



TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Elektrotechnik und Informationstechnik Institut für Regelungs- und Steuerungstheorie

INTRODUCTION TO SCIENTIFIC PROGRAMMING WITH PYTHON

Carsten Knoll

Dresden, 07.10.2017



DRESDEN
concept
Expanding your
Microelectronic
and Power

Outline

Introduction

Symbolic Computation (`sympy`)

Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

About

About me

- Post-doc at [Institute of Control Theory](#)
- Co-Founder of [University Group for Free Software and Free Knowledge](#)
- “Infected” with Python in 2004, active usage since 2008

About

About me

- Post-doc at [Institute of Control Theory](#)
- Co-Founder of [University Group for Free Software and Free Knowledge](#)
- “Infected” with Python in 2004, active usage since 2008

About this course


- Truncated, compressed and translated version of a [one-semester-course](#)
- Formerly organized by Sebastian Voigt, Christoph Statz, Dr.-Ing. Ines Gubsch, Ingo Keller, Peter Seifert and others
- Modules
 1. Introduction to Python
 2. Symbolic computation
 3. Numeric computation
 4. 2d visualization

About

About me

- Post-doc at [Institute of Control Theory](#)
- Co-Founder of [University Group for Free Software and Free Knowledge](#)
- “Infected” with Python in 2004, active usage since 2008

About this course

- Truncated, compressed and translated version of a [one-semester-course](#)
- Formerly organized by Sebastian Voigt, Christoph Statz, Dr.-Ing. Ines Gubsch, Ingo Keller, Peter Seifert and others
- Modules
 1. Introduction to Python
 2. Symbolic computation
 3. Numeric computation
 4. 2d visualization
-  Contains didactic simplifications

Why Python? (1)

Python as Programming Language

- Clean readable syntax (few overhead)
- Allowed paradigms object oriented, procedural, functional
- Useful and powerful standard data types (`list`, `tuple`, `dict`, `set`, ...)
- Extensive and powerful standard library (batteries included)
- Simple modularization (`import this`)
- Good error handling (Exceptions)
- Simple integration of external code (C, C++, Fortran)

Why Python? (1)

Python as Programming Language

- Clean readable syntax (few overhead)
- Allowed paradigms object oriented, procedural, functional
- Useful and powerful standard data types (`list`, `tuple`, `dict`, `set`, ...)
- Extensive and powerful standard library (batteries included)
- Simple modularization (`import this`)
- Good error handling (Exceptions)
- Simple integration of external code (C, C++, Fortran)

⇒

- Easy to learn
- Problem oriented (powerful *and* flexible)
- Motivation ↗, frustration ↘

Additionally: platform independent / free and open source / big and active Community

Why Python? (2)

Python as a tool for scientists

- Symbolic computation (basic algebra, differentiation, solving equations, ...)
- Numeric computation Rechnen (lin. algebra, solving ODEs, optimization, ...)
- Visualization (2D, 3D, in publication quality)
- Grafical user interface (GUI)
- Communication with external (lab) devices (RS232, GPIB, ...)
- Parallelization

Installation: instal-script.txt

- Different methods available (depending on operating system)
- Recommendation: <https://repo.continuum.io/miniconda/>

```
# paste these commands in your terminal line by line
# use CTRL-SHIFT-V

# download the installer
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O
    miniconda.sh

bash ./miniconda.sh -b # execute it in batch mode

# add miniconda3 to system path
echo "export PATH=\"$HOME/miniconda3/bin:\$PATH\"" >> ~/.bashrc

# install important packages via conda
conda install --yes ipython jupyter sympy numpy scipy matplotlib
# install IDE (spyder) and personal debug-helper
pip install spyder ipydex
```

- Background information:
 - We use Python 3.6 (most recent version available with miniconda)
 - Python 3.x is **not** 100% backward compatible, ∃ still much 2.7-code

Possibilities to Invoke Python

- Default interactive interpreter
 - Command: `$ python` or `$ python3`
 - Can execute every Python-command
 - Might be useful as calculator

Possibilities to Invoke Python

- Default interactive interpreter
 - Command: `$ python` or `$ python3`
 - Can execute every Python-command
 - Might be useful as calculator
- IPython (“Enhanced Interactive Python” → **much** better)
 - Command: `$ ipython`
 - Tab completion, smart history, dynamic object information, colored exception tracebacks

