Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 5

SS 2014

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2014-CompPhys.shtml
Abgabedatum: Montag, 12. Mai 2014 vor Beginn der Vorlesung

19. Einfache Animationen

Programmiertechniken

Oft ist es bei der Visualisierung des zeitlichen Ablaufs von physikalischen Prozessen hilfreich, diese als kleine **Animationen** darzustellen. In dieser Aufgabe wollen wir Ihnen einige elementare Animationstechniken in Python zeigen, welche Sie dann bei den übrigen Übungsaufgaben zum Einsatz bringen können.

In einem ersten Beispiel (siehe animation_simple.py) betrachten wir eine einfache Animation, welche das Zeichnen einer bunten Linie nach dem Prinzip des Daumenkinos visualisiert: Nach einer kurzen Pause wird das *gesamte* Bild durch eine neuere Version ersetzt. Im unserem konkreten Fall wird dabei jedes Mal ein weiterer Punkt der Linie zum neuen Bild hinzugefügt.

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

resolution = 80
pixels = np.zeros((resolution, resolution))

p = plt.imshow(pixels, interpolation='none')
p.set_cmap('spectral') # Andere nette Colormaps sind "hot", "gray", etc.
plt.clim(0.0, 1.0) # Setze das Intervall fuer die Farbwerte
plt.colorbar() # Zeichne eine Legende fuer die Farben

# Zeichne diagonale Linie von (0, 0) bis (resolution-1, resolution-1)
for i in range(resolution):
    pixels[i,i] = float(i)/resolution # Setze den Farbwert bei (i, i)
    p.set_data(pixels) # Erneuere die Plot-Daten
    plt.pause(0.05) # Pause in Sekunden

# Dieser show-Befehl sorgt dafuer, dass das Fenster am Ende
# der Programmausfuehrung nicht automatisch geschlossen wird.
plt.show()
```

Ein weiterer Animationseffekt, den Sie etwa bei der Visualisierung der Mondbahnen in Aufgabe 22 verwenden können, entsteht, wenn sie bei der Animation eines Punktes in jedem Schritt den Farbwert der bereits vorhandenen Punkte etwas verringern. Dazu können wir eine der speziellen Eigenschaften der NumPy-Listen verwenden – bei diesen können arithmetische Operationen auf

alle Listeneinträge simultan angewandt werden. Ein Beispiel geben wir Ihnen hier (siehe auch animation_tail.py) :

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

resolution = 80
# Hier erzeugen wir ein NumPy-Array gefuellt mit Nullen.
pixels = np.zeros((resolution, resolution))

p = plt.imshow(pixels, interpolation='none')
p.set_cmap('hot')
plt.clim(0.0, 1.0)
plt.colorbar()

x, y = -30, -30
dx, dy = 1, 0

for t in range(1,1000):
    pixels *= 0.9 # Hier wird jeder Eintrag der Liste
                  # mit 0.9 multipliziert. Dies funktioniert
                  # allerdings nur bei NumPy-Arrays.

    pixels[40+y,40+x] = 1.0 # Setze den neuen Punkt

    if t % 60 == 0: # Wir drehen die Richtung um 90 Grad
        dx, dy = -dy, dx

    x += dx
    y += dy

    p.set_data(pixels) # Erneuere die Plot-Daten
    plt.pause(0.001) # Pause in Sekunden

plt.show()
```

20. Die Zukunft richtig ausgependelt

5 Punkte

Das analytische Lösen von Differentialgleichungen gleicht oft einer Kunst für sich und erfordert nicht selten eine Reihe von Tricks und eine gute Portion Intuition, wenn überhaupt eine Lösung in geschlossener Form möglich ist. Zum Glück gibt es eine Menge numerischer Methoden, mit denen man fast beliebige Differentialgleichungen bearbeiten kann. Die einfachsten, aber dennoch auch in der Praxis häufig verwendeten Algorithmen wollen wir hier implementieren.

Wir studieren dazu ein **Pendel im Schwerefeld** der Erde. In Polarkoordinaten ist die Bewegungsgleichung gegeben als

$$\ddot{\phi}(t) = -\sin(\phi(t)),$$

wobei ϕ den Auslenkungswinkel des Pendels parametrisiert und wir die Masse m sowie die Gravitationskonstante g auf 1 gesetzt haben.

