

NUMPY & SCIPY

Numerisch Rechnen mit Python

Carsten Knoll, Institut für Regelungs- und Steuerungstheorie, TU Dresden

Dresden, Wintersemester 2013/14

Vorbemerkungen

- Ziel: grober Überblick über die Möglichkeiten
- Aufbau:
 - Numpy arrays
 - Numpy (grundlegende Numerik)
 - Scipy (anwendungsorientierte Numerik)

Numpy arrays (I)

- Bisher folgende Container Klassen („Sequenzen“) vorgestellt:
Liste ([1, 2, 3]), Tupel ((1, 2, 3)), String ('1, 2, 3')
- Ungeeignet um damit zu rechnen

```
# sinnvoll bei Strings:  
linie = "-" * 10 # -> "-.-.-.-.-.-.-.-.-.-"
```

```
# zum Rechnen nicht geeignet:  
zahlen = [3, 4, 5]  
res1 = zahlen*2 # -> [3, 4, 5, 3, 4, 5]
```

```
# geht gar nicht:  
res2 = zahlen*1.5  
res3 = zahlen**2
```

Numpy arrays (II)

- Bei Numpy Arrays: Berechnungen elementweise

```
from numpy import array
```

```
zahlen = [3.0, 4.0, 5.0] # Liste mit Gleitkommazahlen
```

```
x = array(zahlen)
```

```
res1 = x*1.5 # -> array([ 4.5,  6. ,  7.5])
```

```
res2 = x**2 # -> array([ 9., 16., 25.])
```

```
res3 = res1 - res2 # -> array([-4.5., -10., -18.5])
```

- Arrays können n Dimensionen haben

```
arr_2d = array( [[1., 2, 3], [4, 5, 6]] )*1.0 # ->
```

```
# array( [[1., 2., 3.],
```

```
#          [4., 5., 6.] ] )
```

```
print arr_2d.shape # -> (2, 3)
```

Numpy arrays (III)

Weitere Möglichkeiten arrays zu erstellen:

```
import numpy as np
x0 = np.arange(10) # wie range(...) nur mit arrays
x1 = np.linspace(-10,10, 200)
    # 200 Werte: array([-10., -9.899497, ..., 10])
x2 = np.logspace(1,100, 1e4) # 10k Werte, Immer gleicher Quotient

x3 = np.zeros(10) # np.ones analog
x4 = np.zeros( (3, 5) ) # Achtung: nur ein Argument! (=shape)

x5 = np.eye(3)
x6 = np.diag( (1,2,3) ) # 3x3-Diagonalmatrix

x7 = np.random.rand(5) # array mit 5 Zufallszahlen
x8 = np.random.rand(4,2) # array mit 8 Zufallszahlen (shape = (4,2))

from numpy import r_, c_ # index-Tricks fuer rows und columns
x9 = r_[-1, 2, 3.] # array([ -1.,  2.,  3.])
x10 = r_[x11, -84, x3] # array([ 1.,  2.,  3., -84, 1.,  1.,  1.])
x11 = c_[x5, x11 , x5] # spaltenweise "stapeln" -> 3x3 array
```

Slicing und Broadcasting

- Slicing: Werte in einem Array adressieren
- Analog wie bei anderen Sequenzen: `x[start:stop:step]`
- Dimensionen durch Kommata getrennt; negative Indizes zählen von hinten

```
a = np.arange(18)*2.0
A = np.array( [ [0,1,2,3,4,5], [6,7,8,9,10,11] ] )

x1 = a[3] # Element Nr. "3" (-> 6.0)
x2 = a[3:6] # Elemente 3 bis 5 -> array([ 6.,  8., 10.])
x3 = a[-3:] # Vom 3.-letzten bis Ende -> array([30.,32.,34.])
# Achtung a, x2 und x3 teilen sich die Daten!
a[-2:]*= -1
print x3[ # > [-30.,-32.,-34.]

y1 = A[:, 0] # erste Spalte von A
y2 = A[1, :3] # ersten drei Elemente der zweiten Zeile
```

- $\exists$ „Broadcasting“: Automatisches vergrößern (z.B. Array + Zahl)

„Broadcasting“

- Numpy's Umgang mit arrays mit unterschiedlichen Abmessungen (Elementweise ausgeführte Berechnungen)
- Trivialbeispiel: 2d-array + Skalar → Skalar wird aufgeblasen
- Anderes Beispiel: 2d-array + 1d-array
- Regel: Die Größe entlang der letzten Achsen beider Operanden müssen übereinstimmen oder eine von beiden muss eins sein.
- Zwei Beispiele:

Image	(3d array):	256	256	3	A	(4d array):	8	1	6	1
Scale	(1d array):			3	B	(3d array):		7	1	5
Result	(3d array):	256	256	3	Result	(4d array):	8	7	6	5

