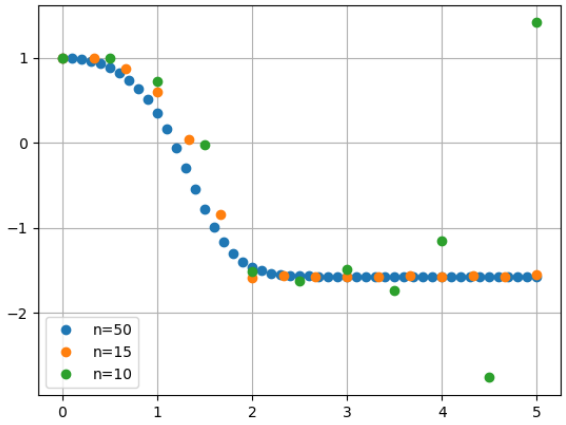
Proposition de corrigé du TP1 : Méthodes numériques pour la résolution d'équations

1 Listes

Exercice TP1.1 *Méthode d'Euler pour la résolution d'équations différentielles*

1,2,3 Après avoir importé les modules nécessaires, on définit `f` et `EulerExplicite1` :

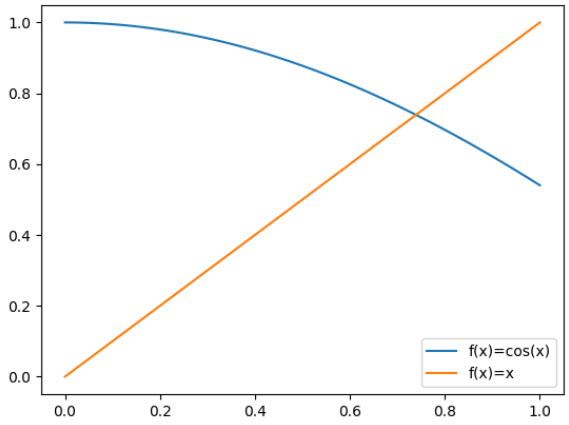
```
import math

def f(y,t):
    return -2*t*math.cos(y)
def EulerExplicite1(f,t0,tf,n,y0):
    t=[0]*(n+1)
    y=[0]*(n+1)
    t[0]=t0
    y[0]=y0
    h=(tf-t0)/n
    for i in range(0,n):
        t[i+1]=t[i]+h
        y[i+1]=y[i]+h*f(y[i],t[i])
    return t,y

## Euler explicite

t0=0.0
tf=5.0
y0=1.0

n=50
temps1, solutionEE1=EulerExplicite1(f,t0,tf,n,y0)

n=15
temps2, solutionEE2=EulerExplicite1(f,t0,tf,n,y0)

n=10
temps3, solutionEE3=EulerExplicite1(f,t0,tf,n,y0)
# Plot de la figure
plt.figure()
plt.plot(temps1,solutionEE1,"o",label='n=50')
plt.plot(temps2,solutionEE2,"o",label='n=15')
plt.plot(temps3,solutionEE3,"o",label='n=10')
plt.grid()
plt.legend()
plt.show()
```

4 Pour la dichotomie :

```
def dichotomie(f, a, b, eps):
    """
    Résout l'équation f(x) = 0 sur l'intervalle [a, b] avec une précision eps.
    Retourne une solution approchée et le nombre d'itérations effectuées.
    """
    assert f(a) * f(b) < 0 ,"La␣fonction␣doit␣changer␣de␣signe␣sur␣l'intervalle␣[a,␣b]"


    compteur = 0
    while (b - a) / 2 > eps:
        c = (a + b) / 2
        if f(c) == 0:
            return c, compteur
        elif f(c) * f(a) < 0:
            b = c
        else:
            a = c
        compteur += 1


    solution = (a + b) / 2
    return solution, compteur
def ftestdicho(x):
    return math.cos(x) - x
    # Test de la fonction
a, b = 0, 1


epsilons = [1e-2, 1e-4, 1e-8]


for eps in epsilons:
    solution, iterations = dichotomie(ftestdicho, a, b, eps)
    print(f"Pour␣epsilon␣=␣{eps},␣la␣solution␣est␣{solution}␣avec␣{iterations}␣itérations.")
```

```
X=np.linspace(a,b,100)


plt.figure()
plt.plot(X,[math.cos(i) for i in X],label='f(x)=cos(x)')
plt.plot(X,X,label='f(x)=x')
plt.legend()
plt.show()
```



