I_{11} Résolution numérique d'équation différentielle : principe et méthodes simples

PCSI 2020 – 2021

Introduction : Nous avons vu lors d'un chapitre précédent comment calculer (numériquement) l'intégrale d'une fonction sur un segment. À partir de méthodes analogues, nous allons voir comment résoudre numériquement une équation différentielle (munie de condition initiale).

I Calcul d'une primitive par la méthode d'Euler

On va essayer de résoudre numériquement l'équation différentielle suivante : $F'(x) = f(x)$ avec pour condition initiale : $F(a) = F_a$. Il s'agit donc simplement de trouver la primitive de f valant F_a en a.

Par définition, $F'(x) = \lim\limits_{dx \to 0} \frac{F(x+dx)-F(x)}{dx}$, on peut donc écrire $F'(x) = \frac{F(x+dx)-F(x)}{dx} + \epsilon(dx)$ avec $\epsilon(dx)$ une fonction qui tend vers 0 en 0. Pour la méthode d'Euler, on va négliger ce terme en $\epsilon(dx)$. L'approximation sera donc d'autant meilleure que dx est petit.

On peut donc écrire :

$$F(x+dx) - F(x) = f(x)dx \Rightarrow F(x+dx) = F(x) + F'(x)dx$$

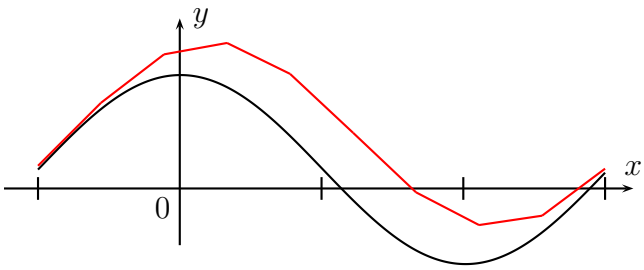
Si l'on connait un point de F, cette formule nous permet de déterminer approximativement un point « un peu plus loin ».

L'approximation étant d'autant meilleure que dx est petit, si on veut chercher un point « beaucoup plus loin », on va procéder par petits pas successifs.

Par exemple ci-dessous en cherchant la primitive [1] de $cos(x+1)$ qui vaut $0{,}2$ en -1.

L'idée de l'algorithme sera donc la suivante :

1. prendre un point de départ (la condition initiale) ;
2. calculer un point un peu plus loin ;
3. recommencer jusqu'à avoir atteint le point souhaité



1. Discrétisation

Puisque l'on ne résout qu'en un certain nombre de points, on dit que l'on discrétise le problème.

Ainsi, si l'on veut résoudre une équation différentielle avec une condition initiale en $x = a$ jusqu'à un point d'abscisse b et ce en ayant à la fin N points pour représenter la fonction, alors on va calculer une estimation de la fonction f en des points x_i équirépartis [2]. $x_i = a + i \times \delta x$ ($i \in [\![0, N-1]\!]$) où δx est l'écart entre deux points.

On a donc $\delta x = \frac{b-a}{N-1}$ (N points donc $N-1$ intervalles)

On va définir 2 tableaux de N points qui contiendront les valeurs de x et les approximations pour les $F(x)$ correspondants.

indices	0	1	2	...	N-1
$tabx$	$tabx[0] = a$	$tabx[1] = a + \delta x$	$tabx[2] = a + 2\delta x$	...	$tabx[N-1] = a + (N-1)\delta x = b$
$taby$	$taby[0] = F(a)$	$taby[1] \simeq F(a + \delta x)$	$taby[2] \simeq F(a + 2\delta x)$	...	$taby[N-1] \simeq F(b)$

1. La solution analytique est : $x \mapsto \sin(x+1) + 0{,}2$.
2. Il existe aussi des algorithmes qui adaptent leur pas en fonction des variations de la fonction, mais ils sont beaucoup plus compliqués.

(plus précisément, taby contient notre approximation pour $F(a+i\delta x)$)
La formule $F(x+\delta x) \simeq F(x) + f(x)\delta x \Leftrightarrow F(a+(i+1)\delta x) = F(a+i\delta x) + f(a+i\delta x)\delta x$ se traduit donc pour les tableaux par $taby[i+1] = taby[i] + \delta x \times f(tabx[i])$
Très important de savoir faire cela.

