Spé Bio 1 & 2 - Lycée Champollion septembre 2024

Informatique - TP N°2

Méthode d'Euler (2)

1. Système d'équations différentielles couplées

Un problème se traduit parfois par des équations différentielles faisant intervenir des grandeurs couplées, donnant un système du type

$$\begin{cases} a'(t) &= f_1(t, a(t), b(t)) \\ b'(t) &= f_2(t, a(t), b(t)) \end{cases}$$

Par exemple considérons le système d'équations différentielles suivant :

$$\begin{cases} a'(t) &= 2a(t) + b(t) \\ b'(t) &= -a(t) + 2b(t)) \end{cases} ,$$

avec les conditions initiales $a(0) = 3, b(0) = 4$.

On peut bien sûr réécrire une fonction euler similaire à la précédente, avec davantage de paramètres (deux valeurs initiales, deux fonctions pour les dérivées a' et b') qui calcule à chaque pas deux valeurs approchées a_k et b_k.

Mais l'idée est plutôt de réutiliser la fonction euler déjà écrite, en considérant une grandeur y **vectorielle** :

Posons $y(t) = \begin{pmatrix} a(t) \\ b(t) \end{pmatrix}$, alors $y'(t) = \begin{pmatrix} a'(t) \\ b'(t) \end{pmatrix}$ et le système peut s'écrire

$$y'(t) = f(t, y)$$

où f est une fonction prenant en arguments un réel et un vecteur, et renvoyant un vecteur. Dans notre exemple, on aurait par exemple, si y est un *tuple* :

```
def f2(t,y):
    """reçoit un réel t et un tuple y à deux coordonnées. Renvoie
    la dérivée de y sous forme d'un tuple à deux coordonnées"""
    a,b = y        # récupération des coordonnées
    return (2*a + b, -a + 2*b)
```

Problème : les tuples (ou les listes) classiques de python se prêtent mal aux calculs de la méthode d'Euler : multiplication par un scalaire (h) et addition.

En effet : 2*(5,3) renvoie (5,3,5,3) et non (10,6) comme on voudrait ;

(5,3) + (1,-1) renvoie (5,3,1,-1) (concaténation) et non (6,2) ;

0.5*(5,3) n'est pas autorisé...

(même problème avec les listes python)

Il nous faut donc passer par les matrices du module numpy (numpy.array) qui autorisent les opérations algébriques souhaitées :

```
>>> 2*(np.array([5,3]))
array([10,  6])
>>> 0.5*(np.array([5,3]))
array([2.5, 1.5])
>>> np.array([5,3]) + np.array([1,-1])
array([6, 2])
```

Réécrivons la fonction f2 :

```
import numpy as np

def f2(t,y):
    """reçoit un réel t et un np.array y (matrice ligne à 2
    coordonnées). Renvoie la dérivée de y sous forme d'un
    np.array (matrice ligne à 2 coordonnées)"""
    a,b = y        # récupération des coordonnées
    return np.array([2*a + b, -a + 2*b])
```

a. Résoudre numériquement le système précédent avec la fonction euler de la première séance et la fonction f2 ci-dessus.

Représenter graphiquement a et b sur $[0\,;2]$ avec $n = 100$.

On sauvegardera sous TP2_Euler_systeme.py.

b. (*) Une résolution plus précise [1]

Le module scipy.integrate propose une fonction odeint qui implémente une résolution beaucoup plus précise que la méthode d'Euler, basée sur la méthode RK4 (Runge-Kutta d'ordre 4). On rappelle que la méthode d'Euler est d'ordre 1 seulement (erreur environ proportionnelle à h^1)

Dans le programme précédent, importer la fonction odeint :

```
from scipy.integrate import odeint
```

puis tracer sur le même graphique les courbes approchées des fonctions a et b obtenues grâce à cette méthode.

La syntaxe d'appel est

```
y_RK4 = odeint(f,y0,Lt)
```

où Lt est la liste des abscisses, et f est une fonction renvoyant la dérivée comme précédemment.

Attention cependant, l'ordre des paramètres de cette fonction f est différent : la dérivée est donnée par f(y,t) !