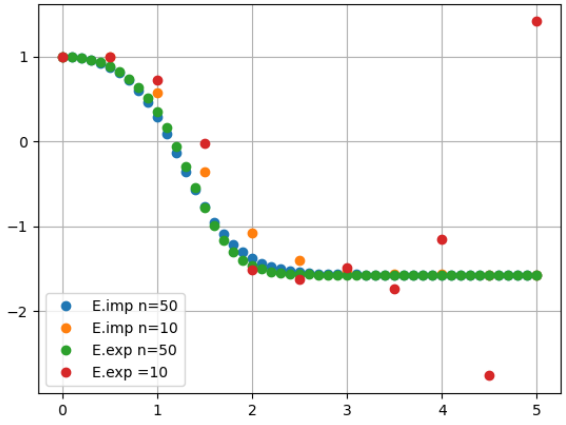
5,6,7 Pour Euler implicite :

```
def euler_implicite(f, t0, tf, n, y0):
    """
    Résolution numérique d'une équation différentielle par la méthode d'Euler implicite.


    :param f: Fonction f(y, t) représentant l'équation différentielle dy/dt = f(y, t)
    :param t0: Temps initial
    :param tf: Temps final
    :param n: Nombre de pas
    :param y0: Condition initiale
    :return: Deux listes, liste_t (abscisses) et liste_y (ordonnées)
    """
    # Calcul du pas de temps
    h = (tf - t0) / n


    # Initialisation des listes
    liste_t = [t0 + i * h for i in range(n + 1)]
    liste_y = [0]*(n + 1)
    liste_y[0] = y0


    # Boucle principale d'Euler implicite
    for i in range(n):
        # Fonction g(x) àrésoudre pour obtenir yn+1
```



```
        g = lambda x: x - liste_y[i] - h * f(x, liste_t[i])


        # Résolution de l'équation par la dichotomie
        liste_y[i+1] = dichotomie(g, liste_y[i] - 5 * h, liste_y[i] + 5 * h,1e-4)
    return liste_t, liste_y
```

```
temps_EI_1 ,sol_EI_1 = euler_implicite (f ,0 ,5 ,50 ,1)
temps_EI_2 ,sol_EI_2 = euler_implicite (f ,0 ,5 ,10 ,1)


plt.figure()
plt.plot(temps_EI_1,sol_EI_1,'o',label='E.imp␣n=50')
plt.plot(temps_EI_2,sol_EI_2,'o',label='E.imp␣n=10')
plt.plot(temps1,solutionEE1,"o",label='E.exp␣n=50')
plt.plot(temps3,solutionEE3,"o",label='E.exp␣=10')
plt.grid()
plt.legend()
plt.show()
```



Exercice TP1.2 *Application à la loi entrée-sortie du Clever*

8,9,10 On implémente directement la fonction `lambda1` :

```
a = 0.14
b = 0.046
L = 0.49
#
alpha = np.linspace(-50,50,101)*np.pi/180 #en radian
#
```

```
def lambda1(alpha):
    return np.sqrt((L*np.cos(-130*np.pi/180+alpha)+a)**2 + (L*np.sin(-130*np.pi/180+alpha)-b)**2)
#
plt.figure()
plt.plot(lambda1(alpha),alpha*180/np.pi)
plt.xlabel('lambda')
plt.ylabel('alpha')
plt.grid()
plt.show()
```