2. Algorithme

```
def primitive(f,a,b,N,Fa):
    # f une fonction dont on cherche la primitive
    # a le point où on connait la condition initiale Fa
    # b le point jusqu'où on veut connaitre la primitive
    # N le nombre de points où l'on va connaitre la primitive
    # Initialisation des variables et tableaux
    dx = (b-a)/(N-1)
    tabx =  [a + i*dx for i in range(N)]#ou np.linspace(a,b,N)
    taby = [0]*N # tableau vide de N points
    taby[0] = Fa # CI, en x = a, donc en position 0 dans le tableau
    #Boucle principale
    for i in range( N-1  ):#un point déjà calculé avec la C.I.
        taby[i+1] = taby[i] + dx*f(tabx[i])
    return tabx,taby #tabx permettra de tracer la courbe

def ma_fonction(x):
    return np.cos(x+1)

resultat = primitive(ma_fonction,-1,6,10,0.2)

plt.plot(resultat[0],resultat[1])
plt.show()
```

II Résolution d'une équation différentielle par la méthode d'Euler

On se propose d'utiliser la même méthode pour résoudre une équation différentielle. Prenons l'équation[3] $y' + 3y = 3$ avec pour condition initiale $y(0.5) = 0$.
En utilisant la même méthode que précédemment, proposer un algorithme permettant de résoudre cette équation différentielle sur l'intervalle [0.5,4] avec 500 points.

$$\frac{y(x+dx) - y(x)}{dx} + 3y(x) = 3 \Leftrightarrow y(x+dx) = y(x) + dx \times (3 - 3y(x))$$

```
N = 500
a,b = 0.5,4
dx = (b-a)/(N-1)
taby = [0]*N # allocation de la mémoire
for i in range(N-1):
    taby[i+1] = taby[i] + dx * ( 3-3*taby[i])
plt.plot(np.linspace(a, b, N), taby)
plt.show()
```

3. La solution analytique est ici connue : $x \mapsto 1 - e^{-3x}$

III Cas général pour une équation du premier ordre

On considère une équation différentielle de la forme $y'(x) = Eq(y(x),x)$. Par exemple $y'(x) + y^3(x) = cos(x)$ se mettrait sous la forme $y' = Eq(y(x),x)$ avec $Eq : (\alpha,\beta) \mapsto -\alpha^3 + cos(\beta)$
La méthode d'Euler donne de la même façon $y(x + dx) = y(x) + dx \times Eq(y(x),x)$

```
def resout_equa_diff(second_membre,a,b,N,ya):
    dx = (b-a)/(N-1)
    tabx = [a + i*dx for i in range(N)]
    taby = [0]*N # tableau vide de N points
    taby[0] = ya # condition initiale, en x = a, donc en position 0
       dans le tableau
    for i in range(N-1):#un point déjà calculé avec la C.I.
        taby[i+1] = taby[i] + dx*second_membre(taby[i],tabx[i])
    return tabx,taby #tabx permettra de tracer la courbe

def mon_equation(yi,xi):
    return -yi**3 + np.cos(xi)
    #correspond à l'équation différentielle y'(x) = -y(x)^3 + cos(x)

resultat = resout_equa_diff(mon_equation,0,6,100,0)
plt.figure()
plt.plot(resultat[0],resultat[1])
plt.show()
```

IV Résolution d'un système d'équation du premier ordre : exemple

On considère un modèle proie-prédateur modélisée par les équations de Lotka-Volterra :

$$\begin{cases} \dfrac{\mathrm{d}\, lievre(t)}{\mathrm{d}t} &= lievre(t) \times (\alpha - \beta \times lynx(t)) \\ \dfrac{\mathrm{d}\, lynx(t)}{\mathrm{d}t} &= -lynx(t) \times (\gamma - \delta \times lievre(t)) \end{cases}$$

On peut discrétiser de la même façon que précédemment,

$$\begin{cases} lievre(t + \mathrm{d}t) &= lievre(t) + \mathrm{d}t \times lievre(t) \times (\alpha - \beta \times lynx(t)) \\ lynx(t + \mathrm{d}t) &= lynx(t) - \mathrm{d}t \times lynx(t) \times (\gamma - \delta \times lievre(t)) \end{cases}$$

```
N, a, b = 1000, 0, 50
alpha, beta, gamma, delta = 0.67, 1.333, 1, 1
dx = (b-a)/(N-1)
tab_lievre = [0]*N
tab_lynx = [0]*N
tab_lievre[0], tab_lynx[0] = 0.5,1.2
for i in range(N-1):
    tab_lievre[i+1] = tab_lievre[i] + dx*tab_lievre[i]*(alpha-beta*tab_lynx[i])
    tab_lynx[i+1] = tab_lynx[i] - dx*tab_lynx[i]*(gamma-delta*tab_lievre[i])
x = np.linspace(a,b,N)
plt.figure()
plt.plot(x,tab_lievre)
plt.plot(x,tab_lynx)
plt.show()
```

