Institut für Theoretische Physik
der Universität zu Köln

Prof. Dr. Simon Trebst
Michael Becker, Peter Bröcker

Computerphysik

Übungsblatt 7

SS 2014

Website: http://www.thp.uni-koeln.de/trebst/Lectures/2014-CompPhys.shtml
Abgabedatum: Montag, 26. Mai 2014 vor Beginn der Vorlesung

28. Lineare Algebra mit Python

Programmiertechniken

Bevor wir uns auf den folgenden Zetteln mit den Algorithmen der **linearen Algebra** beschäftigen, führen wir die wichtigsten Funktionen ein, die im **numpy**-Paket enthalten sind und die wir für die folgenden Aufgaben benötigen.

Eine **Matrix** mit leeren Einträgen erzeugen Sie mit

```
a_matrix = np.empty((rows, cols))
```

wobei *rows* und *cols* die Anzahl der Zeilen und Spalten angeben. Eine Matrix, die bereits mit Nullen gefüllt ist, wird mit

```
a_matrix = np.zeros((rows, cols))
```

initialisiert. Einen **Vektor** können Sie ganz genauso initialisieren, indem Sie nur eine Dimension angeben:

```
a_vector = np.zeros((entries))
```

Ob Sie einen Spalten- oder Zeilenvektor erstellen wollen ist egal, denn die entsprechenden Funktionen erkennen dies automatisch.

Nachdem Sie nun die für Ihren Algorithmus benötigten Matrizen und Vektoren erstellt haben, müssen Sie diese auch mit Werten füllen und manipulieren. Der Zugriff auf ein Element geschieht mit dem von Arrays bekannten eckigen Klammern:

```
print a_matrix[2, 0]
print a_vector[10]
```

Besonders praktisch ist die Möglichkeit, gleich einen ganzen Bereich zu manipulieren, den sie mit der Notation *Start:Ende* angeben können. Angenommen Sie haben eine 10x10 Matrix und wollen die obere rechte 5x5 Matrix ausgeben, dann würden Sie schreiben

```
print a_matrix[0:5, 5:10]
```

denn der Bereich wird durch die Zeilen 0 und 5 und durch die Spalten 5 und 10 begrenzt. Beachten Sie, dass der erste Wert zum Bereich gehört, der Zweite aber nicht. In der Mathematik entspricht dies Intervallen des Typs $[a,b)$. Erwähnt seien noch zwei praktische Schreibweisen. Wenn Sie in einer der Dimensionen alle Werte bis zu oder ab einem Wert benutzen wollen, dann müssen Sie nur einen Index angeben. Konkret sieht dies so aus:

```
print a_matrix[:5, 5:]
```

Wenn Sie alle Einträge bis zum Vorletzten, dem Vorvorletzten etc. auswählen möchten, so geben Sie negative Werte an. Unser vorheriges Beispiel wird dann zu

```
print a_matrix[:-5, -5:]
```

und liest sich: Wähle den Bereich aus, der begrenzt wird durch alle Zeilen bis auf die fünf Letzten und die fünf letzten Spalten.

Widmen wir uns nun den benötigten **Operationen**. Beachten Sie, dass Ausdrücke wie

```
matrix_3 = matrix_1 * matrix_2
another_vector = matrix * vector
```

keine Matrix-Matrix oder Matrix-Vektor Multiplikationen sondern elementweise Operationen sind. Stattdessen verwenden wir die Funktion *dot*

```
matrix_3 = np.dot(matrix_1, matrix_2)
another_vector = np.dot(matrix, vector)
```

Sie werden außerdem lineare **Gleichungssysteme lösen** müssen, was denkbar einfach mit der Funktion *solve* erledigt wird. Sie ist sogar in der Lage allgemeine Gleichungssysteme des Typs $A \cdot X = B$, wobei sowohl A als auch B und X Matrizen sind, zu lösen. Der Aufruf sieht folgendermaßen aus:

```
import numpy.linalg as npl

# solves AX = B for X
X = npl.solve(A, B)
```

Um Algorithmen zu testen ist es von Vorteil **Zufallsmatrizen** als Eingabe zu benutzen um viele verschiedene Testfälle zu generieren. Auch dazu bietet numpy mit dem Modul *numpy.random* die Möglichkeit. Die Funktion *rand* erzeugt einen n-dimensionalen Array mit Zufallszahlen zwischen 0 und 1. Wie dies genau funktioniert, werden wir auf einem der späteren Übungsblätter noch sehen. Die gewünschten Dimensionen übergeben wir als Parameter:

```
import numpy.random as npr

# a 6x8 random matrix
a_random_matrix = npr.rand(6, 8)
```

Alle diese Funktionen werden Sie in den nachfolgenden Aufgaben anwenden. Insbesondere die Möglichkeit ganze Bereiche auf einmal zu manipulieren, kann Ihre Programme erheblich beschleunigen. Verändern Sie Ihr Programm um das Potential eines Plattenkondensators zu berechnen so, dass statt der zwei Schleifen über die Gitterpunkte in x- und y-Richtung nur eine Zeile mit Bereichsnotation übrig bleibt und vergleichen Sie die Geschwindigkeit.

