Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 13

SS 2014

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2014-CompPhys.shtml
Abgabedatum: Montag, 14. Juli 2014 vor Beginn der Vorlesung

52. Nobody is perfect

Programmiertechniken

Messungen physikalischer Observablen sind immer fehlerbehaftet, ob Sie im Labor oder durch Simulationen erlangt werden. Um diese Fehler bei der Präsentation zu berücksichtigen, benutzt man wann immer möglich **Fehlerbalken**, die Sie sicherlich schon während der einführenden Experimentalphysikpraktika kennengelernt haben. In Python ist das Zeichnen von Fehlerbalken sehr einfach, was wir Ihnen mit dieser Aufgabe demonstrieren wollen.

Beginnen wollen wir jedoch damit zu erklären, wie man Ergebnisse auf der Festplatte speichern und wieder einlesen kann. Dazu werden wir das **CSV-Format** verwenden, welches gegenüber binären Formaten den Vorteil hat, direkt lesbar zu sein. Solche Formate werden auch als *human-readable* bezeichnet. Es gibt in Python zwar ein spezielles Modul csv, jedoch sind für unsere Zwecke die Methoden savetxt und loadtxt aus dem *numpy*-Modul besser geeignet, denn in der Regel erhalten wir Daten aus einem Programm, dass *numpy* verwendet, oder wir wollen die Daten in *numpy* weiter verarbeiten. Das Benutzen der oben genannten Funktionen ist denkbar einfach. Angenommen, wir wollen einen Plot speichern, der aus x- und y-Werten, sowie Fehlern auf den y-Werten besteht, die jeweils als ein Array vorliegen. Der erste Schritt ist es, die Daten zusammenzufügen mit der Funktion vstack, um sie dann im zweiten Schritt abspeichern zu können:

```
import numpy.random as npr

# generate random data
x = npr.random(10)
y = npr.random(10)
yerr = npr.random(10)

# join data vertically -v-stack to obtain a data matrix
# also try hstack and see what happens!
data = np.vstack((x, y, yerr)).T

# then save data
np.savetxt('data.csv', data, delimiter=',')
```

Um die Daten nun wieder zu laden, müssen wir nur ausführen:

```
data = loadtxt('data.csv', delimiter=',')
```

Die Daten liegen nach dem Einlesen als Matrix vor und wir können wir gewohnt mit der Slice-Notation auf die Spalten zugreifen.

In beiden Fällen haben wir das Schlüsselwort *delimiter* angegeben. So wählen wir das Zeichen aus, welches benutzt werden soll, um zwei Werte voneinandern zu trennen. CSV stand ursprünglich für *Comma Separated Values*, aber wird nun häufig auch als *Character Separated Values* ausgeschrieben, um die Freiheit in der Wahl des Trennzeichens zu berücksichtigen.

Nun möchten wir die eingelesenen Daten inklusive Fehlerbalken graphisch darstellen. Dazu verwenden wir die *matplotlib*-Methode *errorbar* und geben als Schlüsselwort *yerr* an:

```
plt.errorbar(x, y, yerr=yerr, fmt='o')
```

Als eine der vielen möglichen Optionen, zu denen auch Fehlerbalken in x-Richtung mittels *xerr* gehören, geben wir mit *fmt* an, dass nur Punkte gezeichnet werden sollen, die nicht mit Linien verbunden werden.

Ihre Aufgabe sei es, die Funktion

$$cos(kx) + 1$$

auf dem Intervall $x \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$ mittels einfacher Monte Carlo-Integration für verschiedene k-Werte zu integrieren, d.h. mit uniformer Wahrscheinlichkeitsverteilung. Wählen Sie dazu 100 verschiedene k-Werte aus dem Intervall $(0, 10)$ und berechnen Sie jeweils die Varianz (wie in der Vorlesung gezeigt) mit N=100 gezogenen Zufallszahlen. Speichern Sie Ihre Ergebnisse in einer CSV-Datei wie oben beschrieben und lesen Sie sie wieder ein. Plotten Sie dann das Ergebnis der Integration inklusiver Fehlerbalken gegen den Parameter k. Auf dem vorherigen Übungszettel haben Sie außerdem gelernt, die Funktion *quad* zu benutzen um beliebige Funktionen numerisch zu integrieren. Verwenden Sie diese, um Referenzwerte zu erhalten und plotten Sie diese ebenfalls in das gleiche Bild um zu sehen wie stark die Abweichung ist.

