Informatik I

27. Wissenschaftliches Rechnen mit numpy, scipy und matplotlib

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

06.02.2014

Informatik I

06.02.2014 — 27. Wissenschaftliches Rechnen mit numpy, scipy und matplotlib

27.1 Motivation

27.2 Installation

27.3 Das Modul `numpy`

27.4 Funktionswerte zeich nen)

27.1 Motivation

Motivation

- Python ist eine tolle Programmiersprache, bietet aber mehr ...
- Es existieren Pakete, die wissenschaftliches Rechnen ermöglichen.
- Normalerweise benutzt man dafür die proprietären, kommerziellen Systeme MATLAB oder auch dem Opensource-System Scilab.
- Mit den Paketen `numpy`, `scipy` und `mathplotlib` erhält man aber fast die gleiche Funktionalität.
- Wir werden ein bisschen hinein schnuppern.

27.2 Installation

Installation

- Die genannten Module gehören nicht zur Grundausstattung von Python.
- Es werden verschiedenste Möglichkeiten unter `http://www.scipy.org` angeboten.
- Die einfachsten sind, ein ganzes Paket inklusive Python-Interpreter herunter zu laden und zu installieren.
- Für Windows ist ein portables Paket WinPython: `http://winpython.sourceforge.net/` – kann aus jedem Ordner heraus gestartet werden.
- Für Windows gibt es vorbereitete Ergänzungspakete: `http://www.lfd.uci.edu/~gohlke/pythonlibs/`.

27.3 Das Modul numpy

Der Typ ndarray

- numpy führt einen speziellen Type ndarray ein.
- Dies sind Arrays von fixer Größe mit einheitlichem Basistyp (Ganzzahl, Fließkomma, Strings einer maximalen Länge ...).
- Diese können u.a. mit der Funktion arange erzeugt werden.

Python-Interpreter

```
>>> import numpy as np
>>> x = np.arange(10); x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> type(x)
<class 'numpy.ndarray'>
>>> y = np.arange(10.0); y
array([0., 1., 2., 3, 4., 5., 6., 7., 8., 9.])
>>> z = np.array(['abc', 'efghij', 'k']); z
array(['abc', 'efghij', 'k'], dtype='<U6')
```

Warum Arrays?

arraytest.py

```
import numpy as np
import time
def trad_version():
    t1 = time.process_time()
    x, y, z = range(10**7), range(10**7), []
    for i in range(len(x)):
        z.append(x[i] + y[i])
    return time.process_time() - t1
def numpy_version():
    t1 = time.process_time()
    x, y = np.arange(10**7), np.arange(10**7)
    z = x + y
    return time.process_time() - t1
print("Traditionell:  ", trad_version())
print("NumPy-Version: ", numpy_version())
```


Mehrdimensionale Arrays

- Arrays können mehrere Dimensionen besitzen.
- Die Anzahl der Dimensionen findet sich im Attribut `ndim`, `shape` beschreibt die Form durch ein Tupel.

Python-Interpreter

```
>>> x = np.array( ((11,12,13), (21,22,23), (31,32,33), (41,42,43)
) )
>>> print(x)
[[11 12 13]
 [21 22 23]
 [31 32 33]
 [41 42 43]]
>>> x.ndim
2
>>> x.shape
(4, 3)
>>> print(x[3][1], x[3, 1]
42 42
```


Slicing von Arrays

- Ähnlich wie bei Listen funktioniert auch in numpy das slicen von Arrays.
- Das funktioniert auch über mehrere Dimensionen.
- Allerdings werden keine Kopien erzeugt sondern Views auf das Originalobjekt.

Python-Interpreter

```
>>> y = x[:2,:1]
>>> print(y)
[[11]
 [21]]
>>> y[0,0] = 100
>>> print(x)
[[100 12 13]
 [ 21 22 23]
 [ 31 32 33]
 [ 41 42 43]]
```


Arrays flach klopfen

- Mehrdimensionale Arrays können in 1-D Arrays transformiert werden.
- Dazu gibt es zwei Methoden: `flatten()` und `ravel()`.
- `flatten` erstellt eine Kopie, `ravel` gibt einen View.