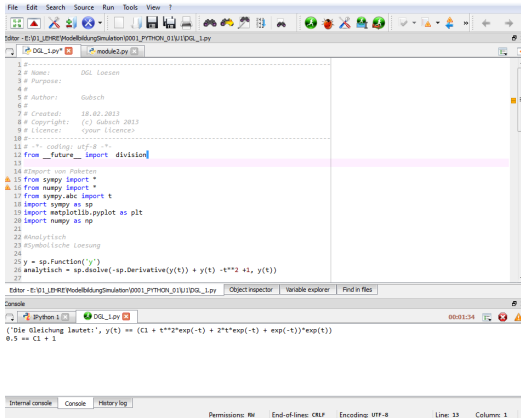
Possibilities to Invoke Python

- Default interactive interpreter
 - Command: `$ python` or `$ python3`
 - Can execute every Python-command
 - Might be useful as calculator
- IPython (“Enhanced Interactive Python” → **much** better)
 - Command: `$ ipython`
 - Tab completion, smart history, dynamic object information, colored exception tracebacks
- Jupyter Notebook (with Python Kernel)
 - Command: `$ jupyter notebook`
 - Backend: (local) webserver; frontend: interactive document in browser
 - Notebooks combine source code, program-output and documentation (incl. \LaTeX -formulas)

Possibilities to Invoke Python

- Default interactive interpreter
 - Command: `$ python` or `$ python3`
 - Can execute every Python-command
 - Might be useful as calculator
- IPython (“Enhanced Interactive Python” → **much** better)
 - Command: `$ ipython`
 - Tab completion, smart history, dynamic object information, colored exception tracebacks
- Jupyter Notebook (with Python Kernel)
 - Command: `$ jupyter notebook`
 - Backend: (local) webserver; frontend: interactive document in browser
 - Notebooks combine source code, program-output and documentation (incl. \LaTeX -formulas)
- Integrated Development Environment (we use spyder)
 - Command: `$ spyder3`
 - Adapted text editor
 - Much more features (we wont use today)

Integrated Development Environment (IDE)



The screenshot shows the Spyder IDE interface. The main editor window displays a Python script with the following content:

```
1 #-----
2 # Name:      DGL Loesen
3 # Purpose:
4 #
5 # Author:    Gubisch
6 #
7 # Created:   18.02.2013
8 # Copyright: (c) Gubisch 2013
9 # Licence:   <your licence>
10 #-----
11 # -*- coding: utf-8 -*-
12 from __future__ import division
13
14 #Import von Paketen
15 from sympy import *
16 from numpy import *
17 from sympy.abc import t
18 import sympy as sp
19 import matplotlib.pyplot as plt
20 import numpy as np
21
22 #Analytisch
23 #Symbolische Loesung
24
25 y = sp.Function('y')
26 analytisch = sp.dsolve(-sp.Derivative(y(t)) + y(t) - t**2 + 1, y(t))
27
```

The console window at the bottom shows the output of the script:

```
{ 'Die Gleichung lautet:', y(t) == (C1 + t**2*exp(-t) + 2*t*exp(-t) + exp(-t))*exp(t) }
0.5 == C1 + 1
```

- Start from command line:
\$ spyder
- Other text editors also possible
- **Important:** Indentation: 4 spaces
- Avoid Tab-characters

Code-Example

Listing: hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

Code-Example

Listing: hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

- Indention have **syntactical** meaning
- de facto standard: 4 spaces (in IDE: block wise with <Tab>(→) and <Shift+Tab> (←))
- [01_appendix-syntax-and-types.pdf](#)

Exercise: Embedded IPython

Listing: ipython1.py

```
import math

# import embedded shell
from ipydx import IPS

a = 10
b = 20.5
c = a + b + 3**2
d = math.sqrt(c)

# run embedded shell
IPS()

# try: math.sqrt?, math.s<TAB>, history (up, down), %magic
# exit with CTRL-D
```

Outline

Introduction

Symbolic Computation (`sympy`)

Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

sympy Overview

Listing: sympy1.py 1:18

```
import sympy as sp
x = sp.Symbol("x")
a, b, c, z = sp.symbols("a b c z") # create several symbols at once

some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b**2))
print(some_formula) # -> -b*c*(-1/(2*b) + 2*a*b*x/c) + 2*a*x*b**2
print(some_formula.expand()) # -> c/2

# some calculus
y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a)
print(y) # -> a**(1/2)*exp(x)*sin(x)

# derive
y.diff(x) # -> 3*sqrt(a)*exp(3*x)*sin(x) + sqrt(a)*exp(3*x)*cos(x)

# trigonometric simplification
print(sp.trigsimp(sp.sin(x)**2+sp.cos(x)**2)) # -> 1
```

