Master : Méthodes Numériques et Statistiques Appliquées (MNSA)
Pr. Youssef OUASSIT

FSAC
FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

TD 5 : numpy et matplotlib

Exercice N°1 : Recherche zéro des fonctions

Méthode par dichotomie :

Commençons par la méthode la plus naturelle, dite de dichotomie (ou bissection). Soit $f: \mathbb{R} \rightarrow \mathbb{R}$ une fonction continue telle que f(a)*f(b) <= 0. D'après le théorème des valeurs intermédiaires, il existe $x^* \in [a, b]$ tel que $f(x^*) = 0$. Si de plus f est monotone, alors x^* est unique.

Principe :

La méthode de dichotomie repose sur le fait que si une fonction continue $f(x)$ change de signe entre deux points a et b, alors il existe au moins un zéro de la fonction dans l'intervalle $[a, b]$. La méthode consiste à diviser l'intervalle en deux sous-intervalles et à réduire progressivement l'intervalle contenant le zéro jusqu'à ce que l'intervalle soit suffisamment petit.

Nous considérons pour la suite de cet exercice la fonction $f(x) = x^2 - 4x - 3$, une fonction non linéaire et convexe.

1) Implémenter la méthode de la recherche de racine par dichotomie :

   dichotomie(f, a, b, tol=1e-6, maxIter=1000)

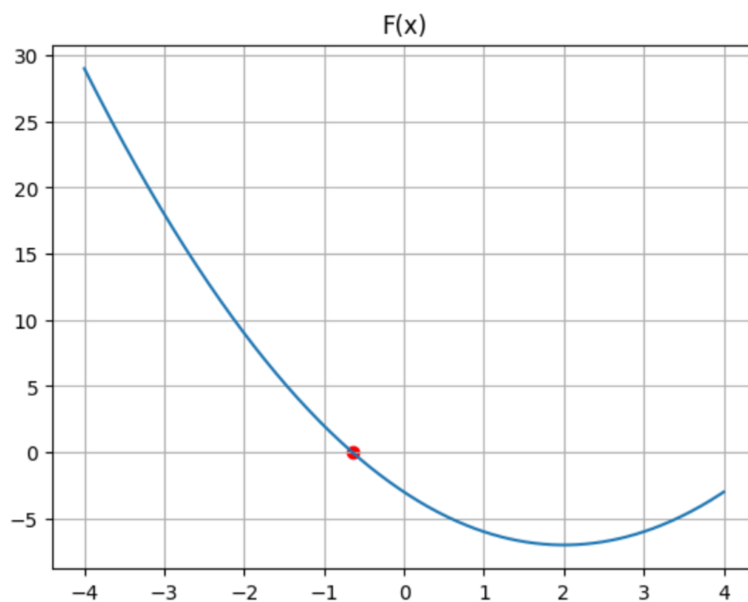
   - Si la fonction dépasse le nombre d'itération maximal avant que le critère d'arrêt soit réalisé, la fonction lève une exception avec un message d'erreur personnalisée.
   - Tester la fonction déclaré pour rechercher la racine de f dans l'intervalle $[-4,4]$. La fonction doit retourner la liste des termes de la suite et la solution finale.

2) Ecrire une fonction **materialiser(f, a, b)** qui va tracer la courbe de la fonction f et matérialiser la solution par un point rouge sur le graphe de la courbe.

3) Trouver la racine de la fonction f dans l'intervalle $[-4,4]$. Vous tracerez la courbe de la fonction et matérialiserez la solution sur le graphe.

