Université du Burundi
IPA/Mathématiques
Mastère en Sciences/ Enseignement des maths
Mastère II / 2024-2025, S1
Cours de Programmation par la pratique
Professeur Nibirantiza Aboubacar

Cours de programmation par la pratique

Plan du cours

Chapitre 1: Objets de base

Chapitre 2: Objets plus élaborés

Chapitre 3: Algorithmes et pratiques

Chapitre 4: Résoudre une équation différentielle ordinaire (EDO) avec Scipy

Chapitre 5. Calcul symbolique

Chapitre 6: Pratiques combinées

N.B. Dans ce cours, nous allons utiliser le langage de programmation python.

0. Introduction

0.1 Déroulement des leçons: toutes les leçons seront constituées par la pratique.

0.2 Des références générales sur python

Voici des références pour compléter ce didacticiel (qui est d'ailleurs construit à partir de ces références).

- Sur internet:
  - Page officielle de python avec son tutorial et la reference de la librairie standard.
  - Page officielle de la librairie scientifique Scipy pour python.
- Notes de cours
- Livres indiqués dans la section Références

0.3 Installation de Python sur votre ordinateur

On conseille d’installer python3 et l'environnement de programmation spyder ou Pycharm ainsi que les librairies scientifiques.

Par exemple utiliser la distribution Anaconda

0.3.0.1 Sur Linux, Ubuntu (recommandé)

Commandes dans un terminal:

```
sudo apt install python3-pip python3-dev
sudo apt install spyder
```

0.3.0.2 Sur Windows : Voir WinPython

0.3.0.3 Sur Mac-Os

0.3.0.4 Sur Android

Pour Android (téléphones, tablettes) il y a Qpython ou autres.

0.3.0.5 En général sur un navigateur

Utilisation de python directement dans un navigateur ou sur repl.

0.4 Utilisation de python avec le logiciel « spyder »
Lancement du logiciel spyder:
dans un terminal, lancer: spyder (ou spyder3)
Il apparaît un environnement de programmation pour le langage python appelé « spyder ».
premiers Tests à faire:

« python en mode interactif »:

En bas à droite, dans la fenêtre « Console IPython » écrire 2+2 et entrée. Il apparait le résultat: 4.

Pour écrire deux lignes d'instructions consécutives, il faut utiliser ctrl-entrée. Ecrire:

```
x=2
print (x+2)
```

et pour terminer tapez « entrée, entrée ». Il apparaît le résultat: 4.

Exemple plus ludique: dans la fenêtre « Console IPython » copier/coller le texte suivant (qui sera expliqué plus tard dans ce cours):

```
%matplotlib inline
from pylab import *
from numpy import *
X = arange(-6, 6, 0.2) # crée une liste de nombres Xi de 6 à 6 (exclu) par pas de 0.2
Y = sin(X) # crée une liste où chaque élément est Yi = sin ( Xi )
plot(X,Y) # dessin des valeurs de Yi en fonction des Xi
xlabel('x') # rajoute titre aux axes
ylabel('y=sin(x)')
show() # montre le dessin
```

Résultat s’affiche immédiatement en image

- **« python en mode programme »:** Dans la fenêtre de gauche qui est un éditeur ordinaire, et qui édite le fichier « temp.py », rajouter les lignes:

```
x=2
print (x+2)
```

et exécuter le programme par Menu/Execution (F5). Il apparaît le résultat dans la console: 4

Programmes avec graphisme:
dans ce cas il est conseillé dans spyder dans le menu **Outils/Préférences/ConsoleIPython/Graphiques/Sortie** : de choisir le mode automatique.
(et relancer la fenetre Ipython)

0.4.1 Lancement de votre programme python dans une fenêtre de commandes
Voici comment exécuter votre programme **temp.py** ci-dessus dans une fenêtre de commandes.

Ouvrir un terminal (fenêtre de commandes)
Aller dans le répertoire qui contient votre programme temp.py
Ecrire:

```
python3 temp.py
```

Dans la suite de ce didacticiel, vous pourrez écrire les lignes de codes proposées en mode interactif ou en mode programme. Pour recopier le(s) code(s) proposé(s), vous pouvez faire « copier/coller »

0.5 (supplément) Utilisation de python avec le logiciel « Jupyter »
Si vous avez choisit de travailler avec « spyder » comme ci-dessous, vous pouvez ignorer cette Section.
Références: sur datacamp
Lancement du logiciel Jupyter

dans un terminal, lancer: anaconda-navigator
Cliquer sur « Launch JupyterLab ». Il apparaît un environnement de programmation pour le langage python appelé « JupyterLab ».
Cliquer sur l'icone « NoteBook/Python3 »

- Mode commandes: si le bord est bleu.

- Touches: a: crée entrée au dessus, b: crée entrée dessous, dd: détruit l'entrée.
- Entrée: passe en mode éditeur.

- Mode éditeur: si le bord est vert.

- Touche escape: passe en mode commandes.

Chapitre 1: Objets de bases

1.1 Nombres entiers et nombres à virgule. Entrée et sortie à l'écran.

Pour créer une variable, il suffit de lui affecter une valeur. Le texte après le symbole # est un commentaire et est ignoré par l'ordinateur. Ecrire:

```
x = 1 # met le chiffre 1 dans la variable x
print (x)
```

résultat: 1

On voit ici une différence importante avec d'autres langages : il n'est pas nécessaire de déclarer a priori le type de la variable x (entier 'int' (integer), nombre flottant 'float', chaine de caractère 'string',...). Pourtant ce type existe bien, si vous écrivez ensuite :

```
print (type(x))
```

résultat: <type 'int'>

Python a donc automatiquement décidé que x était un entier (int = integer). Ré-essayez avec d'autres initialisations : x=1.0 (nombre à virgule flottante ou float), ou x = 'bonjour' (chaine de caractère ou string),...
On peut affecter plusieurs variables à la fois. Ecrire:

```
x,y = 1, 2
print (x,y)
```

résultat: 1 2
ou écrire:

```
    x = y = 1
    print (x)
    print (y)
```

résultat:

```
    1
    1
```

Remarque: dans l'exemple précédent, pour que les deux résultats soient sur la même ligne (sans saut de ligne), il faut rajouter , end=' ' à la fin de la première ligne. Ecrire:

```
    x = y = 1
    print (x,end=' ')
    print (y)
```

résultat: 1 1

Pour demander à l'utilisateur le contenu d'une variable, écrire:

```
    x = input('x=? ') # demande un nombre et le met dans la variable x
    print (x)
    print (type(x))
    y = int(x) # convertit x qui est de type 'str' en type 'int'
    print (y)
    print (type(y))
```

résultat:

```
    x=? 2
    2
    <type 'str'>
    2
    <type 'int'>
```

Remarque 1.1.1. On observe dans l'exemple précédent qu'une variable obtenue par l'instruction input() est de type 'string'. Si on veut la convertir en nombre entier il faut appliquer la fonction int(). Attention : majuscules et minuscules sont différenciées par python, i.e. ma_variable, MA_VARIABLE et Ma_Variable sont trois variables distinctes. Ecrire:

```
ma_variable = 1
Ma_variable = 2
print (ma_variable)
```

résultat:

```
1
```

Pour que la console oublie la valeur de x (et de toutes les variables) il faut écrire: %reset

1.2 Chaînes de caractères

Une chaine de caractère est un tableau contenant des caractères. On peut ainsi construire des mots et des phrases et les manipuler. La numérotation des « cases » commence à zéro.

Ecrire:

```
nom='Vincent'
print (nom)
print (nom[2:6])    # extrait les caractères 2 (inclus) à 6
(exclu),
```

Résultat:

```
Vincent
ncen
```

Remarque 1.2.1. - le premier caractère est donc nom[0] et contient 'V'.

Il est possible d'additionner des chaînes de caractères.

Ecrire:

```
print ('Programmation' + ' Scientifique')
```

résultat:

```
Programmation Scientifique
```

On peut obtenir la longueur d'une chaîne de caractère avec la fonction len(). Ecrire:

```
len('toto')
```

Résultat: 4

1.2.1 Formatage

Il est possible d'écrire une chaîne de caractères en insérant des codes spécifiques dans la chaîne (de la même manière qu'avec la fonction printf en langage C): (%s pour une chaîne de caractères, %d pour un entier, %f pour un nombre à virgule, et optionnellement le nombre de caractères total et après la virgule: %4.2f veut dire 'nombre représenté sur 4 caractères, avec 2 chiffres après la virgule'), suivie de % avec entre parenthèses toutes les variables dont les valeurs sont à insérer. Ecrire:

```
age=5
nom="Vincent"
taille=1.73
maChaine="%s a %d ans et mesure %4.2f m" % (nom,age,taille)
print (maChaine)
```

Résultat: Vincent a 5 ans et mesure 1.73 m

Pour avoir la liste des fonctions utilisables avec des chaînes de caractères, écrire:

```
help(str)
```

1.3 Conversion de variables entre différents types

On peut convertir une variable d'un type à un autre. Ecrire:

```
x=float(1) # nombre réel à partir d'un entier
print (type(x))
```

résultat: <type 'float'>

Ecrire:

```
x=float('1.0') # nombre réel à partir d'une chaîne de caractère
print (type(x))
```

résultat: <type 'float'>

Ecrire:

```
print (x)
```

Résultat: 1.0

Exercice 1.3.1. Ecrire un programme qui demande le nom et l'âge d'une personne. En retour elle affiche l'année de naissance. Par exemple:

```
ton nom=? blaise
ton age=? 25
Salut blaise , tu es né en 1993 ?
```

1.4 Opérations mathématiques

1.4.1 opérations de base

1.4.1.1 Opération de division et puissance

Écrire: `print (7/2, 7//2, 3**2)`

Résultat: 3.5 3 9

Noter que la première opération / a donné la division décimale et la deuxième opération // a donné la division Euclidienne (le résultat est un entier).

1.4.1.2 Modulo

Le « modulo »: n%q est le reste de la division euclidienne de n par q :

Ecrire:

```
print (4%3, -1%3)     # 4 modulo 3,    (-1) modulo 3
print (7.5%3.0, -2.5%3.0)    # 7.5 modulo 3.0, (-2.5) modulo 3.0
```

résultat: 1 2 1.5 0.5

1.4.1.3 Fonctions mathématiques

Pour calculer une racine carrée: (le texte après le signe # est un commentaire)

```
from math import * # cela importe le module math
sqrt(2.)
```

résultat: 1.4142135623730951

Pour avoir la liste des fonctions mathématiques disponibles écrire:

```
help()
```

```
    math
```

Remarque : pour importer un module (tel que **math)**, il y a deux manières de le faire, avec (a) “**import math**” ou (b) “**from math import ***”.
- Dans le cas (a), les fonctions du module math s'utiliseront en les faisant précéder de “math.” (exemple : “**math.sin(math.pi/3.0)”)**.
- Dans le cas (b), il n'est pas nécessaire d'utiliser le préfixe, i.e. python comprendra directement “**sin(pi/3.0)**”. A priori, la première notation n'est à utiliser que si il y a un risque de conflit de noms entre fonctions, si par exemple vous souhaitez redéfinir par ailleurs la fonction sin().

1.4.2 Nombre aléatoire
On peut regarder la documentation sur random.

Pour choisir un nombre à virgule au hasard entre 0 et 1, écrire:

```
    import random
    x = random.random()
    print (x)
```

résultat: 0.9565461152605507

Pour choisir au hasard un nombre entier entre 1 et 6 compris, écrire:

```
    import random
    x = random.randint(1, 6)
    print (x)
```

Résultat: 3

Exercice 1.4.1. Ecrire un programme qui demande un nombre entier x à l'utilisateur et en retour affiche un entier choisit au hasard entre 0 et x. On souhaite le résultat:
x=? 3
2
Solution:

```
    import random
    x = input('x=? ')
    print random.randint(0, x)
```

1.4.3 Nombres complexes

Il est possible de manipuler des nombres complexes en Python, le nombre complexe i étant noté 1j . Ecrire:

```
a=2+1j
type(a)
```

résultat:

```
<type 'complex'>
```

Ecrire:

```
b=3.5+4j
print (a+b)
```

résultat:

```
(5.5+5j)
```

Ecrire:

```
print (a*b)
```

résultat:

```
(3+11.5j)
```

Pour utiliser des fonctions mathématiques sur des complexes, il faut importer le module **cmath** en plus du module **math**. Par exemple pour calculer exp(i π) qui est égal à 1 , écrire:

```
from math import *
from cmath import *
print (exp(1j*pi))
```

résultat:

```
(-1+1.2246467991473532e-16j)
```

1.4.4 (supplément) Nombre en base 2, binaire.

Les 2 symboles utilisés en base 2 sont: 0,1 . Pour indiquer qu'un nombre est écrit en base 2, on précède son écriture du symbole 0b. Ecrire:

```
#... Declaration et Affichage en base deux:
A=0b1001 # declaration en base 2 (prefixe 0b)
print 'A=' , A , ' en binaire = ' , bin(A)

#.. decalages de bits
b= A<<4 # equivalent: b= A * (2**4)
print ' b =', b, ' = ', bin(b)
```

```
c = b>>3 # equivalent: c= b / (2**3)
print ' c =', c, ' = ', bin(c)

#.. operations bits a bits
B = 0b1;
d = A | B # ou inclusif bit a bit cad 1|1 donne 1
e = A ^ B # ou exclusif bit a bit cad 1^1 donne 0
f = A & B # et bit a bit
g = ~A # non A, inversion des bits

print ' B=', B, ' = ', bin(B)
print ' A|B = ', d, ' = ', bin(d)
print ' A^B = ', e, ' = ', bin(e)
print ' A&B = ', f, ' = ', bin(f)
print ' ~A = ', g, ' = ', bin(g)
```

Résultats:

```
A= 9 en binaire = 0b1001
b = 144 = 0b10010000
c = 18 = 0b10010
B= 1 = 0b1
A|B = 9 = 0b1001
A^B = 8 = 0b1000
A&B = 1 = 0b1
~A = -10 = -0b1010
```

Référence pour approfondir:
https://wiki.python.org/moin/BitManipulation

1.4.5 (supplément) Nombre en base 16 (hexadécimale)

Les 16 symboles utilisés en base 16 sont:
0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f . Pour indiquer qu'un nombre est écrit en base 16, on précède son écriture du symbole **0x.** Ecrire:

```
nombre = 0xb # déclaration en base 16
print (nombre) # affichage en base 10
```

Résultat (écriture en base 10):   11

1.5 Listes

Un type particulièrement intéressant est la liste qui permet de construire et manipuler une liste d'objets divers. Référence sur le type list.
Ecrire:

```
ma_liste = [1,4, 'ABC']
print (ma_liste)
```

résultat:

```
[1, 4, 'ABC']
```

Remarque 1.5.1. La numérotation des objets de la liste commence à 0 (i.e. si il y a n éléments dans ma_liste, ils sont accessibles par ma_liste[0]...ma_liste[n-1]).
Ecrire:

```
ma_liste = [1,4, 'ABC']
print (ma_liste[2])
```

résultat: ABC

Ecrire:

```
ma_liste = [1,4, 'ABC']
print (type(ma_liste))
print (type(ma_liste[2]))
```

résultat:

```
<type 'list'>
<type 'str'>
```

1.5.1 Opérations sur les listes

Il est possible de modifier une liste existante. Écrire (le texte après le signe # est un commentaire)

```
L = [1,3,7,5]
L[1]=42 # le 2ème élément de MaListe vaut maintenant 42
print ('L1=', L)

L.append(45) # ajoute l'objet “45” à la fin de la liste
print ('L2=', L)
```

```
    L.insert(2, 33)  # insère (rajoute) l'élément 33 à l'indice 2
dans MaListe
    print ('L3=', L)

    del L[1]    # enleve element de la case 1
    print ('L3a=', L)

    res = L.pop(1)     # enleve element de la case 1 et le donne
    print ('L3b=', L, ' res=',res)

    L.remove(45)     # enlève le première élément trouve égal a 45
    print ('L3c=', L)

    L.reverse()     # inverse l'ordre des éléments de la liste
    print ('L4=', L)

    L.sort()       # trie les éléments de la liste par ordre
croissant
    print ('L5=', L)

    L2 = []       # cree une liste vide
    print ('L6=', L2)

    L3 = list(L)      # copie dans une autre liste
    L3[0] = 0
    L4 = L       # ce n'est pas une vraie copie: liste4 est le meme
objet que MaListe
    L4[0] = -1
    print ('L7=', L3)
    print ('L8=', L)
```

résultat:

```
    L1= [1, 42, 7, 5]
    L2= [1, 42, 7, 5, 45]
    L3= [1, 42, 33, 7, 5, 45]
    L3a= [1, 33, 7, 5, 45]
    L3b= [1, 7, 5, 45] res= 33
    L3c= [1, 7, 5]
    L4= [5, 7, 1]
    L5= [1, 5, 7]
    L6= []
    L7= [0, 5, 7]
```

```
L8= [-1, 5, 7]
```

On peut obtenir la longueur d'une liste (le nombre d'objets dans la liste) avec la fonction len():

```
MaListe = [1,3,7,5]
len(MaListe)
```

résultat: 4

Il est possible de concaténer deux listes avec l'opérateur + :

```
liste1 = [1,'ab',3.5]
liste2 = ['toto',57]
print (liste1 + liste2)
```

résultat: [1, 'ab', 3.5, 'toto', 57]

On peut également supprimer un élément d'une liste avec la fonction del() :

```
MaListe=[1, 'ab', 3.5, 'toto', 57]
print (MaListe)
del(MaListe[1])
print (MaListe)
```

résultat:

```
[1, 'ab', 3.5, 'toto', 57]
[1, 3.5, 'toto', 57]
```

Pour créer des listes particulières de nombres:

```
list(range(2,7)) # liste des entiers consécutifs de 2 (inclu) à 7 (exclu)
```

résultat: [2, 3, 4, 5, 6]

Si on commence de 0 il est plus simple d'écrire:

```
list( range(7)) # liste de 0 (inclu) à 7 (exclu)
```

résultat: [0, 1, 2, 3, 4, 5, 6]

N.B. Pour certains environnement , on peut simplement écrire **range(2,7) pour la liste des entiers consécutifs de 2 (inclu) à 7 (exclu) et**
range(7) pour la liste de 0 (inclu) à 7 (exclu)

Les listes sont des objets particulièrement utiles. Nous verrons que les boucles utilisent souvent une itération sur des éléments d'une liste. La liste complète des fonctions utilisables pour une liste peut être obtenue avec l'aide help(list).

1.5.2 Listes et hasard

Pour choisir au hasard un élément parmi une liste donnée, écrire:

```
    import random
    x = random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])
    print (x)
```

Résultat: 'k'

Pour mélanger au hasard une liste donnée écrire:

```
    import random
    liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
    random.shuffle(liste)
    print (liste)
```

résultat: ['p', 'k', 'w', 'z', 'i', 'b', 'a']

Exercice 1.5.2. Ecrire un programme qui choisit au hasard un élément parmi une liste donnée, comme le programme ci-dessus, mais en utilisant les fonction len() et random.randint().

Solution:

```
    import random
    liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
    print (liste[random.randint(0,len(liste)-1)])
```

1.6 Boucles et opérateurs
1.6.1 opérateurs ==, and, or , not

L'opérateur == permet de comparer deux objets. Ecrire:

```
1==2     # est-ce que 1 est égal à 2 ?
1==1
```
résultat:
```
False
True
```

Les autres opérateurs de comparaison sont <, >, <=, >=,!= (ce dernier signifie différent). Par convention, l'entier 0 correspond à False et les entiers non nuls correspondent à True.
L'opérateur not(x) renvoie la négation (au sens booléen, vrai/faux) d'une expression :

```
not(True)
not(False)
not(0)
```
résultat:
```
False
True
True
```

and et or permettent de réaliser les opérations logiques usuelles (attention il ne s'agit pas d'opérations bit à bit, il ne faut donc utiliser and et or que pour des tests booléens vrai/faux):

```
1 and 0
1 and 1
1 or 0
```

résultat:

```
0
1
1
```

1.6.2 L'instruction if...elif..else

Remarque 1.6.1. “elif” signifie “else if”.
On peut exécuter une instruction conditionnelle avec if. Écrire:

```
x=1
print ('x=',x)
if x==2:                 # est-ce que x est égal à 2 ?
```

```
        print ('x est égal à 2')  # notez les espaces en début de
ligne (indentation)
    elif x==3:
        print ('x est égal à 3')
    else:
        print ("x n'est pas égal à 2")
        print ('ni à 3')
```

résultat:

```
    x=1
    x n'est pas égal à 2
    ni à 3
```

Les espaces en début de ligne (indentation) jouent un rôle important : tant que l'espace reste le même en début de ligne, on reste dans le même “niveau” d'éxécution, de même que dans d'autres langages on utilise des accolades {} pour délimiter les instructions à réaliser. On peut ainsi réaliser des instructions imbriquées en augmentant le niveau d'indentation :

```
    x=12
    if (x%2)==0:            # est-ce que x modulo 2 est égal à 0 ?
        if (x%3)==0:
            if (x%7)==0:
                print ('x est multiple de 2,3 et 7 !')
            else:
                print ('x est multiple de 2 et 3, mais pas de
7...')
        else:
            print ('x est multiple de 2 mais pas de 3')
    else:
        print ('x n'est pas multiple de 2')
```

résultat:

```
    x est multiple de 2 et 3, mais pas de 7…
```

1.6.3 La boucle while

On peut réaliser une série d'instructions tant qu'une condition est réalisée :

```
    x=7
```

```
    while x>=1:
        print (x,' >=1 ! On continue…' )
        x = x-1                 # on pourrait aussi écrire x -= 1
```

résultat:

```
    7 >=1 ! On continue...
    6 >=1 ! On continue...
    5 >=1 ! On continue...
    4 >=1 ! On continue...
    3 >=1 ! On continue...
    2 >=1 ! On continue...
    1 >=1 ! On continue...
```

1.6.4 Les boucles for

Une boucle for s'écrit de la manière suivante :

```
    for i in [2,3,7]:    # De manière générale : for variable in liste:
        print (i)
```

résultat:

```
    2
    3
    7
```

Comme il est fastidieux d'écrire un grand nombre de valeurs, on peut utiliser la fonction range() qui renvoie une liste d'entiers consécutifs:

```
    for i in range(2,5):
        print i
```

résultat:

```
    2
    3
    4
```

Remarque: range(5) est équivalent de range(0,5)

Un autre exemple de boucle : boucle imbriquée

Exercice 1.6.2.Écrire un programme qui affiche les tables de multiplications de 1 à 5.

```
for x in range (1,6):
       for y in range (1,6):
              print ("%d x %d = %d" % (x,y,x*y))
       print ()
```

Résultat:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10

3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15

4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
```

Exercice 1.6.3. « La Suite de Syracuse ». Soit un nombre entier $x_0 \geq 1$. Par récurrence, on définit x_{t+1} à partir de x_t par:

$$x_{t+1} = 1/2\ x_t \text{ si } x_t \text{ est pair ou } x_{t+1} = 1/2(\ 3x_t+1) \text{ si } x_t \text{ est impair}$$

Observer que la valeur de départ $x_0 = 1$ donne la suite infinie périodique 1,2,1,2,1, …

La valeur de départ x_0 =7 donne la suite 7,11,17,26,13,20,10,5,8,4,2,1,2,1,2,1, …

La fameuse **conjecture de Syracuse** non démontrée à ce jour est que partant de toute valeur x_0 la suite arrive forcément au bout d'une date T (qui dépend de x_0) à la suite périodique 1,2,1,2, …

Référence : https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

Ecrire un programme qui demande x_0 à l'utilisateur et affiche les valeurs de la suite (x_t) t jusqu'à ce que la valeur x_t =1 soit atteinte. Exemple de résultat:

x_0=?7
    11 , 17 , 26 , 13 , 20 , 10 , 5 , 8 , 4 , 2 , 1,

Solution:

```
x0 = int(input('Entrer la valeur x0=')) # ou simplement ecrire
x0=int(input('x0= ?'))
while(x0 != 1):
      if(x0%2 == 0):
          x0=x0/2
      else:
         x0=(3*x0+1)/2
      print (x0,',',end=' ')
```

1.7 Mesurer le temps : Les dates

Objectifs : - Comprendre l'utilisation des timestamp ;
- Passer d'une date à un timestamp et inversement.
- Manipuler les dates et les heures

Mise en situation :
Peu importe le format ou la langue de la date, il existe une manière universelle de l'exprimer : en nombre de millisecondes.

L'epoch désigne la date à partir de laquelle est mesuré le temps par les systèmes d'exploitation. Il en existe plusieurs, le plus connu étant l'Unix Epoch (aussi appelé POSIX time).
Il est utilisé par la plupart des langages (JavaScript, Python, C, C++, PHP, etc.) et par les systèmes basés sur Unix.

L'Unix Epoch est fixé au 1er janvier 1970 et est utilisé pour calculer le timestamp dans ces langages.

À retenir:
Les dates sont souvent représentées comme un nombre de millisecondes écoulées depuis le 1er janvier 1970. Un tel nombre est appelé timestamp, et il est possible de convertir un timestamp en date et inversement.

En Python, la fonction time() renvoie le nombre de secondes écoulées depuis epoch (le point où le temps commence).

Ecrire:

```
    from numpy import *
    import time

    t0 = time.time() # mesure de la date actuelle

    for p in range(100000): # un calcul quelconque
      y = cos(p)
    t1 = time.time() # mesure de la date
   print ('Duree: t1-t0 = %6.3f s' %(t1-t0)) # affiche (t1-t0) avec
une précision de 3 chiffres après la virgule, voir section 1.2
```

résultat:

```
                  Duree: t1-t0 = 0.116 s
```

Une librairie Python permet d'obtenir la date et le temps actuels. Voici comment l'utiliser.
Tous les langages possèdent des méthodes ou des classes pour gérer les dates et les temps. Il est donc possible à tout moment d'obtenir la date et le temps actuels. Dans le langage Pyton, il faut utiliser la librairie datetime.
La librairie datetime fournit des méthodes liées à la manipulation des dates et des temps. Pour obtenir la date et le temps actuels, on utilise la méthode now().
Exemple:

```
            from datetime import datetime
            datetime.now()

             Résultat :
               datetime.datetime(2023, 4, 2, 17, 51, 17, 779657)
```

Si vous ne souhaitez obtenir que l'heure, il faut ensuite appeler la méthode time().

Exemple:

```
datetime.now().time()
```

Résultat :

datetime.time(18, 0, 35, 575468)   # time(heure, minute, seconde, microseconde)

Pour obtenir une date plus lisible et compréhensible, vous pouvez utiliser la fonction str() qui convertit la date en chaîne de caractères.

Exemple:

```
from datetime import datetime
str(datetime.now())
```

Resultat : '2023-04-02 17:59:15.720560' # datetime(annee, mois, jour, heure, minute, seconde, microseconde)

Pour plus d'informations, aller par exemple sur :
https://www.gladir.com/CODER/PYTHON/date-et-heure.htm

1.8 Fonctions

Nous avons déjà utilisé les fonctions print() et type(). De manière générale, une fonction appelée f et prenant des paramètres x,y s'utilise de la manière suivante : f(x,y) ou (si la fonction renvoie une valeur y) : y=f(x,y)

Pour définir une nouvelle fonction, il faut utiliser le mot-clef "**def**".

Ecrire:

```
    def MaFonction(nom,age): #definition de la fonction
        print (nom,'a',age,'ans' )       #noter les espaces en début de ligne

    MaFonction('Obélix',5)   #utilisation de la fonction
```

résultat: Obélix a 5 ans

nom et age sont les paramètres de la fonction. On peut appeler la même fonction ci-dessus en utilisant une chaine de caractères pour age. Ecrire

```
    MaFonction('Obélix','cinq')   #utilisation de la fonction
```

résultat: Obélix a cinq ans

Il faut que les types des paramètres fournis à la fonction soient conformes à l'utilisation qui va être faite. Ecrire:

```
    def Addition(a,b):
        return a+b          # Renvoie une valeur avec “return”
    print Addition(1,2)         #deux entiers
    print Addition(1,3.0)       #un entier plus un flottant= un flottant
    print Addition('tata','toto')      # deux chaînes de caractères
```

résultat:

```
    3
    4.0
    tatatoto
```

Ecrire

```
    print Addition(1,'toto')   # entier+chaîne de caractères = erreur !
```

résultat:

```
    Traceback (most recent call last):
      File '<stdin>', line 1, in ?
      File '<stdin>', line 2, in Addition
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Remarque 1.8.1. Comme pour les boucles c'est l'indentation (le nombre d'espaces en début de ligne) qui définit quelles sont les instructions qui font partie de la fonction. Cela remplace l'utilisation d'accolades {} dans d'autres langages.

