Les fichiers en python

En programmation, il est souvent utile de séparer le fichier contenant le programme des fichiers contenant des données utiles au programme. Ces fichiers peuvent servir comme données initiales mais aussi comme résultats du programme. C'est pourquoi, il est très important de pouvoir manipuler les fichiers par des commandes Python.

1. Les fichiers .txt

On ouvre un fichier texte avec la commande :

fichier = open('nom du fichier.txt', 'mode')

Où 'mode' est une lettre :

- **r** : ouverture pour lecture seule. si le fichier n'existe pas, Python signale une erreur.
- **w** : ouverture pour écriture. si le fichier n'existe pas, il est créé, s'il existe, son contenu est écrasé et remplacé par ce qu'on y écrit.
- **a** : ouverture pour ajout. Si le fichier n'existe pas, il est créé, sinon, l'écriture s'effectue à la suite du contenu.

- On ferme ensuite son fichier avec la commande : **fichier.close ().**
- On écrit dans un fichier avec la commande :

fichier.write("j'ecris des mots dans mon fichier"). Si on utilise plusieurs fois write, les chaines de caractères sont écrites les unes à la suite des autres. Pour aller à la ligne, on utilise \n.

- On lit une ligne avec la commande **fichier.readline()** : cette instruction renvoie une chaine de caractère contenant tout le contenu de la ligne lue.
- On lit toutes les lignes avec la commande **fichier.readlines()** : cette instruction renvoie une liste dont le ième élément contient le contenu de la ième ligne sous forme d'une chaine de caractères.
- On lit tout le fichier avec **fichier.read()** : cette instruction renvoie une chaine de caractères contenant tout le contenu du fichier.
- On lit les 30 caractères suivants avec **fichier.read(30)** : cette instruction renvoie une chaine de caractères.

Remarques :

- Tous les fichiers txt manipulés peuvent être ouverts par un éditeur de texte. Cela permet de contrôler ce qu'on fait.
- Par défaut, Python travaille dans le répertoire courant.

On peut lui imposer de travailler dans un autre répertoire ainsi :

```
>>>from os import chdir
>>>chdir("C:\Documents and Settings\ nom utilisateur\Mes documents") où
```

''C :\Documents and Settings\ nom utilisateur\ Mes documents'' est l'adresse du dossier "Mes Documents" et chdir siginifie "change directory" : une fois cette ligne exécutée, Python travaillera donc dans le dossier ''C:\Documents and Settings\nom utilisateur\Mes documents''.

Exemple :

Pour se familiariser avec toutes ces commandes, exécuter ces lignes et bien observer dans son éditeur de texte les modifications apportées. Etre attentif aussi au type des objets renvoyés par Python.

```
>>>fichier=open('test.txt','w')
>>>fichier.write("bonjour a tous")
>>>fichier.close()
>>>fichier=open('test.txt','a')
>>>fichier.write(" et a toutes \n Bienvenue dans Python")
>>>fichier=open('test.txt','r')
>>>fichier.readlines()
['bonjour a tous et a toutes \n', ' Bienvenue dans Python']
>>>fichier.close()
>>>fichier=open('test.txt','a')
>>>fichier.write('\n Il fait beau')
>>>fichier.close()
>>>fichier=open('test.txt','r')
>>>L=fichier.read(10)
>>>L
bonjour a
>>>fichier.read()
tous et a toutes
Bienvenue dans Python
Il fait beau
```

- **Se déplacer dans un fichier : fseek(taille, position)**

  Syntaxe : fichier.seek(taille de déplacement, position de départ)

  Position de départ = 0 : début de fichier
  = 1 : position courante
  = 2 : fin du fichier

  Taille de déplacement : >0 : Déplacement en bas
  <0 : Déplacement en haut

```
>>>fichier=open('test.txt','r')
>>>L=fichier.seek(10,1) # se déplacer de 10 octets à partir de la position courante
```

2. Les fichiers .csv

Les fichiers .csv (comma separated value) sont des fichiers textes.

Leur particularité est la présence d'un caractère séparateur (la virgule ou le point virgule).

On représente souvent ce type de fichier dans un tableau de la façon suivante : chaque ligne correspond à une ligne du tableau et le séparateur correspond aux séparations entre les colonnes.

Ainsi, un fichier .csv peut être visualisé au choix avec un éditeur de texte ou un tableur.

Par exemple :

Prénom, Taille, Code postal

Ali, 151, 31000

Chaimae, 165, 10000

Adili, 177, 30000

se représente dans un tableur ainsi :

Prénom	Taille	Code Postal
Ali	151	31000
Chaimae	165	10000
Adil	177	30000

On utilise sur les fichiers .csv (qui sont donc des fichiers texte) les mêmes commandes que pour les fichiers texte.

Notons que la commande split peut être pratique pour extraire des données.

Commençons par quelques exemples qu'on observera en ouvrant en parallèle un tableur.

Ici, le séparateur est le point-virgule.

```
>>>fi=open('tab.csv','w')
>>>fi.write('Prenom;Age;Ville\nKawtar;18;Fes\nSaid;23;Rabat')
>>>fi.close()
>>>fi=open('tab.csv','r')
>>>L=fi.read()
>>>L.split('\n')
>>>fi.close()
>>>fi=open('tab.csv','a')
>>>fi.write("\n Jalal ;15;Taza")
>>>fi.close()
>>>fi=open('tab.csv','r')
>>>fi.read()
```

3. Exercices :

1. Ecrire un script Python qui prend en argument un fichier texte et qui renvoie le nombre de lignes et le nombre de caractères du fichier. ***def NombreLigneCaractere(fichier)***
2. Ecrire une script Python qui prend en argument un fichier texte et qui renvoie un fichier texte avec le contenu du fichier texte mais sans les sauts de lignes. ***def EliminerSautLigne(fichier)***
3. Ecrire un script Python qui prend en argument deux fichiers fichier1 et fichier2 et qui écrit à la fin de fichier1 le contenu de fichier2. ***def CopierF2EnFinF1(fichier1, fichier2)***

4. Ecrire un script Python qui demande à l'utilisateur de donner un entier n supérieur ou égal à 2 et qui renvoie dans un fichier texte les tables de multiplication des couples (i, j) de [1, n].
5. Ecrire un script Python qui prend en argument deux fichiers fichier1 et fichier2 et qui copie le contenu de fichier1 dans fichier2 à l'envers : la dernière ligne devient la première ligne, l'avant-dernière ligne devient la seconde ligne , ... Les lignes, elles, ne sont pas inversées.
6. Ecrire un script Python qui compare le contenu de deux fichiers texte.
7. Ecrire un script Python qui prend en argument un fichier texte et un caractère c et qui crée un fichier contenant le contenu de fichier dans lequel tous les espaces sont remplacés par le caractère c.

   On pourra utiliser .join.
8. Ecrire un script Python qui prend en argument un fichier csv contenant un répertoire (i.e. dont les colonnes sont nom, code postal, ville) et qui renvoie la liste des codes postaux présents dans ce répertoire. D'abord, avec répétition, puis sans répétition. On pourra utiliser split.
9. Le but de cet exercice est de créer un moteur de QCM avec correction automatique des réponses de l'utilisateur.

Le fichier QCM est au format csv ; il est ainsi structuré :

la question; une réponse possible; une réponse possible; … ; le numéro de la bonne réponse

Exemple de QCM :

- Q1. Quelle est la couleur du cheval blanc de Tarik Bno ziad?:
  - 1 : Bleu;
  - 2 : Blanc ;
  - 3 : Rouge;
- Q2. Comment s'appelle le président syrien ?
  - 1 :Bachar Al Asad;
  - 2:Farouq Char3;
  - 3: A.Khadam;
- Q3. Combien de membres a un unijambiste ?: 1; 2; 3; 4;
- Ecrire un programme qui lit un QCM et qui présente, de manière aléatoire, les questions qui s'y trouvent, lit les réponses de l'utilisateur, les analysent, donne le score réalisé à la fin ainsi que les bonnes réponses aux différentes questions.

Analyse Numérique en Python

1. *Calcul de l'intégral d'une fonction continue*

On commence par définir les fonctions à intégrer :

```
def f1(x):
    return x**2

def f2(x):
    return np.exp(x)

def f3(x,y):
    return np.cos(x)*np.sin(y)
```

Méthode de rectangle droite :

```
def rectangle_droite(f,a,b,n):
    dx=(b-a)/n
    I=0
    x=a
    for i in range(0,n):
        I+=dx*f(x)
        x+=dx
    return I
```

Méthode de rectangle gauche

```
def rectangle_gauche(f,a,b,n):
    dx=(b-a)/n
    I=0
    x=a
    for i in range(0,n):
        x+=dx
        I+=dx*f(x)
    return I
```

Méthode de rectangle milieu

```
def rectangle_milieu(f,a,b,n):
    dx=(b-a)/n
    I=0
    x=a
    for i in range(0,n):

        I+=dx*f(x+dx/2)
        x+=dx
    return I
```

Méthode de trapèze

```
def trapeze(f,a,b,n):
    dx=(b-a)/n

    I=0
    x=a
    for i in range(0,n):

        I+=dx*(f(x)+f(x+dx))/2
        x+=dx
    return I
```

```
def sympson(f,a,b):
    return (b-a)/6*(f(a)+4*f((a+b)/2)+f(b))
```

Appel des fonctions d'intégrale pour la fonction f1

```
I1=si.quad(f1,0,4)
I2=rectangle_droite(f1,0,4,100)
I3=rectangle_gauche(f1,0,4,100)
I4=rectangle_milieu(f1,0,4,100)
I5=trapeze(f1,0,4,100)
I6=sympson(f1,0,4)
```

```
print("I(quad)=",I1)
print("I(droite) = ",I2)
print("I(gauche) = ",I3)
print("I(milieu) = ",I4)
print("I(trapeze) = ",I5)
print("I(sympson) = ",I6)
```

Exemple de résultats pour la fonction f1

```
>>> (executing lines 1 to 68 of "integrale.py")
I(quad)= (21.333333333333336, 2.368475785867001e-13)
I(droite) =  21.014400000000023
I(gauche) =  21.654400000000024
I(milieu) =  21.332800000000027
I(trapeze) =  21.334400000000024
I(sympson) =  21.333333333333332
>>>
```

2. Résolution d'Equations et dérivée

Nous étudions dans ces exercices plusieurs méthodes de calcul approché de solutions d'une équation du type f (x) = 0, f étant une fonction continue d'un intervalle I de R dans R.