29. Spannende Sachen

5 Punkte

Die **Relaxationsmethode** eignet sich nicht nur für die Lösung von partiellen Differentialgleichungen sondern auch, um Systeme, die nach der **Minimierung** einer Variablen streben, zu simulieren. In der analytischen Mechanik haben wir dies etwa im Prinzip der kleinsten Wirkung kennengelernt, wo es darum ging, ein (geeignetes) Wirkungsfunktional zu minimieren.

In dieser Aufgabe wollen wir ähnliches untersuchen und eine **minimale Oberfläche** mit festem, gegebenen Randprofil berechnen. Für eine Fläche $A(x,y) \equiv A_{i,j}$, welche als Funktion von zweidimensionalen kartesischen Koordinaten beschrieben werden kann, gestaltet sich dies besonders einfach – es genügt, ein Relxationsverfahren zu implementieren, welches in jedem Relaxationsschritt für alle Punkte deren Höhendifferenz zu seinen Nachbarn zu verringert. Dazu bildet man schlicht das arithmetische Mittel der Nachbarn

$$A_{i,j}^{(n+1)} = \frac{1}{4}\left(A_{i-1,j}^{(n)} + A_{i+1,j}^{(n)} + A_{i,j-1}^{(n)} + A_{i,j+1}^{(n)}\right)$$

Wir wollen dieses Verfahren anwenden, um die Spanndächer zweier Bauwerke zu berechnen.

1. Der **Kölner Hauptbahnhof** soll im Jahr 2034 zu einem Kreuzungsbahnhof umgebaut werden. Das Gebäude soll einen quadratischen Grundriss haben und benötigt an allen vier Seiten eine Gleiszufahrt. Das Dach des Gebäudes soll an jeder Seite mittig ein Kosinusförmiges Profil erhalten, was die Zufahrt ermöglicht, rechts und links davon aber genau mit der waagerechten Seitenmauer abschließen. Der Architekt bittet Sie nun, ein Profil des Daches zu berechnen, was die Oberfläche unter den gegebenen Bedingungen auf dem Rand minimiert.

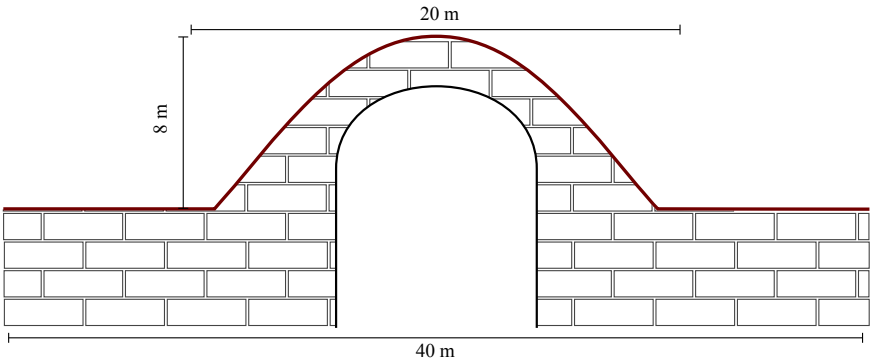


Abbildung 1: Seitenansicht einer der vier Bahnhofsfronten.

Nehmen Sie einen 40m $\times$ 40m großen Grundriss des Daches an und legen Sie für die Gesamtbreite der Kosinus-Bögen 20m, sowie als Höhe 8m fest. Verwenden Sie die Relaxationsmethode, um das Dach zu berechnen und stellen Sie es in einem 3D-Plot dar. Dazu können Sie folgende matplotlib-Befehle verwenden, wobei das Dachprofil in der Variablen `Z` gespeichert sei

