : ce sera une parabole. On démontre même de façon classique que l'ensemble des trajectoires est « enveloppée » par une parabole, dite de sûreté. Cet exercice a pour objet la visualisation de cette parabole.

Représenter une vingtaine de trajectoires avec des angles qui varient régulièrement sur un intervalle $\left[0;\frac{\pi}{2}\right]$ et ajouter la parabole de sûreté.



9 Les vecteurs

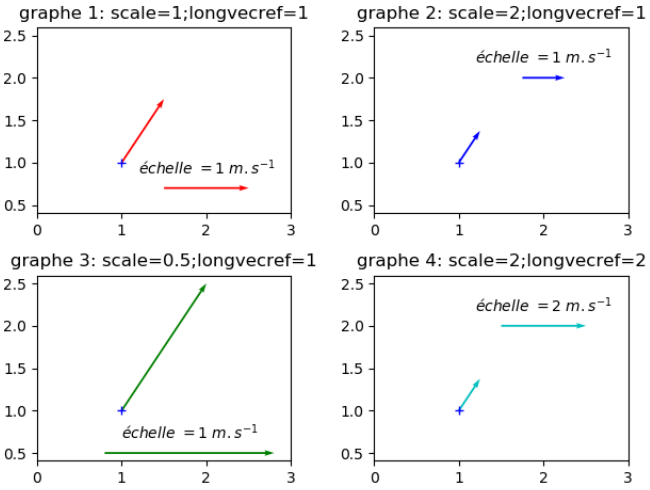
Pour tracer les vecteurs, nous utiliserons la méthode **quiver()** et pour la légende **quiverkey()**. Voici comment s'en servir avec un exemple simple, on trace le vecteur vitesse $\vec{V}$:

```
import matplotlib.pyplot as plt
x=1
y=1
VecX=1 # vitesse selon x de 1 m/s
VecY=1.5 # vitesse selon y de 1.5 m/s
longvecref=1
fig=plt.figure("Tracé d'un vecteur vitesse")
graphe=plt.subplot(111)
graphe.axis('equal') # repère orthonormé
graphe.set_xlim(0,3) # valeurs limites pour axe des x
graphe.set_ylim(0,3) # valeurs limites pour axe des y
graphe.plot(x,y,('b+')) # point aux coordonnées x et y
vecteur=graphe.quiver(x,y,VecX,VecY,angles='xy',scale=1,units='xy',color='r')
graphe.quiverkey(vecteur,0.8,2.1,longvecref,label=r'$échelle \ = 1 \ m.s^{-1}$',coordinates='data')
fig.show()
```

On remarque les deux mots utiles pour tracer des vecteurs. Tout d'abord **quiver** qui permet de tracer le vecteur. Il demande plusieurs arguments. En premier lieu, j'ai mis les coordonnées de départ du vecteur, x et y, ensuite les coordonnées du vecteur, éventuellement la couleur et l'échelle que j'ai choisie, 1 pour l'instant. On pourrait se contenter de **quiver** cependant la longueur du vecteur permet de connaître la valeur de la vitesse, il nous faut donc ajouter une référence. Pour cela, on utilise **quiverkey** qui a pour arguments l'instance créer par l'appel à **quiver** (j'ai choisi de la nommer *vecteur*), ensuite les coordonnées où on souhaite afficher notre échelle (fonction des axes x et y en précisant *coordinates='data'*) et la longueur du vecteur référence (valeur ou variable comme ici). Enfin, on ajoute un petit texte qui accompagne ce vecteur référence.

> **À savoir**
>
> J'ai passé sous silence **angles et units** car ces deux options sont obligatoirement à mettre avec la valeur **'xy'** pour avoir une représentation correcte des vecteurs vitesse et accélération.

Si on joue avec *scale* et *la longueur du vecteur référence*, on obtient des représentations différentes mais cohérentes.