1. à faire chez vous si le temps manque en séance

Il faut donc, pour l'appel, soit définir une autre fonction (h2, par exemple), soit utiliser une fonction anonyme (lambda) si on veut réutiliser notre fonction f2 :

```
y_RK4 = odeint(lambda y,t:f2(t,y),y0,Lt)
```

et y_RK4. n'est plus une liste mais un numpy.array.

2. **Équation différentielle du second ordre**

Soit une équation différentielle du second ordre, de la forme

$$y''(t) = g(t, y(t), y'(t))$$

où g est une fonction connue à 3 variables, avec une condition initiale y_0 sur y et y'_0 sur y'.

Dans le cas d'une équation à coefficient constants, on a une méthode de résolution exacte. Dans les autres cas, on peut construire une solution numérique approchée.

Hélas la méthode d'Euler ne permet que la résolution des équations différentielles d'ordre 1...

On peut cependant s'y ramener assez facilement, en posant

$U(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$, alors $U'(t) = \begin{pmatrix} y'(t) \\ y''(t) \end{pmatrix}$ et l'équation devient alors

$$U'(t) = \begin{pmatrix} y'(t) \\ g(t, y(t), y'(t)) \end{pmatrix} = f(t, U(t))$$

et on peut ainsi résoudre numériquement de la même façon que dans la partie précédente, y et y' jouant alors le rôle des a et b précédents.

a. Une équation simple pour commencer

Modifier le programme précédent pour résoudre numériquement l'équation différentielle

$$y'' = -y$$

sur l'intervalle $[0; 2\pi]$, avec la condition initiale $y(0) = 0$, $y'(0) = 1$. On pourra prendre $n = 100$ puis $n = 500$.

Remarque : ici on connaît la solution exacte : c'est la fonction sinus.

Enregistrer sous TP2_Euler_degre2.py.

b. Chute libre amortie (avec frottement)

On considère une bille de masse m, lâchée d'une hauteur h_0. On note z son altitude. Elle est soumise à la force de pesanteur terrestre et à une force de frottement fluide, opposée à la vitesse et de norme αv^2 ($v = \dot{z}$)

L'altitude $z(t)$ vérifie alors l'équation

$$m\ddot{z} = -mg + \alpha \dot{z}^2$$

— Réécrire cette équation sous la forme $U'(t) = f(t, U(t))$.

avec $U(t) = \begin{pmatrix} z(t) \\ \dot{z}(t) \end{pmatrix}$

(on pourra poser $v(t) = \dot{z}(t)$ pour simplifier les notations)

— Dans un nouveau fichier programme, écrire la fonction f2 du programme précédent afin qu'elle renvoie la dérivée $U'(t)$ (sous forme d'une matrice ligne de type numpy.array).

— Ecrire une fonction chute_amortie(h0,v0,dt) qui renverra les listes T,Z et V contenant les temps, altitudes et vitesses avant impact au sol ($z = 0$).

Il faut donc ici réécrire la fonction euler en l'adaptant au contexte : on ne fournit pas d'intervalle de temps $[a, b]$ puisque la date d'impact n'est pas connue, mais un quantum de temps dt qui joue le rôle du pas h précédent, et le calcul doit s'arrêter dès que $z \leqslant 0$.

— Dans le programme principal, définir les constantes m, α et g et tracer les courbes de z et de $\dot{z}$ grâce aux résultats renvoyés par cette fonction.

On pourra prendre par exemple

$m = 10$ kg, $g = 9.81$ m.s^{-2} et $\alpha = 0.05$ kg.m^{-1}, un pas de temps $dt = 0.1$ s, et considérer qu'on lâche la bille à une altitude de 300 m, avec une vitesse initiale nulle ; puis faire varier ces paramètres.

On doit constater que la vitesse tend à se stabiliser vers une valeur qu'on peut calculer à partir de l'équation différentielle (qu'obtient-on si $\ddot{z} = 0$?)

Remarque : ici on pouvait en réalité se contenter d'une application classique de la méthode d'Euler car l'équation est du premier ordre en $v = \dot{z}$ (puis en réappliquant Euler à v on peut obtenir z).

Ce ne serait pas le cas si un terme en z ou en t apparaissait dans l'équation différentielle.