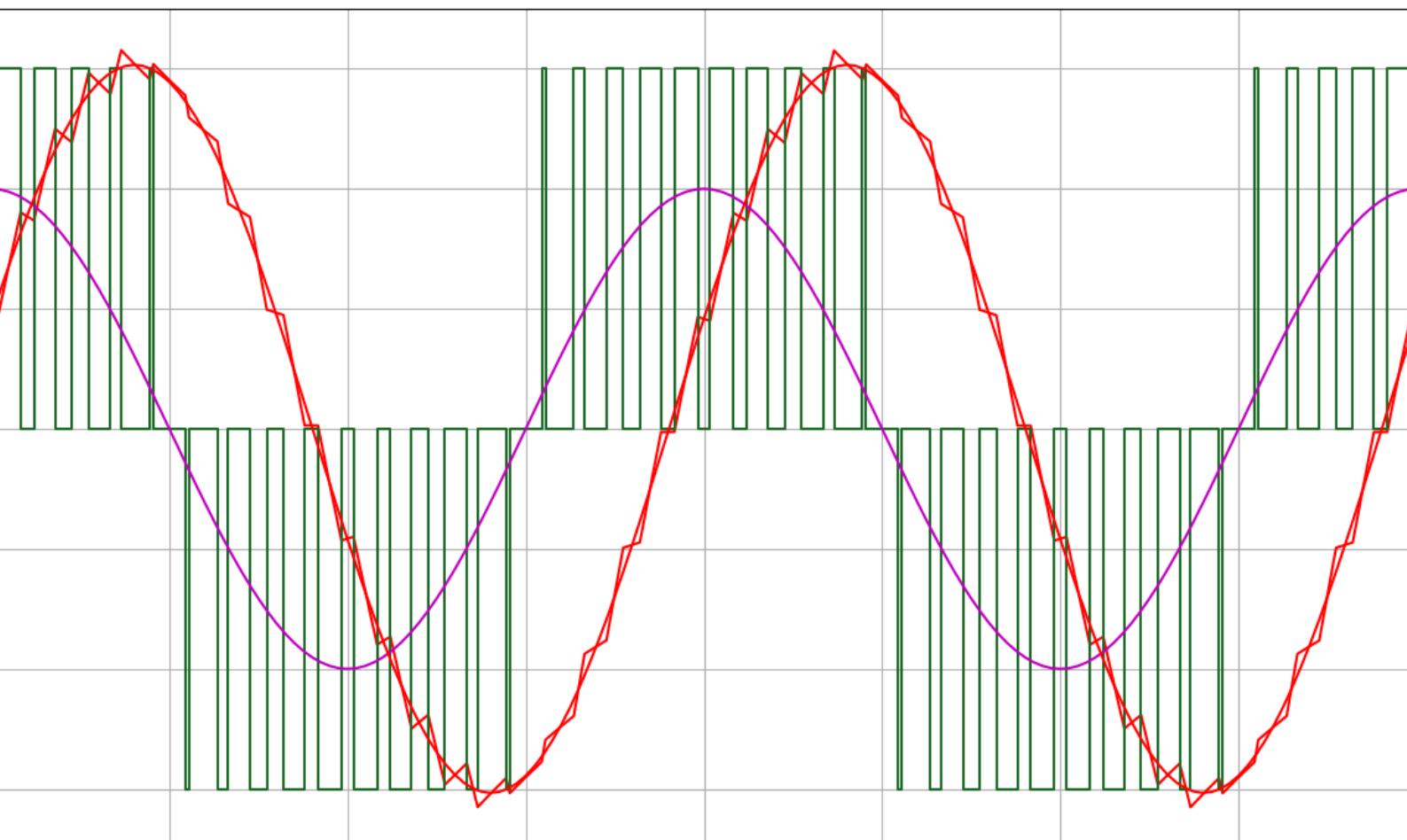


OpenModelica • Python 3.7 • Numpy • Scipy • Matplotlib • OMPython



ordinary differential eq. • differential algebraic eq. • zero crossing detection

## Modellbildung und Simulation

Prof. Dr.-Ing. Th. Köller

# Modellbildung und Simulation

Gliederung der Veranstaltung (Vorlesung/Workshop):

1. Einleitung, Grundlagen Python
2. Lösen von Differentialgleichungen am Beispiel eines Pendels
3. Simulationstechniken (`ode`, `dae`) am Beispiel einer RLC-Schaltung
4. Simulation eines Vierquadrantenstellers (Frequenzumrichter)
5. Numerische Lösungsverfahren (Euler, Heun, Runge-Kutta)
6. Modelica – Eigene Modelle
7. Parameteridentifikation

Simulieren lernt man durch simulieren und analysieren!

## Organisatorisches

- Skriptum - Tipp: Aktivieren Sie das Inhaltsverzeichnis/Bookmarks bei Ihrem pdf-Reader
- Die Vorlesung geht 4-stündig über ein halbes Semester
- Am Ende jeder Veranstaltung werden die Lösungen der Aufgaben vorgestellt. Die Musterlösungen werden verteilt
- Die Musterlösungen der „selbständigen Weiterarbeit“ werden nicht verteilt.
- Ergänzende Literatur:  
Python: <http://scipy-lectures.org/>  
Modelica: <http://book.xogeny.com/>

# Modellbildung und Simulation

## Kapitel 1: Grundlagen

### Lernziele

Am Ende des Kapitels ...

- ✓ ... wissen Sie, was man unter Modellbildung und Simulation versteht
- ✓ ... wissen Sie, wozu Simulation in der Praxis verwendet wird
- ✓ ... kennen Sie den Unterschied zwischen Modellbildung und Modellbau 
- ✓ ... können Sie grundlegende Berechnungen in Python durchführen
- ✓ ... kennen Sie zweidimensionale Visualisierungsmöglichkeiten

# Aufgaben der Modellbildung & Simulation

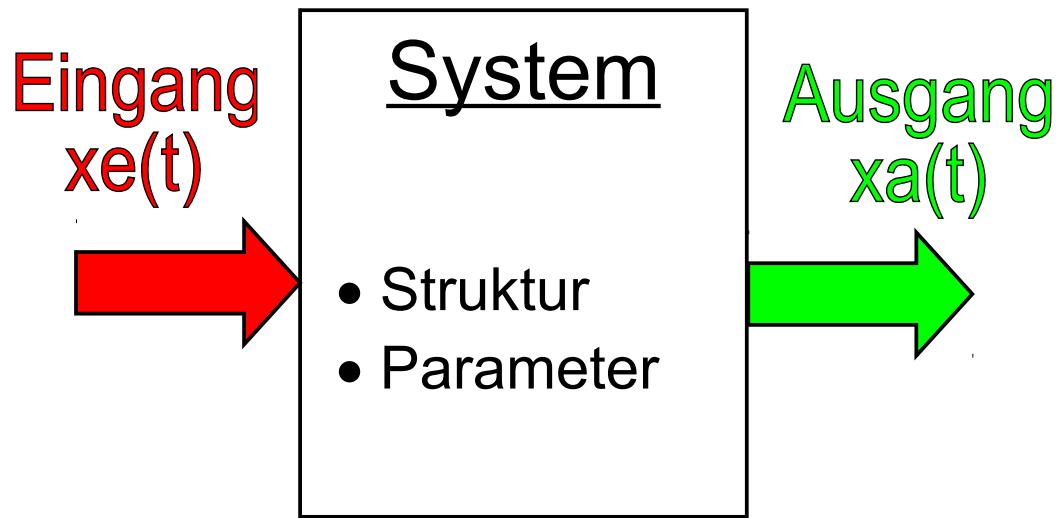
- ✓ Reale Systeme besser verstehen
- ✓ Schnelle Untersuchung unterschiedlicher Varianten
- ✓ Größen beobachten, die messtechnisch schwer oder gar nicht erfasst werden können
- ✓ Time to Market (TTM) Zeit verkürzen

## Lernziele

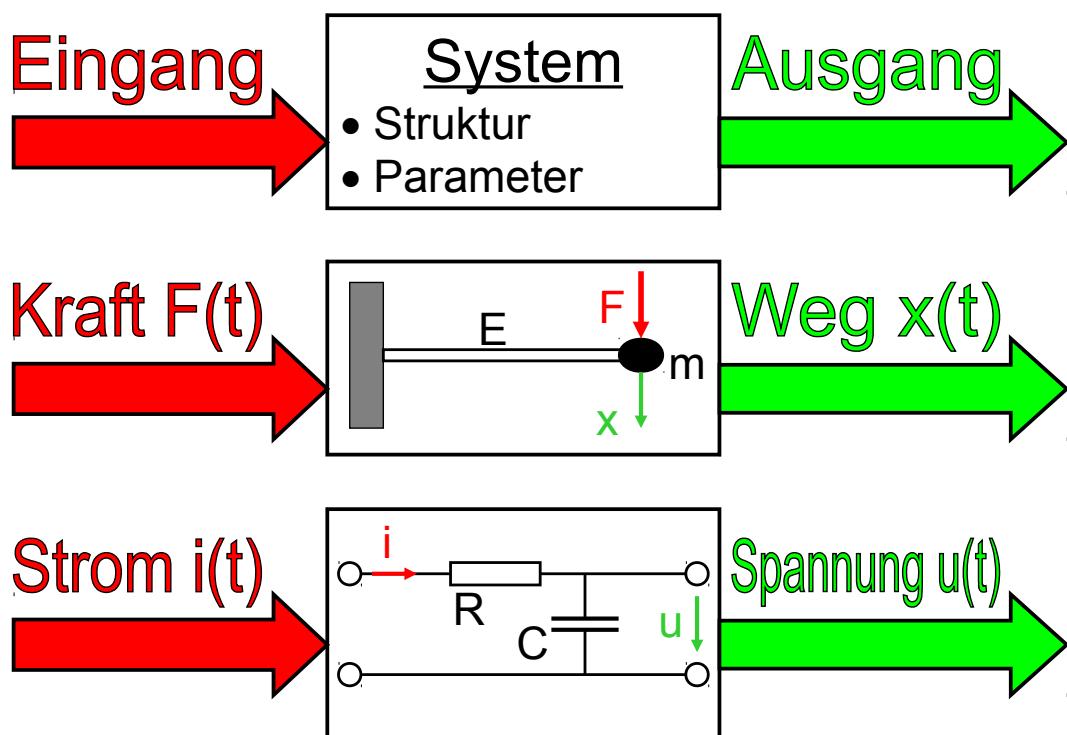
Am Ende des Kapitels ...

- ✓ ... haben Sie gelernt, wie Sie sich eigenständig Informationen verschaffen (online/offline)
- ✓ ... haben Sie anspruchsvolle Aufgaben zur Steuerung des `plot`-Befehls gelöst
- ✓ ... haben Sie erste Skripte in Python erstellt

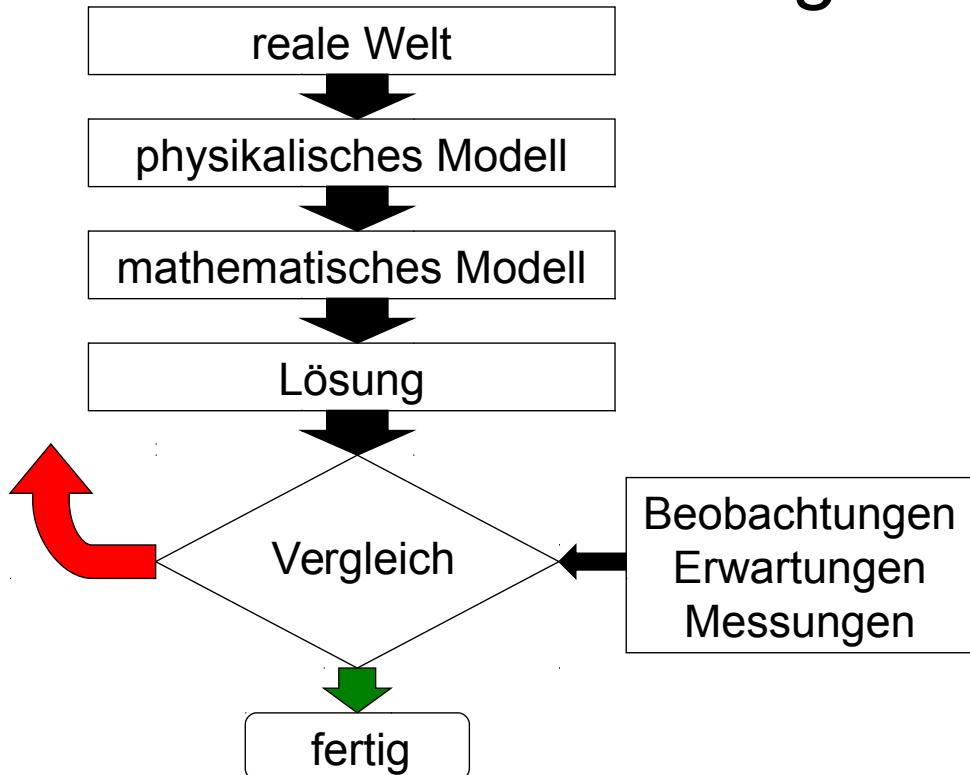
# dynamische Systeme:



## Beispiele:



# Ablauf der Modellbildung:



## Werkzeuge zur Simulation

- ✓ Matlab mit Simulink (Mathworks)
- ✓ Python mit Numpy, Scipy, Matplotlib 😊
- ✓ Scilab mit Xcos
- ✓ Octave
- ✓ Dymola
- ✓ Jmodelica
- ✓ OpenModelica 😊

---

# **Python Tutorial**

## ***Release 1.0***

**Th. Köller**

**09.12.2020**

---

## Inhaltsverzeichnis

---

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Python Tutorial</b>                  | <b>1</b> |
| 1.1      | Einführung in Python . . . . .          | 1        |
| 1.2      | Erste Schritte . . . . .                | 2        |
| 1.3      | Skripte und Packages . . . . .          | 3        |
| 1.4      | Hilfe . . . . .                         | 5        |
| 1.5      | Datenstrukturen . . . . .               | 6        |
| 1.6      | Speichern und Laden von Daten . . . . . | 14       |
| 1.7      | Programmablauf . . . . .                | 15       |
| 1.8      | Funktionen . . . . .                    | 16       |
| 1.9      | Plotten von Funktionen . . . . .        | 19       |
| 1.10     | Scipy-Signal . . . . .                  | 20       |

# KAPITEL 1

---

## Python Tutorial

---

### 1.1 Einführung in Python

Hauptintention der Vorlesung „Modellbildung und Simulation“ ist das Modellieren und Simulieren von physikalischen Systemen. Dazu braucht es Werkzeuge. Im Rahmen der Veranstaltung verwenden wir zwei Werkzeuge: OpenModelica und Python. Das vorliegende Kapitel dient dazu, Sie mit Python vertraut zu machen.

Python ist eine weit verbreitete Skriptsprache für die es zahlreiche Bücher und sehr viele Tutorials gibt. Bei diesem Überangebot an Informationen ist es am Anfang sehr schwer, das Wichtige vom (zunächst noch) Unwichtigem zu trennen. Deshalb finden Sie nachfolgend alle Python-Konstrukte, die für die Veranstaltung „Modellbildung und Simulation“ erforderlich sind. Es ist nur ein kleiner Auszug. Vermutlich haben Sie Interesse daran, sich weiterhin mit dieser wunderbaren Skriptsprache zu beschäftigen. Ein Internetseite möchte ich Ihnen besonders empfehlen: <https://scipy-lectures.org/index.html>.

Dank der „Creative Commons“-Lizenz konnte ich einige Abschnitte aus der angegebenen Quelle für diese kleine Einführung übernehmen.

Ein Dank gilt allen Autoren: <https://scipy-lectures.org/preface.html#authors>

## 1.2 Erste Schritte

Tippen Sie in die IPython Konsole, (das ist das Fenster rechts unten in Spyder):

```
>>> print("Hallo Welt")
Hallo Welt
```

So einfach diese Zeile aussieht, verbirgt sie doch eine Besonderheit. In Python gibt es zwei Entwicklungsstränge: Python 2 und Python 3. In Python 2 würde man die Klammern im `print`-Befehl weglassen. Python 2 ist inzwischen abgekündigt und sollte folglich nicht mehr für Neuentwicklungen verwendet werden.