Exercice 1 – Toutes les méthodes implémentées devront être vérifiées à l'aide des fonctions $f1: x \to x^2 - 2$, $f2: x \to x - e^{-x}$, $f3: \to \sqrt{x} - \frac{1}{2}$ et une marge d'erreur de 10^{-10}

On comparera également aux valeurs théoriques, lorsqu'elles sont calculables, et à la valeur obtenue à l'aide de la fonction newton du module **scipy.optimize**. La fonction newton prend en argument une fonction f et un réel x0 initialisant la méthode, ainsi, qu'un troisième argument optionnel, devant être égal à la fonction dérivée f' . Si cet argument optionnel f' est donné, un zéro de f est calculé à l'aide de la méthode de Newton, sinon à l'aide de la méthode des sécantes.

2.1 Dichotomie

Écrire une fonction dichotomie prenant en argument une fonction f , deux réels a et b et une marge d'erreur ε, et retournant un couple (x, n), tel que x ∈ [a, b] et f (x) = 0, n étant le nombre de partages de l'intervalle [a, b] qu'il a été nécessaire d'effectuer par la méthode de dichotomie. On commencera par un test permettant de s'assurer de l'existence d'une telle valeur de x.

2.2 *Méthode* de la fausse position

2.2.1 Même question avec une fonction **fausse_position** exploitant la méthode de la fausse position. On s'arrêtera lorsqu'on se sera assuré de l'existence d'un zéro ε-proche de la valeur obtenue.

2.2.2 Comparer les performances de fausse_position et de dichotomie sur les fonctions f1, f2, ainsi que sur la fonction $f4: x \to x^3$. Pour cette dernière fonction, considérer l'intervalle initial $[\frac{-1}{2}, 1]$, et la marge d'erreur ε = 10^{-4}.

2.3 *Méthode* de la sécante

2.3.1 Écrire une fonction secante prenant en paramètre une fonction f , deux réels a et b et une marge d'erreur ε, et retournant, en cas de succès, le couple (x, n) obtenu par la méthode de la sécante, x étant une valeur approchée près d'un zéro de f , n étant le nombre d'itérations nécessaires pour obtenir x, l'arrêt s'effectuant dès lors que deux valeurs successives sont ε-proches l'une de l'autre. On limitera le nombre d'itérations à 100. Si cette valeur est atteinte, on décrètera l'échec de la méthode.

2.3.2 En comparant aux valeurs théoriques et aux valeurs fournies par la fonction newton, discuter, de la validité du test d'arrêt.

2.3.3 Comparer les performances de cet algorithme et des précédents

2.4 Méthode de Newton-Raphson

2.4.1 Écrire une fonction mon_newton prenant en paramètre une fonction f , sa déridée f' , un réel a et une marge d'erreur ε, et retournant, en cas de succès, le couple (x, n) obtenu de même que précédemment, mais cette fois par la méthode de Newton-Raphson.

2.4.2 En comparant aux valeurs théoriques et aux valeurs fournies par la fonction newton, discuter, de la validité du test d'arrêt.

2.4.3 Comparer les performances de cet algorithme et des précédents.

2.4.4 Trouver une fonction f admettant un zéro et un réel x0 , telle que la méthode de Newton associée soit bien définie, mais divergente.

3. Dérivation numérique

Afin de pouvoir exploiter la méthode de Newton-Raphson sans avoir à donner l'expression de la dérivée de f , nous cherchons ici à calculer numériquement la dérivée de f . Cette méthode n'est évidemment valable que si f est à variations régulières, dans un sens que nous ne précisons pas.

3.1 Écrire une fonction derivee prenant en paramètre une fonction f et deux réels x et h, et retournant une valeur approchée de f' (x), obtenue en considérant le taux d'accroissement entre x − h et x + h.

3.2 Écrire une fonction trace_derivee prenant en paramètre une fonction f et deux réels a et b, et traçant le graphe de f' sur l'intervalle [a, b] (on consultera les TP précédents pour le tracé de graphes)

3.3 Écrire une fonction choix_de_h, prenant en paramètre une fonction f , sa dérivée théorique f' et un réel x, et traçant le graphe de la fonction qui à y associe l'erreur faite sur le calcul de f' par la méthode précédente en utilisant une valeur de h = 10−y. On fera varier y entre 4 et 8. Confirmer le choix optimal de h obtenu par le calcul dans le cours.

3.4 Reprogrammer une nouvelle fonction mon_newton_bis, exploitant la méthode de Newton associée à la dérivation numérique.

Solution : **Résolution d'Equations et dérivée**

Commençons par définir les trois fonctions qui vont nous servir à illustrer des méthodes :

```
from math import sqrt, exp
from scipy.optimize import newton

def f1(x):
    return x*x-2
def f2(x):
    return x-exp(-x)
def f3(x):
    return sqrt(x)-.5
```

1. Dichotomie

Lorsque f (a)f (b) > 0, on choisit de déclencher l'exception ValueError accompagnée d'un message en précisant la raison. Dans le cas contraire, il s'agit d'itérer deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ vérifiant les propriétés suivantes :

(i) $(a_n)_{n\in\mathbb{N}}$ $est\ croissante\ et$ $(b_n)_{n\in\mathbb{N}}$ est décroissante
(ii) $\forall\, n \in \mathbb{N}, f(a_n) * f(b_n) \leq 0$
(iii) $\lim_{n\to\infty}(a_n - a_n) = 0$

Ceci conduit à la définition suivante :

```
def dichotomie(f, a, b, tol = 1e-10):
    if f(a)*f(b) > 0:
        raise ValueError("f(a) et f(b) doivent avoir un signe différent")
    n = 0
    while abs(b-a) > 2*tol:
        n += 1
        c = (a+b)/2
        if f(c-tol) * f(c+tol) <= 0:
            return c, n
        if f(a)*f(c)>0:
            a = c
        else:
            b = c
    return (a+b)/2, n
```

Testons cet algorithme sur les trois fonctions proposées en exemple, en comparant aux valeurs théoriques :

```
>>> dichotomie(f1,1, 2)
(1.4142135623842478, 29)
>>> sqrt(2)
1.4142135623730951

>>> dichotomie(f2,0, 1)
(0.5671432904200628, 33)
>>> newton(f2, 0, fprime = lambda x:1+exp(-x), tol = 1e-10)
0.567143290409784

>>> dichotomie(f3,0, 1, 1e-10)
(0.25, 2)
>>> 1/4
0.25
```

2. Méthode de la fausse position

Très semblable à la méthode dichotomique, la méthode de la fausse position consiste à considérer non pas le point milieu du segment [a,b] mais l'intersection de l'axe des abscisses avec la corde reliant les points de coordonnées (a,f (a)) et (b , f (b)). Les conditions (i) et (ii) restent vérifiées, mais en revanche nous n'avons plus la garantie que : $\lim_{n\to\infty}(a_n - a_n) = 0$

2.1 Ceci conduit à la définition suivante :

```
def fausse_position(f, a, b, tol = 1e-10):
    if f(a)*f(b) > 0:
        raise ValueError("f(a) et f(b) doivent avoir un signe différent")
    n = 0
    while abs(b-a) > 2*tol:
        n += 1
        c = (a*f(b)-b*f(a))/(f(b)-f(a))
        if f(c-tol) * f(c+tol) <= 0:
            return c, n
        if f(a)*f(c)>0:
            a = c
        else:
            b = c
    return (a+b)/2, n
```

Essayons cette fonction avec les fonctions f_1 et f_2 :

```
>>> fausse_position(f1, 1, 2)
(1.4142135623189167, 13)
>>> fausse_position(f2, 0, 1)
(0.5671432904228786, 11)
```

Ces deux exemples montrent une amélioration de la vitesse de convergence comparativement à la méthode dichotomique. Mais cette méthode peut aussi être beaucoup plus lente, comme le montre l'exemple de la fonction : x :→ x^3

```
>>> fausse_position(lambda x: x**3, -.5, 1, 1e-4)
(-9.999999920641366e-05, 49989996)
```

3. Méthode de la sécante

La méthode de la sécante consiste à appliquer la méthode de la fausse position entre les deux derniers points calculés, autrement dit à itérer une suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée de deux valeurs initiales u_0= a et u_1= b et la relation : $u_{n+2} = \frac{u_n f(u_{n+1}) - u_{n+1} f(u_n)}{f(u_{n+1}) - f(u_n)}$.

