

MEGA MEMO PYTHON

Variables:

-association entre un identificateur et une valeur stockée dans l'ordinateur

-peut être composée de lettres majuscules, minuscules, chiffres, du symbole "_" (underscore)

-**Attention:** peut pas commencer par un chiffre; variable en minuscule ≠ variables minuscules (age ≠ AGE)

-écriture en pseudo-code: $x \leftarrow 7$ en python: `x=7`

Variables GLOBALES: définies EN DEHORS de toute fonction; existe durant l'exécution du programme

Variable LOCALES: définies DANS une fonction, masque toute autre variable de même nom; existe durant l'exécution de la fonction

→ différents types de données car:

Python a besoin de connaître quels types de données sont utilisées pour savoir quelles opération il faut faire avec:

-Entiers: `int`

-Flottant (nombre à virgule): `float`

-Chaine de caractère = succession de caractère:

`"` ou `'` pour écrire des guillemets sans fermer la chaîne de caractère

`\n` retour à la ligne (retour chariot)

`\t` faire une tabulation `\\` écrire un antislash

```
chaîne1='rien'
chaîne2="rien"
print('chaîne=',chaîne1,'ou =',chaîne2)
```

```
>>> chaîne= rien ou = rien
```

```
chaîne='j\'aime tout'    j'aime tout
print(chaîne)            >>>
```

```
a,b=83,'var'
print('retour à la ligne :',a,"\n",b)
print('tabulation :', a,":\t",b)
print('antislash :', a,"\\",b)

retour à la ligne : 83
var
tabulation : 83 :      var
antislash : 83 \ var
>>>
```

affectation d'une variable par l'utilisateur: `a=input`

```
nombre=int(input("nombre="))    nombre=12
flot=float(input("flottant="))   flottant=1.2
chaîne=str(input("chaîne="))     chaîne='douze'
print(nombre,flot,chaîne)        12 1.2 'douze'
```

incrémenter d'une variable: `variable = variable +1` ou `variable +=1`

opération:

addition: `+` ; multiplication: `*` ; soustraction: `-` ; exposant: `**` ;

division: `/` ; division entière: `//` ; modulo: `%`

```
48//12= 4 14//3= 7
48%12= 0 14%3= 2
```

`print()` avec retour à la ligne :

```
print("a",)
print("b",end = "")
print("b",end = "")    a
                        bb
```

la fonction `type()` renvoie le type de la variable passée en paramètre

```
>>> type(3)
<class 'int'>
>>> type(2.8)
<class 'float'>
>>> type('chaîne')
<class 'str'>
```

```
for k in range(2):
    for i in range(4):
        print("OX",end = "")
    print()
    for j in range(4):
        print("XO",end = "")
    print()

OXOXOXOX
XOXOXOXO
OXOXOXOX
XOXOXOXO
```

Booléens: en python, la classe des booléens contient uniquement les éléments `True` et `False`.

opérations de bases entre les booléens

| Opérateur logique | and | or | not |
|----------------------|--------|--------|--------|
| Symbole mathématique | \cap | \cup | \neg |

Structures Conditionnelles:

→ permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.
formes complètes:

→ if : "si..."
→ else : "sinon..."
→ elif : "sinon si..."

```
a=int(input("a="))
if a>0: #si a positif
    print("a positif")
elif a<0: #sinon si a negatif
    print("a négatif")
else: #sinon
    print("a nul")
```

opérateur de comparaison
différent de: !=
égal à: ==
supérieur ou égal à: >=
inférieur ou égal à: <=

Boucles: permettent de répéter une certaine opération autant de fois que nécessaire

→ **while**

permet de répéter une instruction

tant qu'une condition reste vraie

break → peut interrompre une boucle

continue → continuer une boucle,

en repartant directement à la ligne du

while ou **for** (ex: pour supprimer les éléments d'une liste)

→ **for**

spécialisée dans le parcours d'une séquence de plusieurs données

```
chaine='chromodynamique'
for k in chaine:
    if k in 'quantique':
        print(k)
```

n
a
i
q
u
e

```
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # On retourne au while sans exécuter les autres lignes
    print("La variable i =", i)
    i += 1 # Dans le cas classique on ajoute juste 1 à i
```

```
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

Liste & Chaînes de caractères: .La classe str:

Mettre une chaîne de majuscule: upper()
Mettre une chaîne en minuscule: lower()
Retirer les espaces au début et à la fin de la chaîne: strip()
Mettre le premier caractère en majuscule: capitalize()
Centrer la chaîne dans un espace de n caractères: center(n)
la chaîne est renvoyée sans être modifiée

```
chaine='ChAiNe'  
chaine1=chaine.upper()  
chaine2=chaine.lower()  
print("chaine1=",chaine1,"chaine2=",chaine2,"chaine=",chaine)
```

```
chaine1= CHAINE chaine2= chaine chaine= ChAiNe  
>>>
```

```
>>> chaine=' grappa '  
>>> chaine.strip()  
'grappa'  
>>> chaine.upper().center(20)  
'          GRAPPA          '
```

```
>>> chaine='grappa'  
>>> chaine.capitalize()  
'Grappa'
```

Convertir une chaîne de caractère en liste:
→ list(): séparation faite à chaque caractère
→ split(): séparation se fait sur le(s) caractère(s) choisi(s)
L'accès à un caractère de la chaîne se fait par l'intermédiaire des indices mis à l'intérieur de *crochets*.

```
chaine[0]= g chaine[1:5]= rapp chaine[-1]= a
```

→ max(), min() retournent le caractère maximal (resp. minimal) selon l'ordre alphabétique, de la chaîne (attention les majuscules ne sont pas retenues par cette fonction),
→ len() donne le nombre de caractère de la chaîne (espaces inclus),
→ in donne l'existence d'un caractère dans une chaîne de caractère,
→ count("str", beg, end) donne le nombre de fois où l'occurrence "str" apparaît dans la chaîne de caractère, il faut préciser l'index de départ, *beg*, et de fin, *end*, de la recherche (attention les majuscules ne sont pas retenues par cette fonction)

```
>>> chaine.split("a")  
['gr', 'pp', '']  
>>> chaine.split("pa")  
['grap', '']  
>>> list(chaine)  
['g', 'r', 'a', 'p', 'p', 'a']
```

```
>>> max(chaine)  
'r'
```

```
>>> min(chaine)  
'a'
```

```
>>> chaine1='il a la grappa!'  
>>> 'a' in chaine1  
True  
>>> 'b' in chaine1  
False  
>>> chaine1.count('a',0,len(chaine1))  
4
```

Il est possible de concaténer deux chaînes de caractères en une seule

→ str() retourne une chaîne vide
→ replace('old','new',n) remplace n fois dans une chaîne un caractère ('old') par un autre caractère ('new')

```
>>> chaine='grappa'  
>>> chaine.replace('a','o',2)  
'groppo'
```

```
>>> var1='Hé Andrea Andrea'  
>>> var2='Il IL a la grappa! La grappa!'  
>>> var1[0:9]+var2[2:18]  
'Hé Andrea IL a la grappa!'
```

```
>>> var2[11:19]*3  
'grappa! grappa! grappa! '
```

```
p=str(57) # 57 est devenue une chaine de carac  
print(p+' degrés la grappa')
```

```
57 degrés la grappa
```

l'opérateur %. s'applique sur les chaînes de caractères et permet de définir le format lors d'un print()

Listes: Ensemble ordonné d'éléments entouré par des *crochets*. Chaque élément d'une liste est associé à un nombre, sa *position* ou son *index*. Le premier index est 0, le second 1 ...

```
>>> list1 = ['physics', 'chemistry', 1997, 2000]
>>> list2 = [1, 2, 3, 4, 5]
>>> list1[0]
'physics'
>>> list1[-2]
1997
>>> list2[1:5]
[2, 3, 4, 5]
```

```
>>> list1.index(1997)
2
```

max(list) retourne la valeur maximale des éléments d'une liste,
min(list) retourne la valeur minimale des éléments d'une liste
index("val") permet d'obtenir la position du premier élément égal à la valeur "val" dans la liste

```
>>> list1.count(1997)
1
```

```
>>> for k in list2:
    print(k,end="")
```

Il est possible de modifier un ou plusieurs éléments d'une liste à partir de son *index* ou de sa *position*

```
>>> 1997 in list1
True
```

```
12345
```

```
>>> list2=[1,2,3,4,5]
>>> list2[1]=1
>>> list2
[1, 1, 3, 4, 5]
```

→ append():modifie un ou plusieurs éléments d'une liste
Deux listes peuvent être concaténées à l'aide de
→ extend(liste) qui prend un seul argument qui est toujours une liste et ajoute chacun des éléments de cette liste à la liste originale.

```
>>> list3=[1,2,3]
>>> list3.append(4)
>>> list3
[1, 2, 3, 4]
```

→ reverse() « renverser » les positions des valeurs dans une liste

```
>>> List1=[1,2,3,4]
>>> List1.reverse()
>>> List1
[4, 3, 2, 1]
```

```
>>> List1,List2=[1,2,3,4],[4,5,6,7,8]
>>> List1.append(List2)
>>> List1
[1, 2, 3, 4, [4, 5, 6, 7, 8]]
```

```
>>> List1,List2=[1,2,3,4],[4,5,6,7,8]
>>> List1.extend(List2)
>>> List1
[1, 2, 3, 4, 4, 5, 6, 7, 8]
```

Supprimer un élément d'une liste:

→ del() si on connaît sa position
→ remove() si on connaît sa valeur (enlève sa première occurrence si la valeur est plusieurs fois dans la liste)

```
>>> list1=[1,2,3,4,2,5,6]
>>> list1.remove(2)
>>> list1
[1, 3, 4, 2, 5, 6]
>>> del(list1[1])
>>> list1
[1, 4, 2, 5, 6]
>>> del(list1[1:3])
>>> list1
[1, 5, 6]
```

→pop(index) permet d'enlever l'élément d'*index* connu de la liste et de retourner la valeur qui a été enlevée

→ pop() c'est le dernier élément qui est supprimé

```
>>> list1=[1,2,3,4]
>>> list1.pop(1)
2
>>> list1
[1, 3, 4]
```

→insert(index,val) ajouter un élément à une liste

→sum(List) retourne la somme des éléments de List, si List comporte des éléments de type int, float, ou complex

```
>>> List1=[1,2,3,4]
>>> List1.insert(2,20)
>>> List1
[1, 2, 20, 3, 4]
```

```
>>> L=[1,2,1.5]
>>> sum(L)
4.5
```

→range(entier1,entier2,pas) retourne la liste d'entiers [entier1,.....,entier2 -1]

```
>>> L=[]
>>> for k in range(6):
    L.append(k)

>>> L
[0, 1, 2, 3, 4, 5]
```

```
>>> for k in range(2,9):
    L.append(k)

>>> L
[2, 3, 4, 5, 6, 7, 8]
```

```
>>> for k in range(0,100,10):
    L.append(k)

>>> L
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Range dans l'ordre décroissant:

```
>>> L=[]
>>> for k in range(12,-1,-1):
    L.append(k)

>>> L
[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
>>> L=[]
>>> for k in range(12,6,-1):
    L.append(k)

>>> L
[12, 11, 10, 9, 8, 7]
```

Tuples: liste *non-mutable* (elle ne peut, en aucune manière, être modifiée) est défini de la même manière qu'une liste sauf que l'ensemble d'éléments est entouré de *parenthèses* plutôt que de *crochets*

→ impossible d'enlever ou d'ajouter un élément

→ sont d'accès plus rapide que les listes. Si vous définissez un ensemble de valeurs constantes et que tout ce que vous allez faire est le lire sans le modifier → utiliser tuple

→ code plus sûr si vous « protégez en écriture » les données qui n'ont pas besoin d'être modifiées. Utiliser un tuple à la place d'une liste assure que les données sont constantes et le resteront.

```
>>> t = ("a", "b", "mpilgrim", "z", "example")
>>> t[0]
'a'
>>> t[1:3]
('b', 'mpilgrim')
>>> t.index('example')
4
```

Dictionnaire: Permet de définir une relation sous forme de tables entre des *clés* et des *valeurs*

forme: { "clé1" : "valeur1", "clé2" : "valeur2", ... }

→ obtenir les clés de l'ensemble des éléments du dictionnaire d.keys(),

→ obtenir les valeurs de l'ensemble des éléments du dictionnaire d.values(),

→ obtenir les couples clé : valeur de l'ensemble des éléments du dictionnaires d.items()

```
>>> d={"maths":"Gall","Physique":"Roubin","SI":"Nerko"}
>>> d.items()
dict_items([('maths', 'Gall'), ('Physique', 'Roubin'), ('SI', 'Nerko')])
>>> d.keys()
dict_keys(['maths', 'Physique', 'SI'])
>>> d.values()
dict_values(['Gall', 'Roubin', 'Nerko'])
```

```
>>> d['maths']
'Gall'
>>> d['Physique']
'Roubin'
>>> d['SI']
'Nerko'
```

→ possible d'assigner une nouvelle valeur à une clé existante

→ possible d'ajouter de nouvelles paires

clé:valeur à tout moment

→ un dictionnaire n'est pas ordonné !!