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_zlim3d(0,8.0)
```

```
# berechne Dach
stride=3
ax.plot_surface(X,Y,Z,rstride=stride,cstride=stride,
    vmin = np.min(Z), vmax = np.max(Z),cmap='coolwarm')
plt.show()
```

2. Verwenden Sie nun die Methode des **Überrelaxierens** ($\omega > 1$) und untersuchen Sie, ob Sie damit in weniger Schritten die minimale Lösung erreichen. Zur Erinnerung:

$$A_{i,j}^{(n+1)} = (1-\omega)A_{i,j}^{(n)} + \omega\frac{1}{4}\left(A_{i-1,j}^{(n)} + A_{i+1,j}^{(n)} + A_{i,j-1}^{(n)} + A_{i,j+1}^{(n)}\right)$$

   Wie weit Ihre Relaxation während der Laufzeit schon vorangeschritten ist, können Sie überprüfen, indem Sie die Summe der Abweichungen von der letzten zur aktuellen Konfiguration betrachten:

$$\delta A^{(n)} = \sum_i \sum_j |A_{i,j}^{(n)} - A_{i,j}^{(n-1)}|$$

3. *Optionale Zusatzaufgabe*: Wenden Sie sich nun dem **Münchener Olympia-Stadion** zu: Nehmen Sie dazu eine rechteckige Grundfläche von 100m $\times$ 200m an und bringen Sie auf den langen Seiten drei und auf den kurzen Seiten zwei Kosinusbögen beliebiger, unterschiedlicher Höhe und Breite unter. Das Dach soll mittig durch einen Kosinus-förmigen Träger entlang einer der Diagonalen der Grundfläche unterstützt werden.

30. Der heiße Draht

5 Punkte

In der Vorlesung haben wir numerische Lösung der **Wärmeleitungsgleichung** via direkte Euler-Integration besprochen. Dabei haben wir festgehalten, dass die Stabilität einer solchen zeitlichen Entwicklung des Anfangswertproblems stark von der Diskretisierung dx in der räumlichen und dt in der zeitlichen Dimension abhängt und bestimmte Stabilitätsbedingungen eingehalten werden müssen.

Diese Einschränkung wird durch das **Crank-Nicolson-Verfahren**, welches wir diese Woche in der Vorlesung besprochen haben, aufgehoben. Wir wollen dies in dieser Aufgabe überprüfen. Die Iterationsschritte des Algorithmus haben Sie in der Vorlesung ausführlich besprochen. Wie Sie sich erinnern, müssen Sie an einem Schritt ein **lineares Gleichungssystem** lösen, was Sie mit der in unserem Crashkurs in Aufgabe 28 erwähnten Funktion *solve* tun können.

Implementieren Sie das Crank-Nicolson-Verfahren für die eindimensionale Wärmeleitungsgleichung

$$\frac{\partial u}{\partial t} = \kappa\frac{\partial^2 u}{\partial x^2}$$

mit dem Wärmeleitungskoeffizienten κ kennengelernt. Für die Bearbeitung dieser Aufgabe setzen wir $\kappa = 1$. Physikalisch betrachten wir damit etwa die zeitliche Entwicklung der Temperatur eines Drahtes. Als Anfangsverteilung bei $t = 0$ wählen wir

$$u(x,0) = \sin(\pi x), \quad x \in (0,1)\,.$$

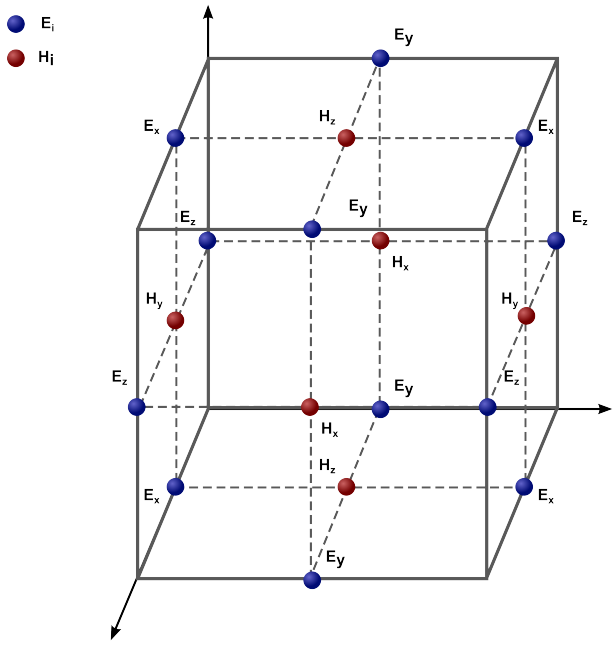
Untersuchen Sie die Stabilität des Crank-Nicolson-Verfahrens, indem Sie 20, 200 und 2000 räumliche Diskretisierungsschritte verwenden.

31. Das bewegte Feld

optionale Aufgabe – 6 Punkte
Abgabe am Montag, 2. Juni

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.