

QUELQUES MÉTHODES NUMÉRIQUES

Introduction

Ce cours est destiné à reprendre certaines méthodes numériques, en particulier celles de résolution d'équations différentielles (mais aussi le calcul d'intégrale).

Votre cours de mathématiques aborde la résolution sur un intervalle I des équations différentielles dites linéaires, c'est à dire de la forme

$$Y'(t) = A(t)Y(t) + B(t) .$$

Dans cette équation, B est une fonction de I dans $\mathcal{M}_{n,1}(\mathbb{R})$, A est une fonction de I dans $\mathcal{M}_n(\mathbb{R})$ et Y est la fonction inconnue, de I dans $\mathcal{M}_{n,1}(\mathbb{R})$.

Exemple :

$$\begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \begin{pmatrix} t & -t \\ 1-t^2 & t^3 \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} + \begin{pmatrix} 1 \\ -t \end{pmatrix} .$$

Sous certaines conditions (continuité de A et B) cette équation accompagnée de la condition ("initiale") $Y(t_0) = Y_0$, t_0 étant un élément de I , admet une unique solution. C'est le théorème de Cauchy-Lipschitz.

Ce type d'équations est un cas particulier d'équations du type : $Y'(t) = F(Y(t), t)$, où F est une fonction de $\mathbb{R}^n \times \mathbb{R}$ dans $\mathbb{R}^n$ (on a assimilé $\mathbb{R}^n$ à $\mathcal{M}_{n,1}(\mathbb{R})$). Exemple :

$$\begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \begin{pmatrix} t^2 x(t) + \sin y(t) \\ x(t)^3 \end{pmatrix} ; \text{ ici } F\left(\begin{pmatrix} x \\ y \end{pmatrix}, t\right) = \begin{pmatrix} t^2 x + \sin(y) \\ x^3 \end{pmatrix} .$$

On sait résoudre certaines de ces équations par l'usage de fonctions usuelles, mais dans de nombreuses situations issues de problèmes physiques, cette résolution "exacte" est impossible. On doit alors introduire des méthodes de résolutions dites "approchées" (ou numériques plutôt).

L'objet de ce cours est de rappeler certaines de ces méthodes (celle d'Euler essentiellement) et de les programmer.

1 Résolution numérique d'une équation différentielle

On l'applique tout d'abord à la résolution d'une équation scalaire $(E) : y'(t) = F(y(t), t)$ sur un intervalle du type $[t_0; t_0 + T]$, avec condition initiale $y(t_0) = y_0$.

1.1 Principe de la méthode d'Euler

La méthode d'Euler consiste à :

- subdiviser l'intervalle en n parties égales. On pose $h = T/n$, appelé pas ; et, pour $0 \leq k \leq n$, $t_k = t_0 + k h$.
- approcher, pour chaque k , $y(t_k)$ par une valeur y_k en partant de $k = 0$ (on a $y(t_0) = y_0$). Pour cela, on remarque que

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(t) dt .$$

Pour obtenir y_{k+1} à partir de y_k , on approche, sur l'intervalle $[t_k; t_{k+1}]$, $y'(t)$ par $y'(t_k)$, égal du fait de l'équation à $F(y(t_k), t_k)$, lui même approché par $F(y_k, t_k)$. Ceci nous donne la relation :

$$y_{k+1} - y_k = F(y_k, t_k) \times h .$$

1.2 Efficacité

La théorie montre que sous certaines hypothèses il existe une constante K telle que, pour tout k ,

$$|y(t_k) - y_k| \leq K |h| .$$

La méthode d'Euler est dite d'ordre 1. Par exemple, pour augmenter la précision d'un facteur 10, il faut diviser h par 10.

Dans ce cours d'informatique, on s'intéressera surtout à la complexité en temps et en espace. Il est clair que la fonction précédente a une complexité linéaire en la taille n de la subdivision.

1.3 Mise en œuvre

On souhaite appliquer la méthode d'Euler au calcul d'une solution approchée de l'équation différentielle $y' + y = \sin(t)$ sur l'intervalle $[0, 30]$ avec la condition initiale $y(0) = 1$.