V Résolution d'un système du premier ordre : cas général

Dans le cas général, on dispose de n équation avec n fonctions inconnues :

$$\left\{\begin{array}{rcl} y_0'(t) & = & f_0(y_0(t),y_1(t),\dots,y_{n-1}(t),t) \\ y_1(t)' & = & f_1((y_0(t),y_1(t),\dots,y_{n-1}(t),t) \\ & \dots & \\ y_{n-1}(t)' & = & f_{n-1}(y_0(t),y_1(t),\dots,y_{n-1}(t),t) \end{array}\right. \quad \Leftrightarrow \quad \begin{pmatrix} y_0' \\ y_1' \\ \vdots \\ y_{n-1}' \end{pmatrix} = F\left[\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}, t\right]$$

On présente ça sous la forme d'une fonction F qui prend en argument un tableau contenant $y_0, y_1,\dots, y_{n-1}$, et le temps t et renvoie un tableau contenant $y_0', y_1',\dots, y_{n-1}'$.
Par exemple pour le système

$$\left\{\begin{array}{rcl} v_x'(t) & = & \omega_0 v_y(t) \\ v_y'(t) & = & -\omega_0 v_x(t) \\ v_z'(t) & = & \cos(3t) \end{array}\right. \quad \Rightarrow \quad F = \left(\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}, t\right) \mapsto \begin{pmatrix} \omega_0 y_1 \\ -\omega_0 y_0 \\ \cos(3t) \end{pmatrix}$$

La connaissance de la fonction F suffit à définir l'équation différentielle. Soit si $\omega_0 = 2$, on codera la fonction F de la façon suivante :

```python
def F(tab,t):
    y0prime = 2 * tab[1]
    y1prime = -2 * tab[0]
    y2prime = np.cos(3*t)
    return y0prime, y1prime, y2prime
```

Le schéma d'intégration est ensuite le même que d'habitude :

$$\left\{\begin{array}{rcl} y_0(t+\mathrm{d}t) & = & y_0(t) + \mathrm{d}t \times f_0(y_0(t),y_1(t),\dots,y_{n-1}(t),t) \\ y_1(t+\mathrm{d}t) & = & y_1(t) + \mathrm{d}t \times f_1(y_0(t),y_1(t),\dots,y_{n-1}(t),t) \\ & \dots & \\ y_{n-1}(t+\mathrm{d}t) & = & y_{n-1}(t) + \mathrm{d}t \times f_{n-1}(y_0(t),y_1(t),\dots,y_{n-1}(t),t) \end{array}\right.$$

$$\Rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}(t+\mathrm{d}t) = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}(t) + \mathrm{d}t \times F\left[\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}(t), t\right]$$

```python
def resout_systeme_equaDiff(Eq,CI,a,b,N):
    dt = (b-a)/(N-1)
    t = np.linspace(a, b, N)
    solution = np.zeros((N,len(CI)))#autant de colonne que d'inconnues
    solution[0,:] = CI # conditions initiales
    for i in range(N-1):
        yprime = Eq(solution[i,:], t[i])#calcul les y' avec l'équa-diff
        solution[i+1,:] = solution[i,:] + dt * np.array(yprime)
        #même calcul qu'avant, mais sur tout le tableau
    return t,solution #renvoie t aussi pour le tracé

temps,resultat = resout_systeme_equaDiff(F,[1,0,0],0,10,1000)
plt.figure()
plt.plot(temps,resultat[:,0])#trace vx(t)
plt.show()
```

VI Un exemple de méthode plus évoluée (pour le TP)

Cette méthode n'est pas au programme, il ne s'agit donc que d'une illustration d'une méthode plus précise.

$$y''(x) \simeq \frac{y'(x+0{,}5dx) - y'(x-0{,}5dx)}{dx} = \frac{\frac{y(x+dx)-y(x)}{dx} - \frac{y(x)-y(x-dx)}{dx}}{dx} = \frac{y(x+dx) - 2y(x) + y(x-dx)}{dx^2}$$

