Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 13

SS 2013

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2013-CompPhys.html
Abgabedatum: Optionale Aufgaben

Auf diesem **optionalen Übungsblatt** geben wir einen kleinen Ausblick darauf, was Sie in der Master-Vorlesung *Computational Many-Body Physics* erwarten wird. In den Programmiertechniken lernen Sie, wie Sie es schaffen, auch mit Python komplexe und dennoch schnelle Codes zu schreiben. Danach untersuchen wir mit der sogenannten Perkolation nach dem Ising-Modell ein weiteres kritisches Phänomen.

49. Rennschlange Cython

Programmiertechniken

Das Semester neigt sich mittlerweile dem Ende entgegen und Sie haben einige Erfahrungen mit Python gesammelt. Einer der großen Vorteile gegenüber anderen Sprachen ist die unkomplizierte Syntax, welche die Entwicklungszeit deutlich verkürzen kann. Andererseits ist Python in vielen Situationen eher langsam, was es gerade für Aufgaben im wissenschaftlichen Bereich ungeeignet zu machen scheint. Dieses Problem lässt sich jedoch in einem gewissen Rahmen in den Griff bekommen, indem man Python mit C verbindet. Diese Verbindung nennt sich **Cython** und wir wollen sie in dieser Aufgabe kurz vorstellen. Die zu erwartenden Laufzeitverbesserungen bringen Cython-Codes damit oftmals nah an die Qualitäten nativer C-Programme. Diese Kombination aus geringerem Programmieraufwand und akzeptablen Laufzeiten erlaubt es daher durchaus, ein wissenschaftliches Projekt in Cython statt in reinem C aufzuziehen.

Das Cython-Modul können Sie über den *Enthought* bzw. *Canopy* Paketmanager installieren. Grundsätzlich ist es möglich, *jeden* Pythoncode mit Cython zu übersetzen, jedoch muss man eigentlich immer kleine Modifikationen vornehmen um den gewünschten Geschwindigkeitszuwachs zu erreichen. Beginnen wir mit dem unkritischen Standardbeispiel *Hello World*. Wir öffnen eine Datei, *helloworld.pyx* (beachten Sie die Endung *.pyx*) mit dem Inhalt:

```
print 'Hello World'
```

Diese Datei wollen wir in eine C-Bibliothek kompilieren, wozu wir noch eine weitere Datei benötigen. Diese Datei heißt meist *setup.py* und sieht folgendermaßen aus:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension(''helloworld'', [''helloworld.pyx''])]
)
```

Die ersten drei Zeilen importieren benötigte Module und sollen nicht weiter erläutert werden. Dann wird eine Pythonklasse *setup* initialisiert, und als Parameter ein Befehl übergeben, nämlich *build_ext*, was bedeutet, dass ein externes Modul erzeugt werden soll. Dann legen wir mit *ext_modules* fest, dass wir die Datei *helloworld.pyx* verwenden wollen, um ein Modul mit Namen *helloworld* zu erzeugen. Um das Modul zu erzeugen, rufen wir auf:

```
python setup.py build_ext --inplace
```

Der letzte Parameter *inplace* bewirkt, dass das Modul im aktuellen Verzeichnis erzeugt wird.

Der letzte Schritt ist, das Modul zu verwenden. Dazu erzeugen wir eine neue Datei, etwa *helloworld_test.py*:

```
import helloworld
```

Wenn Sie diese nun ausführen mit

```
python helloworld_test.py
```

werden Sie wie erwartet die Ausgabe *Hello World* auf dem Bildschirm sehen. Wir widmen uns nun der Frage, welche Modifikationen man vornehmen muss, um ein bestehendes Python-Programm mittels Cython zu beschleunigen.

Deklaration von Variablen

In Python mussten Sie nur selten explizit den Typ einer Variablen angeben. Um eine Variable vom C-Teil des Codes aus verwenden zu können, müssen Sie dies aber tun. Dazu benutzen wir das Schlüsselwort *cdef* um eine *Def*inition in *C* zu kennzeichnen und darauffolgend den Typ der Variable:

```
cdef int test_int = 5
cdef double test_double = 3.14
```

Besonders nützlich waren für uns die Klassen aus der *numpy*-Bibliothek. Um diese zu benutzen und zu beschleunigen, müssen wir sie zunächst deklarieren und erst dann einen Wert zuweisen, wie wir es aus einem normalen Pythonprogramm kennen. Beachten Sie die Zeile mit *cimport*. Diese ist notwendig um *Cython* mitzuteilen, dass es in der *numpy*-Bibliothek Informationen über Optimierungen des Codes gibt.