Pour l'obtenir, j'ai utilisé ce code :

```
import matplotlib.pyplot as plt
x=1
y=1
VecX=0.5 # vitesse selon x de 1 m/s
VecY=0.75 # vitesse selon y de 1.5 m/s
longvecref=1
fig=plt.figure("Tracé d'un vecteur vitesse")
graphe=plt.subplot(221)
graphe.axis('equal') # repère orthonormé
graphe.set_title('graphe 1: scale=1;longvecref=1')
graphe.set_xlim(0,3) # valeurs limites pour axe des x
graphe.set_ylim(0,3) # valeurs limites pour axe des y
graphe.plot(x,y,('b+')) # point aux coordonnées x et y
vecteur=graphe.quiver(x,y,VecX,VecY,angles='xy',scale=1,units='xy',color='r')
graphe.quiverkey(vecteur,2,0.7,longvecref,label=r'$échelle \ = 1 \ m.s^{-1}$',coordinates='data')

graphe2=plt.subplot(222)
graphe2.axis('equal') # repère orthonormé
graphe2.set_title('graphe 4: scale=2;longvecref=1')
graphe2.set_xlim(0,3) # valeurs limites pour axe des x
graphe2.set_ylim(0,3) # valeurs limites pour axe des y
graphe2.plot(x,y,('b+')) # point aux coordonnées x et y
vecteur2=graphe2.quiver(x,y,VecX,VecY,angles='xy',scale=2,units='xy',color='b')
graphe2.quiverkey(vecteur2,2,2,longvecref,label=r'$échelle \ = 1 \ m.s^{-1}$',coordinates='data')

graphe3=plt.subplot(223)
```

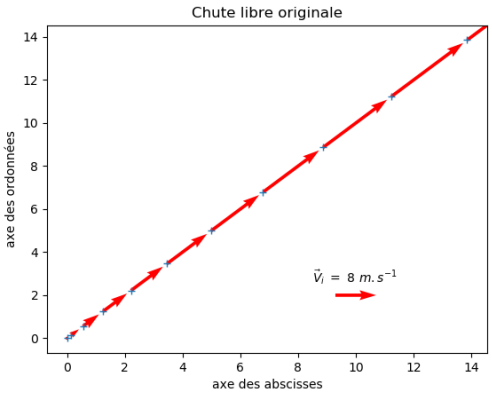
```
graphe3.axis('equal') # repère orthonormé
graphe3.set_title('graphe 3: scale=0.5;longvecref=1')
graphe3.set_xlim(0,3) # valeurs limites pour axe des x
graphe3.set_ylim(0,3) # valeurs limites pour axe des y
graphe3.plot(x,y,('b+')) # point aux coordonnées x et y
vecteur3=graphe3.quiver(x,y,VecX,VecY,angles='xy',scale=0.5,units='xy',color='g')
graphe3.quiverkey(vecteur3,1.8,0.5,longvecref,label=r'$échelle \ = 1 \ m.s^{-1}$',coordinates='data')

graphe4=plt.subplot(224)
graphe4.axis('equal') # repère orthonormé
graphe4.set_title('graphe 4: scale=2;longvecref=2')
graphe4.set_xlim(0,3) # valeurs limites pour axe des x
graphe4.set_ylim(0,3) # valeurs limites pour axe des y
graphe4.plot(x,y,('b+')) # point aux coordonnées x et y
vecteur4=graphe4.quiver(x,y,VecX,VecY,angles='xy',scale=2,units='xy',color='c')
graphe4.quiverkey(vecteur4,2,2,2*longvecref,label=r'$échelle \ = 2 \ m.s^{-1}$',coordinates='data')
fig.tight_layout() # utile pour organiser aux mieux les espaces entre graphes
fig.savefig('test.png')
fig.show()
```

Application directe

Pendant 2 secondes, un point matériel de masse m est en chute libre, dans un repère terrestre dans lequel l'accélération de pesanteur est définie par :

$$\vec{g} \left| \begin{array}{l} g_x = g \times cos\left(\frac{\pi}{4}\right) \\ g_y = g \times sin\left(\frac{\pi}{4}\right) \end{array} \right.$$

Représenter les vecteurs vitesse d'une dizaine de points.



