

# Einleitung zu SciPy

Daniel Herde

2009-10-29 Do

## Contents

<b>1</b>	<b>Vorwort</b>	<b>5</b>
<b>2</b>	<b>Einrichtung</b>	<b>5</b>
<b>3</b>	<b>Grundlagen von Python</b>	<b>5</b>
3.1	Variablen . . . . .	6
3.1.1	Zahlen . . . . .	6
3.1.2	Strings und String-Operationen . . . . .	6
3.1.3	Listen . . . . .	7
3.1.4	Tupel . . . . .	7
3.1.5	Dictionaries . . . . .	8
3.2	Funktionen . . . . .	8
3.2.1	Bedingungen und Schleifen . . . . .	8
3.2.2	Funktionen definieren . . . . .	9
3.3	Funktionale Features . . . . .	9
3.3.1	Anonyme Funktionen . . . . .	9
3.3.2	Generatoren . . . . .	10
3.3.3	Listenmanipulation mit map, filter und reduce . . . . .	11
3.3.4	Listen-Konstrukte . . . . .	11
3.3.5	Currying . . . . .	12
3.4	Objektorientierte Features . . . . .	12
3.4.1	Klassen mit Eigenschaften und Methoden . . . . .	12
3.4.2	public und private . . . . .	13
3.4.3	Vererbung . . . . .	13

3.5	Seltsamere Dinge . . . . .	13
3.5.1	eigene Module bauen . . . . .	13
3.5.2	Schwarzmagie: Eigene Listen-Implementationen . . .	13
<b>4</b>	<b>Ein- / Ausgabe</b>	<b>14</b>
4.1	CSV . . . . .	14
4.2	Matlab-Dateien . . . . .	14
4.3	scipy.io.save und cpickle . . . . .	15
<b>5</b>	<b>Array-Operationen</b>	<b>16</b>
5.1	Arrays mit Numpy . . . . .	16
5.1.1	Arrays erzeugen . . . . .	16
5.1.2	Array-Selektion . . . . .	16
5.1.3	Array-Multiplikation . . . . .	17
5.2	Lineare Algebra mit scipy.linalg . . . . .	17
5.2.1	Grundlegende Operationen . . . . .	17
5.2.2	Eigenwerte und Eigenvektoren . . . . .	17
5.2.3	Gleichungssysteme lösen . . . . .	17
5.2.4	Matrix-Winkelfunktionen . . . . .	17
<b>6</b>	<b>Ausgabe mit Matplotlib</b>	<b>18</b>
6.1	Einfache Plots . . . . .	18
6.2	2D-Plots . . . . .	19
6.2.1	Quiver . . . . .	19
6.2.2	Surface . . . . .	19
6.3	Interaktion mit dem Plot . . . . .	19
6.4	Plotten auf die richtige Art . . . . .	19
6.5	gnuplot-Bindings . . . . .	19
<b>7</b>	<b>Statistik mit scipy.stats</b>	<b>20</b>
7.1	Tests etc. . . . .	20
7.2	Wahrscheinlichkeits-Dichten . . . . .	20
7.2.1	die gesuchte Gauss-PDF . . . . .	20
7.3	Erweiterte Funktionen mit RPy . . . . .	20

<b>8</b>	<b>Andere Sprachen</b>	<b>21</b>
8.1	FORTRAN-Code und Python mit f2py . . . . .	21
8.2	C-Bibliotheken einbinden mit Py++ . . . . .	21
8.3	C/C++ Code einbetten mit scipy.weave . . . . .	22
8.3.1	scipy.weave.inline . . . . .	22
<b>9</b>	<b>GUI mit TKinter</b>	<b>23</b>
9.1	minimale GUI erstellen . . . . .	23
9.2	Ereignisse erstellen und nutzen . . . . .	23
<b>10</b>	<b>Schlimme Dinge</b>	<b>24</b>
10.1	Kurzes if . . . . .	24

Dieses Buch ist einer zweistündigen Suche nach der pdf einer Gauss-Funktion gewidmet.

# 1 Vorwort

Irgendwie fehlt es im deutschsprachigen Raum an einer vernünftigen Dokumentation um mit SciPy anzufangen. Jeder CS/Computational \*-Kurs hat seinen eigenen Stapel an L<sup>A</sup>T<sub>E</sub>X-Dokumenten, aber keiner ist wirklich überzeugend.