Substitute: `<expr>.subs(...)`

- Comparable with `<str>.replace(old, new)`
- Useful for: coordinate transformation manual simplification, (partial) function evaluation
- Returns new expression, original unchanged

Listing: sympy1.py 21:28

```
# 2 arguments:  
# substitute x with z-1  
y2 = y.subs(x, z-1)  
  
# 1 argument (list of 2-tuples):  
# substitute x with z-1 and a with 25 (in this order)  
y3 = y.subs([(x, z-1), (a, 25)])
```

Substitute: `<expr>.subs(...)`

- Comparable with `<str>.replace(old, new)`
- Useful for: coordinate transformation manual simplification, (partial) function evaluation
- Returns new expression, original unchanged

Listing: sympy1.py 21:28

```
# 2 arguments:  
# substitute x with z-1  
y2 = y.subs(x, z-1)  
  
# 1 argument (list of 2-tuples):  
# substitute x with z-1 and a with 25 (in this order)  
y3 = y.subs([(x, z-1), (a, 25)])
```

- Exercise: Use `IPS()` to explore the content of the variables

Numerical Formula Evaluation (lambdify)

Listing: sympy2.py

```
import sympy as sp
from ipydx import IPS

a, b, c, x = sp.symbols("a b c x") # create symbols

# define expression
f = a*sp.sin(b*x)

# create python-function
f_xa_fnc = sp.lambdify((a, b, x), f, modules="numpy")

# evaluate function
print(f_xa_fnc(1.2, 0.5, 3.14))

IPS()

# use magic commands to investigate speedup
# %time f_xa_fnc(1, 1, 1)
# %time f.subs([(a, 1), (b, 1), (x, 1)]).evalf()
```

More Important Methods / Functions / Types

- `sp.Matrix([[x, a+b], [c*x, sp.sin(x)]])`: Matrices
- `<mtrx>.jacobian(xx)`: Jacobian of a vector
- `sp.solve(x**2 + x - a, x)`: solve equations and systems of equations
- `<expr>.atoms()`, `<expr>.atoms(sp.sin)`: get "atoms" (all or specific ones)
- `<expr>.args`: arguments of the respective class (terms of the sum, factors)

- `sp.simplify(...)`: adapt data types
- `sp.integrate(<expr>, <var>)`: integration
- `sp.series(...)`: series expansion
- `sp.limit(<var>, <value>)`: limit
- `<expr>.as_num_denom()`: get numerator and denominator
- `sp.Polynomial(x**7+a*x**3+b*x+c, x, domain='EX')`: polynomials
- `sp.Piecewise(...)`: piecewise defined functions

Outline

Introduction

Symbolic Computation (`sympy`)

Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

Jupyter



- Web application for interactive data science and scientific computing
- Start with:

```
$ cd notebooks  
$ jupyter notebook ./
```

- Important keyboard shortcuts:

Command Mode (press **Esc** to enable)

- **Shift-Return** - execute cell, activate next
- **h** - show keyboard shortcuts
- **m** - change cell type to markdown
- **y** - change cell type to code
- **a** - new cell above

Edit Mode (press **Return** to enable)

- **Shift-Return** - execute cell, activate next
- **Tab** - code-completion or indent
- **Shift-Tab** - tooltip
- **Ctrl-Z** - undo

Jupyter



- Web application for interactive data science and scientific computing
- Start with:

```
$ cd notebooks  
$ jupyter notebook ./
```

- Important keyboard shortcuts:

Command Mode (press **Esc** to enable)

- **Shift-Return** - execute cell, activate next
- **h** - show keyboard shortcuts
- **m** - change cell type to markdown
- **y** - change cell type to code
- **a** - new cell above

Edit Mode (press **Return** to enable)

- **Shift-Return** - execute cell, activate next
- **Tab** - code-completion or indent
- **Shift-Tab** - tooltip
- **Ctrl-Z** - undo

→ Now play around with `example-notebook1.ipynb` and `sympy-notebook1.ipynb`

Outline

Introduction

Symbolic Computation (`sympy`)

Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

Numpy

- Basic numeric algorithms around a special data type:
- `numpy.ndarray`
- No (slow) python loops → almost as fast as C or Fortran
- Explained with material from datacamp.com





Lists Recap

- Powerful
- Collection of values
- Hold different types
- Change, add, remove
- Need for Data Science
 - Mathematical operations over collections
 - Speed



Illustration

```
In [1]: height = [1.73, 1.68, 1.71, 1.89, 1.79]
```

```
In [2]: height
```

```
Out[2]: [1.73, 1.68, 1.71, 1.89, 1.79]
```

```
In [3]: weight = [65.4, 59.2, 63.6, 88.4, 68.7]
```

```
In [4]: weight
```

```
Out[4]: [65.4, 59.2, 63.6, 88.4, 68.7]
```

```
In [5]: weight / height ** 2
```

```
TypeError: unsupported operand type(s) for **: 'list' and 'int'
```



Solution: NumPy

- Numeric Python
- Alternative to Python List: NumPy Array
- Calculations over entire arrays
- Easy and Fast
- Installation
 - In the terminal: `pip3 install numpy`



NumPy

```
In [6]: import numpy as np
```

```
In [7]: np_height = np.array(height)
```

```
In [8]: np_height
```

```
Out[8]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])
```

```
In [9]: np_weight = np.array(weight)
```

```
In [10]: np_weight
```

```
Out[10]: array([ 65.4,  59.2,  63.6,  88.4,  68.7])
```

```
In [11]: bmi = np_weight / np_height ** 2
```

```
In [12]: bmi
```

```
Out[12]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```




NumPy

```
In [6]: import numpy as np
```

Element-wise calculations

```
In [7]: np_height = np.array(height)
```

```
In [8]: np_height
```

```
Out[8]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])
```

```
In [9]: np_weight = np.array(weight)
```

```
In [10]: np_weight
```

```
Out[10]: array([ 65.4,  59.2,  63.6,  88.4,  68.7])
```

```
In [11]: bmi = np_weight / np_height ** 2
```

```
In [12]: bmi
```

```
Out[12]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```

```
= 65.5/1.73 ** 2
```



Comparison

```
In [13]: height = [1.73, 1.68, 1.71, 1.89, 1.79]
```

```
In [14]: weight = [65.4, 59.2, 63.6, 88.4, 68.7]
```

```
In [15]: weight / height ** 2
```

```
TypeError: unsupported operand type(s) for **: 'list' and 'int'
```

```
In [16]: np_height = np.array(height)
```

```
In [17]: np_weight = np.array(weight)
```

```
In [18]: np_weight / np_height ** 2
```

```
Out[18]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```



NumPy: remarks

```
In [19]: np.array([1.0, "is", True])  
Out[19]:  
array(['1.0', 'is', 'True'],  
      dtype='<U32')
```

NumPy arrays: contain only one type

```
In [20]: python_list = [1, 2, 3]
```

```
In [21]: numpy_array = np.array([1, 2, 3])
```

```
In [22]: python_list + python_list  
Out[22]: [1, 2, 3, 1, 2, 3]
```

Different types: different behavior!

```
In [23]: numpy_array + numpy_array  
Out[23]: array([2, 4, 6])
```



NumPy Subsetting

```
In [24]: bmi
Out[24]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])

In [25]: bmi[1]
Out[25]: 20.975

In [26]: bmi > 23
Out[26]: array([False, False, False,  True, False], dtype=bool)

In [27]: bmi[bmi > 23]
Out[27]: array([ 24.747])
```



INTRO TO PYTHON FOR DATA SCIENCE

2D NumPy Arrays



Type of NumPy Arrays

```
In [1]: import numpy as np
```

```
In [2]: np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
In [3]: np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
```

```
In [4]: type(np_height)
```

```
Out[4]: numpy.ndarray
```

ndarray = N-dimensional array

```
In [5]: type(np_weight)
```

```
Out[5]: numpy.ndarray
```



2D NumPy Arrays

```
In [6]: np_2d = np.array([[1.73, 1.68, 1.71, 1.89, 1.79],  
                           [65.4, 59.2, 63.6, 88.4, 68.7]])
```

```
In [7]: np_2d
```

```
Out[7]:
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],  
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
In [8]: np_2d.shape
```

