Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 9

SS 2013

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2013-CompPhys.html
Abgabedatum: Montag, 24. Juni 2013 vor Beginn der Vorlesung

32. Geschwindigkeitskontrolle

Programmiertechniken

Nachdem die erste Programmierhürde genommen ist und ein Programmcode das macht, was er soll, stellt sich häufig die Frage, ob der Code die gleiche Aufgabe nicht auch viel schneller erledigen könnte. Allgemeine Tipps zur **Code-Optimierung** sind rar. Deshalb ist es oft nützlich, verschiedene Varianten zu implementieren und miteinander zu vergleichen. In Python wird dies ermöglicht durch das Paket timeit, welches eine Funktion mehrfach ausführt und die mittlere Rechenzeit zurückgibt.

Um die Funktionalität des Pakets zu testen, wollen wir auf verschiedenen Wegen zwei Matrizen addieren und die dazu benötigte Zeit stoppen. Laden Sie sich dazu die Datei timing.py herunter und starten Sie als erstes das Programm, denn es wird etwa fünf bis zehn Minuten dauern bis ein Resultat ausgegeben wird. Analysieren Sie dann den ausgeführten Code und überlegen Sie was passiert. Erwarten Sie Geschwindigkeitsunterschiede und falls ja, warum? Gibt es andere Operationen aus der linearen Algebra bei denen eine der implementierten Funktionen die klügere Wahl ist?

33. Eigenwerte im Radar

5 Punkte

In der Vorlesung haben wir die **Jacobi-Methode** zur Bestimmung von **Eigenwerten** einer symmetrischen $N \times N$ Matrix ausführlich besprochen. Diese wollen wir in dieser Aufgabe implementieren und die **Ordnung des Algorithmus** experimentell bestimmen. Wir stellen Ihnen darüberhinaus den Quellcode einer anderen Methode, der sogenannten **QR-Zerlegung** zur Verfügung, so daß Sie diese mit der Jacobi-Methode vergleichen können. Dies wollen wir für quadratische, reelle Matrizen mit den Größen 3×3, 4×4, ... bis 15×15 tun. Diese können Sie mit Zufallszahlen füllen, so wie Sie es bereits auf dem vergangenen Übungsblatt gelernt haben. Da die Jacobi-Methode nur für symmetrische Matrizen gültig ist, muss die Eingabematrix zunächst symmetrisch gemacht werden. Überlegen Sie sich zunächst wie Sie dies geschickt sicherstellen können. Implementieren Sie dann die Jacobi-Methode und vergleichen Sie Ihre Ergebnisse mit denen der von uns bereitgestellten QR-Methode. Dazu kopieren Sie die Datei compphys_qr.py in das gleiche Verzeichnis, in dem auch ihr Code liegt und rufen sie folgendermaßen auf:

```
import compphys_qr as cqr
```

```
import numpy.random as npr

M = npr.random((6, 6))
M = M + M.transpose()
evals, evecs = cqr.qr_eigen(M)
```

Mithilfe der in der ersten Aufgabe dieses Blatts vorgestellten Methode *time* können Sie die Zeit stoppen und gegen die Anzahl der Zeilen der Eingabematrix plotten. Bestimmen Sie aus den Daten die Ordnungen der beiden Algorithmen.

Kurzeinleitung zum Begriff der Ordnung eines Algorithmus

Die Geschwindigkeit eines Algorithmus wird mit der sogenannten **Big O**-Notation beschrieben und auch als **Ordnung des Algorithmus** bezeichnet. Wenn n ein Paramter des Algorithmus ist, wie z.B. die Anzahl der Stützstellen bei einer Integration oder die Dimension einer Matrix, dann schreibt man $\mathcal{O}(f(n))$, wobei $f(n)$ die Abhängigkeit der Geschwindigkeit vom Parameter n beschreibt. Diese ist oft polynomial, d.h. $\mathcal{O}(n^a)$, kann aber auch kompliziertere Formen wie $\mathcal{O}(n\ln(n))$ annehmen.

Wir wollen dies an einfachen Beispielen konkret machen. Beginnen wir mit $\mathcal{O}(1)$, wo es keine Abhngigkeit von einem Parameter gibt. Eine Funktion die zwei Werte (Arrays ausgeschlossen) miteinander vergleicht, besitzt zum Beispiel diese Ordnung:

```
def order_zero(a, b):
    return a == b
```

Interessanter sind Algorithmen, die Parameter variabler Größe akzeptieren. Betrachten wir die gleiche Funktion, aber nehmen wir an, dass die Parameter nun zwei Arrays sind, die wir elementweise miteinander vergleichen wollen:

```
def order_one(a, b):
    for i in range(len(a)):
        if a[i] not b[i]:
            return False
    return True
```