La grande vitesse de convergence de cette suite (quand convergence il y a) conduit à choisir comme condition d'arrêt :$|u_{n-1} - u_n| \leq \epsilon$, même si mathématiquement cette condition n'assure pas que un soit proche de sa limite (ni même que la suite converge). En outre, en cas de divergence cette condition peut ne jamais être réalisée, aussi a-t-on pour habitude de majorer le nombre d'itérations autorisées.

3.1 Ceci conduit à la définition :

```
def secante(f, a, b, tol = 1e-10, nmax = 100):
    n = 0
    while abs(b-a) > tol:
        n += 1
        if n >= nmax:
            raise RuntimeError('capacité dépassée')
        a, b = b, (a*f(b)-b*f(a))/(f(b)-f(a))
    return b, n
```

3.2 Les exemples des fonctions données en préambule montrent que cette fonction, quand elle converge, est notablement plus efficace que les précédentes, et que la condition d'arrêt est (pour ces exemples en tout cas) pertinente.

3.3 comparaison

```
>>> secante(f1, 1, 2)
(1.414213562373095, 7)
>>> sqrt(2)
1.4142135623730951
```

```
>>> secante(f2, 0, 1)
(0.5671432904097838, 6)
>>> newton(f2, 0, fprime = lambda x:1+exp(-x), tol = 1e-10)
0.567143290409784

>>> secante(f3, 0, 1)
(0.25000000000000006, 8)
>>> 1/4
0.25
```

4. Méthode de Newton-Raphson

Cette méthode consiste enfin à remplacer la sécante par la tangente calculée au point précédent, autrement dit à itérer la suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée d'une valeur initiale u0= a et la relation :

4.1 Ceci conduit à la définition :

```
def mon_newton(f, fprime, a, tol = 1e-10, nmax = 100):
    n = 0
    b = a-f(a)/fprime(a)
    while abs(b-a) > tol:
        n += 1
        if n >= nmax:
            raise RuntimeError('échec de la convergence')
        a, b = b, b-f(b)/fprime(b)
    return b, n
```

4.2 Ceci conduit à de très bons résultats pour nos deux premières fonctions :

```
>>> mon_newton(f1, lambda x:2*x, 2)
(1.4142135623730951, 4)

>>> mon_newton(f2, lambda x:1+exp(-x), 1)
(0.5671432904097838, 4)
```

Pour la fonction f3, il faut partir d'une valeur initiale a]0; 1/ 4[, faute de quoi la suite $(u_n)_{n\in\mathbb{N}}$

n'est pas définie :

```
>>> mon_newton(f3, lambda x:1/(2*sqrt(x)), 2)
ValueError: math domain error

>>> mon_newton(f3, lambda x:1/(2*sqrt(x)), .1)
(0.25, 4)
```

5. Dérivation numérique

5.1 Pour calculer la dérivée numérique d'une fonction f , on définit simplement :

```
def derivee(f, x, h):
    return (f(x+h)-f(x-h))/2/h
```

5.2 On peut ensuite tracer le graphe de la dérivée numérique d'une fonction, par exemple à l'aide de la fonction suivante :

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

def trace_derivee(f, a, b, nbpoints = 100):
    ax = np.linspace(a, b, nbpoints)
    ay = [derivee(f, t, 1e-5) for t in ax]
    plt.plot(ax, ay)
```

La fonction **linspace** du module numpy permet d'obtenir une subdivision en en précisant le nombre de points.

5.3 Réalisons enfin la comparaison pratique entre les dérivées théorique et numérique en fonction de h à l'aide de la fonction :

```
def choix_de_h(f, fprime, x):
    ax = np.linspace(4, 8, 100)
    ay = [abs(fprime(x)-derivee(f, x, 10**(-y))) for y in ax]
    plt.plot(ax, ay)
```

Version avec dérivée numérique de la méthode de Newton :

