Capacités numériques

Résolution numérique d'équations différentielles

Capacités exigibles :

— Équations différentielles d'ordre 1.
Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1.

— Équations différentielles d'ordre supérieur ou égal à 2.
Transformer une équation différentielle d'ordre n en un système différentiel de n équations d'ordre 1.

— Utiliser la fonction `odeint` (obsolète, remplacée par `solve_ivp`) de la bibliothèque `scipy.integrate` (sa spécification étant fournie).

I Méthode d'Euler

I.1 Principe de la méthode d'Euler

L'algorithme d'Euler permet de résoudre de façon approchée une équation différentielle du type

$$\begin{cases} \dfrac{\mathrm{d}X}{\mathrm{d}t} &= f(t, X) \\ X(t_0) &= X_0 \end{cases}$$

D'après la définition de la dérivée comme limite du taux d'accroissement : $\dfrac{\mathrm{d}X}{\mathrm{d}t}(t) = \lim\limits_{h\to 0} \dfrac{X(t+h) - X(t)}{h}$.

Dans le cadre du schéma Euler explicite, au premier ordre, on peut approximer la dérivée à l'instant t par :

$$\frac{\mathrm{d}X}{\mathrm{d}t}(t) \approx \frac{X(t+h) - X(t)}{h}$$

soit

$$X(t+h) \approx X(t) + h \times f(t, X(t))$$

qui correspond au développement limité de X au premier ordre au voisinage de t.

La résolution numérique consiste à déterminer les valeurs X_i de X aux $(n+1)$ instants $(t_i)_{i\in[\![0,n]\!]}$ séparés de la durée h, soit aux instants : $t_i = t_0 + i \times h$, et $t_{i+1} = t_i + h$.

La valeurs approchées X_i de X à l'instant t_i sont définies par la relation de récurrence

$$X_{i+1} \approx X_i + h \times f(t_i, X_i)$$

Ainsi, connaissant la fonction f (forcément puisqu'on connait l'équation différentielle que l'on souhaite résoudre !) et la condition initiale $X(t_0) = X_0$, on peut déterminer $X_1 = X_0 + h \times f(t_0, X_0)$. Puis on peut déterminer X_2 : $X_2 = X_1 + h \times f(t_1, X_1)$, et ainsi de suite.

h est le pas de calcul, qu'il faudra choisir de façon pertinente. Il devra être suffisamment petit pour rendre compte des variations locales de f. En diminuant h on augmente la précision du calcul, mais on augmente aussi le temps de calcul.

I.2 Implémentation

On va écrire une fonction `euler` qui prendra en argument :

— la fonction `f` qui définit l'équation différentielle ;

— les bornes `t0` et `tf` de résolution ;

— le nombre `n` de pas de calcul ;

— la condition initiale `X0`

et renverra la tableau `X` des `n+1` valeurs approchées déterminées.

```
# On commence par importer les bibliothèques nécessaires :
import numpy as np # pour utiliser les tableaux
import matplotlib.pyplot as plt # pour les représentations graphiques
def Euler(f,X0,t0,tf,n):
    """
    f : fonction définissant l'équation différentielle
    X0 : condition initiale
    t0 : instant initial et tf : instant final
    n : nombre de pas de calcul : h=(tf-t0)/n
    """
    h=                          # (à compléter) pas de calcul
    t=np.linspace(    ,    ,         ) #  (à compléter) liste des temps (n+1 instants de calculs) répartis régulièrement entre t0 et tf
    X=np.zeros(              ) #  (à compléter) tableau qui contient n+1 zéros au départ
    X[0]=       #  (à compléter)  le 1er élément du tableau est la condition initiale
    for i in range(    ,   ): #   (à compléter) il reste n instants à calculer
    # calcul du point suivant (élément i+1) à partir du pt précédent ( élément i) :
        X[i+1]=                                          #  (à compléter)
    return X #la fonction renvoie le tableau X des valeurs de X successives
```

I.3 Exemples

Exercice 1 Régime transitoire du premier ordre en électricité

On étudie la charge du condensateur à travers une résistance R, par un générateur de fem E constante. La tension aux bornes du condensateur vérifie l'équation différentielle

$$\frac{\mathrm{d}u_C}{\mathrm{d}t} + \frac{u_C}{RC} = \frac{E}{RC}$$

Le condensateur est initialement déchargé, donc $u_C(0) = 0$. On souhaite résoudre cette équation différentielle en utilisant la méthode d'Euler.