Par exemple dans le cas du pendule simple dans le cas des grandes amplitudes, l'équation différentielle est $\ddot{\theta} + \omega_0^2 \sin\theta = 0$. On en déduit le schéma d'intégration suivant :

$$\theta(t+dt) = 2\theta(t) - \theta(t-dt) - \omega_0^2 \sin(\theta(t))dt^2$$

Et donc l'algorithme est le suivant :

initialiser les variables comme précédemment;
$\theta[1] \leftarrow \theta[0] + \dot{\theta}_0 \times dt - \frac{1}{2}\omega_0^2 \sin(\theta[0]) \times dt^2$;
(Le premier pas de temps est traité à part en utilisant un développement de Taylor car on ne dispose pas encore de θ au pas de temps précédent);
pour $i \leftarrow 1$ **à** $n-1$ **faire**
  $\theta[i+1] \leftarrow 2\theta[i] - \theta[i-1] - \omega_0^2 \sin(\theta[i]) \times dt^2$
fin

VII utilisation de scipy.integrate

Le sous module `integrate` du module `scipy` contient de nombreuses fonctions en lien avec l'intégration. En particulier la fonction `odeint` (INTegration d'Equations Différentielles Ordinaires permet de réaliser l'intégration d'un système d'équations différentielles du premier ordre.
Il faut donc systématiquement se ramener à un système d'équations du premier ordre même si au début vous n'avez qu'une équation d'ordre 2 ou plus.

Pour une équation d'ordre $n \geqslant 2$, la méthode usuelle est de rajouter des variables représentant les $n-1$ premières dérivées

$$\forall i \in [\![1,n-1]\!], f_i = f^{(i)},$$

puis d'écrire votre équation différentielle en fonction de ces variables.

Exemple : Prenons l'équation différentielles $y''(t) + \frac{\omega_0}{Q}y'(t) + \omega_0^2 y(t) = \omega_0^2 Y_0^2 \cos(\omega t)$. On rajoute une variable $y_1 = y'$ puis on essaye de présenter le système sous la forme suivante :

$$\begin{cases} y' &= F(y,y_1,t) \\ y_1' &= G(y,y_1,t) \end{cases} \text{ soit dans notre cas : } \begin{cases} y' &= y_1 \\ y_1' &= (y''=) - \frac{\omega_0}{Q}y_1(t) - \omega_0^2 y(t) + \omega_0^2 Y_0^2 \cos(\omega t) \end{cases}$$

Les $n-1$ premières équations sont donc faciles : il suffit d'écrire $f_{i-1}' = f_i$ puis la dernière équation ($f_n' = \dots$) est simplement votre équation différentielle en remplaçant $f^{(i)}$ par f_i et en isolant f_n' dans le membre de gauche.

Exercice : Présentez l'équation différentielles $y^{(3)} + y'' - 3y' + 2y^2 = e^{-t}$ sous forme d'un système d'équations différentielles du premier ordre.

$$\begin{cases} y' &= y_1 \\ y_1' &= y_2 \\ y_2' &= -y_2 + 3y_1 - 2y^2 + e^{-t} \end{cases}$$

Cette méthode n'est pas forcément la plus rapide, mais elle est systématique.

La fonction `odeint` prend trois arguments obligatoires :

1. une fonction `func(y,t`$_0$`,...)` qui permet de calculer les seconds membres (renvoyés sous forme d'une liste) et qui prend comme argument obligatoire une liste de n éléments et un temps.
2. les conditions initiales (liste de n conditions initiales)
3. un tableau de temps `t` qui correspond aux instants pour lesquels on veut que la fonction calcule les points.

Elle renvoie un tableau de taille $(\texttt{len}(t) \times n)$ qui contient pour chaque temps les valeurs de y et de ses $n-1$ premières dérivées.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def eq(Y,t):#y^(3) + y'' - 3y' + 2y = e^-t
    Z = [0]*3#va contenir notre résultat
    Z[0] = Y[1]#y' = y_1
    Z[1] = Y[2]#y_1' = y_2
    Z[2] = -Y[2] + 3*Y[1] - 2*Y[0]**2 + np.exp(-t)#y_2' = -y_2 + 3y_1 - 2y^2 + e^-t
    return Z
t = np.linspace(0,4,100) # création de la liste de temps
resultat = odeint(eq,[1,0,1],t) #appel de ODEINT
y = resultat[:,0]#première colonne, toutes les lignes
y_1 = resultat[:,1]
y_2 = resultat[:,2]
plt.plot(t,y)
plt.show()
```

Exercice : Écrivez un programme qui permet de résoudre l'équation différentielle à laquelle obéit le pendule simple à l'aide du solveur de scipy (Rappel : $\ddot{\theta} + \frac{\omega_0}{Q}\dot{\theta} + \omega_0^2 \sin\theta = 0$). Tracez ensuite l'évolution temporelle $\theta(t)$ puis la trajectoire de phase correspondante.

Table des matières