Die erste und einfachste Methode, die Sie in der Vorlesung kennengelernt haben, sind die **Euler-Methoden**. Implementieren Sie sowohl das *forward*- als auch das *backward*-Euler Verfahren und vergleichen Sie die Stabilität der Lösungen. Betrachten Sie dazu neben der Bahnkurve $\phi(t)$ auch die Energie und erklären Sie die Unterschiede.

Schon bei der numerischen Ableitung haben Sie gesehen, wie wichtig die Anzahl der einbezogenen Stützstellen für das korrekte Ergebnis ist. Diese Erkenntnis lässt sich auch auf das Lösen von Differentialgleichungen übertragen, wodurch man auf die Klasse der **Runge-Kutta** Verfahren stößt. Implementieren Sie nun auch das vierstufige Runge-Kutta-Verfahren Verfahren für die obige Bewegungsgleichung des Pendels.

Die Unterschiede zwischen Runge-Kutta und der stabilen Eulervariante werden für diese einfache Gleichung kaum sichtbar sein. Sobald es aber um gekoppelte Differentialgleichungen wie etwa in der ersten Aufgabe geht, kommt die Überlegenheit der Runge-Kutta-Methoden schnell zum Vorschein.

Hinweis

Die zu bearbeitende Differentialgleichung ist zweiter Ordnung, deswegen müssen wir sie umformen in ein System von Differentialgleichungen erster Ordnung. Allgemein geschieht dies so, dass wir die Ableitung n-ter Ordnung, bezeichent durch $y^{(n)}$, als Funktion der anderen Ordnungen auffassen:

$$y^{(n)} = f(x, y^{(1)}, \ldots, y^{(n-1)}) \tag{1}$$

Im nächsten Schritt führen wir dann Funktionen z_i ein, die gleich den $(i-1)$-ten Ableitungen gesetzt werden, also

$$\begin{aligned} z_1 &= y \\ z_2 &= y^{(1)} \\ &\ldots \\ z_n &= y^{(n-1)}. \end{aligned}$$

Die Ableitung von z_i ist so gleich z_{i+1}. Aus dem Vektor aller Ableitungen $\dot{\vec{z}} = (\dot{z_1}, \ldots, \dot{z_n})$, erhält man so ein System mit n gekoppelten Differentialgleichungen.

Für unser Pendel schreiben wir also $\ddot{\phi} = f(t, \phi, \dot{\phi}) = -\sin(\phi)$ und führen neue Funktionen z_1, z_2 ein:

$$\begin{aligned} z_1 &= \phi \\ z_2 &= \dot{\phi} \end{aligned}$$

Das zu bearbeitende System lautet also:

$$\begin{aligned} \dot{z_1} &= z_2 \\ \dot{z_2} &= \ddot{\phi} = -\sin(z_1) \end{aligned}$$

21. Monde auf der Überholspur *5 Punkte*

Die Monde **Janus und Epimetheus** umkreisen den Planeten Saturn *koorbital*, d.h. auf nahezu der gleichen Bahn. Bemerkenswerterweise ist dabei der Bahnunterschied zwischen dem inneren und dem äußeren der beiden Monde deutlich kleiner als deren Durchmesser. Alle vier Jahre begegnen sich die beiden Monde, da der innere Mond ein wenig schneller unterwegs ist als der äußere. Wieso es dabei nicht zu einer Kollision kommt, wollen wir in dieser Aufgabe untersuchen.

Dazu wollen wir das **3-Körper-System** aus Saturn, Janus und Epimetheus simulieren. Für die Simulation bewegen wir uns ins Bezugssystem des Saturn. Zwischen zwei Körpern wirkt dann eine Gravitationskraft gemäß

$$\mathbf{F}_{1\leftarrow 2} = -G\frac{m_1 m_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3}(\mathbf{r}_1 - \mathbf{r}_2), \tag{2}$$

wobei wir die Gravitationskonstante $G = 1$ setzen (wodurch sich die Größen der Parameter natürlich von den realistischen unterscheiden). Berechnen Sie die Bahnen der Monde mittels des **Verlet**-Verfahrens und plotten Sie die Bewegung der Monde als eine Animation. Verwenden Sie folgende Startwerte:

$$\begin{aligned} m_1 &= 1 \\ m_2 &= 4 \\ m_{\text{Saturn}} &= 4 \times 10^4 \\ \mathbf{r}_1 &= (-155, 0) \quad \frac{d\mathbf{r}_1}{dt} = (0, -16.1) \\ \mathbf{r}_2 &= (150, 0), \quad \frac{d\mathbf{r}_2}{dt} = (0, 16.3) \end{aligned}$$