1.8.1 Portée des objets : variables locales et globales

En python, les variables sont locales (par défaut), ce qui signifie qu'elles n'existent qu'au niveau où elles ont été définies. Ecrire:

```
    x=12.3              # Première variable “x”

    def fonction1(): # déclaration d'une fonction
        x=1              # Deuxième variable “x”
        print 'Etape 1, x=',x
```

```
def fonction2(): # déclaration d'une fonction
    x='Toto'         # Troisième variable “x”
    print 'Etape 2, x=',x

print 'Etape 0, x=',x
fonction1()
fonction2()
```

résultat:

```
Etape 0, x= 12.3
Etape 1, x= 1
Etape 2, x= Toto
```

Dans l'exemple ci-dessus, les 3 variables ont le même nom mais sont complètement indépendantes. **On dit qu'elles sont locales.** Il est à noter qu'avec un niveau d'indentation on a accès aux variables du niveau supérieur en lecture, mais on ne peut pas les modifier (contrairement au C/C++ où on peut les lire et les modifier).

1.8.2 Récursivité

La récursivité est un moyen de répéter des blocs d'instructions sans utiliser de boucle while ou for.
La récursivité est lorsqu'une fonction s'appelle elle même.
Regarder l'exemple suivant et essayer de comprendre ce qu'elle fait.
Exemple 1:
Ecrire:

```
def f(n):
    if(n>1):
        return n*f(n-1)
    else:
        return 1
print(f(5))
```

Résultat: 120

Exemple 2:

Soit la suite définie par :
$U_n = 1$ si $n<2$ et $U_n = 3U_{n-1} + U_{n-2}$ si non.

Écrire un programme récursif permettant de calculer le nième terme de la suite;

Résultat :

```
def Suite(n):
   if n <= 2:
     return 1
   else:
     return 3*Suite(n-1)+Suite(n-2)
print(Suite(4)) # un exemple de n=4
```

Exemple 3:

La suite de Fibonacci est définie comme suit :

$F_n = 1$ si $n<2$ et $F_n = F_{n-1} + F_{n-2}$ si non

Écrire un programme récursif calculant Fib(n)

Résultat:

```
def Fib(n):
    if n<2 :
        return 1
    else:
        return Fib(n-1) + Fib(n-2)
print(Fib(4))
```

Pour plus d'informations voir:

https://developpement-informatique.com/article/102/exercices-corriges-de-recursivite-en-python--serie-12

Chapitre 2. Objets plus élaborés

2.1 Vecteurs, matrices et algèbre linéaire

2.1.1 Vecteurs et représentation graphique

Voici comment utiliser arange() pour créer une liste de nombres. Comme pour 'list', les indices commencent à 0. Ecrire:

```
from pylab import *
from numpy import *
```

```
    X=arange(0, 6, 0.2) # crée une liste de nombres X_{i} de 0 à 6
(exclu) par pas de 0.2 (il y a donc 6/0.2 = 30 éléments)
    print ('X = ', X)
    Y=sin(X) # crée une liste où chaque élément est Y_{i}=\sin\
left(X_{i}\right)
    print ('Y = ', Y)

    plot(Y) # dessin des valeurs de \left(Y_{i}\right)_{i} en
fonction de i
    show() # montre le dessin

    plot(X,Y) # dessin des valeurs de Yi en fonction des Xi
    xlabel('x') # rajoute titre aux axes
    ylabel('y=sin(x)')
    show() # montre le dessin
```

N.B: Essaiyer avec un pas trop petit pour comparer les courbes. Par exemple utiliser

```
    X=arange(0, 6, 0.05) # crée une liste de nombres X_{i} de 0 à 6
(exclu) par pas de 0.05
```

résultat:

```
    X = [ 0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5]
    Y = [ 0. 0.47942554 0.84147098 0.99749499 0.90929743 0.59847214
    0.14112001 -0.35078323 -0.7568025 -0.97753012 -0.95892427 -
0.70554033]
```

On peut visualiser le schéma

Pour connaitre le type des tableaux et listes de l'exemple précédent, écrire ensuite:

```
    print (type(X))
```

résultat:

```
    <class 'numpy.ndarray'>
```

Remarque 2.1.1. Ce type 'numpy.ndarray' ne peut contenir que des nombres, et pas d'autres types comme le permet 'list'.

Il y a une autre possibilité pour créer une liste de nombres avec **linspace()**. écrire:

```
    from pylab import *
    from numpy import *
    X=linspace(0,1,10) # 10 nombres X i équidistribués de 0 à 1
compris
    Y=sin(X)
    plot(Y, marker='o', linestyle='solid') # options pour le dessin
    show()
```

résultat: **Visualiser le résultat**

Ecrire:

```
    from numpy import *
    zeros(10)    # liste de 10 nombres tous égaux à zéro
```

Résultat:

```
    array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Documentation:
https://matplotlib.org/api/markers_api.html#module-matplotlib.markers,

Conversion list->array avec la fonction asarray:

Il peut être utile de convertir une liste de nombres de type 'list' en une liste de type 'numpy.ndarray' afin de la dessiner par exemple.

N.B: C'est quoi numpy Ndarray ?
Un numpy. ndarray (généralement appelé array) **est un tableau multidimensionnel homogène: tous les éléments doivent avoir le même type, en général numérique**. Les différentes dimensions sont appelées des axes, tandis que le nombre de dimensions – 0 pour un scalaire, 1 pour un vecteur, 2 pour une matrice, etc.

Ecrire:

```
    from pylab import *
    from numpy import *
    Y = [0,1,4,9,16]
```

```
    print (type(Y))
    Y = asarray(Y)
    print(type(Y))
    plot(Y)
```

Résultat: **Visualiser le résultat**

Copie de tableaux:
On a vu que pour copier une liste L1 dans un nouvel objet L2 il faut écrire **L2=list(L1)** .
De la même façon pour copier un tableau T1 dans un nouveau tableau T2 il faut écrire: **T2 = np.copy(T1).** Ecrire:

```
    import numpy as np
    T1 =np.arange(0, 2, 0.5) # crée une liste de nombres X_{i} de 0
à 2 (exclu) par pas de 0.5
    T2 = np.copy(T1) # copie dans objet distinct
    T3 = T1 # attention, ce n'est pas une vraie copie: T3 sera
seulement un autre nom pour T1
    T2[0] = 10
    T3[0] = 20
    print ('T1 = ', T1)
    print ('T2 = ', T2)
    print ('T3 = ', T3)
```

Résultat:

```
    T1 = [ 20. 0.5 1. 1.5]
    T2 = [ 10. 0.5 1. 1.5]
    T3 = [ 20. 0.5 1. 1.5]
```

2.1.2 Matrices et représentation graphique
2.1.2.1 Premier exemple

Remarque 2.1.2. Attention les indices des tableaux commencent à 0.

```
    from pylab import *
    from numpy import *

    A=zeros([3,3]) # matrice 3 × 3 remplit de zeros
    A[1,1]=2 # on modifie un élément
```

```
    A[2,1]=3
    print (A)
    pcolormesh(A) # dessin des contours des valeurs
    colorbar() # rajoute une barre des valeurs
    show()
```

Resultat: **visualiser le resultat**

2.1.2.2 Autres représentations graphiques

Regarder ces exemples:
https://mpastell.com/2013/05/02/matplotlib_colormaps/

dont les scripts sont les suivants(aller sur ce site pour voir les images):

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

x = linspace(-5, 5, 200)
y = x
X,Y = meshgrid(x, y)
Z = bivariate_normal(X, Y)

for cmap in colormaps():
    fig = figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z, cmap= cmap)
    title(cmap)
```

Resultats: **Visualiser les schemas en executant ces scripts dans spyder** (A défaut, aller sur ce site pour voir les images):

Surface avec ombres:
https://matplotlib.org/2.0.2/examples/mplot3d/custom_shaded_3d_surface.html

2.1.2.3 Création de matrices

Matrice avec valeurs aléatoires:
Ecrire:

```
    from pylab import *
    from numpy import *
```

```
    random.uniform(-1, 1, size=(3,3)) # matrice 3*3 avec éléments
aléatoires dans l'intervalle [ -1,1 ]
```

Résultat:

```
   array([[ 0.40517391, -0.47365213, -0.5208182 ],
    [-0.23394281, 0.0347132 , -0.92326085],
    [-0.94118242, -0.58734857, -0.19587783]])
```

Matrice identité:

```
    from pylab import *
    from numpy import *
    eye(3)
```

Résultat

```
array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]])
```

2.1.2.4 Matrice comme produit tensoriel de vecteurs

L'exemple suivant montre comment à partir de deux vecteurs $(x_i)_i$ et $(y_j)_j$ on peut construire une matrice $(z_{i,j})_{i,j}$. On termine l'exemple par une représentation graphique.

```
    from pylab import *
    from numpy import *

    xmin,xmax,ymin,ymax = -3,3, -3,3 # bornes du domaine
    M = 30 # nombre de points en x et y

    x = linspace(xmin,xmax,M) # vecteur avec M nombres entre xmin et
xmax
    y = linspace(ymin,ymax,M)[:,newaxis] # vecteur dans la 2eme
dimension
    z = exp( -x*x-y*y) # matrice

    levels = linspace(0,1,10) # defini les lignes de niveaux
    contourf(z,levels,extent=(xmin,xmax,ymin,ymax))
    colorbar()
    xlabel('$x$')
    ylabel('$y$')
```

```
    title('fonction $f(x,y)= \exp (-x^2 -y^2)$. Echantillons: $M='+
('%4i'%M)+'$')
```

resultat: **Visualiser le résultat**

2.1.3 Opérations d'algèbre linéaire
Voir scipy, linear algebra sur les adresses suivantes
https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.linalg.html
https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.linalg.html

2.1.3.1 opérations élémentaires
Ecrire:

```
    import numpy as np
    from scipy import linalg
    A = np.array([[2,1],[1,1]]) #matrice
    V = np.array([[1], [1]]) # vecteur colonne
    print ('A=\n', A , '\nV=\n', V)
    print ('A*V=\n', np.dot(A,V)) # produit matrice * vecteur
    print ('A*A=\n', np.dot(A,A)) # produit matrice * matrice
    d = linalg.det(A) #calcule le determinant
    print ('det(A) = ', d)
```

Résultat:

```
 A= [[2 1]
    [1 1]]
    V=
    [[1]
    [1]]
    A*V=
    [[3]
    [2]]
    A*A=
    [[5 3]
    [3 2]]
    det(A) = 1.0
```

Pour le calcul des parties réelles, partie imaginaires, modules, complexe conjuguée , argument en radians
 Ecrire:

```
import numpy as np
x = np.array([ 2 + 3j, 1j, 1])
np.real(x) # partie reelle
y=np.array([ 2.,  0.,  1.])
np.imag(x)  # partie imaginaire
z=np.array([ 3.,  1.,  0.])
t=np.array([ 3.60555128,  1.  ,  1.  ])
np.abs(x) # module
p=np.array([ 0.98279372,  1.57079633,  0.        ])
np.angle(x) # argument en radians
m=np.array([ 2.-3.j,  0.-1.j,  1.-0.j])
np.conj(x) # complexe conjuguée
```

Calcul de l' inverse d'une matrice a avec - numpy.linalg.inv()

```
from numpy.linalg import inv
a = np.array([[1, 3, 3],[1, 4, 3],[1, 3, 4]])
inv(a)
array([[ 7., -3., -3.],[-1.,  1.,  0.],[-1.,  0.,  1.]])
```

Résolution d'un système d'équations linéaires - numpy.linalg.solve()

Pour résoudre le système d'équations linéaires

$$3 * x_0 + x_1 = 9 \text{ et } x_0 + 2 * x_1 = 8$$

```
import numpy as np
    from scipy import linalg
a = np.array([[3,1], [1,2]])
b = np.array([9,8])
x = np.linalg.solve(a, b)
array([ 2., 3.])
```

Pour vérifier que la solution est correcte:

```
np.allclose(np.dot(a,x),b)
    True
```

2.1.3.2 Valeurs propres et vecteurs propres

On rappelle que si M est une matrice diagonalisable n × n alors sa diagonalisation donne $M=PD P^{-1}$ où D est une matrice diagonale des valeurs propres d_j et P est une matrice contenant en colonnes les composantes des vecteurs propres (à droite) U_0, … U_{n-1} , qui vérifient $M U_j = d_j U_j$, avec j= 1,….., n-1.

Ecrire:

```
from pylab import *
from numpy import *
```

```
    M = array(([1,3,3],[1,4,3],[1,3,4]))                # matrice
    D, P = eig(M)              # valeurs propres D et matrice des
vecteurs propres P
    R = dot( dot(P , diag(D)) , inv(P))     # R = P * D * P^(-1)
    print ('M=\n' ,M, '\n P*D*P^(-1) =\n' , R)
   print ('D\n' ,D )     # matrice diagonale des valeurs propres
   print ('P=\n' ,P)    # matrices des vecteurs propres
```

Résultat:

```
    M=  [[1 3 3], [1 4 3],[1 3 4]]

    P*D*P^(-1) =  [[1. 3. 3.] ,[1. 4. 3.] , [1. 3. 4.]]
```

Ecrire:

```
    import pylab as py
    import matplotlib.pyplot as plt
    import numpy as np
    N = 100
    A= 0.1
    M = np.random.uniform(0, A, size=(N,N)) # matrice N*N avec
éléments aléatoires

#  dans [0,A]
    EV = py.eigvals(M)    # calcul des valeurs propres de M
       # dessin des valeurs propres -------------------
    plt.plot(np.real(EV), np.imag(EV), linestyle = 'none', marker =
'o', c = 'lime',
    markersize = 10)
    a = 0.5*sqrt(N)*A
    xmin,xmax,ymin,ymax = -a,a,-a,a
    plt.axis([xmin,xmax,ymin,ymax]) # selectionne la vue
    plt.xlabel('$\Re(z)$')
    plt.ylabel('$\Im(z)$')
    plt.title('Eigenvalues of a random $' + ('%d'%N) +'*'+ ('%d'%N)
+'$ matrix with iid elements in $[0,' + ('%4.2f'%A) +']$')
```

Résultat: **Visualiser le résultat**

Valeurs propres et vecteurs propres avec – numpy.linalg.eig()

```
from numpy.linalg import eig
A = np.array([[ 1, 1, -2 ], [-1, 2, 1], [0, 1, -1]])
array([[ 1,  1, -2],
```

```
       [-1,  2,  1],
       [ 0,  1, -1]])
D, V = eig(A)
D
array([ 2.,  1., -1.])
V
array([[ 3.01511345e-01,-8.01783726e-01,7.07106781e-01],
       [ 9.04534034e-01,-5.34522484e-01,-3.52543159e-16],
       [ 3.01511345e-01,-2.67261242e-01,7.07106781e-01]])
```

Les colonnes de V sont les vecteurs propres de A associés aux valeurs propres qui apparaissent dans D.
Exercice : Vérifier que les colonnes de V sont bien des vecteurs propres de A

2.1.3.3 Matrices creuses (sparse)

Dans de nombreux problèmes on doit utiliser des matrices de taille très grande, (10^4 ou plus) mais avec seulement un petit nombre d'éléments non nuls. Dans ce cas l'ordinateur stocke seulement la liste des éléments non nuls. Ce mode de stockage s'appelle ***« matrice creuse ».***

2.2 Objets graphiques avec Matplotlib

Lire ce site http://math.mad.free.fr/depot/numpy/courbe.html
Site de matplotlib ici
https://matplotlib.org/stable/plot_types/index
(Voir Exemples).
Programmes avec graphisme:
Afin de pouvoir interagir avec les fenêtres graphiques, dans spyder il est conseillé, dans le menu **Outils/Préférences/ConsoleIPython/Graphiques/Sortie** : de choisir le mode automatique.
(et relancer la fenetre Ipython)

2.2.1 Courbe 1Dim simple
Exemple ici.

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

```
    t = np.arange(0.0, 2.0, 0.01) # liste de nombre de 0 a 2 par pas de 0.01
    s = 1 + np.sin(2 * np.pi * t)
    plt.plot(t, s) # points relies (bleus)
    plt.plot(t+0.1, s, 'r.') # red (r) et points isoles (.), on peut faire t+0,5 au lieu de t+0,1 et
                                       #voir la différence sur la figure
    plt.plot(t+0.2, s, 'gx') # green (g) et croix (x)
    plt.xlabel('t(s):temps')
    plt.ylabel('V(mV): Tension')
    plt.title('Titre')
    plt.grid() # une grille
    plt.savefig('V-t.png') # sauvegarde fichier image au format png
    plt.show()
```

Résultat: **Visualiser le schéma dans spyder**

Documentations spécifiques:

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.figure.html

2.2.2 Des sous figures, titres et labels sur les axes

La fonction subplot() permet d'organiser différents tracés à l'intérieur d'une grille d'affichage. Il faut spécifier le nombre de lignes, le nombre de colonnes ainsi que le numéro du tracé.
subplot(xyz): x indique le nombre de ligne, y indique le nombre de colonnes, z le numéro du tracé
(On va y revenir plus loin)

Référence: Titres et labels sur un graphe sur https://matplotlib.org/gallery/subplots_axes_and_figures/figure_title.html

Ecrire:

```
import matplotlib.pyplot as plt
       #-- preparation des figures
fig = plt.figure(1, figsize=(10, 5)) # taille X,Y en pouces (inchs)
plt.subplots_adjust(left=0.1, wspace=0.4, top= 0.8)
plt.suptitle('Titre general', fontsize=16)
     #--- figure 1
xmin,xmax,ymin,ymax = -2,2,-2,2     # fenetre
f1 = plt.subplot(121)       # Il y a 1*2 sous figures. f1 est la figure 1
plt.title('Titre 1')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
f1.plot([0,-1.5], [0,1.5])      # segment (0,0)->(-1.5,1.5)
f1.axis([xmin,xmax,ymin,ymax])      # selectionne la vue
    #--figure 2
amin, amax, bmin, bmax = -3,3, -2, 2
f2 = plt.subplot(122)        # Il y a 1*2 sous figures. f2 est la figure 2
plt.title('Titre 2')
plt.xlabel('a')
plt.ylabel('b')
plt.grid(True)
f2.plot([-2,2], [-1,1])         # segment (-2,-1)->(2,1)
f2.axis([amin,amax,bmin,bmax])    # selectionne la vue
 plt.tight_layout()        # pour que les differentes sous figures ne se superposent pas
```

2.2.3 Des objets graphiques en dimension 2

Référence, exemples sur :
https://matplotlib.org/stable/gallery/shapes_and_collections/artist_reference.html

Ecrire:

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.path as mpath
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
```

```
    from matplotlib.collections import PatchCollection
    from math import *
    fig, ax = plt.subplots()
    #texte
    x,y = 0.5, 0.3
    plt.text(x, y, 'Du texte', ha="center", family='sans-serif',
size=14)

    # Un cercle
    x,y = 0.5,0.5
    r = 0.1
    c = plt.Circle((x,y), r, facecolor= 'green', edgecolor= 'red')
    ax.add_patch(c)

    #Une ellipse
    x,y = 0.2,0.3
    rx,ry = 0.2, 0.1
    angle = 30 # en degres
    e = mpatches.Ellipse((x,y), rx, ry, angle, facecolor= 'yellow',
edgecolor= 'red')
    ax.add_patch(e)

    # un rectangle
    x,y = 0.,0. # coin inferieur gauche
    dx,dy=0.3, 0.1 # taille
    rec = plt.Rectangle((x,y), dx,dy, facecolor= 'blue', edgecolor=
'black')
    ax.add_patch(rec)

    #Une flèche
    x,y = -0.2, 0. # départ
    dx,dy=0.2, 0.3 # déplacement width = 0.1
    a = mpatches.Arrow(x, y, dx, dy, width, facecolor= 'pink',
edgecolor= 'red')
    ax.add_patch(a)

    plt.axis('equal') # pour avoir meme echelle en x,y
    #plt.axis('off') # n'affiche pas le cadre ni les axes
    ax.autoscale_view() # pour que la figure s'ajuste aux objets
    plt.show()
```

En python, on peut tracer les histogrammes, des diagrammes, etc
Voici quelques exemples

Ecrire:

```
 import numpy as np
 import matplotlib.pyplot as plt
 x = np.random.randn(1000)  # Générations de données:1000 points aléatoires

normaux
plt.hist(x, bins=50) # visualisation sur 50 intervalles
```

Resultat: **On peut visualiser le resultat avec spyder**

Astuce: *Écrivez* ***plt.hist(x, bins=x.size)*** *pour construire un histogramme où chaque intervalle correspond en réalité à une valeur. Si votre dataset est très large, divisez plutôt x.size par un nombre: bins=x.size//10 pour créer moins d'intervalles. Cette astuce est très utile pour* ***analyser les pixels d'une photo !***
Pour les courbes planes paramétrées par t, on peut tracer les courbes. Voici un exemple
Ecrire:

```
import numpy as np
import matplotlib.pyplot as plt
T= np.linspace(0,2*np.pi, 1000)
X=[np.sin(3*t) for t in T] # on peut changer les fonctions
Y=[np.sin(2*t) for t in T]
plt.plot(X,Y)
plt.show()
```

Résultat: **On peut visualiser le schéma dans Spyder**

On peut également donner d'autres exemples de courbes paramétrées. Voici par exemples les deux paramétrages gamma et delta du demi-cercle:

1)   gamma(t) = (cos t,sin t) pour 0 < t < pi

2) delta(s) = (-s,sqrt(1-s²)) pour -1 < s < 1

L'algorithme ressemble a celui ci-haut utilisé

Pour les courbes paramétrées en 3D, on peut tracer les courbes. Voici un exemple

```
import numpy as np
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
def f(t):
    return(np.cos(t), np.sin(2*t),t/2*np.pi)

T= np.linspace(0,4*np.pi, 1000)
X=[f(t)[0] for t in T ]
Y=[f(t)[1] for t in T ]
Z=[f(t)[2] for t in T ]
ax=Axes3D(plt.figure())
ax.plot(X,Y,Z)
plt.show()
```

ou bien, on peut faire comme suit:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
T= np.linspace(0,4*np.pi, 1000)
X=[np.cos(t)for t in T ]
Y=[np.sin(2*t) for t in T ]
Z=[t/2*np.pi for t in T ]
ax=Axes3D(plt.figure())
ax.plot(X,Y,Z)
plt.show()
```

Résultat: On peut visualiser le schéma dans Spyder

2.2.4 Dessin d'un champ de vecteur en dimension 2 avec streamplot() ou quiver()

Reference:
https://matplotlib.org/stable/gallery/images_contours_and_fields/quiver_simple_demo.html

a) Création d'un graphique Quiver

Commençons par créer un simple diagramme de quiver contenant une flèche qui expliquera le fonctionnement de la fonction ax.quiver() de Matplotlib. La fonction ax.quiver() prend quatre arguments :

Syntaxe: ax.quiver(x_pos, y_pos, x_dir, y_dir, couleur)

Ici x_pos et y_pos sont les positions de départ de la flèche tandis que x_dir et y_dir sont les directions de la flèche.
Le graphique ci-dessous contient une flèche quiver commençant à x_pos = 0 et y_pos = 0. La direction de la flèche pointe vers le haut et la droite à x_dir = 1 et y_dir = 1.

```
     # Import libraries
import numpy as np
import matplotlib.pyplot as plt
         # Creating arrow
x_pos = 0
y_pos = 0
x_direct = 1
y_direct = 1
      # Creating plot
fig, ax = plt.subplots(figsize = (12, 7))
ax.quiver(x_pos, y_pos, x_direct, y_direct)
ax.set_title('Quiver plot with one arrow')
       # Show plot
plt.show()
```

Résultat: On peut visualiser le schema

b) Quiver Plot avec deux flèches

Ajoutons une autre flèche au tracé passant par deux points de départ et deux directions. En gardant la flèche d'origine commençant à l'origine (0, 0) et pointant vers le haut et vers la droite (1, 1), et créez la deuxième flèche commençant à (0, 0) pointant vers le bas dans la direction (0, -1) .