```
Out[8]: (2, 5)
```

2 rows, 5 columns

```
In [9]: np.array([[1.73, 1.68, 1.71, 1.89, 1.79],  
                  [65.4, 59.2, 63.6, 88.4, "68.7"]])
```

```
Out[9]:
```

```
array([[ '1.73', '1.68', '1.71', '1.89', '1.79'],  
       [ '65.4', '59.2', '63.6', '88.4', '68.7']],  
      dtype='<U32')
```

Single type!



Subsetting

	0	1	2	3	4	
array([[1.73,	1.68,	1.71,	1.89,	1.79],	0
[65.4,	59.2,	63.6,	88.4,	68.7]])	1

```
In [10]: np_2d[0]
```

```
Out[10]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])
```

```
In [11]: np_2d[0][2]
```

```
Out[11]: 1.71
```

```
In [12]: np_2d[0,2]
```

```
Out[12]: 1.71
```




Subsetting

	0	1	2	3	4	
array([[1.73,	1.68,	1.71,	1.89,	1.79],	0
[65.4,	59.2,	63.6,	88.4,	68.7]])	1

```
In [10]: np_2d[0]
```

```
Out[10]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])
```

```
In [11]: np_2d[0][2]
```

```
Out[11]: 1.71
```

```
In [12]: np_2d[0,2]
```

```
Out[12]: 1.71
```

```
In [13]: np_2d[:,1:3]
```

```
Out[13]:
```

```
array([[ 1.68,  1.71],
       [ 59.2,  63.6 ]])
```



Subsetting

	0	1	2	3	4	
array([[1.73,	1.68,	1.71,	1.89,	1.79],	0
[65.4,	59.2,	63.6,	88.4,	68.7]])	1

```
In [10]: np_2d[0]
```

```
Out[10]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])
```

```
In [11]: np_2d[0][2]
```

```
Out[11]: 1.71
```

```
In [12]: np_2d[0,2]
```

```
Out[12]: 1.71
```

```
In [13]: np_2d[:,1:3]
```

```
Out[13]:
```

```
array([[ 1.68,  1.71],  
       [ 59.2,  63.6]])
```

```
In [14]: np_2d[1,:]
```

```
Out[14]: array([ 65.4,  59.2,  63.6,  88.4,  68.7])
```

Broadcasting

- How numpy manages different shapes in element-wise calculations
- Trivial example: 2d-array + scalar \rightarrow scalar is “blown up”
- 2d-array + 1d-array ?

Broadcasting

- How numpy manages different shapes in element-wise calculations
- Trivial example: 2d-array + scalar → scalar is “blown up”
- 2d-array + 1d-array ?

General Rule

Length along the last *axis* of both operands must match or one of both equals 1.

Broadcasting

- How numpy manages different shapes in element-wise calculations
- Trivial example: 2d-array + scalar → scalar is “blown up”
- 2d-array + 1d-array ?

General Rule

Length along the last *axis* of both operands must match or one of both equals 1.

- Example:

RGB-Image	(3d array):	256	256	3
Scale	(1d array):			3
Result	(3d array):	256	256	3

Broadcasting

- How numpy manages different shapes in element-wise calculations
- Trivial example: 2d-array + scalar → scalar is “blown up”
- 2d-array + 1d-array ?

General Rule

Length along the last *axis* of both operands must match or one of both equals 1.

- Example:

RGB-Image	(3d array):	256	256	3
Scale	(1d array):			3
Result	(3d array):	256	256	3

- Powerful but not intuitive → possible confusion

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

→ Recommendation interactive trial and error

List of important numpy-functions

- `arange`, `linspace` (create arrays)
- `min`, `max`, `argmin`, `argmax`, `sum` (array \rightarrow scalars)
- `sin`, `cos`, `exp`, `abs`, `real`, `imag` (array \rightarrow array)
- Change shape: `<arr>.T` (transpose), `reshape`, `flatten`, `vstack`, `hstack`

Linear Algebra

- Matrix multiplication: `dot(a,b)`
- Submodule: `numpy.linalg`:
 - `det`, `inv`, `solve` (lin. eq. system), `eig` (Eigenvalues and vectors),
 - `pinv` (pseudoinverse), `svd` (singular value decomposition), ...

Package scipy

- Based on numpy
- Offers functionality for
 - Physical constants
 - More linear algebra
 - Signal processing (FFT, Filter, ...)
 - Statistics
 - Optimization
 - Interpolation
 - Numerical integration ("simulation")

scipy.optimize.fsolve and fmin

- **fsolve**: find (one) root of a (nonlinear) function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- Important: initial guess
- Example (with $n = 12$): approximate solution of equation $x + 2.3 \cdot \cos(x) = 1$
- $f(x) = x + 2.3 \cdot \cos(x) - 1 \stackrel{!}{=} 0$

Listing: scipy1.py