- Führt manchmal zu Verwirrung/Problemen:
`ValueError: shape mismatch: objects cannot be broadcast to a single shape`

→ Im Zweifel (interaktiv) ausprobieren

- siehe auch `04_broadcast_beispiel.py`

Numpy Funktionen

```
import numpy as np
from numpy import sin, pi # Tipparbeit sparen

t = np.linspace(0, 2*pi, 1000)

x = sin(t) # analog: cos, exp, sqrt, log, log2, ...
xd = np.diff(x) # numerisch differenzieren
# Achtung: xd hat einen Eintrag weniger!
X = np.cumsum(x) # numerisch "integrieren"
      (kummulativ summieren)
```

- Keine python-Schleifen notwendig → schnell
- Vergleiche:

```
# Elementweise:
y1 = np.arange(3) >= 2
      # -> array([False, False, True], dtype=bool)
# Array-weit:
y2 = np.all( np.arange(3) >= 0) # -> True
y3 = np.any( np.arange(3) < 0) # -> False
```

Weitere Numpy Funktionen

- `min`, `max`, `argmin`, `argmax`, `sum` (→ **Skalare**)
- `abs`, `real`, `imag` (→ **Arrays**)
- **Shape ändern**: `.T` (transponieren), `reshape`, `flatten`, `vstack`, `hstack`

Lineare Algebra:

- **Matrix-Multiplikation**: `dot(a, b)` (empfohlen) oder `matrix(a) * matrix(b)`
- **Submodul**: `numpy.linalg`:
 - `det`, `inv`, `solve` (LGS lösen), `eig` (Eigenwerte u. -vektoren),
 - `pinv` (Pseudoinverse), `svd` (Singulärwertzerlegung), ...

Scipy

- Paket, das auf Numpy aufsetzt
- Bietet Funktionalität für
 - Daten-Ein- u. ausgabe (z.B. mat-Format (Matlab))
 - Physikalische Konstanten
 - Noch mehr lineare Algebra
 - Signalverarbeitung (Fouriertrafo, Filter, ...)
 - Statistik
 - Optimierung
 - Interpolation
 - Numerische Integration („Simulation“)

scipy.optimize

- Besonders nützlich: `fsolve` und `fmin`
- `fsolve`: findet Nullstelle einer skalaren Funktion $f : \mathbb{R} \rightarrow \mathbb{R}$ oder eines (nichtlinearen) Gleichungssystems $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- `fmin`: findet Minimum einer Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Bei beiden: Startschätzung wichtig
- Beispiel: Näherungslösung der Gl. $x + 2.3 * \cos(x) = 1$

```
import numpy as np
from scipy import optimize
```

```
def fnc1(x):
    return x + 2.3*np.cos(x) - 1
```

```
sol = optimize.fsolve(fnc1, 0) # -> array([-0.723632])
# Probe:
sol + 2.3*np.cos(sol) # -> array([ 1.])
```

Numerische Integration von DGLn (Theorie)

- „Simulation“ = numerisches Lösen von Differentialgleichungen
- DGL-Systeme (engl. **O**rdinary **D**ifferential **E**quations) in Zustandsdarstellung:
 $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
- Zeitableitung des Zustands $\mathbf{z}$ hängt vom Zustand selber ab (und von t)
- Lösung der DGL: Zeitverlauf $\mathbf{z}(t)$ (hängt vom Anfangszustand $\mathbf{z}(0)$ ab)

Numerische Integration von DGLn (Theorie)

- „Simulation“ = numerisches Lösen von Differentialgleichungen
- DGL-Systeme (engl. **O**rdinary **D**ifferential **E**quations) in Zustandsdarstellung:
 $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
- Zeitableitung des Zustands $\mathbf{z}$ hängt vom Zustand selber ab (und von t)
- Lösung der DGL: Zeitverlauf $\mathbf{z}(t)$ (hängt vom Anfangszustand $\mathbf{z}(0)$ ab)
- Bsp. harmonischer Oszillator mit DGL: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- Vorbereitung: DGL 2. Ordnung $\rightarrow$ zwei DGLn 1. Ordnung:
Zustand: $\mathbf{z} = (z_1, z_2)^T$ mit $z_1 := y, z_2 := \dot{y} \rightarrow$ zwei DGLn:
 $\dot{z}_1 = z_2$ („definitorische Gleichung“)
 $\dot{z}_2 = -2\delta z_2 - \omega^2 z_1$ ($= \ddot{y}$)
- $\exists$ verschiedene Integrationsalgorithmen (Euler, Runge-Kutta, ...)

