TECHNISCHE UNIVERSITÄT DRESDEN

python™

Carsten Knoll

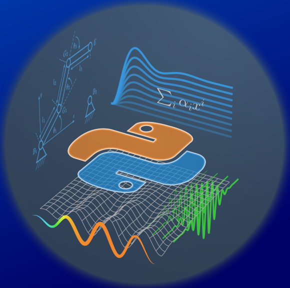
Chair of Fundamentals of Electrical Engineering

Python for Engineers (3)
Pythonkurs für Ingenieur:innen (3)

Numerical Computation with Python - `numpy` and `scipy`
Numerisch Rechnen mit Python - `numpy` und `scipy`

Dresden (Online), 2023-10-24

`https://tu-dresden.de/pythonkurs`
`https://python-fuer-ingenieure.de`



Preliminary Remarks

- Objective of this course:
  - rough overview of the possibilities
  - no completness intended
- Structure:
  - Numpy arrays
  - Numpy (basic numerics)
  - Scipy (application oriented numerics)

Numpy arrays (I)

- So far the following container classes ("sequences") have been presented:
  `list`: `[1, 2, 3]`, `tuple`: `(1, 2, 3)`, `str`: `"1, 2, 3"`
- Unsuitable for calculations

```
# useful for strings
linie = "-." * 10 # -> "-.-.-.-.-.-.-.-.-.-."

# not useful for numerical calculations
numbers = [3, 4, 5]
res1 = numbers*2 # -> [3, 4, 5, 3, 4, 5]

# not possible at all
res2 = numbers*1.5
res3 = numbers**2
```

Numpy arrays (II)

- For Numpy arrays: Calculations are performed **element-wise**

```
from numpy import array

numbers = [3.0, 4.0, 5.0] # list with float objects

x = array(numbers)
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 can have n dimensions

```
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)

Further possibilities to create, `array` objects:

Listing: course04_01_array_creation.py

```
import numpy as np
x0 = np.arange(10) # like range(...) but with arrays
x1 = np.linspace(-10, 10, 200)
        # 200 values: array([-10., -9.899497, ..., 10])
x2 = np.logspace(1, 100, 500) # 500 values, always same ratio

x3 = np.zeros(10) # see also: np.ones(...)
x4 = np.zeros( (3, 5) ) # Caution: takes only **one** argument! (= shape)

x5 = np.eye(4) # 4x4 unity matrix
x6 = np.diag( (4, 3.5, 23) ) # 3x3 diagonal matrix with specified diagonal elements

x7 = np.random.rand(5) # array with 5 random numbers (each between 0 and 1)
x8 = np.random.rand(4, 2) # array with 8 random numbers and shape = (4, 2)

from numpy import r_, c_ # "index tricks" for rows and columns
x9 = r_[6, 5, 4.2] # array([ 6.,  5.,  4.2])
x10 = r_[x9, -84, x3[:2]] # array([ 6.,  5.,  4.2,  -84, 0.,  1.])
x11 = c_[x9, x6 , x5[:-1, :]] # stacking in column direction

assert x11.shape == (3, 8)
```

Slicing and Broadcasting

- slicing: address values inside of an array
- analogous to other sequences: `x[start:stop:step]`;
- first element: `x[0]`
- dimensions are separated by a comma
- negative indices count from the end backwards

Listing: course04_02_slicing.py

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

x1 = a[3] #  element with index 3 (-> 6.0)
x2 = a[3:6] # elements 3 to 5   -> array([  6.,   8.,  10.])
x3 = a[-3:] # from 3rd-last element to the end -> array([30., 32., 34.])
# Caution: a, x2 and x3 share the data (they are only "views" to the data)
a[-2:] *= -1 # change the data in a and observe the change in x3:
print(x3) # -> [-30., -32., -34.]