10 Encore des applications

10.1 Analyse d'un pointage

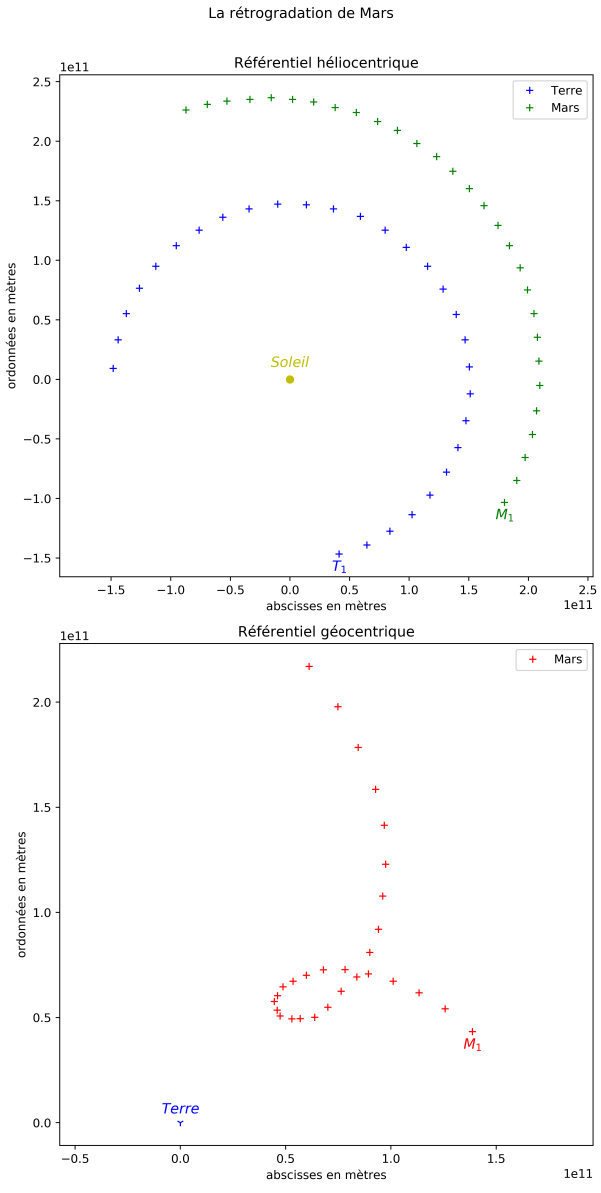
À partir du fichier *pointage.txt*, représenter les vecteurs vitesse et vecteurs accélération. Sauvegarder votre graphe.

10.2 Triche

À partir du fichier d'une chute libre que vous avez créé (mon_fichier.txt, représenter la variation du vecteur vitesse d'un système modélisé par un point matériel entre deux instants voisins ainsi que la somme des forces appliquées sur celui-ci .