1.3.1 Première version

On peut implémenter ce schéma de manière élémentaire en utilisant juste les listes python.

Compléter le programme suivant afin qu'il effectue le calcul et trace le graphique représentant la solution.

```
T = 30.0 ; t0 = 0 ; y0 = 1.0 ; n = 500
h = T/n

t = t0 ; liste_t = [t]
y = y0 ; liste_y = [y0]
for i in range(n):
    y = y + ...
    t = t + h
    # mise à jour des deux listes
    ...
    ...

plt.plot(liste_t , liste_y)
```

Remarque : on avait besoin d'avoir importé numpy et matplotlib.pyplot.

1.3.2 Utilisation de numpy

Dans ce cadre d'analyse numérique, on utilise la plupart du temps des tableaux numpy (un bref rappel est fait en annexe).

Dans la version qui suit, on introduit également le formalisme de l'utilisation de la fonction, appelée F dans l'introduction, qui gère le système.

Compléter le programme suivant afin qu'il effectue le calcul et trace le graphique représentant la solution.

```
def F1(y,t): #Fonction associée au système
    return ...

T=30.0 ; t0=0 ; y0=1.0 ; n=50 # données
h = T/n

temps = np.linspace(t0,t0+T,n+1)
Sol = np.zeros(n+1)
Sol[0] = ...
for i in range(...):
    ...
plt.plot(... , ...)
```

1.4 Utilisation de odeint

Le module `scipy.integrate` de python dispose d'une méthode d'intégration numérique des équations différentielles (plus efficace que la méthode d'Euler). Sur notre exemple, cela donne :

```
import scipy.integrate as integr

SolScipy = integr.odeint(F1,y0,temps)
plt.plot(temps,SolScipy)
```

1.5 Dimension supérieure

La méthode d'Euler peut être adaptée au cas où la fonction recherchée y est vectorielle : $y(t) = (y_1(t), \dots, y_m(t))$. La fonction F qui intervient dans l'équation différentielle $y'(t) = F(t, y(t))$ est alors définie sur $\mathbb{R} \times \mathbb{R}^m$ à valeurs dans $\mathbb{R}^m$. Le principe du schéma est alors le même.

Écrire une fonction `EulerDim2(t0,T,Y0,n,F)` dans le cas où la fonction inconnue est une fonction Y à valeurs dans $\mathbb{R}^2$. La fonction F prend en arguments un tableau de 2 éléments et un temps t et renvoie un tableau de deux éléments. La solution Y renvoyée sera un tableau à 2 lignes et $n + 1$ colonnes. Compléter :

```
def EulerDim2(t0,T,Y0,n,F):
    h=T/n
    temps = np.linspace(0 , t0+T , n+1)
    list_Y = ...
    list_Y[:,0] = ...
    for i in range(n):
        ...
    return list_Y
```

On va tester notre fonction sur le système suivant :

$$Y'(t) = AY(t) \quad \text{avec} \quad A = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}.$$

Compléter le programme suivant afin d'obtenir le tracer du diagramme de phase de la solution.

```
def F(Y,t):
    A = ...
    return ...

T=3.
t0=0
Y0=[0,1]

n=100
h=T/n

Sol = EulerDim2(t0,T,Y0,n,F)
temps = np.linspace(0 , t0+T , n+1)
X = ...
Y = ...
plt.plot(X , Y , 'r')
```

2 Équations d'ordre supérieur

On peut ramener une équation différentielle d'ordre supérieur à 1 à une équation d'ordre 1, d'inconnue une fonction vectorielle.

On explique cela sur l'exemple suivant : $y''(t) + 4y(t) = 0$, avec conditions initiales $y(0) = 0.5$ et $y'(0) = 0.1$.