```
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])
# proof:
sol + 2.3*np.cos(sol) # -> array([ 1.] )
```

scipy.optimize.fsolve and fmin

- **fsolve**: find (one) root of a (nonlinear) function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- Important: initial guess
- Example (with $n = 12$): approximate solution of equation $x + 2.3 \cdot \cos(x) = 1$
- $f(x) = x + 2.3 \cdot \cos(x) - 1 \stackrel{!}{=} 0$

Listing: scipy1.py

```
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])
# proof:
sol + 2.3*np.cos(sol) # -> array([ 1.] )
```

- **fmin**: find minimum of $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Exercise: reproduce the the above result using **fmin**

Theory: Numerical solution of ODEs ("Simulation")

- State space representation
 - System of first order ODEs
$$\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$$
 - Solution (flow) $t \mapsto \mathbf{z}(t)$ (depends on initial condition $\mathbf{z}(0)$)

Theory: Numerical solution of ODEs ("Simulation")

- State space representation
 - System of first order ODEs
$$\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$$
 - Solution (flow) $t \mapsto \mathbf{z}(t)$ (depends on initial condition $\mathbf{z}(0)$)
- Example harmonic oscillator: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- Preparation: ODE of 2nd order \rightarrow two ODEs 1st order:
state: $\mathbf{z} = (z_1, z_2)^T$ with $z_1 := y, z_2 := \dot{y}$
$$\begin{aligned}\dot{z}_1 &= z_2 \\ \dot{z}_2 &= -2\delta z_2 - \omega^2 z_1 \quad (= \ddot{y})\end{aligned}$$
- \exists several integration algorithms (Euler, Runge-Kutta, ...)

Practice: Numerical solution of ODEs ("Simulation")

Listing: scipy2.py

```
from scipy.integrate import odeint
import numpy as np
delta, omega = .1, 1

def rhs(z,t):
    """ rhs = right hand side [function] """
    z1, z2 = z # unpacking state
    z1_dot = z2
    z2_dot = -2*delta*z2 - omega**2*z1
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # 100s dt = 0.1
z0 = [3, 0] # initial state for [z1, z2]
zz = odeint(rhs, z0, tt) # call integration algorithm
```

- Notice: `rhs` is an "ordinary" object (type: `function`)

Practice: Numerical solution of ODEs ("Simulation")

Listing: scipy2.py

```
from scipy.integrate import odeint
import numpy as np
delta, omega = .1, 1

def rhs(z,t):
    """ rhs = right hand side [function] """
    z1, z2 = z # unpacking state
    z1_dot = z2
    z2_dot = -2*delta*z2 - omega**2*z1
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # 100s dt = 0.1
z0 = [3, 0] # initial state for [z1, z2]
zz = odeint(rhs, z0, tt) # call integration algorithm
```

- Notice: `rhs` is an "ordinary" object (type: `function`)
- Exercises:
 - a) Print the shape of the array `zz`
 - b) Plot the function $t \mapsto y(t)$

Practice: Numerical solution of ODEs ("Simulation")

Listing: scipy2.py