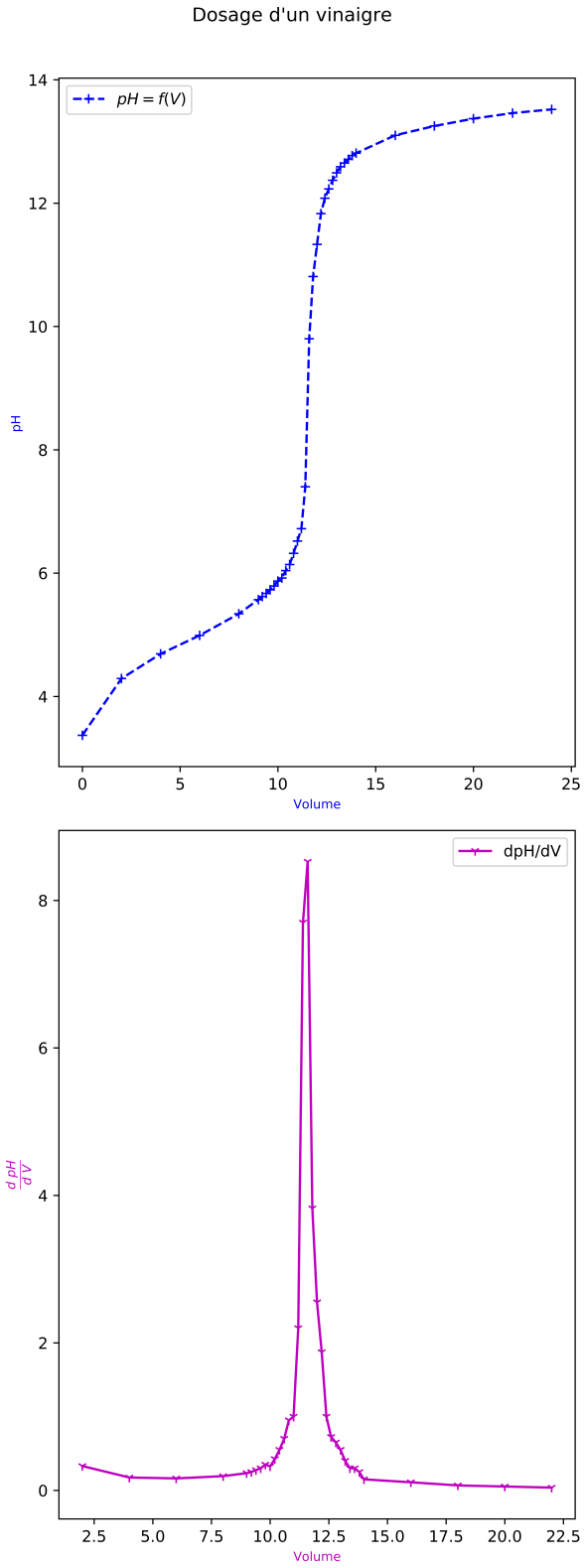
10.3 Rétrogradation de Mars

À partir de l'image *bitmap* représentant les positions de Mars et de la Terre à des intervalles de temps réguliers, représenter la rétrogradation de Mars.



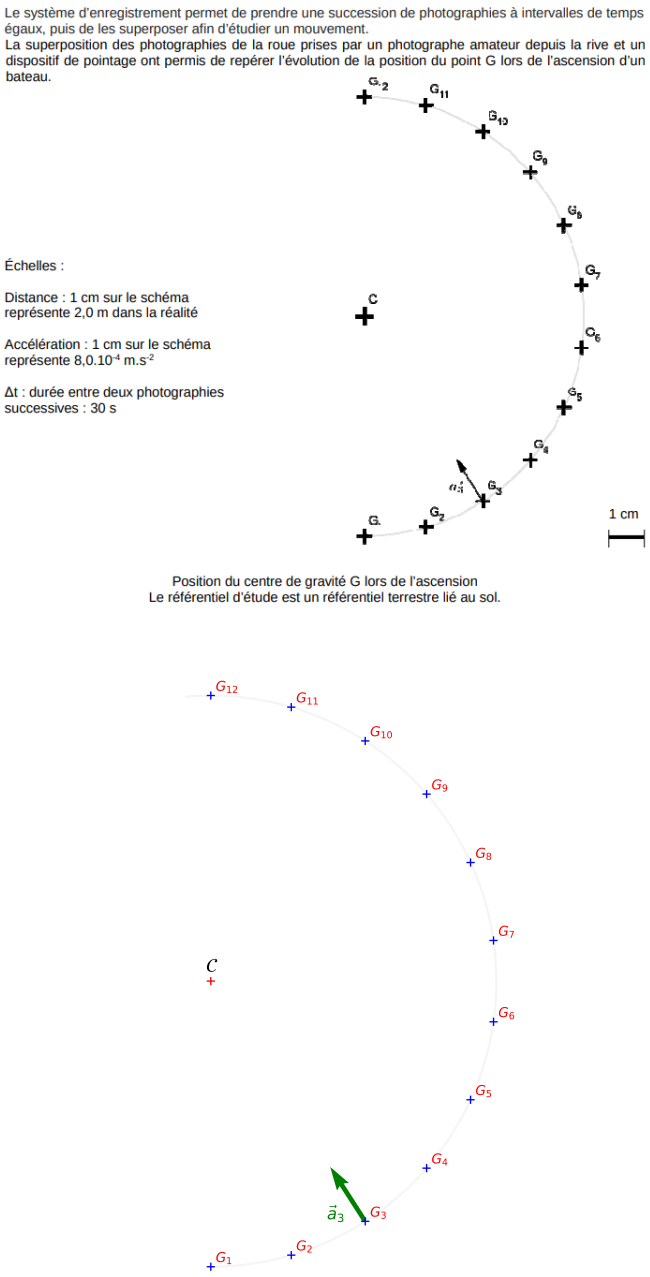
10.4 Dosage du vinaigre

À partir du fichier *DosageVinaigre.txt*, représenter les courbes $pH = f(Volume)$ et $\frac{d\,pH}{d\,V} = f(Volume)$



10.5 Imitation

Reproduire graphiquement un document trouvé sur internet que je trouve particulièrement moche. Il s'agit d'un extrait du bac S de 2016 en Amérique du Sud.