Python-Interpreter

```
>>> x.ravel()
array([100, 12, 13, 21, 22, 23, 31, 32, 33, 41, 42, 43])
>>> x.flatten()
array([100, 12, 13, 21, 22, 23, 31, 32, 33, 41, 42, 43])
>>> x.ravel()[-1] = 0; x.flatten()[-2] = 0
>>> print(x)
[[100 12 13]
 [ 21 22 23]
 [ 31 32 33]
 [ 41 42 0]]
```

Arrays umdimensionieren

- Arrays können mit der Methode reshape() umdimensioniert werden, wobei wieder nur ein View erzeugt wird, keine Kopie!

Python-Interpreter

```
>>> x = np.array(range(24))
>>> y = x.reshape((2, 4, 3))
>>> print(y)
    [[[ 0  1  2]
      [ 3  4  5]
      [ 6  7  8]
      [ 9 10 11]]

     [[12 13 14]
      [15 16 17]
      [18 19 20]
      [21 22 23]]]
```


Arrays konkatenieren

- Arrays können hintereinander gehängt werden. Dabei gibt das optionale, benannte axis-Argument an, in welcher Achse aneinander gehängt werden soll (Default: 0)

Python-Interpreter

```
>>> x = np.array((11, 22))
>>> y = np.array((41, 42))
>>> np.concatenate((x,y))
array([11, 22, 41, 42])
>>> x = np.array(((11, 22),(33,44)))
>>> y = np.array(((99,), (100,)))
>>> np.concatenate((x,y), axis=1)
    array([[ 11,  22,  99],
           [ 33,  44, 100]])
```


Matritzenarithmetik

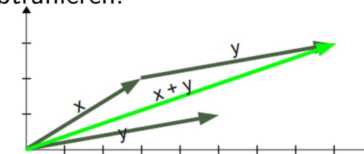
- Innerhalb von NumPy kann man Matritzenarithmetik betreiben.
- Die arithmetischen Standardoperationen $+$, $-$, $*$, $/$, $//$, $**$ und % werden elementweise angewendet.

Python-Interpreter

```
>>> y = np.array([1, 5, 2])
>>> y = np.array([7, 4, 1])
>>> x + y
array([8, 9, 3])
>>> x * y
array([ 7, 20, 2])
>>> x / y
array([0, 1, 2])
>>> x % y
array([1, 1, 0])
```


Vektoraddition und -subtraktion

- Vektoren im euklidischen Raum (2 oder 3-elementige Arrays) kann man addieren und subtrahieren.



Python-Interpreter

```
>>> x = np.array([3,2])
>>> y = np.array([5,1])
>>> z = x + y
>>> z
array([8, 3])
>>> z = x - y
>>> z
array([-2, 1])
```

Skalarprodukt

- Das Skalarprodukt oder innere Produkt ist die Summe der Produkte der Komponenten mit gleichem Index:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

- Der Betrag eines Vektors $\vec{a}$ ergibt sich dann $|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}}$.

Python-Interpreter

```
>>> x = np.array([1,2,3])
>>> y = np.array([-7,8,9])
>>> np.dot(x,y)
36
>>> np.sqrt(np.dot(x,x))
3.7416573867739413
```


Kreuzprodukt / Vektorprodukt

- Das Kreuzprodukt zweier Vektoren ist ein Vektor, der senkrecht auf den beiden Vektoren steht mit einer bestimmten Länge:

$$\vec{a} \times \vec{b} = (|\vec{a}| \cdot |\vec{b}| \cdot \sin \angle(\vec{a}, \vec{b})) \cdot \vec{n},$$

wobei $\vec{n}$ der zu $\vec{a}$ und $\vec{b}$ senkrechte Einheitsvektor ist (Rechte-Hand-Regel).

Python-Interpreter

```
>>> x = np.array([0,0,1])
>>> y = np.array([0,1,0])
>>> np.cross(x,y)
array([-1, 0, 0])
>>> np.cross(y,x)
array([1, 0, 0])
```


Die Matrix-Klasse

- Die Matrix-Klasse ist eine Unterklasse von ndarrays.
- Die Instanzen dieser Klasse haben alle 2 Dimensionen.
- Außerdem bezeichnet $*$ dann das Matrix-Produkt (inneres Produkt für Matrizen!) einer $l \times m$-Matrix mit einer $m \times n$-Matrix, die eine $l \times n$-Matrix wie folgt ergibt: $c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$