Testing  
mehr Testing

Jetzt ist die <sup>1</sup>Frage: Besteht Interesse an einer gedruckten Zusammenfassung, die gleichzeitig als Einführung und als Erinnerungsstütze dienen kann?

Ich vermute ja - und das ist dsas Ziel dieses Guides

## 2 Einrichtung

Der schnellste Weg zu einer funktionsfähigen Scipy-Installation führt über Sage. Sage ist eine vorkompilierte Pythoninstallation mit verschiedenen wissenschaftlichen Paketen, unter anderem a, b und c. Zusätzlich enthält es einen webbasierten Notebook-Modus, ähnlich zu Mathematica.

Für Windows gibt es Enthought Python als Paket um eine funktionsfähige Installation zu erhalten.

Die meisten Linux-Distributionen haben es auch in ihrer Paketverwaltung, allerdings ist es meist eine etwas ältere Version.

Als letzte Option bleibt natürlich immer noch, es selber zu rollen. Die Quellen sind verfügbar unter:

## 3 Grundlagen von Python

In diesem Abschnitt werden die Grundlagen von Python kurz angerissen. Um die Sprache zu erlernen gibt es deutlich bessere Bücher, sehr empfehlenswert sind:

---

<sup>1</sup>Testing

## 3.1 Variablen

Variablen in Python sind dynamisch typisiert und müssen nicht deklariert werden vor der Benutzung. Natürlich ist es möglich explizit den Typ zu ändern.

### 3.1.1 Zahlen

Der Unterschied zwischen Integer und Float liegt in den folgenden zwei Zeilen:

```
a=5
b=3.
```

Der Typ des Ergebnisses wird automatisch gewählt.

```
5/3 != 5/3.
```

Hier sieht man ein kleines Problem: Int/Int wird natürlich Int, Int/Float wird zu Float.

### 3.1.2 Strings und String-Operationen

Python hat ein paar sehr einfach nutzbare String-Manipulations-Routinen.

```
c = "Dies ist ein Test"
```

Man kann einfach Teile von Strings selektieren, Strings bei Trennzeichen unterteilen und zusammenfügen.

```
c[:4]=="Dies"
d=c.split(" ")
d[0]+" "+d[3]=="Dies Test"
```

Eine numerisch deutlich effizientere Methode Strings zusammenzufügen ist

```
" ".join(d)
```

Die Split-Operation liefert eine Liste zurück, siehe nächster Abschnitt.

### 3.1.3 Listen

Listen kann man einfach definieren als

```
e = ["Erstes", "Element", 5, 7.]
```

Die Typen der Elemente dürfen variieren, im allgemeinen werden Listen allerdings als Werkzeug der Wahl für eine Menge homogener Elemente gesehen.

Eine Liste der ersten N Zahlen erzeugt man durch

```
n = range(N)
```

Man kann sie genauso wie Strings auseinandernehmen und zusammenfügen

```
n[:5]
n[2:]
n[[1,5,3]]
n[:5]+[5:]
```

Einzelne Elemente können manipuliert werden über

```
n[0]=1
```

### 3.1.4 Tupel

Für unveränderbare Objekte existieren Tupel. Im Prinzip verhält es sich wie eine Liste, deren Elemente nicht manipuliert werden können.<sup>2</sup> Erzeugt wird sie als

```
t=(1,2,'ab','cd')
```

Da sie unveränderbar sind, können sie im Gegensatz zu Listen auch als Schlüsselwörter in Dictionaries eingesetzt werden.

Sowohl Tupel als auch Listen können einfach entpackt werden

```
v=(0.1,2.5,1.3)
x,y,z=v
```

Dies ist relevant, da die meisten ausgefalleneren Funktionen ein Tupel zurückgeben.