Q1. * Récrire l'équation différentielle sous la forme de celle du § I.1, et identifier la fonction f définissant ici l'équation différentielle.

Q2. Définir, dans l'éditeur, les grandeurs nécessaires : $E = 5$ V, $R = 1000\ \Omega$ et $C = 100$ nF.

Q3. Définir, en python, la fonction `f_charge(t,uc)` qui prend en argument deux flottants (l'instant `t` et la tension `uc`) et renvoie $\frac{\mathrm{d}u_C}{\mathrm{d}t}$ définie par l'équation différentielle (autrement dit la valeur de $f(t, u_c)$ définie précédemment).

Q4. Écrire l'instruction permettant de récupérer, dans `solution` le tableau des valeurs de u_C par application de la fonction `euler`. On choisira $t_0 = 0$, $t_f = 10 \times RC$, $n = 10000$.

Q5. Représenter l'évolution de la tension aux bornes du condensateur.

```
t=np.linspace(t0,tf,n+1) # instants
plt.plot(t,solution) # représentation des valeurs de uc en fonction du temps
plt.title('Titre')
plt.xlabel('abscisse (unité)')
plt.ylabel('ordonnée(unités)')
plt.show() # affiche le graphique
```

Q6. Reprendre en choisissant comme condition initiale $u_C(0) = \frac{E}{2}$.

Exercice 2 Étude de la montée en vitesse d'un moteur à courant continu

On étudie la montée en vitesse d'un moteur à courant continue dont la tension d'alimentation passe instantanément de 0 à U_{alim}. Si l'on néglige l'inductance du bobinage, l'évolution de la vitesse de rotation du moteur à courant continu est régie par l'équation différentielle suivante :

$$J_{\text{eq}}\frac{\mathrm{d}\omega}{\mathrm{d}t} = \frac{K_c}{R}U_{\text{alim}} - \left(f + \frac{K_c \times K_e}{R}\right)\omega(t)$$

On prendra les caractéristiques du système Maxpid :

- Constante électrique : $K_e = 5,25.10^{-2}$ V · s · rad^{-1}
- Constante de couple : $K_c = 5,25.10^{-2}$ N · m · A^{-1}
- Résistance aux bornes : $R = 2,07\ \Omega$
- Inertie équivalent ramenée au rotor moteur : $J_{eq0} = 6,96.10^{-6}$ kg · m^2 (valeur pour l'arbre moteur seul)
- Facteur de frottements fluides : $f = 0$ N · m · s · rad^{-1} (frottements fluides négligés)

La vitesse initiale du rotor moteur est considérée nulle et on alimente le moteur sous la tension $U_{\text{alim}} = 24$ V.

Q1. * Réécrire l'équation différentielle sous la forme de celle du § I.1, et identifier la fonction f définissant ici l'équation différentielle.

Q2. Représenter l'évolution de la vitesse de rotation du moteur entre les instants $t_0 = 0$ s et $t_f = 2$ s.

Sur un système, le moteur met en mouvement une chaîne cinématique ayant une inertie plus ou moins importante. L'inertie de la chaîne cinématique influe directement sur la valeur de J_{eq}. Dans la suite, on va donc s'intéresser à l'évolution de la réactivité du système lorsque J_{eq} augmente.