Sie können sich leicht überlegen, dass die Laufzeit nun linear von der Anzahl der Elemente in den Arrays abhängt. Die Tatsache, dass es eine *for*-Schleife gibt, ist übrigens auch ein guter Hinweise darauf. Diesen Algorithmus würden wir also nun mit Ordnung $\mathcal{O}(n)$ angeben. Eine Feinheit dieser Notation lässt sich auch an diesem Beispiel erkennen: Es mag durchaus sein, dass sich die beiden Arrays schon im ersten Element unterscheiden und die Iteration beendet ist. Genauso gut kann es aber auch erst das letzte Element sein und die Laufzeit wäre tatsächlich linear abhängig von der Arrayröße n. Die Big-O Notation beschreibt also ein **worst-case**-Szenario, also die maximale Laufzeit des Algorithmus.

Ein Beispiel für einen Algorithmus der Ordnung $\mathcal{O}(n^2)$ ist die Matrixaddition quadratischer Matrizen, den Sie in der vorherigen Aufgabe gesehen haben:

```
def order_two(a, b):
    m, n = a.shape
    for j in range(m):
        for i in range(m):
            a[i, j] += b[i, j]
```

Die durchgeführte Operation ist die Addition `a[i, j] += b[i, j]`. Diese wird aber innerhalb zweier verschachtelter Schleifen aufgerufen, die jeweils über n Elemente iterieren. Somit werden $n \cdot n = n^2$ Additionen ausgeführt und der Algorithmus erhält die Ordnung $\mathcal{O}(n^2)$.

34. Anharmonischer Oszillator

5 Punkte

Nachdem wir kürzlich den harmonischen Oszillation untersucht haben, wollen wir uns in dieser Aufgabe dem **anharmonischen Oszillator** zuwenden. Letzterer sei durch den folgenden Hamilton-Operator (mit $m = \omega = \hbar = 1$) beschrieben

$$\hat{H} = \frac{1}{2}\left(\hat{p}^2 + \hat{x}^2 + \lambda \hat{x}^4\right), \tag{1}$$

wobei der zusätzliche quartische Term $\hat{x}^4$ mit positivem Vorfaktor $\lambda \in \mathbb{R}^+$ auftritt.

Wir wollen nun die Eigenzustände und die dazugehörigen Energien dieses anharmonischen Oszillators (1) berechnen und dazu das in der Vorlesung kurz vorgestellte **variationelle Verfahren** einsetzen. Um die Eigenzustände variationell zu beschreiben wollen wir dabei auf die ersten N Eigenzustände $\{|n\rangle\}$ des *harmonischen* Oszillators zurückgreifen.

Implementieren Sie das variationelle Verfahren, indem Sie die Matrix des Hamilton-Operators in der endlichen Basis der ersten N Eigenzustände des harmonischen Oszillators aufstellen, siehe dazu auch die folgenden Hinweise. Wählen Sie $N = 5$, $N = 10$ und $N = 20$ mit $\lambda = 0.1$. Geben Sie jeweils die Energien des Grundzustands und der ersten beide angeregten Zustände aus und plotten Sie die Wellenfunktionen. Wiederholen Sie diese Rechnungen nun für $\lambda = 0.5$ und $\lambda = 2$. Beschreiben Sie, wie sich die Energien verhalten.

Einige Hinweise:

- Hilfreich ist auch hier wieder die Einführung der *Auf-* und *Absteigeoperatoren* $a^\dagger$ und a, die schon bei der Lösung des harmonischen Oszillators verwendet wurden. Diese sind definiert durch

$$a^\dagger |n\rangle = \sqrt{n+1}\,|n+1\rangle$$
$$a\,|n\rangle = \sqrt{n}\,|n-1\rangle$$

  und gehorchen bosonischen *Kommutatorrelationen*

$$[a, a^\dagger] = aa^\dagger - a^\dagger a = 1, \quad [a,a] = [a^\dagger, a^\dagger] = 0.$$

  Orts- und Impulsoperator können damit wie folgt dargestellt werden

$$x = \frac{1}{\sqrt{2}}(a^\dagger + a)$$
$$p = \frac{i}{\sqrt{2}}(a^\dagger - a).$$

- Schreiben Sie zunächst den Hamiltonian aus Gl. (1) unter Verwendung der Auf- und Absteiger $a^\dagger$ und a. Rechnen Sie dann die Matrixdarstellung der Operatoren a und $a^\dagger$ aus. Damit können Sie schließlich die komplette Hamiltonmatrix aufstellen. Zur Erinnerung: Die Matrixelemente der Auf- und Absteiger sind

$$a^\dagger_{nm} = \langle n|\, a^\dagger\, |m\rangle = \sqrt{m+1}\,\delta_{n,m+1}$$
$$a_{nm} = \langle n|\, a\, |m\rangle = \sqrt{m}\,\delta_{n,m-1}.$$