```
def mon_newton_bis(f, a, tol = 1e-10, nmax = 100):
    return mon_newton(f, lambda x: derivee(f, x, 1e-5), a, tol, nmax)
```

4. Simulation numériques des équations différentielles sous Python

4.1 Introduction :

Déterminer la solution analytique d'une équation différentielle est à tout point de vue ce qu'il y a de plus intéressant. Toutefois il ne faut pas pour autant perdre de vue : en dehors des équations différentielles linéaires à coefficients constants ou d'ordre peu élevé, on est en général bien incapable de résoudre analytiquement une équation différentielle, a fortiori lorsqu'elle est non linéaire.

On doit alors faire appel à des méthodes de résolution numérique, pour calculer une solution approchée au moyen d'ordinateurs. Ces méthodes de résolution font appel à des schémas numériques, qui sont des algorithmes de calcul permettant de résoudre numériquement une équation différentielle.

Il existe différentes méthodes itératives pour résoudre ce type d'´equation différentielle. Celles-ci consistent à calculer une suite de points (x_n, y_n) à partir d'un point initial (x_0, y_0) et telle que $y(x_n)=y_n$ et $x_{n+1} = x_n+h$ avec h fixé (le pas). Cependant, on distingue quatre méthodes différentes telles que :

- La méthode d'Euler est la plus simple des méthodes. Elle consiste à approximer localement la fonction par la tangente en un point. La suite de point se calcule alors aisément par la relation $y_{n+1} = h*(f(y_n , x_n))+ y_n$.
- La méthode du point milieu est similaire à la méthode d'Euler par son principe. En effet, celle-ci consiste à calculer un point médian entre y_n et y_{n+1} par la méthode précédente. Une fois ce calcul fait, on utilise une parallèle à la tangente en ce point milieu passant par y_n pour définir y_{n+1}. Cette méthode permet de mieux prendre en compte les variations de la fonction entre y_n et y_{n+1}.
- La méthode de Heun utilise la moyenne des pentes au point initial et au point qui aurait été obtenu par la méthode d'Euler pour définir la position du point suivant.
- Enfin la méthode de Runge-Kutta, comme les méthodes précédentes, considère la moyenne de quatre points situ´es entre y_n et y_{n+1}.

  Dans ce cours nous limiterons aux deux méthodes dites d'Euler et de Runge Kutta. (dont la première fait partie du programme d'informatique en classe CPGE)

4.2 Forme de cauchy

La forme de Cauchy est un système de n équations différentielles du premier ordre qui est de la forme :

$$\begin{cases} y'_1(x) = f_1(x, y_1(t)) \\ \vdots \\ \vdots \\ \vdots \\ y'_n(x) = f_n(x, y_n(t)) \end{cases}$$

où x est une variable indépendante, généralement le temps ou la distance, où **y(t)** est un vecteur à **n** éléments et où la dérivée par rapport à la variable indépendante est indiquée par le symbole prime comme dans **y'.** Avec les notations vectorielles, ce système s'écrit : **y'(x) = f(x , y)** La fonction **f(x , y)** est généralement contenue dans une unité différente, telle qu'une fonction en python.

4.3 Résolution numérique des équations différentielles du premier ordre :

a. Principe générale :

Soit è résoudre une équation différentielle de la forme : $\frac{dy(x)}{dx} + ay(x) = b$

Sachant que :

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}$$

Alors l'équation différentielle s'écrit :

$\frac{y(x+h)-y(x)}{h} + ay(t) = b$

$y(x+h) - y(x) = (-ay(x) + b) * (h)$

$$\Longrightarrow y(x+h) = y(x) + \big(b - ay(x)\big) * h$$

$On\ note\ f(y,x) = b - ay(x)$ alors : $\boldsymbol{y(x+h) = y(x) + f(y(x),x) * h}$

b. *Notion de Pas explicite et/ou implicite*

Soit le pas $h = x_{n+1} - x_n$, ou généralement $h = \frac{x_f - x_i}{N}$, N étant le nombre de points à résoudre pour y sur l'intervalle $[x_i, \dots , \dots\dots x_f]$.

i. méthodes à un pas explicite

La formulation générale des méthodes à un pas explicite est :

$$\begin{cases} \text{pour} \quad y_0 \text{ donnée} \\ y_{k+1} = y_k + f(x_k, y_k, h) \end{cases}$$

Où la fonction f définit la méthode utilisée.

ii. méthodes à un pas implicite

La formulation générale des méthodes à un pas implicite est :

$$\begin{cases} \text{pour} \quad y_0 \quad \text{donnée} \\ y_{k+1} = y_k + f(x_k, y_k, y_{k+1}, h) \end{cases}$$

L'obtention de la solution à chaque abscisse nécessite la résolution d'une équation. Ces méthodes sont obtenues en intégrant l'équation différentielle et en utilisant des formules d'intégration numérique pour le second membre. L'ordre du schéma est égal au degré du polynôme pour lequel l'intégration est exacte.

Résultats théoriques :

- Si la fonction f est lipchitzienne par rapport à la deuxième variable alors les méthodes explicites sont stables
- Les méthodes implicites sont toujours stables.
- Les méthodes sont consistantes si : $\forall\, x \in [a,b]; f(y,x,0) = f(y,x)$

c. *Méthodes d'Euler explicite:*

En parlant des solutions numériques aux Equations différentielles ordiniaires, chacune commence par la méthode d'Euler, puisqu'elle est facile à comprendre et simple pour programmer.

Exemple :

Considérons une équation de premier ordre : $y'(x) + ay(x) = b \quad avec\ y(0) = y_0 = 0$

Sa solution analytique est de la forme : $y(x) = \left(y_0 - \frac{b}{a}\right) e^{-a1t} + \frac{b}{a}$ soit $y(x) = \frac{b}{a}(1 - e^{-ax})$

Ce qui peut être obtenu par l'utilisation d'une méthode conventionnelle ou la technique de transformé de Laplace. Cependant, une solution analytique exacte n'existe pas pour chaque équation ; même si elle existe, il n'est pas facile de trouver même en utilisant un ordinateur équipé des possibilités du calcul symbolique. C'est pourquoi nous devrions étudier les solutions numériques aux équations différentielles. Comment traduisons-nous l'équation en forme qui peut facilement être manipulée par ordinateur?

Tout d'abord, nous devons remplacer la dérivée : $y'(x) = \frac{dy(x)}{dx}$ dans l'équation par un dérivé numérique $\frac{dy}{dx} = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}$, où le pas « h » est déterminé sur les conditions d'exactitude et les contraintes de calcul.

Euler a été le premier qui a proposé un schéma d'approximation. Ce schéma consiste à approcher l'intégrale :

$$\int_{x_k}^{x_{k+1}} f(y(x), x)dx \text{ par } \emptyset(y_k, x_k, h) = f(y_k, x_k).$$

Le schéma dit d'Euler explicite s'écrit alors :

D'où l'équation de récurrence :

$$y[k+1] = y[k] + f(y[k], x[k]) * h$$

i. *Principe de la méthode d'Euler Explicite :*

On part d'un point M_0 de coordonnées (x_0, y_0) appartenant à la solution approchée. Le point suivant est le point M_1 de coordonnées (y_1, x_1), tel que $x_1 = x_0 + h$ et tel que le point M_1 est sur la droite passant par M_0 de pente $f(\ y_0, x_0)$, On itère le procédé en partant du point $M(y_0, x_0)$ et en reprenant à chaque étape le point M_i comme nouveau point M_1, autant de fois qu'il le faut pour que x décrive l'intervalle demandé. Voir la figure suivante :

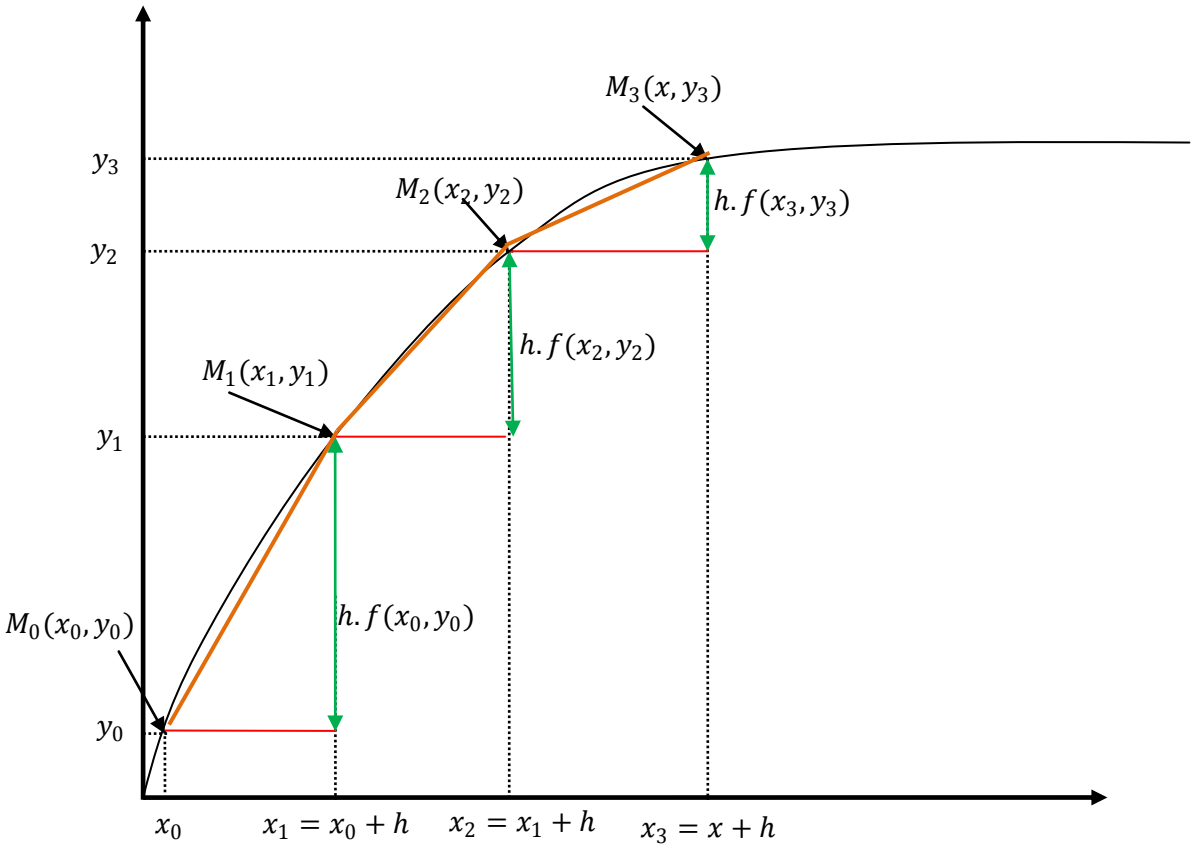


Figure 1 : principe de la méthode d'Euler.

Quoique sa basse exactitude la garde d'être employée couramment pour résoudre des odes, elle nous donne un indice au concept de base de la solution numérique pour une équation simplement et clairement.

ii. *Implémentation en python:*

On commence par charger les bibliothèques nécessaires :

```
import numpy as np

import  matplotlib.pyplot as plt
```

Puis on définit la fonction f(y,x)

```
def derivee(y,x ) :
    # résolution d'une equation de la forme y'+ay=b
    global a , b
    return   b-a*y
```

Puis on définit la fonction d'Euler explicite :

```
def  EulerExplicite(fd , y0, xi, xf, N) :
      #fd: fonction d'integration
      #y0: condition initiale
      #xi: début de simulation : (temps initial)
      #xf: fin de simulation : (temps final)
      #N : nombre de pas
      #Lx: vecteur de temps (axe des abscisses)
      #Ly: vecteur de solutions (axe des ordonnées)
      Ly = np.zeros(N)
      Ly[0] =y0   #on met la première valeur dans le vecteur Ly
      Lx = np.linspace(xi,xf,N)
      h = (xf-xi) / N   # la pas d'intégration
      for k in range(N-1):
            Ly[k+1] = Ly[k] + h* fd(Ly[k], Lx[k])
      return Lx, Ly
```

Exemple d'appel : fichier ***test1.py***

```
# résolution d'une equation de la forme y'+ay=b
# soit une solution analytique de la forme y(x)=k.exp(-a*x)+b/a avec y(0)=0
# donc k=-b/a ==> y(x)=b/a*(1-exp(-a*x))
a  , b  ,   n =2 , 10   , 50
x,sol=EulerExplicite(derivee, 0,0,5,n)
plt.plot(x,sol, label="euler n="+str(N))
plt.title(' Résolution d equation y'+a.y=b avec a=2,b=10')
plt.xlabel('x')
plt.ylabel('y(t)')
plt.grid()
plt.show()
```

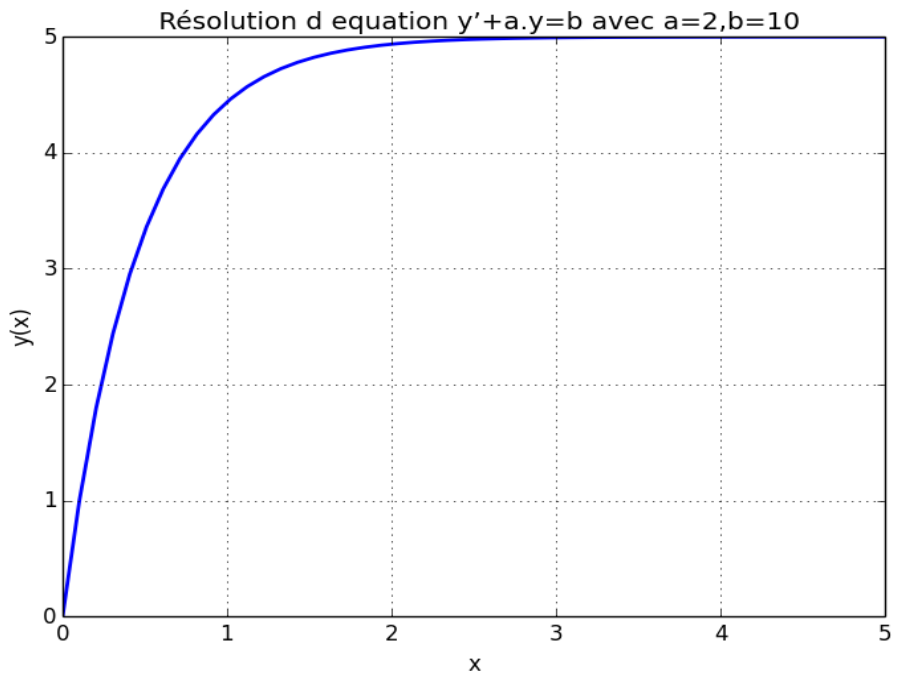


Figure2: Solution de l'équation différentielle de la forme y'+2y=10 avec y(0)=0

Effet du nombre de pas (n) sur le résultat obtenu : ***test2.py***