→ les clés ou les valeurs

peuvent aussi être des float, int,...

→ del() permet d'effacer un élément d'un dictionnaire à partir de sa clé

→ clear() efface tout

```
>>> d['Physique']='JPP'
>>> d
{'maths': 'Gall', 'Physique': 'JPP', 'SI': 'Nerko'}
```

```
>>> d['Français']='Bonant'
>>> d
{'maths': 'Gall', 'Français': 'Bonant', 'Physique': 'JPP', 'SI': 'Nerko'}
```

```
>>> del(d['Français'],d['SI'])
>>> d
{'maths': 'Gall', 'Physique': 'JPP'}
```

```
>>> d.clear()
>>> d
{}
```

Fichiers:

→ fichiers binaires: nécessitent de connaître le format binaire d'écriture pour être lus

(ex: PDF, JPEG, un exécutable, mp3, mp4)

→ fichiers textes: contiennent des caractères uniquement, ouvrables avec un éditeur de texte, contenu (texte, ponctuation, nombres,...) souvent divisé en lignes. (ex: fichier csv, page web HTML, etc...)

Un fichier fait parti d'une ressource nécessaire d'acquérir avant de s'en servir puis la libérer après usage (pour un fichier cela se fait en l'ouvrant puis en le fermant)

→ ouvrir un fichier: `fichier = open("chemin d'accès", [mode])` [mode] peut valoir:

- 'r' : ouverture en lecture (Read) ;

- 'w' : ouverture en écriture (Write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé,

- 'a' : ouverture en écriture en mode ajout (Append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

- 'r+' : ouverture en lecture et écriture

→ `fichier.write("texte")` : écrit la chaîne de caractère "texte" dans le fichier, (pour aller à la ligne il faut que la chaîne se termine par "\n")

→ `fichier.writelines(liste)`: écrit tous les éléments d'une liste dans le fichier

→ `fichier.close()`: ferme le fichier

→ `fichier.read(n)`: lit n caractères (lit tout si `read()`)

→ `fichier.readline()`: lit la ligne suivante

→ `fichier.readlines()`: lit tout le fichier dans une liste de ligne

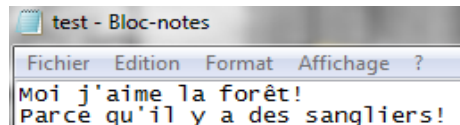
→ `ligne.replace('a','b')`: remplace tous les 'a' de la ligne par 'b'

→ `ligne.replace('a','b',n)`: remplace les n premiers 'a' de la ligne par 'b'

```
fichier=open("test.txt","w")
fichier.write(" veni!\n vidi!\n\n vici!\n ")
fichier.close()
```

```
veni!
vidi!

vici!
```



```
Fichier  Edition  Format  Affichage  ?
Moi j'aime la forêt!
Parce qu'il y a des sangliers!
```

```
>>> fichier=open("test.txt","r")
>>> fichier.readline()
'Moi j'aime la forêt!\n'
```

```
>>> fichier=open('test.txt','r')
>>> fichier.read(10)
'Moi j'aime'
>>> fichier.read(5)
' la f'
>>> fichier.readline()
'orêt!\n'
```

```
>>> fichier=open("test.txt","r")
>>> fichier.readlines()
['Moi j'aime la forêt!\n', "Parce qu'il y a des sangliers!\n"]
```

```
fichier=open('test2.txt','r')
ligne=fichier.readline()
print("ligne non modifiée:",ligne)
ligne=ligne.replace('a','A')
print("ligne modifiée:", ligne)
fichier.close()
```

```
ligne non modifiée: j'aime la grappa
ligne modifiée: j'Aime lA grAppA !
```

→ `ligne.split('caractère')`: permet de séparer en plusieurs morceaux une chaîne de caractère. Les chaînes obtenues sont mises dans une liste

→ `ligne.rstrip('str')`: recommandée pour supprimer le (ou les) caractère(s) de fin de ligne situés au bout de chaque ligne (le fichier origine n'est pas modifié)

Python fonctionne avec un "curseur" qui se déplace dans le fichier, les fonctions read commencent là où se trouve le curseur

→ `fichier.tell()`: donne la position du curseur

position initiale du curseur: `fichier.tell()` retourne 0

→ `fichier.seek()`: change la position du curseur

```
>>> ligne=fichier.readline()
>>> l=ligne.split('e')
>>> l
['Moi j'aim', ' la forêt!\n']
```

```
>>> fichier.tell()
22
>>> fichier.seek(0)
0
>>> fichier.tell()
0
```

```
"Moi j'aime la forêt!\n"
>>> v=ligne.rstrip("\n")
>>> v
'Moi j'aime la forêt!'
```


Problèmes stationnaires (invariant au cours du temps) à 1 dimension (une seule variable)

→ Résolution approchée d'une équation algébrique du type $f(x) = 0$ (toute équation peut être mise sous la forme $f(x) = 0$)

→ Sur un intervalle $[a, b]$, il est **impossible** de trouver exactement c tel que $f(c) = 0$ (précision limitée du codage des nombres en IEEE754; erreurs d'arrondis liées aux calculs (avec cumulation des erreurs !))

→ Nécessaire de définir un critère de convergence: une valeur ε telle que $|f(c)| < \varepsilon$ (en pratique $\varepsilon \approx 10^{-8}$)

→ La plupart des méthodes de recherche de racines d'une fonction, se comportent bien si **une seule racine est présente dans l'intervalle d'étude**. S'il y en a plusieurs, il convient de restreindre la fonction.

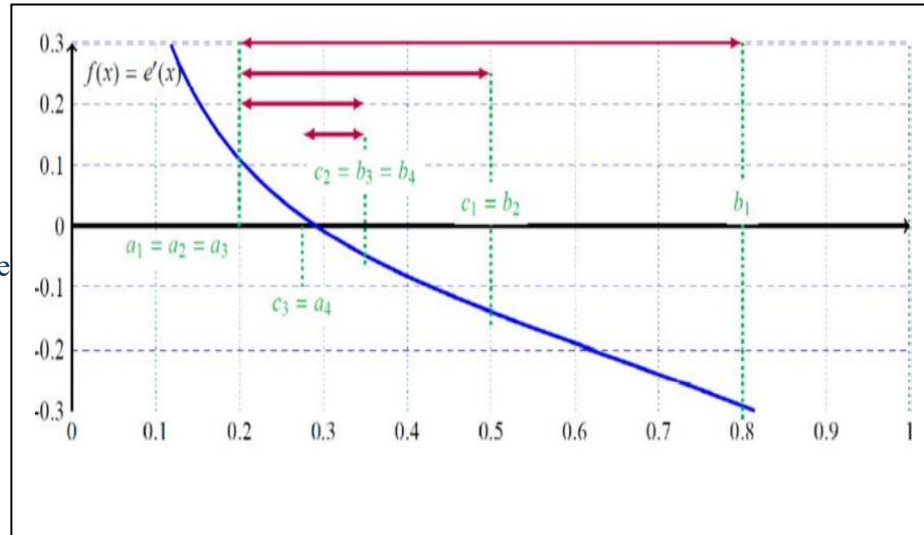
→ Séparer la racine r_i revient à trouver l'intervalle $]a_i; a_{i+1}[$ où cette racine est **unique** !

Méthode de dichotomie:

- 1) Diviser l'intervalle $[a, b]$ en deux parts égales $[a, c]$ et $[c, b]$ avec c milieu de $[a, b]$
- 2) Conserver l'intervalle $[a, c]$ ou $[c, b]$ contenant la racine (phase de test)
- 3) Y reproduire l'opération, jusqu'à ce que le critère de convergence soit satisfait.

```
import pylab as pl
import math as m
import time
```

```
def f(x):
    return m.cos(x)
```



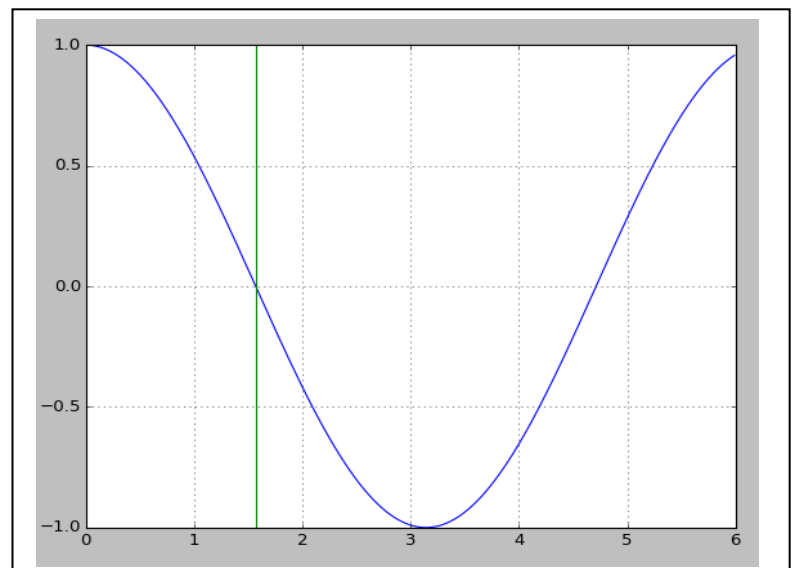
```
def dichotomie(a,b,eps,itemax):
    i=0
    e=eps
    c=(a+b)/2
    while abs(f(c))>=e and i<=itemax and (b-a)>2*e:
        if f(c)*f(b)<=0: #la racine est entre m et b
            a=c #donc intervalle réduit à gauche
        elif f(c)*f(a)<=0: #la racine est entre a et m
            b=c # donc intervalle réduit à droite
        c=(a+b)/2 #milieu du nouvel intervalle
        print('c numéro',i+1,'=',c)
        i+=1
    return c
```

```
c numéro 1 = 1.5
c numéro 2 = 1.75
c numéro 3 = 1.625
c numéro 4 = 1.5625
c numéro 5 = 1.59375
c numéro 6 = 1.578125
c numéro 7 = 1.5703125
c numéro 8 = 1.57421875
c numéro 9 = 1.572265625
c numéro 10 = 1.5712890625
Valeur de x pour f(x)=0 : 1.5712890625
Temps d'exécution : 0.4270711343763348 s
```

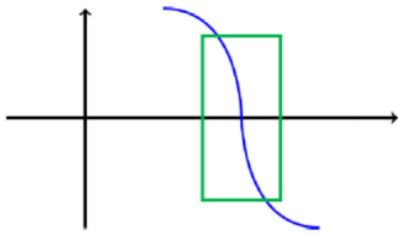
```
Lx=[]
Ly=[]
Lx=pl.arange(0,6,0.01)
for x in Lx:
    Ly.append(f(x))

t1=time.clock()
m=dichotomie(0,2,10**(-8),9)
t2=time.clock()
print('Valeur de x pour f(x)=0 :',m)
print('Temps d\'exécution :',t2-t1,'s')

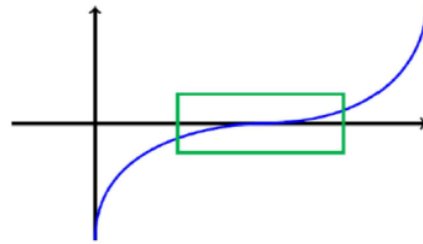
#Tracer la courbe
pl.close()
pl.plot(Lx,Ly)
pl.grid()
pl.plot([m,m],[-1,1])
pl.show()
```



Le test « tant que $ite < itemax$ » sur le nombre d'itérations ite assure l'arrêt de la méthode même si celle-ci ne converge pas.



Courbe à **fort** gradient aux alentours de la racine, le test « tant que $|f_c| > \epsilon$ » sera plus précis.



Courbe à **faible** gradient aux alentours de la racine, le test « tant que $b_n - a_n > 2\epsilon$ » sera plus précis.

Si la fonction f est définie et continue sur l'intervalle $[a, b]$ et n'admet **qu'une seule racine** sur l'intervalle $[a, b]$, alors la méthode de dichotomie converge.