53. Durchhänger

5 Punkte

In dieser Aufgabe wollen wir erste Erfahrungen mit dem in der Vorlesung besprochenen **Metropolis-Algorithmus** sammeln und dazu das Durchhängen eines elastischen Seils numerisch simulieren.

Dazu erinnern wir uns zunächst an die Herleitung des Energiefunktionals für ein solches durchhängendes Seil, wie wir es in der analytischen Mechanik besprochen haben: Die potentielle Energie eines Seils entlang einer Seillinie $u(x)$ besteht aus zwei Beiträgen – neben einem Gravitationsbeitrag ist ein Beitrag zu berücksichtigen, der sich aus der internen Spannung des Seils ergibt. Wird ein Seilstück der Länge dx um eine vertikale Verschiebung du ausgelenkt, ergibt sich eine Änderung der internen Spannungsenergie, welche sich als Änderung der Seillänge multipliziert mit einer Elastizitätskonstante σ berechnen lässt

$$\delta V_S = \sigma \left(\sqrt{dx^2 + du^2} - dx \right) = \sigma dx \left(\sqrt{1 + (\partial_x u)^2} - 1 \right).$$

Wenn wir nun die Ableitung $\partial_x u(x)$ als klein annehmen, können wir die Wurzel Taylor-entwickeln

und finden in führender Ordnung

$$\delta V_s \approx \sigma dx \frac{(\partial_x u(x))^2}{2}.$$

Die Energie aus dem Gravitationspotential ist wie gewohnt

$$\delta V_G = mgu(x)dx\,.$$

Fügen wir beide Beiträge zusammen und integrieren über die Länge des Seils, ergibt sich also dessen Gesamtenergie als

$$V = \int_{x_0}^{x_1} \mathrm{d}x \left[\frac{\sigma}{2}(\partial_x u(x))^2 + mgu(x)\right]. \tag{1}$$

In dieser Aufgabe setzen wir $m = g = 1$, während wir die Elastizitätskonstante auf einen Wert von $\sigma = 3$ ansetzen wollen. Das Seil wollen wir zwischen $x_0 = -10$ und $x_N = 10$ spannen, d.h. $u(x_0) = u(x_N) = 0$.

Zur numerischen Simulation diskretisieren wir weiterhin das Seil, indem wir es als Konglomerat von $N = 81$ einzelnen Segmenten betrachten. Entsprechend müssen wir dann auch die Ableitung und das Integral in Gl. (1) diskret berechnen.

Implementieren Sie nun den Metropolis-Algorithmus, um die Ruhelage des Seils zu berechnen. Schlagen Sie dazu kleine vertikale Verschiebungen in einem Bereich von $du \in [-0.02, 0.02]$ vor und akzeptieren Sie diese mit einer Wahrscheinlichkeit

$$p_{\mathrm{acc}} = \min\left(1, e^{-\Delta V/T}\right).$$

Nehmen Sie dazu eine fiktive Temperatur von $T = 0.001$ an.

Sie sollten ausreichend viele Updates laufen lassen, wir schlagen $5000 \times N$ vor. Um zu sehen, wie der Algorithmus eine optimale Konfiguration findet, plotten Sie das Seil nach jedem 50. Sweep (also jeweils $50 \times N$ Updates).