Graphiquement, pour $\lambda = 0.4$ on lit $\alpha = -23$

11,12,13,14 Résolution avec la dichotomie, on doit définir la fonction f qui tarduit l'équation $\lambda(\alpha) - 0,4 = 0$

```
# Fonction f(alpha, lambda0=0.4)
def f(alpha, lambda0=0.4):
    return lambda0 - lambda1(alpha)


# Méthode de dichotomie reformulée
def dichotomie(f, a, b, eps=1e-10):
    compteur = 0
    erreurs = [] # Liste pour suivre les erreurs àchaque itération
    solutions = [(a + b) / 2] # Liste des solutions àchaque itération

    while abs(a - b) > eps and compteur < 5000:
        c = (a + b) / 2
        if f(c) * f(a) < 0: # f(a) et f(c) ont des signes différents
            b = c
        else:
            a = c

        compteur += 1
        erreurs.append(abs(a - b))
        solutions.append((a + b) / 2)

    return (a + b) / 2, compteur, erreurs, solutions

# Exécution de la méthode de dichotomie
tic = time.time()
xd, iterations, erreurs, solutions = dichotomie(f, -50 * np.pi / 180, 50 * np.pi / 180)

# Affichage des résultats
print('Dichotomie:', xd * 180 / np.pi, 'Temps écoulé:', time.time() - tic, 'Nombre d\'itérations:', iterations)
```

15 En calculant la dérivée exacte et en l'implémentant dans `fp(alpha` on peut écrire `newton` :

```
# Calcul de la dérivée exacte
def fp(alpha):
    return (a * L * np.sin(-130 * np.pi / 180 + alpha) + b * L * np.cos(-130 * np.pi / 180 + alpha)) / lambda1(alpha)

# Méthode de Newton avec dérivée exacte
def newton(f, fp, xini, eps=1e-10):
    """
    f  : fonction dont on cherche la racine
    fp : dérivée de la fonction f
    xini : point initial pour commencer l'approximation
    eps : tolérance pour la convergence (précision)

    Retourne la solution, le nombre d'itérations, la liste des erreurs, et les approximations successives.
    """
    niter = 0
    erreurs = [] # Liste pour suivre les erreurs àchaque itération
```



```
    solutions = [] # Liste des approximations successives
    x = xini
    xo = 1e8 # Une valeur initiale éloignée pour commencer

    while abs(f(x) - f(xo)) > eps and niter < 500:
        xo = x
        fp1 = fp(x) # Calcul de la dérivée exacte en x
        x = x - f(x) / fp1 # Mise àjour de x avec la méthode de Newton

        niter += 1
        erreurs.append(abs(x - xo)) # Suivi de l'erreur
        solutions.append(x) # Ajout de la solution courante

    return x, niter, erreurs, solutions

# Exécution de la méthode de Newton
tic = time.time()
xn, itern, erreursn, solutionsn = newton(f, fp, 0)

# Affichage des résultats
print('Newton exacte :', xn * 180 / np.pi, 'Temps écoulé :', time.time() - tic,\\
 'Nombre d\'itérations :', itern)
```

16 En calculant la dérivée approchée par son taux d'accroissement :

```
# Méthode de Newton avec dérivée approchée
def newton2(f, xini, eps=1e-10, h=1e-1):
    """
    f  : fonction dont on cherche la racine
    xini : point initial pour commencer l'approximation
    eps : tolérance pour la convergence (précision)
    h : incrément pour la dérivée approchée

    Retourne la solution, le nombre d'itérations, \\
    la liste des erreurs, et les approximations successives.
    """
    niter = 0
    erreurs = [] # Liste pour suivre les erreurs àchaque itération
    solutions = [] # Liste des approximations successives
    x = xini
    xo = 1e8 # Une valeur initiale éloignée pour commencer

    while abs(f(x) - f(xo)) > eps and niter < 500:
        xo = x
        # Dérivée approchée par différence finie
        fp1 = (f(x + h) - f(x)) / h
        x = x - f(x) / fp1 # Mise àjour de x avec la méthode de Newton
        niter += 1
        erreurs.append(abs(x - xo)) # Suivi de l'erreur
        solutions.append(x) # Ajout de la solution courante

    return x, niter, erreurs, solutions

# Exécution de la méthode de Newton avec dérivée approchée
tic = time.time()
xn2, itern2, erreursn2, solutionsn2 = newton2(f, 0)

# Affichage des résultats
print('Newton approchée :', xn2 * 180 / np.pi, 'Temps écoulé :', time.time() - tic,\\
 'Nombre d\'itérations :', itern2)
```

Exercice TP1.3 *Application au Gyroscope*

L'équation s'écrit $A\ddot{\beta} + B\dot{\beta}' + C\beta = K1.u(t)$ ou s'écrit $\frac{\ddot{\beta}}{\omega_0^2} + \frac{2\xi}{\omega_0}\beta + \beta = K.u(t)$