Pour voir clairement le point de départ et d'arrivée, nous allons définir les limites de l'axe à [-1.5, 1.5] en utilisant la méthode ax.axis() et en passant les arguments sous la forme [x_min, x_max, y_max, y_min] .

En ajoutant un argument supplémentaire scale=value à la méthode ax.quiver(), nous pouvons gérer les longueurs des flèches pour qu'elles paraissent plus longues et s'affichent mieux sur le tracé.

Exemple:

```
                 # Import libraries
import numpy as np
```

```
import matplotlib.pyplot as plt
      # Creating arrow
x_pos = [0, 0]
y_pos = [0, 0]
x_direct = [1, 0]
y_direct = [1, -1]
      # Creating plot
fig, ax = plt.subplots(figsize = (12, 7))
ax.quiver(x_pos, y_pos, x_direct, y_direct,scale = 5)
ax.axis([-1.5, 1.5, -1.5, 1.5])
            # show plot
plt.show()
```

Résultat: On peut visualiser le schéma

c) **Coloring Quiver Plot**

La méthode ax.quiver() de la bibliothèque matplotlib de python fournit un attribut optionnel color qui spécifie la couleur de la flèche. L'attribut de couleur du Quiver nécessite les mêmes dimensions que les tableaux de position et de direction.
Ci-dessous se trouve le code qui modifie les diagrammes de quiver que nous avons créés précédemment:

```
   # Import libraries
import numpy as np
import matplotlib.pyplot as plt
     # Defining subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize =(14, 8))
   # first subplot
   # Creating arrows
x = np.arange(0, 2.2, 0.2)
y = np.arange(0, 2.2, 0.2)
X, Y = np.meshgrid(x, y)
u = np.cos(X)*Y
v = np.sin(y)*Y
n = -2

   # Defining color
color = np.sqrt(((v-n)/2)*2 + ((u-n)/2)*2)

   # Creating plot
```

```
ax1.quiver(X, Y, u, v, color, alpha = 0.8)
ax1.xaxis.set_ticks([])
ax1.yaxis.set_ticks([])
ax1.axis([-0.2, 2.3, -0.2, 2.3])
ax1.set_aspect('equal')
ax1.set_title('meshgrid function')

    # second subplot
  # Creating arrows (Création des fléchés)
x = np.arange(-2, 2.2, 0.2)
y = np.arange(-2, 2.2, 0.2)
X, Y = np.meshgrid(x, y) # utilisation des flèches
z = X * np.exp(-X**2 -Y**2)
dx, dy = np.gradient(z)
n = -2

    # Defining color
color = np.sqrt(((dx-n)/2)*2 + ((dy-n)/2)*2)

    # Creating plot
ax2.quiver(X, Y, dx, dy, color)
ax2.xaxis.set_ticks([])
ax2.yaxis.set_ticks([])
ax2.set_aspect('equal')
ax2.set_title('gradient')

    # show figure
plt.tight_layout()
plt.show()
```

Résultat: On peut montrer le schéma

Faisons un autre exemple plus complexe en écrivant comme suit :

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

X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
U, V = np.meshgrid(X, Y)
fig, ax = plt.subplots()
q = ax.quiver(X, Y, U, V)
```

```
ax.quiverkey(q, X=0.3, Y=1.1, U=10,label='Quiver key, length = 10', labelpos='E')
plt.show()
```

Ecrire:

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

  #fonction qui definit le champ de vecteur F
    def F(x, y):
        return x, -y

    # reseau de points en x,y
    xmin, xmax, dx = -2,2, 0.2
    ymin, ymax, dy = -2,2, 0.2
    X,Y = np.meshgrid( np.arange(xmin, xmax, dx), np.arange(ymin, ymax, dy) )
    Fx, Fy = F(X,Y) # calcule le champ de vecteur sur le reseau

    #----------------
    plt.streamplot(X,Y, Fx,Fy)
    plt.title('avec streamplot')

    #----------------
    plt.figure() # nouvelle figure
    norme = np.sqrt(Fx**2+Fy**2) # norme du champ F
    strm = plt.streamplot(X, Y, Fx, Fy, color = norme, linewidth = norme, cmap=plt.cm.inferno, density=[1, 2], arrowstyle='->', arrowsize=1.5)
    plt.colorbar(strm.lines) # montre l'echelle
    plt.title("avec streamplot et plus d'options")

    #----------------
    plt.figure() # nouvelle figure
    plt.quiver(X,Y,Fx,Fy)
    plt.title('avec quiver')
```

Résultat: **Visualiser le schema dans spyder**

2.2.5 Dessin des lignes de champ passant par des points spécifiés avec streamplot()

Ecrire:

```
import numpy as np
import matplotlib.pyplot as plt
      #definit le champ de vecteur F
def F(x, y):
   return x, -y
    # reseau de points en x,y
nx, ny = 64, 64 # nombre de points
x = np.linspace(-2, 2, nx) # 1dim array
y = np.linspace(-2, 2, ny)
X, Y = np.meshgrid(x, y) # 2dim array
Fx, Fy = F(X,Y) # calcule le champ de vecteur sur le reseau

           # Liste de points (x,y)
points = np.array([[1, 1, -1, -1, 1], [0, 1, 1, -1, -1]])
plt.streamplot(x, y, Fx, Fy, start_points = points.T)
plt.plot(points[0], points[1], linestyle ='none', marker='o') # dessin des points
plt.title('Lignes de champ passant par des points spécifiques')
```

Résultat: **Visualiser le schéma**

2.2.6 Créer une animation

Reference.
Dans l'exemple suivant on considère N nombres complexes
$z_j (x) = \exp (i2 \pi (j+x) /N)$ avec les indices j= 0,1, … N-1 et un déphasage fixé x ∈ [0,1] . Ces N nombres sont équidistribués sur le cercle unité $| z_j(x) | =1$ et on remarque que $z_j(1) = z_{j+1} (0)$. Ainsi en faisant varier x=0 → 1 on verra les N points tourner. Remarquer que
$X_j (x) = Re (z_j(x)) = \cos (2 \pi (j+x) /N)$ et $Y_j (x) = Im (z_j(x)) = \sin (2 \pi (j+x) /N)$.
Ecrire:

```
import matplotlib.pyplot as plt
import numpy as np
import subprocess
from math import *
N, P = 5, 10 # nombre de points et nombre d'images,
xmin,xmax,ymin,ymax = -2,2,-2,2
for p in range(P):   # boucle p = 0,1..,P-1
    x = p/P
    j = np.arange(N)     # liste 0->(N-1)
```

```
        X = np.cos(2*pi*(j+x) /N)      # liste des X_j(x)
        Y = np.sin(2*pi*(j+x) /N)      # liste des Y_j(x)
        plt.cla()               # efface le graphisme precedent
        plt.plot(X,Y, linestyle='none', marker='o')
        plt.axis([xmin,xmax,ymin,ymax])      # selectionne la vue
        plt.pause(0.01)         # montre la figure et attend 0.01 
sec.
   plt.show()          #laisse la figure à la fin et attend qu'on 
la referme
```

Pour voir le résultat, ouvrir un terminal (fenêtre de commandes) et lancer le programme par: python3 Nom_du_programme.py

2.2.7. Créer une animation et crée un fichier gif animé avec convert ou gifsicle

Voici le même programme que ci-dessus, mais avec deux instructions supplémentaires (savefig() qui sauve chaque image dans la boucle et convert... qui fabrique la vidéo finale), afin de fabriquer à la fin un fichier gif animé. Il faut avoir installé le logiciel imagemagick. Nous avons une Animation avec Matplotlib

Vous découvrirez ici comment créer une animation avec **Python** et **Matplotlib**

Exemple 1:

```
    import matplotlib.pyplot as plt
    import numpy as np
    import subprocess
    from math import *

    N, P = 5, 10 # nombre de points et nombre d'images,
    xmin,xmax,ymin,ymax = -2,2,-2,2
    for p in range(P): # boucle p = 0,1..,P-1
      x = p/P
      j = np.arange(N) # liste 0->(N-1)
      X = np.cos(2*pi*(j+x) /N) # liste des X_j(x)
      Y = np.sin(2*pi*(j+x) /N) # liste des Y_j(x)
      plt.cla() # efface le dessin precedent
      plt.plot(X,Y, linestyle='none', marker='o')
      plt.axis([xmin,xmax,ymin,ymax]) # selectionne la vue
      plt.savefig('image_' + str(p).zfill(4) + '.png', dpi=50)
        # sauve fichier image (pour ensuite gif anime). dpi: 
precision de l'image
```

```
      plt.pause(0.001) # montre la figure et attend 0.001 sec.

    #--- cree un fichier gif anime. delay en 1/100s.----------------------------
    subprocess.getoutput('convert -delay 10 -loop 0 image_*.png animation.gif')
                             #fabrique fichier gif final
    subprocess.getoutput('rm image_*.png') # efface les fichiers images temporaires
    print ('fichiers animation.gif cree. Veuillez le lire avec firefox par exemple.')
```

Pour voir le résultat, ouvrir un terminal (fenêtre de commandes) et lancer le programme par: python3 programme.py

2.2.8. Animation avec effacement

a) Animation avec le module **animation** de Matplotlib

Nous allons utiliser la fonction FuncAnimation() du module **animation**.

Dans ce script, nous allons définir une fonction animate() qui met à jour la courbe pour chaque image.

Exemple 2:

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

k = 2*np.pi
w = 2*np.pi
dt = 0.01
xmin = 0
xmax = 3
nbx = 151
x = np.linspace(xmin, xmax, nbx)
fig = plt.figure() # initialise la figure
line, = plt.plot([], [])
plt.xlim(xmin, xmax)
plt.ylim(-1, 1)
def animate(i):
    t = i * dt
    y = np.cos(k*x - w*t)
    line.set_data(x, y)
    return line,
 ani = animation.FuncAnimation(fig, animate, frames=100,
                              interval=1, blit=True, repeat=False)
```

```
plt.show()
```

La fonction FuncAnimation() dispose d'un argument avec une étiquette appelée interval, qui est le temps en millisecondes entre deux appels de la fonction de mise à jour, ici animate().
Nous y définissons une fonction init() qui est affectée au paramètre init_func de FuncAnimation(). Ceci entraine un appel de cette fonction avant la première image. Cette approche n'est toutefois pas indispensable pour les usages qui sont réalisés le plus souvent.

Voici l'exemple:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
k = 2*np.pi
w = 2*np.pi
dt = 0.01
xmin = 0
xmax = 3
nbx = 151
x = np.linspace(xmin, xmax, nbx)
fig = plt.figure() # initialise la figure
line, = plt.plot([], [])
plt.xlim(xmin, xmax)
plt.ylim(-1, 1)
def init():
    line.set_data([], [])
    return line,
def animate(i):
    t = i * dt
    y = np.cos(k*x - w*t)
    line.set_data(x, y)
    return line,
 ani = animation.FuncAnimation(fig, animate, init_func=init, 
frames=100,
                              interval=1, blit=True, repeat=False)
plt.show()
```

b) Animation sans le module animation

Nous présentons ici une technique d'animation plus basique qui n'utilise pas le module **animation**. Cette technique n'est pas recommandée mais elle peut servir pour des animations simples. Pour

des animations plus élaborées, l'utilisation du module **animation** est préférable.

Exemple:

```
import numpy as np
import matplotlib.pyplot as plt
k = 2*np.pi
w = 2*np.pi
dt = 0.01

x = np.linspace(0, 3, 151)
for i in range(50):
    t = i * dt
    y = np.cos(k*x - w*t)
    if i == 0:
        line, = plt.plot(x, y)
    else:
        line.set_data(x, y)
    plt.pause(0.05) # pause avec duree en secondes
    plt.show()
```

c) Animation sans effacement

Exemple:

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

k = 2*np.pi
w = 2*np.pi
dt = 0.01

x = np.linspace(0, 3, 151)

for i in range(50):
            t= i * dt
            y = np.cos(k*x - w*t)
           plt.plot(x, y)
          plt.pause(0.01) # pause avec duree en secondes
plt.show()
```

2.2.9. Affichage de plusieurs tracés dans la même figure

a) Style «pyplot» - Utilisation de subplot()

La fonction subplot() permet d'organiser différents tracés à l'intérieur d'une grille d'affichage. Il faut spécifier le nombre de lignes, le nombre de colonnes ainsi que le numéro du tracé.

subplot(xyz): **x** indique le nombre de ligne, **y** indique le nombre de colonnes, **z** le numéro du tracé

3.3.1 Exemple de disposition en colonne

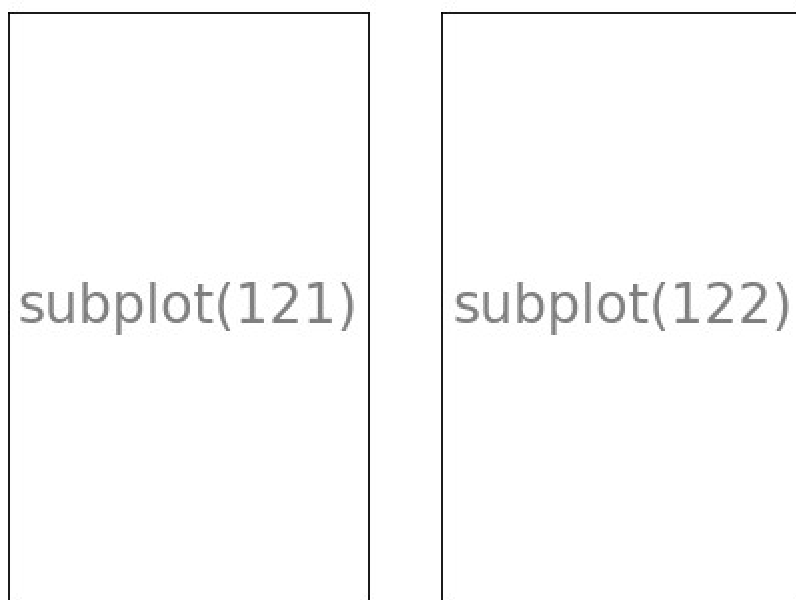
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.subplot(211)
plt.plot(t1, f(t1), "bo")
plt.plot(t2, f(t2), "k")

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), "r--")
plt.show()
```

Exemple de disposition en colonne

subplot(211)

subplot(212)

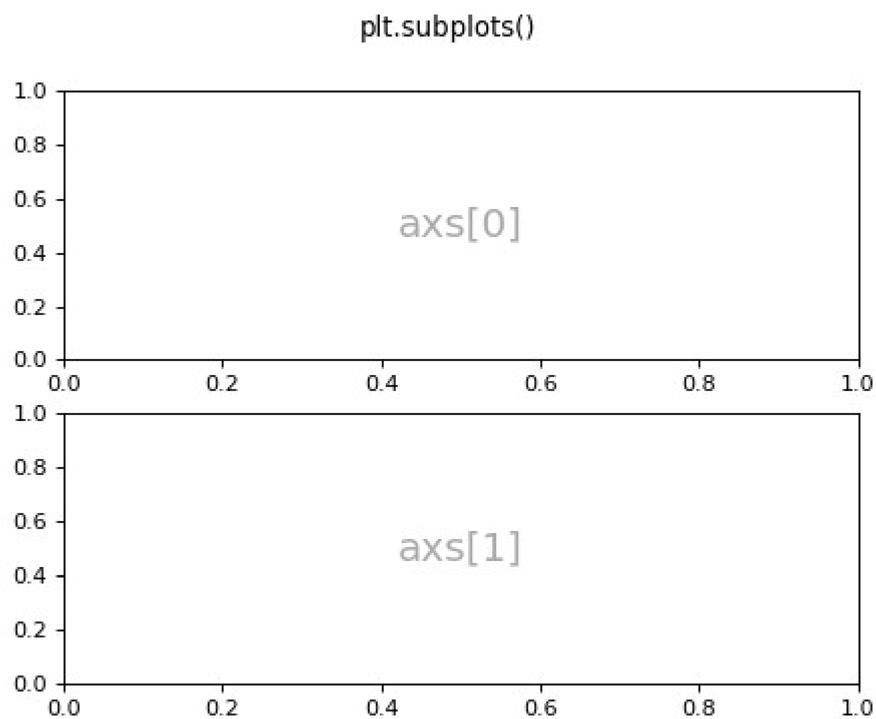
Exemple de disposition en ligne



b) Style « Orienté Objet » - Utilisation de subplots()

Exemple de disposition en colonne

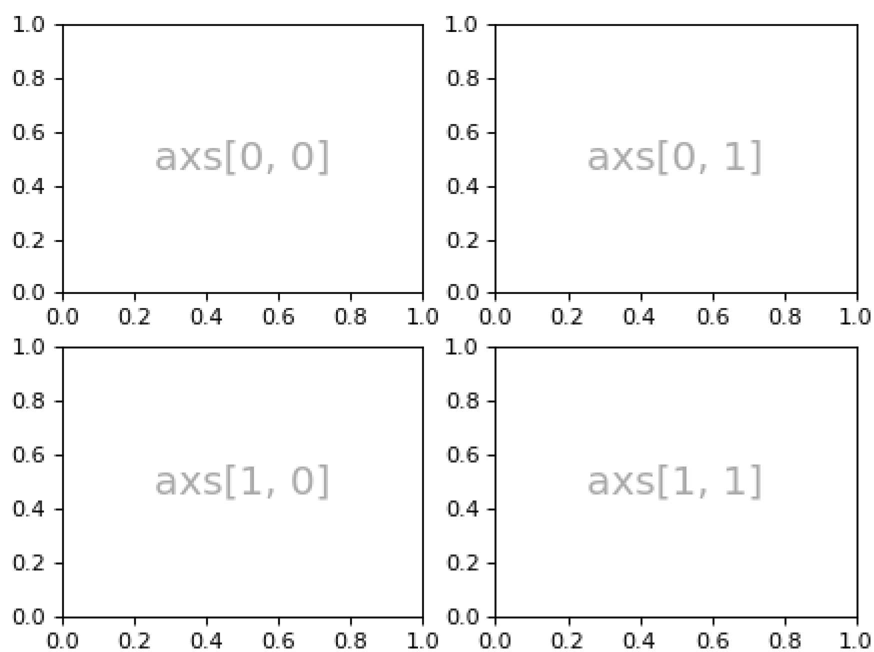
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

fig, axs = plt.subplots(2, 1)

axs[0].plot(t1, f(t1), "bo")
axs[0].plot(t2, f(t2), "k")

axs[1].plot(t2, np.cos(2*np.pi*t2), "r--")

plt.show()
```



Résultat: On peut le visualiser avec spyder

Exemple de disposition en grille

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

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(t1, f(t1), "b.")
axs[0, 1].plot(t2, f(t2), "k")
axs[1, 0].plot(t2, np.cos(2*np.pi*t2), "r--")
axs[1, 1].plot(t2, np.cos(2*np.pi*t2), "g.")

plt.show()
```



Résultat : On peut le visualiser avec spyder

Chapitre 3. Algorithmes et pratiques

3.1. Définitions et exemples

Un algorithme est une suite finie d'actions élémentaires assemblées de manière logique. Le but d'un algorithme est de réaliser une action complexe en la décomposant en actions simples, c'est-à-dire pouvant être effectuées sans réflexion.

Lorsqu'on traduit un algorithme dans un langage de programmation, on dit que l'on **implémente** cet algorithme, la transcription est alors appelée **programme informatique.**

Il existe différentes façons de présenter un algorithme. Notre but est d'implémenter un algorithme. Nous faisons ici la présentation d'un algorithme et la syntaxe la plus proche des langages de programmation.

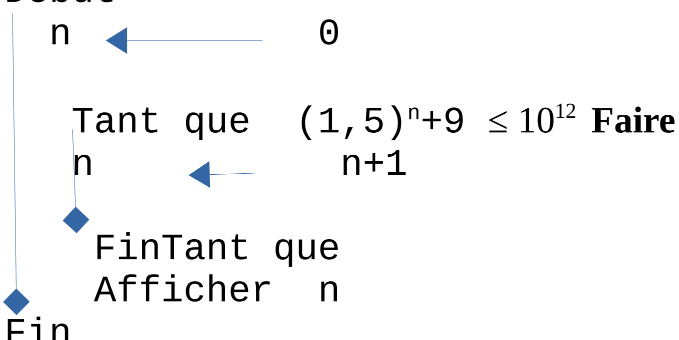
Tous les programmes seront écrits en python. Nous avons choisi ce langage pour la simplicité de sa syntaxe et la lisibilité qui permettent une meilleure compréhension des concepts fondamentaux.

Exemples

1. Pour une suite (u_n) définie par $u_n=(1,5)^n+9$. On veut déterminer le plus petit entier naturel tel que $u_n > 300$.

L'algorithme est le suivant

Variable: n (entier)
Début



n ← 0

Tant que $(1,5)^n+9 \leq 10^{12}$ **Faire**
n ← n+1

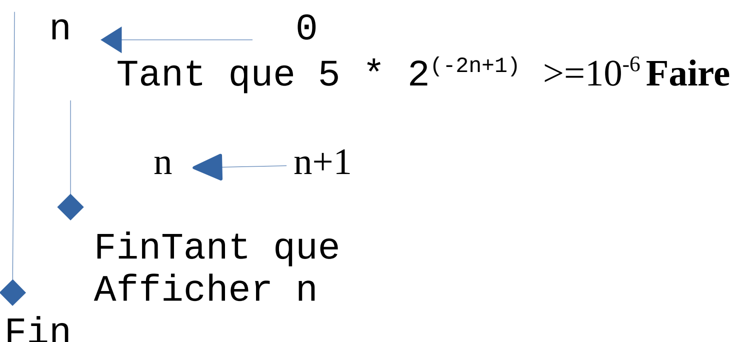
FinTant que
Afficher n
Fin

2. Soit (u_n) la suite définie par $u_n = 5.\ 2^{(-2n+1)}$. Déterminer le plus petit entier naturel n tel que $u_n < 10^{-6}$.

Une fois fait les calculs 5. $2^{(-2n+1)} < 10^{-6}$, on obtient $(0,25)^n < 10^{-7}$

L’algorithme est le suivant

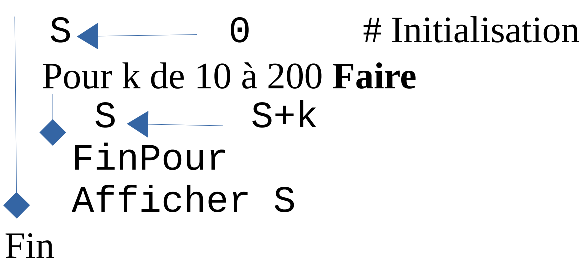
Variable : n (entier)
Début



```
n ← 0
   Tant que 5 * 2^(-2n+1) >=10^-6 Faire
      n ← n+1
   FinTant que
   Afficher n
Fin
```

3. Algorithme qui permet de calculer puis afficher la somme S=10+11+…...+200.

Variables: S, k (entiers)
Début



```
S ← 0        # Initialisation
Pour k de 10 à 200 Faire
   S ← S+k
FinPour
Afficher S
Fin
```

3.2. Création des fonctions et procédures

A part les fonctions usuelles (par exemples, exp, ln, sin, cos, etc), on peut créer en algorithmique toute sorte de fonctions, comme la fonction f définie par $f(x)=3x^2-x+2$ ou des fonctions plus originales comme la fonction g qui à tout nombre n renvoie la chaîne de caractères «nombre pair « ou « nombre impair» selon la valeur de n.

Structure générale de l’écriture d’un algorithme

Nom de l’algorithme
Rôle: description rapide du rôle de l’algorithme
Entrée(s): donnée(s) saisie(s)

Sortie(s): données affichée(s)
Variables globales: nom_variable1(type), nom_variable2(type), etc

……………………..Fonctions et procédures………………………
Fonction nom_fonction(liste des paramètres et types) : type du résultat
Rôle : description rapide du rôle de l'algorithme
Variables locales : nom_variable1(type), nom_variable2(type), etc
Début
<instructions>
FinFonction

3.2.1. Exemples

1. Le plus petit chiffre de 3528 et 2. L'algorithme suivant détermine combien d'entiers naturels de 4 chiffres ont leur plus petit chiffre égal à 2. Celui-ci fait appel à la fonction **pluspetitchiffre(n)** qui renvoie à tout entier $n \geq 0$ le plus petit chiffre de n.