Q3. Tracer sur un même graphe l'évolution de la vitesse de rotation de l'arbre moteur (entre les instants $t_0 = 0$ s et $t_f = 2$ s) pour $J_{\text{eq}} = J_{\text{eq0}}$; $J_{\text{eq}} = 2 \times J_{\text{eq0}}$; $J_{\text{eq}} = 4 \times J_{\text{eq0}}$; $J_{\text{eq}} = 8 \times J_{\text{eq0}}$; $J_{\text{eq}} = 16 \times J_{\text{eq0}}$; $J_{\text{eq}} = 32 \times J_{\text{eq0}}$; $J_{\text{eq}} = 64 \times J_{\text{eq0}}$

Q4. Commenter l'évolution du temps de réponse du système.

II Résolution d'équations différentielles d'ordre 2 avec les fonctions python

II.1 Fonction `solve_ivp`

Plusieurs fonctions sont déjà définies dans Python pour résoudre numériquement des équations différentielles.

On va utiliser ici la fonction `solve_ivp` disponible dans la bibliothèque `scipy.integrate` qui permet de résoudre les équations différentielles sous la forme

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}t} &= f(t,y) \\ y(0) &= y_0 \end{cases}$$

où y est un vecteur de taille N et f une fonction de $\mathbb{R}^N$ dans $\mathbb{R}^N$.

Après avoir importé la bibliothèque, on peut regarder la spécification de la fonction en saisissant dans la console :

```
import scipy.integrate as sci # on importe la bibliothèque nécessaire
>>> help(sci.solve_ivp)
solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False,
    events=None, vectorized=False, args=None, **options)
     fun : callable
         Right-hand side of the system. The calling signature is ''fun(t, y)'
    '.
         Here 't' is a scalar, and there are two options for the ndarray 'y':
         It can either have shape (n,); then 'fun' must return array_like
    with
         shape (n,). Alternatively it can have shape (n, k); then 'fun'
         must return an array_like with shape (n, k), i.e. each column
         corresponds to a single column in 'y'. The choice between the two
         options is determined by 'vectorized' argument (see below). The
         vectorized implementation allows a faster approximation of the
    Jacobian
         by finite differences (required for stiff solvers).
     t_span : 2-tuple of floats
         Interval of integration (t0, tf). The solver starts with t=t0 and
         integrates until it reaches t=tf.
     y0 : array_like, shape (n,)
         Initial state. For problems in the complex domain, pass 'y0' with a
         complex data type (even if the initial value is purely real).
     t_eval : array_like or None, optional
         Times at which to store the computed solution, must be sorted and
    lie
         within 't_span'. If None (default), use points selected by the
    solver.
Returns
-------
     Bunch object with the following fields defined:
     t : ndarray, shape (n_points,)
         Time points.
     y : ndarray, shape (n, n_points)
         Values of the solution at 't'.
```

Pour notre utilisation, on définira :

— la fonction $f(t,y)$;

— l'intervalle de temps de résolution (t_0, t_f), tuple de 2 flottants ;

— le vecteur de condition initiale y_0 ;

— le tableau des instants de résolution.

Pour l'étude du régime transitoire du premier ordre de l'exercice 1, avec les mêmes paramètres que précédemment :

```
resol=sci.solve_ivp(f_charge,(0,10*R*C),np.array([0]),t_eval=np.linspace
    (0,10*R*C,10001))
temps=resol.t # on récupère les instants de résolutions
# resol.y est le tableau des valeurs de y, chaque colonne correspondant à un
    instant de résolution.
theta=resol.y[0] # récupération de l'unique ligne de resol.y : u_c(t)
```

Exercice 3 Régime transitoire du premier ordre en électricité *(suite)*

Mettre en œuvre la fonction `solve_ivp` sur l'exemple du régime transitoire du premier ordre, et tracer l'évolution de u_c en fonction du temps.

II.2 Pendule pesant

Exercice 4 Pendule pesant

L'équation différentielle du pendule pesant

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \omega_0^2 \sin(\theta) = 0$$

ne peut pas être résolue analytiquement. Sa résolution nécessite une méthode numérique.
L'équation du pendule pesant est une équation du deuxième ordre, qui ne peut pas être résolue directement en utilisant la méthode d'Euler, qui s'applique à une équation différentielle du 1er ordre.
Il est nécessaire de récrire l'équation différentielle du pendule pesant sous la forme d'une équation vectorielle du premier ordre.