---

<sup>2</sup>Die Python-FAQ vergleicht allerdings eine Liste zu Arrays in anderen Sprachen und Tupel zu Structs

### 3.1.5 Dictionaries

Ein Dictionary, also Wörterbuch, ist definiert über eine Liste von Schlüsseln mit zugeordneten Werten. Erzeugt und ausgelesen werden sie mit

```
d = {'Apfel': 'rot', 'Birne': 'gelb'}
d['Apfel']
```

Geändert oder erweitert werden Wörterbücher auf die gleiche Weise, abhängig davon ob der Schlüssel bereits existiert:

```
d['Gras'] = 'grün'
```

## 3.2 Funktionen

### 3.2.1 Bedingungen und Schleifen

Flusssteuerung in Python ist relativ trivial. Es ist wichtig zu beachten dass Blöcke statt mit geschwungenen Klammern, wie in C, in Python über die Einrückung definiert werden<sup>3</sup>.

```
if (a==2):
    print 1
    print 2
else:
    print 2
```

In Python gibt es nur Kopfschleifen der while-Form (?):

```
while a<10:
    a+=1
    print a
```

Alternativ ist es möglich über eine Liste von Elementen zu iterieren:

```
n = range(10)
for i in n:
    print i
```

---

<sup>3</sup>Es gibt dazu zwei verschiedene Schulen, die Vertreter von 4 Leerzeichen Einrückung und die Vertreter von 8 Leerzeichen Einrückung. Die Ketzer die an Tabulatoren glauben wurden inzwischen verbannt.

## 3.2.2 Funktionen definieren

Hier ist eine Beispielfunktion mit ihrer Verwendung:

```
def testing(a=5, *b, **c):  
    print a  
    print b  
    print c  
    return b,c,a
```

Sie kann aufgerufen werden ohne jeden Parameter

```
res=testing()  
5  
( )  
{ }  
res==( { }, ( ), 5)
```

Die Variable a hat jetzt den als Standard vergebenen Wert, b - die Variable die alle überschüssigen unbenannten Variablen abfängt ist eine leere Liste und c, die alle unbekannt benannten Parameter sammelt ist ein leeres Dictionary.

Ein etwas ausgefallener Aufruf kann nun so aussehen:

```
res=testing(1,2,3,4,5,test1=1, test2=2)  
1  
(2, 3, 4, 5)  
{'test1': 1, 'test2': 2}  
res==( {'test1': 1, 'test2': 2}, (2, 3, 4, 5), 1)
```

Hier wird nun klar was b und c machen.

## 3.3 Funktionale Features

### 3.3.1 Anonyme Funktionen

Ein sehr praktisches Werkzeug sind anonyme Funktionen, die direkt an der Stelle definiert werden können wo man sie benötigt.

```

def create_exp(basis):
    """
    Erzeuge eine Funktion exp der Form basis**x == exp(x)
    """
    return lambda x: basis**x

e2 = create_exp(2)
e2(2) == 4
e2(3) == 8

```

### 3.3.2 Generatoren

In Python spielen Operationen, die auf Listen angewendet werden oder bei denen über eine Liste iteriert wird eine grosse Rolle. Ein Generator ist eine Beschreibung des Listeninhalts und kann direkt an Stelle einer Liste verwendet werden, wenn darüber iteriert werden soll.

```

def powers(lim=100000):
    yield 1
    i = 2
    while i<lim:
        yield i
        i=i**2

```

Darüber kann nun iteriert werden:

```

for i in powers():
    print i

```

Manuell würde man einen Generator wie folgt verwenden:

```

pow10k = powers(lim=10000)
while 1:
    try:
        print pow10k.next()
    except StopIteration:
        break

```

Man kann Generatoren genauso in map, filter und reduce-Ausdrücken einsetzen.

### 3.3.3 Listenmanipulation mit map, filter und reduce

Manchmal soll eine Operation auf eine Liste von Elementen angewendet werden. In dem Fall kann man eine Schleife über die Elemente verwenden und eine neue Liste erzeugen - eine ausnehmend langsame Konstruktion in Python - oder man verwendet `map`. Dies wendet eine abbildende Funktion auf eine Liste von Elementen an, in der Form von:

```
def square(z):
    return z**2

squares = map(square, range(100))
```

Damit erhält man nun eine Liste aller Quadratzahlen. Interessiert man sich nun für die Liste aller ungeraden Quadratzahlen, kann man die Liste entsprechend filtern<sup>4</sup>:

```
odd_squares = filter(lambda x: x%2==1, squares)
```

Um nun eine Liste auf einen Wert zu reduzieren, gibt es `reduce`. Dieses akzeptiert eine Funktion die zwei Eingangswerte nimmt und einen zurückgibt und wendet das wiederholt von vorne auf die Liste an um sie auf einen Wert zu reduzieren. Sinnvolle Verwendungen sind zum Beispiel