Für die Animation sollten Sie den Ausschnitt $x \in (-200, 200), y \in (-200, 200)$ zeichnen. Geben Sie außerdem für jeden Zeitschritt den Abstand der Monde vom Planeten in eine Datei aus. Wenn Sie anschließend diese Abstände vergleichen, sollten sie deutlich sehen können, wie die Monde einer Kollision ausweichen. Erläutern Sie, wie dies von statten geht.

22. Dancing with the stars

optionale Aufgabe – 3 Punkte
Abgabe am Montag, 12. Mai

Nachdem wir in der letzten Aufgabe die Dynamik eines 3-Körper-Systems berechnet haben, wollen wir in dieser Aufgabe die ersten Schritte in Richtung eines ***n*-Körper-Systems** gehen und dabei noch mehr Sterne zum Tanzen bringen.

Implementieren Sie hierzu den **Verlet**-Algorithmus für ein System aus n Massen. Die Gravitationskonstante setzen wir wieder auf $G = 1$, und alle Massen seien dieses Mal $m_i = 1$.

Wir wollen drei verschiedene Systeme simulieren, deren Anfangswerte wie folgt gegeben seien:

- $n = 3$:

$$\begin{aligned} \vec{x}_1 &= (0.97000436, -0.24308753) & \vec{v}_1 &= (0.466203685, 0.43236573) \\ \vec{x}_2 &= -\vec{x}_1 & \vec{v}_2 &= \vec{v}_1 \\ \vec{x}_3 &= (0,0) & \vec{v}_3 &= -2\vec{v}_1 \end{aligned}$$

  Dieses System ist das einzige *stabile*, d.h. kleine Ungenauigkeiten in der Integration beeinflussen das Ergebnisse nicht stark und die Dynamik bleibt über lange Zeiten stabil. Die mathematisch Interessierten unter Ihnen finden die Herleitung dieses stabilen 3-Körper-Systems in **diesem Artikel** aus dem Jahre 2000.

- $n = 4$:

$$\begin{aligned} \vec{x}_1 &= (1.382857, 0) & \vec{v}_1 &= (0, 0.584873) \\ \vec{x}_2 &= (0, 0.157030) & \vec{v}_2 &= (1.871935, 0) \\ \vec{x}_3 &= -\vec{x}_1 & \vec{v}_3 &= -\vec{v}_1 \\ \vec{x}_4 &= -\vec{x}_2 & \vec{v}_4 &= -\vec{v}_2 \end{aligned}$$

- $n = 5$:

$$\begin{aligned} \vec{x}_1 &= (0.439775, -0.169717) & \vec{v}_1 &= (1.822785, 0.128248) \\ \vec{x}_2 &= (-1.268608, -0.267651) & \vec{v}_2 &= (1.271564, 0.168645) \\ \vec{x}_3 &= (-1.268608, 0.267651) & \vec{v}_3 &= (-1.271564, 0.168645) \\ \vec{x}_4 &= (0.439775, 0.169717) & \vec{v}_4 &= (-1.822785, 0.128248) \\ \vec{x}_5 &= -\sum_{i=1}^{4} \vec{x}_i & \vec{v}_5 &= -\sum_{i=1}^{4} \vec{v}_i \end{aligned}$$

  Im Gegensatz zum 3-Körper-System sind die beiden System mit 4, bzw. 5 Sternen nicht stabil – ihre Dynamik läuft nach langen Zeiten auseinander. Wählen Sie in Ihrer Simulation daher die Zeitschritte ausreichend klein, um die geschlossenen Bahnen lange genug zu erhalten!

Beschreiben Sie für all drei Systeme die sich ergebende Choreographie der Sterne.

