TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik Institut für Regelungs- und Steuerungstheorie

PYTHONKURS FÜR INGENIEUR:INNEN

Numerisch Rechnen mit Python - `numpy` und `scipy`

Carsten Knoll

`tu-dresden.de/pythonkurs`

Dresden, 23.11.2020

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

```
# sinvoll 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

```python
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

```python
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, `array`-Objekte zu erstellen:

Listing: 02b_01_arrays_erzeugen.py

```
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) # 10000 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 für rows und columns
x9 = r_[6, 5, 4.] # array([ 6.,  5.,  4.])
x10 = r_[x9, -84, x3[:2]] # array([ 6.,  5.,  4.,  -84, 0.,  1.])
x11 = c_[x9, x6 , x5] # in Spalten-Richtung stapeln -> 7x3 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

Listing: 02b_02_slicing.py

```
import numpy as np
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
```

- ∃ „Broadcasting“: Automatisches vergrößern (z.B. Array + Zahl)

„Broadcasting“

- Numpy’s Umgang mit Arrays mit unterschiedlichen Abmessungen („shapes“) (bei elementweise ausgeführte Berechnungen)
- Trivialbeispiel: x (2d-array) + y (float) $\rightarrow$ y wird auf Shape von x „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:

3d-array*1d-array = 3d-array

`img.shape`	(256,	256,	3)
`scale.shape`			(3,)
`(img*scale).shape`	(256,	256,	3)

4d-array+3d-array = 4d-array

`A.shape`	(8,	1,	6,	1)
`B.shape`		(7,	1,	5)
`(A+B).shape`	(8,	7,	6,	5)

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

$\rightarrow$ Im Zweifel Doku lesen uder (interaktiv) ausprobieren

- siehe auch `02b_03_broadcast_beispiel.py`

Broadcasting-Beispiel

Listing: 02b_03_broadcast_beispiel.py

```
import numpy as np
import time

E = np.ones((4, 3))  # -> shape=(4, 3)
b = np.array([-1, 2, 7])  # -> shape=(3,)
print(E*b)  # -> shape=(4, 3)

b_13 = b.reshape((1, 3))
print(E*b_13)  # -> shape=(4, 3)

print("\n"*2, "Achtung, die nächste Anweisung erzeugt einen Fehler.")
time.sleep(2)

b_31 = b_13.T  # Transponieren -> shape=(3,1)
print(E*b_31) # broadcasting error
```

Erinnerung: `E*b_13` ist **keine** Matrix-Vektormultiplikation (siehe Folie 10)

TECHNISCHE UNIVERSITÄT DRESDEN

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) # num. "integrieren" (kummulativ summieren)
```

- Keine python-Schleifen notwendig → Numpy-Funkt. sind schnell wie C-Code
- Vergleichsoperationen:

```
# 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)` (empfolen)
  - `a@b` (@-Operator in Python 3.5 eingeführt)
  - `np.matrix(a)*np.matrix(b)` (vom Kursleiter nicht empfohlen)
- 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 (Fouriertransformation, Filter, ...)
  - Statistik
  - Optimierung
  - Interpolation
  - Numerische Integration („Simulation“)

scipy.optimize

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

```
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.])
```

Num. 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)

Num. 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 überführung in „Zustandsraumdarstellung“
  (eine DGLn 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“)
  $\dot{z}_2 = -2\delta z_2 - \omega^2 z_1 \quad (= \ddot{y})$
- $\exists$ verschiedene Integrationsalgorithmen (Euler, Runge-Kutta, ...)

Num. 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 überführung in „Zustandsraumdarstellung“
  (eine DGLn 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“)
  $\dot{z}_2 = -2\delta z_2 - \omega^2 z_1 \quad (= \ddot{y})$
- $\exists$ verschiedene Integrationsalgorithmen (Euler, Runge-Kutta, ...)