Méthode de Lagrange:

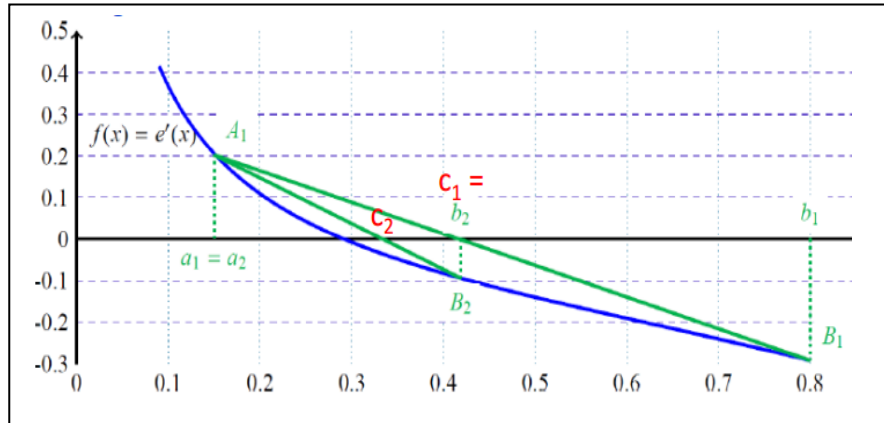
- 1) Diviser l'intervalle $[a, b]$ en deux parts $[a, c]$ et $[c, b]$ avec c point d'intersection entre l'axe des abscisses et la droite passant par $(a, f(a))$ et $(b, f(b))$
- 2) Conserver l'intervalle $[a, c]$ ou $[c, b]$ contenant la racine (phase de test)
- 3) Y reproduire l'opération, jusqu'à ce que le critère de convergence soit satisfait.

→ Equation de la corde:

$$f(x) = y = f(a) + \frac{f(b) - f(a)}{b - a} (x - a)$$

comme $f(c) = 0$ on a donc:

$$f(a) + \frac{f(b) - f(a)}{b - a} (c - a) = 0 \Leftrightarrow c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$



```
import pylab as pl
import math as m
import time
```

```
def f(x):
    return m.cos(x)
```

```
def Lagrange(a,b,eps,itemax):
    i=0
    e=eps
    c=(a*f(b)-b*f(a))/(f(b)-f(a))
    while abs(f(c))>=e and i<=itemax and (b-a)>2*e:
        if f(c)*f(b)<=0: #la racine est entre c et b
            a=c #donc intervalle réduit à gauche
        elif f(c)*f(a)<=0: #la racine est entre a et c
            b=c # donc intervalle réduit à droite
        c=(a*f(b)-b*f(a))/(f(b)-f(a))
        print('m numéro',i+1,'=',c)
        i+=1
    return c
```

```
c numéro 1 = 1.5739063237228796
c numéro 2 = 1.570783521943903
c numéro 3 = 1.5707963268154532
c numéro 4 = 1.5707963267948966
c numéro 5 = 1.5707963267948966
c numéro 6 = 1.5707963267948966
c numéro 7 = 1.5707963267948966
c numéro 8 = 1.5707963267948966
c numéro 9 = 1.5707963267948966
c numéro 10 = 1.5707963267948966
Valeur de x pour f(x)=0 : 1.570796326794896
Temps d'exécution : 0.5039449262386657 s
```

```
t1=time.clock()
m=Lagrange(0,2,10**(-42),9)
t2=time.clock()
print('Valeur de x pour f(x)=0 :',m)
print('Temps d\'exécution :',t2-t1,'s')
```

Convergence non assurée pour cette méthode.

Elle permet cependant dans bien des cas une résolution plus rapide que la *Dichotomie*.

Méthode de Newton:

Si la fonction C^1 sur l'intervalle $[a,b]$:

$$f(b) = f(a) + f'(a) \cdot (b-a) + o(b-a)$$

donc pour x_1 racine:

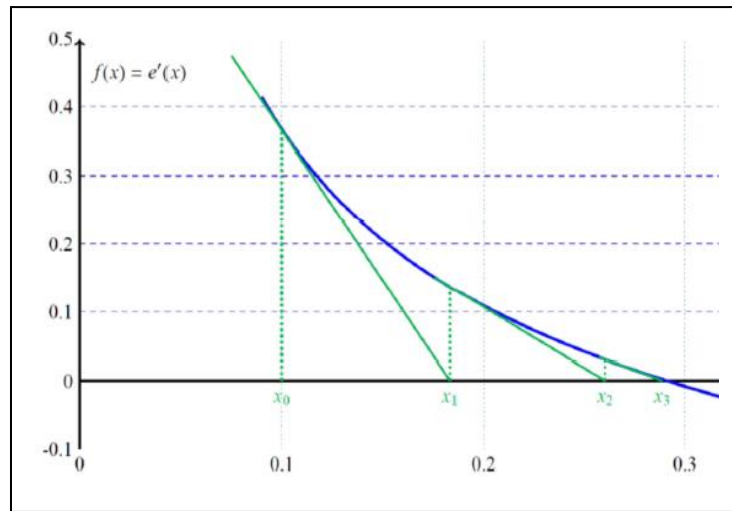
$$f(x_1) = f(x_0) + f'(x_0) \cdot (x_0 - x_1) = 0 \Leftrightarrow x_1 = x_0 - f(x_0) / f'(x_0)$$

```
import pylab as pl
import math as m
import time
```

```
def f(x):
    return m.cos(x)
def df(x):
    return -(m.sin(x))
```

```
def Newton(x0,eps,itmax):
    i,e=0,eps
    xi=x0
    while abs(f(xi))>e and i<itmax:
        i+=1
        if df(xi)==0: #ça plante quand f'(xi)=0
            break
        else:
            xi=xi-(f(xi)/df(xi))
            print('xi numéro',i,'=',xi)
    return xi
```

```
t1=time.clock()
m=Newton(2,10**(-42),12)
t2=time.clock()
print('Valeur de x pour f(x)=0 :',m)
print('Temps d\'exécution :',t2-t1,'s')
```



```
xi numéro 1 = 1.5423424456397141
xi numéro 2 = 1.5708040082580965
xi numéro 3 = 1.5707963267948966
xi numéro 4 = 1.5707963267948966
xi numéro 5 = 1.5707963267948966
xi numéro 6 = 1.5707963267948966
xi numéro 7 = 1.5707963267948966
xi numéro 8 = 1.5707963267948966
xi numéro 9 = 1.5707963267948966
xi numéro 10 = 1.5707963267948966
xi numéro 11 = 1.5707963267948966
xi numéro 12 = 1.5707963267948966
Valeur de x pour f(x)=0 : 1.5707963267948966
Temps d'exécution : 0.5617181176718072 s
```

Selon le gradient de la courbe et le choix de x_0 , la méthode sera plus ou moins efficace.
La méthode de Newton lorsqu'elle converge est souvent plus rapide que les méthodes précédentes.

Méthode du Point Fixe:

→ Consiste à transformer $f(x) = 0$ en une équation équivalente $g(x) = x$ ou g est une fonction auxiliaire "bien" choisie. c tel que $f(c) = 0$ est le point fixe de g , tel que $g(c) = c$. Approcher les zéros de f revient à approcher les points fixes de g .

→ Pour que ça converge, **g doit être contractante** au voisinage I de c :

$$|g'(x)| < 1, \forall x \in I, I \text{ voisinage de } c,$$

→ Dans ce cas on définit la suite (x_n) telle que:

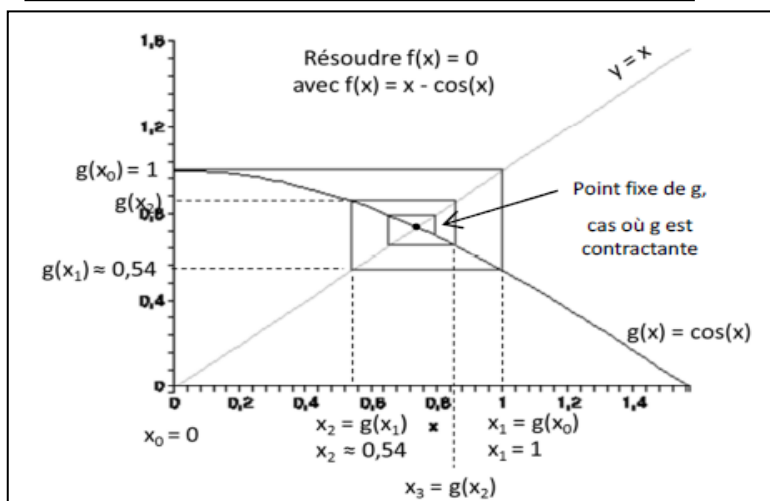
$$\begin{cases} x_0 \text{ dans le voisinage de } I \text{ de } c \\ \forall n \geq 0, x_{n+1} = g(x_n) \end{cases} \quad \text{et} \quad \lim_{n \rightarrow \infty} x_n = c$$

```
import math as m
def g(x):
    return m.cos(x)
```

Méthode du point fixe
 $g(x) = x$

```
def Point_Fixe(x0,eps,itmax):
    i,e=0,eps
    xi=x0
    while abs(g(xi)-xi)>e and i<itmax:
        i+=1
        xi=g(xi)
    return xi
```

```
Point Fixe c= 0.7390851332151607
>>>
```



Intégration Numérique:

Interpolation → déterminer une fonction (dans un ensemble donné), passant par un certain nombre de points imposés. Utile lorsqu'une loi est donnée à partir d'une liste de points et qu'il est nécessaire d'évaluer le résultat en des points intermédiaires.

Approximation → déterminer une fonction passant "au mieux" à proximité des points donnés. Utile lorsqu'une loi théorique est recherchée à partir de points de mesure (nombreux, mais entachés de bruits de mesure).

Interpolation Polynomiale: On cherche, dans une expression du polynôme de degré $\leq n$ prenant les mêmes valeurs qu'une fonction donnée en $n + 1$ points deux à deux distincts donnés.

Soit (x_0, \dots, x_n) $n+1$ complexes distincts 2 à 2 on cherche la famille de polynômes (L_0, \dots, L_n) tel que:

$\forall (i, j) \in [0, \dots, n]^2, L_i(x_j) = \delta_{i,j}$ (0 si $i \neq j$ ou 1 si $i = j$) ainsi:

$\forall (i, j) \in [0, \dots, n]^2, L_i(x) = \lambda \prod_{j \neq i}^n (x - x_j)$ et

$L_i(x_i) = 1$ donc $\lambda = \frac{1}{\prod_{j \neq i}^n (x_i - x_j)}$

On a (L_0, \dots, L_n) qui constitue une base de $C_n[x]$

ainsi

$$P(x) = \sum_{i=0}^n \lambda_i L_i(x)$$

et

$$P(x_i) = y_i = \lambda_i$$

au final:

$$P(x) = \sum_{i=0}^n y_i \prod_{j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)}$$

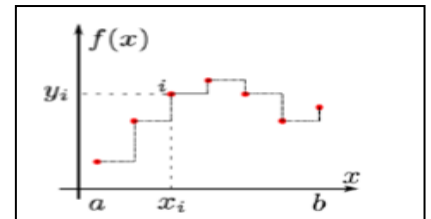
Pour 3 points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$

$$P(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$

→ Pas idéale dès que le nombre de points augmente : le polynôme interpolant peut alors présenter des oscillations entre les points (phénomène de Runge). L'interpolation peut alors être localement très éloignée des points. Pour éviter les oscillations sur certaines fonctions: réaliser une interpolation polynomiale de faible degré mais par morceaux.

→ *Interpolation par morceaux de degrés 0:*

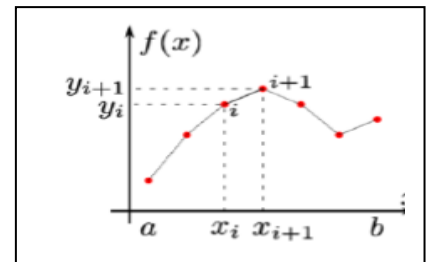
Entre deux points, la valeur de la fonction vaut une constante égale à la valeur du point précédent, du point suivant ou encore égale à la moyenne des valeurs des points encadrant (fonction interpolante non-continue).



→ *Interpolation par morceaux de degrés 1:*

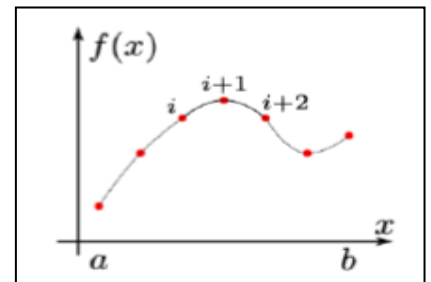
Loi affine $(ax + b)$ entre deux points successifs, passant par les deux points (fonction interpolante continue mais pas sa dérivée)

$$\forall x \in [x_i, x_{i+1}], \quad y = y_i + (x - x_i) \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$



→ *Interpolation par morceaux de degrés 2:*

Loi parabolique $(ax^2 + bx + c)$ est adoptée sur chaque intervalle regroupant 3 points successifs, passant par les 3 points. (fonction est continue mais sa dérivée ne l'est pas car elle présente des discontinuités de la pente entre chaque portion de parabole)



Intégration numérique → intégrer (de façon approchée) une fonction sur un intervalle borné $[a, b]$, à partir d'un calcul ou d'une mesure en un nombre fini de points (avec un pas d'échantillonnage h généralement constant).