Probieren Sie ein paar Befehle in der Konsole aus:

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<type 'int'>
>>> print(b)
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<type 'str'>
>>> b + b
'hellohello'
```

Wie sie sehen, gibt es in Python eine implizite Variablen-deklaration. In C muß eine Variable zunächst deklariert werden, bevor man sie verwendet. Das ist in Python nicht so. Das kann jedoch unangenehme Effekte haben, wie das folgende Beispiel zeigt:

### Warnung: Integer Division

In Python 2:

```
>>> a=3
>>> b=2
>>> a / b
1
```

In Python 3:

```
>>> a / b
1.5
```

### Schreiben Sie immer Dezimalpunkte bei floats:

```
>>> a=3.0
>>> b=2.0
```

## 1.3 Skripte und Packages

Tippt man alle Befehlszeilen nur in den IPython-Interpreter, so sind sie beim Schließen des Programmes verloren. Die Lösung sind sogenannte Skripte. Der Quelltext wird mit Hilfe eines Editors (linkes Fenster bei Spyder) eingegeben und dann in einer Textdatei mit der Endung .py gespeichert. Durch die Taste F5 kann man dieses Skript dann starten.

### 1.3.1 Skripte - (Spyder Grundeinstellungen)

Erstellen Sie ein kleines Skript mit dem Namen test.py und folgendem Inhalt:

```
a=2
b=3
print('a = '+str(a))
```

Führen Sie das Skript aus und schauen Sie sich die Variablen im Variablenmanager (rechtes oberes Fenster, erster Reiter) an. Erwartungsgemäß sind beide Variablen vorhanden.

Löschen Sie jetzt die obere Zeile mit a=2 und führen Sie das Skript erneut aus. Sie werden feststellen, dass die Variable a nach wie vor vorhanden ist. Dies ist kein gutes Verhalten, schließlich wollen Sie beim Start eines Skriptes immer die gleichen Ausgangsbedingungen haben. Spyder kann man so einstellen, dass alle Variablen zu Beginn der Ausführung eines Skriptes gelöscht werden. Dies geschieht unter: Werkzeuge>Voreinstellungen>Ausführen. Dort den Haken bei Alle Variablen vor der Ausführung löschen setzen.

Unter Werkzeuge>Voreinstellungen>IPython-Konsole sollten Sie noch eine weitere Einstellung machen. Gehen Sie auf den Reiter Grafik und setzen Sie das Grafik-Backend auf Automatisch. So landen alle noch zu erstellenden Diagramme in einem zusätzlichem Fenster und nicht in der IPython-Konsole.

### 1.3.2 Packages

Der große Vorteil von Python ist die leichte Erweiterbarkeit. Der ursprüngliche Sprachumfang von Python ist nicht sonderlich groß. Überall auf der Welt werden jedoch Erweiterungen für die unterschiedlichsten Anwendungen entwickelt. Diese zusätzlichen Features werden in sogenannten packages bereitgestellt. Mit dem import Befehl lassen sich diese Packages importieren. Das package os gewährt beispielsweise Zugriff auf Betriebssystemfunktionen, wie das folgende Beispiel zeigt:

```
In [1]: import os
In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>
In [3]: os.listdir('.')
Out[3]:
['conf.py',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
'basic_types.rst',
'control_flow.rst',
'functions.rst',
'python_language.rst',
'reusing.rst',
'file_io.rst',
'exceptions.rst',
'workflow.rst',
'index.rst']
```

Oder auch:

```
In [4]: from os import listdir
```

Import des Numpy-Packages, so dass es mit dem kurzen Namen np genutzt werden kann:

```
In [5]: import numpy as np
```

### Warnung:

```
from os import *
```

Das nennt man *star import*. **Vermeiden Sie diese Art, packages zu importieren**

- Der Sourcecode ist schwer lesbar, da man nicht mehr weiß, woher die Funktionen kommen.
- Die Tab-Completion (siehe unten) funktioniert nicht mehr.
- Viele Variablennamen sind nicht mehr verwendbar. Z.B. *name*, da dieser Name schon von *os.name* benutzt wird.
- Möchten Sie unterschiedliche Packages oder Module verwenden gibt es u.U. Namenskonflikte.

In der Vorlesung werden wir bei fast allen Beispielen die packages numpy, scipy und matplotlib gebrauchen. Jedes Python-Skript, das Sie erstellen, sollte also mit den folgenden Befehlszeilen beginnen:

```
import numpy as np # data arrays
import scipy # scientific computing
import matplotlib.pyplot as plt # Visualisierung
from numpy import pi as pi # für die Zahl pi
```

## 1.4 Hilfe

Python ist außerordentlich umfangreich und leistungsfähig. Sicher kann man sich nur einen kleinen Teil des Sprachumfangs merken. Man ist folglich darauf angewiesen, möglichst schnell Unterstützung und Hilfe zu finden.

### 1.4.1 Hilfe Offline

Wenn Sie im Editor oder der Konsole eine Klammer eingeben, so erscheint automatisch im oberen rechten Fenster die zugehörige Hilfe. Probieren Sie es aus:

```
In [1]: import matplotlib.pyplot as plt
In [2]: plt.plot(
```

Sie können die Hilfe auch durch STRG-I aufrufen:

```
In [3]: plt.plot<STRG-I>
```

Verwenden Sie Funktionen aus zusätzlichen Packages (wie hier `matplotlib`) muß das zugehörige Package natürlich erst geladen werden.

Möchten Sie wissen, welche Methoden es innerhalb des Packages gibt, so geben Sie ein:

```
In [4]: plt.<TAB>
```

Das nennt man Tab-Completion.

### 1.4.2 Hilfe Online

Die Offline-Hilfe ist gut geeignet, wenn man einen Befehl eigentlich schon kennt und lediglich die genaue Syntax noch einmal nachlesen möchte. Online gibt es natürlich wesentlich umfangreichere Informationen:

- <https://docs.python.org/3/>
- <http://ipython.org/ipython-doc/stable/index.html>
- <https://docs.scipy.org/doc/numpy/>
- <https://docs.scipy.org/doc/scipy/reference/>
- <https://matplotlib.org/contents.html>

Python lebt aber von Beispielen. Häufig findet man in den unterschiedlichsten Foren (Python-Forum, stackoverflow, ...) tolle Programmbeispiele, die man übernehmen kann. Ohne Suchmaschine kann man in Python kaum effizient programmieren.

In der Veranstaltung „Modellbildung und Simulation“ sind wir darauf jedoch kaum angewiesen, da wir nur einen kleinen Teil der Python-Möglichkeiten nutzen.

## 1.5 Datenstrukturen

### 1.5.1 Skalare

#### Integer

```
>>> 1 + 1
2
>>> a = 4
>>> type(a)
<type 'int'>
```

#### FLOATS

```
>>> c = 2.1
>>> type(c)
<type 'float'>
```

#### Complex

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> type(1. + 0j)
<type 'complex'>
```

#### Booleans

```
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<type 'bool'>
```

---

**Tipp:** Verwenden Sie die IPython Konsole als Taschenrechner

---

```
>>> 7. * 3.
21.0
>>> 2**10
1024
>>> 8 % 3 # modulo
2
```

## 1.5.2 Vektoren und Matrizen

### Listen

**Tipp:** Eine Liste ist eine geordnete Sammlung von Objekten.

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> type(colors)
<type 'list'>
```

Der Zugriff auf die Elemente erfolgt über eckige Klammern:

```
>>> colors[2]
'green'
```

Mit negativen Indizes kann man auf die letzten Elemente zugreifen:

```
>>> colors[-1]
'white'
>>> colors[-2]
'black'
```

**Warnung:** Die Inizierung beginnt bei 0 (wie in C), nicht bei 1 (wie bei Scilab oder Matlab)!

Sublisten erhält man durch Slicing:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[2:4]
['green', 'black']
```

**Warnung:** colors[start:stop] enthält die Elemente von start (eingeschlossen) bis stop (ausgeschlossen). colors[start:stop] hat also (stop - start) Elemente.

**Slicing Syntax:** colors[start:stop:step]

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[3:]
['black', 'white']
>>> colors[:3]
['red', 'blue', 'green']
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> colors[::2]
['red', 'green', 'white']
```

Listen können natürlich verändert und erweitert werden:

```
>>> colors[0] = 'yellow'
>>> colors
['yellow', 'blue', 'green', 'black', 'white']
>>> colors[2:4] = ['gray', 'purple']
>>> colors
['yellow', 'blue', 'gray', 'purple', 'white']
```

Die Elemente einer Liste müssen nicht den gleichen Typ haben:

```
>>> colors = [3, -200, 'hello']
>>> colors
[3, -200, 'hello']
>>> colors[1], colors[2]
(-200, 'hello')
>>> colors.append('pink')
>>> colors
[3, -200, 'hello', 'pink']
```

## Numpy - Arrays

Insbesondere bei der Verarbeitung von numerischen Daten eignet sich das `numpy.array` meist besser, als die Liste. Mit `numpy-arrays` kann man sowohl Vektoren (eindimensionales Array), als auch Matrizen (zweidimensionales Array) sehr gut realisieren. Zeilenvektoren kann man durch eindimensionale Arrays darstellen. Für Spaltenvektoren braucht man aber das zweidimensionale Array. Grundsätzlich können Arrays unterschiedliche Typen beinhalten. Der Standardtyp ist der `float`-Typ. In „Modellbildung und Simulation“ werden wir keine anderen Typen verwenden. Arrays und Listen sind sehr ähnlich. Das Array-Objekt ist jedoch näher an der Hardware. Die Verarbeitung großer Datenmengen ist deshalb bei Arrays in der Regel schneller.

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

- 1-D - Arrays:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> a.shape
(4,)
>>> len(a)
4
```

- 2-D - Arrays:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])      # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)          # returns the size of the first dimension
2
>>> b = np.array([0, 1, 2, 3])
>>> b.reshape(4,1)  # Transponieren
array([[0],
       [1],
       [2],
       [3]])
```

Häufig braucht man ...

- Vektoren mit gleichen Abständen:

```
>>> a = np.arange(10) # 0 .. n-1 (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- oder Vektoren mit einer festgelegten Anzahl von Elementen:

```
>>> c = np.linspace(0, 1, 6)    # start, end, num-points
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

- ... oder auch Matrizen mit Nullen und Einsen etc.

```
>>> a = np.ones((3, 3)) # (3, 3) ist übrigens ein Tupel
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[ 1.,  1.,  1.])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Der Zugriff auf Einzelemente funktioniert fast genau so, wie bei Listen:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Bei zweidimensionalen Arrays braucht man natürlich auch zwei Indizes:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10 # dritte Reihe, zweite Spalte
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

### Bemerkung:

- In 2D entspricht der erste Index der Zeile und der zweite Index der Spalte
- Achtung: In der Mathematik beginnt die Indizierung bei 1, in Python aber bei 0.

**Slicing** bei Arrays funktioniert so:

```
>>> a = np.arange(10)
>>> a
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step], Das end-Element ist nicht enthalten
array([2, 5, 8])
```

Nicht alle Komponenten müssen angegeben werden. Die default-Elemente für *start*, *stop* und *end* sind: 0, *letztes Element*, und 1.

```
>>> a[1:3]
array([1, 2])
>>> a[::-2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

Der *append*-Befehl funktioniert auch bei Arrays, die Syntax ist aber etwas anders:

```
>>> a = np.arange(3,5)
>>> np.append(a, [5,4,3])
array([3, 4, 5, 4, 3])
>>> b=np.array([]) # leerer Array
>>> b=np.append(b, [1, 2, 3])
>>> b=np.append(b, [1, 2, 3])
>>> b
array([1., 2., 3., 1., 2., 3.])
```

Eine kleine Grafik veranschaulicht die Indizierung und das Slicing...

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  |
| 10 | 11 | 12 | 13 | 14 | 15 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 30 | 31 | 32 | 33 | 34 | 35 |
| 40 | 41 | 42 | 43 | 44 | 45 |
| 50 | 51 | 52 | 53 | 54 | 55 |

Auch bei Zuweisungen kann man das Slicing verwenden:

```
>>> a = np.arange(10)
>>> a[5:] = 10
>>> a
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> a
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

## Rechnungen mit Arrays

Sind alle Daten in einem Array angeordnet, so kann man mathematische Operationen sehr leicht auf alle Elemente anwenden. Dabei werden implizite Schleifen durchlaufen, die sehr viel schneller ablaufen, als wenn man selbst eine Schleife in Python programmieren würde.

Einfache Operationen mit Skalaren:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

Operationen elementweise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
```

**Warnung:** Array-Multiplikation ist keine Matrizenmultiplikation:

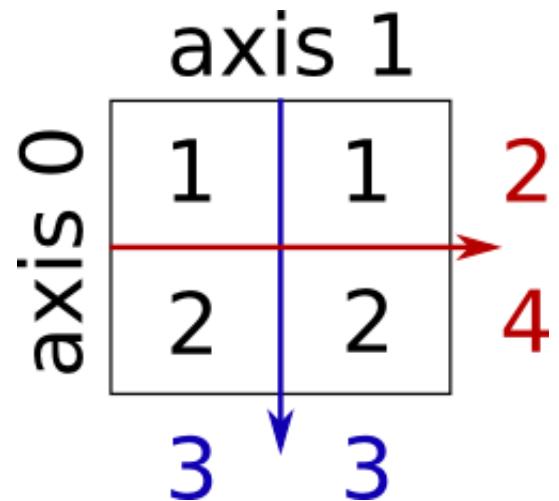
Die Matrizenmultiplikation funktioniert mit `dot`:

```
>>> A=np.array([[2,3,4],[1,5,6]])
>>> b=np.array([[1],[2],[3]])
>>> np.dot(A,b)
array([[20],
       [29]])
```

Häufig braucht man ...

Die Berechnung von Summen:

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```



Summen entlang der Zeilen oder Spalten:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)      # Summen der Spalten (nullte Dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)      # Summen der Zeilen (erste Dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```

Extremwerte:

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()    # Index des Minimums
0
>>> x.argmax()    # Index des Maximums
1
```

## Weitere Datenstrukturen

Python verfügt über weitere sehr leistungsfähige Datenstrukturen, wie: tuples – () und dictionaries – {}.

```
>>> tup_01 = ('music', 'engineering', 1997, 2000)
tup_01
('music', 'engineering', 1997, 2000)
>>> instrumente={'holz': ['klarinette', 'saxophon'], 'blech': [
    ↪'posaune', 'trompete']}
instrumente['blech'][1]
'trompete'
```

Diese Datenstrukturen sind für viele Programmieraufgaben sehr gut geeignet. Für numerische Probleme sind sie aber von untergeordneter Bedeutung, so dass wir im Rahmen der Veranstaltung „Modellbildung und Simulation“ auf die Verwendung dieser Datenstrukturen verzichten.

## 1.6 Speichern und Laden von Daten

Wenn man große Datenmengen erzeugt hat, muß man diese Daten in Dateien speichern können. Dies ist insbesondere dann wichtig, wenn das Erzeugen der Simulationsdaten sehr zeitintensiv war. *Numpy* liefert für diese Aufgabe die Befehle *np.savez* und *np.load*. Testen Sie:

```
In [1]: a=1; b=2
In [2]: np.savez('datafile.npz', a_var=a, b_var=b)
In [3]: data=np.load('datafile.npz')
In [4]: print(data.files)
['a_var', 'b_var']
In [5]: del a
In [6]: a=data['a_var']
In [7]: print(a)
```

Messdaten werden nicht selten in Textdateien abgelegt. So bietet z.B. jedes Oszilloskop die Möglichkeit Messwerte in einer Textdatei zu speichern. Solche Textdateien, deren Werte meist durch Komma getrennt sind, lassen sich sehr gut mit dem *loadtxt*-Befehl einlesen.

Beispielsweise lädt der Befehl:

```
In [8]: A = np.loadtxt('daten.txt', delimiter=',')
```

die Werte der Datei *daten.txt* in das Array *A* ein. Der *delimiter* gibt an, durch welches Zeichen die Daten voneinander getrennt sind.

## 1.7 Programmablauf

### 1.7.1 if/elif/else

```
>>> if 2**2 == 4:
...     print('Na klar!')
...
Na klar!
```

Abschnitte werden in Python durch das Einrücken des Quelltextes unterteilt. Dies nennt man *indentation*. In C ist das gleichbedeutend mit dem Setzen von geschweiften Klammern.

---

**Tipp:** Tippen Sie folgenden Quelltext in den Editor und testen Sie das Programm! Überprüfen Sie, ob das Programm bei einer anderen *indentation* noch funktioniert!

---

```
a = 10

if a == 1:
    print(1)
elif a == 2:
    print(2)
else:
    print('A lot')
```

### 1.7.2 for/range

Schleifen mit einem Index:

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```

Es geht aber auch so:

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
Python is cool
Python is powerful
Python is readable
```

## 1.8 Funktionen

### 1.8.1 Funktionen mit und ohne Rückgabewerte

Funktionen müssen nicht zwangsläufig Werte zurückgeben:

```
In [1]: def test():
...:     print('ein erster Test')
...:
...:

In [2]: test()
ein erster Test
```

---

**Bemerkung:** Auch bei Funktionen ist die *indentation* wichtig.

---

Möchte man Ergebnisse zurückgeben, verwendet man das Schlüsselwort `return`.

```
In [3]: def disk_area(radius):
...:     return 3.14 * radius * radius
...:

In [4]: disk_area(1.5)
Out[4]: 7.064999999999995
In [5]: 1e-3*np.round(disk_area(1.5)*1e3)
Out[5]: 7.065
```

---

**Bemerkung:** Was braucht man, um eine Funktion zu definieren?

- Das Schlüsselwort `def`,
  - danach der Funktionsname,
  - die Argumente der Funktion getrennt durch Kommata,
  - der Funktionsrumpf
  - und ggf. das `return`-Statement mit den Rückgabewerten.
-

## 1.8.2 Argumente der Funktion

Im einfachsten Fall unterscheidet die Funktion die Argumente anhand der Position ihres Auftretens. Fehlt ein Argument, so wird eine Fehlermeldung ausgegeben.

```
In [5]: def multiply_it(x, y):
....:     return x * y
....:

In [6]: multiply_it(3, 4)
Out[7]: 12

In [8]: multiply_it(3)
-----
TypeError: multiply_it() missing 1 required positional argument: 'y'
```

Bei der Argumentenliste hat man jedoch die Möglichkeit, *default*-Werte vorzugeben.

```
In [9]: def multiply_it(x, y=2):
....:     return x * y
....:

In [10]: multiply_it(3)
Out[11]: 6
```

Besonders interessant sind die Schlüsselwörter *args* und *kwargs*. Mit diesen Schlüsselwörtern kann man Funktionen schreiben, bei denen die Anzahl der Argumente während des Programmierens noch nicht feststeht. Diese Funktionalität werden wir in „Modellbildung und Simulation“ jedoch nicht nutzen.

- *\*args*: Hierbei handelt es sich um Argumente, die anhand der Position identifiziert werden.
- *\*\*kwargs*: Die *kwargs*-Argumente werden direkt durch ihren Variablenamen identifiziert.