23. Differentialgleichungen mit SciPy

Programmiertechniken

Mit dieser Aufgabe setzen wir unsere Tour durch das `SciPy`-Paket fort und erkunden die Möglichkeiten, **gewöhnliche Differentialgleichungen** zu lösen. Die dafür relevanten Funktionen finden sich im Untermodul `integrate`, das wir bereits auf dem vorherigen Übungsblatt zum Berechnen bestimmter Integrale verwendet haben.

Als Beispiel wollen wir heute einen **seltsamen Attraktor** berechnen. Ein solcher Attraktor ist der Endzustand eines **dynamischen Systems**, welcher eine Untermenge des zugehörigen Phasenraums definiert, auf die das System konvergiert und dann nicht mehr verlässt. Die "Seltsamkeit" besteht darin, dass diese Untermenge **fraktale Strukturen** besitzt. Der wohl bekannteste seltsame Attraktor ist der **Lorenz-Attraktor**, welchen der Meteorologe Edward Lorenz 1963 zur Beschreibung von Luftströmungen untersucht hat und damit das Forschungsgebiet der **Chaos-Theorie** gestartet hat.

Wir wollen hier den eng verwandten **Rössler-Attraktor** untersuchen und es Ihnen selbst überlassen, die unten stehenden Codes so anzupassen, dass Sie den Lorenz-Attraktor selbst berechnen können. Der Rössler-Attraktor definiert sich durch die folgenden Differentialgleichungen

$$\frac{dx}{dt} = -y - z \tag{3}$$

$$\frac{dy}{dt} = x + ay \tag{4}$$

$$\frac{dz}{dt} = b + z(x - c) \tag{5}$$

wobei wir als Parameter $a = 0.2, b = 0.2, c = 5.7$ wählen.

Dieses System von Gleichungen erster Ordnung möchten wir nun mithilfe des `integrate`-Moduls lösen. Der erste Schritt ist wie schon bei der Integration die Definition einer eigenen Funktion, welche in diesem Fall die Differentialgleichung implementiert:

```
def roessler(t, q):
    x, y, z = q

    a = 0.2
    b = 0.2
    c = 5.7

    f = [ - y - z, x + a * y, b + z * (x - c)]
```

Die Funktion berechnet also die rechte Seite des obigen Gleichungssystems und gibt die Werte als Liste zurück.

Diese Funktion wird nun an einen Algorithmus zur Integration übergeben, wobei uns dabei verschiedenste Integrationsmethoden zur Verfügung stehen. Mit der Funktion `odeint` steht eine gemeinsame Schnittstelle zur Verfügung, über die man anhand eines Parameters den gewünschten Algorithmus auswählen kann.

Dazu importieren wir `ode` aus dem `integrate`-Modul und initialisieren mit unser eben definierten Funktion. Wir legen dann außerdem noch die Anfangsbedingungen fest (`initial_conditions`) und wählen als Algorithmus beispielsweise eine Runge-Kutta Methode aus (hier `dopri5`):

```
from scipy.integrate import ode
```

```
initial_conditions = [-9., 0, 0]

solver = ode(roessler)
solver.set_initial_value(initial_conditions, 0.)
solver.set_integrator('dopri5')
```

Der letzte Schritt ist das Lösen der Differentialgleichung, welches iterativ passiert. Um die Zeitspanne unserer Lösungsmenge festzulegen, definieren wir eine Schrittweite `dt`, sowie einen Endzeitpunkt `t_final`. Die verwendeten Zeitwerte und die zugehörigen Funktionenwerte fügen wir sukzessive einer jeweils eigenen Liste hinzu:

```
dt = .01
t_final = 100

ts = []
values = []

while solver.successful() and solver.t < t_final:
    solver.integrate(t + dt)
    t.append(solver.t)
    values.append(solver.y)
```

Da der Rössler-Attraktor ein dreidimensionales Objekt ist, können wir die genommene Trajektorie im Phasenraum visualisieren, was mit folgendem Code passiert:

```
from numpy import array
from pylab import figure, show, xlabel, ylabel
from mpl_toolkits.mplot3d import Axes3D

fig = figure()
ax = Axes3D(fig)
ax.plot(values[:,0], values[:,1], values[:,2])
show()
```

Fügen Sie die oberen Ausschnitte so zusammen, dass Sie das Phasenbild des Attraktors erhalten. Sie können den Code nun leicht so modifizieren, dass Sie zum Beispiel auch den Lorenz-Attraktor untersuchen können.