L'intégration des polynômes étant très simple, l'opération consiste généralement à construire une interpolation polynomiale par morceaux (de degré plus ou moins élevé) puis d'intégrer le polynôme sur chaque morceau. La précision de l'intégration numérique peut s'améliorer en augmentant le nombre de points n (en diminuant h) ou en augmentant le degré de l'interpolation polynomiale (sous réserve de bonnes propriétés de continuité de la courbe).

Méthode des rectangles :

→ La fonction à intégrer est interpolée par un polynôme de degré 0 (à savoir une fonction constante)

$$\sigma_0 = a < \sigma_1 < \dots < \sigma_{n-1} < \sigma_n = b$$

```
import math as m

def f(x):
    return m.sqrt(1-x**2)
```

```
def rect_gauche(a,b,n):
    s=0
    for k in range(0,n):
        sigma = a+k*(b-a)/float(n)
        s=s+((b-a)/float(n))*f(sigma)
    return s

print('pi par le rect_gauche =',4*rect_gauche(0,1,10000))
```

```
def rect_droite(a,b,n):
    s=0
    for k in range(0,n):
        sigma =a+(k+1)*(b-a)/float(n)
        s=s+((b-a)/float(n))*f(sigma)
    return s
print('pi par le rect_droite =',4*rect_droite(0,1,10000))
```

```
def rect_milieu(a,b,n):
    s=0
    for k in range(0,n):
        sigma1 = a+k*(b-a)/float(n)
        sigma2 =a+(k+1)*(b-a)/float(n)
        s=s+((b-a)/float(n))*f((sigma2+sigma1)/2)
    return s

print('pi par le rect_milieu =',4*rect_milieu(0,1,10000))
```

```
pi par le rect_gauche = 3.141791477611317
pi par le rect_droite = 3.141391477611317
pi par le rect_milieu = 3.1415929980246284
```

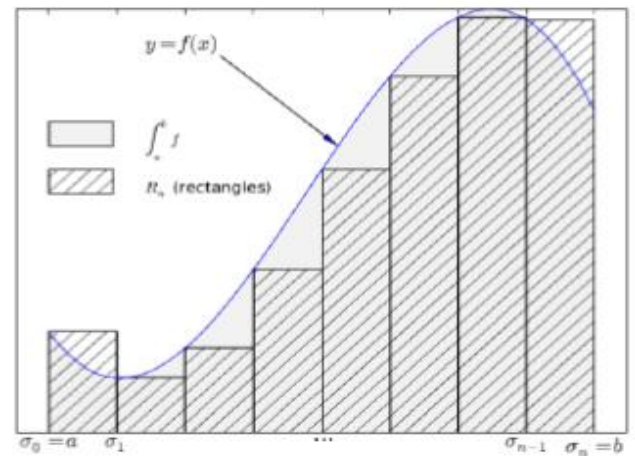
Méthode des trapèzes :

→ La fonction à intégrer est interpolée par un polynôme de degré 1, ce qui conduit à considérer l'aire de trapèzes.

```
def trapeze(a,b,n):
    s=0
    for k in range(0,n):
        sigma = a+k*(b-a)/float(n)
        sigma1 =a+(k+1)*(b-a)/float(n)
        s=s+((b-a)/float(2*n))*(f(sigma)+f(sigma1))
    return s

print('pi par le trapeze =',4*trapeze(0,1,10000))
```

```
pi par le trapeze = 3.141591477611326
```



→ Point d'insertion du rectangles à gauche:

$$R_n = \frac{b-a}{n} \sum_{k=0}^{n-1} f(\sigma_k)$$

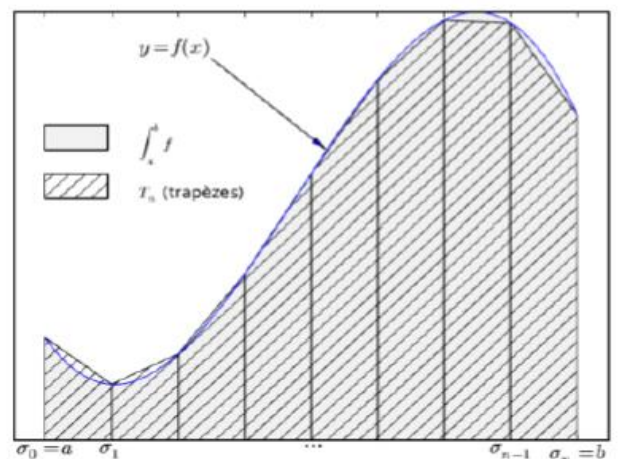
→ Point d'insertion du rectangles à droite:

$$R_n = \frac{b-a}{n} \sum_{k=0}^{n-1} f(\sigma_{k+1})$$

→ Point d'insertion du rectangles au milieu:

$$R_n = \frac{b-a}{n} \sum_{k=0}^{n-1} f\left(\frac{\sigma_k + \sigma_{k+1}}{2}\right)$$

$$\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2} dx$$



$$T_n = \frac{b-a}{2n} \sum_{k=0}^{n-1} (f(\sigma_k) + f(\sigma_{k+1}))$$

Méthode de Simpson :

→ La fonction à intégrer est interpolée par un polynôme de degré 2

On peut interpoler f à l'aide des valeurs en:

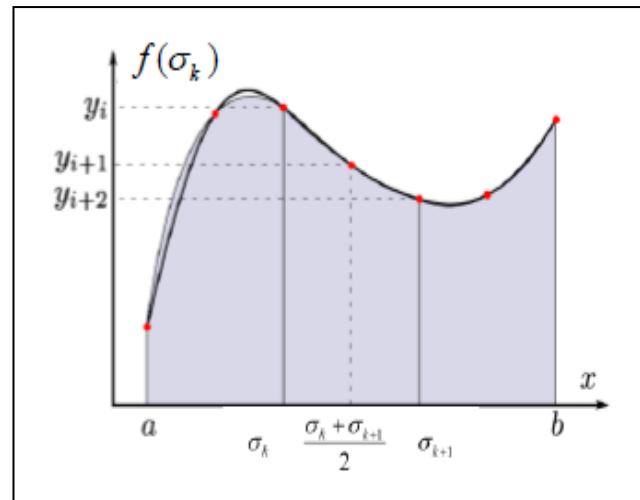
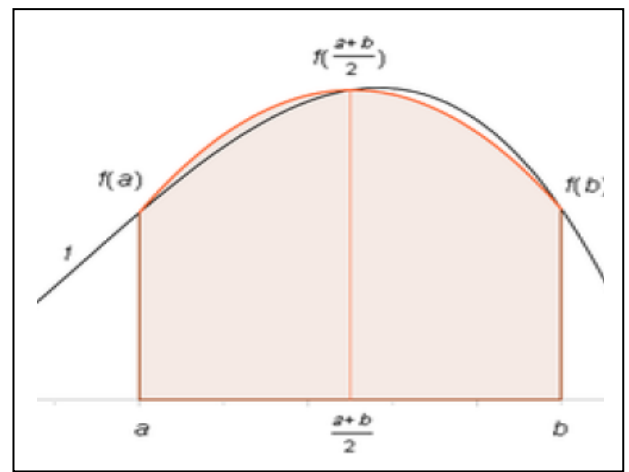
$$a = \sigma_k, \quad m = \frac{\sigma_k + \sigma_{k+1}}{2}, \quad b = \sigma_{k+1}$$

On a alors **localement** des paraboles proches de f .

$$P(x) = f(a) \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \frac{(x-a)(x-m)}{(b-a)(b-m)}$$

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

$$P_n = \frac{b-a}{6n} \sum_{k=0}^{n-1} \left(f(\sigma_k) + 4f\left(\frac{\sigma_k + \sigma_{k+1}}{2}\right) + f(\sigma_{k+1}) \right)$$



```
def Simpson(a,b,n):
    s=0
    for k in range(0,n):
        sigma = a+k*(b-a)/float(n)
        sigma1 = a+(k+1)*(b-a)/float(n)
        s=s+((b-a)/float(6*n))*(f(sigma)+4*f((sigma+sigma1)/2)+f(sigma1))
    return s

print('pi par Simpson =', 4*Simpson(0,1,10000))
```

pi par Simpson = 3.141592491220199

Calcul de l'erreur pour la méthode des rectangles avec point d'insertion à gauche :

Sur l'intervalle élémentaire:

$$[\sigma_k; \sigma_{k+1}]$$

$$f(x) - f(\sigma_k) = f'(c_k)(x - \sigma_k) \quad c_k \in [\sigma_k; \sigma_{k+1}]$$

$$\Rightarrow \int_{\sigma_k}^{\sigma_{k+1}} f(x) dx - h \cdot f(\sigma_k) = \int_{\sigma_k}^{\sigma_{k+1}} f'(c_k)(x - \sigma_k) dx \quad \text{avec } h = \sigma_{k+1} - \sigma_k = \frac{b-a}{n}$$

Si f' est bornée, i.e. $\exists M = \sup_{c \in [a,b]} |f'(c)|$

$$\Rightarrow \left| \int_{\sigma_k}^{\sigma_{k+1}} f(x) dx - h \cdot f(\sigma_k) \right| \leq M \int_{\sigma_k}^{\sigma_{k+1}} (x - \sigma_k) dx = \frac{M}{2} (\sigma_{k+1} - \sigma_k)^2 = M \frac{(b-a)^2}{2n^2}$$

En sommant :

$$\Rightarrow \left| \int_a^b f(x) dx - h \cdot \sum_{i=0}^{n-1} f(\sigma_i) \right| \leq M \frac{(b-a)^2}{2n}$$

| Méthode | Erreur |
|---------------------------------|-------------------------------|
| Rectangles à gauche ou à droite | $O\left(\frac{1}{n}\right)$ |
| Rectangles milieu | $O\left(\frac{1}{n^2}\right)$ |
| Trapèzes | $O\left(\frac{1}{n^2}\right)$ |
| Simpson | $O\left(\frac{1}{n^4}\right)$ |

Dérivation Numérique :

→ Equation différentielle ordinaire (ode) d'ordre 1 :

$$\frac{dy(t)}{dt} = f(t, y(t))$$

f définie et continue sur I
un intervalle réel compact
(fermé et borné)

Méthode d'Euler:

→ Résoudre le problème de Cauchy:

$$\begin{aligned} y'(t) &= f(t, y(t)) \\ y(t_0) &= y_0 \end{aligned} \quad \text{pour tout } t \in [t_0, t_{MAX}]$$

→ Discrétiser l'espace temps en intervalles: $t_0, t_1, \dots, t_{n-1}, t_n = t_{MAX}$
avec h , le pas de temps

→ Taylor à l'ordre 1 donne:

$$t_{k+1} = t_k + h$$

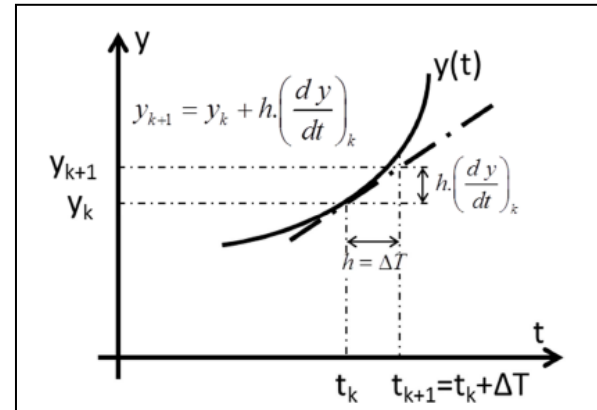
$$\Delta T = h = \frac{t_{MAX} - t_0}{n}$$

$$y(t_k + h) = y(t_k) + h \cdot \frac{dy(t_k)}{dt} + o(h)$$

$$\frac{dy(t_k)}{dt} \approx \frac{y(t_k + h) - y(t_k)}{h}$$

→ Problème de Cauchy approché:

$$\begin{aligned} y_{k+1} &= y_k + h \cdot f(t_k, y_k) \\ y(t_0) &= y_0 \end{aligned} \quad \text{pour tout } t_{k+1} \in [t_0, t_{MAX}] \text{ avec } t_{k+1} = t_k + h$$