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

def dichotomie(f, a, b, tol=1e-6, maxIt = 1000):
    assert f(a)*f(b)<=0 , 'Pas de solution dans [a,b]'
    i = 0
    while (b-a)>tol and i<maxIt :
        m = (a+b)/2
        if f(a)*f(m) < 0 :
            b = m
        else:
            a = m
        i += 1
    return a

def materialiser(f, a, b):
```

```
    xs = dichotomie(f, a, b)

    X = np.linspace(a , b , i**2)
    Y = f(X)

    plt.scatter( [xs] , [f(xs)] , color='red' )

    plt.plot(X, Y)
    plt.grid(True)
    plt.title('F(x)')

materialiser(lambda x: x**2 -4*x - 3, -4, 4)
```



Méthode de Newton :

La méthode de recherche de racine par dichotomie n'est pas très efficace. Elle a de plus l'inconvénient de nécessiter la définition d'un intervalle de recherche. La méthode de Newton est plus performante. Elle repose sur un développement de Taylor à l'ordre 1.

En effet, on rappelle que pour toute fonction f dérivable :

$$f(x) \simeq f(x_0) - f'(x_0)(x - x_0)$$

Pour trouver un zéro de cette fonction d'approximation, il suffit de calculer l'intersection de la droite tangente avec l'axe des abscisses, c'est-à-dire résoudre l'équation affine :

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

Ce qui donne :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

est plus proche de la racine que x_0. Ainsi, la méthode de Newton consiste à construire la suite suivante :

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

et à l'itérer jusqu'à convergence.

1. Implémenter l'algorithme de Newton pour trouver la racine d'une fonction réelle dérivable. La fonction **newton(x0, f, fp, tol=1e-6, maxIter=1000)** devra retourner la liste des valeurs de la suite ainsi que la solution approchée. Vous ferez attention au critère d'arrêt (convergence).

   Choisissez un des critères d'arrêt suivants :

   - Tolérance sur l'incrément $|x_k - x_{k-1}| \leq tol$:
   - Tolérance sur la Valeur de la Fonction (f(x)) : $|f(x_k)| \leq tol$

2. Proposer une version adaptée de la fonction **materialiser_newton(x0, f, fp)** qui dessine la courbe de la fonction f et matérialise le min de cette fonction.
3. Proposer une autre version simplifiée de la fonction **newtonv2 (x0, f)**

```python
def newton(x0, f, fp, tol=1e-6, maxIt=1000):
    i = 0
    x1 = x0 - f(x0)/fp(x0)
    while f(x1) > 0 and i < maxIt:
        x0 = x1
        x1 = x0 - f(x0)/fp(x0)
        i += 1
    return x1