```
In [14]: def variable_args(*args, **kwargs):
....:     print('args is', args)
....:     print('kwargs is', kwargs)
....:

In [15]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'y': 2, 'x': 1, 'z': 3}
```

### 1.8.3 Globale Variablen

Globale Variablen können innerhalb einer Funktion zwar gelesen, nicht jedoch beschrieben werden. So wird im nachfolgenden Beispiel die globale Variable *a* gelesen:

```
In [16]: a=2
In [17]: def test_global(x):
....:     return a * x

In [18]: b=test_global(5)
In [19]: print('a = ',a,'; b = ', b)
Out[20]: a =  2 ; b =  10
```

Versucht man innerhalb der Funktion auf die globale Variable zuzugreifen, wird stattdessen eine lokale Variable angelegt:

```
In [21]: a=2
In [22]: def test_global(x):
....:     a=3
....:     return a * x

In [23]: b=test_global(5)
In [24]: print('a = ',a,'; b = ', b)
Out[25]: a =  2 ; b =  15
```

Beschreiben kann man globale Variablen, wenn sie innerhalb der Funktion *global* deklariert werden:

```
In [26]: a=2
In [27]: def test_global(x):
....:     global a
....:     a=3
....:     return a * x

In [28]: b=test_global(5)
In [29]: print('a = ',a,'; b = ', b)
Out[30]: a =  3 ; b =  15
```

---

**Bemerkung:** Fast immer deutet die Verwendung von globalen Variablen auf einen schlechten Programmierstil hin. Globale Variablen sollten folglich nur selten in begründeten Ausnahmen benutzt werden.

---

## 1.9 Plotten von Funktionen

Die Visualisierung von Daten ist für uns besonders wichtig. Für diese Aufgabe verwenden wir das Package *Matplotlib*.

Auf <https://matplotlib.org/> finden Sie einen riesigen Fundus an Beispielen. Wir brauchen selbstverständlich nur einen kleinen Teil dieses umfassenden Programmpaketes. Meist beschränken wir uns darauf, Funktionen mit einer Veränderlichen zu plotten.

Wir starten mit einer einfachen Sinusfunktion:

```
x=np.arange(0, 2*pi, 0.1)
y=np.sin(x)
fig=plt.figure(1, figsize=(10,6)); fig.clf()
ax=fig.add_subplot(111)
ax.grid()
ax.plot(x,y)
```

Vergessen Sie nicht, alle nötigen Packages zu importieren:

```
import numpy as np # data arrays
import scipy # scientific computing
import matplotlib.pyplot as plt # Visualisierung
from numpy import pi as pi # für die Zahl pi
```

Möchten wir zwei Diagramme in einem Fenster unterbringen, so geht das folgendermaßen:

```
x=np.arange(0, 2*pi, 0.1)
y=np.sin(x)
fig=plt.figure(1, figsize=(10,6)); fig.clf()
ax=fig.add_subplot(211)
ax.grid()
ax.plot(x,y)
ax=fig.add_subplot(212)
ax.grid()
ax.plot(x,2*y, color='darkgreen', linestyle='--', linewidth=5.0)
fig.tight_layout() # schafft den nötigen Platz bei mehreren
→Subplots
```

Das Plotten einer Funktion erfolgt immer nach dem gleichen Schema:

- Öffnen eines *Figures* mit `fig=plt.figure(Nummer des Fensters, ggf. Größenänderung)`
- Löschen aller Element mit `fig.clf()`. Das ist insbesondere wichtig, wenn wir unser Skript mehrfach ausführen.
- Erzeugen eines Achsenobjektes mit `ax=fig.add_subplot(x, y, z)`. Dabei ist *x* die Anzahl der Diagramme, die untereinander dargestellt werden sollen (Zeilen), *y* die Anzahl der Diagramme nebeneinander (Spalten) und *z* adressiert das jeweilige Diagramm.
- `ax.plot` plottet dann die Funktion.

- Alle anderen Angaben haben kosmetische Wirkung. So können wir die Farbe, die Liniengeschwindigkeit, die Linienart etc. beeinflussen.

Folgende Befehle werden am häufigsten für die Veränderung der Darstellung benötigt:

```
ax.set_xlim()          # für die Beschränkung des zu druckenden Bereiches
ax.set_title()         # Für den Titel des Diagramms
ax.set_xlabel()         # Für die Achsenbeschriftung der x-Achse
ax.set_xticks()        # Wenn uns die Standardticks nicht gefallen
ax.set_xticklabels()   # Wenn wir die Bezeichnungen an den Ticks verändern wollen
```

Eine schöne Einführung in das Plotten zweidimensionaler Funktionen findet man beispielsweise hier: [https://matplotlib.org/users/pyplot\\_tutorial.html](https://matplotlib.org/users/pyplot_tutorial.html)

An dieser Stelle soll auf zwei unterschiedliche Konventionen hingewiesen werden. Hat man lediglich einen Subplot, so erscheint der Aufruf `ax=plt.subplot(111)` sinnlos. Man kann dann auch direkt die Methode `plt.plot` verwenden. Aus Gründen der Einheitlichkeit werden wir aber immer zunächst ein Achsenobjekt, wie oben dargestellt, erstellen und alle Methoden dann auf dieses Achsenobjekt anwenden.

## 1.10 Scipy-Signal

In der Systemtheorie oder der Regelungstechnik werden häufig Übertragungsfunktionen aufgestellt und dann diskutiert. Dabei sind z.B. Sprungantworten, die Pole von Übertragungsfunktionen oder Bode-Diagramme von besonderer Bedeutung. Mit dem *Signal*-Modul im *Scipy*-Package kann man diese Aufgaben besonders einfach lösen.

So kann man beispielsweise eine Übertragungsfunktion einfach über das Zähler- und Nennerpolynom definieren:

```
from scipy import signal
sys=signal.TransferFunction([1], [1, 1]) # G=1/(s+1)
print(sys.poles)
```

und das Bode-Diagramm und die Sprungantwort plotten:

```
omega, mag, phase = signal.bode(sys)
t, y = signal.step(sys)

fig=plt.figure(1, figsize=(10,6)); fig.clf()
ax=fig.add_subplot(111)
ax.plot(t, y)

fig=plt.figure(2, figsize=(10,6)); fig.clf()
ax=fig.add_subplot(211)
ax.semilogx(omega,mag)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
ax=fig.add_subplot(212)
ax.semilogx(omega,phase)
```

## Grundlagen - Python

### Aufgabe 1:

Erzeugen Sie die Variable  $a=3$ , die Liste  $b=[2 3]$  und das Array  $c=[1, 3, 5 \dots 77]$ . Speichern Sie lediglich die Variablen  $b$  und  $c$  in einer Datei mit dem Namen „test.npy“ ab. Löschen Sie die Variablen und lesen Sie die Datei wieder ein. Vergewissern Sie sich, dass die Variable  $a$  jetzt nicht mehr vorhanden ist.

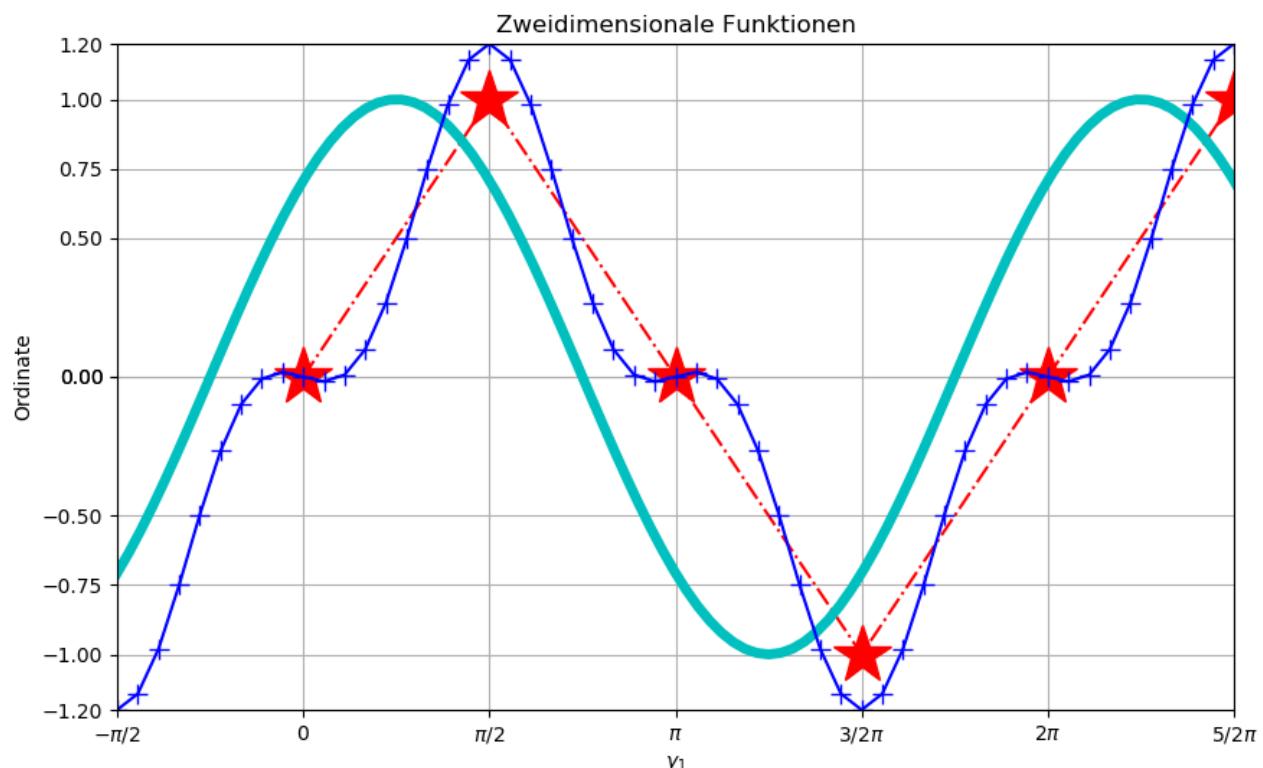
### Aufgabe 2:

Lesen Sie die Datei 'magic\_matrix.csv' mit dem Befehl 'np.loadtxt' ein. Was ist *magic* an dieser Matrix? Was ist eine csv-Datei und mit welchen anderen Programmen kann man die csv-Datei bearbeiten?

1. Berechnen Sie die Summe aller Elemente.
2. Berechnen Sie die Position des größten und des kleinsten Elements in der Matrix. Wie groß sind diese beiden Extremwerte?
3. Bilden Sie einen neuen Spaltenvektor  $v_1$  bestehend aus der dritten Spalte der Matrix  $A$ .
4. Bilden Sie die Summe aller Elemente, die größer sind als 40. (Tipp: Es ist nur eine kurze Befehlszeile erforderlich.)
5. Bilden Sie einen Zeilenvektor  $v_2$  bestehend aus den Elementen  $A_{3,4}$  bis  $A_{7,4}$ .

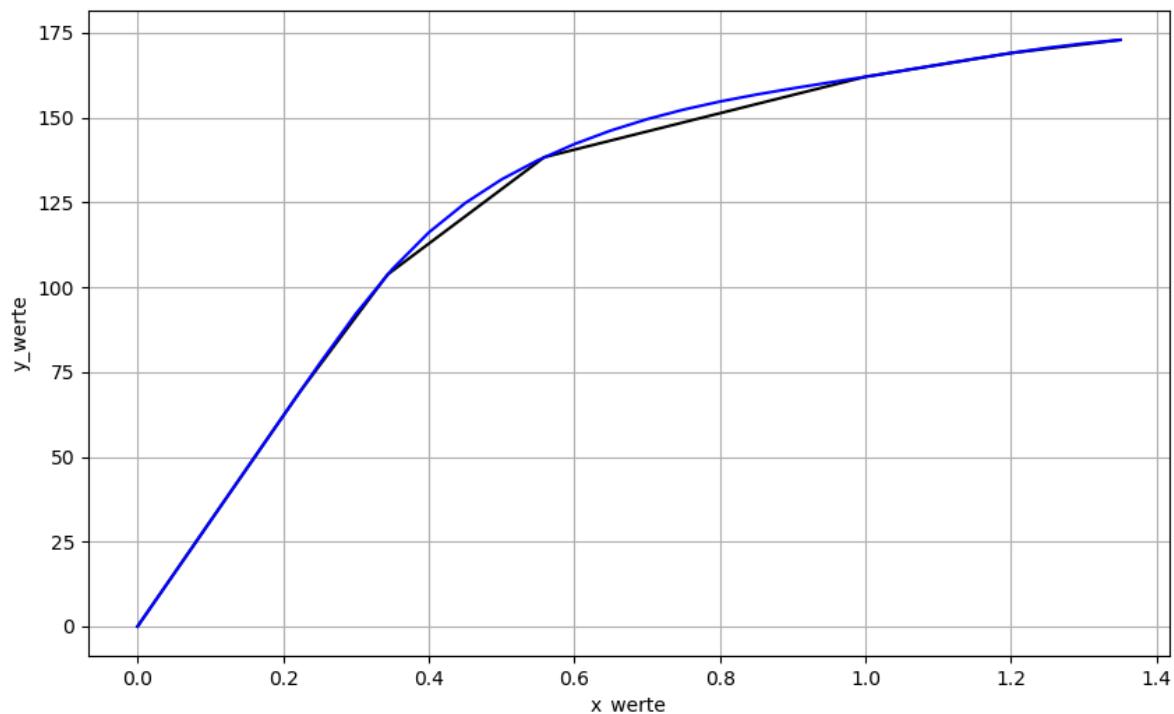
### Aufgabe 3:

Erstellen Sie ein Python-File, welches exakt das Diagramm aus der folgenden Abbildung erzeugt.



**Aufgabe 4:**

Von einem Kommilitonen erhalten Sie eine Kennlinie, welche mit Matlab generiert wurde. Von der Kennlinie sind lediglich einige Stützwerte bekannt (`mkl.mat`). Realisieren Sie eine Spline-Interpolation, um einen ‚weichen‘ Kurvenverlauf zeichnen zu können. (Tipp: `scipy.io.loadmat`, `scipy.interpolate.interp1d`). Erstellen Sie ein Diagramm ähnlich der folgenden Abbildung.



# Modellbildung und Simulation

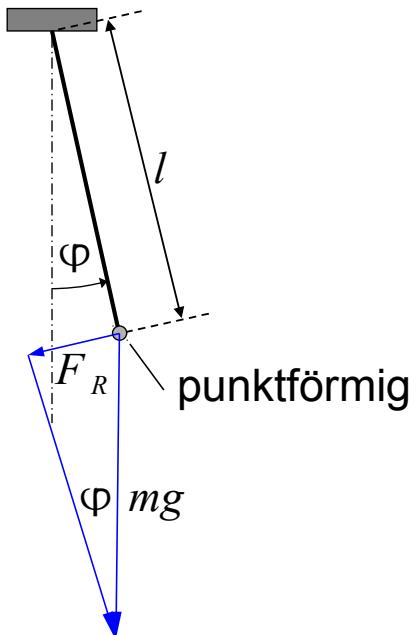
## Kapitel 2: Lösen von gewöhnlichen Differentialgleichungen (ode – ordinary differential equation) am Beispiel eines Pendels

### Lernziele

Am Ende des Kapitels ...

- ✓ ... sind Sie in der Lage, lineare und nichtlineare Differentialgleichungssysteme mit Python und OpenModelica zu lösen
- ✓ ... haben Sie die Vorteile einer Verknüpfung von Python und OpenModelica kennen gelernt.
- ✓ ... haben Sie Möglichkeiten zur Steuerung der numerischen Lösung, wie Anfangs- und Endzeit, kennengelernt
- ✓ ... kennen Sie die Unterschiede zwischen einem mathematischen und einem physikalischen Pendel

# Mathematisches Pendel



Rückstellmoment:

$$M_R = l m g \sin(\varphi)$$

Kleine Winkel:

$$M_R \approx l m g \varphi$$

Trägheitsmoment

(entgegen Koordinatenrichtung):

$$M_T = m l \ddot{x} = m l^2 \ddot{\varphi}$$

$$\sum M = 0 \rightarrow l m g \varphi + m l^2 \ddot{\varphi} = 0$$

$$\rightarrow \ddot{\varphi} + \frac{g}{l} \varphi = 0$$

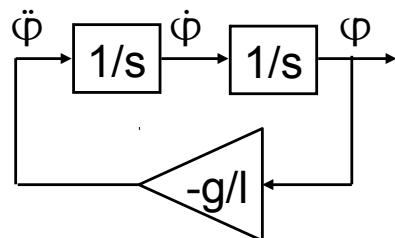
# Mathematisches Pendel

$$\ddot{\varphi} + \frac{g}{l} \varphi = 0 \quad \lambda^2 + \frac{g}{l} = 0 \rightarrow \lambda_{1,2} = \pm \sqrt{-\frac{g}{l}} = \pm j \sqrt{\frac{g}{l}}$$

$$\text{Eigenkreisfrequenz: } \omega_0 = \sqrt{\frac{g}{l}}$$

Die Eigenfrequenz ist unabhängig von der Masse!

Visualisierung der  
Differentialgleichung:



# Lösen der Differentialgleichung

## Drei Varianten

- Variante 1: Lösung mit der Python-Funktion solve\_ivp
- Variante 2: Kombination von Python und OpenModelica - Blockschaltbild
- Variante 3: Kombination von Python und OpenModelica – Modelica-Code

### Lösung durch die solve\_ivp-Funktion

$$\ddot{\varphi} + \frac{g}{l} \varphi = 0 \rightarrow \dot{x}_1 = x_2, \quad \dot{x}_2 = -\frac{g}{l} x_1$$

```
from scipy.integrate import solve_ivp
```

```
t_max=5;  
t=np.linspace(0, t_max, 101)
```

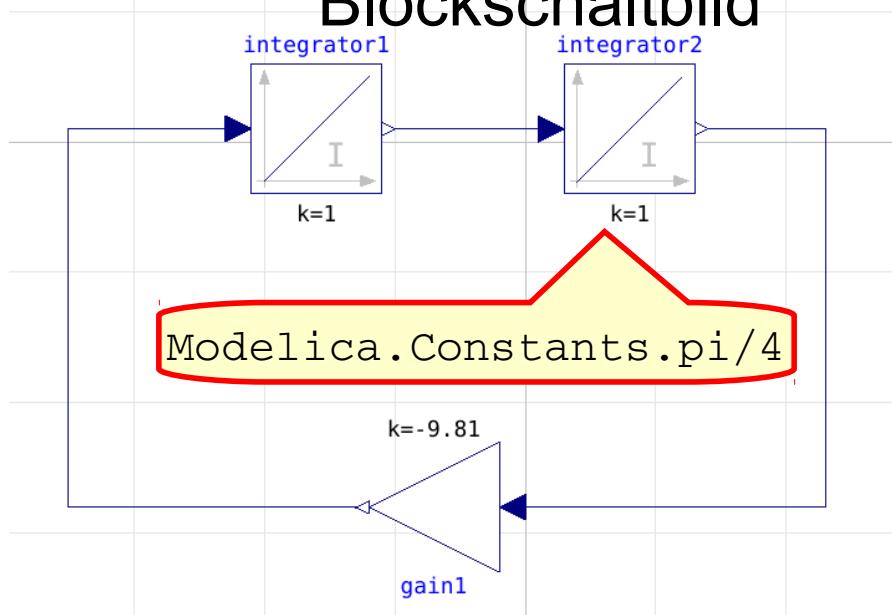
Definition der Zeitpunkte  
für die Lösung

```
def xdot_fkt(t, x, *args):  
    xdot= [x[1], -g/l*x[0]]  
    return xdot
```

Startwerte für x\_1; x\_2

```
x0=[phi_0, 0]  
sol = solve_ivp(xdot_fkt, [0, t_max], x0,  
t_eval=t, method='LSODA', args=(g, l))  
t=sol.t; x=sol.y;
```

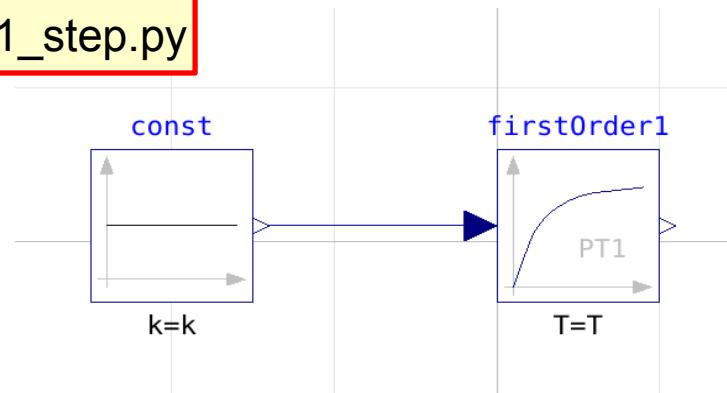
# Lösung durch ein Modelica - Blockschaltbild



```
[t]=mod.getSolutions('time')  
[phi]=mod.getSolutions('integrator2.y')
```

## Kombination Python-OpenModelica Einfaches Beispiel vorab

Siehe Datei: pt1\_step.py



pt1\_step.mo wird in der Vorlesung aufgebaut

```
mod.setParameters('firstOrder1.k'=2.0)
```

Die Parameter lassen sich von Python aus verändern.

# Lösung durch Modelica - Code

```
model pendel
  Real phi(start=pi/4);
  Real phi_dot;
  parameter Real l=1.0;
  constant Real g=9.81;
  constant Real pi=Modelica.Constants.pi;
equation
  der(phi)=phi_dot;
  der(phi_dot) + d*phi_dot + g*phi/l=0;
end pendel;
```

Datei: pendel.mo

## Start von OpenModelica durch Python