Wenn Sie die Wellenfunktionen in der Ortsraumdarstellung plotten wollen, verwenden Sie für die Basisfunktionen die bereits bekannte Ortsraumdarstellung der Basiszustände:

$$\langle x|\, n\rangle = \phi_n(x) = \pi^{-\frac{1}{4}} \frac{1}{\sqrt{2^n n!}} H_n(x) e^{-x^2/2}, \qquad \text{mit } n = 0, 1, 2, \ldots, \tag{2}$$

mit den Hermite-Polynomen H_n (vgl. Aufgabe 27). Plotten Sie Ihre Ergebnisse auf dem Intervall $x \in [-2, 2]$.

- Beachten Sie, daß die hier verwendeten Eigenzustände des harmonischen Oszillators eine *Orthonormalbasis* bilden.
- Zur Lösung des letztendlichen Eigenwertproblems können Sie die Funktion `eigh` aus dem NumPy-Modul `linalg` benutzen:

  ```
  import numpy as np
  evals, evecs = np.linalg.eigh(matrix)
  ```

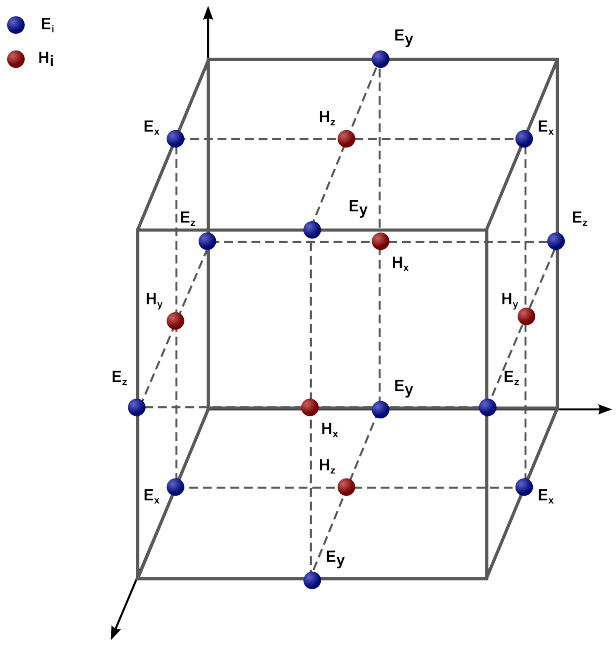
  Wobei danach `evals` die Eigenwerte und `evecs` die Eigenvektoren der Matrix `matrix` enthalten.

35. Das bewegte Feld

optionale Aufgabe – 12 Punkte

Die **Maxwell-Gleichungen** haben wir bereits in der KTP II ausführlich studiert. Eine exakte Lösung ist oftmals schwierig zu ermitteln und nur in Spezialfällen analytisch möglich. Deshalb bedienen wir uns numerischer Methoden, etwa der in der Praxis sehr erfolgreich eingesetzten **Yee-Vischen Methode**, die in der Vorlesung kurz skizziert wurde. Als eine der Standardmethoden wird sie in kommerziellen Softwarepaketen zur Simulation von verschiedenen Versuchsanordnungen umgesetzt, etwa um kompakte Antennen zu bauen. Das Ziel dieser Aufgabe ist es, die **Feldverteilung einer Dipolantenne** zeitaufgelöst zu bestimmen.

Zunächst aber wollen wir noch einmal die wesentlichen Schritte des Yee-Vischen Verfahrens zusammentragen. Grundlage ist die von Yee eingeführte spezielle Art des Diskretisierung der Felder, die in der folgenden Abbilung dargestellt ist:



Der Yee-Vischen-Algorithmus ist eine **Halbschrittmethode**. Konkret bedeutet dies für die Maxwell-Gleichungen, dass das elektrische Feld mit den Informationen vom Zeitpunkt t am Zeitpunkt $t+\frac{\Delta t}{2}$ berechnet wird und darauf basierend dann das magnetische Feld zum Zeitpunkt $t+\Delta t$.