11 Modélisation

Dans le programme de seconde, il y a une partie où il faut *représenter un nuage de points associé à la caractéristique d'un dipôle et modéliser la caractéristique de ce dipôle à l'aide d'un langage de programmation.* Pour se faire, j'utilise le fichier *Ohm.txt* dans lequel l'intensité qui traverse un conducteur ohmique est relevée pour différentes tensions à ses bornes. On peut employer *.scatter* plutôt que *.plot* puisqu'on ne veut pas relier tous les points mais modéliser par une droite moyenne. Je vous présente, ici, trois façons différentes pour se faire :

11.1 linregress

```
from scipy.stats import linregress
import matplotlib.pyplot as plt
import numpy as np

U,I=np.loadtxt('Ohm.txt',skiprows=4,delimiter=';',
    unpack=True,usecols=(0,1))
fenetre,graphe=plt.subplots(1,1,figsize=(8,10))
fenetre.canvas.set_window_title("Loi d'Ohm")
i=I/1000 #pour avoir des ampères
graphe.scatter(i,U,marker='+',alpha=0.5)

(a,b,rho,_,_)=linregress(i,U)
graphe.plot(i,a*i+b,'g-',label=r'$modélisation \ :
    \ U=%.1f*I+%.3f$'%(a,b))
graphe.set_xlabel('Intensité dans la résistance en
    Ampère')
graphe.set_ylabel('Tension aux bornes de la ré
    sistance')
graphe.set_title('Loi d\'Ohm')
graphe.text(0,8,"corrélation=%.3f"%rho,style='
    italic')
graphe.legend()
fenetre.show()
```

La fonction *linregress* retourne 5 variables mais seules les 3 premières sont intéressantes (pente, ordonnée à l'origine et cœfficient de corrélation rho) d'ou **(a,b,rho,_,_)**. On utilise ensuite *a* et**b** pour tracer la droite.

11.2 polyfit

Méthode adaptée pour modéliser par une fonction polynomiale. Avec un degré 1, on arrive à une fonction affine.

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

U,I=np.loadtxt('Ohm.txt',skiprows=4,delimiter=';',
    unpack=True,usecols=(0,1)) # on crée un tableau
     de 10 valeurs équirépartie commençant à 0.
fenetre,graphe=plt.subplots(1,1,figsize=(8,10))
fenetre.canvas.set_window_title("Loi d'Ohm")
i=I/1000 #pour avoir des ampères
graphe.scatter(i,U,marker='+',alpha=0.5)

a,b=np.polyfit(i,U,1) #1 précise le degré
graphe.plot(i,a*i+b,'g-',label=r'$modélisation \ :
    \ U=%.1f*I+%.3f$'%(a,b))
graphe.set_xlabel('Intensité dans la résistance en
    Ampère')
graphe.set_ylabel('Tension aux bornes de la ré
    sistance')
graphe.set_title('Loi d\'Ohm')

graphe.legend()
fenetre.show()
```

11.3 curve_fit

Permet de modéliser en théorie n'importe quelle fonction. Par contre, il faut aider la modélisation en lui donnant des valeurs approchées pour chaque constante si cela ne marche pas. Dans notre cas, ce n'est pas la peine.

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

def func(x,a,b):
  """ Fonction qui donne l'équation de la modé
      lisation"""
    return a*x+b


U,I=np.loadtxt('Ohm.txt',skiprows=4,delimiter=';',
    unpack=True,usecols=(0,1)) # on crée un tableau
     de 10 valeurs équirépartie commençant à 0.
fenetre,graphe=plt.subplots(1,1,figsize=(8,10))
fenetre.canvas.set_window_title("Loi d'Ohm")
i=I/1000 #pour avoir des ampères
graphe.scatter(i,U,marker='+',alpha=0.5)
popt,pcov=curve_fit(func,i,U)
graphe.plot(i,func(i,*popt),'b--',label='fit a=%.3f
    , b=%.3f'%tuple(popt)) #insister sur l'étoile
graphe.set_xlabel('Intensité dans la résistance en
    Ampère')
graphe.set_ylabel('Tension aux bornes de la ré
    sistance')
graphe.set_title('Loi d\'Ohm')
graphe.legend()
fenetre.show()
```