Euler Explicite:

$$y_{k+1} = y_k + h \cdot f(t_k, y_k)$$

```
import pylab as pl
import math as m
```

on prend comme exemple l'ode 1 d'un moteur à cc (réponse à un échelon unitaire)

```
def f(t,y):
    return (530-y)/0.2
```

$$\tau \cdot \omega'(t) + \omega(t) = \omega_c$$

donc

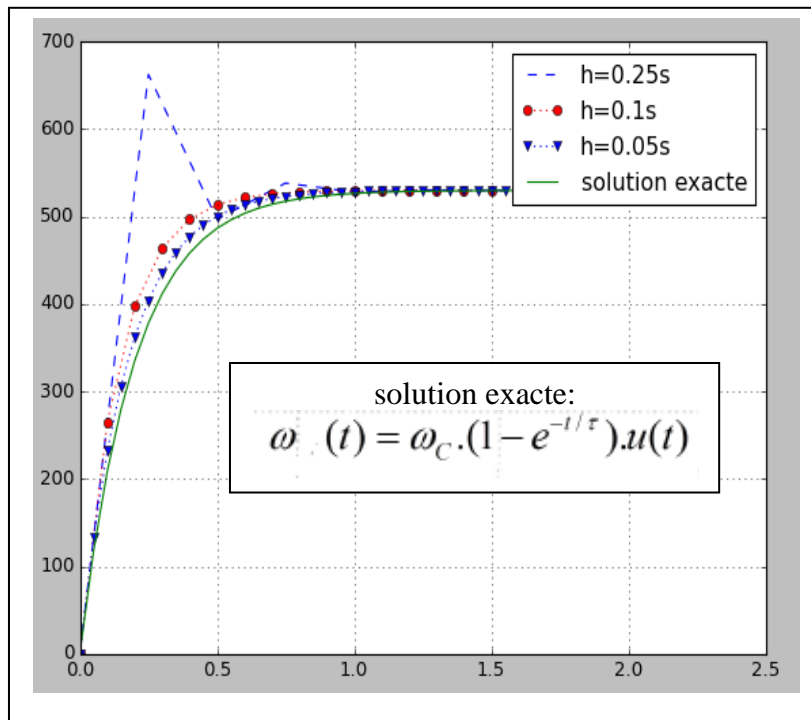
$$\omega'(t) = f(\omega, t) = (\omega_c - \omega(t))/\tau$$

on pose $\omega_c = 530 \text{ rad/s}$ et $\tau = 0.2 \text{ s}$

```
def Euler_exp(tmax,y0,h):
    L_y,L_t=[0],[y0]
    y=y0
    t=0
    while t<tmax:
        t=t+h
        y=y+h*f(t,y)
        L_y.append(y)
        L_t.append(t)
    return L_t,L_y
```

```
#on fait plusieurs calculs en modifiant h:
m1,n1=Euler_exp(2,0,0.25)
m2,n2=Euler_exp(2,0,0.1)
m3,n3=Euler_exp(2,0,0.05)
#on calcul la solution exacte:
m4,n4=[],[]
m4=pl.arange(0,2,0.05)
for t in m4:
    n4.append(530*(1-m.exp(-float(t)/0.2)))
```

```
#on trace les courbes:
pl.close()
pl.plot(m1,n1,'--',label='h=0.25s')
pl.plot(m2,n2,'r:o',label='h=0.1s')
pl.plot(m3,n3,'b:v',label='h=0.05s')
pl.plot(m4,n4,label='solution exacte')
pl.legend()
pl.grid()
pl.show()
```



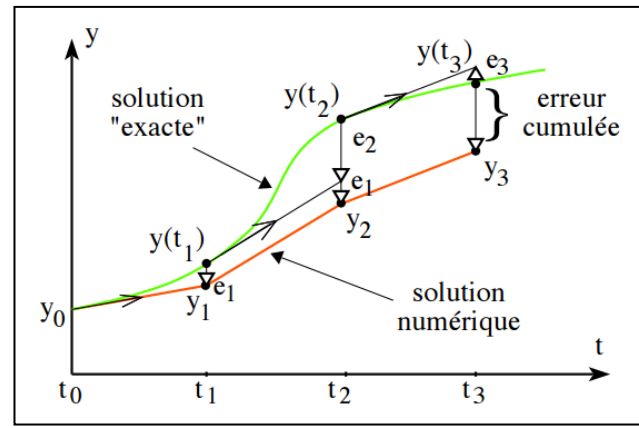
Il est judicieux de prendre un pas de temps h petit
devant le temps caractéristique τ du problème ... quand
on le connaît à l'avance !

→ L'erreur commise à chaque pas (erreur locale) dépend de la concavité de la solution car on a négligé dans Taylor :

$$\frac{1}{2} \frac{d^2 y}{dt^2}(t_k) \cdot h^2$$

→ Si la dérivée seconde est positive, la méthode numérique donnera une erreur locale négative. L'erreur est à chaque pas proportionnelle à h^2 mais d'amplitude et de signe variable

→ L'erreur cumulée au bout de N pas est la somme des erreurs algébriques commises à chaque pas, On peut dire qu'elle est majorée par la somme des valeurs absolues des erreurs locales



Euler Implicite:

$$y_{k+1} = y_k + h \cdot f(t_k, y_{k+1})$$

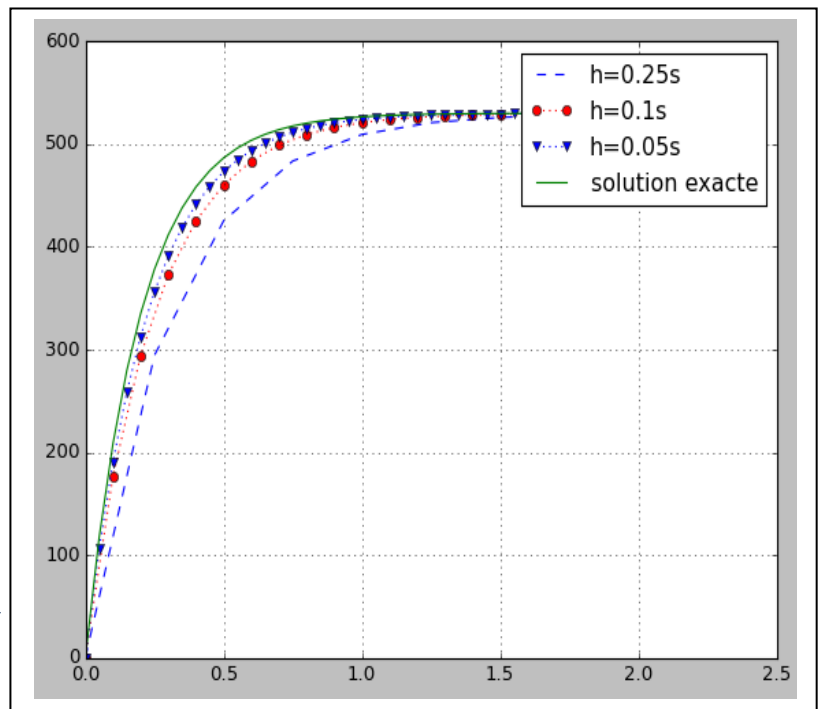
cas pour l'ex du moteur

$$\omega_{Mk+1} = \omega_{Mk} + h \cdot \frac{(\omega_C - \omega_{Mk+1})}{\tau_M} \Leftrightarrow \omega_{Mk+1} = \frac{\tau_M}{\tau_M + h} \cdot \left(\omega_{Mk} + \frac{h}{\tau_M} \cdot \omega_C \right)$$

```
import pylab as pl
import math as m

def f(t,y):
    return (530-y)/0.2

def Euler_impl(tmax,y0,h):
    L_y,L_t=[0],[y0]
    y=y0
    t=0
    while t<tmax:
        t=t+h
        y=(0.2/(0.2+h))*(y+(h/0.2)*530)
        L_y.append(y)
        L_t.append(t)
    return L_t,L_y
```



Le schéma implicite est plus stable que le schéma explicite au détriment de la rapidité (car il implique souvent la résolution numérique d'une équation de type $f(x) = 0$ par méthode itérative)

ODE d'ordre 2:

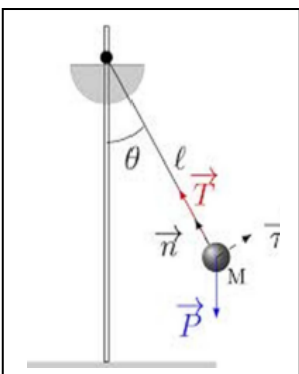
$$\frac{d^2 y}{dt^2} = f(t, y(t), \frac{dy}{dt})$$

→ Ramener à la résolution d'un système de deux ODE d'ordre 1

$$\begin{aligned} \frac{dy}{dt} &= v \\ \frac{dv}{dt} &= f(t, y(t), \frac{dy}{dt}) \end{aligned}$$

→ Il est nécessaire de connaître les conditions initiales $y(0)$ et $v(0)$!

exemple du pendule simple:



$$\frac{d^2 \theta(t)}{dt^2} + \frac{g}{l} \sin(\theta(t)) = 0$$

on pose:

$$\begin{pmatrix} \dot{\theta} \\ \theta \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

donc:

$$\begin{cases} \frac{d\theta(t)}{dt} = \dot{\theta}(t) \\ \frac{d\dot{\theta}(t)}{dt} = -\frac{g}{l} \sin(\theta(t)) \end{cases} \Leftrightarrow \begin{cases} \frac{dy_2(t)}{dt} = y_1(t) \\ \frac{dy_1(t)}{dt} = -\frac{g}{l} \sin(y_2(t)) \end{cases}$$

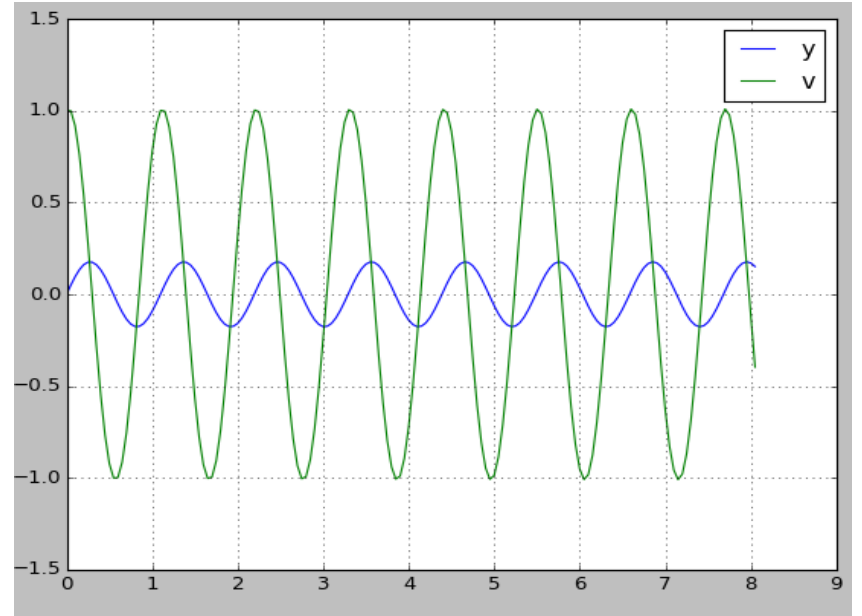
```

import pylab as pl
import math as m

def f(t,y,v):
    return -(9.81/0.3)*m.sin(y)

def Euler_ode2(tmax,y0,v0,h):
    L_y,L_t,L_v=[0],[y0],[v0]
    y,v=y0,v0
    t=0
    while t<tmax:
        t=t+h
        v=v+h*f(t,y,v)
        y=y+h*v
        L_y.append(y)
        L_t.append(t)
        L_v.append(v)
    return L_t,L_y,L_v

```



```

#on fait plusieurs calculs en modifiant h:
m1,n1,v1=Euler_ode2(8,0,1,0.05)
#on trace les courbes:
pl.close()
pl.plot(m1,n1,label='y')
pl.plot(m1,v1,label='v')
pl.legend()
pl.grid()
pl.show()

```

avec numpy :

```

def euler2(f,t_max,h,y0,v0):
    t = np.arange(0.,t_max,h)    # initialisations
    y = np.zeros(len(t))
    v = np.zeros(len(t))
    y[0], v[0] = y0, v0
    for k in range(len(t)-1):    # boucle d'intégration
        y[k+1] = y[k] + h*v[k]
        v[k+1] = v[k] + h*f(t[k],y[k],v[k])
    return t,y,v

```

Euler Explicite avec Numpy:

```

import numpy as np
def euler(f,t_max,h,y0):
    t = np.arange(0.,t_max,h)    # Initialisation des array numpy
    y = np.zeros(len(t))
    y[0] = y0                    # condition initiale
    for k in range(len(t)-1):    # Attention au -1 -> IndexError: index out of bounds
        y[k+1] = y[k] + h*f(t[k],y[k])
    return t,y

```

Création de tableaux & calcul matriciel:

→ Un tableau est la liste des listes de lignes,

→ Pour atteindre un terme de la matrice A: $A[N^{\circ}\text{ligne}][N^{\circ}\text{colonne}]$

```
>>>a = [0, 1, 2, 3]
>>>b = a
```

a et b désignent le même tableau (ce sont des *alias*). Si on modifie un terme de a, b sera modifié. Si on affecte à a un nouveau tableau, b ne sera pas modifié. Utiliser: `b=a.copy()`

Recherche par balayage dans un tableau

```
def appartient(x, a):
    i = 0
    for i in range(len(a)):
        #i allant de 0 à nb de lignes
        for j in range(len(a[i])):
            #j allant de 0 à nb de colonnes
            if a[i][j] == x:
                return True
    return False
a=[[1,2,3],[4,5,6]]
x=3
print(appartient(x,a))
```

Attention au partage de tableau en mémoire ! ex

On peut construire un tableau `v = [0, 1, 2]`, puis l'utiliser trois fois pour chacune des lignes de la matrice : `m = [v, v, v]`