```
from OMPython import ModelicaSystem
from help_fkt import delete_OM_files

modelname='pendel'
mod=ModelicaSystem(modelname+'.mo',modelname)
mod.setSimulationOptions('stopTime=5.0')
mod.simulate()
[t]=mod.getSolutions('time')
[phi]=mod.getSolutions('phi')
delete_OM_files(modelname)
```

Siehe auch:

<https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/ompython.html>

# Kombination von Python und OpenModelica

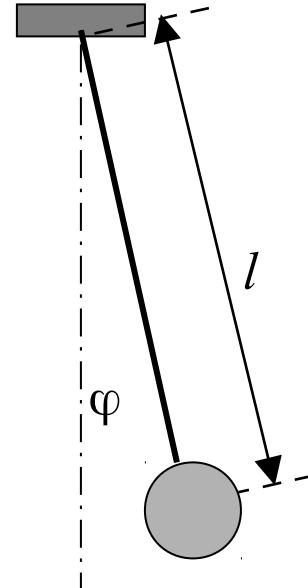
- Vorteil Python: Gute Visualisierung der Ergebnisse (matplotlib). Die Weiterverarbeitung der Simulationsergebnisse ist einfach möglich
- Vorteil OpenModelica: Die Differentialgleichungen werden visualisiert. Änderungen am Modell sind leicht realisierbar.
- Aber: Die häufig praktizierte Methode, Differentialgleichungen über Blockschaltbilder zu visualisieren ist nicht wirklich gut.

## Simulation der Schwingungen eines Pendels

Daten:  $g = 9,81 \text{ m/s}^2$   
 $l = 1 \text{ m}$

Anfangsbedingungen:  $\dot{\varphi}(t=0)=0$   
 $\varphi(t=0)=\pi/4$

Zeitintervall:  $0 \leq t \leq 5 \text{ s}$



**Aufgabe 1:** Simulieren Sie die vereinfachte Gleichung des mathematischen Pendels ( $\ddot{\varphi} + \frac{g}{l} \varphi = 0$ ) mit den drei dargestellten Varianten und visualisieren Sie  $\varphi(t)$ .

**Aufgabe 2:** Integrieren Sie einen Dämpfer, der die Schwingungen abklingen lässt.  
Differentialgleichung:  $\ddot{\varphi} + d\dot{\varphi} + \frac{g}{l} \varphi = 0$        $d = 0,3 \text{ 1/s}$   
Programmieren Sie wiederum alle drei Varianten wie in Aufgabe 1.

**Aufgabe 3:** Simulieren Sie das Verhalten des nichtlinearen, mathematischen, gedämpften Pendels mit der `solve_ivp`-Methode und vergleichen Sie das Ergebnis mit der linearisierten Gleichung. Stellen Sie dazu beide  $\varphi(t=0)$ -Verläufe in einem Diagramm dar.

**Aufgabe 4:** Stellen Sie die DGL für das nichtlineare, physikalische Pendel auf und erstellen Sie ein Modelica-Blockmodell. Bedenken Sie, dass der Luftwiderstand immer entgegen der Bewegungsrichtung wirkt.

(Gleichung für den Luftwiderstandskraft:  $F = c_w \frac{\rho}{2} v^2 A$  ;

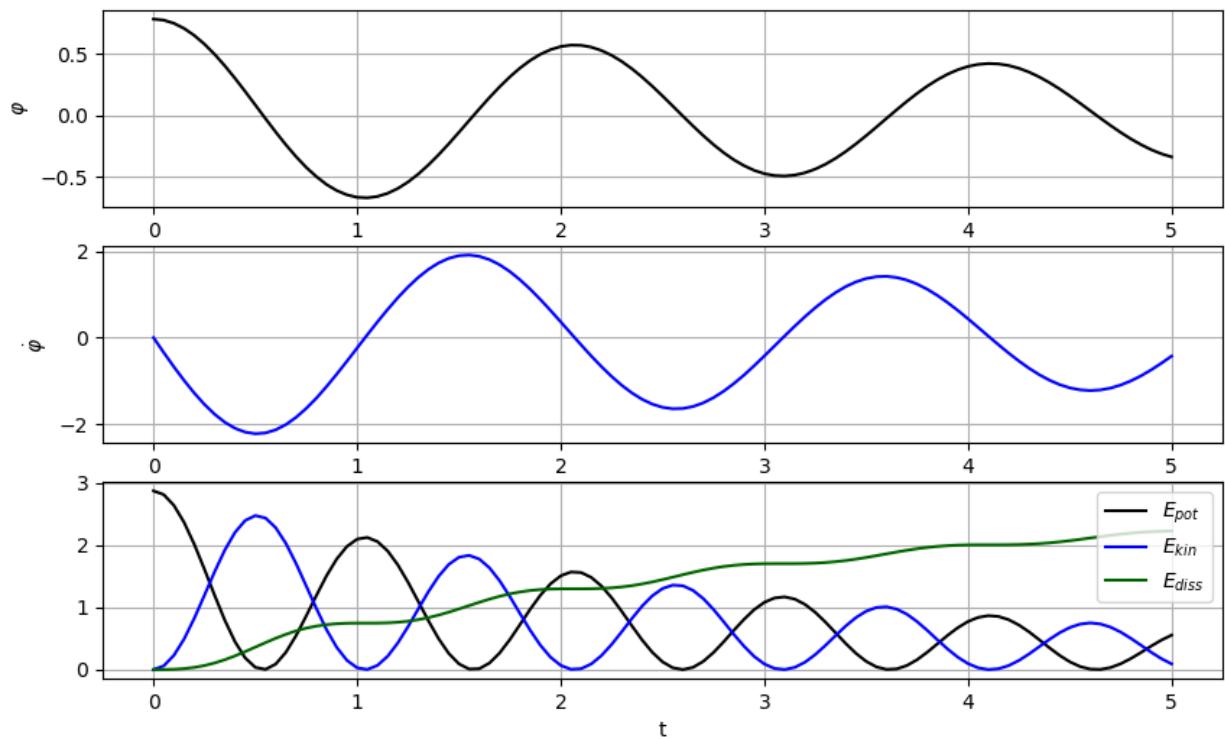
Kugel:  $c_w = 0,35$ ,  $\varnothing = 10 \text{ cm}$  ;  $\rho = 1.188 \text{ kg/m}^3$  (Luft) ;  $m = 1 \text{ kg}$  ;  
 $0 \leq t \leq 500 \text{ s}$  )

Der Dämpfungsparameter soll vom Python Skript an das Modelica-Modell übergeben werden.

Bei der Berechnung des Trägheitsmomentes kann ein Anteil vernachlässigt werden. Welcher?

----- Selbständige Weiterarbeit-----

**Aufgabe 5:** Analysieren Sie die energetischen Verhältnisse für das mathematische Pendel ( $d=0.3$ ,  $\varphi(t=0)=\pi/4$ ). Erstellen Sie ein Diagramm wie das nachstehende. Welches Problem tritt auf, wenn Sie mit der linearisierten Differentialgleichung rechnen?



# Modellbildung und Simulation

## Simulationstechniken

### Lernziele

Am Ende des Kapitels ...

- ✓ ... haben Sie die Zustandsraum-Darstellung noch besser verinnerlicht
- ✓ ... kennen Sie die „Schnittstelle“ zwischen einer physikalischen Aufgabenstellung und der mathematischen Lösungsmethode
- ✓ ... haben Sie gelernt, wie man ein Modell erstellt, die Zustandsgrößen identifiziert und das beschreibende Differentialgleichungssystem in die Zustandsform bringt

# Lernziele

Am Ende des Kapitels ...

- ✓ ... haben Sie gelernt, dass die Energiespeicher das Systemverhalten charakterisieren
- ✓ ... haben Sie gelernt, dass die Systemmatrix die Dynamik eines Systems beschreibt
- ✓ ... haben Sie gelernt, charakteristische Frequenzen in Signalen zu erkennen
- ✓ ... haben Sie gelernt, dass der umständliche Weg zur Aufstellung der Zustandsgleichung vermieden werden kann
- ✓ ... haben Sie noch mehr über Modelica gelernt

Simulation des Systemverhaltens:



Umformung der Systemgleichungen in Zustandsform, damit die Lösung mit der `solve_ivp`-Funktion erfolgen kann.

ODER:

Aufstellen der Potentialgleichungen und Flußgleichungen mit direkter Realisierung in Modelica.

ODER:

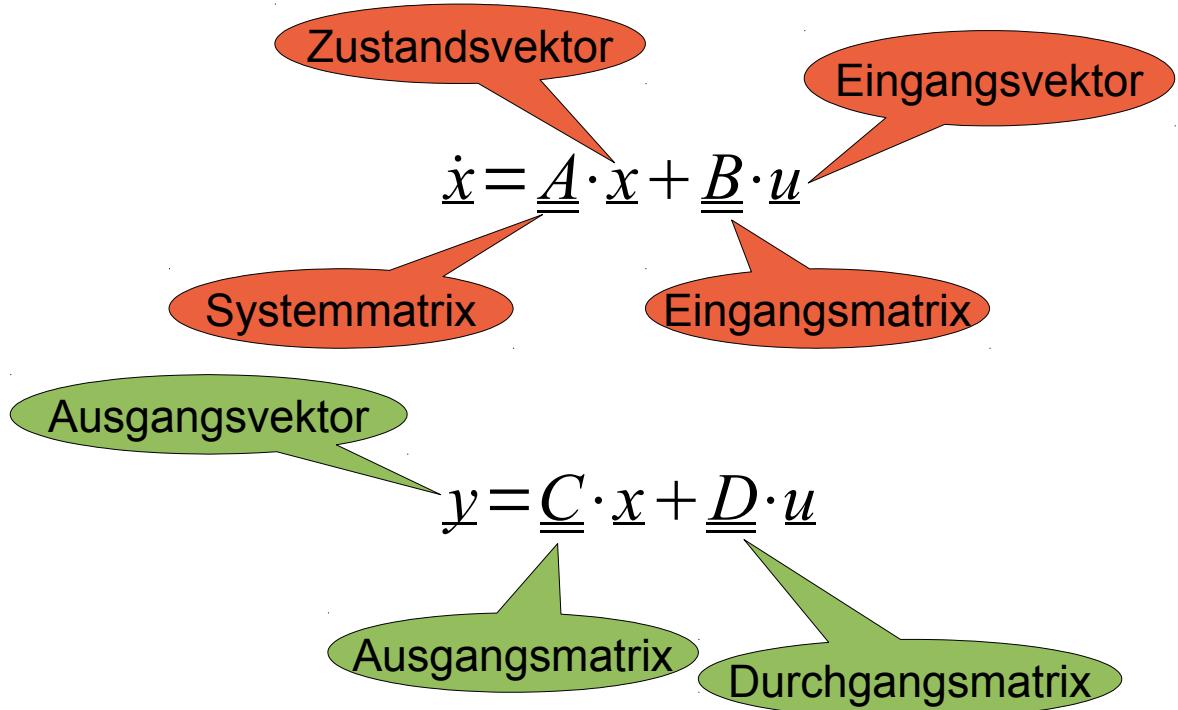
Indirekte Angabe der Gleichungen durch Verwendung vorab definierter Module.

# Zustandsvektor:

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

„Energiespeicher  
eines Systems“

# Zustandsraumdarstellung



## Dimensionen in der Zustandsraumdarstellung

$$\begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ n \end{array} = \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ n \end{array} + \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ m \end{array} \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ m \end{array}$$

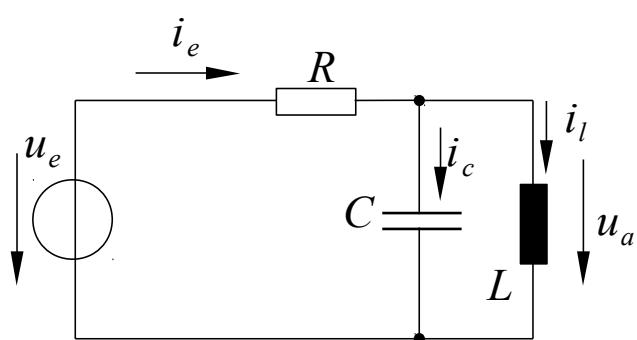
$$\dot{\underline{x}} = \underline{\underline{A}} \cdot \underline{x} + \underline{\underline{B}} \cdot \underline{u}$$

$$\begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ l \end{array} = \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ l \end{array} + \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ m \end{array} \begin{array}{c} \boxed{\phantom{0}} \\ \boxed{\phantom{0}} \\ m \end{array}$$

$$y = \underline{\underline{C}} \cdot \underline{x} + \underline{\underline{D}} \cdot \underline{u}$$

## Beispiel: RLC-Schaltkreis

Ersatzschaltbild:



$$u_e(t) = \hat{u} \sin(2\pi f t)$$

Maschengleichung:

$$-u_e + R i_e + u_a = 0$$

Knotenpunktgleichung:

$$i_e - C \frac{du_a}{dt} - i_l = 0$$

Induktivität:

$$L \frac{di_l}{dt} = u_a$$

# Zwei Zustandsgrößen („Energiespeicher“)

Kondensator: Spannung  $u_a$

Induktivität: Strom  $i_l$

Zustandsvektor:  $\underline{x} = \begin{bmatrix} u_a \\ i_l \end{bmatrix}$

Die Systemmatrix steht damit fest. Um die Matrizen B, C und D zu bestimmen, müssen die Eingangs- und Ausgangsgrößen definiert werden.

Eingangsgröße:  $u_e$

Ausgangsgrößen:  $u_a, i_l, i_e$

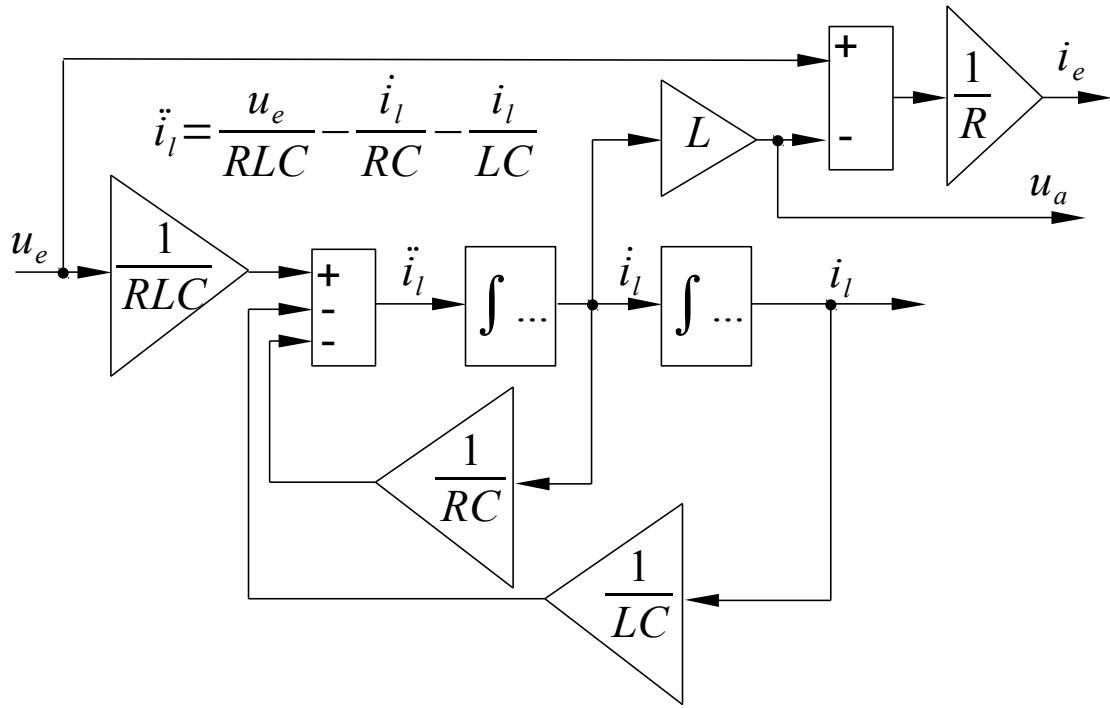
## Zustandsgleichungen der RLC-Schaltung

$$\begin{bmatrix} \dot{u}_a \\ \dot{i}_l \end{bmatrix} = \begin{bmatrix} -\frac{1}{RC} & -\frac{1}{C} \\ \frac{1}{L} & 0 \end{bmatrix} \cdot \begin{bmatrix} u_a \\ i_l \end{bmatrix} + \begin{bmatrix} \frac{1}{RC} \\ 0 \end{bmatrix} \cdot u_e$$

$$\begin{bmatrix} u_a \\ i_l \\ i_e \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -\frac{1}{R} & 0 \end{bmatrix} \cdot \begin{bmatrix} u_a \\ i_l \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{R} \end{bmatrix} \cdot u_e$$

In dieser Form kann das Problem leicht mit der `solve_ivp` - Funktion gelöst werden.

# Blockschaltbild



## Aufwand bei der Modellbildung

- Aufwändige Umformarbeiten müssen durchgeführt werden
- Beide Darstellungsformen sind wenig flexibel, wenn die Schaltungstopologie verändert werden soll
- Besser wäre es, wenn man die Maschen- und Knotengleichungen direkt angeben könnte
- Noch besser wäre es, wenn man die Gleichungen für einzelne Elemente vorgibt und diese dann miteinander verbündet
- Beide Ideen kann man mit Modelica verwirklichen

# DAE anstatt ODE

- Die Gleichungen können als echte Gleichungen und nicht als Zuweisungen vorgegeben werden. Die Umformarbeiten zur Erstellung der ode (ordinary differential equation) werden vom Modelica-Compiler vorgenommen. Der Benutzer wird damit nicht belastet.