```
# résolution d'une equation de la forme y'+ay=b
# soit une solution analytique de la forme y(x)=kexp(-at)+b/a avec y(0)=0
# donc k=-b/a ==> y(x)=b/a*(1-exp(-a*x))
a  , b  =2,  10
n=20
x,sol=EulerExplicite(derivee, 0,0,5,n)
plt.plot(x,sol,Label='Euler n='+str(n))
n=100
x1,sol1=EulerExplicite(derivee, 0,0,5,n)
plt.plot(x1,sol1,Label='Euler n='+str(n))
n=1000
x2,sol2 =EulerExplicite(derivee, 0,0,5,n)
plt.plot(x2,sol2,Label='Euler n='+str(n))
n=10000
x3,sol3 =EulerExplicite(derivee, 0,0,5,n)
plt.plot(x3,sol3,Label='Euler n='+str(n))
plt.legend()
plt.title(' Résolution de l equation y'+a.y=b avec a=2,b=10')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.grid ()
plt.show()
```

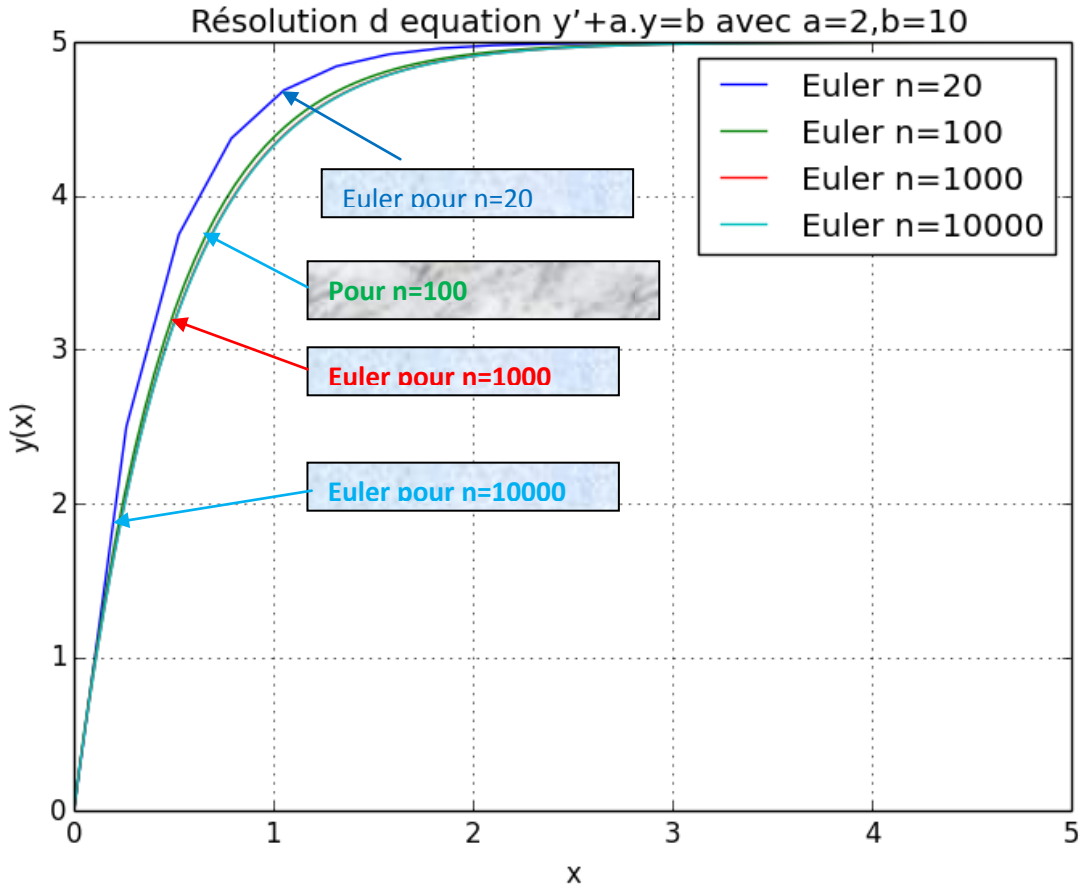


Figure3: Solution de l'équation différentielle de la forme y'+2y=10 : effet du nombre de pas

a. *Solution d'Euler pour n=20*
b. *Solution d'Euler pour n=100*
c. *Solution d'Euler pour n=1000*
d. *Solution d'Euler pour n=10000*

En comparant avec la solution exacte y(x)=b/a(1-exp(-ax))* : ***test2.py***

```
plt.figure(2)
n=20
x1,sol1=EulerExplicite(derivee, 0,0,5,n)
solA1=b/a*(1-np.exp(-a*x1)) # solution exacte
plt.plot(x1,sol1,label='Euler n='+str(n))
plt.plot(x1,solA1,Label='Sol. Analytique pour n= '+str(n))
n=100
x1,sol3=EulerExplicite(derivee, 0,0,5,n)
sol4=b/a*(1-np.exp(-a*x1)) # solution exacte
plt.plot(x1,sol3,Label='Euler n='+str(n))
plt.plot(x1,sol4,Label='Solution Exacte pour n= '+str(n))
n=10000
x2,sol5=EulerExplicite(derivee, 0,0,5,n)
sol6=b/a*(1-np.exp(-a*x2)) # solution exacte
plt.plot(x2,sol5,Label='Euler n='+str(n))
plt.plot(x2,sol6,Label='Solution Exacte pour n= '+str(n))
plt.title('' Résolution d equation y'+a.y=b avec a=2, b=10'')
plt.xlabel('x)
plt.ylabel('y(x)')
plt.grid()
plt.legend()    plt.show()
```

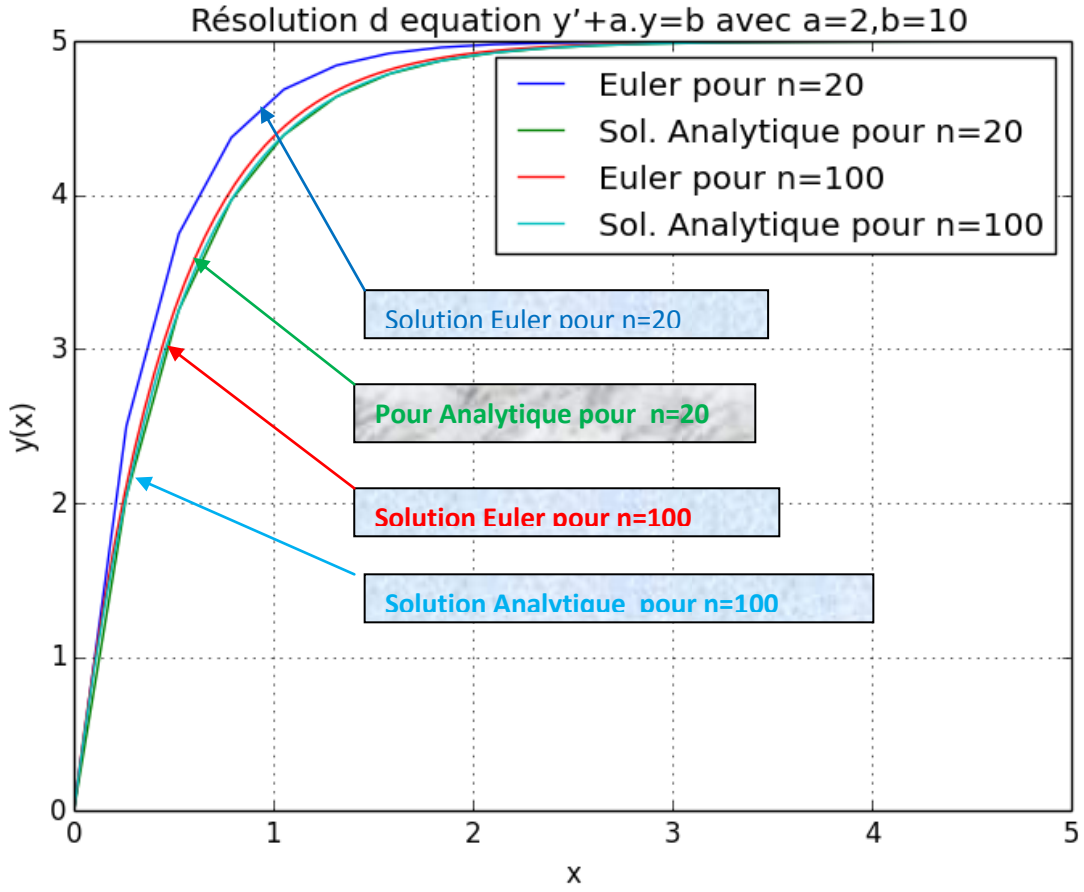


Figure4: Solution de l'équation différentielle de la forme y'+2y=10 : effet du nombre de pas

a. *Solution d'Euler pour n=20*
b. *Solution analytique pour n=20*
c. *Solution d'Euler pour n=100*
d. *Solution analytique pour n=100*

d. Méthode de Runge Kutta :

i. Principe de la méthode (Runge Kutta):

Runge et Kutta ont développé une des méthodes de résolution numérique pour les équations différentielles les plus utilisées : **la méthode de Runge-Kutta.**

Il existe plusieurs façons d'appliquer cette méthode que l'on différencie grâce à « l'ordre d'application ». On obtient ainsi (de la méthode la moins précise à la plus précise) :

- Runge-Kutta d'ordre 1 (RK-1),

- Runge-Kutta d'ordre 2 (RK-2),

- Runge-Kutta d'ordre 3 (RK-3),

- Runge-Kutta d'ordre 4 (RK-4),

- Etc …

→ ***Dans ce cours on se limitera au cas de Runge-Kutta d'ordre 2 et 4 (RK-2 & RK-4)***

Principe de la méthode : Evaluer la dérivée à mi-chemin !

La méthode de Runge- Kutta d'ordre deux est la plus simple; elle combine deux itérations successives de la méthode d'Euler explicite. Dans un premier temps la dérivée en (t_k, y_k) est évaluée pour faire une première estimation du point. L'estimation provisoire de y_{k+1} en ce point est ensuite utilisée pour affiner le calcul de la dérivée. Une nouvelle approximation de celle-ci est obtenue en prenant sa valeur à mi- parcours. C'est cette valeur de la dérivée qui sera ensuite utilisée pour estimer le prochain pas t_{k+1}. La procédure d'intégration se résume ainsi :

$$\begin{cases} k_1 = f(t_k, y_k) \\ k_2 = f\left(t_k + \dfrac{h}{2}, y_k + \dfrac{k_1}{2}\right) \\ y_{k+1} = y_k + \dfrac{h}{3} * (k_1 + 2 * k_2) \end{cases}$$

ii. Implémentation sur Python :

➔ **Runge Kutta d'ordre 2**