def newtonV2(x0, f, tol=1e-6, maxIt=1000):
    fp  = lambda x : (f(x0+tol)-f(x0))/tol
    i = 0
    x1 = x0 - f(x0)/fp(x0)
    while f(x1) > 0 and i < maxIt:
        x0 = x1
        x1 = x0 - f(x0)/fp(x0)
        i += 1
    return x1
```

Exercice N° 2 : Pivot de Gauss

La résolution des systèmes linéaires est une tâche fondamentale en mathématiques et en sciences appliquées, avec plusieurs méthodes disponibles pour trouver les solutions. Plusieurs méthodes ont été proposée dans littérature, on distingue essentiellement :

- Les méthodes directes : Comme l'élimination de Gauss, décomposition LU, …
- Les méthodes itératives : Telles que les méthodes de Jacobi, de Gauss-Seidel, le gradient conjugué, …

Chaque méthode a ses propres avantages et inconvénients, et le choix de la méthode dépend souvent de la nature du système linéaire à résoudre, de la précision requise, et des ressources computationnelles disponibles.

Dans cet exercice, nous allons nous concentrer sur la méthode d'élimination de Gauss en traitant uniquement les cas simples, sans aborder les cas particuliers tels que les pivots nuls ou les matrices singulières.

L'exercice est divisé en plusieurs étapes avec des fonctions à implémenter :
A la colonne j, on effectue les opérations suivantes :
1 - On choisit un pivot non nul dans la colonne j dans une ligne d'indice supérieur (≥) à j
2 – On ramène le pivot sur la ligne j en effectuant éventuellement un échange de lignes
3 – On effectue les opérations suivantes sur les lignes j < i ≤ n :

$$L_j \leftarrow L_j - \frac{a_{ji}}{ajj} L_i$$

4 – On résout le système avec la phase de remontée après triangulariser le système.

1) La recherche d'un pivot :

Nous allons adopter la stratégie de pivot partiel, qui consiste à choisir le plus grand élément (en valeur absolue) dans la colonne courante comme pivot et à échanger la ligne courante avec la ligne contenant ce plus grand élément.

Ecrire une fonction « **chercher_pivot (M, j)** » qui prend en paramètre une matrice M et l'indice d'une colonne de la matrice M et renvoie un indice de ligne j>=j tel que | M_{ij} | soit maximal.

2) Les échanges de lignes :

On écrit maintenant une fonction « **echange_lignes(M, i, j)** » qui permet d'échanger la i-ème et la j-ème ligne de la matrice M.

3) Les transvections

On écrit une fonction **transvection(M, i, j, mu)** qui permet d'appliquer une transvection de la j ème ligne à la i ème ligne avec le scalaire mu.

$$L_j \leftarrow L_j - \frac{M_{ji}}{Mii} L_i$$

4) Triangulariser le système

Ecrire une fonction **triangulaire(A, b)** qui rend le système triangulaire supérieur

5) Phase de remontée

Ecrire une fonction **remontee(A, b)** qui prend en paramètre un système triangulaire supérieur et retourne le vecteur x.

Algorithme de remontée triangulaire :

Pour i = n; n-1; … ; 1 faire

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} A_{ij} x_j}{A_{ii}}$$

6) Recoller les morceaux :