- Der Benutzer braucht nur noch eine DAE (differential algebraic equation) vorgeben
- Das bekannteste und am weitesten entwickelte Programm, dass auf dieser Grundidee arbeitet, ist das kommerzielle Programm Dymola.
- OpenModelica ist eine leistungsfähige open-souce Alternative.

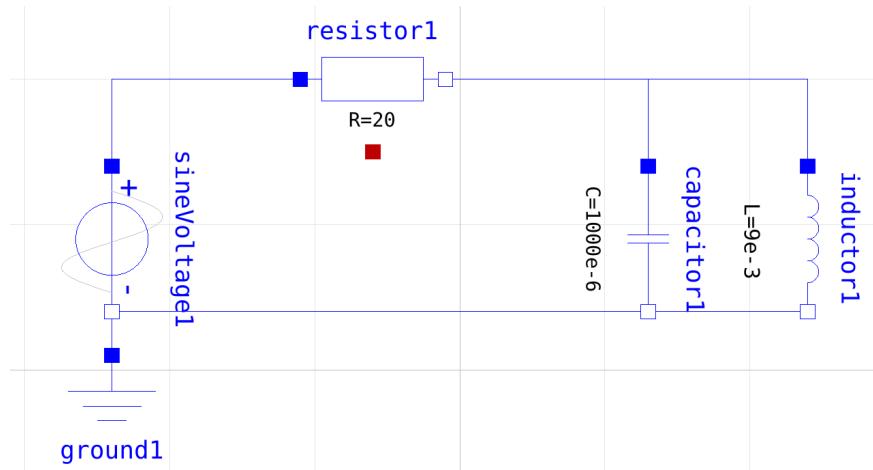
```
model A2
  parameter Real pi=Modelica.Constants.pi;
  Real u_e, i_L(start=0), i_e, u_a(start=0);
  parameter Real u_dach=2;
  parameter Real f=1;
  parameter Real R=20;
  parameter Real L=9e-3;
  parameter Real C=1000e-6;
equation
  u_e=u_dach*sin(2*pi*f*time);
  -u_e + R*i_e + u_a=0;
  i_e - C*der(u_a) - i_L=0;
  der(i_L)=u_a/L;
  annotation(
    uses (Modelica(version = "3.2.2")));
end A2;
```

## Simulation einer RLC- Schaltung

Das Einschwingverhalten einer RLC-Schaltung (Schwingkreis) an einer sinusförmigen Spannungsquelle soll untersucht werden.

Parameter der Schaltung:  $\hat{u}=2 \text{ V}$ ,  $f=1 \text{ Hz}$ ,  $R=20 \Omega$ ,  $L=9 \text{ mH}$ ,  $C=1000 \mu \text{F}$ .

- Aufgabe 1:** Analysieren Sie die Lösung über die Zustandsraumdarstellung (A1\_vorlage.py) mit Hilfe der Funktion `solve_ivp`. Ergänzen Sie die beiden Gleichungen der Zustandsdarstellung! Entfernen Sie `method='LSODA'`. Was passiert? Begründen Sie ausführlich, warum sich die Schaltung annähernd rein ohmsch verhält.
- Aufgabe 2:** Lösen Sie die Aufgabe mit der Verknüpfung von Python und Modelica-Code.
- Aufgabe 3:** Lösen Sie die Aufgabe mit Python/OpenModelica (modularer Ansatz). Setzen Sie die Parameter  $R$ ,  $L$  und  $C$  in Ihrem Python-Skript.



- Aufgabe 4:** Welche charakteristische Frequenzen können Sie in der Ausgangsspannung erkennen? Wie nennt man diese Frequenzen?

----- Selbständige Weiterarbeit -----

- Aufgabe 5:** Stellen Sie händisch eine Übertragungsfunktion  $G = \frac{U_a(s)}{U_e(s)}$  auf und berechnen Sie die Pole von  $G$  sowie die Eigenwerte der Systemmatrix. Geben Sie die Übertragungsfunktion in Python ein, berechnen Sie die gedämpfte Eigenfrequenz (vergleichen Sie mit Aufgabenteil 4) und zeichnen Sie das Bode-Diagramm ( $100 \text{ s}^{-1} < \omega < 1000 \text{ s}^{-1}$ )!
- Tipp: `signal.TransferFunction`, `signal.bode`, `.poles`, `.imag`

# Modellbildung und Simulation

## Vierquadrantensteller

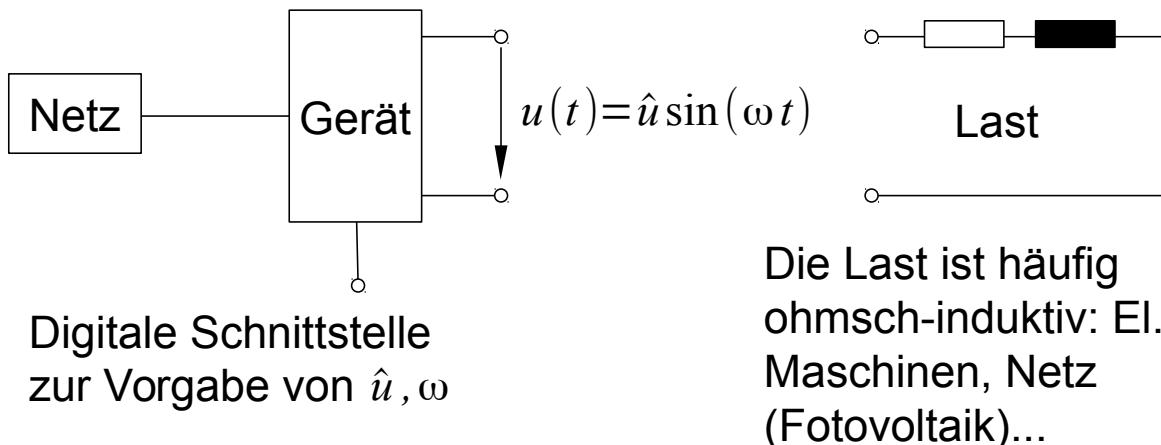
### Lernziele

Am Ende des Kapitels ...

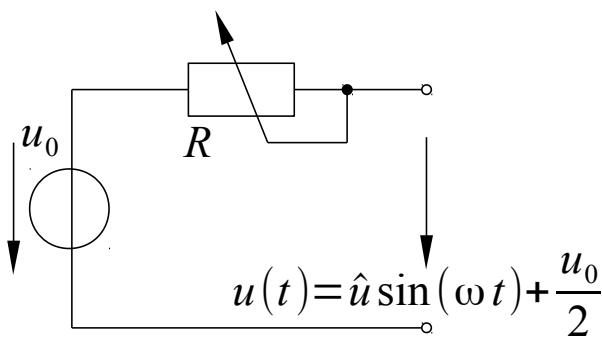
- ✓ ... wissen Sie, wie man in der Energie- und Automatisierungstechnik eine Spannung mit einstellbarer Amplitude und Frequenz erzeugt.
- ✓ ... haben Sie einen „Vierquadrantensteller“ simuliert.
- ✓ ... kennen Sie die Problematik in der Simulation, Unstetigkeitsstellen zeitlich exakt zu treffen.
- ✓ ... wissen Sie, was ein Stromrippel ist.
- ✓ ... haben Sie die Funktionsweise eines Frequenzumrichters verstanden

# Anforderung

In der Energie- und Automatisierungstechnik benötigt man häufig ein Gerät, mit dem man eine sinusförmige Spannung mit gewünschter Frequenz und Amplitude erzeugen kann.



## Naiver Ansatz



$R$  wird so verändert, dass  $u(t)$  sinusförmig wird. Für  $R$  wird üblicherweise ein Linearspannungsregler verwendet.

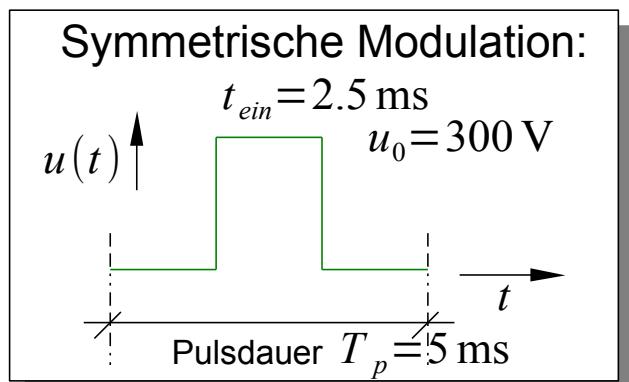
Annahme: Eine Gleichspannungsquelle ist vorhanden. (Üblicherweise gewinnt man  $u_0$  durch Gleichrichtung der Netzzspannung.)

Wegen des schlechten Wirkungsgrades verbietet sich diese Methode für große Leistungen!

# Verbesserter Ansatz

Man stellt die gewünschte Spannung nur mittelwertmässig ein. Das Tiefpassverhalten der Last sorgt dann für einen (annähernd) kontinuierlichen Stromverlauf.

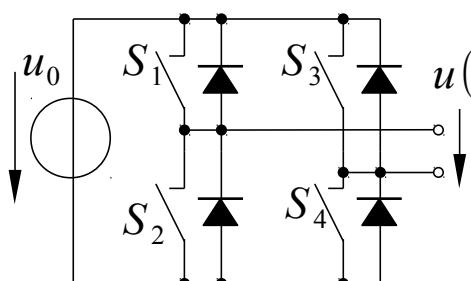
Beispiel: Für  $u_0 = 300 \text{ V}$  soll eine Ausgangsspannung von  $150 \text{ V}$  realisiert werden.



Die Spannung  $u_0$  wird für  $2.5 \text{ ms}$  an die Last gelegt. Für den Rest der Zeit wird die Last kurzgeschlossen.

Auch negative Spannungen sollen realisiert werden.  
Lösung: Umpolen der Spannungsquelle.

## Schaltung: Vierquadrantensteller



Durch geschicktes Schalten lässt sich (mittelwertmässig) eine sinusförmige Spannung realisieren.

Es gibt drei Zustände:

$$u(t) = u_0 : S_1 \text{ und } S_4$$

$$u(t) = -u_0 : S_2 \text{ und } S_3$$

$$u(t) = 0 : S_2 \text{ und } S_4 \text{ oder } S_1 \text{ und } S_3$$

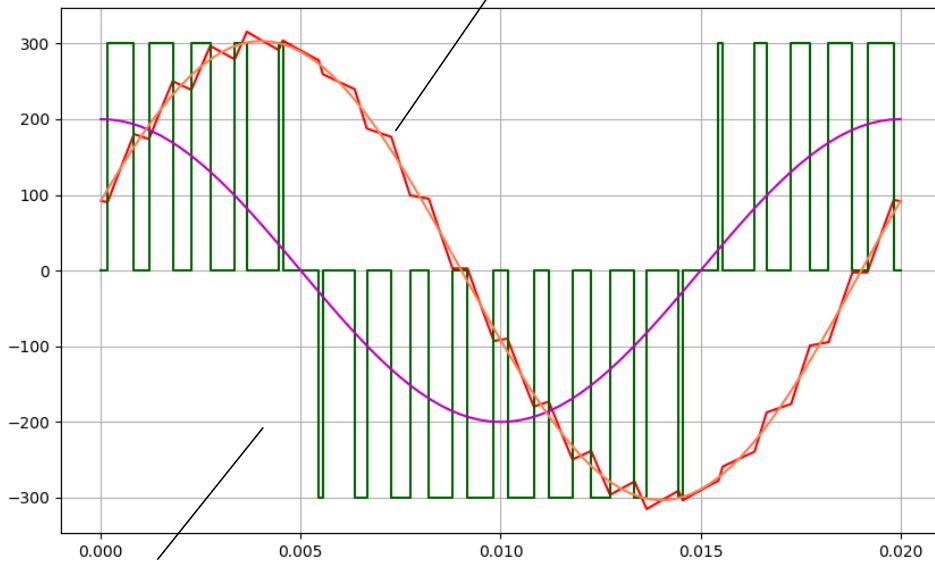
Die Schalter werden durch Transistoren realisiert.

Frage: Wozu braucht man die Dioden?

# Lösung

$T_p = 1 \text{ ms}$

Strom: Sollwert (orange) und Istwert (rot)



Pulsmuster der Spannung  $u(t)$  (grün) bilden  
mittelwertmässig eine Cosinuskurve.

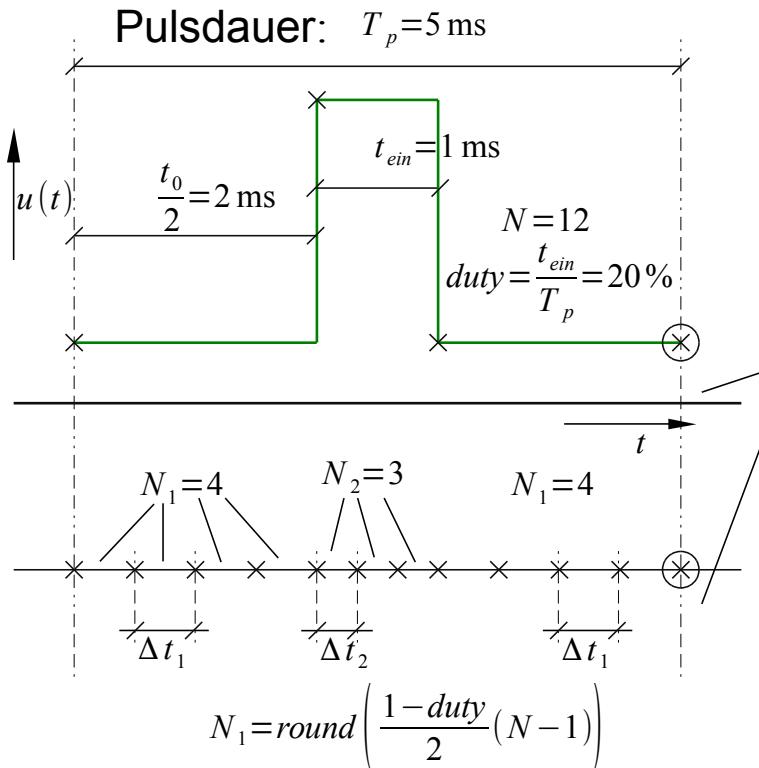
## Lösung der Differentialgleichung

Die Spannung an der ohmsch-induktiven Last ändert sich jeweils sprungförmig. Wegen der Unstetigkeiten des Spannungsverlaufes muss der resultierende Verlauf des Stromes abschnittsweise berechnet werden. Die Lösung kann auch analytisch berechnet werden.

$$Li + Ri = u_0 \rightarrow i(t) = i_\infty + (i_0 - i_\infty) e^{-\frac{t}{T}} \quad \text{mit} \quad i_\infty = \frac{u_0}{R}, \quad T = \frac{L}{R}$$

Hausaufgabe: Lösen Sie die Differentialgleichung mit Hilfe der Laplace-Transformation

# Vorgabe von Rechenpunkten



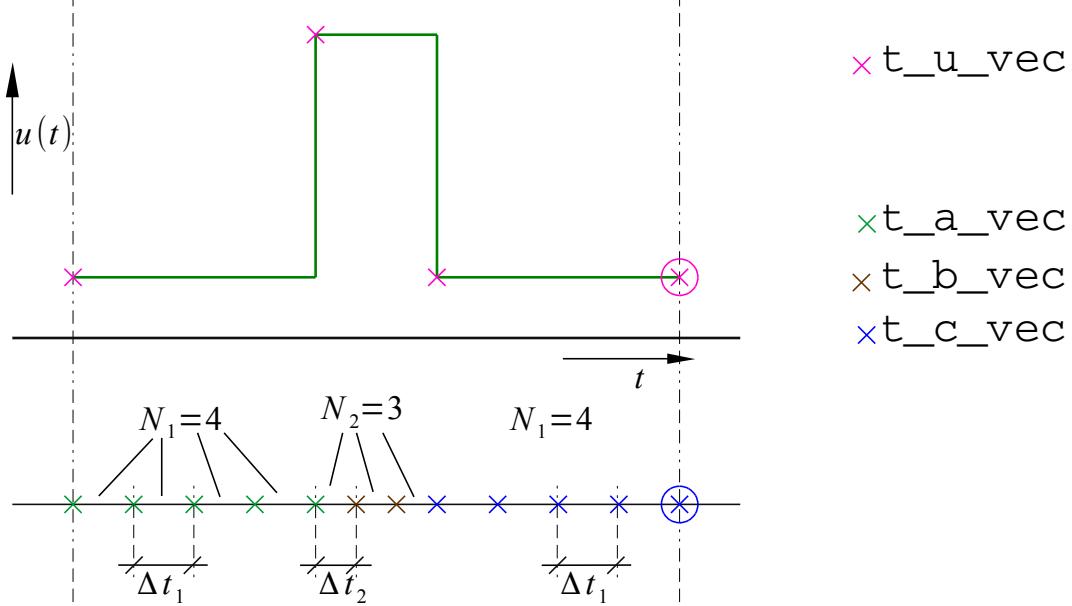
Zur Berechnung des Stromes muss jeder Puls in drei Abschnitte unterteilt werden

Punkte gehören zum nächsten Zyklus

Wichtig ist das genaue „Treffen“ der Unstetigkeitsstellen

## Teilvektoren

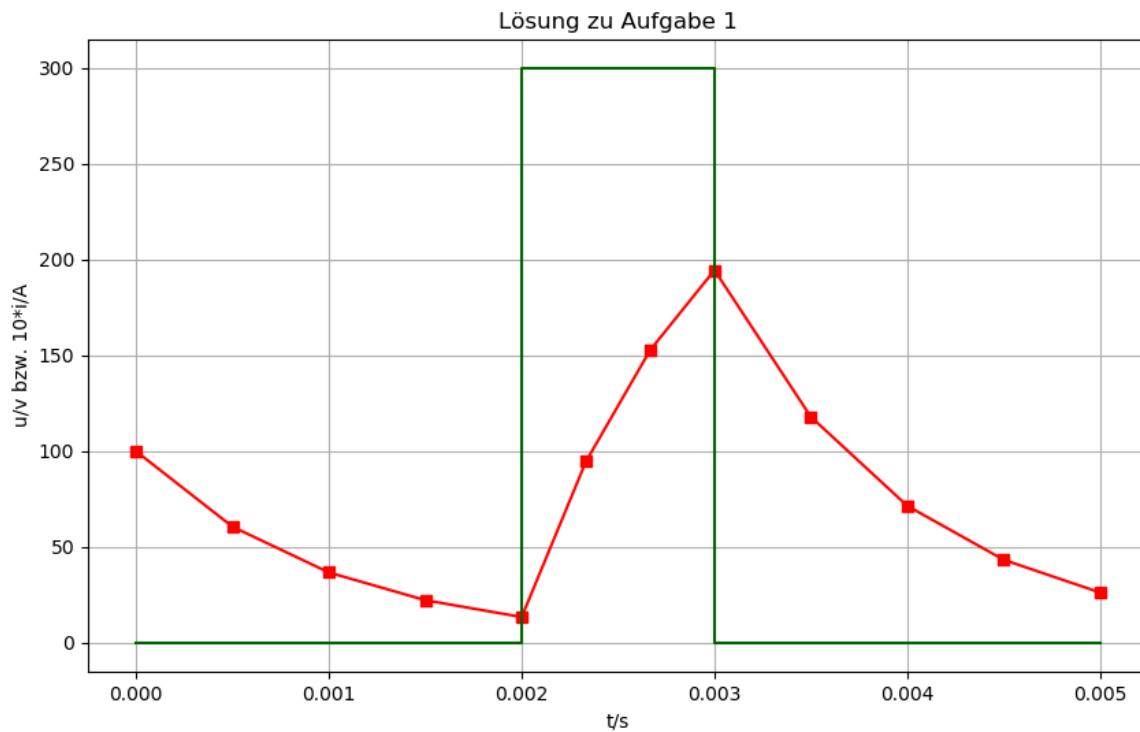
```
u=np.array([0, u_0, 0, 0])
```



## Simulation eines Vierquadrantenstellers

### Aufgabe 1:

Berechnen Sie den Verlauf des Stromes an einer ohmsch-induktiven Last ( $R=10\Omega, L=10\text{ mH}$ ) für  $T_p=5\text{ ms}, u_0=300\text{ V}, i_0=10\text{ A}$  und einer relativen Einschaltzeit von  $duty=20\%$ . Berechnen Sie zunächst  $N_1, N_2, t_{\text{ein}}, t_0_{\text{halbe}}, \Delta t_1$ , und  $\Delta t_2$ . Erstellen Sie dann jeweils drei Vektoren für die Zeitabschnitte ( $t_a_{\text{vec}}, t_b_{\text{vec}}, t_c_{\text{vec}}$ ) und die dazugehörigen Stromverläufe ( $i_a_{\text{vec}}, i_b_{\text{vec}}, i_c_{\text{vec}}$ ). Setzen Sie diese einzelnen Vektoren zu einem Vektor für die Zeit und einen Vektor für den Strom zusammen. Berechnen Sie den Stromverlauf an  $N=12$  Punkten. Erstellen Sie darüber hinaus für die Spannung einen Vektor  $t_u$  so, dass die Spannung mit dem `ax.plot(?, ?, drawstyle='steps-post')` – Befehl dargestellt werden kann.



### Aufgabe 2:

Erweitern Sie Ihr Programm aus Aufgabenteil 1 der Art, dass Ihr Programm beliebige Werte für  $duty$  und  $N$  verarbeiten kann. Bedenken Sie dabei folgendes:

- Für sehr kleine Einschaltzeiten müssen mittig zumindest zwei Punkte verbleiben  $N_2=1$ . Ist  $N$  ungerade, müssen mittig drei Punkte verbleiben, damit die Symmetrie erhalten bleibt  $N_2=2$ .
- Für sehr große Einschaltzeiten müssen rechts und links zumindest zwei Punkte verbleiben.  $N_1=1$

### Aufgabe 3:

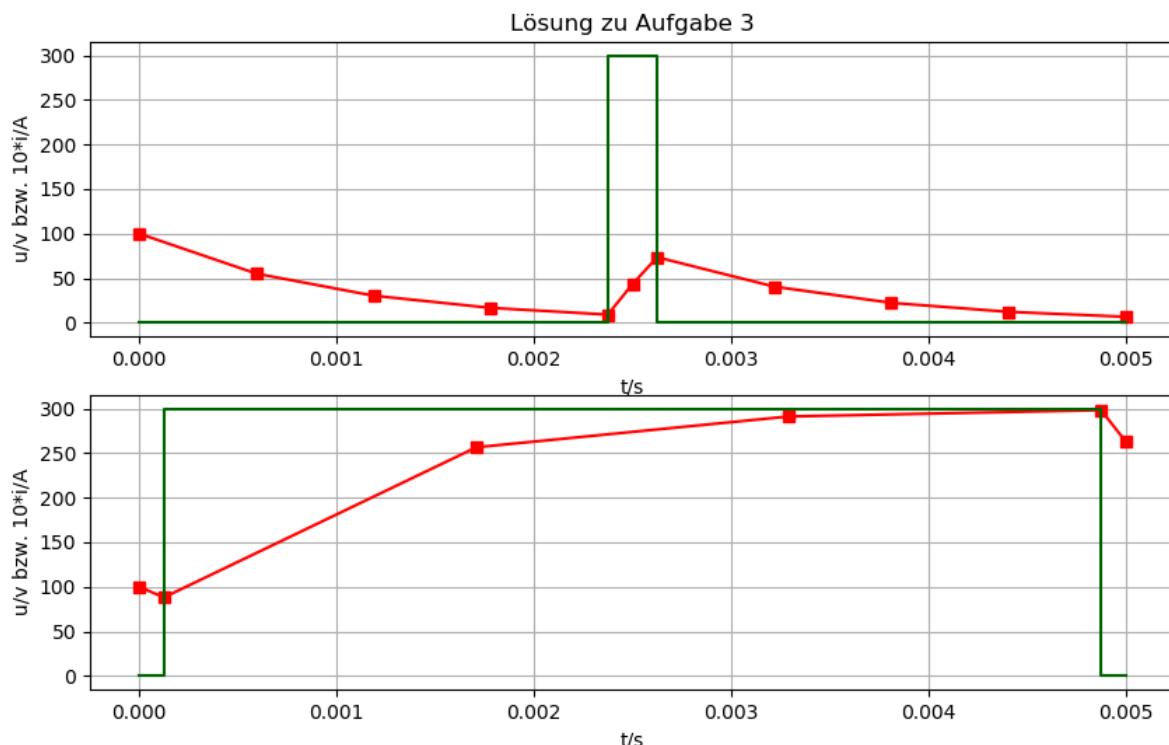
Die in den Aufgaben 1 und 2 implementierte Funktionalität soll in eine Funktion gepackt werden:

```
def RL_circuit(duty, T_p, N, u_0, i_0, R, L):  
    # Quelltext  
    return i, t
```

Überzeugen Sie sich von der korrekten Arbeitsweise Ihrer Funktion mit folgenden Aufrufen:

```
i, t=RL_circuit(0.05, 5e-3, 11, 300, 10, R, L);  
i, t=RL_circuit(0.95, 5e-3, 6, 300, 10, R, L);
```

Stellen Sie die Ergebnisse wie folgt dar:



### Aufgabe 4:

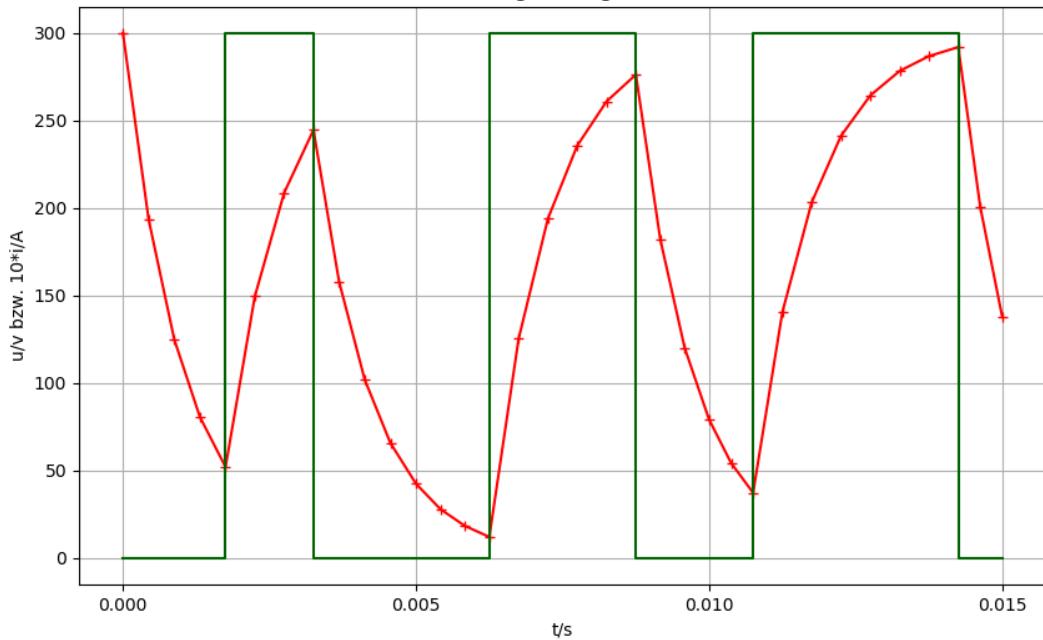
Berechnen Sie den Verlauf des Stromes an 3 aufeinander folgenden Zyklen. Die Mittelwerte der Spannungen sind dazu in einem Vektor gegeben:

```
u_mittel_vec=np.array([90, 150, 210])
```

Verwenden Sie eine `for`-Schleife zur Berechnung des Strom- und Spannungsverlaufes.

Weitere Daten:  $N=12, T_p=5\text{ ms}$ . Stellen Sie Ihre Ergebnisse wie folgt dar:

Lösung zu Aufgabe 4



----- Selbständige Weiterarbeit -----

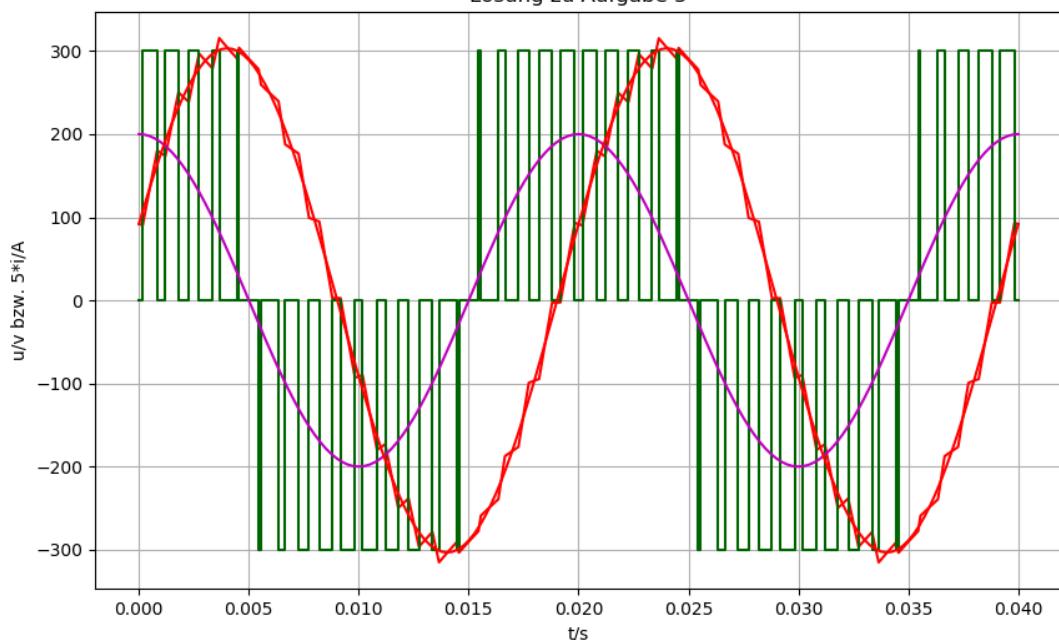
### Aufgabe 5:

Berechnen Sie den Verlauf des Stromes, wenn Sie mittelwertmässig eine sinusförmige Spannung  $u(t)=200 \text{ V} \cdot \cos(2\pi 50 \cdot t)$  nachbilden. Vergleichen Sie den Stromverlauf mit dem Ergebnis, das Sie bei einem kontinuierlichen Verlauf der Spannung erhalten.

Berechnen Sie analytisch den Startwert für  $i(t)$ , damit kein Einschwingvorgang stattfindet. Bedenken Sie dass  $duty$  immer positiv ist. Weiterhin gelte:

$R=1 \Omega$ ,  $L=10 \text{ mH}$ ,  $N=6$ ,  $T_p=1 \text{ ms}$ . Stellen Sie die Ergebnisse wie folgt dar:

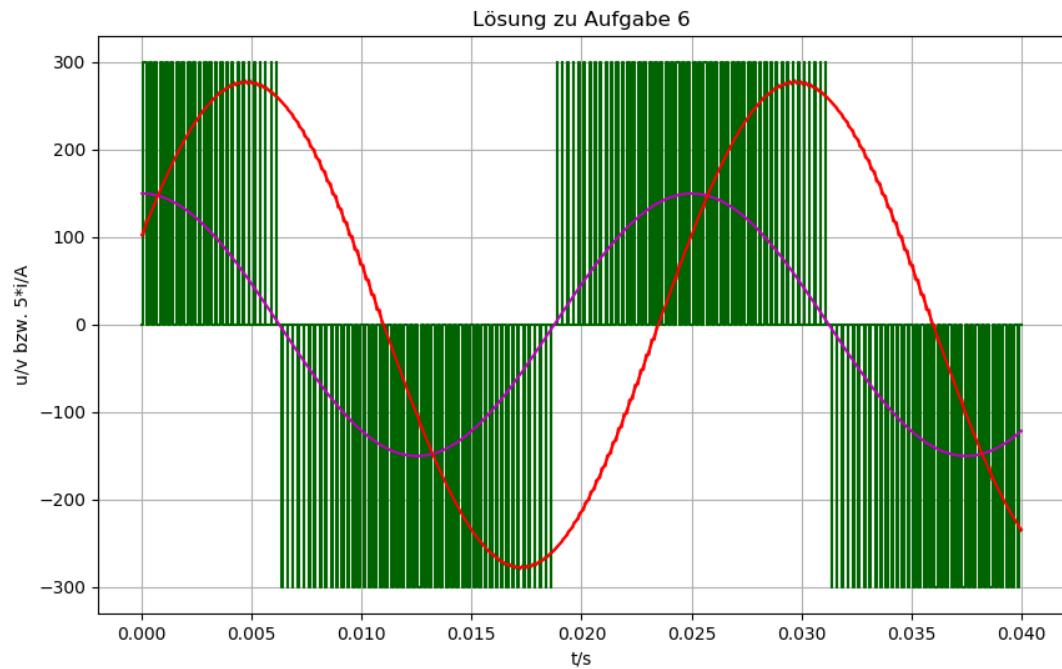
Lösung zu Aufgabe 5



**Aufgabe 6:**

Experimentieren Sie mit dem Skript aus Aufgabe 5 und simulieren Sie das Verhalten, wenn Sie eine Spannung mit der Amplitude  $\hat{u}=150\text{ V}$  und der Frequenz  $f=40\text{ Hz}$  bei einer Pulsdauer von  $T_p=250\mu\text{s}$  erzeugen wollen. Überzeugen Sie sich davon, dass

1. bei dieser kleinen Pulsdauer der Stromrippel sehr klein wird und der Strom annähernd sinusförmig aussieht. (In der Antriebstechnik realisiert man üblicherweise  $T_p=125\mu\text{s}(8\text{ kHz})$  oder sogar  $T_p=62.5\mu\text{s}(16\text{ kHz})$ .)
2. sich sowohl die Amplitude als auch die Frequenz der Spannung sehr leicht einstellen lassen.
3. annähernd keine Verluste entstehen. (In der Praxis sind die Schalter nicht ideal, so dass dadurch Verluste entstehen. Siehe Vorlesung: Leistungselektronik)

**Aufgabe 7:**

Lösen Sie die Differentialgleichung für das RL-Glied mit Hilfe der Laplace-Transformation!

# Modellbildung und Simulation

## Numerische Lösungsverfahren

### Lernziele

Am Ende des Kapitels ...

- ✓ ... kennen Sie unterschiedliche Verfahren zur Lösung der Zustandsgleichung
- ✓ ... sind Sie in der Lage, Differentialgleichungen selbst numerisch zu lösen (interessant für embedded systems)
- ✓ ... verstehen Sie die implementierten Lösungsverfahren besser und können numerische Probleme so besser lösen
- ✓ ... kennen Sie die Grenzen der Lösungsverfahren bei Unstetigkeiten und wissen, wie man solche Probleme löst

# Lernziele

Am Ende des Kapitels ...

- ✓ ... wissen Sie, was eine „zero crossing detection“ ist
- ✓ ... haben Sie mit logarithmischen Plots gearbeitet
- ✓ ... haben Sie gelernt, numerischen Rechenergebnissen gegenüber misstrauisch zu sein
- ✓ ... haben Sie die Bedeutung der Parameter `rtol` und `atol` kennen gelernt und wissen, wie man das Lösungsverfahren beeinflusst

## Numerische Lösung von gewöhnlichen Differentialgleichungen

Jede Differentialgleichung höherer Ordnung lässt sich in ein Differentialgleichungssystem erster Ordnung umformen

$$\dot{\underline{x}} = \underline{A} \cdot \underline{x} + \underline{B} \cdot \underline{u} \quad \text{Allgemein: } \dot{\underline{x}} = f(\underline{x}, \underline{u})$$

Integration:

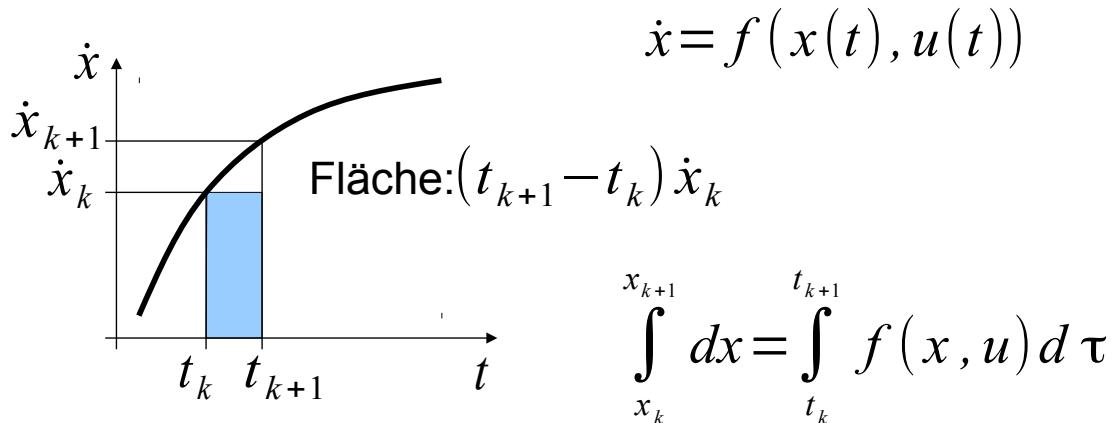
$$\underline{x} = \underline{x}_0 + \int_{t_0}^t f(\underline{x}, \underline{u}) d\tau$$

Diskretisierung bei konstanter Zeitschrittweite:

$$t_k = t_0 + k \cdot \Delta t$$

Im folgenden: Beschränkung auf eine Veränderliche

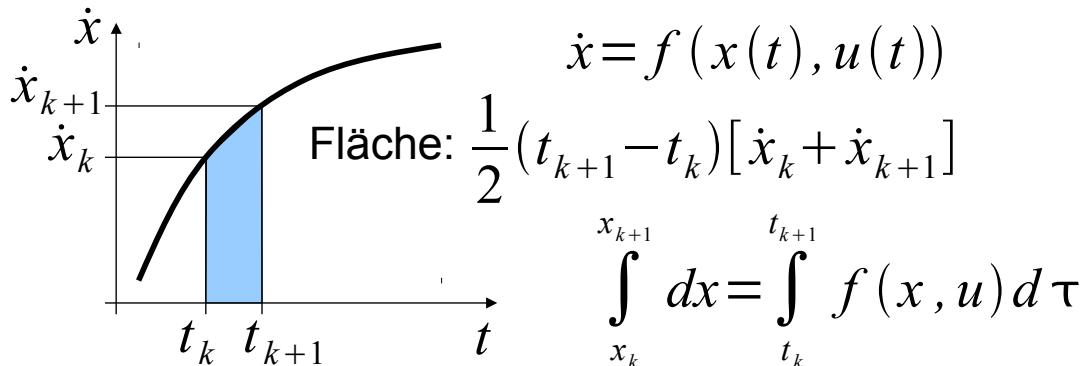
# Euler-Verfahren



$$\rightarrow x_{k+1} \approx x_k + (t_{k+1} - t_k) \cdot f(t_k, x_k, u_k)$$

$x_{k+1} \approx x_k + \Delta t \cdot f(t_k, x_k, u_k)$  bei konstanter Zeitschrittweite

# Trapezregel (Heun-Verfahren)



$$\rightarrow x_{k+1} \approx x_k + \frac{(t_{k+1} - t_k)}{2} \cdot [f(t_k, x_k, u_k) + f(t_{k+1}, x_{k+1}, u_{k+1})]$$

1                            2

Lösung durch Prädiktor-Korrektorschritt

1     $k_1 = f(t_k, x_k, u_k) \quad \hat{x} = x_k + \Delta t \cdot k_1$

2     $k_2 = f(t_{k+1}, \hat{x}, u_{k+1}) \quad x_{k+1} = x_k + 0.5 \cdot \Delta t \cdot (k_1 + k_2)$

# RUNGE-KUTTA - Verfahren

weitergehende Mittelung der Ableitungen

$$x_{k+1} \approx x_k + \Delta t \cdot \frac{k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4}{6}$$

mit

$$k_1 = f(t_k, x_k, u_k)$$

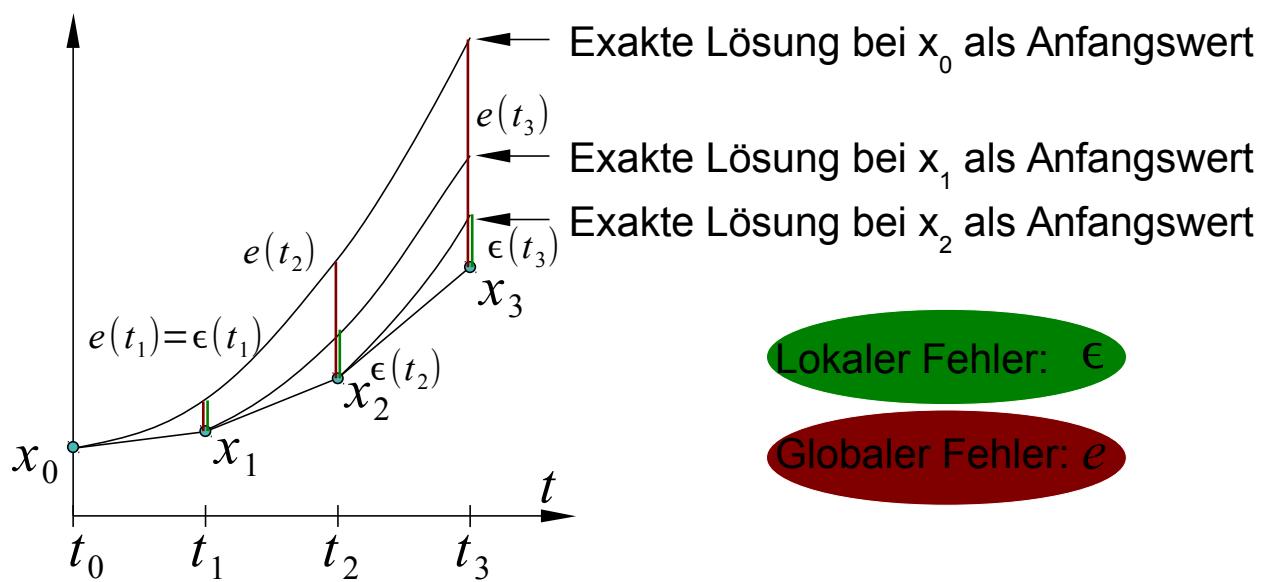
$$k_2 = f(t_k + \Delta t / 2, x_k + \Delta t / 2 \cdot k_1, u_{k+1/2})$$

$$k_3 = f(t_k + \Delta t / 2, x_k + \Delta t / 2 \cdot k_2, u_{k+1/2})$$

$$k_4 = f(t_k + \Delta t, x_k + \Delta t \cdot k_3, u_{k+1})$$

Frage: Welche Vor- und Nachteile haben die unterschiedlichen Verfahren?

## Visualisierung: Lokaler- und globaler Fehler



Schrittweite:  $\Delta t$

# Definition: Lokaler- und globaler Fehler

## Globaler Fehler:

Der globale Fehler  $e(t_k)$  ist gleich der Differenz zwischen der numerischen und der exakten Lösung an der Stelle  $e(t_k)$ .

## Lokaler Fehler:

Der lokale Fehler  $\epsilon(t_k)$  beschreibt die Abweichung zwischen exakter und numerischer Lösung nach einem Schritt. Wenn also die exakte Lösung den Anfangswert des vorherigen numerisch berechneten Wertes hat.

## Güte der unterschiedlichen Verfahren

Bei kleiner Zeitschrittweite erwartet man eine große Genauigkeit aber auch eine große Rechenzeit.

Bei extrem kleiner Zeitschrittweite wird man ungenaue Ergebnisse erhalten, weil dann die Zahlenauflösung des Rechners eine Rolle spielt.

Von einem Lösungsverfahren wird man mindestens erwarten, dass sich der globale Fehler halbiert, wenn man mit halber Zeitschrittweite rechnet. Man spricht dann von der Konvergenzordnung 1.

Nimmt der globale Fehler bei Halbierung der Zeitschrittweite um den Faktor  $0.5^p$  ab,  
so hat das Verfahren die Konvergenzordnung  $p$ .

# Schätzung des lokalen Fehlers

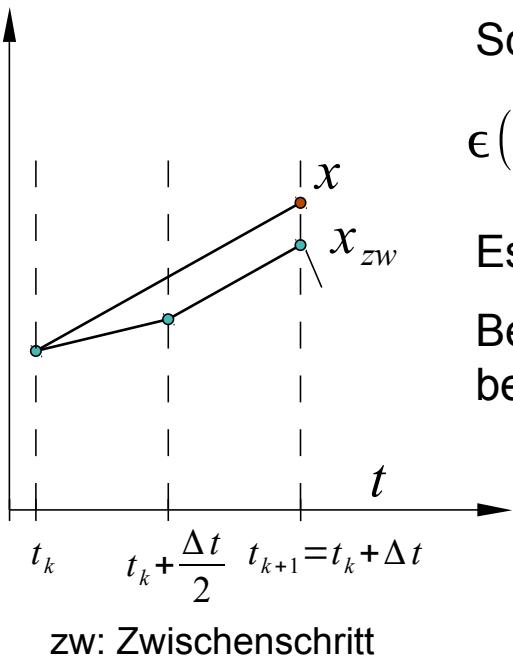
In der Regel kennt man weder den lokalen noch den globalen Fehler, da die exakte Lösung unbekannt ist.

Den lokalen Fehler kann man durch Berechnung eines zusätzlichen Zwischenschrittes bei  $\Delta t/2$  abschätzen.

Es gibt Bereiche in der Lösung, in denen sich die Zustandsgrößen rasch ändern, in anderen Zeitbereichen verhalten sich die Zustandsgrößen jedoch annähernd statisch. Beispiel: Einschwingvorgang.

Die Schätzung des lokalen Fehlers lenkt den Blick auf eine Möglichkeit zur Schrittweitensteuerung. Dabei wird die Schrittweite so verändert, dass der lokale Fehler annähernd gleich bleibt.

## Schrittweitensteuerung beim Euler-Verfahren (ohne Herleitung)



Schätzwert für den lokalen Fehler:

$$\epsilon(t_{k+1}, \Delta t) \approx \hat{\epsilon}_{k+1} := 2\Delta = 2|x_{zw} - x|$$

Es gilt:  $\epsilon \sim \Delta t^2$

Berechnung einer neuen Schrittweite bei vorgegebenem lokalem Fehler:

$$\epsilon_{soll} = tol \sim \Delta t_{soll}^2$$

$$\rightarrow \Delta t_{neu} = \Delta t \sqrt{\frac{tol}{\hat{\epsilon}_{k+1}}}$$

# Einführung einer relativen (rtol) und absoluten Fehlergrenze (atol)

In der Praxis führt man in der Regel zwei Fehlergrenzen ein, eine absolute (atol) und eine relative (rtol).

$$tol = \max(atol, rtol \cdot |x|)$$

Ist  $x$  sehr klein, so ist die Prüfung auf relative Toleranz nicht sinnvoll, bei sehr großen Werten von  $x$  (z.B.  $10^{30}$ ) erscheint es aber wenig sinnvoll, eine absolute Genauigkeit von z.B.  $10^{-8}$  anzustreben. Eine Kombination aus relativer- und absoluter Toleranz erscheint am sinnvollsten.

## Einstellung der relativen und absoluten Toleranz