```
import numpy as np
cimport numpy as np

cdef np.ndarray[np.double_t, ndim=2] data
data = np.zeros((6, 4), dtype=np.double)
```

Folgende Punkte sind zu beachten: In der C-Definition der Variable müssen wir mindestens den Datentyp und die Dimension des Arrays angeben. Für unsere Anwendungen ist es ausreichen zu wissen, dass es die Typen *np.double_t* und *np.int_t* gibt. In der Initialisierung müssen wir erneut den Typ angeben, dieses mal jedoch ohne das *_t* am Ende. Diese spezielle Notation ist wohl noch eine historisch gewachsene Eigenschaft der Module, die uns nicht weiter beschäftigen soll.

Funktionen

Ein wichtiger Bestandteil komplexerer Programme sind Funktionen. Um eine Funktion in C-Code auszulagern, müssen wir auch diese zunächst mit *cdef* umdefinieren. Es gibt allerdings ein wichtiges Detail zu beachten: Um auch aus dem Pythonteil eines Programms auf eine Funktion zugreifen zu können, müssen wir sie mit *c***p***def* deklarieren. Weißman vorher, dass eine Funktion sowieso nur aus dem C-Teil des Codes aufgerufen wird, kann man den Code auf diesem Weg schlanker halten. Parameter können entweder als Python- oder als C-Variablen übergeben werden und auch eine gemischte Notation ist möglich:

```
# function accessible from C and Python code, takes mixed arguments
cpdef some_function(a, b, double c):
[...]

# function only accessible from C, takes numpy argument
cdef other_function(np.ndarray[np.int_t, ndim=2] some_array):
[...]
```

C-Funktionen

Als letzten Punkt wollen wir noch kurz erwähnen, wie sich beliebige C-Bibliotheken in Python benutzen lassen. Dies macht zum Beispiel Sinn bei mathematischen Funktionen, die für C-Programme in der Datei *math.h* deklariert sind. Wir benutzen dafür das Schlüsselwort *extern*:

```
cdef extern from ‘‘math.h’’:
    double exp(double x)
```

Von nun an können wir die Funktion *exp* mit einer Variable des Typs *double* benutzen, und erhalten eine Variable gleichen Typs zurück. Alle diese Schritte finden auf der C-Ebene des Programms statt.

Anwendung

Mittlerweile haben Sie vielleicht schon den Code für das Ising-Modell fertig geschrieben. Auf der Website stellen wir Ihnen in der Datei **ising_cython.tar.gz** das gleiche Grundgerüst wie zuvor, aber nun im Cython-Stil zur Verfügung. Ihren Update- und Messmechanismus können Sie einfach kopieren. Führen Sie dann die Datei *ising_cython_skeleton.py* aus und vergleichen Sie die Geschwindigkeit mit den reinen Python-Programm aus der vergangenen Übung.

50. Perkolation – eine löchrige Sache *12 Bonus-Punkte*

Die **Perkolationstheorie** befasst sich mit dem Phänomen der **Clusterbildung**. Als Cluster werden in diesem Zusammenhang zusammenhängende Gebiete auf zufällig besetzten Gittern bezeichnet. Der Perkolationsübergang – von vielen kleinen nicht zusammenhängenden Clustern zu einem großen, das System spannende Cluster – ist dabei eine der einfachsten Inkarnationen eines **kontinuierlichen Phasenübergangs**.

In dieser Aufgabe wollen wir eben diesen Phasenübergang studieren. Nachdem wir ein mit einer gegebenen Wahrscheinlichkeit zufällig besetztes Gitter erzeugt haben, ist das zentrale Element dieser Untersuchung ein effizienter Algorithmus, der es erlaubt, die diversen Cluster in einer solchen zufälligen Konfiguration zu identifizieren. Dazu können wir auf einen in der Informatik wohl bekannten Algorithmus zurückgreifen, den **union-find** Algorithmus zur Berechnung der Äquivalenzklassen einer Menge. Die zugehörige Äquivalenzrelation ist dabei gegeben durch die Identifizierung zweier gleichbesetzter Gitterpositionen, die über einen Pfad von genau so besetzten Gitternpositionen verbunden sind. In der statistischen Physik firmiert dieser Algorithmus auch unter dem Namen **Hoshen-Kopelman** Algorithmus.

Hier also das Perkolationsproblem im Detail: Man nehme ein Quadratgitter und besetze die Quadrate (Positionen) mit der Wahrscheinlichkeit p mit einer 1 und mit der Wahrscheinlichkeit $1-p$ mit einer 0. Zusammenhängende Bereiche von 1'en werden dann als Cluster bezeichnet. Deren Identifizierung anhand des Hoshen-Kopelman Algorithmus funktioniert wie folgt: Beginnend am oberen, linken Rand scannen wir das Gitter zeilenweise von links oben nach rechts unten. Jeder Position ordnen wir bei diesem Scan ein Label zu, d.h. eine natürliche Zahl, welche den jeweiligen Cluster identifizieren soll. Besitzt eine (besetzte) Position noch kein Label, dann ordnen wir ihr ein neues, unbenutztes Label zu. Direkt benachbarte, besetzte Positionen erhalten das gleiche Label. Wichtig ist nun folgender Fall – wenn an eine besetzte Position zwei Cluster grenzen, die *nicht* das gleiche Label besitzen (überlegen Sie, wann dies auftreten kann), so entscheiden wir uns für das kleinere der beiden. In diesem Fall müssen wir zusätzlich abspeichern, dass es darüberhinaus eine Verbindung zwischen den beiden Clustern mit unterschiedlichen Labeln gibt. Dass sie unterschiedliche Label besitzen ist dabei nur ein Artefakt des Algorithmus; tatsächlich ist es natürlich der gleiche Cluster. Dieser Prozess ist in Abbildung 1 dargestellt.

Nachdem Sie einmal durch das Gitter gescannt haben und jeder besetzten Position ein Label zugeordnet haben, scannen Sie noch einmal und wenden die abgespeicherten Verbindungsinformation zwischen Clustern an, um die Nummerierung zu vervollständigen. Sie sehen, dass wir mit diesem Algorithmus für ein Gitter mit N Positionen in $O(N)$ Schritten die Cluster identifizieren können.

Ihre Aufgabe sei es nun, den obigen Hoshen-Kopelman Algorithmus zu implementieren. Wir stellen Ihnen wie beim Ising-Code in der Datei **percolation_skeleton.py** ein Gerüst zur Verfügung, in welches Sie diesen Algorithmus einbauen sollen. Alternativ zur regulären Pythonvariante, können Sie diese Aufgabe auch mit Hilfe von **Cython** lösen. Stellen Sie die Cluster-Konfigurationen für verschiedene Wahrscheinlichkeiten p graphisch dar – dabei können Sie etwa die Clustergrösse farblich codieren.

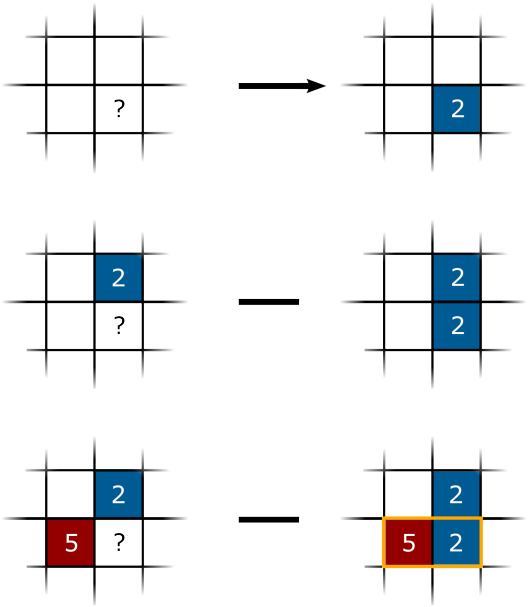

Abbildung 1: Regeln zur Clusterbildung im Hoshen-Kopelman Algorithmus: Grenzt an eine besetzte Position kein anderer Cluster, so wird die Position mit einem neuen Label versehen (oben). Grenzt nur ein Cluster an die Site, so wird das gleiche Label übernommen (mitte). Grenzen zwei Cluster an die gleiche Site, so wird das kleinere der beiden Labels verwendet und die beiden Cluster verbunden (unten).

Interessante Messgrößen, die Sie studieren können, sind die Anzahl der gebildeten Cluster und die Wahrscheinlichkeit, dass sich ein so genannter perkolierender Cluster gebildet hat. Ein perkolierender Cluster erstreckt sich von links nach rechts oder von oben nach unten, also einmal quer durch das gesamte Gitter.

Probieren Sie dabei selbst, wie groß Sie das System wählen können, um in vertretbarer Zeit Resultate zu erhalten. Tragen Sie dann die eben genannten Messgrößen gegen die Wahrscheinlichkeit p eine besetzte Position mit einer 1 zu initialisieren, auf. Vergleichen Sie die Plots mit denen des Ising-Modells – insbesondere die Wahrscheinlichkeit, einen perkolierenden Cluster zu finden, mit der Magnetisierung als Funktion der Temperatur im Ising-Modell.