Algorithme Combien_nbde4chiffres_pluspetitchiffre2
Rôle : determine combien de nb de 4 chiffres ont pour plus petit chiffre 2
Entrée(s) : …………..
Sortie(s) : données affichée(s)
Variables globales : Nombre de nb de 4 chiffres de plus petit chiffre 2

Variables globales : nb, compt(entiers)
…………………………Fonction……………………………………………….
Fonction : Pluspetitchiffre(n:entier) : entier
Rôle : renvoie le plus petit chiffre du nombre n
Variables locales : k, min (entiers), ch(chaine)
Début…………………………………….
chiffremin ← 9 # Initialisation
ch ← str(n) # chaine de caractères associée à n
Pour k de 0 à longueur(ch)-1 Faire
Si int(ch[k])<chiffremin **Alors**
chiffremin ← int(ch[k])

```
    FinSi
     FinPour
     Retourner chiffremin
FinFonction
```

……………………...Partie principale……………………………….

```
Début
 Compt ←      0       # Initialisation du compteur
     Pour nb de 1000 à 9999 Faire
            Si pluspetitchiffre(nb) ==2 Alors
       compt ←          compt +1
            Fin Si
     Fin Pour
  Afficher compt
Fin
```

2. La fonction puissance(x,n) suivante renvoie à tout réel x ; et tout entier naturel n, le nombre x^n. Elle utilise une boucle. On dit qu'il s'agit d'une **fonction itérative.**

Fonction puissance(x,n)
Rôle : calcul itératif de x^n
Variables locales : p, k (entiers)
Début :

```
  p ← 1
  Si n > 0 Alors
    Pour k de 1 à n Faire
            p ←         p*x
    FinPour
  Fin Si
  Retourner(p)
FinFonction
```

3.3. Pratique et implémentation en python

Le but de cette section est de former un algorithme et ensuite l'implémenter en python.

a) Algorithmes

1) On voudrait ici écrire un algorithme permettant de calculer des expressions du type

$x_0.I_3 + x_1.A + x_2.A^2 + \ldots\ldots.. + x_n.A^n$ pour A une matrice carrée d'ordre 3 et pour des coefficients réels $x_0, x_1, \ldots\ldots, x_n$ (les coefficients et les éléments de la matrice étant rentrées par l'utilisateur). Nous

définirons une matrice carrée 3x3 comme une liste dont les éléments sont les listes des éléments de chaque ligne comme suit

$[[a_{11},a_{12}\ a_{13}],\ [a_{21},a_{22}\ a_{23}],\ [a_{31},a_{32},\ a_{33}]]$

L'algorithme est le suivant:

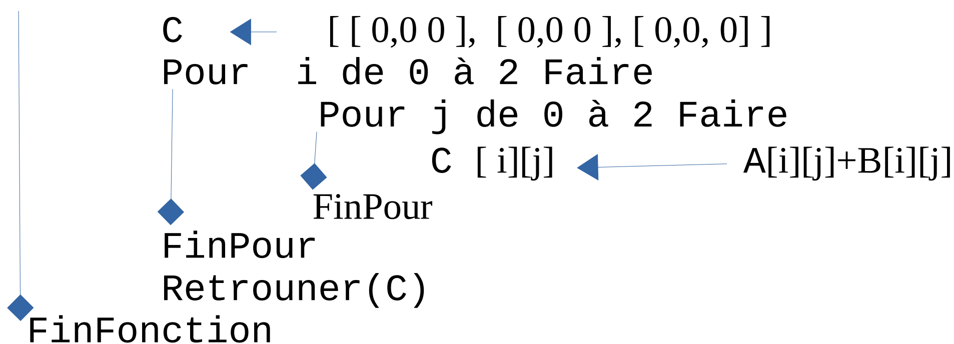
```
Début :
      Afficher'' Calcul de  x_0*I3 +x_1*A+x_2*A_2+……..+x_n*A^n ''
      Afficher ''Rentrer n:''
      Saisir N
      listecoefs [ ] ◄── # Initialisation des coefficients : liste
                                          vide
      matA     ◄──      [[0,0,0], [0,0,0], [0,0,0]] # Initialisation
                                       de la matrice A

      Pour p de 0 à N Faire
           Afficher  '' x_'', p, ''=''
           Saisir coef   # Saisie des coefficients x_0,….., x_n
           Ajouter coef à listecoefs
      FinPour

      Pour   p de 0 à 2 Faire
             Pour a de 0 à 2 Faire
                     Afficher ''a_'', p+1, '' ,'', q+1, ''=''
                     Saisir matA[p][q]# Saisie  des éléments de A
             FinPour
      FinPour

         S  ◄──         [[0,0,0], [0,0,0], [0,0,0]]
      Pour p de 0 à N Faire
          S ◄──         so(S, produitR(listecoefs [p],puissance(A,p)))
      FinPour
      Afficher ''Matrice obtenue :''
      Afficher( S[0])   # premiere ligne de la matrice x_0*I_3+…..
                                 +x_n*A
      Afficher( S[1])   # premiere ligne de la matrice x_0*I_3+…..
                                       +x_n*A
      Afficher( S[2])   # premiere ligne de la matrice x_0*I_3+…..
                                             +x_n*
Fin
```

2) Le rôle de la fonction so(A,B) est de faire la somme des matrices A et B.

L’algorithme est le suivant

Fonction so(A,B:listes):liste
Rôle :……………………………
Variables locales : i,j(entiers), C (liste)

Début

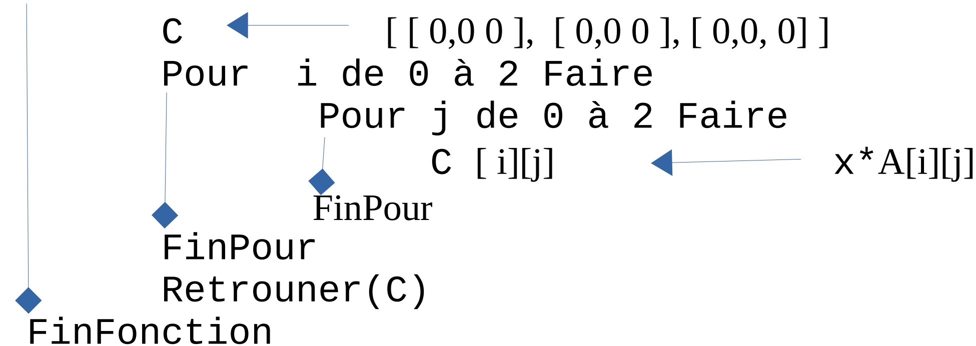


```
        C ← [ [ 0,0 0 ], [ 0,0 0 ], [ 0,0, 0] ]
        Pour  i de 0 à 2 Faire
                Pour j de 0 à 2 Faire
                     C [ i][j] ← A[i][j]+B[i][j]
                FinPour
        FinPour
        Retrouner(C)
 FinFonction
```

3) Écrire une fonction produitR(x,A) qui à tout réel x et à une matrice A carrée d’ordre 3, retourne la matrice x.A.

L’algorithme est le suivant :

Fonction produitR(x : réel,A:liste):liste
Rôle : Calcul le produit xA
Variables locales : i,j(entiers), C (liste)
Début



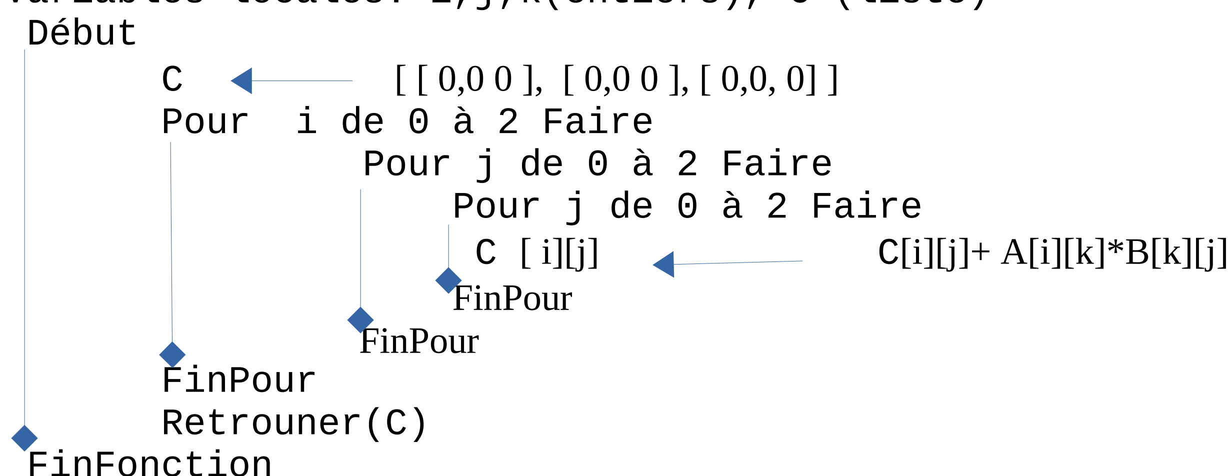
```
        C ← [ [ 0,0 0 ], [ 0,0 0 ], [ 0,0, 0] ]
        Pour  i de 0 à 2 Faire
                Pour j de 0 à 2 Faire
                     C [ i][j] ← x*A[i][j]
                FinPour
        FinPour
        Retrouner(C)
 FinFonction
```

4) Écrire une fonction produitR(A,B) qui à toute matrice A et à une matrice B toutes carrées d'ordre 3, retourne la matrice AxB.

On rappelle que si A=(a_{ij}) et B=(b_{ij}), C=(c_{ij}) et C=AxB alors pout tous i,j∊{1,2,3} on a alors : $c_{ij}=a_{i1}.b_{1j}+a_{i2}.b_{2j}+a_{i3}.b_{3j}$

Fonction produitR(A,B:listes):liste
Rôle: Calcul le produit AB

Variables locales: i,j,k(entiers), C (liste)



```
Début
      C ←  [ [ 0,0 0 ],  [ 0,0 0 ], [ 0,0, 0] ]
      Pour  i de 0 à 2 Faire
               Pour j de 0 à 2 Faire
                   Pour j de 0 à 2 Faire
                    C  [ i][j]  ←       C[i][j]+ A[i][k]*B[k][j]
                   FinPour
               FinPour
      FinPour
      Retrouner(C)
FinFonction
```

b) Implémentation en python

```
############### Fonctions ###########################
def so(A,B):
          C=[[ 0,0, 0], [ 0,0, 0], [ 0,0, 0]]
    for i in range(0,3) :
                 for j in range(0,3):
         C[i][j]=A[i][j]+B[i][j]
    return(C)
 def produitR(x, A):
          C=[[ 0,0, 0], [ 0,0, 0], [ 0,0, 0]]
    for i in range(0,3) :
                 for j in range(0,3):
         C[i][j]=x*A[i][j]
    return(C)
```

```
def produit(A,B) :
        C=[[ 0,0, 0], [ 0,0, 0], [ 0,0, 0]]
  for i in range(0,3) :
            for j in range(0,3):
                  for k in range(0,3):
         C[i][j]=C[i][j]+A[i][k]*B[k][j]
  return(C)


def puissance(A,n) :
      if n==0:
        return ([[1,0,0],[0,1,0],[0,0,1]])
      else:
        return(produit(puissance(A,n-1),A)

######################Partie principale #####################

print(''Calcul de x_0*I_3+x_1*A+……….+x_n*A '')
N=int(input(''Rentrer n : ''))
listcoefs= []
matA=[[ 0,0, 0], [ 0,0, 0], [ 0,0, 0]]
for p in range (N+1) :
        coef=float(input('x_'+str(p)+'=') )
        listcoefs.append(coef)
for p in range(3):
        for q in range(3) :
              matA[ p] [q]=float( input('a_'+str(p+1)+','+str(q+1)+'=') )
S= [[ 0,0, 0], [ 0,0, 0], [ 0,0, 0]]
for p in range (N+1) :
        S=so( S,produitR(listecoefs[p],puissance(matA,p) ) )
        print(''Matrice obtenue :'' )
print(S[0])
print(S[1])
print(S[2])
```

Par exemple, pour A=[[1,2, 3], [2,3, 4], [3,4, 5]], n=2, x_0=1, x_1=2 et x_2=3, on obtient :

Après exécution, on obtient

- [45,64, 84], la première ligne de la matrice obtenue
- [64,94, 122], la deuxième ligne de la matrice obtenue
- [84,122, 161], la troisième ligne de la matrice obtenue

3.4. Intégration: Méthodes d'approximations

Le but de cette section est d'obtenir une approximation d'une intégrale définie du type

$$J=\int_a^b f(x)dx$$

pour une certaine fonction f: [a,b] → ℝ trop compliquée pour à priori déterminer la valeur de J à la main. Des méthodes d'approximations déterministes et probabilistes seront introduites pour obtenir une approximation de J. Il existe de nombreuses méthodes pour réaliser une intégration numérique. Nous allons considérer ici quelques méthodes simples.

a)Méthodes des rectangles

Dans cette méthode, on calcule l'intégrale numérique en réalisant une somme de surfaces de rectangles. Le domaine d'intégration est découpé en intervalles et on fait comme si la fonction restait constante sur chaque intervalle. Sur chaque intervalle, on réalise ainsi l'approximation suivante :

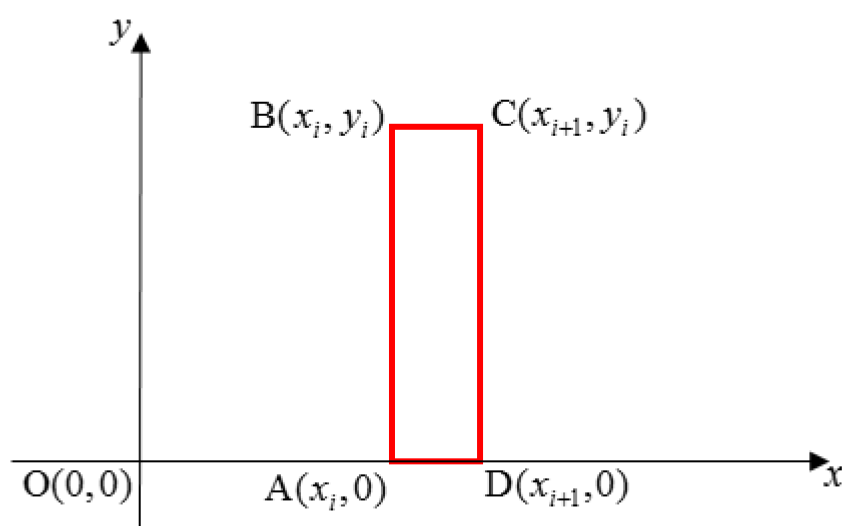
$$J_n = \int_a^b f(x)\,dx \approx (b-a)f(\alpha)$$

où α est une abscisse appartenant à l'intervalle limité par a et b .

Nous nous limiterons ici aux cas où α =a ou b, ce qui signifie que pour chaque intervalle nous considérons comme constante la valeur prise par la fonction à l'extrémité gauche ou droite de l'intervalle.

Comme exemple, nous allons réaliser un programme d'intégration pour α = a et nous visualiserons les rectangles.

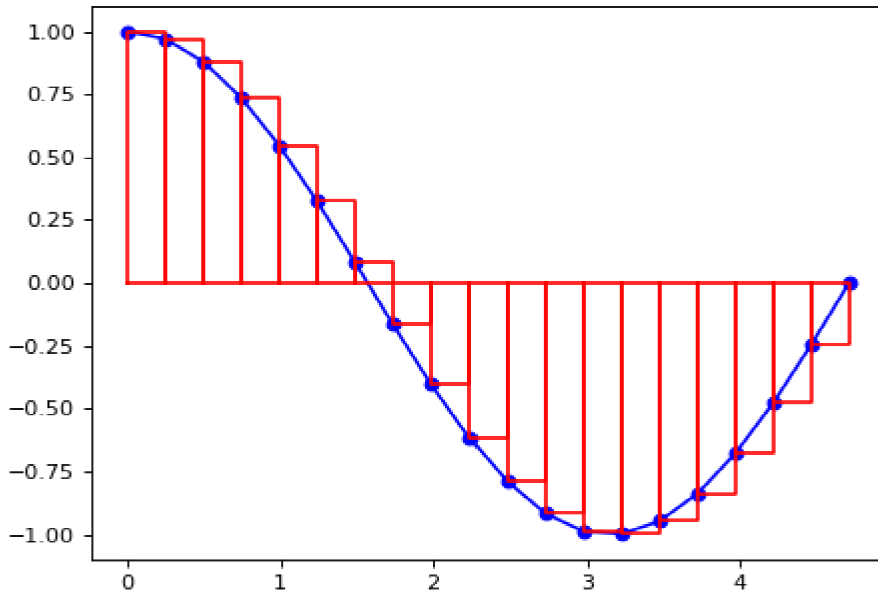
Pour tracer un rectangle ABCD (voir figure ci-dessous), il suffit de faire un plot avec les coordonnées de A, B, C, D et A. On termine par A pour fermer le tracé.



Implémentation en python

On peut visualiser la figure pour # intégration numérique par la méthode des rectangles avec alpha = a

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

xmin = 0
xmax = 3*np.pi/2
nbx = 20
nbi = nbx - 1 # nombre d'intervalles

x = np.linspace(xmin, xmax, nbx)
y = np.cos(x)
plt.plot(x,y,"bo-")

integrale = 0
for i in range(nbi):
    integrale = integrale + y[i]*(x[i+1]-x[i])  # dessin du
rectangle
    x_rect = [x[i], x[i], x[i+1], x[i+1], x[i]] # abscisses des
sommets
    y_rect = [0   , y[i], y[i]  , 0     , 0   ] # ordonnees des
sommets

plt.plot(x_rect, y_rect,"r")
print("integrale =", integrale)
plt.show()
```



Exercice

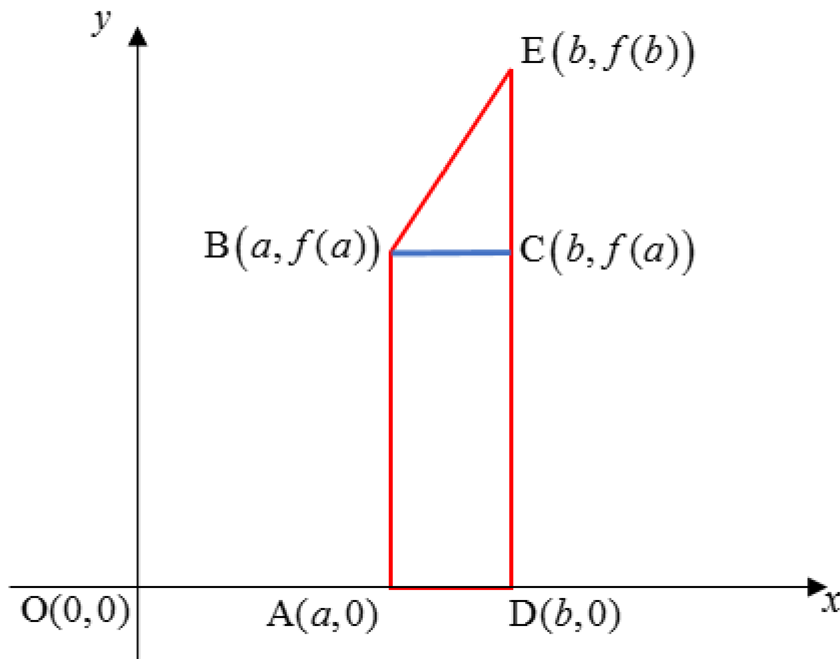
Faire de même pour $\alpha = b$.

b) Méthode des trapèzes

Comme son nom l’indique, cette méthode d’intégration utilise une somme de surfaces de trapèzes.
Sur chaque intervalle, on réalise alors l’approximation suivante :

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2} \left(f(a)+f(b) \right)$$

Justification de la formule



Pour calculer la surface du trapèze ABED, on fait la somme des aires du rectangle ABCD et du triangle rectangle BEC.

surface du rectangle ABCD = AD x AB = (b-a)f (a)

surface du triangle rectangle

$$BEC = \frac{BC\,x\,CE}{2} = \frac{(b-c)[f(b)-f(a)]}{2}$$

$$\text{surface du trapèze ABED} = \frac{b-a}{2}[2f(a)+f(b)-f(a)] = \frac{b-a}{2}[f(a)+f(b)]$$

Exercice

Faire un programme similaire au précédent avec cette fois la méthode des trapèzes en utilisant les mêmes valeurs numériques pour la fonction. Réaliser de même la visualisation des trapèzes.

c) Intégration par la méthode de Simpson

Il existe de nombreuses méthodes pour réaliser une intégration numérique. Nous allons considérer la méthode de Simpson.
La méthode de Simpson permet le calcul approché d'une intégrale avec la formule suivante :

$$\int_a^b f(x)\,dx = \frac{b-a}{6}[f(a)+4f(\frac{a+b}{2})+f(b)]$$

Nous allons voir de façon détaillée maintenant d'où viennent les coefficients $\frac{1}{6}$ et $\frac{2}{3}$ (qui apparaît sous la forme de $\frac{4}{6}$).

Pour obtenir la formule de Simpson, on va réaliser une interpolation avec un polynôme de degré 2. Un polynôme étant une fonction très facile à intégrer, on approche l'intégrale de la fonction f sur l'intervalle, par l'intégrale du polynôme sur ce même intervalle $[a,b]$.

Interpolation par un polynôme de degré 2

A. On veut approximer l'intégrale $\int_a^b f(x)dx$

1. On prend trois points x0,x1,x2 avec x0=a, x2=b et x1= $\frac{a+b}{2}$

2. On interpole la fonction f(x) par un polynome quadratique P_2(x) passant par ces trois points:

P_2(x0)=f(x0), P_2(x1)=f(x1), P_2(x2)=f(x2)

3. L'idée: intégrer le polynôme a la place de la fonction, car on peut intégrer exactement un polynôme.

B. Formule de Simpson pour un intervalles
L'intégrale du polynôme quadratique donne:

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x0)+4f(x1)+f(x2)]$$

avec

. $h=\frac{x2-x1}{2}$ = demi-largeur de l'intervalle

. Les coefficients 1,4,1 viennent de l'intégration exacte du polynôme

. Le point du milieu x1 a un poids plus fort (4) car il reflete la courbure

C’est déjà plus précis que la méthode des trapèzes, qui ne prend pas la courbure en compte.

C. Étendre Simpson a plusieurs intervalles.

Si on divise [a,b] en n sous-intervalles pairs (n doit etre pair), on applique Simpson sur chaque paire de sous-intervalles :

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x0)+4\sum_{i=1,3,5,...}^{n-1} f(xi)+2\sum_{i=2,4,6,...}^{n-2} f(xi)+f(xn)]$$

avec . $h=\frac{(b-a)}{n}$

. Les indices impairs (1,3,5,…) ont le coefficient 4

. Les indices pairs (2,4,6,…) ont le coefficient 2

. Les extrêmes x0,xn ont le coefficient 1

D. Implémentation en python

1) Cette premiere est la plus simple :

```
import numpy as np
#example usage
a=0
b=2
n=10 #must be even
h=(b-a)/n
x=np.linspace(a,b,n+1)
y=x**2 # example of f(x)=x**2
# start with endpoints
S=y[0]+y[-1]
# Add odd indices (weight 4)
S +=4*np.sum(y[1:-1:2])
# Add even indices (weight 2)
S +=2*np.sum(y[2:-1:2])
# Final result
integral=(h/3)*S
print("Approximate integral:", integral)
```

2) On a une deuxieme implementation:

```
import numpy as np
#  define the function
```

```
def f(x):
    return x**2 # exemple function
# Simpson method
def simpson(a,b,n):
    if n% 2!=0:
        raise ValueError(" n must be even for Simpson's method ")
    h=(b-a)/n
    x=np.linspace(a,b,n+1)
    y=f(x)
# start with endpoints
    S=y[0]+y[-1]
# Add odd indices (weight 4)
    S +=4*np.sum(y[1:-1:2])
# Add even indices (weight 2)
    S +=2*np.sum(y[2:-1:2])
# Final result
    integral=(h/3)*S
    return integral
#example usage
a=0
b=2
n=10 #must be even
result=simpson(a,b,n)
print("Approximate integral:", result)
```

Ici, on va montrer d'où viennent ces coefficients 1,4 et 2

A) Point de départ: interpolation quadratique

On prend trois points: x0,x1, x2
avec x1=x0+h, x2= x0+2h

On exprime f(x) par un polynôme de degré 2: $P_2(x)$ qui passe par les points

(x0,f(x0)), (x1,f(x1)), (x2,f(x2)).

B) Écriture du polynôme (Forme de Lagrange)

Le polynome interpolateur est:

$$P_2(x)=f(x0)L_0(x) +f(x1)L_1(x) +f(x2)L_2(x)$$

avec

$$L_0(x) = \frac{(x-x1)(x-x2)}{(x0-x1)(x0-x2)}$$

$$L_1(x) = \frac{(x-x0)(x-x2)}{(x1-x0)(x1-x2)}$$

$$L_2(x)=\frac{(x-x0)(x2-x0)}{(x2-x1)(x2-x1)}$$

C) Intégration du polynôme:

On veut:

$$\int_{x0}^{x2} f(x)dx \approx \int_{x0}^{x2} P_2(x)dx$$

Donc:

$$\int_{x0}^{x2} P_2(x)dx = f(x0)\int L_0(x)dx + f(x1)\int L_1(x)dx + f(x2)\int L_2(x)dx$$

Toute la clé est ici: Les coefficients viennent des intégrales des L_i(x).

Après calcul, on obtient ;

$$\int_{x0}^{x2} L_0(x)dx = \frac{h}{3}, \int_{x0}^{x2} L_1(x)dx = \frac{4h}{3}, \int_{x0}^{x2} L_2(x)dx = \frac{h}{3}$$

Formule obtenue alors devient

$$\int_{x0}^{x2} f(x)dx = \frac{h}{3}[f(x0) + 4f(x1) + f(x2)].$$

Voilà l'origine des coefficients 1,4 et2.

3.5. Transformation de Fourier, FFT et DFT

3.5.1 Description mathématique.

Une équation aux dérivées partielles (EDP) décrit souvent :

- une diffusion (chaleur)
- une propagation (ondes)

Le problème: Elles sont difficiles à résoudre dans le domaine direct (temps/espace)

Solution clé :

- appliquer la transformée de Fourier
- transformer l'EDP en équation beaucoup plus simplement

Le principe fondamental :
La transformee de Fourier est une application

$F : f(t) \to \hat{f}(\xi)$ telle que $\hat{f}(\xi) = \int_{\infty}^{+\infty} f(t) e^{-2\pi i \xi t} dt$

La transformee inverse est

$$f(t) = \int_{-\infty}^{+\infty} \hat{f}(\xi) e^{2\pi i \xi t} d\xi$$

Il existe plusieurs propriétés fondamentales:

- linéarité,
- translation,
- modulation,
- convolution,
  (très important en traitement du signal) et
- Théorème de Parseval (Conservation de l'énergie).

La magie vient de cette propriété suivante

$$F\left(\frac{d^2 f}{dx^2}\right) = -(2\pi\xi)^2 \hat{f}(\xi)$$

Exemple 1: Équation de la chaleur

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

u(x,t)= température

Étape 1: On transforme seulement en espace:

$$u(x,t) \to \hat{u}(\xi,t)$$

Étape 2: Simplification

On obtient: $\frac{\partial u}{\partial t} = (2\pi\xi)^2 \hat{u}$

On obtient une équation différentielle simple

Étape 3: Resolution

$$\hat{u}(x,t) = \hat{u}(\xi,0) e^{\alpha(2\pi\xi)^2 t}$$

Interprétation:

- Les hautes fréquences disparaissent vite
- le signal devient lisse
- diffusion = filtrage naturel

En principe:

- on diagonalise un opérateur $\frac{d^2}{dx^2}$
- les fonctions $e^{i\xi x}$ sont des vecteurs propres

Donc Fourier = théorie spectrale appliquée.

3.5.2 Introduction à la FFT et à la DFT

La Transformée de Fourier Rapide, appelée **FFT** *Fast Fourier Transform en anglais*, est un algorithme qui permet de calculer des Transformées de Fourier Discrètes (**DFT** *Discrete Fourier Transform en anglais)*.

Dans le cas de NumPy, l'implémentation de la DFT est la suivante: On remplace l'intégrale par une somme discrete:

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left(-2\pi i \frac{mk}{n}\right)$$ avec k= 0,...., n-1.

La DFT inverse est donnée par

$$a_n = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left(2\pi i \frac{mk}{n}\right)$$ avec m = 0, ….., n-1.

Elle diffère de la transformée directe par le signe de l'argument de l'exponentielle et par la normalisation à 1/n par défaut.

Exemples simples

On considère le signal:
f(t) = sin(2π 10t)+sin(2π 40t)

- Quelle est sa transformée de Fourier
- Vérifier avec Python

Correction théorique:

sin(2π ft)= $\frac{e^{2\pi i f t}-e^{-2\pi i f t}}{2i}$

Donc:

- pics en ± 10
- pics en ± 40

Correction Python

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

fs = 500
t = np.linspace(0, 1, fs)

signal = np.sin(2*np.pi*10*t) + np.sin(2*np.pi*40*t)

fft = np.fft.fft(signal)
freq = np.fft.fftfreq(len(signal), 1/fs)

plt.plot(freq[:fs//2], np.abs(fft)[:fs//2])
plt.title("Spectre du signal")
plt.xlabel("Fréquence")
plt.show()
```

Résultat: Tu observes:

- **pic a 10 Hz**
- **pic a 40 Hz**

Maintenant, on passe a un Mini-projet suivant.