ça donne une matrice (3, 3) mais montre un partage du tableau v entre les lignes ex: si on affecte un nouvel élément à `m[0][1]`, c'est toute la colonne qui est modifiée (`m[0][1]`, `m[1][1]`, `m[2][1]`)

$$m = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$$

```
>>> v = [0, 1, 2]
>>> m = [v, v, v]
>>> m
[[0, 1, 2], [0, 1, 2], [0, 1, 2]]
>>> m[0][1]
1
>>> m[0][1]=4
>>> m
[[0, 4, 2], [0, 4, 2], [0, 4, 2]]
```

Créer une matrice de grande taille:

```
def creer_matrice(n,p,v):
    return [[v]*p for k in range(n)]
```

```
>>> print (creer_matrice(3,3,12))
[[12, 12, 12], [12, 12, 12], [12, 12, 12]]
```

→ Dimension d'une matrice:

Vérifie qu'un tableau de tableaux m représente bien une matrice de dimensions (n, p)

```
def dimensions(m):
    n = len(m)
    assert n > 0
    #assert retourne une erreur si la condition n'est pas réalisée
    p = len(m[0]) #la première ligne existe car n>0
    assert p > 0
    #on vérifie que toutes les lignes de m ont bien la longueur p
    for r in m:
        assert len(r) == p
    return (n, p)
```

→ Transposition:

```
def transpose(m):
    n, p = dimensions(m)
    t = creer_matrice(p,n,None)
    for j in range(p):
        for i in range(n):
            t[j][i] = m[i][j]
    return t
```

```
>>> m=[[1,2,3],[4,5,6]]
>>> transpose(m)
[[1, 4], [2, 5], [3, 6]]
```

→ Multiplication de Matrices:

```
def mult_matrice(a, b):
    n, p = dimensions(a)
    q, r = dimensions(b)
    assert q == p #vérifie si on peut multiplier
    c=creer_matrice(n,r,0)
    for i in range(n):
        for j in range(r):
            for k in range(p):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

```
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[2,4,6],[1,3,5],[8,8,8]]
print('a*b =',mult_matrice(a,b))
```

```
a*b = [[28, 34, 40], [61, 79, 97], [94, 124, 154]]
```

Tableaux Numpy:

→ Tableaux numpy homogènes (constitués d'éléments du même type).
→ Taille des tableaux numpy fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme on le ferait pour une liste. L'empreinte du tableau en mémoire est invariable

→ np.array() former un tableau à partir de listes
→ np.arange(val_initiale, val_finale, pas) retourne un tableau
→ np.zeros((m, n)) tableau de 0 de dimensions (m,n)
→ np.ones((m, n)) tableau de 1 de dimensions (m,n)
→ np.eye(n) matrice identité

```
>>> np.array([[1,2,3],[4,5,6]])  
array([[1, 2, 3],  
       [4, 5, 6]])  
>>> np.arange(0,20,4)  
array([ 0,  4,  8, 12, 16])
```

```
>>> np.zeros((2,3))  
array([[ 0.,  0.,  0.],  
       [ 0.,  0.,  0.]])
```

```
>>> np.eye(3)  
array([[ 1.,  0.,  0.],  
       [ 0.,  1.,  0.],  
       [ 0.,  0.,  1.]])
```

```
>>> np.ones((3,2))  
array([[ 1.,  1.],  
       [ 1.,  1.],  
       [ 1.,  1.]])
```

→ shape : indique le format du tableau, sous la forme du tuple du nombre d'éléments dans chaque direction
→ alen : donne la première dimension d'un tableau (la taille pour un vecteur, le nombre de lignes pour une matrice)
→ size : donne le nombre total d'éléments
→ ndim : renvoie le nombre d'indices nécessaires au parcours du tableau (1 pour un vecteur, 2 pour une matrice)

```
>>> a=np.array([[1,2,3],[4,5,6]])
```

```
>>> np.shape(a)  
(2, 3)  
>>> np.alen(a)  
2  
>>> np.ndim(a)  
2
```

```
>>> np.size(a)  
6  
>>> np.size(a,0)  
2  
>>> np.size(a,1)  
3
```

Lire les valeurs d'une matrice

```
>>> m=np.array([[10*i +j for j in range(8)] for i in range(5)])  
>>> m  
array([[ 0,  1,  2,  3,  4,  5,  6,  7],  
       [10, 11, 12, 13, 14, 15, 16, 17],  
       [20, 21, 22, 23, 24, 25, 26, 27],  
       [30, 31, 32, 33, 34, 35, 36, 37],  
       [40, 41, 42, 43, 44, 45, 46, 47]])
```

```
>>> m[1:4,2:6]  
array([[12, 13, 14, 15],  
       [22, 23, 24, 25],  
       [32, 33, 34, 35]])  
>>> #ligne 1 à 4  
>>> #colonne 2 à 5
```

```
>>> m[1::2,1::2]  
array([[11, 13, 15, 17],  
       [31, 33, 35, 37]])  
>>> #lignes et colonnes impaires
```

```
>>> m[3]  
array([30, 31, 32, 33, 34, 35, 36, 37])  
>>> #vecteur-ligne en position 3
```

```
>>> m[0,0]  
0  
>>> m[2,3]  
23
```

```
>>> m[:,2::2]  
array([[ 0,  2,  4,  6],  
       [20, 22, 24, 26],  
       [40, 42, 44, 46]])  
>>> #lignes et colonnes paires
```

```
>>> m[:,3]  
array([ 3, 13, 23, 33, 43])  
>>> #vecteur colonne position 3
```

```
>>> m[:,2::3]  
array([[ 0,  3,  6],  
       [20, 23, 26],  
       [40, 43, 46]])  
>>> # 1 ligne sur 2  
>>> # 1 colonne sur 3
```

```
>>> m[2:,4:]  
array([[24, 25, 26, 27],  
       [34, 35, 36, 37],  
       [44, 45, 46, 47]])  
>>> #à partir de la ligne2  
>>> #à partir de la colonne 4
```

```
>>> m[:,2::4]  
array([[ 0,  1,  2,  3],  
       [10, 11, 12, 13]])  
>>> # 2 premières lignes  
>>> # 4 premières colonnes
```

```
>>> m[::-1]  
array([[40, 41, 42, 43, 44, 45, 46, 47],  
       [30, 31, 32, 33, 34, 35, 36, 37],  
       [20, 21, 22, 23, 24, 25, 26, 27],  
       [10, 11, 12, 13, 14, 15, 16, 17],  
       [ 0,  1,  2,  3,  4,  5,  6,  7]])  
>>> #inverse l'ordre des lignes
```

```
>>> m[:,::-1]  
array([[40, 41, 42, 43, 44, 45, 46, 47],  
       [30, 31, 32, 33, 34, 35, 36, 37],  
       [20, 21, 22, 23, 24, 25, 26, 27],  
       [10, 11, 12, 13, 14, 15, 16, 17],  
       [ 0,  1,  2,  3,  4,  5,  6,  7]])  
>>> #inverse l'ordre des lignes
```

Si on écrit un entier x dans un tableau de flottants, no problème (x est converti en le flottant correspondant).
Mais si on écrit un flottant x dans un tableau d'entiers, alors x est converti en un entier (**par troncature, pas par arrondi !!!**) Les tableaux Numpy ont leur « data type » et leur taille (le nombre total d'éléments) sont fixés lors de leur création. Pour créer la copie d'un tableau d'un data type: astype

```
>>> a=np.array([[1,2],[3,4]])
>>> a,id(a) #contenu de a et son adresse
(array([[1, 2],
        [3, 4]]), 67950592)
>>> a.dtype
dtype('int32')
>>> #on a un tableaux d'entiers
```

```
>>> a[0][0]=6.666
>>> a
array([[6, 2],
       [3, 4]])
>>> id(a)
67950592
>>> #l'adresse change pas
```

```
>>> b=a.astype(float)
>>> b
array([[ 6.,  2.],
       [ 3.,  4.]])
>>> id(b)
67950432
>>> b.dtype
dtype('float64')
```

Ecrire dans une matrice:

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

```
>>> m[0]=[6,7,8,9];m
array([[ 6,  7,  8,  9],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
>>> #écriture d'1 ligne
```

```
>>> m[:,1]=[6,7,8];m
array([[ 0,  6,  2,  3],
       [10,  7, 12, 13],
       [20,  8, 22, 23]])
>>> #écriture d'1 colonne
```

```
>>> m[1,2]=999;m
array([[ 0,  1,  2,  3],
       [10, 11, 999, 13],
       [20, 21, 22, 23]])
```

```
>>> m[::-1]=m;m
array([[20, 21, 22, 23],
       [10, 11, 12, 13],
       [ 0,  1,  2,  3]])
>>> #ordre des lignes inversé
```

```
>>> m[:,::-1]=m;m
array([[ 3,  2,  1,  0],
       [13, 12, 11, 10],
       [23, 22, 21, 20]])
>>> #ordre des colonnes inversé
```

Si a est un tableau numpy, a.reshape(n,p) renvoie une copie redimensionnée

Suppression de la ligne n → delete(a,n,1)

Suppression de la colonne p → delete(a,p,0)

Ajout d'une ligne → append(a,[[...],],1)

Ajout d'une colonne → append(a,[[...],...],0)

```
>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

```
>>> a=np.arange(6);a
array([0, 1, 2, 3, 4, 5])
>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
```

```
>>> a=np.arange(12).reshape(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
>>> np.delete(a,2,0)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 9, 10, 11]])
>>> #suppression ligne 2
```

```
>>> np.delete(a,2,1)
array([[ 0,  1],
       [ 3,  4],
       [ 6,  7],
       [ 9, 10]])
>>> #suppression colonne 2
```

```
>>> c=np.append(a, [[68],[78],[88],[98]],1);c
array([[ 0,  1,  2, 68],
       [ 3,  4,  5, 78],
       [ 6,  7,  8, 88],
       [ 9, 10, 11, 98]])
>>> #ajout d'1 col apres la derniere col
```

```
>>> b=np.append(a, [[68,78,88]],0);b
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [68, 78, 88]])
>>> #ajout d'1 ligne apres la derniere ligne
```

Transposer une matrice → a.transpose() ou a.T

Produit Matriciel → dot(a,b)

```
>>> a=np.arange(1,5);a
array([1, 2, 3, 4])
>>> b=np.arange(11,15);b
array([11, 12, 13, 14])
```

```
>>> np.dot(a,b)
130
>>> #produit scalaire a.b
```

Calcul produit $m.m^t$:

```
>>> m=np.arange(20).reshape(4,5)
>>> np.dot(m,m.T)
array([[ 30,  80, 130, 180],
       [ 80, 255, 430, 605],
       [130, 430, 730, 1030],
       [180, 605, 1030, 1455]])
```

Calculer pour une matrice carrée: - l'inverse → linalg.inv(a)

- le déterminant → linalg.det(a)

Résoudre le système $ax=b$ → linalg.solve(a,b)

Déterminer le degré de mobilité ou d'hyperstatisme → np.rank(a)

add(a,b) additionne terme à terme les éléments de a et b (autre syntaxe possible : a + b)

subtract(a,b) soustrait terme à terme les éléments de b à ceux de a (autre syntaxe possible : a - b)

multiply(a,b) multiplie terme à terme les éléments de a et b (autre syntaxe possible : a * b)

divide(a,b) quotients (flottants) terme à terme des éléments de a par ceux de b (autre syntaxe : a/b)


```
import numpy as np
```

Pivot de Gauss:

```
def cherche_Pivot(M,i):  
    l=i  
    while M[l][i]==0 :  
        l+=1  
    return l
```

```
def permutation(M,i,j):  
    M[i],M[j] = M[j],M[i]
```

```
def transvection(M,i,j,a):  
    Li,Lj,p = M[i],M[j],len(M[i])  
    M[i] = [Li[k]+a*Lj[k] for k in range(p)]
```

```
def Pivot_de_Gauss(A0,B0):  
    from copy import deepcopy  
    A=deepcopy(A0)  
    B=deepcopy(B0)  
    n=len(A)  
    print('A au départ =', '\n', np.array(A))  
    print('B au départ =', '\n', np.array(B))  
    print('ETAPE 1: Triangulariser A')  
  
    #Mise sous forme Triangulaire  
  
    for i in range(n-1):  
        j=cherche_Pivot(A,i)  
        if j>i:  
            permutation(A,i,j)  
            permutation(B,i,j)  
            print(np.array(A), '\n')  
  
            pivot=A[i][i]  
            for l in range(i+1,n):  
                alpha=-A[l][i]/pivot  
                transvection(A,l,i,alpha)  
                transvection(B,l,i,alpha)  
                print(np.array(A), '\n')  
  
    #Remontée  
  
    print('Etape 2: Faire la remontée')  
  
    for i in range(n-1,-1,-1):  
        pivot=A[i][i]  
        for l in range(i):  
            alpha=-A[l][i]/pivot  
            transvection(A,l,i,alpha)  
            transvection(B,l,i,alpha)  
            print(np.array(A), '\n')  
  
    print('A=', '\n', np.array(A), '\n')  
    print('B=', '\n', np.array(B), '\n')  
    print('Etape 3: Calculer la Solution')  
  
    #Calcul des Solutions:  
  
    sol=[0]*n  
    for i in range(n):  
        sol[i]=B[i][0]/A[i][i]  
    return sol
```

```
A0=[[1,1,2],[1,-1,-1],[1,0,1]]  
B0=[[5],[1],[3]]  
s=Pivot_de_Gauss(A0,B0)  
print('sol=',s)
```