# for 2d arrays: first index -> row; second index column; separator: comma
y1 = A[:, 0] # first column of A (index 0)
y2 = A[1, :3 ] # first three elements of the second column (index 1)
```

- ∃ “broadcasting”: automatically adapt the shape (e.g. array + scalar)

Broadcasting (Theory)

(optional slide)

- $\hat{=}$ Numpy's handling of arrays with different shapes (for element-wise calculations)
- trivial example: x (2d array) + y (float) $\rightarrow$ y is „blown up“ to match the size of x
- Nontrivial example: 2d array + 1d array = ?
- Rule: The size along the **last axis** of each operand
  a) must be the same or
  b) one of those sizes must be one.
- Two examples:

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)

- Common error message:

```
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

$\rightarrow$ Recommendation: read the docs and experiment interactively

- see also: `course04_03_broadcasting_example.py`

Broadcasting Example

Listing: course04_03_broadcasting_example.py

```python
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, "Caution, the next statements result in an error.")
time.sleep(2)

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

Reminder: `E*b_13` is **not** matrix vector multiplication (see also slide 10).

Numpy functions

```python
import numpy as np
from numpy import sin, pi # Save typing work

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

x = sin(t) # analog: cos, exp, sqrt, log, log2, ...
xd = np.diff(x) # differentiate numerically
# Caution: xd has one entry less!
X = np.cumsum(x) # integrate numerically (cumulative summation)
```

- No python loops needed $\rightarrow$ Numpy functions very are fast (like C code)
- Comparison operations:

```python
# element-wise:
y1 = np.arange(3) >= 2
    # -> array([False, False, True], dtype=bool)
# for complete array:
y2 = np.all( np.arange(3) >= 0) # -> True
y3 = np.any( np.arange(3) < 0) # -> False
```

Further Numpy Functions

(optional slide)

- **`min`**, **`max`**, `argmin`, `argmax`, **`sum`** ($\to$ scalar values)
- **`abs`**, `real`, `imag` ($\to$ arrays)
- change shape: `.T` (transpose), `reshape`, `flatten`, `vstack`, `hstack`

linear algebra:

- matrix multiplication:
  - `a@b` (recommended, @-operator was introduced in Python 3.5)
  - `dot(a, b)` (old and safe way)
  - `np.matrix(a)*np.matrix(b)` (not recommended by me)
- Submodule: `numpy.linalg`:
  - `det`, `inv`, `solve` (solve linear equation system), `eig` (eigenvalues and vectors),
  - `pinv` (pseudo inverse), `svd` (singular value decomposition), ...

Scipy

- Own package, based on Numpy
- Offers functionality for
  – Data input & output (e.g. mat format (Matlab))
  – Physical constants
  – More linear algebra
  – Signal processing (Fourier transform, filter, ...)
  – Statistics
  – Optimization
  – Interpolation
  – Numerical Integration (“Simulation”)

`scipy.optimize` (1)

- Very useful: `fsolve` (docs) and `minimize` (docs)
- `fsolve`: find roots (zeros) of a scalar (nonlinear) function $f : \mathbb{R} \to \mathbb{R}$ or of a vector function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$
- initial estimation of solution is necessary
- example: approximate solution of the nonlinear equation

$$x + 2.3 \cdot \cos(x) \overset{!}{=} 1 \quad \Leftrightarrow \quad x + 2.3 \cdot \cos(x) - 1 \overset{!}{=} 0$$

Listing: course04_06_fsolve_example.py

```python
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])
# check:
print(sol, sol + 2.3*np.cos(sol)) # -> [-0.72363261] [1.]
```

`scipy.optimize` (2)

- `minimize`: finds minimum of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ (docs)
- interface to various minimization algorithms.
- allows many optional arguments
  (e.g. specification of bounds or (in)equation constraints).
- can also solve equations, by minimizing the quadratic **equation error**,
  see following example (same mathematical problem as before, but different solution approach)

Listing: course04_07_minimize_example.py

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

def fnc2(x):
    return (x + 2.3*np.cos(x) - 1)**2 # quadratic equation error

res = optimize.minimize(fnc2, 0) # Optimization with initial estimate 0
# check:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [-0.7236326] [1.00000004]

# now with limits and with changed start estimation -> other solution
res = optimize.minimize(fnc2, 0.5, bounds=[(0, 3)])
# check:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [2.03999505] [1.00000003]
```