Mini-projet : Analyse du signal et filtrage naturel

Le signal est définie par la formule:

signal = $\sin(2\pi.5t)+\sin(2\pi.50t)$

Objectif

- Créer un signal
- Ajouter du bruit

- Utiliser la FFT pour le nettoyer

Explication:

- FFT ---- Passage en frequence
- Suppression des hautes frequences = suppression du bruit
- IFFT ----- Retour du signal propre

Code python.

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

# 1. Signal propre
fs = 1000
t = np.linspace(0, 1, fs)

signal = np.sin(2*np.pi*5*t) + np.sin(2*np.pi*50*t)

# 2. Ajout de bruit
bruit = signal + 0.8*np.random.randn(len(t))

# 3. FFT
fft = np.fft.fft(bruit)
freq = np.fft.fftfreq(len(t), 1/fs)

# 4. Filtrage (garder basses fréquences)
fft_filtre = fft.copy()
fft_filtre[np.abs(freq) > 20] = 0

# 5. Retour au temps
signal_filtre = np.fft.ifft(fft_filtre)

# 6. Affichage
plt.figure(figsize=(10,5))

plt.plot(t, bruit, label="Signal bruité", alpha=0.5)
plt.plot(t, signal_filtre.real, label="Signal filtré", linewidth=2)

plt.legend()
plt.title("Filtrage par FFT")
plt.show()
```

Explication ligne par ligne

1. Import des outils

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

numpy :

- calcul scientifique
- FFT, sinus, tableaux

matplotlib :

- affichage graphique

2. Création du signal propre

```
fs= 1000
```

fréquence d'échantillonnage (1000 points)

signifie :

- 1000 mesures en 1 seconde

```
t = np.linspace(0, 1, fs)
```

vecteur temps :

$t \in [0,1]$

avec 1000 points

```
signal = np.sin(2*np.pi*5*t) + np.sin(2*np.pi*50*t)
```

signal composé de **2 fréquences** :

- 5 Hz (lent)
- 50 Hz (rapide)

Interprétation :

- mélange de deux vibrations

3. Ajout du bruit

```
bruit = signal + 0.8*np.random.randn(len(t))
```

on ajoute du bruit aléatoire

- randn() = bruit gaussien
- 0.8 = intensité du bruit

résultat : signal “sale” (comme en réalité)

4. Transformée de Fourier

```
fft = np.fft.fft(bruit)
```

transforme le signal en fréquences

on passe de :

- temps → spectre fréquentiel

```
freq = np.fft.fftfreq(len(t), 1/fs)
```

génère les fréquences associées
détails :

- len(t) = nombre de points
- 1/fs = pas de temps

5. Filtrage fréquentiel

```
fft_filtre = fft.copy()
```

copie du spectre
pour ne pas modifier l'original

```
fft_filtre[np.abs(freq) > 20] = 0
```

LIGNE CLÉ
on garde seulement les fréquences ≤ 20 Hz
ce que ça fait :

- supprime :
    - bruit haute fréquence
    - composante 50 Hz (en partie)
- garde :
    - 5 Hz (signal principal)

mathématiquement :
on applique un **filtre passe-bas**

6. Retour au signal

```
signal_filtre = np.fft.ifft(fft_filtre)
```

transformée inverse
fréquences → temps
on récupère un signal "nettoyé"

7. Important

```
signal_filtre.real
```

on prend la partie réelle
Pourquoi ?

- calculs en complexes
- signal physique = réel

8. Affichage

```
plt.figure(figsize=(10,5))
```

taille du graphique

```
plt.plot(t, bruit, label="Signal bruité", alpha=0.5)
```

trace le signal bruité

- alpha=0.5 → transparent

```
plt.plot(t, signal_filtre.real, label="Signal filtré", linewidth=2)
```

trace le signal filtré

- plus épais → visible

```
plt.legend()
```

affiche les labels

```
plt.title("Filtrage par FFT")
```

titre du graphique

```
plt.show()
```

affiche tout

Intuition globale

Ce code fait :

1. crée un signal (5 Hz + 50 Hz)
2. ajoute du bruit
3. passe en fréquence
4. supprime certaines fréquences
5. reconstruit le signal

Idée clé à retenir

Le filtrage =
garder certaines fréquences et supprimer les autres

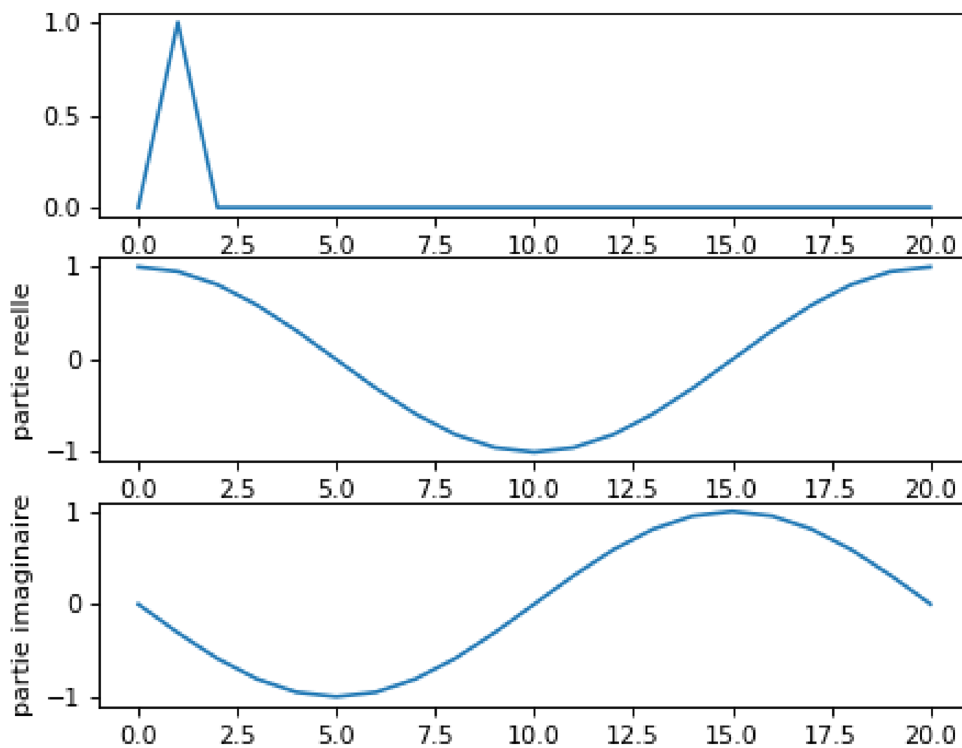
Lecture physique

- bruit = fréquences parasites
- filtrage = nettoyage du signal
- FFT = outil pour voir et agir sur ces fréquences

Dans Python il y a moyen de visualiser la partie réelle et imaginaire de la transformée. Ici, on montre trois exemples.

```
import numpy as np
import matplotlib.pyplot as plt
n = 20
```

```
# definition de a
a = np.zeros(n)
a[1] = 1
# visualisation de a
# on ajoute a droite la valeur de gauche pour la périodicité
plt.subplot(311)
plt.plot( np.append(a, a[0]) )
# calcul de A
A = np.fft.fft(a)
# visualisation de A
# on ajoute a droite la valeur de gauche pour la  périodicité
B = np.append(A, A[0])
plt.subplot(312)
plt.plot(np.real(B))
plt.ylabel("partie reelle")

plt.subplot(313)
plt.plot(np.imag(B))
plt.ylabel("partie imaginaire")

plt.show()
```

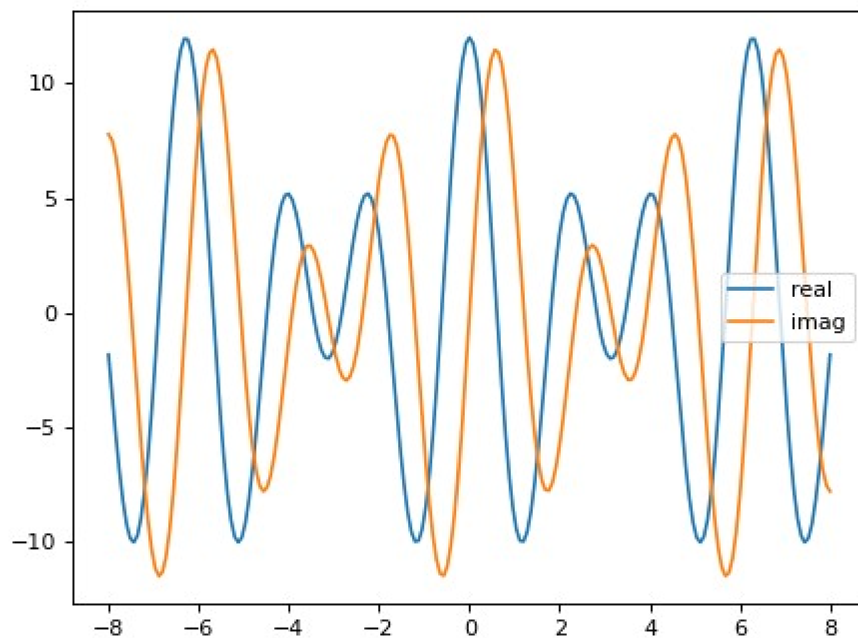


Le deuxième exemple, concerne la visualisation des valeurs complexes avec une échelle colorée avec PyLab

On cherche à visualiser une fonction z= f(x) où x est un réel et z un nombre complexe. On va prendre comme exemple la fonction

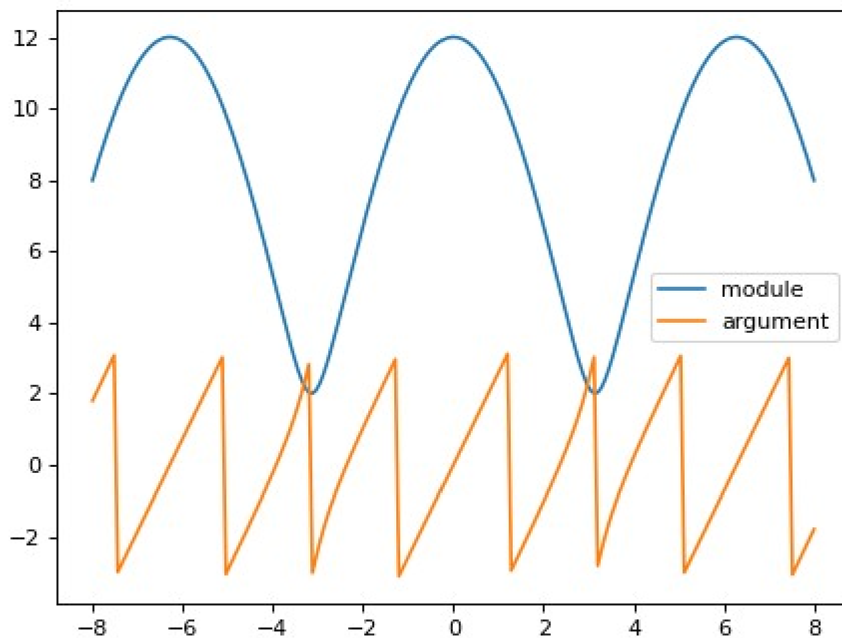
$$f(x)=5e^{2ix}+7e^{3ix}$$

```
from pylab import*
x = linspace(-8,8,201)
z = 5*exp(2j*x) + 7*exp(3j*x)
plot(x,real(z),label='real')
plot(x,imag(z),label='imag')
legend()

show()
```



Le troisième exemple concerne la visualisation du module et de l’argument.

```
from pylab import *

x = linspace(-8,8,201)
z = 5*exp(2j*x) + 7*exp(3j*x)

plot(x,abs(z),label='module')
plot(x,angle(z),label='argument')
legend()

show()
```



Ces trois exemples nous fournissent des outils importants dans le traitement du signal comme nous l’avons vu dans le code python du mini-projet: analyse du signal et filtrage natruel.

Un autre exemple plus pédagogique que le premier est le suivant.

Objectif: Voir clairement comment le filtrage agit.

Exemple : signal lent + bruit rapide
Idée
- signal utile = variation lente (ex : température)
- bruit = variations rapides (parasites)

Donc :

- on garde les basses fréquences
- on supprime les hautes

Code python

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

# Temps
fs = 200
t = np.linspace(0, 2, fs)

# Signal utile (lent)
signal = np.sin(2*np.pi*2*t)

# Bruit haute fréquence
bruit = 0.5*np.sin(2*np.pi*30*t)

# Signal total
signal_bruite = signal + bruit

# FFT
fft = np.fft.fft(signal_bruite)
freq = np.fft.fftfreq(len(t), 1/fs)

# Filtrage : garder basses fréquences
fft_filtre = fft.copy()
fft_filtre[np.abs(freq) > 5] = 0

# Retour au temps
signal_filtre = np.fft.ifft(fft_filtre)

# Affichage
plt.figure(figsize=(10,6))

plt.plot(t, signal_bruite, label="Signal bruité", alpha=0.5)
plt.plot(t, signal, label="Signal original", linestyle="dashed")
plt.plot(t, signal_filtre.real, label="Signal filtré", linewidth=2)

plt.legend()
plt.title("Filtrage simple : enlever le bruit rapide")
plt.show()
```

Pourquoi cet exemple est plus pédagogique ?

1. Fréquences très séparées

- signal utile → **2 Hz** (lent)
- bruit → **30 Hz** (rapide)

facile à comprendre visuellement

2. Filtre simple

```
fft_filtre[np.abs(freq) > 5] = 0
```

on dit :
“je garde seulement ce qui varie lentement”

3. Résultat clair

Tu verras :
- signal bruité → oscillations rapides
- signal filtré → courbe lisse proche de l’original
-

Intuition à retenir (très important)

Le filtrage revient à dire :
- **lent = utile**
- **rapide = bruit**

Voici maintenant un Code complet (avec spectres)
On va voir avant et apres filtrage, pour comprendre exactement ce qui se passe.

Objectif pédagogique
Visualiser :
- le signal dans le temps
- les fréquences présentes (spectre)
- l’effet du filtrage

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

# 1. Temps
fs = 200
```

```
t = np.linspace(0, 2, fs)

# 2. Signal utile (lent)
signal = np.sin(2*np.pi*2*t)

# 3. Bruit rapide
bruit = 0.5*np.sin(2*np.pi*30*t)

# 4. Signal total
signal_bruite = signal + bruit

# 5. FFT
fft = np.fft.fft(signal_bruite)
freq = np.fft.fftfreq(len(t), 1/fs)

# 6. Filtrage (passe-bas)
fft_filtre = fft.copy()
fft_filtre[np.abs(freq) > 5] = 0

# 7. Signal filtré
signal_filtre = np.fft.ifft(fft_filtre)

# ======================
# AFFICHAGE
# ======================

plt.figure(figsize=(12,8))

# --- Signal temporel ---
plt.subplot(2,2,1)
plt.plot(t, signal_bruite)
plt.title("Signal bruité")

plt.subplot(2,2,2)
plt.plot(t, signal_filtre.real)
plt.title("Signal filtré")

# --- Spectre avant ---
plt.subplot(2,2,3)
plt.plot(freq, np.abs(fft))
plt.title("Spectre AVANT filtrage")
plt.xlim(0, 50)
```

```
# --- Spectre après ---
plt.subplot(2,2,4)
plt.plot(freq, np.abs(fft_filtre))
plt.title("Spectre APRES filtrage")
plt.xlim(0, 50)

plt.tight_layout()
plt.show()
```

Comment lire les résultats

1. Signal bruité (en haut à gauche)
   Tu vois :
   - une oscillation lente (signal)
   - petites oscillations rapides (bruit)

2. Signal filtré (en haut à droite)
   Résultat :
   - courbe lisse
   - proche du signal original

3. Spectre AVANT filtrage (bas gauche)
   Tu verras deux pics :
   - **2 Hz** → signal utile
   - **30 Hz** → bruit

   + éventuellement un peu de “flou” autour

4. Spectre APRÈS filtrage (bas droite)
   magie :
   - le pic à **30 Hz disparaît**
   - il reste seulement **2 Hz**

Intuition clé (très importante)

Le spectre te dit :
“de quoi est composé ton signal”
Le filtrage fait :
“je garde certaines fréquences, je supprime les autres”

Petit plus (très utile)
Tu peux changer :

```
fft_filtre[np.abs(freq) > 5] = 0
```

et tester :

- **3 → filtre plus strict**
- **10 → filtre plus souple**

tu verras directement l'effet

On voit maintenant le **Filtrage par FFT = équation de la chaleur (diffusion)**

1. Idée clé

Ce qu'on a fait avec le filtrage :

```
fft_filtre[np.abs(freq) > 5] = 0
```

on **supprime brutalement** certaines fréquences

L'équation de la chaleur fait presque la même chose…
mais de façon **naturelle et progressive** :

$$\hat{u}(\xi,t)=\hat{u}(\xi,0)\,e^{-\alpha(2\pi\xi)^2 t}$$

Traduction simple

Fréquence	Effet	
basse fréquence :	Reste	
haute fréquence :	disparaît	rapidement

EXACTEMENT comme un filtre passe-bas !

Code : filtrage "physique" (type chaleur)
Voici une version améliorée du code

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

# Temps
fs = 200
t = np.linspace(0, 2, fs)

# Signal
signal = np.sin(2*np.pi*2*t)
bruit = 0.5*np.sin(2*np.pi*30*t)
signal_bruite = signal + bruit
```

```
# FFT
fft = np.fft.fft(signal_bruite)
freq = np.fft.fftfreq(len(t), 1/fs)

# Diffusion (filtrage progressif)
alpha = 0.1
T = 1 # "temps de diffusion"

filtre = np.exp(-alpha*(2*np.pi*freq)**2 * T)
fft_diffuse = fft * filtre

# Retour
signal_diffuse = np.fft.ifft(fft_diffuse)

# Affichage
plt.figure(figsize=(12,8))

# Signal
plt.subplot(2,2,1)
plt.plot(t, signal_bruite)
plt.title("Signal bruité")

plt.subplot(2,2,2)
plt.plot(t, signal_diffuse.real)
plt.title("Signal après diffusion")

# Spectres
plt.subplot(2,2,3)
plt.plot(freq, np.abs(fft))
plt.title("Spectre initial")
plt.xlim(0,50)

plt.subplot(2,2,4)
plt.plot(freq, np.abs(fft_diffuse))
plt.title("Spectre après diffusion")
plt.xlim(0,50)

plt.tight_layout()
plt.show()
```

On maintanant voir sur un cas reel de l’**électrocardiogramme (ECG) :**
un cas réel parfait pour comprendre Fourier + filtrage.

1. C’est quoi un ECG ?

Un **ECG** mesure l’activité électrique du cœur.

Un signal ECG contient :

- signal utile → battements du cœur
- bruit → parasites

Types de bruit fréquents

- bruit haute fréquence (machines)
- mouvement du patient
- interférences électriques (50 Hz)

Donc : parfait pour le filtrage !

2. Objectif du mini-projet

On va :

1. simuler un ECG simple
2. ajouter du bruit réaliste
3. analyser le spectre
4. filtrer avec FFT

3. Code complet (ECG simplifié)

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

# 1. Temps
fs = 300 # fréquence d'échantillonnage
t = np.linspace(0, 5, fs*5)

# 2. Simulation ECG simple (battements)
ecg = np.sin(2*np.pi*1.2*t) # rythme cardiaque ~1.2 Hz

# 3. Ajout de bruit réaliste
bruit_haute_freq = 0.3*np.sin(2*np.pi*50*t) # interférence
électrique
bruit_random = 0.2*np.random.randn(len(t)) # bruit aléatoire

signal_bruite = ecg + bruit_haute_freq + bruit_random

# 4. FFT
fft = np.fft.fft(signal_bruite)
freq = np.fft.fftfreq(len(t), 1/fs)
```

```
# 5. Filtrage (passe-bas)
fft_filtre = fft.copy()
fft_filtre[np.abs(freq) > 5] = 0 # garder fréquences utiles

# 6. Retour au signal
signal_filtre = np.fft.ifft(fft_filtre)

# ======================
# AFFICHAGE
# ======================

plt.figure(figsize=(12,8))

# Signal
plt.subplot(2,2,1)
plt.plot(t, signal_bruite)
plt.title("ECG bruité")

plt.subplot(2,2,2)
plt.plot(t, signal_filtre.real)
plt.title("ECG filtré")

# Spectre avant
plt.subplot(2,2,3)
plt.plot(freq, np.abs(fft))
plt.title("Spectre avant")
plt.xlim(0,60)

# Spectre après
plt.subplot(2,2,4)
plt.plot(freq, np.abs(fft_filtre))
plt.title("Spectre après")
plt.xlim(0,60)

plt.tight_layout()
plt.show()
```

Ce que tu vas voir quand on visualise:

Signal ECG bruité

- oscillation principale (~1 Hz)

- perturbée par :
  - bruit rapide
  - oscillation à 50 Hz

Spectre AVANT

Tu verras :
- pic vers **1–2 Hz** → cœur
- pic à **50 Hz** → interférence
- bruit un peu partout

Spectre APRÈS

Après filtrage :
- 50 Hz disparaît
- le cœur reste

Signal filtré
beaucoup plus propre :
- battements visibles
- bruit réduit

Interprétation réelle (très importante)

Ce que tu viens de faire est utilisé en :
- hôpitaux
- montres connectées
- cardio-monitoring

Fréquences typiques ECG

Composant	Fréquence
cœur	~1 Hz
respiration	~0.2 – 0.5 Hz
bruit électrique	50 Hz

Conclusion

On vient de faire :

- **traitement du signal réel**

- **analyse fréquentielle**
- **filtrage utilisé en médecine**

On passe à un niveau très concret et utilisé en pratique médicale : détecter automatiquement les battements du cœur (pics ECG)

1. Idée simple

Dans un ECG, chaque battement correspond à un **pic (appelé complexe QRS)**.

Objectif :

- détecter ces pics
- calculer le rythme cardiaque

2. Principe de l'algorithme (version simple)

On va :

1. filtrer le signal (déjà fait)
2. chercher les **maxima locaux**
3. garder seulement les pics "assez grands"

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

# 1. Temps
fs = 300
t = np.linspace(0, 5, fs*5)

# 2. ECG simplifié
ecg = np.sin(2*np.pi*1.2*t)

# Ajout de bruit
bruit = 0.3*np.sin(2*np.pi*50*t) + 0.2*np.random.randn(len(t))
signal = ecg + bruit

# 3. FFT + filtrage
fft = np.fft.fft(signal)
freq = np.fft.fftfreq(len(t), 1/fs)

fft_filtre = fft.copy()
fft_filtre[np.abs(freq) > 5] = 0

signal_filtre = np.fft.ifft(fft_filtre).real

# 4. Détection des pics
```

```
seuil = 0.5 # seuil à ajuster
pics = []

for i in range(1, len(signal_filtre)-1):
if signal_filtre[i] > seuil:
if signal_filtre[i] > signal_filtre[i-1] and signal_filtre[i] >
signal_filtre[i+1]:
pics.append(i)

# 5. Calcul du rythme cardiaque
temps_pics = t[pics]
intervalles = np.diff(temps_pics)

frequence_cardiaque = 60 / np.mean(intervalles)

# 6. Affichage
plt.figure(figsize=(10,5))
plt.plot(t, signal_filtre, label="ECG filtré")
plt.scatter(t[pics], signal_filtre[pics], color='red',
label="Battements détectés")

plt.legend()
plt.title(f"Fréquence cardiaque ≈ {frequence_cardiaque:.1f} BPM")
plt.show()
```

4. Explication simple

Détection des pics

```
if signal_filtre[i] > signal_filtre[i-1] and signal_filtre[i] >
signal_filtre[i+1]
```

signifie :

- point plus grand que ses voisins
  donc maximum local

Seuil

```
seuil = 0.5
```

évite de détecter :

- du bruit
- de petits pics

Fréquence cardiaque

```
frequence_cardiaque = 60 / np.mean(intervalles)
```

logique :

- intervalle moyen entre battements = durée d’un cycle
- donc :

BPM= 60/durée

5. Résultat attendu
Tu verras :
- courbe ECG propre
- points rouges = battements
- **fréquence ≈ 70–80 BPM (selon simulation)**

6. Limites (important)
Ce modèle est **simplifié**
Un vrai ECG contient :
- plusieurs pics (P, QRS, T)
- formes complexes

7. Version réelle (en pratique)

En médecine, on utilise :
- filtrage avancé
- dérivées du signal
- algorithmes type **Pan-Tompkins**

Lien avec Fourier
Ce qu’on a fait :
- FFT → nettoyer le signal
- analyse temporelle → détecter les pics

combinaison des deux mondes :
- fréquence
- temps

Approximation de la Transformée de Fourier

Quand on approxime une transformée de Fourier, qu’est-ce qu’on cherche vraiment ?