```
def  RK2(derivee,y0,xi,xf,N)
      #fct: fonction d'intégration
      #y0: condition initiale
      #xi: début d'intervalle de résolution
      #xf: fin de l'intervalle de résolution
      #N: nombre de pas d'intégration
      Ly =np.zeros(N)
      Lx=np.linspace(xi,xf,N)
      h = (xf-xi) / N # le pas d'intégration
      Ly[0]=y0  #on met la première valeur (condition initiale)
      for i in range(N-1):
            k1=h* derive(Ly[i], Lx[i);
            k2=h* derive(Ly[i]+k1/2, Lx[i]+h/2);
            Ly[i+1]=Ly[i]+k2;
      return  Lx, Ly
```

Exemple d'appel : fichier test3.py

```
# résolution d'une equation de la forme y'+a1y=a2
# soit une solution analytique de la forme y(x)=kexp(-a1.x)+a2/a1 avec
#y(0)=0 % donc k=-a2/a1 ==> y(x)=a2/a1*(1-exp(-a1.x))
a1  ,  a2,  n  =2,   10   , 50
x,y =RK2(derivee,1,0,5,n)
plt.plot(x,y, Label ='RK 2 pour n= '+str(n))
plt.grid()
plt.title(' Résolution de l equation y'+a1y=a2 avec a1=2, a2=10')
plt.xlabel(' x ')
plt.ylabel('y(x)')
plt.legend()
plt.show()
```

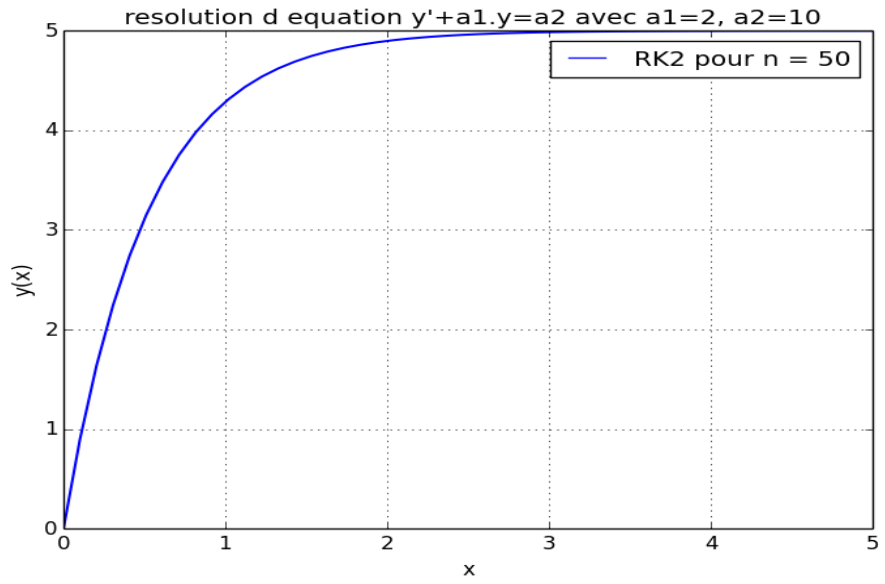


Figure 5 : solution de l'équation différentielle de la forme y'+ay=b (méthode runge kutta 2)

Effet du nombre de pas n : ***test4.py***

```
# résolution d'une equation de la forme y'+a1y=a2
# soit une solution analytique de la forme y(t)=k.exp(-a1.x)+a2/a1
a1  ,   a2  ,   n= 2,   10  ,     5
x1,sol1 =RK2(derivee, 1,0,5,n);
plot(x1,sol1,label='RK2 n='+str(n))
n=100;
x2,sol2 =RK2(derivee, 1,0,5,n);
plot(x2,sol2,label='RK2 n='+str(n))
n=1000
x3,sol3 =RK2(derivee, 1,0,5,n);
plot(x3,sol3,label='RK2 n='+str(n))
n=1000
x4, sol4=RK2(derivee, 1,0,5,n);
plot(x4,sol4,label='RK2 n='+str(n))
plt.legend()
plt.grid()
plt.title(' Résolution d equation y'+a1y=a2 avec a1=2,a2=10')
plt.xlabel(' x ' )
plt.ylabel(' y(x) ' )
plt.legend( )
```

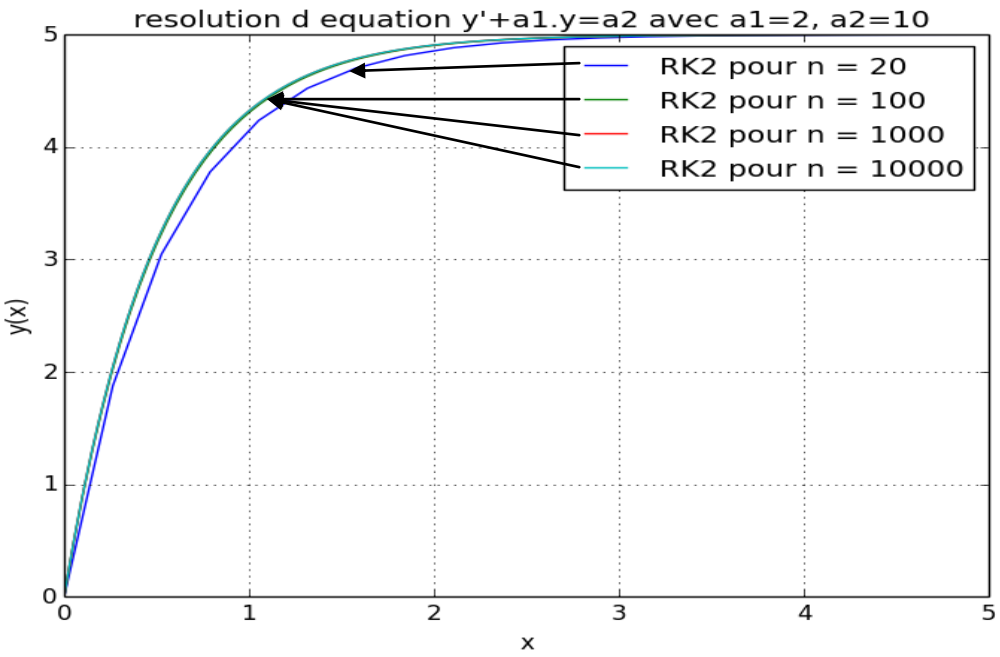


Figure 6: solution d'équation différentielle : y'+2y=10 (y(0)=0)

➔ Runge Kutta d'ordre 4 :

Dans le cas de RK-4, on évalue la dérivée en faisant la moyenne de 4 dérivées approximatives (K1, K2, K3 et K4), calculées chacune à mi-parcours.

Sachant que $\begin{cases} y' = f(y_k, x_k) \\ y(x_0) = 0 \end{cases}$, et que le pas est noté h, on a alors :

$$\begin{cases} \boldsymbol{k_1 = f(y_k, x_k)} \\ \boldsymbol{k_2 = f\left(y_k + \frac{h}{2} * k_1, x_k + \frac{h}{2}\right)} \\ \boldsymbol{k_3 = f\left(y_k + \frac{h}{2} * k_2, x_k + \frac{h}{2}\right)} \\ \boldsymbol{k_4 = f(y_k + h * k_3, x_k + h)} \end{cases}$$

On pose alors $\boldsymbol{y_{k+1} = y_k + \frac{h}{6} * (k_1 + 2 * k_2 + 2 * k_3 + k_4)}$

➔ Implémentation sur Python :