Num. Integration of ODEs (Theory)

- "simulation" = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
  - time derivative $\dot{\mathbf{z}}$ of the state vector $\mathbf{z}$ depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
  $\rightarrow$ "**i**nitial **v**alue **p**roblem" (IVP), (German: Anfangswertproblem)

Num. Integration of ODEs (Theory)

- “simulation” = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
  - time derivative $\dot{\mathbf{z}}$ of the state vector $\mathbf{z}$ depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
  $\rightarrow$ “**i**nitial **v**alue **p**roblem” (IVP), (German: Anfangswertproblem)
- example: harmonic oscillator with ODE: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- preparation: transformation to state space representation
  (one ODE of 2nd order $\rightarrow$ two ODEs of 1st order):
  State: $\mathbf{z} = (z_1, z_2)^T$ with $z_1 := y$, $z_2 := \dot{y} \rightarrow$ two ODEs:

$$\begin{aligned} \dot{z}_1 &= z_2 && \text{(“definitional equation”)} \\ \dot{z}_2 &= -2\delta z_2 - \omega^2 z_1 && (= \ddot{y}) \end{aligned}$$

- $\exists$ various integration algorithms (Euler, Runge-Kutta, ...)
- accessible via universal function: `scipy.integrate.solve_ivp` (docs)

Num. Integration of ODEs (Theory)

- "simulation" = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
  - time derivative $\dot{\mathbf{z}}$ of the state vector $\mathbf{z}$ depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
  $\rightarrow$ "**i**nitial **v**alue **p**roblem" (IVP), (German: Anfangswertproblem)
- example: harmonic oscillator with ODE: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- preparation: transformation to state space representation
  (one ODE of 2nd order $\rightarrow$ two ODEs of 1st order):
  State: $\mathbf{z} = (z_1, z_2)^T$ with $z_1 := y$, $z_2 := \dot{y} \rightarrow$ two ODEs:

$$\begin{aligned} \dot{z}_1 &= z_2 && \text{("definitional equation")} \\ \dot{z}_2 &= -2\delta z_2 - \omega^2 z_1 && (= \ddot{y}) \end{aligned}$$

- $\exists$ various integration algorithms (Euler, Runge-Kutta, ...)
- accessible via universal function: `scipy.integrate.solve_ivp` (docs)

detailed explanations in separate notebook: $\rightarrow$ simulation_of_dynamical_systems.ipynb

Num. Integration of ODEs (Application)

Listing: course04_04_solve_ivp_example.py

```
import numpy as np
from scipy.integrate import solve_ivp

delta = .1
omega_2 = 2**2
def rhs(t, z):
    """ rhs means 'right hand side [function]' """
    # argument t must me present in the function head, but it can be ignored in the body
    z1, z2  = z # unpacking the state vector (array) into its two components
    z1_dot = z2
    z2_dot = -(2*delta*z2 + omega_2*z1)
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # independent variable (time)
z0 = [10, 0]   # initial state for z1 and z2 (=y, and y_dot)
res = solve_ivp(rhs, (tt[0], tt[-1]), z0, t_eval=tt) # calling the integration algorithm
zz = res.y # array with the time-development of the state
           # (rows: components; columns: time steps)

from matplotlib import pyplot as plt
plt.plot(tt, zz[0, :]) # plot z1 over t
plt.show()
```

- function `rhs` is “ordinary” Python object

→ Can be passed as argument to another function (here: `solve_ivp`)

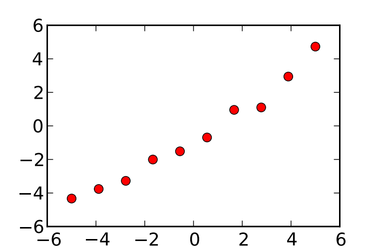
- note: function `odeint` is predecessor of `solve_ivp`.

  main difference: argument order and return object

Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` (docs):

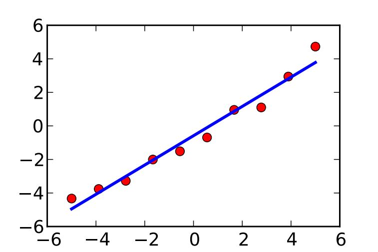


Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` (docs):