1. En posant $Y(t) = (y(t), y'(t))$, écrire l'équation différentielle sous la forme $Y'(t) = F(t, Y(t))$.

2. Écrire la fonction $F(Y, t)$ associée au problème.

```
def F3(Y, t):
    return ...
```

3. Écrire une suite d'instruction permettant de tracer la solution obtenue avec $T = 10$, $n = 100$ (on pourra réutiliser une fonction précédemment écrite).

```
T = 10.
t0 = 0
Y0 = np.array([0.5, 0.1])
n = 100

Sol2 = ...

temps = np.linspace(0, t0+T, n+1)
listY = ...

plt.plot(temps, listY, 'r')
```

3 Calcul approché d'intégrale

3.1 Méthode des rectangles

1. Rappelons la formule de calcul approché de l'intégrale d'une fonction f sur un intervalle $[a; b]$ par la méthode des rectangles (à gauche) avec une subdivision en n intervalles :

$$R_n = \frac{b-a}{n} \sum_{k=0}^{n-1} f\left(a + k \frac{b-a}{n}\right).$$

Écrire une fonction `intRectangle(a,b,f,n)` qui renvoie cette approximation.

2. La méthode des trapèzes est donnée par :

$$T_n = \sum_{k=0}^{n-1} \frac{b-a}{n} \frac{f\left(a + k \frac{b-a}{n}\right) + f\left(a + (k+1) \frac{b-a}{n}\right)}{2}.$$

Écrire une fonction `intTrapeze(a,b,f,n)` qui renvoie cette approximation. On veillera à ne calculer qu'une seule fois la valeur de f en chaque point.

3.2 Utilisation de `scipy.integrate`

On obtient une valeur approchée de l'intégrale (ainsi qu'une majoration de l'erreur) par la commande

```
print(integr.quad(f,1,2))
```

3.3 Méthode de Monte-Carlo

Cette méthode consiste à calculer une intégrale en utilisant le hasard. Prenons un l'illustrer, on reprend le calcul de $\int_1^2 \frac{1}{x} dx$, c'est à dire de la surface de la partie A située entre l'axe des abscisses, la courbe de $x \mapsto 1/x$ et les abscisses 1 et 2.

Pour cela, on utilise une variable aléatoire X suivant une loi uniforme sur $[1, 2] \times [0, 1]$. La probabilité $P(X \in A)$ est égale à la surface recherchée. D'après la loi des grands nombres, si (X_n) est une suite de v.a.i.i.d. de même loi que X , la fréquence avec laquelle les X_n tombent dans A tend vers $P(X \in A)$. On va donc simuler une suite de telles variables aléatoires et estimer cette fréquence.

Rappelons qu'après `import` du module `random`, une simulation d'une variable aléatoire uniforme sur $[0, 1]$ s'obtient par la commande `random()`. On peut donc obtenir un couple selon la loi uniforme sur $[1, 2] \times [0, 1]$ par les commandes suivantes :

```
import random as rd
x , y = 1 + rd.random() , rd.random()
```

Écrire un programme estimant la surface de A en utilisant la méthode expliquée ci-dessus.

4 Annexe : quelques rappels sur NumPy

On fait juste quelques rappels sur la manipulation des tableaux. Essayer les commandes suivantes et observer leur résultat (Après avoir importé NumPy : `import numpy as np`).

- Pour déclarer un tableau : `A=np.array([[1,2],[0,3]])` ; on peut aussi initialiser avec des zéros : `np.zeros([2,3])` ; une matrice colonne : `B=np.array([[0],[1]])` ; un tableau à une dimension : `C=np.array([3,2])` .
- Pour accéder à un coefficient : `A[1,0]` ; on peut aussi l'affecter : `A[1,0]=5` .
- Pour accéder à une ligne : `A[1,:]` , à une colonne `A[:,0]` ; on peut aussi affecter une colonne : `A[:,0]=[6,7]` .
- Pour multiplier deux matrices : `np.dot(A,B)` . Attention : `A*B` ne fait pas du tout la même chose. On peut bien sûr additionner deux tableaux (de même format) et multiplier un tableau par une constante.
- Pour connaître les dimensions d'un tableau : `np.shape(A)` . Tester également avec `B` , `C` , `A[:,0]` .
- Bien remarquer le format de `A[:,0]` et le fait que `numpy` accepte de faire `np.dot(A,C)` .
- Pour obtenir une subdivision d'un intervalle (pour tracer une courbe par exemple) : `np.linspace(0,1,11)` (lorsque l'on connaît les bornes et le nombre de subdivisions) ou `np.arange(0,1.1,0.1)` (lorsque l'on connaît les bornes et le pas).