Python-Interpreter

```
>>> x = np.array( ((2,3), (3, 5)) )
>>> y = np.array( ((1,2), (5, -1)) )
>>> print(x * y)
[[ 2 6]
 [15 -5]]
>>> x = np.matrix( ((2,3), (3, 5)) )
>>> y = np.matrix( ((1,2), (5, -1)) )
>>> x * y
    matrix([[17,  1],
            [28,  1]])
```


Inverse Matrix

- Mithilfe von numpy.linalg.inv kann man eine quadratische Matrix invertieren (wenn sie invertierbar ist)
- Dann gilt: $A \cdot A^{-1} = E$, wobei E die Einheitsmatrix ist (nur die nur Nullen besitzt außer auf der Hauptdiagonalen, wo sie Einsen hat).
- Die Einheitsmatrix erhält man mit numpy.eye(dim).

Python-Interpreter

```
>>> np.eye(2)
    array([[ 1.,  0.],
           [ 0.,  1.]])

>>> a = np.mat(((3, 5, 1), (1, -1, 2), (2, 0, -1)))
>>> a * np.linalg.inv(a)
    matrix([[  1.00000000e+00,  -5.55111512e-17,   0.00000000e+00],
            [ -8.32667268e-17,   1.00000000e+00,   5.55111512e-17],
            [ -9.71445147e-17,  -5.55111512e-17,   1.00000000e+00]])
```

Lineare Gleichungssysteme (1)

- In dem `linalg`-Paket gibt es eine große Menge von weiteren interessanten Funktionen.
- Unter anderem können lineare Gleichungssysteme gelöst werden.
- Beispielsystem:

$$\begin{aligned} 4x_1 - 2x_2 + 7x_3 &= -7 \\ -x_1 + 5x_2 + 3x_3 + 2x_4 &= 12 \\ -x_1 + 2x_2 + 5x_3 - 2x_4 &= -8 \\ x_1 + 2x_2 + 3x_3 + 4x_4 &= 14 \end{aligned}$$

- Solch ein System kann man unter Benutzung der Matrix-Vektor-Multiplikation schreiben als $A \cdot \vec{x} = \vec{b}$.
- Dann können wir das System `numpy.linalg.solve()` übergeben.

Lineare Gleichungssysteme (2)

- Unser Gleichungssystem in Matrix-Vektor-Multiplikations-Schreibweise:

$$\begin{pmatrix} 4 & -2 & 7 & 0 \\ -1 & 5 & 3 & 2 \\ -1 & 2 & 5 & -2 \\ 1 & 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -7 \\ 12 \\ -8 \\ 14 \end{pmatrix}$$

Python-Interpreter

```
>>> A = np.array(((4,-2,7,0), (-1,5,3,2), (-1,2,5,-2),
(1,2,3,4)))
>>> b = np.array((-7,12,-8,14))
>>> np.linalg.solve(A, b)
array([1., 2., -1., 3.])
```


Polynome

- Mit NumPy lassen sich auch gut Polynome darstellen, manipulieren und man kann mit ihnen rechnen: Der Datentyp `poly1d`.
- Allgemeine Form: $p(x) = \sum_{i=n}^{0} = a_n \cdot x^n + \ldots + a_1 \cdot x + a_0$
- Wir betrachten $p(x) = 3x^2 - 2x - 1$.

Python-Interpreter

```
>>> p = np.poly1d([3,-2,-1])
>>> p(0)
5
>>> p(4)
45
>>> np.roots(p)) # Nullstellen
array([ 1. , -0.33333333])
>>> p.deriv() # Ableitung
poly1d([ 6, -2])
>>> p.integ() # unbestimmtes Integral
poly1d([ 1., -1., -1., 0.])
```