Numerische Integration (Umsetzung)

```
from scipy.integrate import odeint
def rhs(z,t):
    """ rhs = 'right hand side [function]' """
    z1, z2 = z # Entpacken
    z1_dot = z2
    z2_dot = -(2*delta*z2+omega_2*z1)
    return [z1_dot, z2_dot]

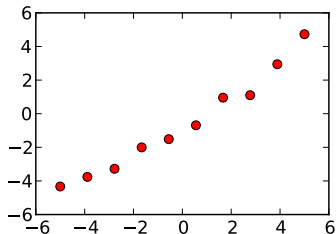
tt = np.arange(0,100,.01) # unabhaengige Variable (Zeit)
z0 = [10,0] # Anfangszustand fuer y, und y_dot
zz = odeint(rhs, z0, tt) # Aufruf des Integrators
```

- `rhs` ist ein „ganz normales“ Objekt
- kann als Argument an eine andere Funktion (`odeint`) übergeben werden

Regression, Interpolation

04_interp_beispiel.py

Regression (`scipy.polyfit`):

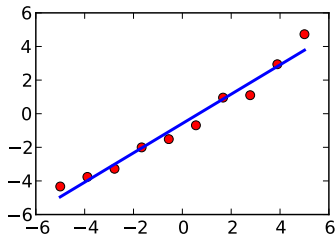


Regression, Interpolation

04_interp_beispiel.py

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)

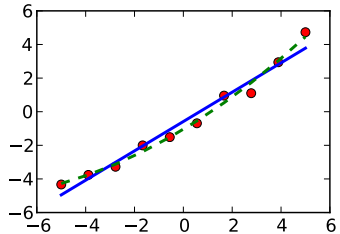


Regression, Interpolation

04_interp_beispiel.py

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung



Regression, Interpolation

04_interp_beispiel.py

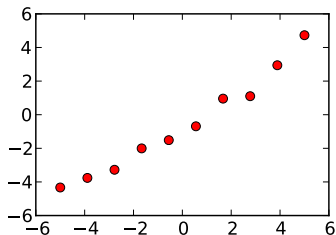
Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)



Regression, Interpolation

04_interp_beispiel.py

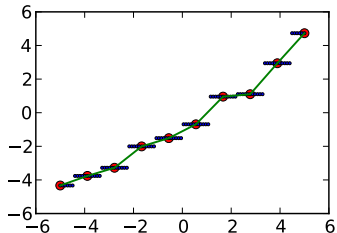
Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)



Regression, Interpolation

04_interp_beispiel.py

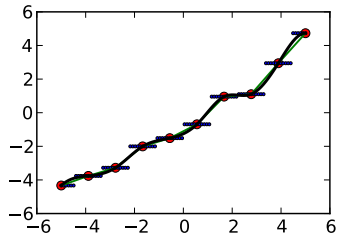
Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)



Regression, Interpolation

04_interp_beispiel.py

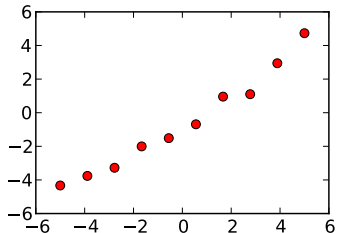
Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)



Mischform (auch `scipy.interpolation`):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)

Regression, Interpolation

04_interp_beispiel.py

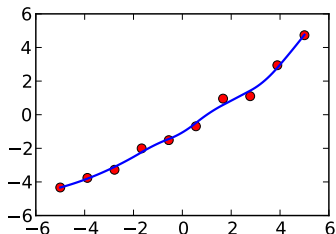
Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)



Mischform (auch `scipy.interpolation`):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)

Regression, Interpolation

04_interp_beispiel.py

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

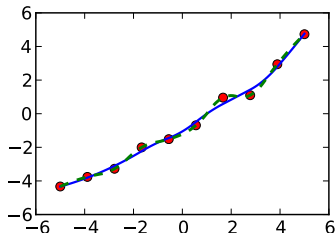
Interpolation

(`scipy.interpolation`):

- Stückweise polynomial (spline)
- (beliebiger Ordnung)

Mischform (auch `scipy.interpolation`):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)



Links

- http://www.scipy.org/Tentative_NumPy_Tutorial
- http://www.scipy.org/NumPy_for_Matlab_Users
- <http://www.scipy.org/EricksBroadcastingDoc>
- http://scipy.org/Numpy_Example_List_With_Doc (umfangreich)

- <http://docs.scipy.org/doc/scipy/reference/> (Tutorial + Referenz)
- <http://www.scipy.org/Cookbook>

Zusammenfassung

- NumPy arrays
- Weitere NumPy Funktionen
- SciPy Funktionen (Numerische Integration, Interpolation + Regression)