Wir betrachten nun drei verschiedene Potentiale:

1. Zunächst lassen wir das Seil frei im Gravitationspotential hängen, d.h.

$$V(x) = g \cdot u(x). \tag{2}$$

2. Als zweites Potential haben wir einen **Schrank**, den wir zudecken wollen:

$$V(x) = \begin{cases} 10^{10} & \text{wenn } 1.5 \leq |x| \leq 6 \text{ und } u(x) < 2, \\ 10^{10} & \text{wenn } |x| \leq 1.5 \text{ und } u(x) < 3, \\ 10^{10} & \text{wenn } u(x) < 0, \\ g \cdot u(x) & \text{sonst.} \end{cases} \tag{3}$$

3. Im dritten Teil betrachten wir eine Hängebrücke, wie z.B. die **Golden Gate Bridge**

$$V(x) = \begin{cases} 10^{10} & \text{wenn } 5.5 \leq |x| \leq 6 \text{ und } u(x) < 3, \\ 10^{10} & \text{wenn } u(x) < 0, \\ g \cdot u(x) & \text{sonst.} \end{cases} \tag{4}$$

Plotten Sie für alle drei Potentiale das Seil, nachdem es eine Ruhelage gefunden hat.

54. Unterwegs mit Metropolis

5 Punkte

In dieser Aufgabe wollen wir den **Metropolis-Algorithmus** in einer antepassten Form für die Lösung eines analytisch nicht fassbaren **Optimierungsproblems** benutzen. Dieser abgewandelte Algorithmus wird **Simulated Annealing** genannt und in vielen Bereichen auf eben jene Lösung von Optimierungsproblemen angewandt.

Wir stellen uns dazu einen Kaufmann vor, der in Europa von Stadt zu Stadt reist, um seine Waren zu verkaufen. Er hat eine grosse Kundschaft, die quer über Europa verteilt ist, und muss daher viele Städte besuchen. Nun fragt er sich jedes Mal aufs Neue, in welcher *Reihenfolge* er die Städte besuchen sollte, damit er möglichst wenig reisen muss.

Wir versuchen nun, dem Kaufmann zu helfen und seine Route zu optimieren. Dazu übertragen wird das Problem auf die uns bekannte Form einer Energiefunktion über Mikrozuständen. Diese *Mikrozustände* sind bereits vollständig charakterisiert durch die *Reihenfolge* der Städte, die wiederum feste Koordinaten besitzen. Die *Energie* eines Mikrozustands entspricht dann der *Länge* des zurückzulegenden Weges über alle Städte in der festgelegten Reihenfolge.

Wir wissen, dass der Metropolis-Algorithmus uns Mikrozustände samplen kann, die zu einer gegebenen Temperatur passende Energie besitzen. Insbesondere wissen wir auch, dass die minimale Energie des Systems bei $T = 0$ angenommen wird. Die zentrale Idee ist nun, mittels eines lokalen Updates von einem Mikrozustand zu einem anderen von einer endlichen Temperatur das System abzukühlen und es damit in seinen Grundzustand mit minimaler Energie zu bringen, was einer Lösung des ursprünglichen Problems gleichkommt.

Gehen Sie dazu wie folgt vor:

1. Überlegen Sie sich (im Kopf oder auf einem Blatt Papier) ein geeignetes **lokales Update**, um von einem Mikrozustand in einen anderen zu gelangen.
2. Implementieren Sie eine Repräsentation der Mikrozustände, sowie das lokale Update
3. Implementieren Sie einen **Metropolis-Schritt** bei einer festen Temperatur T des Systems.
4. Überlegen Sie sich, welche *Temperaturen* welche Bedeutung im System haben und welche Temperaturen Sie für den Algorithmus benötigen, sofern am Anfang ein *chaotisches* und am Ende ein *geordnetes* Verhalten dominieren soll.
   Tip: Was ist die mittlere Energie des Systems bei einem chaotischen Verhalten und wie hängt die Temperatur damit zusammen?
5. Fügen Sie den Temperaturgradienten von Ihrer maximalen zu Ihrer minmalen Temperatur in Ihr Program ein

6. *Testen* Sie Ihr Program für eine Anzahl von 20 zufällig verteilten Städten in einem zweidimensionalen Bereich beliebiger Grösse, indem Sie das System in 100 Schritten von Ihrer maximalen zu Ihrer minimalen Temperatur bringen. *Plotten* Sie den optimierten Weg. Stimmt er mit Ihren Erwartungen überein?

Das Problem ist in der Informatik unter dem Namen **Travelling Salesman Problem** bekannt und eines der Standardbeispiele für ein Problem, das exakt nur in nicht-polynomieller Zeit gelöst werden kann. Was bedeutet dies für unseren Metropolis-Ansatz?