```
scipy.integrate.solve_ivp(fun, t_span, y0,  
method='RK45', t_eval=None, dense_output=False,  
events=None, vectorized=False, **options)
```

**rtol, atol:** float or array\_like, optional

Relative and absolute tolerances. The solver keeps the local error estimates less than  $atol + rtol * abs(y)$ . . . . Default values are 1e-3 for rtol and 1e-6 for atol.

# Probleme bei Unstetigkeiten

- Die vorgestellten Verfahren versagen bei Unstetigkeiten
- Unstetigkeiten entstehen häufig durch Schalthandlungen
- Ideale Bauelemente (z.B. Dioden) weisen Unstetigkeiten auf
- Lösung des Problems: Erkennen der Unstetigkeitsstelle, Anhalten des Lösungsverfahrens, Neustart des Lösungsverfahrens mit neuen Anfangswerten
- Die Aufgabe ist vergleichsweise einfach, wenn der Zeitpunkt der Unstetigkeit bekannt ist (Vierquadrantensteller!)
- Ist der Zeitpunkt der Unstetigkeit (Begriffe: „zero crossing detection“ (Der Nulldurchgang irgendeiner Größe muss erkannt werden.) bzw. „rootfinding“) nicht bekannt, so muss dieser iterativ bestimmt werden

## Beispiel: Bouncing Ball

The diagram shows a ball at height  $h$  above a horizontal ground line. Three vectors point downwards from the ball:  $mg$  (gravitational force),  $m\dot{v}$  (velocity), and  $c v^3$  (dissipation force). To the right of the ball, the equations of motion are listed:

$$m \ddot{v} + mg + cv^3 = 0$$
$$\rightarrow$$
$$\dot{v} = -g - \alpha v^3 \text{ mit } \alpha = \frac{c}{m}$$
$$\dot{h} = v$$
$$h = 0 \rightarrow v := -v$$

Die Deformation des Balles bei Bodenkontakt wird vernachlässigt

Für  $h=0$  wird stattdessen das Vorzeichen der Geschwindigkeit verändert. Die kinetische Energie bleibt dabei gleich.

## Numerische Lösungsverfahren

**Aufgabe 1:** Programmieren Sie das Euler-Verfahren, das Heun-Verfahren und das Runge-Kutta-Verfahren zur Lösung der folgenden Differentialgleichung:

$$\dot{x} = -\frac{2tx^2}{t^2+1} \text{ mit } x(t=0)=2 .$$

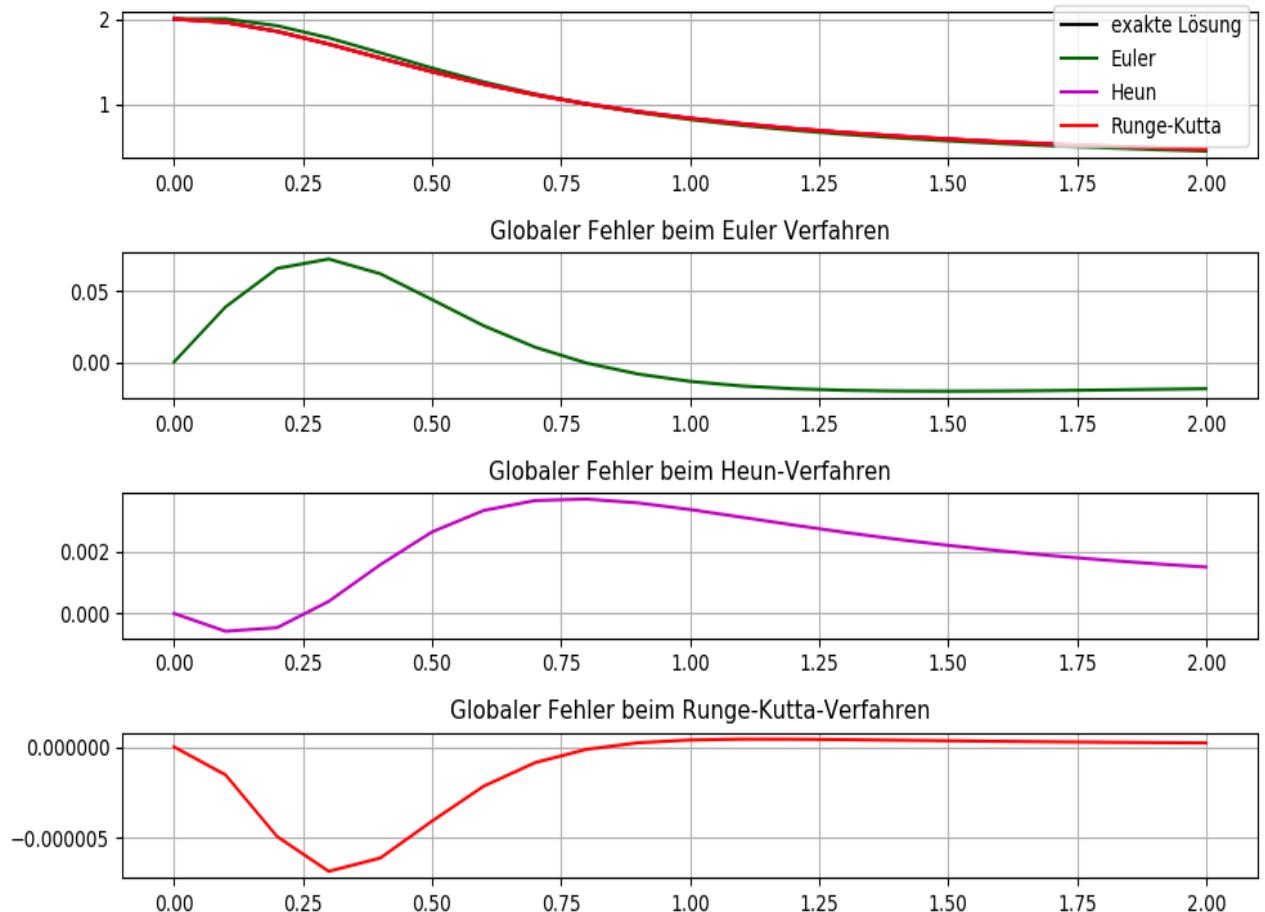
Die exakte Lösung lautet:  $x(t) = \frac{1}{\ln(t^2+1)+0.5}$ .

Lagern Sie die Berechnung von  $\dot{x}$  in folgende Funktion aus  
def xdot\_fkt(t, x):