27.4 Funktionswerte zeich nen)

matplotlib im Einsatz

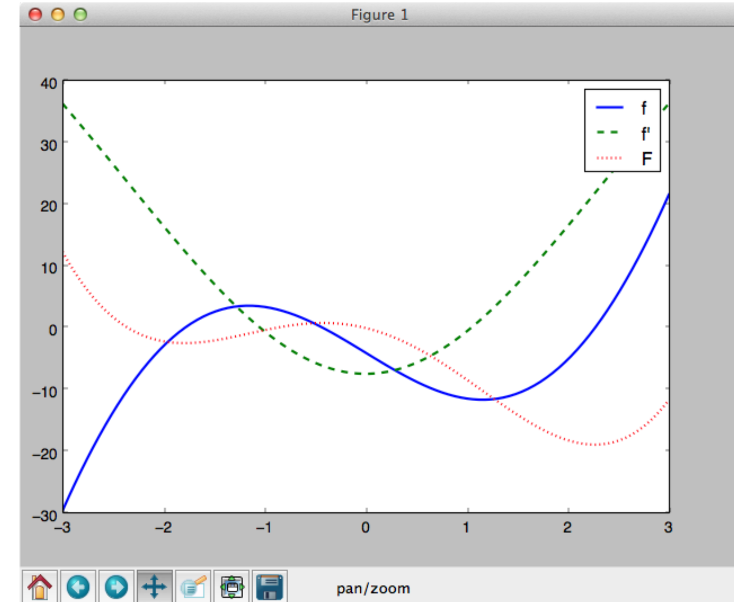
`mplbeispiel.py`

```
import numpy as np; import scipy as sp
import matplotlib as mpl; import matplotlib.pyplot as plt
import scipy.misc; import scipy.integrate

def f(x):
    return x**3 - 10*np.sin(x) - 4
def df(x):
    return sp.misc.derivative(f, x)
@np.vectorize
def F(x):
    return sp.integrate.quad(f, 0, x)[0]
X = np.linspace(-3,3, 100)
Y = f(X)
Y1 = df(X)
Y2 = F(X)

plt.plot(X, Y, linewidth=2, label="f")
plt.plot(X, Y1, linewidth=2, linestyle="dashed", label="f'")
plt.plot(X, Y2, linewidth=2, linestyle="dotted", label="F")
plt.legend()
plt.show()
```


Ausgabe der Plot-Funktion



Die Funktionsdefinitionen

- Die Funktion `f()` ist nicht weiter ungewöhnlich, außer dass sie statt `math.sin()` die Funktion `numpy.sin()` nutzt (wird gleich erklärt).
- Die Funktion `df()` berechnet die erste Ableitung von `f` an der Stelle `x` mit Hilfe der scipy-Funktion `scipy.misc.derivative()`.
- Die Funktion `F()` berechnet den Wert des bestimmten Integrals von 0 bis x über f mit Hilfe der scipy-Funktion `scipy.integrate.quad()`. Diese Funktion liefert ein Paar bestehend aus dem Näherungswertes des Integrals ist und einer oberen Schranke für den Approximationsfehler.

Vektorisierung

- `X` ist ein Array der Länge 100, das von -3 bis +3 läuft:
  `array([-3. , -2.93939394, -2.87878788, ..., 3.])`
- Interessanterweise werden die für Zahlen definierten Funktionen auf dieses Array angewandt!
- Das funktioniert, da alle arithmetischen Operationen und die von `numpy` bereit gestellten Operationen und Funktionen auch Vektoren nehmen und verarbeiten können.
- Für die Stammfunktion F geht das nicht. Aber hier vektorisiert der Dekorator `numpy.vectorize` die Funktion!
- Dies ist langsamer als die eingebaute Unterstützung.

Visualisierung

- Mit `matplotlib.pyplot.plot` kann man beliebige Daten visualisieren.
- Übergibt man eine einfach eine Liste oder ein Array von Zahlen, so werden diese als Funktionswerte der Indizes (0, 1, . . .) aufgefasst werden.
- Übergibt man zwei gleich lange Listen oder Arrays, wird die erste Liste als X-Werte interpretiert.
- Mit den benannten Parametern `linewidth`, `linestyle`, `label` kann man die verschiedenen Kurven unterscheidbar gestalten.
- `matplotlib.pyplot.legend()` erzeugt eine Legende.
- `matplotlib.pylib.show()` erzeugt dann ein Ausgabefenster mit allen Kurven, da in den verschiedensten Grafik-Formaten gespeichert werden kann.

Ausblick

- `SciPy` bietet zusammen mit allen dazugehörigen Modulen eine sehr komfortable und effiziente Umgebung für wissenschaftliches Rechnen – ähnlich zu MATLAB u.ä. Systemen.
- Der Array-Datentyp ist extrem effizient implementiert.
- Arrays erlauben eine große Menge wichtiger Matrix-Operationen.
- Die Vektorisierung erlaubt es, schnell für große Bereiche von Werten Funktionswerte zu erzeugen – und dann mit Hilfe von `matplotlib` zu visualisieren.
- Es gibt in `SciPy` jede Menge von Optimierungsverfahren und Solvern, die man einsetzen kann.