Écrivons le problème de Cauchy associé à l'équation différentielle :

$$\frac{1}{\omega_0^2}\ddot{\beta} + \frac{2\xi}{\omega_0}\dot{\beta} + \beta = Ku(t)$$

Pour simplifier le système, nous définissons une nouvelle variable $q(t) = \dot{\beta}(t)$. Nous pouvons alors réécrire l'équation sous forme d'un système de deux équations différentielles du premier ordre :

$$\begin{cases} \dot{\beta}(t) = q(t) \\ \dot{q}(t) = -\omega_0^2\beta(t) - 2\xi\omega_0 q(t) + \omega_0^2 Ku(t) \end{cases}$$

Les conditions initiales sont :

$$\beta(0) = 0, \quad q(0) = 0$$

Ainsi, le problème de Cauchy associé est le système suivant avec les conditions initiales :

$$\begin{cases} \dot{\beta}(t) = q(t) \\ \dot{q}(t) = -\omega_0^2\beta(t) - 2\xi\omega_0 q(t) + \omega_0^2 Ku(t) \end{cases}$$

avec $\beta(0) = 0$ et $q(0) = 0$.

Écrivons les deux équations de récurrence permettant de résoudre l'équation différentielle par la méthode d'Euler explicite.

$$\beta_{n+1} = \beta_n + \Delta t \cdot q_n$$

$$q_{n+1} = q_n + \Delta t \cdot \left(-\omega_0^2\beta_n - 2\xi\omega_0 q_n + \omega_0^2 Ku(t_n)\right)$$

avec $\beta(0) = 0$ et $q(0) = 0$ comme conditions initiales, et $u(t)$ étant une fonction de Heaviside.

```
# t0: temps initial
# beta0: valeur initiale
# q0: vitesse initiale
# T: temps de simulation
# w0: pulsation propre non amorti
# ksi: coef d'amortissement
# K: gain
#
A=6.6E-6
B=1.5E-4
C=(4.4E-6)+(0.0049)
K1=0.0015
w0=(C/A)**0.5
ksi=0.5*w0*B/C
K=K1/C
T=1
t0=0
beta0=0
q0=0
N=500
#
#gyroscope explicite
def gyro_explicite(t0,beta0,q0,N,T,w0,ksi,K):
    h=(T-t0)/N
    listet=[t0]
    listebeta=[beta0]
    listeomega=[q0]
    for i in range(N):
```



```
        listet.append(t0+(i+1)*h)
        listebeta.append(listebeta[-1]+h*listeomega[-1])
        listeomega.append(listeomega[-1]+h*w0*(w0*K-2*ksi*listeomega[-1]-w0*listebeta[-2]))
    return listet,listebeta

X1,Y1=gyro_explicite(t0,beta0,q0,N,T,w0,ksi,K)
legende="Solution euler explicite N = "+str(N)
plt.plot(X1,Y1,label=legende)
plt.grid()
#
#
```

```
def second_ordre_exacte(t0,T,w0,ksi,K,N):
    x=np.linspace(t0,T,N+1)
    y=K*(1-np.exp(-ksi*w0*x)*(np.cos(w0*np.sqrt(1-ksi**2)*x)+(ksi/np.sqrt(1-ksi**2))*np.sin(w0*np.s
    return x,y
#
X2,Y2=second_ordre_exacte(t0,T,w0,ksi,K,N)
plt.plot(X2,Y2,label='Solution exacte')

def gyro_implicite(t0,beta0,q0,N,T,w0,ksi,K):
    h=T/N
    listet=[t0]
    listebeta=[beta0]
    listeomega=[q0]
    for i in range(N):
        listet.append(listet[-1]+h)
        listeomega.append((listeomega[-1]+h*K*w0*w0-h*w0*w0*listebeta[-1])/(1+2*ksi*w0*h+h*h*w0*w0)
        listebeta.append(listebeta[-1]+h*listeomega[-1])
    return listet,listebeta

X3,Y3=gyro_implicite(t0,beta0,q0,N,T,w0,ksi,K)
legende="Solution euler implicite N = "+str(N)
plt.plot(X3,Y3,label=legende)
plt.xlabel('temps')
plt.ylabel('y(t)')
plt.title('Comparaison Euler/Solution exacte pour un second ordre')
plt.legend(loc=1)
plt.show()
```