```
def RungeKutta4(derivee,y0,xi,xf,N):
      Lx =np.zeros(N)
      x=np.linspace(xi,xf,N)
      h = (xf-xi) / N ; % le pas d'intégration
      y[0]=y0  # on initiale le vecteur temps par xi
      for k in range(N-1):
            K1= h*derivee(Lx[k],Ly[k]);
            K2= h*derive (Ly[k]+K1/2,Lx[k]+h/2);
            K3= h*derivee (Ly[k]+K2/2,Lx[k]+h/2);
            K4= h*derivee (Ly[k]+K3,Lx[k]+h);
            Y[k+1]=y[k]+1/6*(K1+2*K2+2*K3+K4);
       return Lx, Ly
```

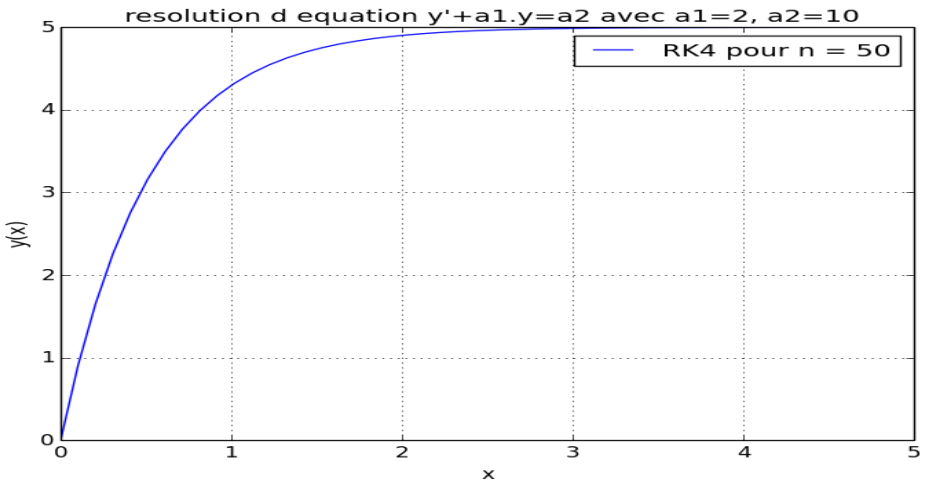
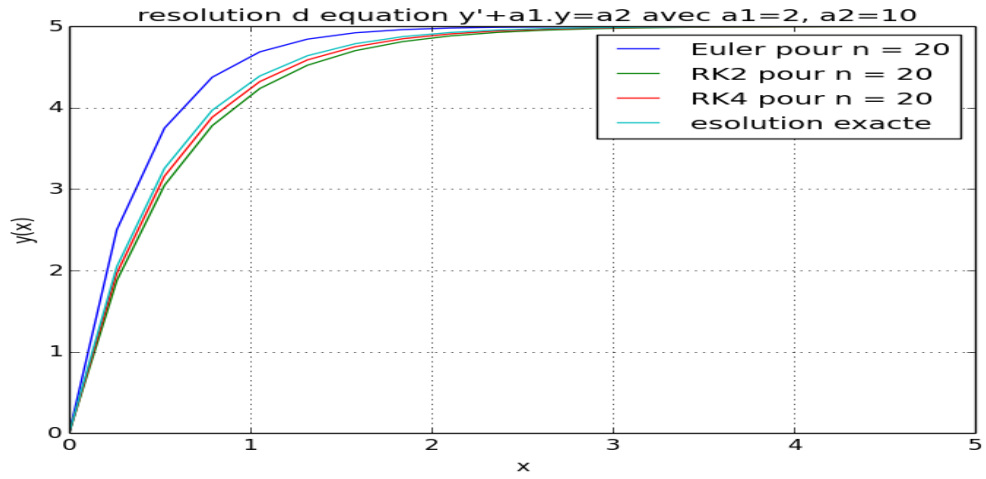


Figure 7 : solution de l'équation différentielle de la forme y'+2y=10 (méthode runge kutta 4)

Comparaison entre les différentes techniques avec celle de la solution analytique
Pour n=20



Pour n=50

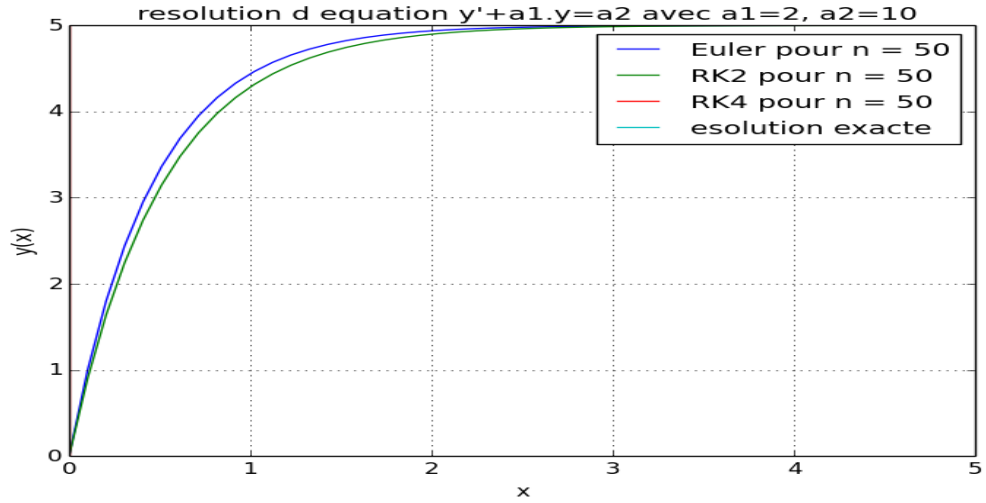


Figure 8 : Comparaison des méthodes pour différentes valeurs de n

j. *Résolution numérique des équations différentielles du second ordre :*

a. *Principe générale :*

Soit è résoudre une équation différentielle de la forme :

$$\frac{d^2y(x)}{dx^2} + a_1.\frac{dy(x)}{dx} + a_2.y(x) = a_3$$

On a alors $\frac{d^2y(x)}{dx^2} = -a_1\frac{dy(x)}{dx} - a_2y(x) + a_3$

Posons $\frac{dy(x)}{dx} = v(x)$ → l'équation devienne : $\frac{dv(x)}{dx} = -a_1\frac{dy(x)}{dx} - a_2y(x) + a_3$

Soit le nouveau système : $\begin{cases} \frac{dy(x)}{dx} = v(x) \\ \frac{dv(x)}{dx} = -a_1 * v(x) - a_2 * y(x) + a_3 \end{cases}$

Il s'agit bien d'un système à deux équations du 1er ordre.

Donc on choisit une variable $Y = \begin{bmatrix} y(x) \\ \frac{dy(x)}{dx} \end{bmatrix} = \begin{bmatrix} y(x) \\ v(x) \end{bmatrix}$

Donc $Y' = \frac{dY(x)}{dx} = \begin{bmatrix} \frac{dy(x)}{dx} \\ \frac{dv(x)}{dx} \end{bmatrix} = \begin{bmatrix} v(x) \\ a_3 - a_2 * y(x) - a_1 * \frac{dy(x)}{dx} \end{bmatrix}$

On définit une nouvelle fonction $F(x,Y) = \begin{bmatrix} v(x) \\ a_3 - a_2 * y(x) - a_1 * \frac{dy(x)}{dx} \end{bmatrix}$

Sachant que :

$$\frac{dY}{dx} = \lim_{h \to 0} \frac{Y(x+h) - Y(x)}{h}$$

Alors l'équation différentielle s'écrit : $Y(x+h) - Y(x) = F(Y(x),x) * h)$

$$\Longrightarrow Y(x+h) = Y(x) + F(Y(x),x) * h$$

b. Implémentation sur Python :

Définissons la fonction F :

```
def derivee2(Y, x ):
      global a1, a2, a3
      dy=Y[2]
      ddy=-a1*Y[2]-a2*Y[1]+a3
      dY=[dy, ddy]
      return dY
```

➢ *La fonction d'Euler Explicite est la suivante:*

```
def EulerExplicite2(F ,ci ,xi ,xf ,n) :
       h=(xf-xi)/n
       x=np.linspace(xi, xf, n)
       Y=np.zeros(n,2)
       Y[0,:]=ci
      for i in range (n-1):
           Y[i+1,:]=Y[i,:]+h*F(Y[i,:], x[i])
      return x,Y
```

➢ *Programme appelant :* ***test5.py***