A au départ =

```
[[ 1  1  2]  
 [ 1 -1 -1]  
 [ 1  0  1]]
```

B au départ =

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

ETAPE 1: Triangulariser A

```
[[ 1.  1.  2.]  
 [ 0. -2. -3.]  
 [ 1.  0.  1.]]
```

```
[[ 1.  1.  2.]  
 [ 0. -2. -3.]  
 [ 0. -1. -1.]]
```

```
[[ 1.  1.  2. ]  
 [ 0. -2. -3. ]  
 [ 0.  0.  0.5]]
```

Etape 2: Faire la remontée

```
[[ 1.  1.  0. ]  
 [ 0. -2. -3. ]  
 [ 0.  0.  0.5]]
```

```
[[ 1.  1.  0. ]  
 [ 0. -2.  0. ]  
 [ 0.  0.  0.5]]
```

```
[[ 1.  0.  0. ]  
 [ 0. -2.  0. ]  
 [ 0.  0.  0.5]]
```

A=

```
[[ 1.  0.  0. ]  
 [ 0. -2.  0. ]  
 [ 0.  0.  0.5]]
```

B=

```
[[ 3.]  
 [-4.]  
 [ 0.]]
```

Etape 3: Calculer la Solution

```
sol2= [3.0, 2.0, 0.0]
```

Le problème de ce programme c'est que dès qu'une ligne s'annule, ça plante ! Du coup ça marche que pour les systèmes ayant autant de solutions que d'inconnues!

```
[[ 1. -1.  1.  1.]  
 [ 0.  7. -6. -8.]  
 [ 0.  0.  0. -3.]  
 [ 0.  0.  0.  0.]]
```

Traceback (most recent call last):

```
File "C:\Users\Andrea\Desktop\fichier python\pivot de gauss.py", line 80, in <module>  
    print(Pivot_de_Gauss(A2,B2))  
File "C:\Users\Andrea\Desktop\fichier python\pivot de gauss.py", line 32, in Pivot_de_Gauss  
    j=cherche_Pivot(A,i)  
File "C:\Users\Andrea\Desktop\fichier python\pivot de gauss.py", line 16, in cherche_Pivot  
    while M[1][i]==0 :  
IndexError: list index out of range
```

Tris:

```
def insertion(L,i):
    p=i
    while p>0 and L[p]<L[p-1]:
        L[p],L[p-1]=L[p-1],L[p]
        p=p-1

def tri_insertion(L):
    for i in range(1,len(L)):
        insertion(L,i)
    return L
```

au départ : [3, 5, 1, 4, 2]
 on permute 5 et 1 : [3, 1, 5, 4, 2]
 on permute 3 et 1 : [1, 3, 5, 4, 2]
 on permute 5 et 4 : [1, 3, 4, 5, 2]
 on permute 5 et 2 : [1, 3, 4, 2, 5]
 on permute 4 et 2 : [1, 3, 2, 4, 5]
 on permute 3 et 2 : [1, 2, 3, 4, 5]
 à l'arrivée : [1, 2, 3, 4, 5]

Tri Rapide (Quicksort):

```
def tri_rapide(L):
    if len(L)<=1:
        return L
    else:
        L1,L2=[],[]
        cle=L.pop()
        for x in L:
            if x<cle:
                L1.append(x)
            else:
                L2.append(x)
        return tri_rapide(L1)+[cle]+tri_rapide(L2)
```

- On sélectionne arbitrairement un élément clé de la liste L à trier.

L =

| | | |
|--|-----|--|
| | clé | |
|--|-----|--|

- On crée deux tableaux L1 et L2 qui contiennent respectivement les éléments de L inférieurs et supérieurs à clé.

| |
|----|
| L1 |
|----|

| |
|-----|
| clé |
|-----|

| |
|----|
| L2 |
|----|

- Par un appel récursif, on trie les tableaux L1 et L2. On fusionne alors L1, [clé] et L2.

| |
|----------|
| L1 triée |
|----------|

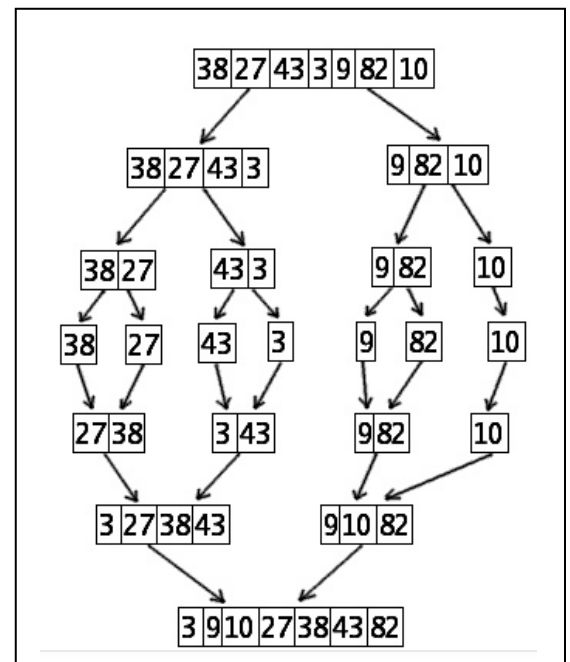
| |
|-----|
| clé |
|-----|

| |
|----------|
| L2 triée |
|----------|

Tri Fusion

```
def fusion(L1,L2):
    s=[]
    for x in L1:
        while L2!=[] and x>L2[0]:
            s.append(L2[0])
            del(L2[0]) #suppression du 1er terme de L2
        s.append(x)
    s=s+L2 #les termes restant de L2 sont tous > x
    return s

def tri_fusion(L):
    if len(L)<2:
        return L
    else:
        L1=tri_fusion(L[:len(L)//2])
        L2=tri_fusion(L[len(L)//2:])
    return fusion(L1,L2)
```



Tri Sélection:

```
def tri_selection(L):
    for i in range(len(L)):
        min=i
        for j in range(i,len(L)):
            if L[min]>L[j]:
                min=j
        if min is not i:
            L[i],L[min]=L[min],L[i]
    return L
```

Liste non triée = [5, 8, 9, 2, 7, 7, 0]
 [0, 8, 9, 2, 7, 7, 5]
 [0, 2, 9, 8, 7, 7, 5]
 [0, 2, 5, 8, 7, 7, 9]
 [0, 2, 5, 7, 8, 7, 9]
 [0, 2, 5, 7, 7, 8, 9]
 [0, 2, 5, 7, 7, 8, 9]
 [0, 2, 5, 7, 7, 8, 9]
 Liste triée = [0, 2, 5, 7, 7, 8, 9]

Tri par Insertion Dichotomique:

```
def dichot(L,e):
    n=len(L)
    g,d=0,n-1
    while g<=d:
        m=(g+d)//2
        if e==L[m]:
            return m
        elif e<L[m]:
            d=m-1
        else:
            g=m+1
    return g
```

→ But: dans une liste
TRIEE, retourne la
position d'un terme e (pas
forcement dans la liste)
de sorte que lorsque l'on
injecte e dans cette liste,
elle reste triée !!

```
liste non triée = [2, 5, 9, 1, 3, 4, 8]
Ltriée = [2]
position que doit avoir 5 dans Ltriée = 1
Ltriée = [2, 5]
position que doit avoir 9 dans Ltriée = 2
Ltriée = [2, 5, 9]
position que doit avoir 1 dans Ltriée = 0
Ltriée = [1, 2, 5, 9]
position que doit avoir 3 dans Ltriée = 2
Ltriée = [1, 2, 3, 5, 9]
position que doit avoir 4 dans Ltriée = 3
Ltriée = [1, 2, 3, 4, 5, 9]
position que doit avoir 8 dans Ltriée = 5
Ltriée = [1, 2, 3, 4, 5, 8, 9]
au final Ltriée = [1, 2, 3, 4, 5, 8, 9]
```

```
def tri_insertion_dicho(L):
    Ltriée=[]
    for x in L:
        if len(Ltriée)==0:
            Ltriée.append(L[0])
        else:
            p=dichot(Ltriée,x)
    #trouve la position que doit avoir x
    #pour que la liste reste triée
    Ltriée.insert(p,x)
    #insert x à la position p dans Ltriée
    return Ltriée
```

Tri à Bulles:

```
def tri_bulle(L):
    n=len(L)
    permute_element=True
    while permute_element==True:
        permute_element=False#arrête la boucle while dès que
        #la liste est triée et donc qu'il n'y a plus rien à permuer
        for i in range(0,n-1):
            if L[i]>L[i+1]:
                permute_element=True
                L[i],L[i+1]=L[i+1],L[i]
    return L
```

```
Liste non triée= [2, 7, 7, 0]
[2, 7, 7, 0]
[2, 7, 7, 0]
[2, 7, 0, 7]
[2, 7, 0, 7]
[2, 0, 7, 7]
[2, 0, 7, 7]
[0, 2, 7, 7]
[0, 2, 7, 7]
[0, 2, 7, 7]
[0, 2, 7, 7]
[0, 2, 7, 7]
[0, 2, 7, 7]
Liste triée = [0, 2, 7, 7]
```

→ consiste à comparer deux à deux et successivement tous les couples consécutifs d'éléments d'une liste donnée et à les permuter dans le cas où le deuxième est plus petit que le premier. On répète l'opération jusqu'à ce que la liste soit triée

```
def tri_gnome(L):
    i,n=0,len(L)
    while i<n-1:
        if L[i+1]>=L[i]:
            i=i+1
        else:
            L[i],L[i+1]=L[i+1],L[i]
            if i>0:
                i=i-1
    return L
```

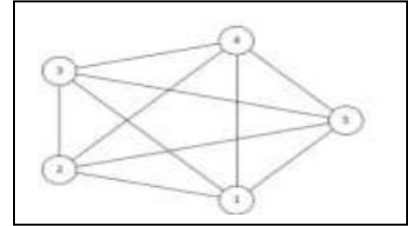
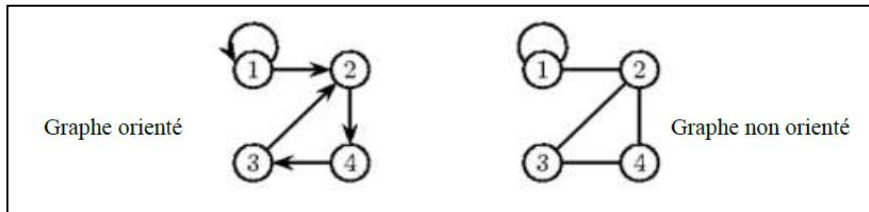
```
Liste non triée = [2, 7, 1, 0]
on avance car 2 < 7 : [2, 7, 1, 0]
on permute 1 et 7 et on recule : [2, 1, 7, 0]
on permute 1 et 2 et on recule : [1, 2, 7, 0]
on avance car 1 < 2 : [1, 2, 7, 0]
on avance car 2 < 7 : [1, 2, 7, 0]
on permute 0 et 7 et on recule : [1, 2, 0, 7]
on permute 0 et 2 et on recule : [1, 0, 2, 7]
on permute 0 et 1 et on recule : [0, 1, 2, 7]
on avance car 0 < 1 : [0, 1, 2, 7]
on avance car 1 < 2 : [0, 1, 2, 7]
on avance car 2 < 7 : [0, 1, 2, 7]
Liste triée = [0, 1, 2, 7]
```

Graphes $G = (S, A) \rightarrow$ donnée d'un ensemble fini **S** de points (= ensemble des sommets)
et d'un ensemble **A** de liens entre ces points (= ensemble des arêtes)

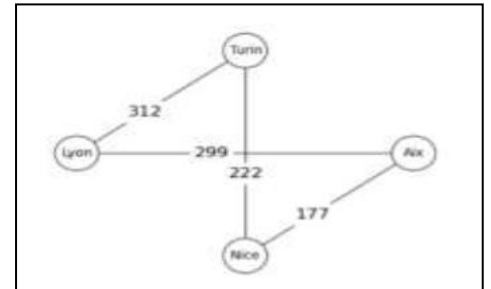
.Si $\{u, v\} \in A$ est une arête de G (souvent noté $u - v$), les sommets u et v sont adjacents.

.Ordre d'un graphe \rightarrow cardinal de son ensemble de sommets.

