Résolution numérique de l'équation d'un pendue simple

Capacités exigibles

- Mettre en œuvre la méthode d'Euler à l'aide d'un langage de programmation pour simuler l'évolution d'un système à une variable.

I Documents

Document 1 : Équation du pendule simple

L'état du pendule simple est caractérisé par son angle θ qui évolue selon l'équation différentielle suivante :

$$\frac{\partial^2\theta}{\partial t^2} + {\omega_0}^2 \sin\theta = 0$$

Avec $\omega_0 = \sqrt{\frac{g}{l}}$ la pulsation des faibles oscillations.

Document 2 : Méthode d'Euler explicite pour une équation d'ordre 1

Lorsque l'on a une équation différentielle, il est parfois impossible d'exprimer des solutions générales. Cependant on peut en trouver une forme approchée à l'aide de méthode numérique.

Prenons l'exemple d'une équation différentielle du premier ordre quelconque pour une fonction $u(t)$. On peut la mettre sous la forme

$$\dot{u} = f(u)$$

Par exemple $\dot{u} = -ku$ pour une équation linéaire...

Discrétisation :
On va considérer que le temps est discret, et on note $\mathrm{d}t$ le pas d'évolution temporel. Ainsi, on pourra indicer les valeurs prises par u :

$$u(t) = u(n\cdot \mathrm{d}t) \underset{\text{discrétisation}}{\longrightarrow} u_n$$

Évolution :
On peut alors calculer l'état u_{n+1} à partir de l'état u_n. Pour cela, on peut écrire le développement limité de u :

$$u(t+\mathrm{d}t) = u(t) + \dot{u}\,\mathrm{d}t = u(t) + f(u)\,\mathrm{d}t$$

$$\downarrow \text{ discrétisation}$$

$$u_{n+1} = u_n + f(u_n)\,\mathrm{d}t$$

On choisit alors une situation initiale u_0, puis l'algorithme calcule tous les u_n de proche en proche. Ce qui nous donne au final l'évolution de $u(t)$!

⚠ Évidemment, plus $\mathrm{d}t$ est petit, plus la résolution sera précise. Par contre elle prendra plus de temps

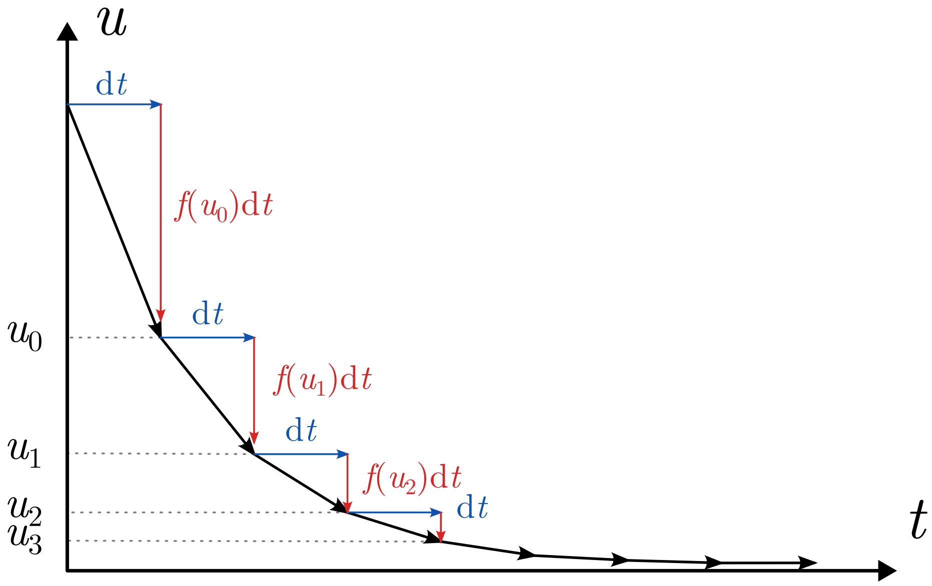

Figure 1 – Représentation schématique de la forme des solutions dans le cas où $f(u) = -ku$. On retrouve bien l'allure exponentielle attendue

Document 3 : Généralisation aux ordres supérieurs

Il est facile de généraliser la méthode d'Euler pour des équations différentielles aux ordres 2, 3, …

Prenons l'exemple d'une équation d'ordre 2. Cette fois-ci on peut la mettre sous la forme

$$\ddot{u} = f(u, \dot{u})$$

On pose alors le vecteur

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

Cette nouvelle grandeur suit elle-même une équation différentielle du premier ordre ! En effet, on a

$$\dot{U} = \begin{pmatrix} \dot{u} \\ \ddot{u} \end{pmatrix} = F(U) \qquad \text{avec} \quad F\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ f(x,y) \end{pmatrix}$$

On peut alors appliquer la méthode précédente à ce vecteur, ce qui nous donnera une suite contenant pour chaque instant la valeur de u et de sa dérivée $\dot{u}$. On retrouve ici le théorème de Cauchy-Lipschitz : il faut donner comme conditions initiales U_0, c'est-à-dire u_0 ET $\dot{u}_0$, pour calculer les états suivants de proche en proche.