Pour cela, on pose $X = \begin{pmatrix} \theta \\ \dfrac{\mathrm{d}\theta}{\mathrm{d}t} \end{pmatrix}$

Q1. * Exprimer $\dfrac{\mathrm{d}X}{\mathrm{d}t}$ sous la forme $\begin{pmatrix} \dots \\ \dots \end{pmatrix}$, on exprimera les deux lignes en fonction de ω_0, θ et $\dfrac{\mathrm{d}\theta}{\mathrm{d}t}$.

En déduire la fonction $f(t, X)$ qui définit l'équation différentielle sous la forme $\dfrac{\mathrm{d}X}{\mathrm{d}t} = f(t, y)$

Q2. Écrire en python la fonction `f_pendule(t,X)` qui prend en argument un flottant `t` et un tableau `numpy X` de deux éléments, et renvoie le vecteur $\dfrac{\mathrm{d}X}{\mathrm{d}t}$, c'est-à-dire un tableau `numpy` de deux éléments, celui défini à la question précédente.

Q3. On utilise la fonction `solve_ivp` pour résoudre numériquement cette équation différentielle sur $\left[0, 4 \times \dfrac{2\pi}{\omega_0}\right]$ (4 périodes de petites oscillations), pour 10000 pas de calculs.
Le faire pour des conditions initiales au choix.

```
# Définition des paramètres
w0=1 # rad/s (ou autre valeur)
t0= # (à compléter) instant initial
tf= #  (à compléter) instant final
N= #  (à compléter) nombre de pas de résolution

# Conditions initiales
theta0= # (à compléter) angle initial
dtheta0= # (à compléter) vitesse angulaire initiale
CI=  # (à compléter) tableau numpy qui contient les deux valeurs
   précédentes

# instants de calculs
```

```
t=np.linspace(    ,    ,    ) # (à compléter) tableau des instants de
   calculs

# résolution
solution=  # (à compléter) utilisation de la fonction solve_ivp

# récupération de l'angle et de la vitesse angulaire
theta=  # (à compléter)
dtheta= # (à compléter)
```

Q4. Tracer les graphes de $\theta(t)$ et $\dfrac{\mathrm{d}\theta}{\mathrm{d}t}$.

Q5. On veut étudier l'influence des conditions initiales sur l'évolution ultérieure.

Pour cela, on résout l'équation différentielle pour différentes conditions initiales : $\dot{\theta}(0) = 0$ et différentes valeurs de $\theta(0)$ choisies dans l'intervalle $]0, \pi[$.

Compléter le script ci-dessous.

```
# résolution
# On définit la liste des angles et/ou vitesses initiales que l'on veut
   étudier
liste_theta0=[0.1,0.3,0.5,1,2,2.9] # valeurs de theta(0) à tester
plt.figure() # on trace toutes les évolutions sur le même graphe
for theta0 in liste_theta0: # parcours de la liste des valeurs de theta0
   choisies
    # pour chaque valeur de theta(0) on résout l'équation différentielle
    # on récupère la liste des theta
    # on trace le graphe de theta(t)
    CI= np.array([    ,    ]) # (à compléter) tableau de la CI testée (ici
    on ne change que l'angle initial pour une vitesse angulaire initiale
   nulle
    solution = #  (à compléter) résolution avec solve_ivp
    theta=  #  (à compléter) on récupère les angles
    #  tracé de theta(t)
    plt.plot(t,theta,label='theta0='+str(theta0)+'rad')
plt.xlabel(r'$t$ (s)')
plt.ylabel(r'$\theta$ (rad)')
plt.grid()
plt.legend(loc='best')  # légende
plt.show()
```

Q6. Commenter les courbes obtenues.