- linear regression (blue)

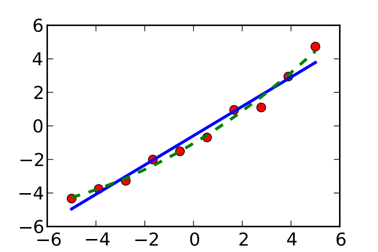


Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order



Regression, Interpolation

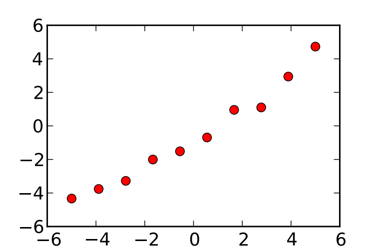
(optional slide)

regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with `scipy.interpolation.interp1d` (docs):

- piecewise polynomial (→ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



Regression, Interpolation

(optional slide)

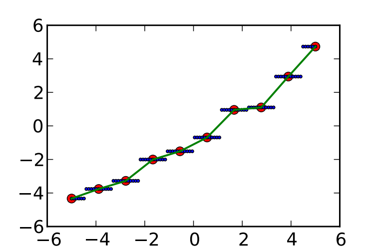
regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` (docs):

- piecewise polynomial (→ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



Regression, Interpolation

(optional slide)

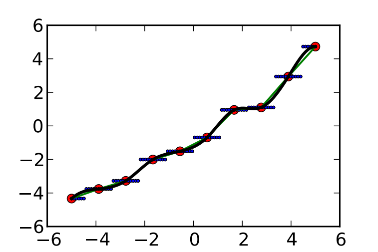
regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` (docs):

- piecewise polynomial (→ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



Regression, Interpolation

(optional slide)

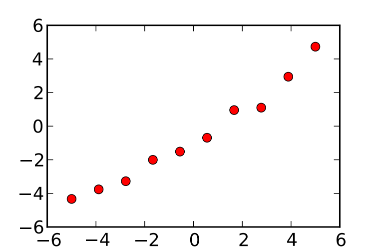
regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` (docs):

- piecewise polynomial (→ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



hybrid form with `scipy.interpolation.splrep` (docs):

- “smoothed spline” (smoothness adjustable via coefficient)

Regression, Interpolation

(optional slide)

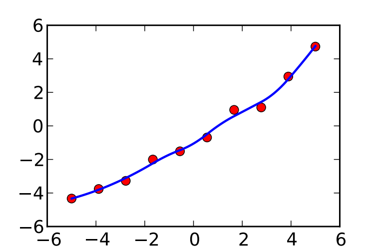
regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` (docs):

- piecewise polynomial ($\rightarrow$ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



hybrid form with `scipy.interpolation.splrep` (docs):

- “smoothed spline” (smoothness adjustable via coefficient)

see also: `course04_05_interp_example.py`

Regression, Interpolation

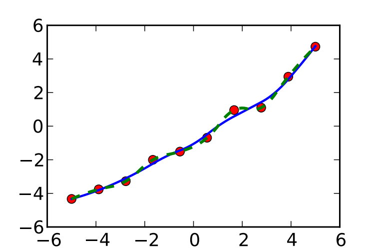
(optional slide)

regression with `numpy.polyfit` (docs):

- linear regression (blue)
- or higher order

interpolation with `scipy.interpolation.interp1d` (docs):

- piecewise polynomial ($\rightarrow$ „Spline“)
- arbitrary order
  (here: orders 0, 1, 2)



hybrid form with `scipy.interpolation.splrep` (docs):

- “smoothed spline” (smoothness adjustable via coefficient)

see also: `course04_05_interp_example.py`

Summary

- `numpy` arrays
- further `numpy`-functions
- `scipy` functions (numerical integration, optimization, regression, interpolation)

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` (extensive)

- `http://docs.scipy.org/doc/scipy/reference/` (tutorial + reference)
- `http://www.scipy.org/Cookbook`