1. Le problème de départ
La vraie transformée de Fourier est:

$$\hat{f}(\xi)=\int_{-\infty}^{+\infty} f(t)\, e^{-2\pi i \xi t}\, dt$$

Mais en pratique :
- on ne connaît pas f(t) partout
- on a seulement :
  - des **valeurs discrètes**
  - sur un **intervalle fini**

2. Donc on veut approximer quoi ?

On veut approximer :

$\hat{f}(\xi)$ (le contenu fréquentiel du signal)

c’est-à-dire :

combien de chaque fréquence est présente dans le signal

3. Concrètement

Si tu as un signal :

f(t) = données mesurées (ECG, son, etc.)

l’approximation te donne :

- les fréquences dominantes
- leur amplitude
- leur rôle dans le signal

4. Comment on approxime ?

On remplace l’intégrale par une somme :

$$X_k = \sum_{n=0}^{N-1} f(t_n) e^{\frac{-2\pi i k_n}{N}}$$

C’est exactement la **DFT (transformée discrète)**

5. Ce qu’on cherche vraiment (intuition)

On projette le signal sur :

$$e^{2\pi i \xi t}$$

des “ondes de base”

Donc :

approximer la transformée = **décomposer le signal en ondes simples**

6. Interprétation très concrète

Quand tu fais : `np.fft.fft(signal)`

tu obtiens :

- un tableau de nombres complexes
- chaque case = une fréquence

et `np.abs(fft)`

te donne :

“combien de cette fréquence est présente”

Bref,

Approximer la transformée de Fourier =

passer d’un signal
à ses **fréquences**

Exemple : reconnaître une fréquence dans un signal

Idée
On va prendre un signal très simple :
$$f(t)=\sin(2\pi.2t)$$
donc :

- **une seule fréquence : 2 Hz**

1. Ce qu’on veut faire

Approximer la transformée de Fourier revient à poser la question :
“Est-ce que le signal contient la fréquence 1 ? 2 ? 3 ? …”

2. Version intuitive (sans code)

On teste une fréquence ξ en calculant: $\int f(t)e^{-2\pi i\xi t}dt$
interprétation :

- si ξ=2 → ça “colle” → résultat grand
- sinon → ça oscille → résultat ≈ 0

3. Version Python pédagogique

On va **tester quelques fréquences à la main** :

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

# Temps discret
t = np.linspace(0, 1, 100)

# Signal (2 Hz)
signal = np.sin(2*np.pi*2*t)

# Fréquences testées
frequences = [1, 2, 3, 4]

resultats = []
```

```
for f in frequences:
projection = np.sum(signal * np.exp(-2j*np.pi*f*t))
resultats.append(np.abs(projection))

# Affichage
plt.bar(frequences, resultats)
plt.xlabel("Fréquence testée")
plt.ylabel("Amplitude")
plt.title("Approximation de la transformée de Fourier")
plt.show()
```

4. Résultat attendu

Tu verras :

- fréquence 1 → presque 0
- fréquence 2 → **grand pic**
- fréquence 3 → presque 0
- fréquence 4 → presque 0

5. Interprétation

Ce code fait exactement ça :

“je teste chaque fréquence et je mesure si elle est présente”

6. Ce que tu viens de faire (important)

Tu as approximé : $\hat{f}(\xi)$

en remplaçant l’intégrale par: $\sum f(t_n) e^{-2\pi i \xi t}$

7. Version ultra intuitive

Imagine :

- ton signal = musique
- chaque fréquence = une note

la transformée de Fourier répond :

“quelles notes sont présentes ?”

8. Lien avec FFT

Ce que tu as fait à la main :

```
for f in frequences:
```

la FFT le fait :

- pour **toutes les fréquences**
- **très rapidement**

Chapitre 4. Résoudre une équation différentielle ordinaire (EDO) avec Scipy

4.1. Résoudre numériquement une EDO d'ordre un avec le langage python

Une équation différentielle ordinaire (**EDO)** est une équation de la forme

$$dy/dt = f(y,t)$$

où l'inconnue est la fonction y à n composantes:

$$y:\ t \in IR \rightarrow y(t) \in IR,$$

alors que la fonction $f: (y,t) \in IR \times IR \rightarrow IR$ et la condition initiale $y(0) \in IR$ sont supposées connues.

Exemple: y'= y+1 ou dy/ dt = y+1

Référence de **odeint** ou reference de ode.https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.integrate.odeint.html

Le but est d'introduire les méthodes de base permettant de résoudre des équations différentielles ordinaires du premier ordre du type:

dy(t)/dt= f(y,t) avec la condition initiale y(0)=y0.

On va utiliser une fonction appelée **odeint() ou solve_ivp**.
Odeint ou solve_ivp sont des fonction de Scipy.
Le principe est de donner les conditions initiales y(0) , d'écrire la fonction f(y,t) , et de donner un tableau de valeurs du temps (t_j) $j=1 \rightarrow N$. En retour la fonction odeint() renvoit un tableau de valeurs $(y(t_j)) = 1 \rightarrow N$ tel que y(t) est la solution de l'ODE dy/dt =f(y,t) avec les conditions initiales spécifiées.
On peut passer des paramètres extérieurs à la fonction.

Exemple 1

On considère l'équation différentielle ordinaire à n=1 composante suivante où l'inconnue est la fonction y:t ∈ R → y(t) ∈ R :

dy/ dt =y(t) sin(t)

Remarque 4.2.1.
L'équation s'écrit dy/dt =f(y,t) avec la fonction est f(y,t) =y(t) sin(t) .

La solution est $y(t) = y(0)e^{1-\cos(t)}$ **.**

Ecrire:

```
%matplotlib inline
from scipy.integrate import odeint #ou solve_ivp instead of
                                          #odeint
from pylab import*
from numpy import*

y_in = 0.1 # condition initiale
tmax= 50 # intervalle de temps
N = 501 # nombre de points en temps

def f(y,t):
     return y*sin(t) #on peut changer en y*cos(t)-t pour voir
ce qui change

t = linspace(0,tmax,N) # tableau de t = 0->tmax avec N valeurs
y = odeint(f, y_in,t)
plot(t,y)
xlabel('$t$')
ylabel('$y(t)$')
title("Solution de $y'(t)=y sin(t)$")
```

Résultat: On peut visualiser le résultat avec spyder

Exemple 2

On veut résoudre un système d'équations différentielles

$$\begin{cases} x' = y \\ y' = -x \end{cases}$$

On l’implemente comme suit :

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt


#Equation differentielle

def system(Y,t):
    x,y= Y
    return [y,-x]
#temps
t= np.linspace(0,10,100)
Y0=[1,0]

solution=odeint(system,Y0,t)
x=solution[:,0]
y=solution[:,1]
plt.plot(x,y)
plt.xlabel(‘’y’’)
plt.ylabel(‘’x’’)
plt.title(‘’Trajectoire circulaire avec odeint’’)
plt.show()
```

La solution ressemble a

```
solution = [
[x0,y0],
[x1,y1]
[x2,y2],
...
]
```

Chaque ligne =un instant t_i
chaque colonne = une variable
Ici la notation
x=solution[:,0]

signifie que le « 0 » veut dire « colonne 0 : première colonne» et signifie « prends toutes les valeurs de x »

Même chose pour y=solution[:,1]: colonne 1 pour y.

4.2. Généralités sur les équations différentielles ordinaires (EDO)

Principe de base:

Transformer une équation différentielle d'ordre d en EDO

Prenons un exemple. L'équation du deuxième ordre en temps:

$d^2y/dt^2 = dy/dt + \sin(y)$ (qu'on peut l'écrire également comme suit:

$y'' = y' + \sin(y)$)

peut se ré-écrire comme une équation du premier ordre $dY/dt=F(Y,t)$ à deux variables en posant $Y=(y_1,y_2)$ et $y_1 = y$, $y_2 = dy/dt$ et $F(Y,t) = (y_2, y_2 + \sin(y_1))$.

Plus généralement, une équation différentielle d'ordre d à 1 variable t :

$$d^dy/dt^d = f(y(t), dy/dt, \dots, d^{d-1}y/dt^{d-1}, t)$$

peut se réécrire comme une équation d'ordre 1 avec d variables $Y=(y_1(t), \dots, y_d(t))$.

Il suffit de poser $y_1(t) = y(t)$, $y_2(t) = dy/dt$, … , $y_d(t) = d^{d-1}y/dt^{d-1}$,
c'est à dire

$$dy_1/dt\,(t) = y_2(t), \dots, dy_{d-1}(t)/dt = y_d(t), dy_d(t)/dt = f(y_1, y_2, \dots, y_{d-1}, t).$$

C'est à dire, l'écrire comme un système de d équations différentielle du premier degré.

Ainsi l'équation se ré-écrit:

$$dY/dt=F(Y,t)$$

avec $F(Y,t) = (y_2, y_3, \dots y_d, f(y_1, \dots y_{d-1}, t))$.

Exemple 1

Équation du second ordre: forme $y'' = ay$

Ça va se compliquer un peu : la fonction odeint se sait résoudre que des équations d'ordre 1 mais peut en résoudre plusieurs d'un coup. Autrement dit, on peut résoudre Y'= A.Y où Y est un vecteur et A une matrice.
Ainsi, si on veut résoudre l'équation différentielle Y''= A.Y , on va poser Y= (y,y') et notre équation d'ordre 2 se ramène à résoudre Y'=(y',a.y), qu'on peut écrire en colonne.
Ce qui nous donnerait comme code :

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
a=-1
temps = np.linspace(0, 50,1000)
# L'équation différentielle sous forme de fonction
def equation(Y,temps):
         (y,dy)=Y
         return [dy,a*y]
# Pour que odeint renvoit séparément les valeurs de Y et de Y', il faut rajouter .T à la fin
Y,dY=odeint(equation,[1,0],temps).T
plt.plot(temps,Y)
plt.show()
```

Détaillons les changements par rapport à l'ordre 1 :

- Cette fois-ci, dans la fonction équation, Y désigne un vecteur qu'on explicite par la commande (y,dy)=Y pour pouvoir les utiliser par la suite les valeurs y et dy.
  On explicite ce qu'elle renvoi c'est à dire :
- comme expliqué juste avant.
- odeint **attend comme pour l'ordre 1 une fonction qui caractérise l'équation différentielle, des valeurs initiales et la liste des temps mais cette fois-ci renvoi la liste des couples (y,dy) pour chaque temps. Or en général on veut les valeurs de y d'un coté et les valeurs de dy de l'autre. Pour cela, on rajoute** .T **(qui fait simplement une transposition) à la fin de la ligne.**

Exemple 2

Cet exemple peut être généralisée à tout ordre.

Considérons une EDO d'ordre 3: y'''+2y''-y'+y=0. Cette EDO doit être transformée en un système d'équations d'ordre un. Ainsi, on pose
y0 = y, y1 = y', y2 = y''
et on obtient le système suivant

$$\begin{cases} dy0 = y1 \\ dy1 = y2 \\ dy2 = -2y2 + y1 - y0 \end{cases}$$

Le code python est le suivant:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Definition du systeme

def edo(y,t):
    y0,y1,y2 = y  # y,y',y''
    dy0dt=y1
    dy1dt= y2
    dy2dt= 2*y2+y1-y0
    return [dy0dt, dy1dt,dy2dt]

# conditions initiales
y0_init= 1.0 # y(0)
y1_init=0.0 # y'(0)
y2_init= 0.0 # y''(0)

Y0= [y0_init, y1_init,y2_init]

# Temps
t= np.linspace(0,10,1000)

#Resolution

solution = odeint(edo,Y0,t)

#Extraction de y(t)
y=solution[:,0]

# Affichage
```

```
plt.plot(t,y)
plt.xlabel('t')
plt.ylabel('y(t)')
plt.title(''Solution de l'EDO d'ordre 3'')
plt.grid()
plt.show()
```

Dans ce code, la fonction edo(y,t) prend

- y: vecteur [y,y',y'']
- t: variable temps
- et retourne les dérivées

Exemple 3

On considère l'équation différentielle ordinaire à n=3 composantes suivante appelées équations de Lorenz.
L'inconnue est la fonction $y:t \in R \rightarrow y(t)=(y0(t),y1(t),y2(t)) \in R^3$: système de 3 équations du premier degré suivantes:

$dy_0/dt = -\sigma(y_0(t)-y_1(t))$

$dy_1/dt = \rho y_0(t)-y_1(t)-y_0(t)y2(t)$

$dy_2/dt = -\beta y_2(t)+y_0(t)y_1(t)$

On connaît les conditions initiales y(0) et les paramètres $\sigma = 10$, $\beta = 8/3$, $\rho = 28$.

Ecrire:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

sigma, beta, rho = 10, 8/3., 28 # parametres
y_in = [0, 1, 1.05] # Conditions initiales
tmax= 100 # intervalle de temps
N = 10000 # nombre de points en temps

def f(y,t,sigma, beta, rho):
   """The Lorenz equations."""
   f0 = -sigma*(y[0] - y[1])
   f1 = rho*y[0] - y[1] - y[0]*y[2]
   f2 = -beta*y[2] + y[0]*y[1]
   return f0,f1,f2
```

```
    t=linspace(0,tmax,N) # tableau de t = 0->tmax avec N valeurs
    y = odeint(f,y_in,t,args=(sigma,beta,rho))

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot(y.T[0], y.T[1], y.T[2])


 # rotate the axes and update
    for angle in range(0, 360):
          ax.view_init(30, angle)
          plt.draw()
          plt.pause(.001) # montre image

 plt.show()
```

Résultat: (on a fabriqué ce fichier gif animé d'après la Section 2.2.6 avec gifsicle)

On peut visualiser le résultat avec Spyder

Si on veut résoudre sans utiliser les fonctions odeint ou solve_ivp, on fait recours par exemple aux méthodes de résolution d'approximation.

Exemple 4: **Méthode d'Euler**

a) Introduction

Cette méthode d'Euler consiste à résoudre de manière approchée une EDO en discrétisant le temps avec un pas h et d'approximer la dérivée temporelle sur chaque intervalle de longueur h. Il y a deux façons de faire:

La première est d'approximer par différence finie avant:

$$\dot{x}(t) \approx \frac{x(t+h)-x(t)}{h}$$

et la seconde par différence finie arrière:

$$\dot{x}(t) \approx \frac{x(t)-x(t-h)}{h}$$

Les inconnues etant les évaluations de la solution aux temps $t_i = ih$ pour i ≥ 0, cad
x_i= x(t_i). L'équation différentielle peut ainsi être approchée a l'aide des différences finies avant par:

$$\frac{x_{i+1}-x_i}{t_{i+1}-t_i}=f(t_i,x_i)$$

Ce qui donne la formule d'Euler explicite: $x_{i+1}=x_i+(t_{i+1}-t_i).f(t_i,x_i)$.

Avec l'approximation par différences finies arrières, on obtient la méthode d'Euler implicite:

$$x_i=x_{i-1}+(t_i-t_{i-1}).f(t_i,x_i)$$.

La formule d'Euler explicite permet de calculer directement tous les x_i par récurrence en connaissant x_0. En revanche la formule d'Euler implicite nécessite a chaque pas de temps la résolution d'une équation non linéaire pour x_i par exemple avec la méthode de Newton.

Voici un exemple:

f(t,x)= sin(t)-x # x(t): x fonction de t
On veut résoudre l'EDO:
\dot{x}(t)/dt = sin(t)-x

Le programme de résolution est le suivant:

```
import numpy as np
import matplotlib.pyplot as plt
def euler_explicit(f,x0,t):
     x=np.zeros((len(t),len(x0)))
     x[0]=x0
     for i in range(len(t)-1):
          x[i+1]=x[i]+(t[i+1]-t[i])*f(t[i],x[i])
     return x
    # definit les données du problème
f= lambda t,x : np.sin(t)-x
x0=np.array([1]) #taille de la figure
plt.figure(figsize=(8,5))
plt.title(r"Solution de l'equation $\dot{x}+x=\sin(t)$ par methode d'Euler explicite")
#differentes discretisations
```

```
for N in [10,20,50,100]:
    t= np.linspace(0,10,N)
    sol=euler_explicit(f,x0,t)
    plt.plot(t,sol,label=f"solution avec {N} points")

exact = (np.sin(t)-np.cos(t)+3*np.exp(-t))/2
plt.plot(t,exact,label="solution exacte")
plt.legend()
```

Résultat: On peut le visualiser avec spyder

ou bien on peut faire comme suit.

b) Principe de la méthode d'Euler

Les ED linéaires d'ordre 1 peuvent bien sûr s'écrire sous la forme

$$y'(t)=F(y(t),t)$$

avec $F:\mathbb{R}^2\to\mathbb{R}$ une fonction de classe C^1 sur un intervalle $[t_f,t_0]$.

Exemples:

- l'équation $y'(t)=(1+t^2)y(t)$ peut s'écrire $y'(t)=F(y(t),t)$ avec $F(a,b)=(1+b^2)a$
- l'équation $y'(t)=y(t)$ peut s'écrire $y'(t)=F(y(t),t)$ avec $F(a,b)=a.$

L'idée principale est que «localement la courbe de la fonction y ressemble à sa tangente». Ainsi si h est proche de 0, on a

$$y(t_0+h)\approx y(t_0)+hy'(t_0)=y(t_0)+hF(y(t_0),t_0)$$

Donc $y(t_0+h)$ peut être approchée par la quantité $y(t_0)+hF(y(t_0),t_0)$

On découpe ainsi l'intervalle de temps $[t_f,t_0]$ en n segments de même longueur

$h=\frac{t_f-t_0}{n}$ (on dit que h est le pas). On dispose ainsi de n + 1 temps $t_k=a+kh$ pour $k\in\{0,1,\dots,n\}$.

On va alors approximer la solution y à l'instant t_k par le nombre y_k défini par la relation de récurrence:

$$y_{k+1}=y_k+hF(y_k,t_k)$$

On initialise enfin avec la condition initiale $y_0=y(t_0)$.

```
"""Données:
F(y,t) une fonction t0,t1 deux réels avec t0 < t1
y0 un réel n un entier
Résultat: le tuple constitué de la liste des temps [t0,...,tn] et
la liste des (qui constituent une approximation de la solution y
sur [t0,tf]  de l'ED y'=F(y,t) avec la condition initiale y(t0) =
y0
"""
def euler(F,t0,tf,y0,n):
               h = (tf-t0)/n
               y = y0
               t = t0
               Y = [y0]
               T = [t0]
               for k in range(n): # n itérations donc n+1 points
                        y = y + h*F(y,t)
                        t = t + h
                        Y.append(y)
                        T.append(t)
               return T,Y
```

La quantité $h=\frac{(t_f-t_0)}{n}$ n est appelé le pas. Plus le pas est petit, meilleure sera l'approximation.

Exemple concret

On veut résoudre des ED du type y'(t) = F (y(t),t) avec y(t0) = y0. Elle prend en argument la fonction F , la condition initiale y0 et une liste de temps commençant à t0. Par exemple, pour résoudre y' = y(1-y) sur [0, 1] avec y(0) = 1. On pourra écrire le script suivant:

```
import numpy as np
import matplotlib.pyplot as plt
def F(y,t):
      return y*(1-y)
#parametres
t0=0
y0=0.1
h=0.1
n=100
#Listes pour stocker les resultats
t_values=[t0]
y_values=[y0]
# Methode d’Euler
t=y0
y=y0
for i in range(n):
     y= y +h*F(t,y) # formule y_{n+1}=y_n+h*F(t_n,y_n)
     t= t+h
     t_values.append(t)
     y_values.append(y)
# Affichage du graphique
plt.plot(t_values, y_values)
plt.xlabel(«t«)
plt.ylabel(«y»)
plt.title(«Methode d’Euler:y(t)»)
plt.show()
```

4.3. Applications

a) Mouvement d'une planète

Voici par exemple le mouvement d'une planète autour d'un soleil si on ne considère que la force de gravitation provenant du soleil. On place l'origine du repère sur le soleil.

On travaille en 2D, on résous le système d’équations différentielles suivant:
$x'=v_x$, $y'=v_y$, $v'_x= -GM/r^3\, x$, $v'_y= -GM/r^3\, y$ avec $r=(x^2+y^2)^{1/2}$
et GM est une constante qu’on prend egale a 1 pour simplifier.
Le code python est le suivant:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
GM=1
def system(Y,t):
    x,y,vx,vy = Y
    r=np.sqrt(x**2+y**2)
    dxdt=vx
    dydt=vy
    dvxdt=-GM*x/r**3
    dvydt=-GM*y/r***3
    return[dxdt,dydt,dvxdt,dvydt]
t=np.linspace(0,20,1000)
x0=1
y0=0
vx0=0
vy0=1
Y0=[x0,y0,vx0,vy0]

solution= odeint(system,Y0,t)
# Extraction des résultats
x=solution[ :,0]
y=solution[ :,1]

# Affichage
plt.plot(x,y)
plt.scatter(0,0,label="Soleil")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Orbites d’une planete")
ple.axis("equal")
plt.legend()
plt.grid()
plt.show()
```

On peut également utiliser le code python suivant:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
temps = np.linspace(0, 100,10000)
# On considère un force en 1/r² avec des conditions initiales non
réalistes
```

```
def eq_mouvement(mobile,temps):
              x,y,dx,dy=mobile
              return
[dx,dy,-x/(x**2+y**2)**1.5,-y/(x**2+y**2)**1.5]

X,Y,dX,dY=odeint(eq_mouvement, (1,0,0,0.5), temps).T
plt.plot(X,Y)
plt.grid()
plt.show()
```

Remarque : Ici, pour résoudre, on décompose selon les axes x et y et on résout en réalité en même temps les équations en x et en y qu'on ramène à de l'ordre 1. On se ramène donc à une équation différentielle d'ordre 1 mais d'un vecteur de dimension 4.

b) Étude du mouvement d'un projectile

Étudions le mouvement d'un projectile en considérant d'un coté le mouvement sans frottement et de l'autre avec frottement pour pouvoir les comparer.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
from math import *
temps = np.linspace(0, 50,1000)
g=9.81
# On ne considere que la pesanteur
def eq_mouvement(mobile,temps):
       x,y,dx,dy=mobile
       return [dx,dy,0,-g]
X,Y,dX,dY=odeint(eq_mouvement, (0, 30,10,10), temps).T
# On ne garde que les valeurs de Y positives
indice=0
for y in Y :
        if y<0:
            break
        indice+=1
plt.plot(X[:indice+1],Y[:indice+1], label="Sans frottement")
# Si on considere des frottements -Cv*vec(v)
C=0.001
def eq_mouvement_frottement(mobile,temps):
              x,y,dx,dy=mobile
              v=np.sqrt(dx**2+dy**2)
```

```
              return [dx,dy,-C*v*dx,-g-C*v*dy]
Xf,Yf,dXf,dYf=odeint(eq_mouvement_frottement, (0, 30,10,10), temps).T
# On ne garde que les valeurs de Yf positives
indice=0
for y in Yf :
      if y<0:
         break
    indice+=1
plt.plot(Xf[:indice+1],Yf[:indice+1],label="Avec frottement")
plt.legend()
plt.show()
```

Quelques explications :

- La démarche est exactement la même que dans le cas d'une planète : on décompose selon les x et les y et on se ramène à résoudre une équation différentielle d'ordre 1 pour un vecteur de dimension 4.
- Petite nuance purement esthétique : avant d'afficher le résultat, on ne garde que les valeurs qui donne une ordonnée positive car le projectile touche le sol sinon donc les valeurs négatives n'ont pas de sens.
-

c) Étude du mouvement d'un projectile avec vent latéral

Rajoutons au cas précédent un petit vent latéral et observons le résultat en 3D.
Pour pouvoir mieux observer le résultat, il vaut mieux copier-coller le code dans un interpréteur (comme Edupython par exemple).

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
from math import *
from mpl_toolkits import mplot3d

axes=plt.axes(projection='3d')
temps = np.linspace(0, 20,5000)
g=9.81

def eq_mouvement(mobile,temps):
         x,y,z,dx,dy,dz=mobile
         return [dx,dy,dz,sin(2*np.pi*temps),0,-g]
```

```
X,Y,Z,dX,dY,dZ=odeint(eq_mouvement, (0,0, 30,0,10,10), temps).T
# On ne garde que les valeurs de Y positives
indice=0
for z in Z :
     if z<0:
     break
     indice+=1

axes.plot3D(X[:indice+1],Y[:indice+1],Z[:indice+1])
plt.show()
```

d) Étude d'un ressort

Voici un code permettant d'observer l'évolution d'un ressort de manière animée avec matplotlib. Le problème est que sur ce site, il est impossible de voir des animations, il faudra donc le copier coller dans un interpréteur sur votre ordi (comme Edupython par exemple) pour voir le résultat.
Petit nuance par rapport aux cas précédents : Au lieu de résoudre l'équation différentielle du mouvement sur un intervalle de temps fixé une bonne fois pour toute et observer le résultat, ici, on résout une équation différentielle au fur et à mesure de l'animation sur un petit intervalle (noté dt) en prenant comme valeur initiale à chaque étape le résultat de l'étape précédente. Cela permet de ne pas fixer a priori la fin de l'animation même si on sait que plus le temps est grand plus les erreurs dans la résolution de l'équation différentielle deviennent grandes et donc ce qui s'affiche s'éloigne de la réalité.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

# On fixe nos constantes
longueur_vide= 0.40
k=20
longueur_initiale=0.5
duree=10 # durée de l'animation

fps=30 # 30 images par secondes en général
dt=1/fps # ecart de temps entre chaque image
g=9.81
y0=-longueur_initiale
```

```
v0=0
Dt=np.array([0,dt]) # l'intervalle de temps entre 2 mises à  jour

def eq_mouvement(mobile,temps):
       y,dy=mobile
       return [dy,-g+k*(-y-longueur_vide)]

#initialisation du graphique
t=0
line,=plt.plot([],[],"o-")
plt.ylim(-2,0.1) # On fixe notre axe y
texte = plt.text(-0.05, -0.01, '')

# Toutes les dt sec, on calcule les nouvelles coordonnées a  partir des anciennes et on l'affiche
while t<duree:

#On part d'une longueur_0 et d'une vitesse nulle
   L,DL=odeint(eq_mouvement, (y0,v0), Dt).T
   y0=L[1]
   v0=DL[1]
   line.set_data([0,0],[0,y0])
   texte.set_text("temps = {:0.3f} s".format(t))
   plt.pause(dt)
   t+=dt
plt.show()
```

e) Étude d'un pendule

Voici un code pour regarder l'évolution d'un pendule. Plus précisément, on regarde la différence entre la résolution de l'équation complète et l'équation simplifiée (où on suppose sin(theta)=theta).
Equations du mouvement :

$$\theta''+\frac{g}{L}\sin(\theta)=0 \quad \text{ou en systeme} \quad \theta'=\omega \text{ et } \omega'=\frac{-g}{L}\sin(\theta)$$

Comme pour le ressort, c'est une animation et pour la voir il faut copier-coller le code dans un interpreteur (comme Edupython par exemple).