On écrit une fonction **resolution(A, b)** qui prend en paramètre la matrice A et la matrice b et retourne le vecteur X contenant la solution du système.

<u>Tester le programme complet sur le système suivant :</u>

$$\begin{cases} 2x + y - z = 8 \\ -3x - y + 2z = -11 \\ -2x + y + 2z = -3 \end{cases}$$

La solution de système est : la solution : x=2, y=3, z=−1

<u>Exercice N°3 : Résolution numérique des équations différentielles</u>

I. Problème de Cauchy

En analyse, un problème de Cauchy est un problème constitué d'une équation différentielle dont on recherche une solution vérifiant une certaine condition initiale. Cette condition peut prendre plusieurs formes selon la nature de l'équation différentielle. Pour une condition initiale adaptée à la forme de l'équation différentielle, le **théorème de Cauchy-Lipschitz** assure l'existence et l'unicité d'une solution au problème de Cauchy.

$$\begin{cases} y' = F(y,t) \\ y(t0) = y0 \end{cases}$$

II. Résolution numérique des équations différentielles

<u>Méthode :</u>

1. Résolution numérique approchée sur l'intervalle $I=[t_0, t_0 + T]$ de longueur T (durée de simulation)
2. Discrétisation par découpage de l'intervalle de I selon un pas constant h,
3. Échantillonnage de la solution aux instants $t_i = t_0 + i\ h$ pour 1 <= i <= n ou $t_{i+1}=t_i+h$.
4. Finalement, la méthode numérique renvoie une liste $(y_0,y_1,,\ldots,y_N)$ contenant les valeurs approchées de $y(t_n)$ pour les différents instants t_n.

À chaque pas de la boucle, pour calculer $y(t_{i+1})$, on peut s'appuyer :

— sur la dernière valeur calculée $y(t_i)$: méthodes à un pas

— sur plusieurs valeurs $y(t_{i-k})$ (k >= 0) antérieurement calculées : méthodes à plusieurs pas (initialisation nécessaire par méthode à un pas)

III. Méthode Euler pour la résolution des équations différentiels d'ordre 1

On souhaite résoudre des équations scalaires du type :

$Y' = f(y,t)$ pout $t \in [t_0, t_N]$ avec $y_0=y(t_0)$

Nous allons utiliser le schéma explicite de Euler :

$$y_{i+1} = y_i + h\, f(y_i, t_i)$$

Q1- Ecrire une fonction « **euler1d(f, y0, t0, tN, N)** » permettant de résoudre cette équation par la méthode d'Euler en découpant l'intervalle en N pas de temps pour une fonction f à fournir. La fonction renverra les valeurs de la fonction yn sous forme d'un tableau numpy.

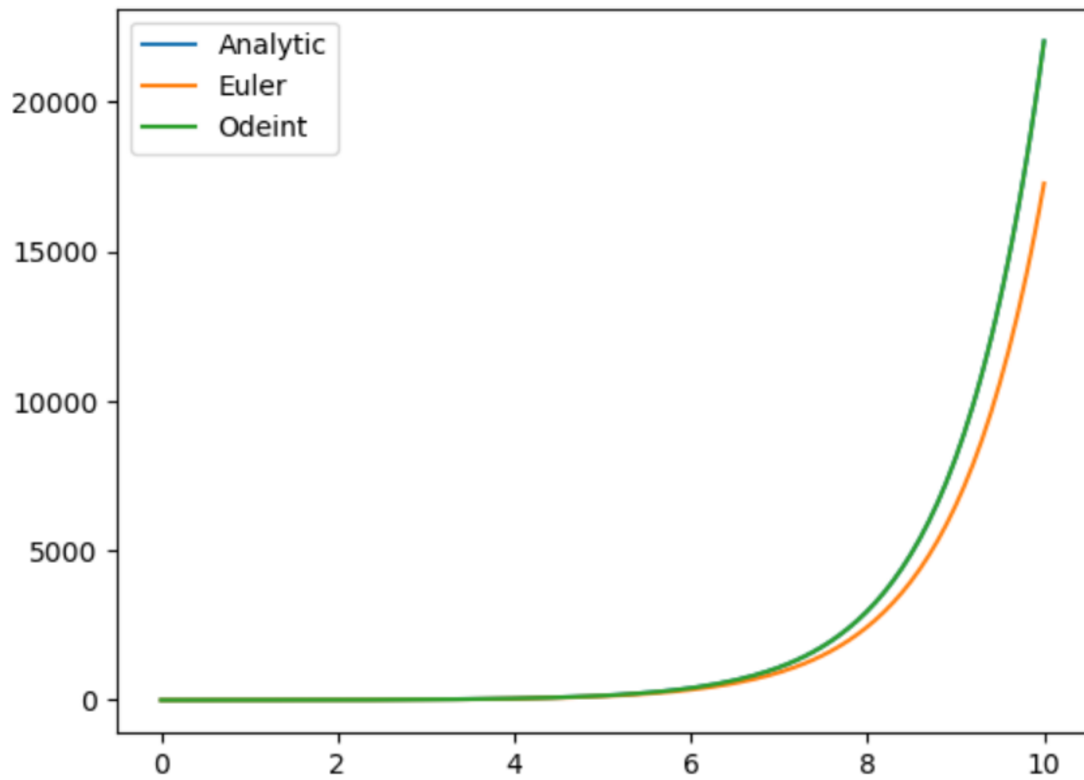
On s'intéresse d'abord à l'équation y'(t) = y(t) avec $y(t_0=0)=1$ et pour laquelle on dispose d'une solution exacte $y=y_0e^t$

Q2 - Ecrire une fonction « **cauchy1d(y, t)** » qui joue le rôle de la fonction f(y,t) associé à ce problème de Cauchy.

Q3- Tester la résolution pour différentes valeurs de N sur l'intervalle [0, 10]. Pour cela on crée une fonction « comparaison(f, y0, t0, tN, N) » qui :

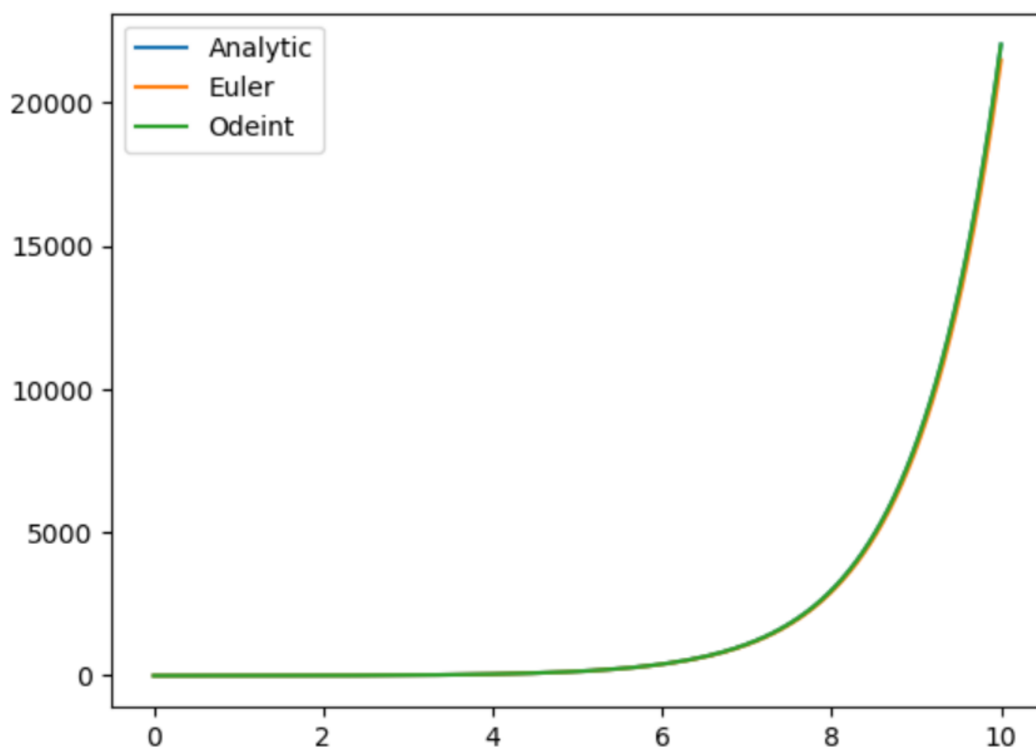
- Calcule la solution numérique approchée et exacte ;
- Calcule la solution avec la fonction odeint du module scipy
- Calcule et renvoie l'erreur globale ;
- Superpose le graphe de la solution approchée à celui de la solution exacte.