Speziell gilt:

$$
\begin{aligned}
E_x(i,j,k,t+\frac{\Delta t}{2}) &= C_{E,1}E_x(i,j,k,t-\frac{\Delta t}{2}) \\
&\quad + C_{E,2}\left(\frac{B_z(i,j+1,k,t)-B_z(i,j,k,t)}{\Delta h} - \frac{B_y(i,j,k+1,t)-B_y(i,j,k,t)}{\Delta h}\right) \\
E_y(i,j,k,t+\frac{\Delta t}{2}) &= C_{E,1}E_y(i,j,k,t-\frac{\Delta t}{2}) \\
&\quad + C_{E,2}\left(\frac{B_x(i,j,k+1,t)-B_x(i,j,k,t)}{\Delta h} - \frac{B_z(i+1,j,k,t)-B_z(i,j,k,t)}{\Delta h}\right) \\
E_z(i,j,k,t+\frac{\Delta t}{2}) &= C_{E,1}E_z(i,j,k,t-\frac{\Delta t}{2}) \\
&\quad + C_{E,2}\left(\frac{B_y(i+1,j,k,t)-B_x(i,j,k,t)}{\Delta h} - \frac{B_x(i,j+1,k,t)-B_x(i,j,k,t)}{\Delta h}\right) \\
B_x(i,j,k,t+\Delta t) &= C_{B,1}B_x(i,j,k,t) \\
&\quad + C_{B,2}\left(\frac{E_y(i,j,k+1,t)-E_y(i,j,k,t)}{\Delta h} - \frac{E_z(i,j+1,k,t)-E_z(i,j,k,t)}{\Delta h}\right) \\
B_y(i,j,k,t+\Delta t) &= C_{B,1}B_y(i,j,k,t) \\
&\quad + C_{B,2}\left(\frac{E_z(i+1,j,k,t)-E_z(i,j,k,t)}{\Delta h} - \frac{E_x(i,j,k+1,t)-E_x(i,j,k,t)}{\Delta h}\right) \\
B_z(i,j,k,t+\Delta t) &= C_{B,1}B_z(i,j,k,t) \\
&\quad + C_{B,2}\left(\frac{E_x(i,j+1,k,t)-E_x(i,j,k,t)}{\Delta h} - \frac{E_y(i+1,j,k,t)-E_y(i,j,k,t)}{\Delta h}\right)
\end{aligned}
$$

Die Diskretisierung in Raum und Zeit ist dabei angegeben durch Δh bzw. Δt. Noch zu klären ist die genaue Form der Vorfaktoren $C_{(E,B),(1,2)}$. Diese setzen sich aus der elektrischen und magnetischen Permeabilität ε und μ, sowie zwei Verlusttermen, σ_E und σ_B, zusammen. Beide setzen wir auf einen sehr kleinen Wert 10^{-6} um den Algorithmus stabil zu machen. Die Konstanten sind dann gegeben durch:

$$C_{E,1} = \frac{1 - \frac{\sigma_E \Delta t}{2\varepsilon}}{1 + \frac{\sigma_E \Delta t}{2\varepsilon}} \qquad C_{B,1} = \frac{1 - \frac{\sigma_B \Delta t}{2\mu}}{1 + \frac{\sigma_B \Delta t}{2\mu}}$$

$$C_{E,2} = \frac{\frac{\Delta t}{\varepsilon \Delta h}}{1 + \frac{\sigma_E \Delta t}{2\varepsilon}} \qquad C_{B,2} = \frac{\frac{\Delta t}{\mu \Delta h}}{1 + \frac{\sigma_B \Delta t}{2\mu}}$$

Die Quelle unseres Feldes soll ein **schwingender Dipol** sein, den wir dadurch realisieren, dass die z-Komponente des elektrischen Feldes E_z in der Mitte des Gitters sinusförmig variiert:

$$E_z(N/2, N/2, N/2, t) = \sin(2\pi t/T)$$

Wir nehmen an, dass die Quelle sehr viel kleiner ist, als der Gitterabstand, wodurch sie sehr einfach zu implementieren ist. Plotten Sie, nachdem Sie den Algorithmus implementiert haben, den Betrag des elektrischen Feldes in der (x, z) und der (x, y) Ebene, die den Ursprung schneiden, zu verschiedenen Zeiten. Mit Ursprung bezeichnen wir hier den Ort des schwingenden Dipols, das heißt den Punkt $(N/2, N/2, N/2)$. Mit Hilfe der Funktion plot_surface von matplotlib können Sie außerdem die z-Komponente des elektrischen Felds in der (x, y) Ebene darstellen.

Ein Tipp noch zur praktischen Implementierung: Sie simulieren nun ein drei dimensionales System, das sehr viele Gitterpunkte enthält. Starten Sie daher mit einer kleinen Zahl von Gitterpunkten, wie $N = 40$. Benutzen Sie außerdem die *slice*-Notation von numpy, die wir auf dem vorherigen Übungsblatt eingeführt haben, um Ihr Programm zu beschleunigen. Um die diskretisierten Ableitungen auszurechnen, empfehlen wir Ihnen die Funktion *diff* aus dem numpy-Paket zu benutzen. Die Angabe *axis* bezeichnet den veränderten Index aus den obigen Iterationsformeln.