.Graphe complet d'ordre n (noté K_n) \rightarrow l'unique graphe non orienté tel que toute paire de sommets (distincts) soit reliée.



.Graphe pondéré (ou valué) \rightarrow graphe (orienté ou non) où les arêtes sont affectées d'un poids qui est un nombre réel. On considérera pour certains algorithmes le seul cas où le poids affecté est strictement positif. (exemple des situations de distances dans un réseau routier)



.Chemin \rightarrow suite consécutive d'arcs dans un graphe orienté.

(Dans le cas d'un graphe non orienté on parle de chaîne.)

.Cycle \rightarrow chemin (ou chaîne) pour lequel le sommet de départ et le sommet d'arrivée sont identiques.

.Pour un graphe non pondéré: -Longueur d'une chaîne = son nombre d'arêtes (si graphe non orienté)

-Longueur d'un chemin = son nombre d'arcs (si graphe orienté)

.Pour un graphe pondéré: -Longueur d'une chaîne = somme du poids de ses arêtes (si graphe non orienté)

-Longueur d'un chemin = somme du poids de ses arcs (si graphe orienté)

.Un graphe est connexe s'il existe un chemin entre tout couple de sommets.

.Algorithme de parcours en profondeur = consiste à déterminer s'il existe un chemin d'un sommet à un autre et donc tester s'il est connexe ou non. Principe = partir d'un sommet s et de le marquer puis d'appeler récursivement la fonction sur tous les sommets non marqués qui sont reliés à s . Nous utiliserons une liste L , initialement vide, qui correspond aux sommets marqués (déjà visités).

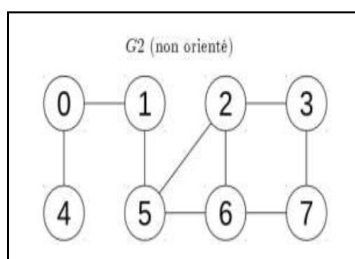
.Un graphe est Hamiltonien s'il a au moins un cycle passant par tous les sommets exactement une fois (cycle Hamiltonien).

```
def DFS(G,s,L):
    L.append(s)
    for sommet in G.voisins(s):
        if not sommet in L:
            DFS(G,sommet,L)
    return L
```

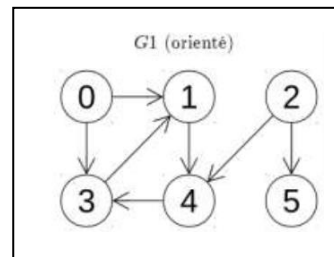
Soit $G=(S,A)$ un graphe non pondéré. Matrice d'adjacence M = la matrice carrée d'ordre $\text{card}(S)$, dont chaque élément M_{ij} est égal à 0 s'il n'y a pas de d'arête liant i à j et 1 s'il existe une arête reliant i à j .

Si G est non orienté, sa matrice d'adjacence est symétrique.

Dans le cas d'un graphe pondéré, M_{ij} est égal au poids de l'arête liant i à j .



$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$



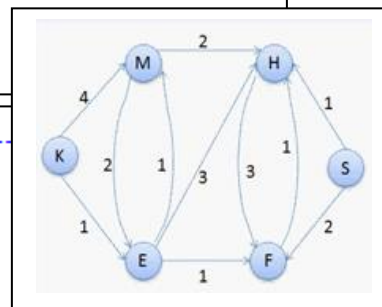
$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$L = [[1, 4], [0, 5], [3, 5, 6], [2, 7], [0], [1, 2, 6], [2, 5, 7], [3, 6, 1]]$

$L = [[1, 3], [4], [4, 5], [1], [3], []]$


```
def Dijkstra_dictionnaire(G,s_debut):
#-----INITIALISATION-----
    infini=sum(sum(G[sommet][suc] for suc in G[sommet]) for sommet in G) +1
    #on choisi infini = somme de tous les poids du graphe +1
    #soit un majorant de la plus grande longueur d'un chemin qq du graphe
    s_connu={s_debut:[0,[s_debut]]}
    #à la fin s_connu={'s du graphe':[longueur ,[chemin le plus court]]}
    s_inconnu={k:[infini,''] for k in G if k!=s_debut}
    #tous les autres sommets assortis de la longueur infini avec aucun
    #prédécesseur pour l'instant
    for suivant in G[s_debut]:
        s_inconnu[suivant]= [G[s_debut][suivant],s_debut]
        #initialisation avec les sommets voisins de s_debut

#-----RECHERCHE-----
    while s_inconnu and (s_inconnu[k][0]<infini for k in s_inconnu):
        #tant que s_inconnu est non vide et qu'il possède des sommets "atteignables"
        u=min(s_inconnu)
        longueur_u,precedent_u =s_inconnu[u]
        for v in G[u]:
            if v in s_inconnu:
                d=longueur_u + G[u][v]
                if d<s_inconnu[v][0]:
                    s_inconnu[v]=[d,u]
                    #u prend la place de s_debut,
                    #d est la nouvelle distance au sommets voisins
        s_connu[u]=[longueur_u, s_connu[precedent_u][1]+[u]]
        del s_inconnu[u]
    return s_connu
```



-----INITIALISATION-----

Sommet de départ ---> K
 Au départ s_inconnu =
 {'M': [21, ''], 'F': [21, ''], 'H': [21, ''], 'E': [21, '']}
 K a pour sommet voisin : M ---> s_inconnu[M] = [4, 'K']
 K a pour sommet voisin : E ---> s_inconnu[E] = [1, 'K']
 s_inconnu devient :
 {'M': [4, 'K'], 'F': [21, ''], 'H': [21, ''], 'E': [1, 'K']}

-----RECHERCHE-----

u = min(s_inconnu)= E
 on atteint E :
 -Par le chemin le plus court: ['K', 'E']
 -De longueur: 1
 s_inconnu devient : {'M': [2, 'E'], 'F': [2, 'E'], 'H': [4, 'E']}

u = min(s_inconnu)= F
 on atteint F :
 -Par le chemin le plus court: ['K', 'E', 'F']
 -De longueur: 2
 s_inconnu devient : {'M': [2, 'E'], 'H': [3, 'F']}

u = min(s_inconnu)= H
 on atteint H :
 -Par le chemin le plus court: ['K', 'E', 'F', 'H']
 -De longueur: 3
 s_inconnu devient : {'M': [2, 'E']}

u = min(s_inconnu)= M
 on atteint M :
 -Par le chemin le plus court: ['K', 'E', 'M']
 -De longueur: 2
 s_inconnu devient : {}
 ---> fin de la boucle

```
G={ 'K':{ 'M':4, 'E':1},
      'M':{ 'E':2, 'H':2},
      'E':{ 'M':1, 'H':3, 'F':1},
      'H':{ 'F':3},
      'F':{ 'H':1, 'S':2}}
```

Dijkstra_dictionnaire(G,'K')

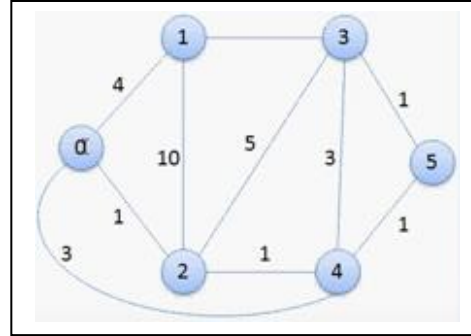
en sortie:
 s_connu = { 'E': [1, ['K', 'E']],
 'M': [2, ['K', 'E', 'M']],
 'F': [2, ['K', 'E', 'F']],
 'H': [3, ['K', 'E', 'F', 'H']],
 'K': [0, ['K']]}

```
def Dijkstra_matrice(G,s_debut):
#-----INITIALISATION-----
    nb_sommets=len(G)
    infini=sum(sum(ligne) for ligne in G)+1
    s_connu={s_debut:[0,[s_debut]]}
    s_inconnu={k:[infini,''] for k in range(nb_sommets) if k!=s_debut}
    for suivant in range(nb_sommets):
        if G[s_debut][suivant]!=0:
            s_inconnu[suivant]=[G[s_debut][suivant],s_debut]

#-----RECHERCHE-----
    while s_inconnu and (s_inconnu[k][0]<infini for k in s_inconnu):
        u=min(s_inconnu)
        longueur_u,precedent_u=s_inconnu[u]
        for v in range(nb_sommets):
            if G[u][v] and v in s_inconnu:
                d=longueur_u + G[u][v]
                if d<s_inconnu[v][0]:
                    s_inconnu[v]=[d,u]
        s_connu[u]=[longueur_u, s_connu[precedent_u][1]+[u]]

        del s_inconnu[u]

    return s_connu
```



```
s_connu= {0: [0, [0]], 1: [4, [0, 1]], 2: [1, [0, 2]], 3: [5, [0, 1, 3]],
4: [2, [0, 2, 4]], 5: [3, [0, 2, 4, 5]]}
```

```
[[ 0, 4, 1, 0, 3, 0],
 [ 4, 0, 10, 1, 0, 0],
 [ 1, 10, 0, 5, 1, 0],
 [ 0, 1, 5, 0, 3, 1],
 [ 3, 0, 1, 3, 0, 1],
 [ 0, 0, 0, 1, 1, 0]]
```

```
import numpy as np
def conversion_adj(A):
    n=len(A)
    L=np.zeros((n,n))
    for i in range(n):
        for j in A[i]:
            L[i,j]=1
    return L
```

```
-Liste de couples de A :
[[0, 1], [2, 0], [1, 2], [2, 3], [3, 1]]
-Matrice d'adjacence :
[[ 1.  1.  0.  0.  0.]
 [ 1.  0.  1.  0.  0.]
 [ 0.  1.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  1.  0.  1.  0.]]
```

```
def conversion_list(L):
    n=len(L)
    A=[]
    for k in range(n):
        A.append([])
    for i in range(n):
        for j in range(n):
            if L[i,j]==1:
                A[i].append(j)
    return(A)
```

```
-Matrice d'adjacence :
[[ 1.  1.  0.  0.  0.]
 [ 1.  0.  1.  0.  0.]
 [ 0.  1.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  1.  0.  1.  0.]]
-Liste de couple:
[[0, 1], [0, 2], [1, 2], [2, 3], [1, 3]]
```

Piles:

- Un objet = concept, idée ou toute entité du monde physique ou virtuel.
- Un conteneur = objet permettant de stocker d'autres objets (non forcément distincts).
- Une structure de données = conteneur dynamique (i.e. de longueur modifiable) destinée à organiser des données informatiques pour maximiser l'efficacité de certaines opérations

Un conteneur S peut être étudié selon trois points de vue :

- l'accès, c'est-à-dire la manière d'accéder aux éléments du conteneur.
- le stockage, c'est-à-dire la manière de stocker les éléments du conteneur ;
- le parcours, c'est-à-dire la manière de parcourir les éléments du conteneur.

Soit E un ensemble non vide. Une pile p d'éléments de E est une structure de données qui met en œuvre le principe : **dernier entré, premier sorti** ou LIFO (Last - In - First - Out)

Une pile P est :

- soit vide (de profondeur 0) ;
- soit une pile de profondeur $n > 1$, de la forme $p = (q, s)$ où s est un élément de E, appelé sommet de p et q une pile de profondeur $n-1$, appelé queue de p.

Opérations primitives :

```
def est_vide(p):  
    return len(p)==0  
#renvoi Vrai si pile vide  
#sinon renvoi Faux
```

```
def pile_vide(p):  
    return []  
#retourne une pile vide
```

```
def empiler(p,s):  
    p.append(s)  
#ajoute s au sommet de p
```

```
def depiler(p):  
    if est_vide(p):  
        print("La pile est vide!")  
    else:  
        return(p.pop())  
#retourne et supprime le sommet  
#d'une pile non vide p
```

```
def sommet(p):  
    if est_vide(p):  
        print("La pile est vide!")  
    else:  
        return(p[-1])  
#retourne sans supprimer le  
#sommet d'une pile p non vide
```

```
def afficher_pile(p):  
    q=[]  
    print('-----')  
    print('-SOMMET-')  
    print('-----')  
    while not est_vide(p):  
        s=depiler(p)  
        print(s)  
        empiler(q,s)  
    while not est_vide(q):  
        empiler(p,depiler(q))  
    print('-----')  
    print('--BASE--')  
    print('-----')  
  
afficher_pile([1,2,3,4,5,6])
```

```
-----  
-SOMMET-  
-----  
6  
5  
4  
3  
2  
1  
-----  
--BASE--  
-----
```

```
def retireCar(p,car):  
    q=[]  
    while not est_vide(p):  
        s=depiler(p)  
        if s!=car:  
            empiler(q,s)  
    while not est_vide(q):  
        empiler(p,depiler(q))  
    return afficher_pile(p)  
  
print(retireCar(['i','n','f','i','n','i'],'i'))
```

```
-----  
-SOMMET-  
-----  
n  
f  
n  
-----  
--BASE--  
-----
```