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

```
from scipy.integrate import odeint

g=9.81
L=1.0

def pendule(y,t,g,L):
     theta, omega,=y
     dtheta_dt=omega
     domega_dt=-(g/L)*np.sin(theta)
     return[dtheta_dt,domega_dt]

theta0=nppi/4
omega0=0.0
y0= [theta0,omega0]

t=np.linspace(0,10,250)
solution= odeint(pendule,y0,t,args=(g,L))
theta=solution[:,0]

x=L*np.sin(theta)
y=-L*np.cos(theta)

plt.figure()
plt.plot(x,y)
plt.title(‘’Trajectoire’’)
plt.xlabel(‘‘x’’)
plt.ylabel(‘‘y’’)
plt.axis(‘’equal’’)
plt.grid()
plt.show()
```

Code python avec animation sans odeint.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.animation import FuncAnimation

g=9.81
L=1.0
dt=0.05

theta=np.pi/4 # angle initial 45 degres
omega=0.0 #vitesse angulaire
```

```
xdata,ydata=[],[] # stackage
fig,ax=plt.subplots() # creation figure
line,=ax.plot([],[],'o-',lw=2)
ax.set_xlim(-1.2,1.2)
ax.set_ylim(-1.2,1.2)
def update(frame):
    global theta,omega

    alpha=-(g/L)*np.sin(theta) # Equation du mvt
    omega+=alpha*dt
    theta+=omega*dt
    x=L*np.sin(theta) #position
    y=-L*np.cos(theta)
    line.set_data([0,x],[0,y])
    return line
ani=FuncAnimation(fig,update,frames=200, interval=50)
plt.title("Pendule simple")
plt.show()
```

f) Étude du problème des trois corps

Voici un code pour regarder l'évolution d'un système composé de 3 corps. Comme précédemment, il faut copier coller le code dans un interpréteur pour voir l'animation.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
from math import *

# Constantes
m1,m2,m3,xi1,yi1,xi2,yi2,vix1,viy1,vix2,viy2,duree=10**13,10**13,2*
10**13,1,0,0,1,0,20,-20,0,20
G=6.6742*10**(-11)
temps = np.linspace(0, duree,300*duree)

def donner_force(m1,x1,y1,m2,x2,y2):
#donne les coordonnées de la force qu'exerce l'objet2 sur l'objet 1
temp=G*m1*m2/((x2-x1)**2+(y2-y1)**2)**1.5 # C'est juste pour éviter
deux fois le même calcul
    return (x2-x1)*temp,(y2-y1)*temp
# Equation vérifiée par les 3 objets
def eq_mouvement(mobile,temps):
```

```
      x1,y1,x2,y2,dx1,dy1,dx2,dy2=mobile
# Comme on est dans le repère barycentrique, on peut déduire les coordonnées de l'objet 3 :
      x3=-(m1*x1+m2*x2)/m3
      y3=-(m1*y1+m2*y2)/m3
# Les différentes forces exercées sur 1:
      F12x,F12y=donner_force(m1,x1,y1,m2,x2,y2)
      F13x,F13y=donner_force(m1,x1,y1,m3,x3,y3)
# Les différentes forces exercées sur 2:
    F21x,F21y=-F12x,-F12y
    F23x,F23y=donner_force(m2,x2,y2,m3,x3,y3)
    return [dx1,dy1,dx2,dy2,(F12x+F13x)/m1,(F12y+F13y)/m1,
(F21x+F23x)/m2,(F21y+F23y)/m2]
X1,Y1,X2,Y2,dX1,dY1,DX2,DY2=odeint(eq_mouvement,
(xi1,yi1,xi2,yi2,vix1,viy1,vix2,viy2), temps).T
X3=-(m1*X1+m2*X2)/m3
Y3=-(m1*Y1+m2*Y2)/m3
line,=plt.plot([0,0,0],[0,0,0],".")
#plt.plot(0,0,"ro")
plt.xlim(min(min(X1),min(X2),min(X3))-
5,max(max(X1),max(X2),max(X3))+5)
plt.ylim(min(min(Y1),min(Y2),min(Y3))-
5,max(max(Y1),max(Y2),max(Y3))+5)
#On dessine les points
for i in range(len(X1)):
    line.set_data([X1[i],X2[i],X3[i]],[Y1[i],Y2[i],Y3[i]])
    plt.pause(1/30) # Pour faire une pause entre chaque image

plt.show()
```

Chapitre 5: Calcul symbolique

Python n’inclut pas par défaut certains concepts mathématiques. Un exemple déjà vu concerne les vecteurs et les matrices numériques qui sont implémentés dans le module Numpy. Le but ici est d’introduire le module Sympy qui permet de faire du calcul symbolique. Par exemple, $\sqrt{8}$ est représenté par défaut en python comme un flottant. L’avantage de Sympy est que $\sqrt{8}$ est gardé en tant racine et même automatiquement simplifié.

Dans l'utilisation du module sympy, il est important de noter que lorsqu'on fait un appel à une fonction mathématique qui fait partie du module sympy, comme sin, cos, exp, etc... ces fonctions n'ont pas nécessairement les mêmes comportements que les mêmes fonctions dans un autre module (par exemple **math** ou **numpy**), c'est donc une bonne pratique d'importer les modules en leur donnant un surnom comme **sp** pour sympy, **np** pour numpy , etc... Ainsi l'appel d'un sinus sur une variable symbolique par **sp.sin(x)** et sin(45) auront chacun le résultat attendu.
Étudiez les trois exemples suivants afin de bien comprendre le comportement des opérations sin(x)

5.1. Symboles et expressions symboliques

Avant de pouvoir utiliser des variables symboliques, il faut les declarer comme symboles :

```
x= sp.Symbol(‘’x”) # définit le symbole x
y=sp.Symbol(‘’y”, real=True) # définit la variable réelle y
e=sp.Symbol(r”\varepsilon”, real=True, positive=True) # definit epsilon positif
```

Ensuite, il est possible de faire des opérations entre symboles :

```
import sympy as sp
from math import *
from IPython.display import *
%matplotlib inline
sp.init_printing(use_latex=True)
x=sp.symbols('x')
ysymbolique=sp.sin(x)
x = 45
ynumerique=sin(x)
display(ysymbolique)
display(ynumerique)
Le resultat sera de la forme: - sin(x)
```

-

0.8509034255341184
Les expressions qui peuvent être écrites comme une somme ou un produit peuvent être construites avec sy.summation() ou sy.product(), respectivement. Chacune de ces fonctions accepte une expression qui représente un terme de la somme ou du produit, puis un tuple indiquant la variable d'indexation et les valeurs qu'elle

doit prendre. Par exemple, le code suivant construit la somme et le produit donnés ci-dessous.

```
import sympy as sp
from math import *
x, y, i = sp.symbols('x y i')
sp.summation(x + i*y, (i, 1, 4)) # Sum over i=1,2,3,4.

  Solution: 4*x + 10*y
 sp.product(x + i*y, (i, 0, 5)) # Multiply over i=0,1,2,3,4,5.

    Solution: x*(x + y)*(x + 2*y)*(x + 3*y)*(x + 4*y)*(x + 5*y)
Résultat:   Voir dans spyder
```

5.2. Algèbre linéaire avec sympy

Les expressions pour la somme et le produit dans l'exemple précédent sont automatiquement simplifiées. Des expressions plus compliquées peuvent être simplifiées avec une ou plusieurs des fonctions suivantes.

Fonction Description

sp.cancel(): Annule les facteurs communs au numérateur et au dénominateur. sp.expand(): Développe une expression factorisée.
sp.factor(): Factoriser une expression développée.
sp.radsimp(): Rationalise le dénominateur d'une expression.
sp.simplify(): Simplifie une expression.
sp.trigsimp(): Simplifie uniquement les parties trigonométriques de l'expression.

Exemple:

```
import sympy as sp
from math import *
x = sp.symbols('x')
expr = (x**2 + 2*x + 1) / ((x+1)*((sp.sin(x)/sp.cos(x))**2 + 1))
print(expr)
sp.simplify(expr)
```

Solution: (x+1) cos(x)*cos(x)

Le générique sp.simplify() essaie de simplifier une expression de toutes les manières possibles. Ceci est souvent coûteux en calcul; l'utilisation de simplificateurs plus spécifiques lorsque cela est possible réduit le coût.

Exemple:

```
import sympy as sp
from math import *
```

```
x, y, i = sp.symbols('x y i')
expr = sp.product(x + i*y, (i, 0, 3))
print(expr)
                  # solution: x*(x + y)*(x + 2*y)*(x + 3*y)
expr_long = sp.expand(expr) # Expand the product terms.
print(expr_long)
                  # Solution: x**4 + 6*x**3*y + 11*x**2*y**2 +
6*x*y**3
expr_long /= (x + 3*y)
print(expr_long)
                   # Solution: (x**4 + 6*x**3*y + 11*x**2*y**2 +
6*x*y**3)/(x + 3*y)
expr_short = sp.cancel(expr_long) # Cancel out the denominator.
             # Solution: x**3 + 3*x**2*y + 2*x*y**2
sp.factor(expr_short) # Factor the result.
              # Solution: x*(x + y)*(x + 2*y)
               # Simplify the trigonometric parts of an expression.
sp.trigsimp(2*sy.sin(x)*sy.cos(x))
           # Solution: sin(2*x)
```

1. Les simplifications renvoient de nouvelles expressions ; ils ne modifient pas les expressions existantes en place.
2. L'opérateur == compare deux expressions pour une égalité structurelle exacte, non algébrique équivalence. Simplifiez ou développez les expressions avant de les comparer avec ==.
3. Les expressions contenant de l'avoine peuvent ne pas se simplifier comme prévu.

Utilisez toujours des nombres entiers et SymPy rationnels dans les expressions, le cas échéant.

Exemple:

```
import sympy as sp
from math import *
x, y, i = sp.symbols('x y ')
expr = 2*sp.sin(x)*sy.cos(x)
sy.trigsimp(expr)
           # Solution: sin(2*x)
print(expr)
2*sin(x)*cos(x) # The original expression is unchanged.
2*sy.sin(x)*sp.cos(x) == sp.sin(2*x)
   #Solution: False # The two expression structures differ.
 sp.factor(x**2.0 - 1)
```

    #Solution: x**2.0 - 1 # Factorization fails due to the 2.0.

Utilisation du calcul symbolique pour solutionner un système d'équations linéaires

Le package sympy est utilisé
Exemple:

```
import sympy as sp
# active le module de calcul symbolique, et la ligne suivante:
x,y=sp.symbols('x,y')  # définit x et y comme variables algébriques
symboliques.
# on définit ensuite les deux équations à résoudre
eq1= 2*x+  y+1
eq2=-4*x-5*y+4
# et on solutionne
solution=sp.solve((eq1,eq2),x,y)
solution
```

ou bien faire comme suit :

```
import sympy as sp
x,y=sp.symbols('x,y')
solution=sp.solve((2*x+  y+1,-4*x-5*y+4),x,y)
print(solution)
print(type(solution))
```

Résultat : On peut visionner la solution : { **x=-3/2, y=2** }

Sympy peut également résoudre des systèmes d'équations en algèbre linéaire. Un système d'équations linéaires Ax = b est résolu d'une manière différente que dans NumPy et SciPy :
au lieu de définir la matrice A et le vecteur b séparément, définir la matrice augmentée M = [A|b] et appelez **y.solve_linear_system()** sur M .
Les matrices SymPy sont définies avec sp.Matrix(), avec la même syntaxe que les tableaux NumPy à 2 dimensions. Par exemple, le code suivant résout le système donné ci-dessous.
x + y + z = 5
2x + 4y + 3z = 2
5x + 10y + 2z = 4

```
import sympy as sp
from IPython.display import *
x, y, z = sp.symbols('x y z')
 # Define the augmented matrix M = [A|b].
 M = sp.Matrix([ [1, 1, 1, 5], [2, 4, 3, 2], [5, 10, 2, 4] ])
 # Solve the system, providing symbolic variables to solve for.
sp.solve_linear_system(M, x, y, z)
 # Solution : {x: 98/11, y: -45/11, z: 2/11}
```

Dans le calculus,

SymPy est également équipé pour effectuer des opérations de calcul standard, y compris les dérivées, les intégrales et la prise de limites. Comme d'autres éléments de SymPy, les opérations de calcul peuvent être temporellement coûteuses, mais ils donnent des solutions exactes chaque fois que des solutions existent.
Différenciation, la commande sp.Derivative() crée une forme fermée, dérivée non évaluée d'une expression. C'est comme mettre d/dx devant une expression sans réellement calculer symboliquement la dérivée. L'expression résultante a une méthode doit() qui peut être utilisée pour évaluer la dérivée réelle. De manière équivalente, sy.diff() prend immédiatement la dérivée d'une expression.
sp.Derivative() et sp.diff() acceptent une seule expression, puis la variable ou les variables par rapport auxquelles la dérivée est prise.

Exemple:

```
import sympy as sp
from IPython.display import *

x, y = sp.symbols('x y')
f = sp.sin(y)*sp.cos(x)**2
    # Make an expression for the derivative of f with respect to x.
df = sp.Derivative(f, x)
print(df)
   #Derivative(sin(y)*cos(x)**2, x)
 df.doit() # Perform the actual differentiation.
   # Solution : -2*sin(x)*sin(y)*cos(x)
# Alternatively, calculate the derivative of f in a single step.
 sp.diff(f, x)
  #Solution :  -2*sin(x)*sin(y)*cos(x)
# Calculate the derivative with respect to x, then y, then x again.
sp.diff(f, x, y, x)

# Solution: 2*(sin(x)**2 - cos(x)**2)*cos(y)
# Cette expression peut être simplifiée.
```

5.3. Exemples variés de calcul symboliques avec le module sympy

A. Ici on présente des calculs simples.

1. Résolution d’une équation algébrique

Problème

Résoudre: $x^2-5x+6=0$

Code sympy

```
import sympy as sp
x = sp.symbols('x')
eq = x**2 - 5*x + 6
solutions = sp.solve(eq, x)
print(solutions)
```

Résultat: Solution: x=2, x=3

2. Développement d'une expression

Problème

Développer $(x+2)^3$

Code sympy

```
import sympy as sp
x = sp.symbols('x')
expr = (x+2)**3
develop = sp.expand(expr)
print(develop)
```

Résultat x**3+6x**2+12x+8

3. Calcul d'une dérivée symbolique

Problème

Calculer la dérivée de $f(x)=x^3+2x^2+5x$

Code sympy

```
import sympy as sp
x = sp.symbols('x')
f = x**3 + 2*x**2 + 5*x
df = sp.diff(f, x)
print(df)
```

Resultat: $f'(x)=3x^2+4x+5$

4. Calcul d'une intégrale symbolique

Problème

Calculer l'intégrale $\int (x^2+3x+1)\,dx$

Code Sympy

```
import sympy as sp
x = sp.symbols('x')
f = x**2 + 3*x + 1
integrale = sp.integrate(f, x)
print("L'integrale est égale a :", integrale)
```

Résultat: I= $\frac{x^3}{3}+\frac{3x^2}{2}+x$

5. Valeurs propres de la matrice A= $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

Code sympy

```
import sympy as sp
A = sp.Matrix([[1,2],[3,4]])
valeurs = A.eigenvals()
print(valeurs)
```

Résultat: $\lambda_1 = \frac{5+\sqrt{33}}{2}$, $\lambda_2 = \frac{5-\sqrt{33}}{2}$

6. Factorisation symbolique

Problème

Factoriser le polynôme $x^2 - 5x + 6$

Code sympy

```
import sympy as sp
x = sp.symbols('x')
expr = x**2 - 5*x + 6
factorisation = sp.factor(expr)
print(factorisation)
```

Résultat: $(x-2)(x-3)$

Bref, le module **SymPy** permet :

- calcul différentiel
- calcul intégral
- résolution d’équations
- séries de Taylor
- équations différentielles
- calcul matriciel
- algèbre symbolique complète.

B. Ensuite ici on présente des exemples avancés de calcul symbolique avec SymPy dans trois domaines : algèbre de Lie, géométrie différentielle et mécanique hamiltonienne. Les exemples incluent code + interprétation mathématique.

B.1.Algèbre de Lie : calcul du crochet de Lie

1. Considérons l’algèbre de Lie so(3) avec générateurs

$$X_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} , \quad X_2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

Le Code sympy

```
import sympy as sp
X1 = sp.Matrix([[0,0,0], [0,0,-1], [0,1,0]])
X2 = sp.Matrix([[0,0,1], [0,0,0], [-1,0,0]])
```

```
lie_bracket = X1*X2 - X2*X1
print(lie_bracket)
```

Le résultat est [X1,X2]=X3 avec $X_3 = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

Cela confirme la structure de l'algèbre so(3).

2. Algèbre de Lie : constantes de structure

On peut vérifier [Xi,Xj]= ϵ_ijkX_k

```
import sympy as sp
X1 = sp.Matrix([[0,0,0],[0,0,-1],[0,1,0]])
X2 = sp.Matrix([[0,0,1],[0,0,0],[-1,0,0]])
X3 = sp.Matrix([[0,-1,0],[1,0,0],[0,0,0]])
print(X1*X2 - X2*X1)
```

Résultat: [X1,X2]=X3

Les constantes de structure sont donc C_123=1

B. 2. Géométrie différentielle : Champ de vecteurs

1. Considérons un champ X=x∂x+y∂y.

On calcule la divergence de X.

Le code sympy

```
import sympy as sp
x,y = sp.symbols('x y')
X = sp.Matrix([x,y])
div = sp.diff(X[0],x) + sp.diff(X[1],y)
print(div)
```

Résultat: div(X)=2

Cela signifie que le champ est source uniforme.

2. Calcul d'un crochet de Lie de champs de vecteurs

Soient X=x∂y et Y=y∂x

Le code sympy

```
import sympy as sp
x,y = sp.symbols('x y')
X = sp.Matrix([0,x])
Y = sp.Matrix([y,0])
bracket = sp.Matrix([
X[0].diff(x)*Y[0] + X[1].diff(x)*Y[1] -
(Y[0].diff(x)*X[0] + Y[1].diff(x)*X[1]),

X[0].diff(y)*Y[0] + X[1].diff(y)*Y[1] -
```

```
(Y[0].diff(y)*X[0] + Y[1].diff(y)*X[1])
])
print(bracket)
```
Résultat : [X,Y]=x∂x-y∂y

B.3. Géométrie différentielle : métrique et courbure simple

Considérons la métrique du plan polaire $ds^2 = dr^2 + r^2 d\theta$

Code sympy

```
import sympy as sp
r,theta = sp.symbols('r theta')
g = sp.Matrix([[1, 0], [0, r**2]])
g_inv = g.inv()
print(g_inv)
```

Résultat: $g^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{r^2} \end{pmatrix}$.

B.4. Mécanique Hamiltonienne : équations de Hamilton

Les équations sont

$$\dot{q} = \frac{\partial H}{\partial p}, \qquad \dot{p} = \frac{-\partial H}{\partial q}$$

Prenons l'oscillateur harmonique $H = \frac{p^2}{2m} + \frac{1}{2} m\omega^2 q^2$

Résultat: $\dot{q} = \frac{p}{m}$, $\dot{p} = -m\omega^2 q$

Maintenant, on veut tracer les trajectoires, solution de ce système d'equations de Hamilton:

Leur solution donne des trajectoires elliptiques dans l'espace de phase (q,p). Ce script utilise **SymPy** pour la solution analytique et **Matplotlib** pour tracer les courbes.

Le code sympy et **Matplotlib** :

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

# variables
t = sp.symbols('t')
```

```
m = 1
w = 1

# fonctions
q = sp.Function('q')(t)
p = sp.Function('p')(t)

# Hamiltonien
H = p**2/(2*m) + m*w**2*q**2/2

# Equations de Hamilton
dqdt = sp.diff(H,p)
dpdt = -sp.diff(H,q)

# système d'équations différentielles
eq1 = sp.Eq(sp.diff(q,t), dqdt)
eq2 = sp.Eq(sp.diff(p,t), dpdt)

# solution symbolique
solution = sp.dsolve((eq1,eq2))
print(solution)

# constantes arbitraires
C1 = 1
C2 = 0

# solutions explicites
q_sol = C1*sp.cos(w*t) + C2*sp.sin(w*t)
p_sol = -m*w*C1*sp.sin(w*t) + m*w*C2*sp.cos(w*t)

# fonctions numériques
q_func = sp.lambdify(t,q_sol,'numpy')
p_func = sp.lambdify(t,p_sol,'numpy')

# temps
t_vals = np.linspace(0,20,400)

q_vals = q_func(t_vals)
p_vals = p_func(t_vals)

# tracé de la trajectoire dans l'espace de phase
plt.plot(q_vals,p_vals)
plt.xlabel("q")
```

```
plt.ylabel("p")
plt.title("Trajectoire Hamiltonienne dans l'espace de phase")
plt.grid()
plt.show()
```

Interprétation du graphique

Le graphique obtenu montre :

- une ellipse dans l'espace de phase
- énergie constante

$$H=\frac{p^2}{2m}+\frac{1}{2}m\omega^2q^2=E$$

ce qui est l'équation d'une ellipse.

On peut aussi tracer :

a) Le portrait de phase pour plusieurs conditions initiales

On a le code suivant:

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

# variables
t = sp.symbols('t')
m = 1
w = 1

# fonctions
q = sp.Function('q')(t)
p = sp.Function('p')(t)

# Hamiltonien
H = p**2/(2*m) + m*w**2*q**2/2

# Equations de Hamilton
dqdt = sp.diff(H,p)
dpdt = -sp.diff(H,q)

# système d'équations différentielles
eq1 = sp.Eq(sp.diff(q,t), dqdt)
eq2 = sp.Eq(sp.diff(p,t), dpdt)

# solution symbolique
```

```
solution = sp.dsolve((eq1,eq2))
print(solution)

# constantes arbitraires
C1 = 1
C2 = 0

# solutions explicites
q_sol = C1*sp.cos(w*t) + C2*sp.sin(w*t)
p_sol = -m*w*C1*sp.sin(w*t) + m*w*C2*sp.cos(w*t)

# fonctions numériques
q_func = sp.lambdify(t,q_sol,'numpy')
p_func = sp.lambdify(t,p_sol,'numpy')

# temps
t_vals = np.linspace(0,20,400)

q_vals = q_func(t_vals)
p_vals = p_func(t_vals)

# tracé de la trajectoire dans l'espace de phase
for C1 in [0.5,1,1.5,2]:
  q_sol = C1*sp.cos(w*t)
  p_sol = -m*w*C1*sp.sin(w*t)
  q_func = sp.lambdify(t,q_sol,'numpy')
  p_func = sp.lambdify(t,p_sol,'numpy')
  plt.plot(q_func(t_vals),p_func(t_vals))

plt.plot(q_vals,p_vals)
plt.xlabel("q")
plt.ylabel("p")
plt.title("Trajectoire Hamiltonienne dans l'espace de phase")
plt.grid()
plt.show()
```

Cela montre **toutes les orbites hamiltoniennes**.

b) Le mouvement temporel q(t)

Au code de départ, on ajoute le code suivant, cad:

```
import sympy as sp
import numpy as np
```

```
import matplotlib.pyplot as plt
# variables
t = sp.symbols('t')
m = 1
w = 1
# fonctions
q = sp.Function('q')(t)
p = sp.Function('p')(t)

# Hamiltonien
H = p**2/(2*m) + m*w**2*q**2/2

# Equations de Hamilton
dqdt = sp.diff(H,p)
dpdt = -sp.diff(H,q)

# système d'équations différentielles
eq1 = sp.Eq(sp.diff(q,t), dqdt)
eq2 = sp.Eq(sp.diff(p,t), dpdt)

# solution symbolique
solution = sp.dsolve((eq1,eq2))
print(solution)

# constantes arbitraires
C1 = 1
C2 = 0

# solutions explicites
q_sol = C1*sp.cos(w*t) + C2*sp.sin(w*t)
p_sol = -m*w*C1*sp.sin(w*t) + m*w*C2*sp.cos(w*t)

# fonctions numériques
q_func = sp.lambdify(t,q_sol,'numpy')
p_func = sp.lambdify(t,p_sol,'numpy')

# temps
t_vals = np.linspace(0,20,400)

q_vals = q_func(t_vals)
p_vals = p_func(t_vals)
plt.plot(t_vals,q_vals)
plt.xlabel("t")
```

```
plt.ylabel("q(t)")
plt.show()
```

B.5. Crochet de Poisson avec SymPy
Le crochet de Poisson est

$$\{f,g\}=\frac{\partial f}{\partial q}\frac{\partial g}{\partial p}-\frac{\partial f}{\partial p}\frac{\partial g}{\partial q}$$

Code sympy

```
import sympy as sp
q,p = sp.symbols('q p')
f = q**2
g = p**2
poisson = sp.diff(f,q)*sp.diff(g,p) - sp.diff(f,p)*sp.diff(g,q)
print(poisson)
```

Résultat: $\{q^2,p^2\}=4qp$

B.6. Dynamique Hamiltonienne complète

Code sumpy

```
import sympy as sp
t = sp.symbols('t')
q = sp.Function('q')(t)
p = sp.Function('p')(t)
m,w = sp.symbols('m w')
H = p**2/(2*m) + m*w**2*q**2/2
dqdt = sp.diff(H,p)
dpdt = -sp.diff(H,q)
print(dqdt)
print(dpdt)
```

B.7. Le calcul explicite du champ hamiltonien associé à une fonction sur la sphère S2 (orbite coadjointe de SU(2)) avec SymPy.