II Énoncé

But

Programmer la résolution de l'équation différentielle du pendule simple.

1. Introduisez les paramètres de la simulation et donnez leur les valeurs que vous voulez (on les ajustera à la fin) :
   - `N` le nombre de points temporels de la simulation
   - `dt` le pas de temps
   - `omega` la pulsation du pendule

2. Codez la fonction `pendule(U)`, renvoyant la dérivée du vecteur $U = (\theta, \dot{\theta})$ sous forme d'un tableau `numpy`
3. Initialisez une matrice `Etats` avec la fonction `np.zeros`. Vous choisirez les dimensions de `Etats`, de sorte que la n^ième^ ligne stocke la valeur de U_n.

   Introduisez ensuite dans cette matrice l'état initial de votre choix, par exemple

$$\begin{cases} \theta(0) = \pi/2 \\ \dot{\theta}(0) = 0 \end{cases}$$

4. Codez alors l'algorithme de propagation, qui remplira toute la matrice `Etats`.
5. **Analyse :**
   a) Représentez l'allure de $\theta(t)$ pour différentes conditions initiales. Commentez l'influence de `dt`.
   b) Représentez le portrait de phase $\dot{\theta}(\theta)$ pour différentes conditions initiales. Comment peut-on voir simplement les limites de l'intégration par la méthode d'EULER ?

Pour aller plus loin :
À la place de vos lignes d'affichage, introduisez le bout de code suivant :

```
1  def func_anim(k):
2
3      [theta, v_theta] = Etats[int(k * vitesse_anim) % N]
4      line.set_data([?, ?], [?, ?])
5      return line,
6
7  vitesse_anim = 10   # Vitesse de l'animation. Ne change pas les calculs, mais vous pouvez l'ajuster pour voir une évolution satisfaisante
8
9  fig = plt.figure()
10 ax = fig.add_subplot(111)
11 ax.set_aspect('equal')  # Axes x et y représentés avec la même échelle
12 ax.set_xlim((-1.1, 1.1))
13 ax.set_ylim((-1.1, 1.1))
14
15 # line va contenir les coordonnées des points à afficher. On pourra les mettre à jour avec la fonction line.set_data
16 line, = ax.plot([], [])
17
18 anim = FuncAnimation(fig, func_anim, interval=10)
19 # interval représente le nombre de millisecondes à attendre entre deux frames
20
21 plt.show()
```

Et n'oubliez d'ajouter la ligne d'import au début de votre programme :

```
from matplotlib.animation import FuncAnimation
```

6. Recopiez ce code et complétez la ligne 4 en remplaçant les ? par ce qu'il faut, de sorte que l'animation trace en tout temps la tige du pendule (de l'origine à la masse)

III Annexe

Pour ce TP vous aurez besoin des modules numpy et pyplot :

```
import numpy as np
import matplotlib.pyplot as plt
```

Vous aurez ensuite à utiliser les fonctions suivantes :

```
np.array(L)
```

Transforme une liste classique en un tableau numpy. Il s'agit de l'objet de base sur lequel on travaille. Les opérations effectuées ont la même syntaxe que pour des flottants.

Exemple :

```
>>> x = np.array([1, 2, -1.5])
array([ 1. ,  2. , -1.5])
>>> x + 10
array([ 11. ,  12. , 8.5])
>>> 2 * x
array([ 2. ,  4. , -3.])
>>> x ** 2
array([ 1. ,  4. , 2.25.])
```

```
np.zeros(dim)
```

Créer un tableau remplis de 0, de dimension dim.

Exemple :

```
>>> x = np.zeros((2, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Vous pouvez ensuite changer les valeurs avec le slicing :

```
>>> x[0, 1] = -5     # Dans la première ligne, change la deuxième valeur
>>> x[1, :] = [1, 2, 3, 4]  # Les : signalent "toutes les valeurs"
>>> x
array([[0., 0., -5., 0.],
       [1., 2., 3., 4.]])
```

```
np.linspace(xi, xf, N)
```

Créer un tableau 1D allant de xi à xf en N points.

Exemple :

```
>>> np.linspace(0, 1, 11)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
np.sin(x), np.cos(x)
```

Calcule le sinus ou cosinus d'un tableau x (qui peut être de dimension (1,) donc être simplement un flottant).