```
from scipy.integrate import odeint
import numpy as np
delta, omega = .1, 1

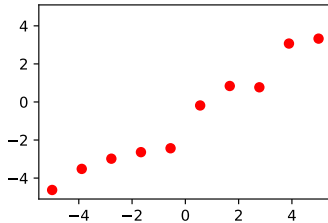
def rhs(z,t):
    """ rhs = right hand side [function] """
    z1, z2 = z # unpacking state
    z1_dot = z2
    z2_dot = -2*delta*z2 - omega**2*z1
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # 100s dt = 0.1
z0 = [3, 0] # initial state for [z1, z2]
zz = odeint(rhs, z0, tt) # call integration algorithm
```

- Notice: `rhs` is an "ordinary" object (type: `function`)
- Exercises:
 - a) Print the shape of the array `zz`
 - b) Plot the function $t \mapsto y(t)$
 - c) Simulate the [Van der Pol oscillator](#) and plot the phase portrait

Regression, Interpolation

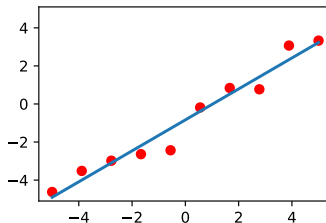
Regression (`scipy.polyfit`):



Regression, Interpolation

Regression (`scipy.polyfit`):

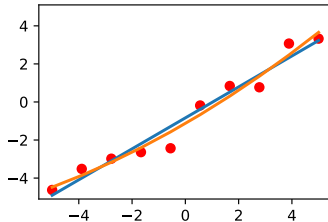
- Linear regression



Regression, Interpolation

Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order



Regression, Interpolation

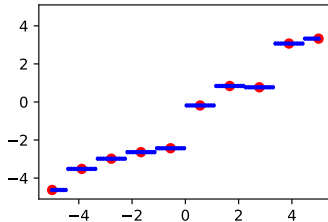
Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 0)



Regression, Interpolation

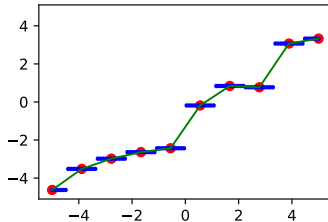
Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 1)



Regression, Interpolation

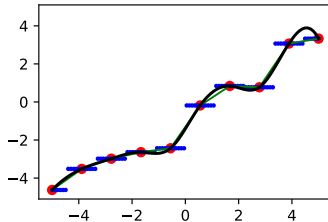
Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 2)



Regression, Interpolation

Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

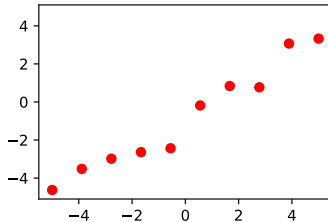
Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 2)

Hybrid (also `scipy.interpolation`):

- "smoothened spline" (smoothness coefficient)



Regression, Interpolation

Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

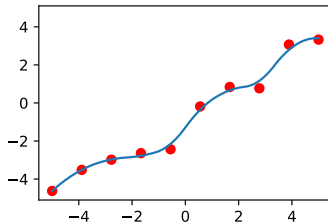
Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 2)

Hybrid (also `scipy.interpolation`):

- "smoothened spline" (smoothness coefficient)



Regression, Interpolation

Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

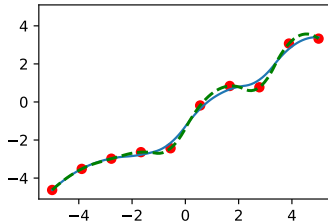
Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (→ Spline)
- Arbitrary order (here: order = 2)

Hybrid (also `scipy.interpolation`):

- "smoothened spline" (smoothness coefficient)



Regression, Interpolation

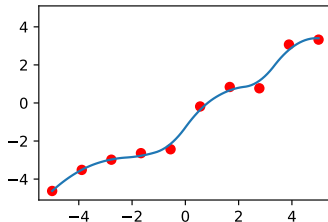
Regression (`scipy.polyfit`):

- Linear regression
- Also: higher order

Interpolation

(`scipy.interpolation`):

- Piecewise polynomial (\rightarrow Spline)
- Arbitrary order (here: order = 2)



Hybrid (also `scipy.interpolation`):

- "smoothened spline" (smoothness coefficient)

see `scipy-interpolation.py`

Outline

Introduction

Symbolic Computation (`sympy`)

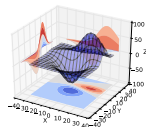
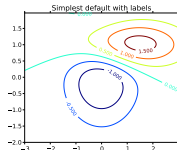
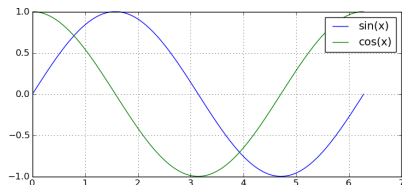
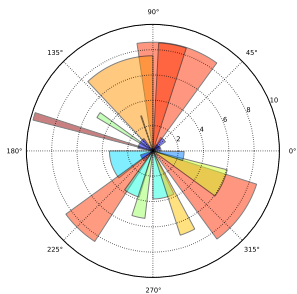
Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

matplotlib Overview (1)



Much more examples (including code): <https://matplotlib.org/gallery.html>



Overview (2)

- Defacto standard for 2d-plots with python
- (also some limited 3d-features)
- Supports many kinds of plots (see [gallery](#))
- Two types of results
 - Interactive (with zooming, panning)
 - Publication ready vector graphics
- Support for \LaTeX -formulas (axis-label, title, ...)
- Animations possible
- Drawback: Performance

Simple Plot

Listing: matplotlib1.py