```
a1,   a2,   a3= 1  , 2  ,  5
ci=[0,0]
n=100
x,sol =EulerExplicite2(derivee2,ci,0,20,n)
plt.plot(x, sol[:,0],label='y(x)')
plt.plot(x, sol[:,1], label='dy(x)/dx')
plt.grid()
plt.title('Résolution d equation differentielle d2y/dt2+a1.dy/dt+a2.y=a3
avec a1=1, a2=2 et a3=5')
plt.xlabel('temps en s')
plt.ylabel('y(t) ')
plt.legend();
```

➢ *Résultat :*

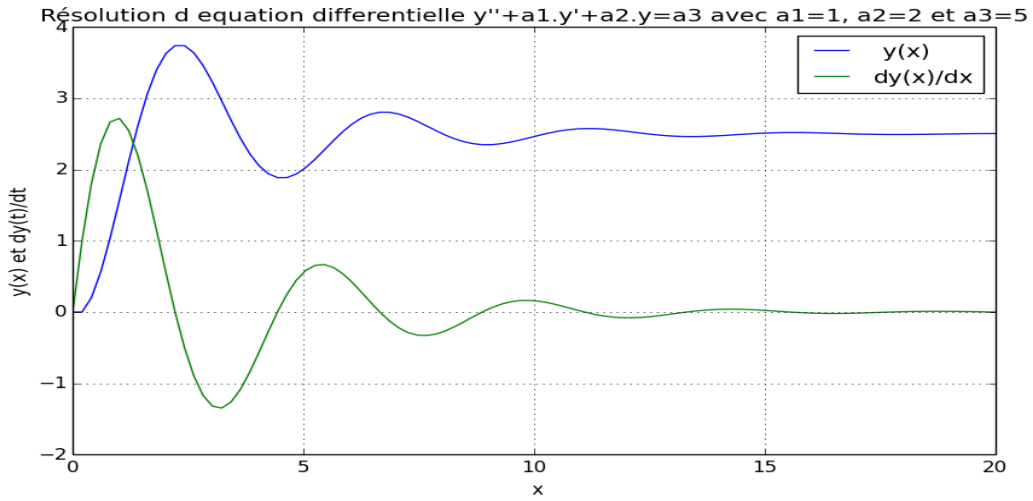


Figure9: Solution d'une équation différentielle de la forme y''+y'+2y=5 par la méthode d'Euler

➢ *La fonction de Runge Kutta d'ordre 4*

```
def  RK4(fd ,ci, xi, xf, n) :
      h=( xf-xi)/n;
      x=np.linspace(xi, xf , n)
      y=np.empty((n,np.size(ci)),dtype=float)
      y[0,:]=ci
      for i in range( n-1):
            K1=h* fd (sy[i,:], x[i)
            K2=h* fd (sy[i,:]+1/2*K1, x[i]+h/2)
            K3=h* fd (sy[i,:]+1/2*K2, x[i]+h/2)
            K4=h* fd (sy[i,:]+ K3, x[i]+h);
            y[i+1,:]=y[i,:]+1/6*(K1+2*K2+2*K3+K4)
       return x,y
```

➢ Programme appelant : ***test6.py***

```
a1=1
a2=2
a3=5
ci=np.array([0,0]) # conditions initiales
n=10
x , sol =RK4( derivee2,ci,0,20,n);
plt.plot(x , sol[ :,0] , label=' y(x) ')
plt.plot(x , sol[ :,1] , label='dy(x)/dx ')
plt.grid()
plt.title('Résolution d equation diff d2y/dx2+a1.dy/dx+a2.y=a3 avec a1=1,
a2=2 et a3=5')
plt.xlabel('x ')
plt.ylabel('y(x) & dy(x)/dx ')
plt.legend()
plt.show()
```

➢ Résultat : pour les conditions initiales ci=[0 , 0]

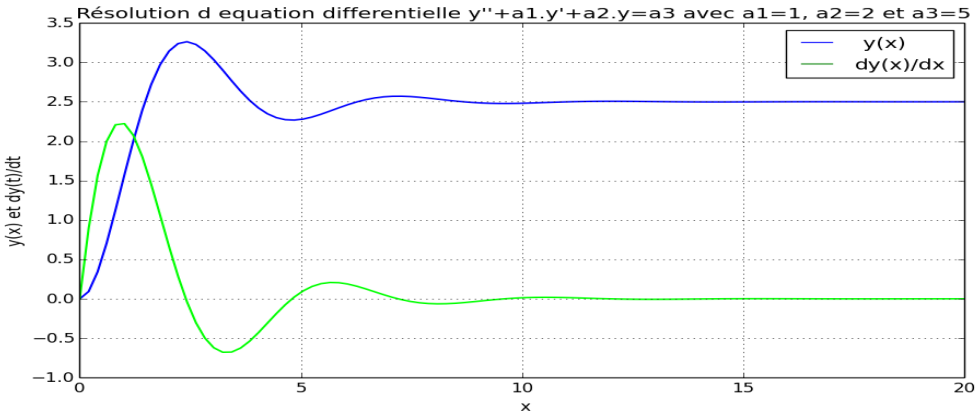


Figure 10 : Solution d'une équation différentielle de la forme y''+y'+2y=5 par la méthode de Runge-Kutta4

➢ ***Comparaison des différentes méthodes y comprise la méthode odeint sous python***

Programme principal : test6.py

```
def derivee2(Y, x  ): # on redéfinit la function derivee2
      global a1, a2, a3
      dy=Y[2]
      ddy=-a1*Y[2]-a2*Y[1]+a3
      dY=np.array([dy, ddy])
      return dY
a1=1,a2=2 a3=5 , n=100
ci=np.array([0,0])
x1,sol1 =EulerExplicite2(derivee2, ci ,0,20,n)
x2,sol2 =RK4( derivee2, ci ,0,20,n)
sol3 =odeint( f2,ci, x2)
plt.plot(x1,sol1[:,0], label='y(t): Euler')
plt.plot(x2,sol2[:,0], , label='y(t): RK4')
plt.plot(x2,sol3[:,0], , label='y(t): Odeint')
plt.title(' Résolution d equation diff d2y/dx2+a1dy/dx+a2.y=a3 avec
a1=1,a2=2, a3=5')
plt.legend()
plt.xlabel('x ')
plt.ylabel('y(x)')
plt.grid()
```

Résultats :

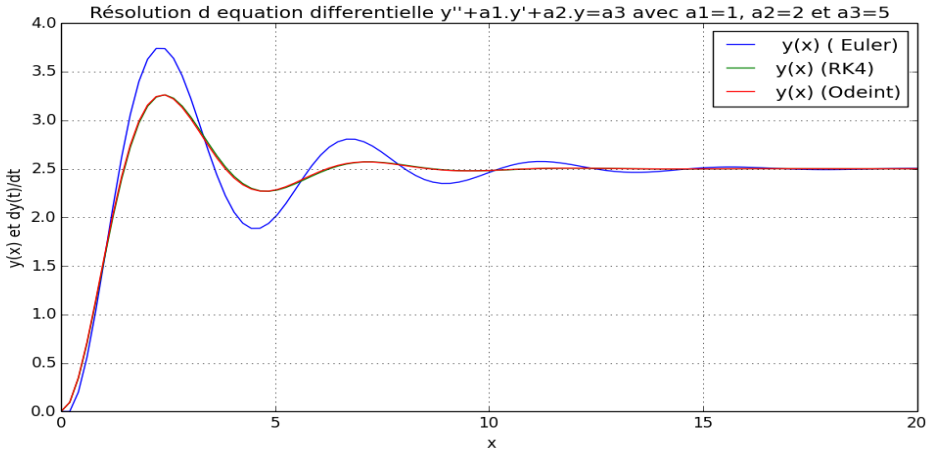


Figure11: Comparaison des Solutions pour n=100 (Resultat de python)

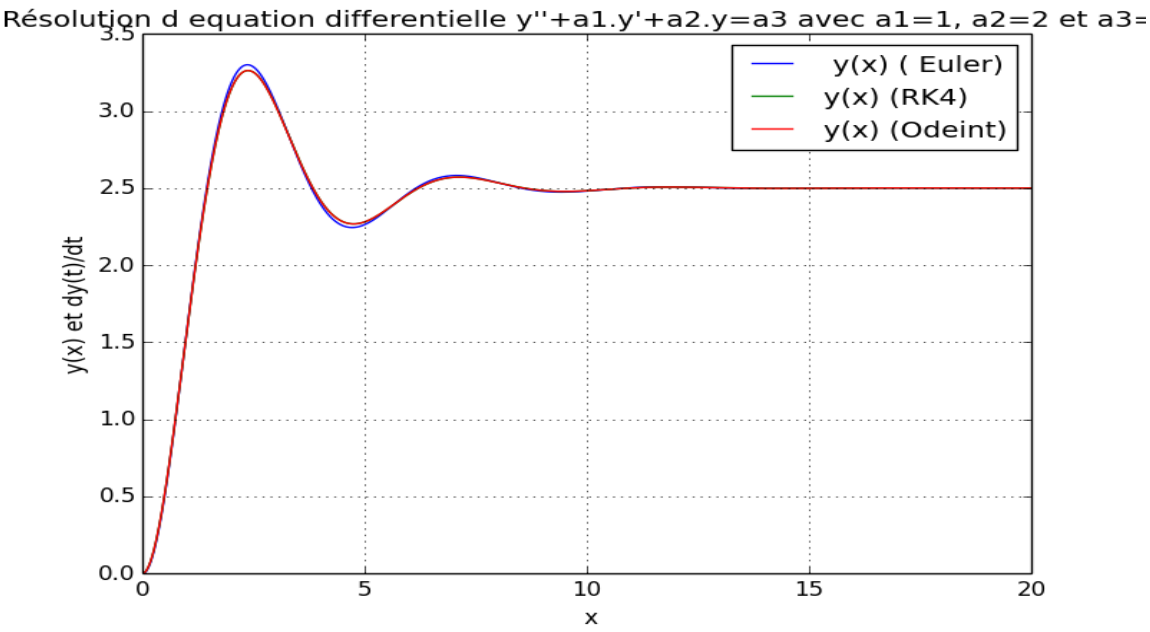


Figure12: Comparaison des Solutions pour n=1000 (Python)