On travaille sur la sphère $x^2+y^2+z^2-r^2$ munie de la forme symplectique de Kirillov–Kostant–Souriau (KKS).
Les crochets de Poisson sur l'orbite sont

$$\{x,y\}=z,\{y,z\}=x,\{z,x\}=y.$$

Cela correspond à la structure de l'algèbre de Lie su(2).

Pour le Champ hamiltonien, on procède comme suit:

Pour une fonction f(x,y,z), le champ hamiltonien est

$X_f(g)=\{g,f\}$

En coordonnées :

$X_f=(\{x,f\},\{y,f\},\{z,f\})$

Exemple : f=z

Prenons f(x,y,z)=z

Alors $\dot{x}=\{x,z\}=y$, $\dot{y}=\{y,z\}=-x$, $\dot{z}=0$.

C’est une **rotation autour de l’axe z.**

Calcul avec Sympy, Code sympy

```
import sympy as sp
# variables
x,y,z = sp.symbols('x y z')

# fonction hamiltonienne
f = z

# crochets de Poisson sur S^2
def poisson(a,b):

return (sp.diff(a,x)*sp.diff(b,y)*z
- sp.diff(a,y)*sp.diff(b,x)*z
+ sp.diff(a,y)*sp.diff(b,z)*x
- sp.diff(a,z)*sp.diff(b,y)*x
+ sp.diff(a,z)*sp.diff(b,x)*y
- sp.diff(a,x)*sp.diff(b,z)*y )

# champ hamiltonien
Xx = poisson(x,f)
Xy = poisson(y,f)
Xz = poisson(z,f)

print("Champ hamiltonien X_f :")
print("dx/dt =",Xx)
print("dy/dt =",Xy)
print("dz/dt =",Xz)
```

Résultat: Le programme donne $X_f=(y,-x,0)$

Donc, $X_f=y\frac{\partial}{\partial x}-x\frac{\partial}{\partial y}$

Interprétation géométrique
Ce champ génère :

- une rotation sur la sphère
- les cercles de latitude

Les trajectoires sont donc $x^2+y^2=cont$ avec z constant.

B.8.**Construire symboliquement la structure de Kähler sur** S^2 **(orbite coadjointe de SU(2)) et vérifier la compatibilité** $g(X,Y)=\omega(X,JY)$.

On rappelle la théorie d'abord.
On est dans le cadre d'une variété symplectique (M,ω) munie d'une structure presque complexe J.
1. Conditions de compatibilité
On dit que J est **compatible** avec ω si :

1. $\omega(JX,JY)=\omega(X,Y)$ pour tous X,Y
2. $\omega(X,JX)>0$ pour tout $X\neq 0$

La deuxième condition signifie que la forme bilinéaire définie par

$$g(X,Y)=\omega(X,JY)$$

est définie positive.

On peut vérifier que g est **une métrique riemannienne**. En effet, on vérifie que g est symétrique,i.e, $g(Y,X)=\omega(Y,JX)$. On utilise le fait que Comme ω est antisymétrique $\omega(Y,JX)=-\omega(JX,Y)$ et ensuite on utilise la compatibilité $\omega(JX,JY)=\omega(X,Y)$, ce qui implique que $\omega(JX,Y)=-\omega(X,JY)$.
Donc $g(Y,X)=\omega(X,JY)=g(X,Y)$.

Dans une base locale, on peut représenter:

- g par une matrice G
- ω par une matrice antisymétrique Ω
- J par une matrice J

La définition $g(X,Y)=\omega(X,JY)$ donne, en écriture matricielle:

$$X^T G Y=X^T \Omega J Y,\forall X,Y$$

Donc les matrices doivent vérifier: $G=\Omega J$,ce qui est exactement l'écriture compacte : $g=\omega J$.

2. **Interprétation géométrique**

Cette relation signifie que :

- ω mesure une aire symplectique
- J fait une rotation de 90∘ dans chaque plan complexe
- leur composition produit la métrique riemannienne.

C'est la structure fondamentale d'une **variété presque kählérienne**.

3. **Structures géométriques sur** S^2

Sur la sphère de rayon r avec coordonnées (θ,ϕ) :

- métrique riemannienne: $g=r^2(d\theta^2+\sin^2\theta d\phi^2)$

- forme symplectique (KKS): $\omega=r\sin\theta d\theta\wedge d\phi$

- structure complexe: $J(\partial_\theta)=\frac{1}{\sin\theta}\partial_\phi$, $J(\partial_\phi)=-\sin\theta\partial_\theta$

Ces trois objets donnent une **structure de Kähler**.

4. **Implémentation avec SymPy**

```
import sympy as sp

# coordonnées
theta, phi, r = sp.symbols('theta phi r')

# matrice de la métrique
g = sp.Matrix([
[r**2, 0],
[0, r**2*sp.sin(theta)**2]
])

print("Metric g =",g)

# forme symplectique
omega = sp.Matrix([
[0, r*sp.sin(theta)],
[-r*sp.sin(theta), 0]
```

```
])

print("Symplectic form ω =",omega)

# structure complexe J
J = sp.Matrix([[0, -sp.sin(theta)],
[1/sp.sin(theta), 0]
])

print("Complex structure J =",J)
```

5. Vérification de la compatibilité

Condition de Kähler: $g=\omega J$
Calcul:au code precedent, on ajoute:

```
test = omega*J
sp.simplify(test)
print(‘’test=’’,test)
```

Résultat: $\omega J=\begin{pmatrix} r^2 & 0 \\ 0 & r^2\sin^2\theta \end{pmatrix}$ ce qui donne exactement g.

6.Interprétation géométrique
La sphère vérifie donc :

- $d\omega=0$
- $J^2=-I$
- $g(X,Y)=\omega(X,JY)$

Donc (S^2,ω,J,g) est une variété de Kahler.

Chapitre 6: Pratiques combinées.

Dans ce chapitre, on présente des exercices combinant tous les aspects de ces notes de cours. On s’exerce à tous les cotés en programmation python.

<u>Question 1.</u>
Je remplis une tirelire de la manière suivante:

-Je commence par déposer 1000 F dans la tirelire (un Lundi)

- puis j’y dépose 2010 F le jour suivant (un mardi)
- puis 3020 F le jour suivant (un mercredi)
- puis 4030F le jour suivant (un jeudi)
- Etc. (chaque jour j’y dépose 1010F de plus que la veille).

Par exemple, le 8eme jour (un Lundi), j’y dépose 8070F et le contenu totale de la tirelire et
1000+2010+3020+4030+5040+6050+7060+8070=36280F

Je veux que lorsque le contenu total de la tirelire dépasse 150000F, je casse la tirelire.

Le but est de déterminer le contenu de la tirelire lorsque je la casserai, ainsi que le jour de la semaine correspondant (lundi,mardi,mercredi,jeudi, vendredi,samedi ou dimanche).

1) Écrire la fonction récursive **depot(n)** qui retourne la somme que j’ai déposée dans la tirelire le n-eme jour. Par exemple, depot(1)=1000 et depot(4)=4030

2) Ecrire une fonction **jour(n)** qui retourne le jour de la semaine correspondant au n-ieme depot. Par exemple jour(1)=’Lundi’, jour(2)=’Mardi’ et jour(15)=’vendredi’. Cette fonction va utiliser la liste suivante
L=[‘Lundi’,’mardi’,’mercredi’,’jeudi’,’vendredi’,’samedi’,’dimanche’]

3) Ecrire une fonction **contenu(n)** qui, a un entier n ≥ 1, renvoie la somme totale en Francs, contenue dans la tirelire le n-ieme jour.
Par exemple, contenu(3)=1000+2010+3020=6030

4) Implémentes l’algorithme et exécutes le programme qui affiche le jour et le contenu de la tirelire.

Question 2

Implémentes et exécutes un programme python qui résous et affiche le résultat du système d’équations différentielles suivant avec le module **odeint.**

$$x' = y$$
$$y' = \frac{y}{2} - x - y^3$$

Question 3: Étude d’une fonction

A. On veut étudier une fonction f (x) sur un intervalle [a,b], par ex:

$$f(x) = x\cos 2x \cos 3x \quad \text{sur } [0,7]$$

pour déterminer son maximum.

1. Algorithme
- on écrit une fonction f(x) pour calculer la valeur de $x\cos 2x\cos 3x$ en fonction de x
- avec cette fonction on calcule la valeur en 50 points équirépartis sur l’intervalle [0,7]
- on stocke le résultat dans 2 tableaux numpy: X pour les valeurs de x, et`Y pour les valeurs de la fonction
- on trace la courbe
- on calcule une valeur approchée de la dérivée en chacun des points du tableau $x_i = X[i]$ en utilisant une approximation par différences centrées, i.e.

$$f'(x) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}}$$

Attention pour le premier et le dernier point, la formule doit être adaptée !

$$f'(x_0) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad \text{et} \quad f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

-on écrit le résultat dans un tableau numpy noté DY

-on trace la courbe de la dérivée
-on détermine la position du maximum de la courbe sur l’intervalle d’étude de façon précise en utilisant la dérivée.

- En notant que le maximum de cette courbe est le dernier maximum local à la position , i.e. qui vérifie:

$f'(x_{max})=0 \, avec \, f'(x_{max}-\varepsilon)>0$ et $f'(x_{max}+\varepsilon)<0$

Pour cela on détermine les 2 derniers points consécutifs (k,k+1) de DY tels que : DY[k]>0 et DY[k+1]<0

La valeur de x_{max} est alors l'intersection du segment (X[k], DY[k]) à (X[k+1], DY[k+1]) avec l'axe des x

On mettra la valeur de x_{max} dans la variable xmax et la valeur de $f(x_{max})$ dans fmax

2. Programmation

```
# code Python calcule et trace fonction
### BEGIN SOLUTION
f = lambda x:x*np.cos(2*x)*np.cos(3*x)
X = np.linspace(0,7,50)
Y = f(X)
plt.plot(X,Y,'x-',lw=2)
### END SOLUTION
# code Python calcul et trace dérivée
### BEGIN SOLUTION
DY = np.zeros(X.size)
for i in range(1,X.size-1):
    DY[i] = (Y[i+1]-Y[i-1])/(X[i+1]-X[i-1])
DY[0] = (Y[1]-Y[0])/(X[1]-X[0])
DY[-1]= (Y[-1]-Y[-2])/(X[-1]-X[-2])
plt.plot(X,DY,'-xg',lw=2)
### END SOLUTION
# calcul du maximum
### BEGIN SOLUTION
k = X.size-1
# selectionne intervalle ou f' est < 0
while DY[k] > 0: k=k-1
# recherche changement de signe
while DY[k] < 0: k=k-1
xmax = X[k] - DY[k]*(X[k+1]-X[k])/(DY[k+1]-DY[k])
fmax = f(xmax)
print(" maximun en x={} f(x)={} (max={} en
{})".format(xmax,fmax,np.max(Y),X[np.argmax(Y)]))
### END SOLUTION
```

B. Calcul des racines d'une fonction

Soit la courbe cubique avec $P(x)=4x^3-2x$
Pour une valeur de R donnée, on cherche à déterminer la valeur de x telle que P(x)=R, c.a.d la racine de l'équation , $P(x)=R$ i.e, Trouver $x \in \mathbb{R}\, tel\, que\, P(x)=R$

Attention: ce problème peut admettre une ou plusieurs solutions.

1. Analyse du problème

Géométriquement, le problème revient à déterminer l'intersection de la courbe y=P(x) avec la droite horizontale y=R .

On trace donc la courbe y=P(x) et la droite y=R pour R=1.0

On calcule ensuite la valeur de la racine avec la fonction **fsolve** de la bibliothèque **numpy.optimize.**

Attention : on doit choisir un point de départ proche de la racine que l'on souhaite calculer !

```
# tracer
### BEGIN SOLUTION
R = 1.0
P = lambda x: 4*x*(x**2-2)
X = np.linspace(-2,2,101)
plt.figure(figsize=(10,8))
plt.plot(X,P(X),label='y=P(x)')
plt.plot([X[0],X[-1]],[R,R],'--',label='y=R')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title("etude des racines de P(x)=R");
### END SOLUTION
```

2. calcul de la solution suivant R

on utilise **fsolve** pour résoudre l'équation et on vérifie en partant de 0

Que se passe-t-il si on part de -1, -0.9, ou -0.8?

```
from scipy.optimize import fsolve
### BEGIN SOLUTION
```

```
F = lambda x : P(x)-R
x0 = fsolve(F,0.)[0]
#x0 = fsolve(F,-0.8)[0]
print("solution :",x0)
print("erreur   :",P(x0)-R)
### END SOLUTION
solution : -0.1260001925862561
erreur   : -1.1102230246251565e-16
# verification graphique
### BEGIN SOLUTION
plt.figure(figsize=(10,8))
plt.plot(X,P(X),label='y=P(x)')
plt.plot([X[0],X[-1]],[R,R],'--',label='y=R')
plt.plot([x0],[P(x0)],'o',markersize=12)
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title("etude des racines de P(x)=R");
### END SOLUTION
```

Question 4

Une des activités de Première consiste à « Simuler à l’aide d’un langage de programmation, la propagation d’une onde périodique ». Il faut donc faire une animation à l’écran. Pour cela, il existe des fonctions de matplotlib destinées aux animations mais elles ne sont pas toujours simples à utiliser. Heureusement, Matplotlib permet une autre technique plus rudimentaire. Elle fonctionne bien dans des situations simples. Elle consiste à dessiner la figure, faire une pause puis effacer l’image. Il faut ensuite dessiner la figure suivante, faire une pause et l’effacer à nouveau. Et ainsi de suite… Pour cela, on utilise les fonctions :

```
plt.cla() # cette fonction efface l’image
…
…
…
plt.pause(0.01) # on fait une pause de 0.01 seconde
```

Dessiner une flèche qui tourne en recopiant le code suivant :

```
import matplotlib.pyplot as plt
import math
for i in range(100):
```

```
      plt.cla() #efface l'image
      plt.axis([-5, 5, -5, 5])
      plt.quiver(0, 0, 4*math.cos(i/10), 4*math.sin(-i/10), angles='xy', scale=1, scale_units='xy')
     plt.pause(0.01) #la valeur de la pause
plt.show()
```

Voici maintenant l'animation attendue pour le niveau Première : « Représenter un signal
périodique et illustrer l'influence de ses caractéristiques (période, amplitude) sur sa
représentation. Simuler à l'aide d'un langage de programmation, la propagation d'une onde
périodique. »
Voir notes
Question 5.

On modélise le mouvement de vibration d'un atome dans une molécule en simplifiant à l'extrême :
On va supposer que l'atome étudié A se déplace selon la dimension x Є R, et que les
autres atomes sont fixes. Ainsi l'atome A subit des forces de la part des autres atomes,
décrites par la fonction énergie potentielle V (x)
Son énergie (Hamiltonien) est alors : $H(x,p)=\frac{p^2}{2m}+V(x)$ (1.1)
où $p\in\mathbb{R}$ est l'impulsion et m>0 est la masse de l'atome.
En pratique, la fonction énergie potentielle sera un polynôme de degré d fixé :
$V(x)=V_0+V_1x+V_2x^2+......+V_dx^d$ avec $V_j\in\mathbb{R}$ et $j=0\rightarrow d$ (1.2)

Noter que les unités physiques (énergie, longueur, masse,...) peuvent être choisies telles
que tous les paramètres (ђ,m,...) ont des valeurs de l'ordre de 1.
Par exemple pour un puits de potentiel anharmonique :

$$V(x)=0{,}5x^2+0{,}002x^4$$

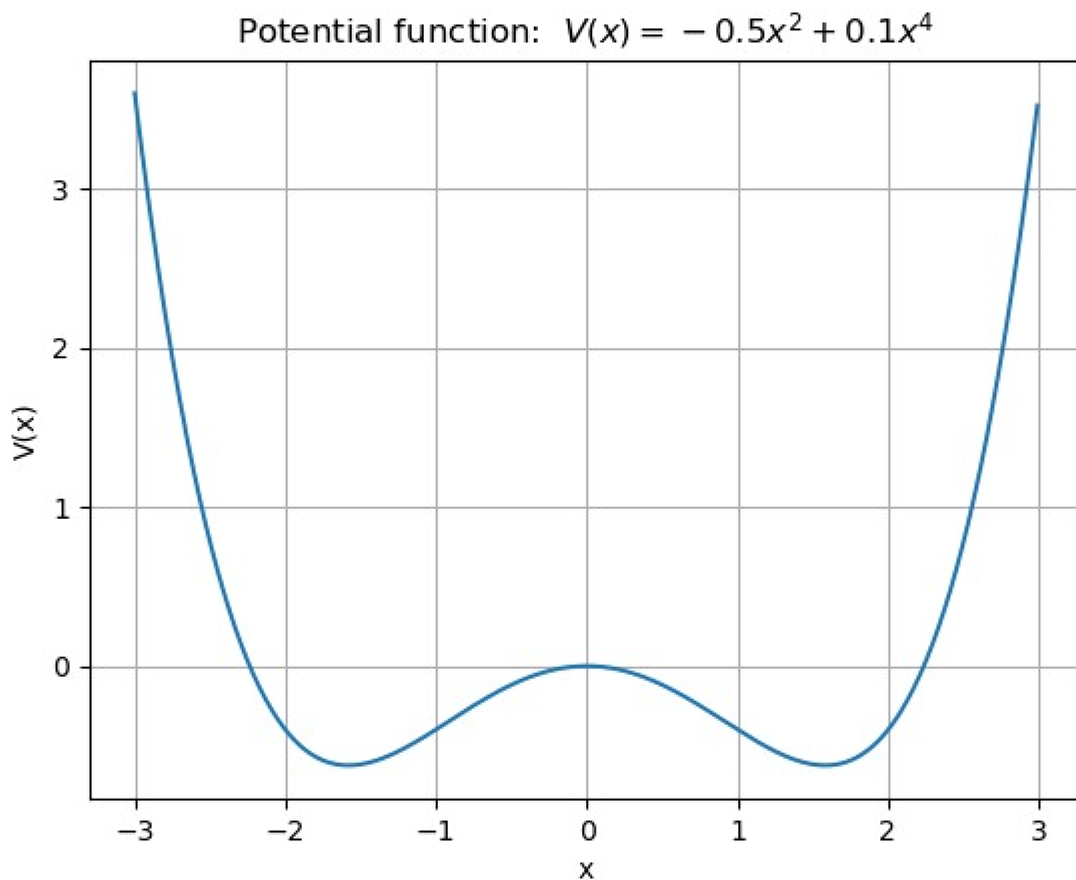
et pour pour un double puits de potentiel :

$$V(x)=-0{,}5x^2+0{,}005x^4$$

Dans cet exercice, on étudie l'évolution classique de l'atome modélisé par une particule ponctuelle.

a. Dessin de la fonction potentiel
Variables :
— Les coefficient $(V_i)_{i=0 \to d}$ sont stockés dans une liste appelée V.
— l'intervalle d'étude [xmin , xmax].
Fonctions:
— Une fonction Potentiel()
— entrée : x ∈ IR
— sortie V(x)
— Une fonction Dessin_V() qui dessine la fonction V(x) pour l'intervalle [xmin , xmax].
Vérifier que le programme fonctionne avec des exemples de V (x).

Résultat attendu:



Programme informatique cfr Potentiel_V0.py

b. Évolution de la particule classique

(1) A partir de eq.(1.1), écrire les équations de mouvement de Hamilton
donnant dx/dt et dp/dt ?
(2) Supposons donné (x_0, p_0) à l'instant t = 0, et un pas de temps dt fixé. L'algorithme
d'Euler est l'expression qui exprime les valeurs (x_1 , p_1) à l'instant dt à partir de
(x_0 , p_0) et dx/dt et dp/dt au premier ordre. Et ainsi de suite.
(3) Donner l'expression de (x_i, p_i) à l'instant $x_i = i dt$ à partir de (x_{i-1}, p_{i-1}) et dx/dt
et dp/dt ?

Solution:
Les équations de mouvement de Hamilton sont obtenues a partir de l'équation de Hamilton et on obtient

$$\frac{dx}{dt} = \frac{\partial H}{\partial p} = \frac{p}{m} \quad \text{et} \quad \frac{dp}{dt} = \frac{-\partial H}{\partial x} = \frac{-dV}{dx}$$

On a $x_i = x(t)\, et\, x_{i+1} = x(t+dt) = x(t) + dt(\frac{dx}{dt}) + O(t^2) \approx x(t) + dt(\frac{dx}{dt})$

De même, on a

$$p_{i+1} \approx p_i + dt(\frac{dp}{dt})$$

c. Exercice de programmation

– En utilisant l'algorithme de Euler précédent, écrire une fonction Evolution_classique()
– entrée :
x,p :position et impulsion de la particule à t = 0.
t : temps d'intégration
opt_des: Si **opt_des=1,** cette fonction fera le dessin de la particule classique sur la fenêtre sous forme d'un point.
– sortie :
x,p :position et impulsion de la particule à la date t.

Dans le programme principal, rajouter l'appel de la fonction **Evolution_classique().**
Vérifier que le programme fonctionne.
Programmation informatique cfr **Evolution_classique()** avec la méthode d'Euler ou odeint.

Question 6

On veut implémenter l'algorithme d'Euclide étendu qui, en plus du pgcd, donne des coefficients d'une relation de Bézout, c'est-à-dire des entiers u et v tels que au+bv=pgcd(a,b).
On rappelle l' algorithme d'Euclide .
On souhaite calculer le pgcd de $a,b \in N^*$. On peut supposer $a \geq b$.
On calcule des divisions euclidiennes successives. Le pgcd sera le dernier reste non nul.
On a le lemme suivant:
Soient $a,b \in N^*$. Écrivons la division euclidienne a = bq + r .
Alors on a
pgcd(a,b)=pgcd(b,r) . On voit également que r=a-bq.
Donc on a:
- division de a par b, $a = bq_1 + r_1$.
Par le lemme, $pgcd(a, b) = pgcd(b, r_1)$ et si
$r_1 = 0$ alors pgcd(a, b) = b sinon on continue:
- $b = r_1q_2+r_2$, $pgcd(a,b) = pgcd(b,r_1)= pgcd(r_1,r_2)$,

- $r_1 = r_2q_3+r_3$, $pgcd(a,b)=pgcd(r_2,r_3)$,
- ...
- $r_{k-2} = r_{k-1} q_k + r_k$, $pgcd(a,b) = pgcd(r_{k-1} ,r_k)$,
- $r_{k-1} = r_k q_k + 0$. On a $pgcd(a, b) = pgcd(r_k,0) = r_k$

Comme à chaque étape le reste est plus petit que le quotient on sait que $0 \leq r_{i+1} < r_i$.
Ainsi l'algorithme se termine car nous sommes sûr d'obtenir un reste nul, les restes formant une suite décroissante d'entiers positifs ou nuls : $b > r_1 > r_2 > . . . \geq 0$.

Exemple.
Calculons le pgcd de a = 600 et b = 124 .
$600 = 124 \times 4 + 104$
$124 = 104 \times 1 + 20$
$104 = 20 \times 5 + 4$
$20 = 4 \times 5 + 0$
Ainsi le pgcd(600, 124) = 4.

Exercice de programmation

1.Créer une fonction pgcd prenant en argument deux entiers positifs et et renvoyant PGCD(a;b). Utilises l'instruction **while.**
2.Coefficient de Bezout.
On veut créer une fonction bezout prenant en argument deux entiers naturels a et b(avec b
non nul) et renvoyant des entiers u et v tel que au+bv=pgcd(a,b).
a. Si b=0, que peut-on prendre pour (u,v)?
b. Montrer que si q et r sont respectivement le quotient et le reste de la division euclidienne
de a par b et si (u,v) est un couple de coefficients de Bézout de b et r alors (v,u-qv) est un couple de coefficients de Bézout de a et b.
Par exemple, les coefficients de Bézout de 19 et 8 étant (3,-7), comment en déduire ceux de 27 et 8 ?
c. À l'aide des questions précédentes, programmer la fonction récursive bezout.
d. Modifier cette fonction pour qu'elle renvoie en plus le PGCD en premier.

```
>>>>>>bezout(4,18)
>>>>> (2, -4, 1)
```

<u>Question 7</u>

Résolution de l'équation ax+by=c
Créer une fonction **eqlin** prenant en entrée trois entiers relatifs non nuls a, b et c et renvoyant une solution (x,y), si elle existe, de l'équation ax+by =c.
Par exemple.

```
>>> eqlin(4,7,3)
       (6, -3)
>>> eqlin(7,-4,1)
     (-1, -2)
>>> eqlin(6,3,4)
    'pas de solutions'
```

<u>Question 8</u>

Créer une fonction F(x,y) qui définit (ou qui permet de tracer les trajectoires engendrées par) le champ de vecteurs

$$X = -y\frac{\partial}{\partial x} + x\frac{\partial}{\partial y}$$

Question 9

-Définir une fonction **matA()** qui demande à l'utilisateur de saisir une matrice comme la liste dont les éléments sont les listes des éléments de chaque ligne.
- Ecrire une fonction **produitR(x,A)** qui à un réel x et à une matrice A carrée d'ordre 4, retourne la matrice xA.
- Ecrire une fonction **produit(A,B)** qui à une matrice A et une matrice B toutes carrées d'ordre 4 retourne la matrice AxB
- Ecrire une fonction récursive **puissance(A,n)** qui retourne A^n, à une matrice A d'ordre 4, et un entier naturel n.
- Ecrire une fonction **factoriel(n)** qui retourne la factoriel d'un nombre naturel n
- Implémentes et exécutes sous python un programme demandant à l'utilisateur un entier n et qui fait appel aux fonctions définies en a), b), c), d) et e) pour afficher l'exponentiel d'une matrice. (On devra limiter la série définissant l'exponentielle à n fixer).

Références

1. **Julien Guilld.** Programmation par la pratique python. Dunod, 2021

2. Alexandre Casamayou-Boucau Pascal Chauvin Guillaume Connan. Programmation e n python pour les mathématiques. *Cours et exercices.* 2è édition, Dunod, 2012, 2016.

3.Gérard Swinnen. Apprendre à programmer avec Python. *Avec plus de 40 pages d'exercices corrigés*. Eyrolles, 2009.