Ausführliche Erläuterung in separatem Notebook: $\rightarrow$ Simulation dynamischer Systeme.ipynb

Num. Integration (Umsetzung)

Listing: 02b_04_odeint_beispiel.py

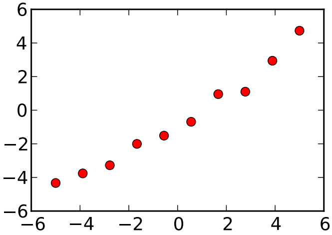
```
import numpy as np
from scipy.integrate import odeint
delta = .1
omega_2 = 2**2
def rhs(z,t):
    """ rhs heißt 'right hand side [function]' """
    z1, z2  = z # Entpacken
    z1_dot = z2
    z2_dot = -(2*delta*z2 + omega_2*z1)
    return [z1_dot, z2_dot]

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

from matplotlib import pyplot as plt
plt.plot(tt, zz[:, 0])
plt.show()
```

- Die Funktion `rhs` ist „ganz normales" Objekt

→ Kann als Argument an eine andere Funktion (hier: `odeint`) übergeben werden

Regression, Interpolation

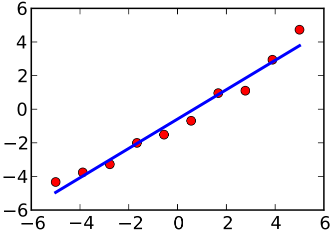
Regression (`scipy.polyfit`):



Regression, Interpolation

Regression (`scipy.polyfit`):

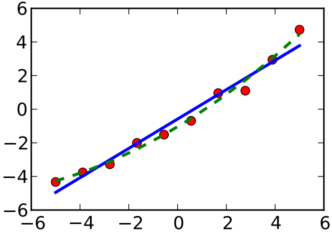
- Regressionsgerade (blau)



Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung



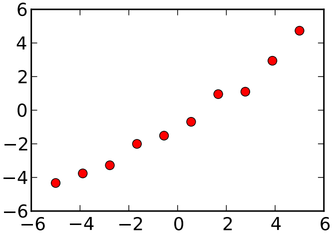
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation
(`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung
  (hier: 0., 1., 2. Ordnung)



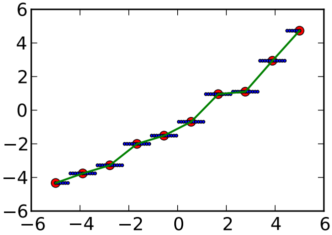
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation (`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung (hier: 0., 1., 2. Ordnung)



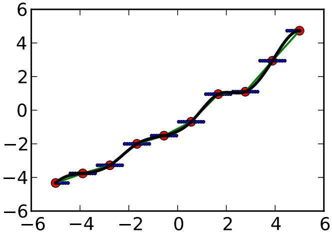
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation (`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung
  (hier: 0., 1., 2. Ordnung)



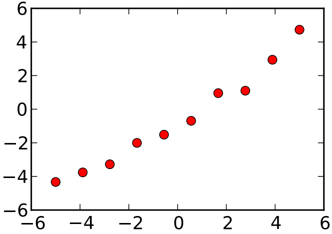
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation (`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung
  (hier: 0., 1., 2. Ordnung)



Mischform (auch `scipy.interpolation`):

- „geglätter Spline“ (Glattheit über Koeffizient einstellbar)

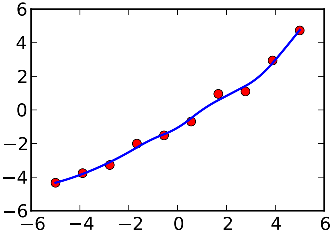
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation (`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung (hier: 0., 1., 2. Ordnung)



Mischform (auch `scipy.interpolation`):

- „geglätter Spline“ (Glattheit über Koeffizient einstellbar)

Siehe auch `02b_05_interp_beispiel.py`

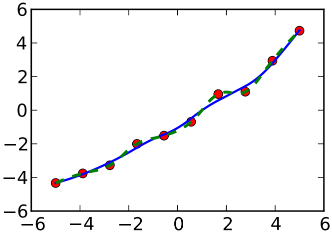
Regression, Interpolation

Regression (`scipy.polyfit`):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation (`scipy.interpolation`) :

- Stückweise polynomial ($\rightarrow$ Spline)
- beliebige Ordnung
  (hier: 0., 1., 2. Ordnung)



Mischform (auch `scipy.interpolation`):

- „geglätter Spline“ (Glattheit über Koeffizient einstellbar)

Siehe auch `02b_05_interp_beispiel.py`

Zusammenfassung

- `numpy`-Arrays
- Weitere `numpy`-Funktionen
- `scipy`-Funktionen (Numerische Integration, Interpolation + Regression)

Links

- http://www.scipy.org/Tentative_NumPy_Tutorial
- http://www.scipy.org/NumPy_for_Matlab_Users
- https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
- http://scipy.org/Numpy_Example_List_With_Doc (umfangreich)

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