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
reduce(lambda x,y: x+' '+y, a)
```

auch wenn man das einfacher ausdrücken könnte oder

```
reduce (lambda x,y: x*y, [1,2,3,5,7,11])
```

### 3.3.4 Listen-Konstrukte

Es gibt ein weiteres Konstrukt, das ähnlich eingesetzt werden kann wie `map` und `filter`. Die Anweisung

---

<sup>4</sup>Beachtet die Verwendung eines Lambda-Konstrukts um einen kurzen, leicht verständlichen Ausdruck zu erhalten

```
[a*b for a in range(100) \
    for b in range(100) \
    if isprime(a) and isprime(b)]
```

generiert eine Liste aller Zahlen die das Produkt zweier Primzahlen kleiner 100 sind.

### 3.3.5 Currying

Ab und zu will man eine Funktion mehrmals mit teilweise denselben Argumenten verwenden, gemäß DRY<sup>5</sup> ist es sinnvoll sich eine neue Funktion zu erzeugen, die diese Argumente bereits gesetzt hat.

Ab Python 3.0 gibt es dazu `functools.partial`:

Davor implementiert man etwas derartiges am besten mit `lambda`:

```
def simulation(x0,y0):
    """
    hier wird irgendeine Form von Simulationscode stehen
    """
```

```
sim100 = lambda y0: simulation(100, y0)
```

Die erste Lösung ist natürlich etwas eleganter. Die `functools`-Bibliothek bietet übrigens noch einige weitere interessante Features.

## 3.4 Objektorientierte Features

Die Idee hinter Objekten<sup>6</sup> ist es Eigenschaften und Funktionen die sie manipulieren zusammenzufassen und damit eine gewisse Menge Komplexität vor dem Anwender zu verstecken / es einfacher handhabbar machen.

### 3.4.1 Klassen mit Eigenschaften und Methoden

Im Prinzip schreibt man einfach `class` und packt dann alles hinein, was in die Klasse soll<sup>7</sup>:

---

<sup>5</sup>Don't Repeat Yourself.

<sup>6</sup>Das ist irgendwo zwischen sehr ungenau und falsch, eine deutlich bessere Einführung gibt es in dicken Büchern die sich dem Thema widmen

<sup>7</sup>Das ist wieder ein schlechtes Beispiel, Die Lektion für Python lautet: Keine(!) getter und setter - dafür gibt es Eigenschaften.

## BESSERES BEISPIEL WÄHLEN

```
class GoFeld ():
    def __init__ (self, size):
        self.size = size
        self.field = scipy.zeros([size,size])

    def set (self, x, y, number):
        self.field[x,y] = number

    def get (self, x, y)
        return self.field[x,y]
```

Genutzt wird es, indem man eine Instanz des Objekts erzeugt und es dann manipuliert.

```
field1 = GoFeld(19)
print field1.get(0,0)
field1.set(1,1,2)
print field1.get(1,1)
```

### 3.4.2 public und private

Sofern nicht anders deklariert sind alle Eigenschaften und Methoden von Python-Klassen öffentlich.

### 3.4.3 Vererbung

## 3.5 Seltsamere Dinge

### 3.5.1 eigene Module bauen

### 3.5.2 Schwarzmagie: Eigene Listen-Implementationen

Das schöne an Python ist, dass ein Grossteil der Sprache direkt in Python realisiert ist. Damit kann man interessante Konstrukte erzeugen, wie zum Beispiel Klassen, die sich wie ein Listen-Objekt verhalten und damit wie eins verwendet werden können.

## 4 Ein- / Ausgabe

Scipy bietet einige nützliche Funktionen um Daten einfach einzulesen und auszugeben, sowie Arrays effizient zu speichern.

### 4.1 CSV

Comma Separated Value(CSV)-Dateien einzulesen ist straightforward mit

```
a=scipy.io.read_array('/CSV/Datei.txt')
```

Natürlich kann man genauer spezifizieren was eingelesen werden soll:

```
b=scipy.io.read_array('/CSV/Datei.txt', separator=" ", \
columns=(1,2,(4,9,2),-3), lines=(2,-1), comment='#', \
linesep='\r\n', missing=0.)
```

Hier werden der Separator zwischen den Spalten, die einzulesenden Spalten und Zeilen, der Indikator für Kommentar-Zeilen und der Zeilenbruch explizit angegeben. Die Variable "missing" stellt den Wert dar der in Felder eingetragen wird, für die kein Wert bestimmt werden konnte.

Die Syntax der Spalten- und Zeilenauswahl ist etwas eigen, man kann direkt die Nummern angeben, einen Bereich mit (start, stop, step) und eine negative Zahl, welche die Schrittweite angibt, mit der bis zum Ende der Zeile oder Datei gegangen wird.

Diese Funktion ist leider DEPRECATED, also Absatz neuschreiben mit `numpy.loadtxt`.

Genauso kann man einfach ein Array in eine CSV-Datei ausgeben, falls man ein externes Programm zum Plotten verwenden will oder ähnliches.

```
scipy.io.write_array('/CSV/output.txt', a)
```

Details zur Benutzung finden sich im Hilfestring.

### 4.2 Matlab-Dateien

Auch wenn Matlab von Grund auf geschaffen wurde um Arrays zu manipulieren, gibt es manchmal gute Gründe eine echte Programmiersprache wie Python zu verwenden - in dem Fall ist es allerdings nötig Kompatibilität zu Matlab herzustellen.

```
c=scipy.io.loadmat('/Matlab/file.*')
```

Damit werden eine Matlab-Datei eingelesen und die darin enthaltenen Elemente in einem Dictionary mit den in der Datei angegebenen Namen bereitgestellt:

```
data = c['measurements']
```

Auf ähnliche Weise kann einfach ein Array in eine Matlab-Datei schreiben:

```
scipy.io.savemat('/Matlab/processed.file')
```

### 4.3 `scipy.io.save` und `cpickle`

Um Ergebnisse wohlkommentiert zu speichern und einfach weiter verarbeiten zu können verwendet jeder ein eigenes System.

Eine sehr einfache Lösung ist es ein Dictionary zu definieren mit allen wichtigen Informationen und den entsprechenden Tabellen und es einfach mit

```
scipy.io.save_as_module('dateiname',dict)
```

in ein Python-Modul zu speichern das einfach mit

```
import dateiname
print dateiname.key
```

importiert werden kann und die im Dictionary gespeicherten Informationen als Eigenschaften enthält.

Alternative gibt es das `cPickle`-Modul, um einfach ein Objekt einzuwecken. Geschrieben wird die Datei mit

```
import cPickle
cPickle.dump(dict, 'dateiname')
```

und genauso kann man das Objekt wiederherstellen mit

```
dict.cPickle.load('dateiname')
```

Das ganze wird in einem Binärformat gespeichert und benötigt damit deutlich weniger Speicher als die üblicherweise verwendeten CSV-Dateien.

## 5 Array-Operationen

Wissenschaftliche Arbeit läuft meistens aus mehr oder minder ausgefallene Manipulation von Arrays heraus. Von Haus aus liefert Python keine Bibliothek für schnelle Array-Operationen mit. Allerdings liefert NumPy, kurz für Numerical Python, die entsprechende Funktionalität.

### 5.1 Arrays mit Numpy

#### 5.1.1 Arrays erzeugen

Arrays können entweder aus verschachtelten Listen oder initialisiert mit einem bestimmten Wert erzeugt werden.

```
a = scipy.array([[1,2,3],[4,5,6],[7,8,9]])
```

erzeugt ein Array mit den gegebenen Werten und Dimensionen.

```
b = scipy.ones([2,2])
```

erzeugt ein Array der Dimension 2x2, gefüllt mit Einsen.

#### 5.1.2 Array-Selektion

Einzelne Elemente können in der folgenden Weise selektiert werden:

```
c = a[0,0]
```

Und alle Elemente einer Dimension wie folgt:

```
d = a[:,0]
```

Dies liefert ein eindimensionales Array mit allen Elementen der ersten Spalte zurück.

Es ist auch möglich Tupel zur Indizierung zu verwenden:

```
a[1:3,[0,1]]
```

Dies liefert ein Array zurück das selektiert wurde als die Zeilen von (inklusive) 1 bis (exklusive) 3. und den Spalten 0 und 1 von A.

Es ist einfach möglich das transponierte eines Arrays zu verwenden, mittels