```
def euler1d(f,y0,t0,tN,N):
    t, h =np.linspace(t0,tN, N, retstep=True)
    y=np.zeros(N)
    y[0]=y0
    for i in range(0, N-1):
        y[i+1]=y[i]+h*f(y[i], t[i])
    return y

def cauchy1d(y,t):
    return y

def comparaison1D(f, y0, t0, tN, N):
    t=np.linspace(t0,tN, N)

    yEuler = euler1d(f,y0,t0,tN,N)

    yAnalytic = y0*np.exp(t)

    R = odeint(f,y0,t)
    yOdeint = R[:, 0]

    plt.plot(t, yAnalytic)
    plt.plot(t, yEuler)
    plt.plot(t, yOdeint)
    plt.legend(("Analytic", "Euler", "Odeint"))
    plt.show()

    err1 = max(np.abs(yEuler-yAnalytic))
    err2 = max(np.abs(yOdeint-yAnalytic))

    return err1, err2

comparaison1D(cauchy1d, 1,0, 10, 200)     # pour N=200
```




```
[7]:  (4754.223145186341, 0.003976824220444541)
```

```
comparaison1D(cauchy1d, 1,0, 10, 2000)     # pour N=2000
```




```
[8]:  (542.318677697087, 0.003955431471695192)
```