```
from matplotlib import pyplot as plt
import numpy as np

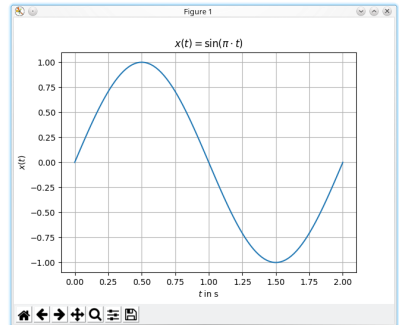
tt = np.linspace(0, 2, 100)
xx = np.sin(np.pi*tt)

plt.plot(tt, xx)

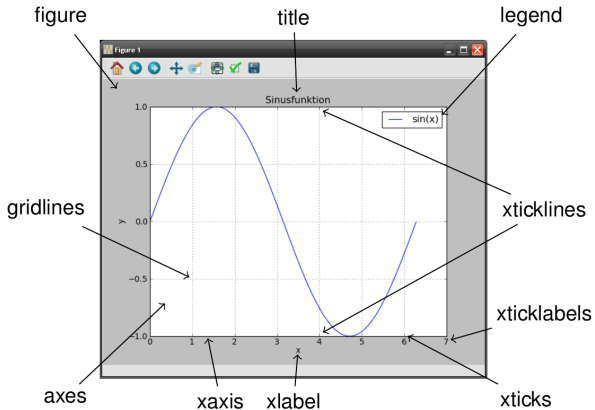
# add some candy
plt.grid()
plt.xlabel("$t$ in s")
plt.ylabel("$x(t)$")

plt.title(r"$x(t)=\sin(\pi \cdot t)$")
# note the usage of a raw-string
# (beginning with r")
# -> no special meaning of
# backslash-character (\)

plt.show()
```



Terminology



Cheatsheet: Line Styles

Three possibilities:

1. very short as format string:

```
plot(x, y, '--')
```

2. short with abbreviated keyword arguments :

```
plot(x, y, ls='--')
```

3. long with full keyword arguments:

```
plot(x, y, linestyle='dashed')
```

short form	long form	Output
' 'oder ' '	None	
'_'	'solid' (<i>default</i>)	_____
'--'	'dashed'	- - - - -
'_.'	'dashdot'	- . - . - .
':'	'dotted'

Cheatsheet: Line Styles

Three possibilities:

1. very short as format string:

```
plot(x, y, '--')
```

2. short with abbreviated keyword arguments :

```
plot(x, y, ls='--')
```

3. long with full keyword arguments:

```
plot(x, y, linestyle='dashed')
```

short form	long form	Output
' 'oder ' '	None	
'_'	'solid' (<i>default</i>)	_____
'--'	'dashed'	- - - - -
'_.'	'dashdot'	- . - . - .
':'	'dotted'

Exercise: Draw three fancy curves with different linestyles (and widths and colors).

Outline

Introduction

Symbolic Computation (`sympy`)

Interactive Documents with Jupyter Notebook

Numeric Computation (`numpy` and `scipy`)

2d Visualization (`matplotlib`)

Final remarks

Important but not covered topics

- Exception handling and usage of `assert`
- `lambda`-functions and list comprehension
- Scoping, Decorators (Functions of Functions)
- Test driven development (Link) / unittests
 - Software that tests software
 - No new feature without a test
 - Python: `import unittest`
- Documentation
 - Without docs your software is unlikely to be used (even by you).
 - Minimum: `"""Write docstrings!"""`
 - Better: use [Sphinx](#)
- Be aware of [PEP8](#) (style guide / best practices)
- Use a separate `virtualenv` for each project
- Revision control: Use [git](#)!

Links

Official docs

- <http://docs.sympy.org/latest/modules/>
- <https://docs.scipy.org/doc/numpy-1.13.0/reference/>
- <https://docs.scipy.org/doc/scipy/reference/>
- <https://matplotlib.org/contents.html>

Use also: google, [stackoverflow](#), ...