```
a.transpose()
```

```
a.T
```

### 5.1.3 Array-Multiplikation

Arrays können einfach elementweise mit einer Zahl multipliziert werden:

```
e = 5. * a
```

Genauso ist es möglich Arrays mit gleich großen Arrays elementweise zu multiplizieren:

```
f = a*a
```

Es ist natürlich auch möglich arrays vektoriell zu multiplizieren:

```
b = scipy.ones([10])
```

```
c = scipy.dot(b,b)
```

Das äußere Produkt kann man in der folgenden Form erhalten:

```
d = scipy.outer(b,b)
```

## 5.2 Lineare Algebra mit `scipy.linalg`

Am Anfang steht wie immer

```
import scipy.linalg
```

### 5.2.1 Grundlegende Operationen

Quadratische Matrizen lassen sich invertieren mit

```
e = scipy.linalg.inv(d)
```

Die Determinante erhält man mit

```
det_d = scipy.linalg.det(d)
```

dazu wird die LAPACK-Implementation der LU-Faktorisierung verwendet.

### 5.2.2 Eigenwerte und Eigenvektoren

### 5.2.3 Gleichungssysteme lösen

### 5.2.4 Matrix-Winkelfunktionen

## 6 Ausgabe mit Matplotlib

Es gibt verschiedene Möglichkeiten die Ergebnisse nun zu visualisieren. Man kann entweder ein spezialisiertes Werkzeug, wie Gnuplot, dafür verwenden - oder eins der Pythonmodule mit ähnlicher Funktionalität<sup>8</sup>.

Im ersten Teil dieses Kapitels wird matplotlib aus dem Pylab-Paket(?) demonstriert. Alternativ gibt es auch Pakete die erlauben gnuplot von python aus anzusteuern. Insofern,

```
import pylab
```

### 6.1 Einfache Plots

Das Minimal-Beispiel um einen ersten Plot zu erzeugen sieht gewohnt kurz aus:

```
x = scipy.linspace(0.,3*scipy.pi,1000)
y = scipy.sin(x)
pylab.plot(x,y)
```

Hier ist eine ausführlichere Plot-Anweisung:

```
pylab.plot(x1,y1,x2,y2,Form, Farbe, weitere Details)
```

Weiter interessante Eigenschaften sind

```
pylab.axis([xmin, ymin, xmax, ymax])
pylab.xlabel('X-Beschriftung, kann \LaTeX sein')
pylab.ylabel('Y-Beschriftung, kann \LaTeX sein')
pylab.title('Titelzeile')
```

Alternativ kann man ein Plot-Objekt erzeugen und seine Eigenschaften verändern.

---

<sup>8</sup>Natürlich sollte man auch in diesem Fall strikt zwischen IO, Simulationslogik und Visualisierung trennen. Und es wird sich niemand an diese Empfehlung halten.

## 6.2 2D-Plots

### 6.2.1 Quiver

### 6.2.2 Surface

## 6.3 Interaktion mit dem Plot

Matplotlib kann auch interaktiv benutzt werden. Dies ist durch ein einfaches Ereignis-getriebenes System realisiert. Zuerst benötigt man eine Methode die aufgerufen wird, wenn etwas passiert:

```
def getCoordinates(event):  
    print event.xdata, event.ydata
```

Diese Funktion bindet man an ein Ereignis mit bevor man den Plot anzeigen lässt

```
pylab.connect('button_press_event',getCoordinates)  
pylab.show()
```

Details findet man in der Matplotlib-Hilfe unter “event handling”.

## 6.4 Plotten auf die richtige Art

Die saubere Art, plots zu realisieren, führt über ein Plot-Objekt. Ein Beispielprogramm mit den bisher verwendeten Funktionen würde so aussehen:

## 6.5 gnuplot-Bindings

Es scheint ein Python-Modul zu geben, das über pipes mit Gnuplot interagiert. Mal die Demo anschauen. Gibt es noch andere Möglichkeiten unter Python zu plotten?

## 7 Statistik mit `scipy.stats`

### 7.1 Tests etc.

### 7.2 Wahrscheinlichkeits-Dichten

#### 7.2.1 die gesuchte Gauss-PDF

### 7.3 Erweiterte Funktionen mit `RPy`

## 8 Andere Sprachen

Es gibt ein kleines Problem damit Python für wissenschaftliche Zwecke einzusetzen: Es ist langsam, wenn man versucht ohne Zusatzbibliotheken Numbercrunching zu betreiben. Darum gibt es `scipy` und ähnliche Module, die einem erlauben die meisten Operationen als Serie verschiedener eingebauter, effizienter Funktionen zu schreiben.

### 8.1 FORTRAN-Code und Python mit `f2py`

Numpy liefert direkt ein Werkzeug mit um ein Python-Interface für Fortran-Code zu erzeugen.

Zuerst wird in einer separaten Datei (hier: `testing.f`) die Fortran-Routine angelegt:

```
SUBROUTINE
```

Um jetzt ein Interface dazu zu erzeugen, muss nur `f2py` über die Konsole aufgerufen werden:

```
f2py -c -m testing testing.f
```

Damit wurde nun ein Python-Modul erzeugt das direkt in Python importiert werden kann.

```
import testing
```

Die Umsetzung der Parameter ist allerdings relativ unhandlich - es sind keinerlei Information darüber enthalten welche der Variablen Input/Output darstellen, etc. Dazu ist es möglich die Interface-Definition manuell zu überarbeiten. Details dazu finden sich im Netz.

### 8.2 C-Bibliotheken einbinden mit `Py++`

Es ist möglich einen Wrapper für C++-Deklarationen zu erzeugen, der sie unter Python zur Verfügung stellt, ähnlich zu der Funktionsweise von `f2py`. Ich muss leider etwas mehr darüber lesen.

## 8.3 C/C++ Code einbetten mit `scipy.weave`

“Die meisten” reicht natürlich meistens nicht, insbesondere wenn man das Problem nicht elegant als Vektoroperation ausdrücken kann. Für diesen Fall gibt es `scipy.weave`, ein Modul das erlaubt C/C++-Erweiterungen in Python zu verwenden oder direkt in den Code einzubetten.

### 8.3.1 `scipy.weave.inline`

Diese Funktion erlaubt C-Code direkt in den Python-Quellcode einzubetten. Hier ist ein Minimal-Beispiel:

```
import scipy
import scipy.weave

len = 100
a = scipy.arange(len)
scipy.weave.inline('
    int i;
    for (i=0; i<len; i++) {
        a[i]=a[i]*2;
    };',
    arg_names = ['len', 'a'])
print a
```

Sobald das Programm an der entsprechenden Stelle ankommt wird das Code-Segment kompiliert (sofern nicht bei einem vorherigen Durchlauf schon geschehen), die bei `arg_names` angegebenen Variablen konvertiert und der Code ausgeführt. Es ist keine `return`-Anweisung notwendig, am Ende werden die Argumente zurückkonvertiert und können einfach weiter genutzt werden.

Auf diese Weise ist es möglich innere Schleifen bzw. die Update-Routine bei einer Simulation durch schnellen C-Code zu realisieren, aber für das gesamte Parameter / IO-Handling den Komfort Pythons nutzen zu können.

## 9 GUI mit TKinter

Im Prinzip ist damit alles abgedeckt was man an Handwerkszeug braucht. Ab und zu reicht aber ein Konsolen-basiertes Skript nicht<sup>9</sup> bzw. jemand anders will das Programm benutzen<sup>10</sup>. Dann kann man mit TKinter relativ einfach eine kleine GUI dazu erstellen. Dieses Kapitel ist noch etwas unzureichender als die anderen - für eine vollständigere Einführung in TKinter empfehlen sich [...]

### 9.1 minimale GUI erstellen

### 9.2 Ereignisse erstellen und nutzen

---

<sup>9</sup>unvorstellbar.

<sup>10</sup>An dieser Stelle würde ich gerne nochmal auf die Bedeutung von Kommentaren hinweisen

## 10 Schlimme Dinge

Ein paar Dinge die mir über den Weg gelaufen sind bei der Recherche hierfür.

### 10.1 Kurzes if

Es gibt ja in C/C++ die Form `<Bedingung>?<wenn ja>:<wenn nein>`, wenn man keine Lust hat einen Drei-Zeiler daraus zu machen. In Python nimmt es folgende Form:

```
<wenn ja> if <Bedingung> else <wenn nein>
```