Mais dans celui-ci, c'est nécessaire :

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
def func(x,a,b):
    return np.sin(a*x+b)

xData=np.linspace(0,20,50)
np.random.seed(5468) #on plante la graine pour
    avoir toujours les memes valeurs dans yBruit
yBruit=np.random.random(size=xData.size)*0.08 #crée
     des petites fluctuations
yData=func(xData,0.5,1)+yBruit

plt.plot(xData,yData,'g+')
popt,pcov=curve_fit(func,xData,yData,p0=[0.7,30]) #
    p0= On aide curve_fit

plt.plot(xData,func(xData,*popt),'b--',label='fit a
    =%.3f, b=%.3f'%tuple(popt)) #insister sur l'é
    toile
plt.legend()
plt.show()
```

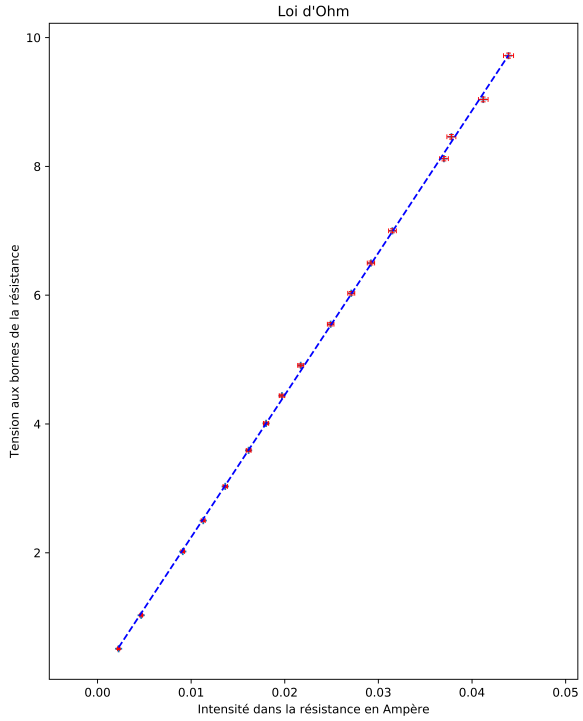
Dans cet exemple, on crée artificiellement des valeurs expérimentales avec le module random de numpy.

12 Barres d'erreur

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

def func(x,a,b):
    return a*x+b

```

```
8  U,I=np.loadtxt('Ohm.txt',skiprows=4,delimiter=';',
       unpack=True,usecols=(0,1)) # on crée un tableau
        de 10 valeurs équirépartie commençant à 0.
9  fenetre,graphe=plt.subplots(1,1,figsize=(8,10))
10 fenetre.canvas.set_window_title("Loi d'Ohm")
11 i=I/1000 #pour avoir des ampères
12 graphe.scatter(i,U,marker='+',alpha=0.5)
13 popt,pcov=curve_fit(func,i,U)
14 graphe.errorbar(i, U, yerr = 0.005*U+0.05,xerr
       =0.005*i+0.0005,fmt = 'none', capsize = 2,
       ecolor = 'b', elinewidth = 0.2, capthick = 0.2)
15 graphe.plot(i,func(i,*popt),'b--',label='fit a=%.3f
       , b=%.3f'%tuple(popt)) # insister sur l'étoile
16 graphe.set_xlabel('Intensité dans la résistance en
       Ampère')
17 graphe.set_ylabel('Tension aux bornes de la ré
       sistance')
18 graphe.set_title('Loi d\'Ohm')
19 graphe.legend()
20 fenetre.show()
```

Tout ce qui concerne les barres d'erreur se trouve à la ligne 14. *fmt*='None' est nécessaire pour que les points ne soient pas reliés un à un. Le reste concerne les paramètres des marqueurs.



Application directe

Modéliser la courbe obtenue grâce au fichier *pointage.txt* avec les deux dernières méthodes.