```
xdot=(-2.*t*x**2)/(t**2+1.)
return xdot
```

Verwenden Sie for-Schleifen, um die einzelnen Zeitwerte für die unterschiedlichen Verfahren zu bestimmen. Zeitschrittweite:  $\Delta t=0.1$ .

Stellen Sie Ihre Ergebnisse wie folgt dar:

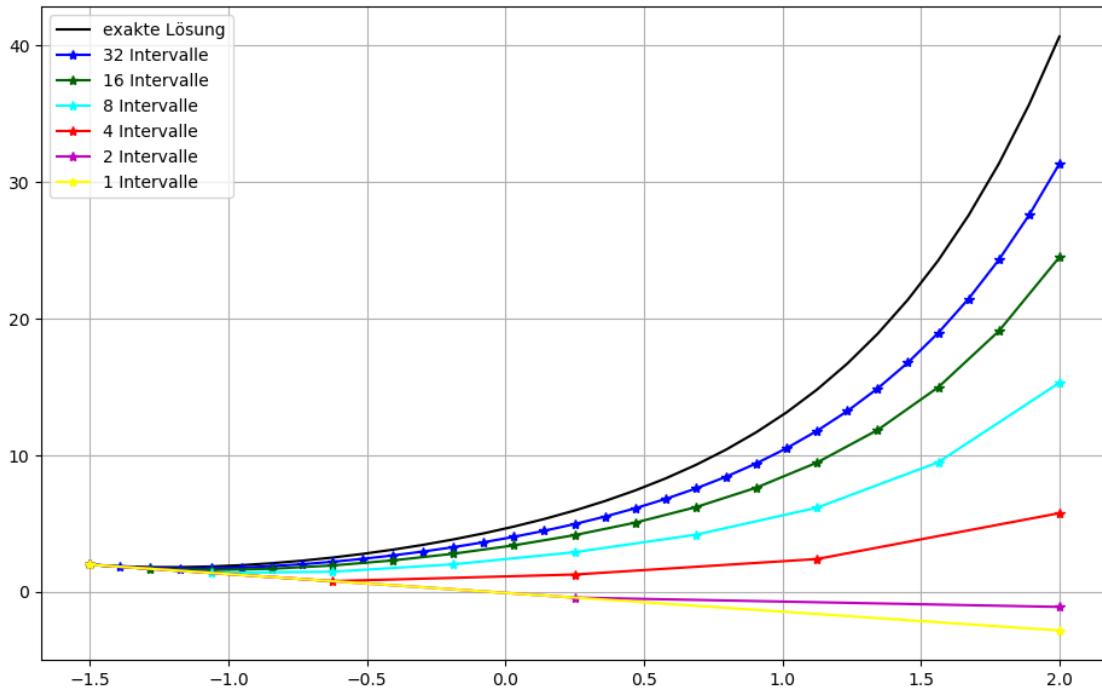


**Aufgabe 2:** Lösen Sie die folgende Differentialgleichung mit dem Euler-Verfahren und vergleichen Sie unterschiedliche Zeitschrittweiten.

$\dot{x} = x + t^3$  mit  $x(-1.5) = 2$  im Intervall  $-1.5 \leq t \leq 2$ . Die exakte Lösung lautet:

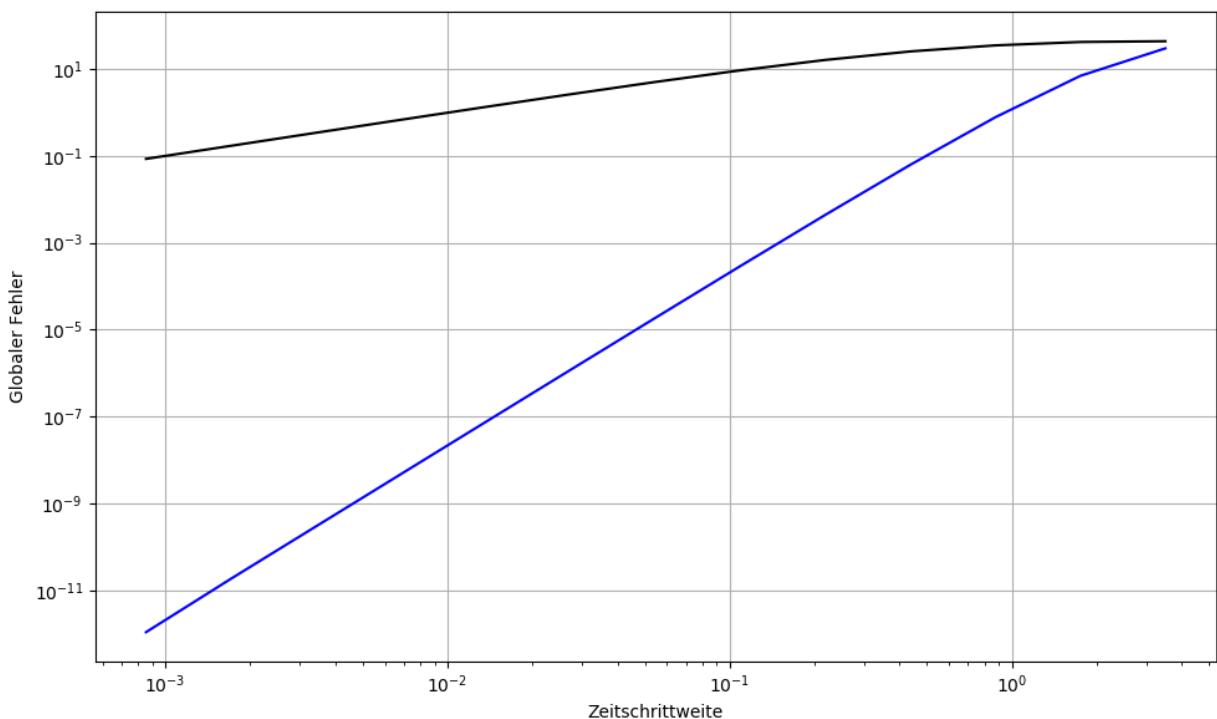
$$x(t) = -t^3 - 3t^2 - 6t - 6 + \frac{19}{8}e^{1.5+t}$$

Stellen Sie Ihre Ergebnisse wie folgt dar:



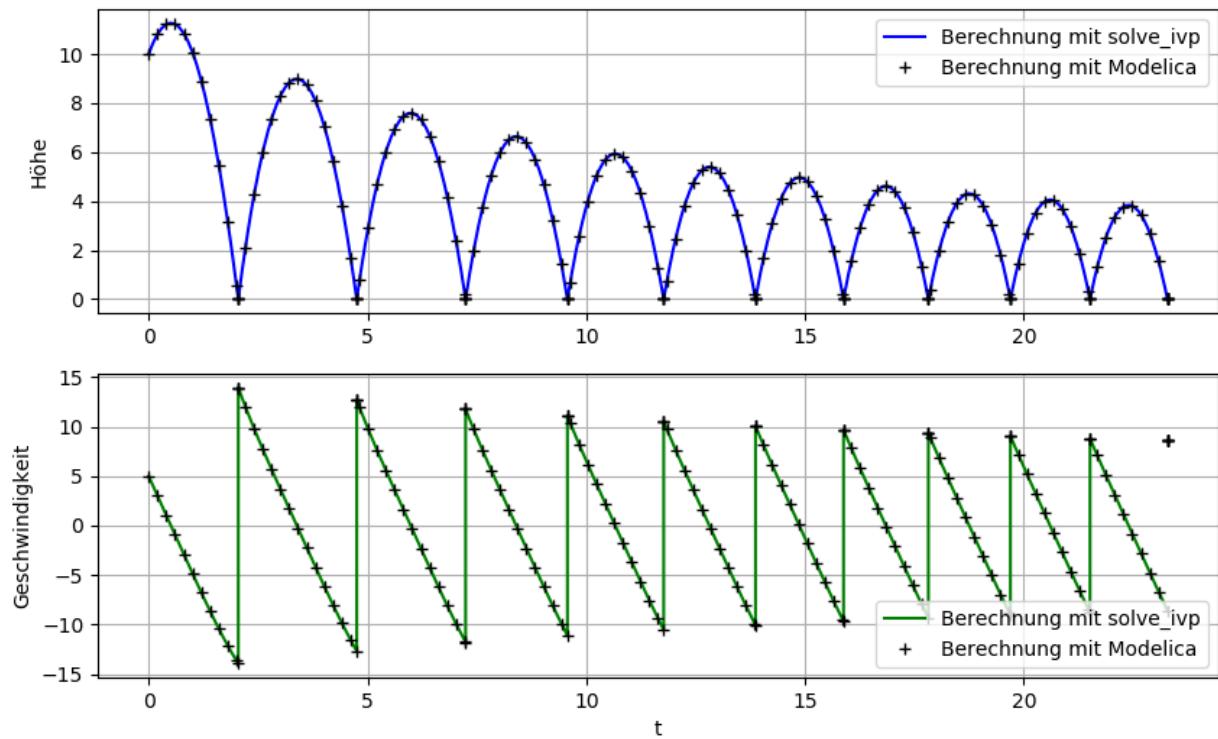
**Aufgabe 3:** Berechnen Sie  $x(t=2)$  für die Differentialgleichung aus Aufgabenteil 2 und stellen Sie den globalen Fehler in Abhängigkeit der Zeitschrittweite für das Euler- und das Runge-Kutta-Verfahren dar.

Wie kann man aus dieser Abbildung die Ordnung der Verfahren ablesen?



----- Selbständige Weiterarbeit -----

**Aufgabe 4:** Analysieren Sie die Dateien A4\_vorlage.py und bounce.mo. Erstellen Sie die Dateien A4.py zur Berechnung der nachfolgenden Abbildung. Warum bewegt sich der Ball für  $t=0$  nach oben?



# Modellbildung und Simulation

## Modelica – Eigene Modelle

### Lernziele

Am Ende des Kapitels ...

- ✓ ... haben Sie intensiv mit OpenModelica gearbeitet
- ✓ ... haben Sie den objektorientierten Ansatz von Modelica verstanden
- ✓ ... haben Sie die Besonderheit von idealen Schaltelementen erfasst
- ✓ ... wissen Sie, wie man mit Modelica eine „zero crossing detection“ realisiert
- ✓ ... können Sie eigene Modelica Modelle erstellen
- ✓ ... wissen Sie, dass man Elemente mit Unstetigkeiten in einer parametrierten Darstellung realisieren kann

# DAE anstatt ODE

- Durch Modelica werden einzelne Komponenten beschrieben, ohne vorher festzulegen, was Eingangs- und was Ausgangsgrößen sind.
- Die Gleichungen können als echte Gleichungen und nicht als Zuweisungen vorgegeben werden. Die Umformarbeiten zur Erstellung der ode (ordinary differential equation) werden vom Modelica-Compiler vorgenommen. Der Benutzer wird damit nicht belastet.
- Das bekannteste und am weitesten entwickelte Programm, dass auf dieser Grundidee arbeitet, ist das kommerzielle Programm Dymola.
- OpenModelica stellt eine kostenlose und leistungsfähige OpenSource-Alternative dar.

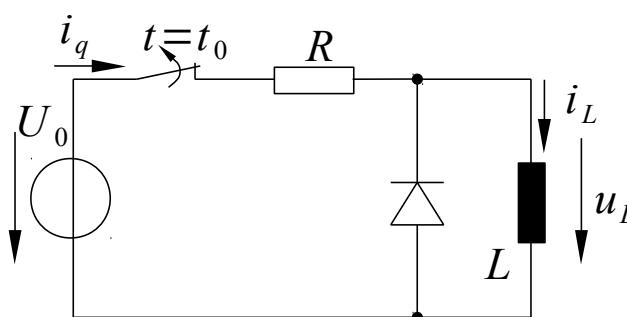
# Modelica

- Modelica bietet einen objektorientierten Ansatz zur Beschreibung von dynamischen Systemen und vermeidet das Problem der Umformung DAE in ode
- Modelica ist eine offene Beschreibungssprache für dynamische Systeme ([www.modelica.org](http://www.modelica.org))
- Viele Modelica-Blöcke sind in der Standardbibliothek bereits vorhanden und können intuitiv genutzt werden
- Viele weitere Modelica-Blöcke werden bei der Standardinstallation bereits mitinstalliert und können zusätzlich geladen werden
- Eigene Modelica-Blöcke können leicht implementiert und in eigenen Bibliotheken gespeichert werden

# Modelica

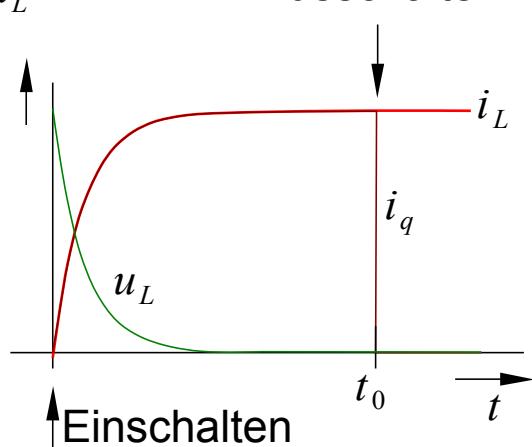
- Modelica Modelle werden in Textdateien mit der Endung .mo definiert
- Die Beschreibungssprache ist sehr umfangreich, für einfache lineare Modelle jedoch annähernd selbsterklärend
- Allen Zweipolen ist gemein, dass der Strom der hineinströmt auch wieder herausfließt und dass der Spannungsabfall gleich der Potentialdifferenz an Eingang und Ausgang ist
- Die Bedingungen „Flussgleichung“ und „Potentialgleichung“ sind bei allen anderen Systemen identisch. Beispiele für Potentialgrößen: Geschwindigkeit, Druck, Dichte. Beispiele für Flussgrößen: Kraft, Moment, Volumenstrom
- Hydraulik-, oder Mechanikkomponenten lassen sich deshalb ebenfalls einfach aufbauen

## Beispiel: RL-Schaltkreis mit Freilaufdiode



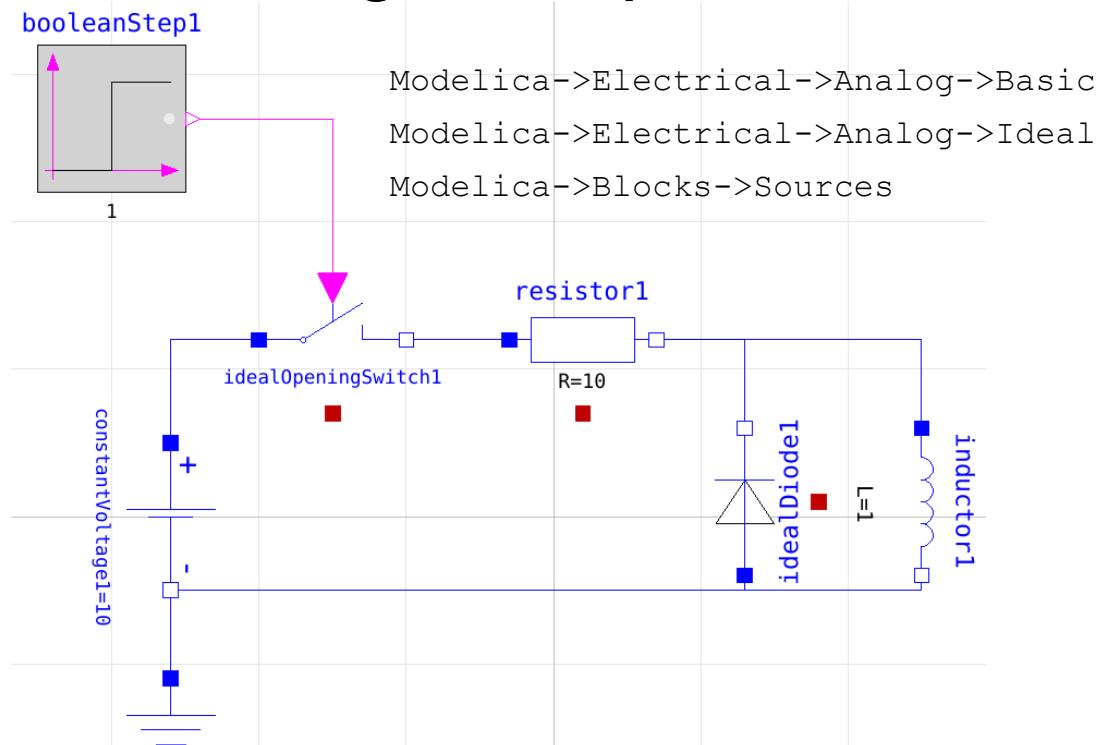
Erwartetes Ergebnis:

Ausschalten



Die Aufgabe ist (bei idealen Elementen) auch leicht analytisch lösbar

# Realisierung mit OpenModelica



## Beispielmodell: Resistor

```
partial model OnePort
  SI.Voltage v;
  SI.Current i;
  PositivePin p
  NegativePin n
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

```
within ModSimBib;

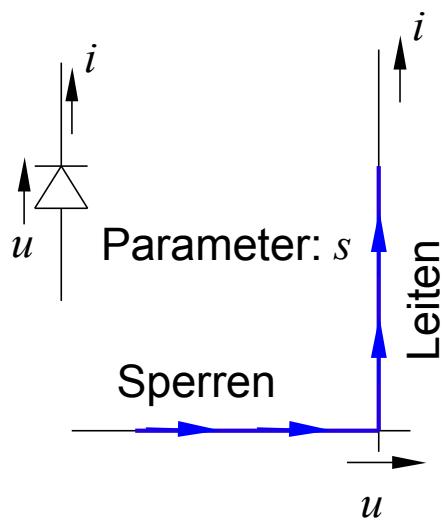
model own_resistor
  parameter Real R=1;
  extends Modelica.Electrical.Analog.Interfaces.OnePort;

equation
  v = R*i;
end own_resistor;
```

Die Formulierung der Gleichungen im `equation`-Block ist beliebig. Gleichbedeutend wären z.B.:  $i=v/R$  oder  $v/R=i=1$ . Es handelt sich um echte „Gleichungen“ nicht um Zuweisungen, wie in anderen Programmiersprachen.

# Einführen eines Parameters für unstetige Kennlinien

Beispiel Diode:



Unstetig verlaufende Kennlinien sind ein Problem für die „zero crossing detection“  
Abhilfe: Einführung eines sich kontinuierlich ändernden Parameters s

Im Sperrzustand entspricht s der Spannung, im leitenden Zustand dem Strom

## Diode innerhalb der ModSimBib

```
within ModSimBib;  
model own_diode  
extends  
    Modelica.Electrical.Analog.Interfaces.OnePort;  
protected  
    Real s;  
equation  
    v = s * (if s < 0 then 1 else 0);  
    i = s * (if s < 0 then 0 else 1);  
end own_diode;
```

Ein `if`-Befehl führt automatisch zu einer „zero crossing detection“. Vermeiden könnte man dies durch:  
`if noevent(s<0) then ...`

# Idealer Schalter („Öffner“)

```
within ModSimBib;  
model own_switch  
  extends  
    Modelica.Electrical.Analog.Interfaces.OnePort;  
    Modelica.Blocks.Interfaces.BooleanInput control;  
protected  
  Real s;  
equation  
  v = s * (if control then 1 else 0);  
  i = s * (if control then 0 else 1);  
end own_switch;
```

control= true → Schalter geöffnet  
control= false → Schalter geschlossen

## Vorteile durch die Modellerstellung mit Modelica

- Einzelkomponenten können modelliert werden, ohne die tatsächliche Verwendung vorab zu wissen
- Die Einzelkomponenten können intuitiv verwendet werden. Beispielsweise können (wie bei LTspice) Schaltpläne direkt eingegeben werden.
- Die komponentenorientierte Modellierung funktioniert auch bei anderen physikalischen Systemen. Schauen Sie, wie viele unterschiedliche Modelle bereits in der Standardbibliothek vorhanden sind.
- Insbesondere Unstetigkeiten („zero crossings“) lassen sich sehr einfach mit Modelica verarbeiten.

# Eigene Bibliotheken verwenden

Tools->Options->Libraries



Unter Add ... /ModSimBib/package.mo auswählen und OpenModelica neu starten.

## Vierquadrantensteller mit Modelica

Anhand des bereits untersuchten Vierquadrantenstellers wird gezeigt, wie einfach die Modellierung eines Systems mit „zero crossing detection“ in Modelica ist.

Idee: Zu Beginn eines Zyklus' wird ein Zähler gestartet und solange inkrementiert, bis die Mitte des Zyklus' erreicht ist. Dann wird er wieder dekrementiert bis zum Ende des Zyklus. Der Zähler `cnt` startet bei null und endet bei eins. Aus einem Vergleich des Sollwertes der Spannung mit dem Zählerstand gewinnt man die Information, wann die Ausgangsspannung ein- oder ausgeschaltet wird.

$$\bar{u}_a = u_0 \frac{T_{ein}}{T_p} = u_{soll}$$

$$cnt(t) = \frac{2}{T_p} \cdot t$$

$$cnt\left(t = \frac{T_p - T_{ein}}{2}\right) = 1 - \frac{T_{ein}}{T_p} = 1 - \frac{u_{soll}}{u_0}$$

# Vierquadrantensteller mit Modelica (1)

```
model vierqst
  parameter Real T_p = 1e-3; // Pulsdauer
  constant Real pi=Modelica.Constants.pi;
  discrete Real u_hold, u_zk;
  Real u_out; // Hilfsvariable zum Plotten
  Modelica.Blocks.Interfaces.RealInput u_e;
protected
  Real dir(start=-1), cnt(start=0);
  PositivePin p1, p2; NegativePin n1, n2;
equation
  when sample(0,T_p/2) then
    dir=-pre(dir);
  end when; Vereinfachte Darstellung
  when sample(0, T_p) then
    u_hold = u_e; // Sample (Hold durch discrete)
    u_zk = p1.v - n1.v;
  end when;
```

# Vierquadrantensteller mit Modelica (2)

```
der(cnt)=dir*2/T_p;
if cnt < 1 - abs(u_hold) / u_zk then
  // Kurzschluß und Leerlauf (mit Bezug zu Ground)
  p1.i = 0; p2.v - n2.v = 0; n2.v - n1.v = 0;
  0 = p2.i + n2.i + n1.i;
elseif u_hold > 0 then
  // Direkt verbunden
  p2.v = p1.v; n2.v = n1.v; p1.i + p2.i = 0;
  n1.i + n2.i = 0;
else Vereinfachte Darstellung
  // über Kreuz verbunden
  p2.v = n1.v; n2.v = p1.v;
  p1.i + n2.i = 0; n1.i + p2.i = 0;
end if;
end vierqst;
```

# Vierquadrantensteller mit Modelica

## Zusammenfassung

- Die Realisierung des Vierquadrantenstellers mit Modelica ist deutlich komplexer aber auch deutlich weniger aufwändig.
- Der Sourcecode ist wesentlich besser lesbar.
- Das mit Modelica erstellte Modell kann leicht wiederverwendet und erweitert werden.
- Der modulare Ansatz gestattet beispielsweise die Speisung der Schaltung mit einer Kondensatorbatterie (wie in der Praxis üblich).

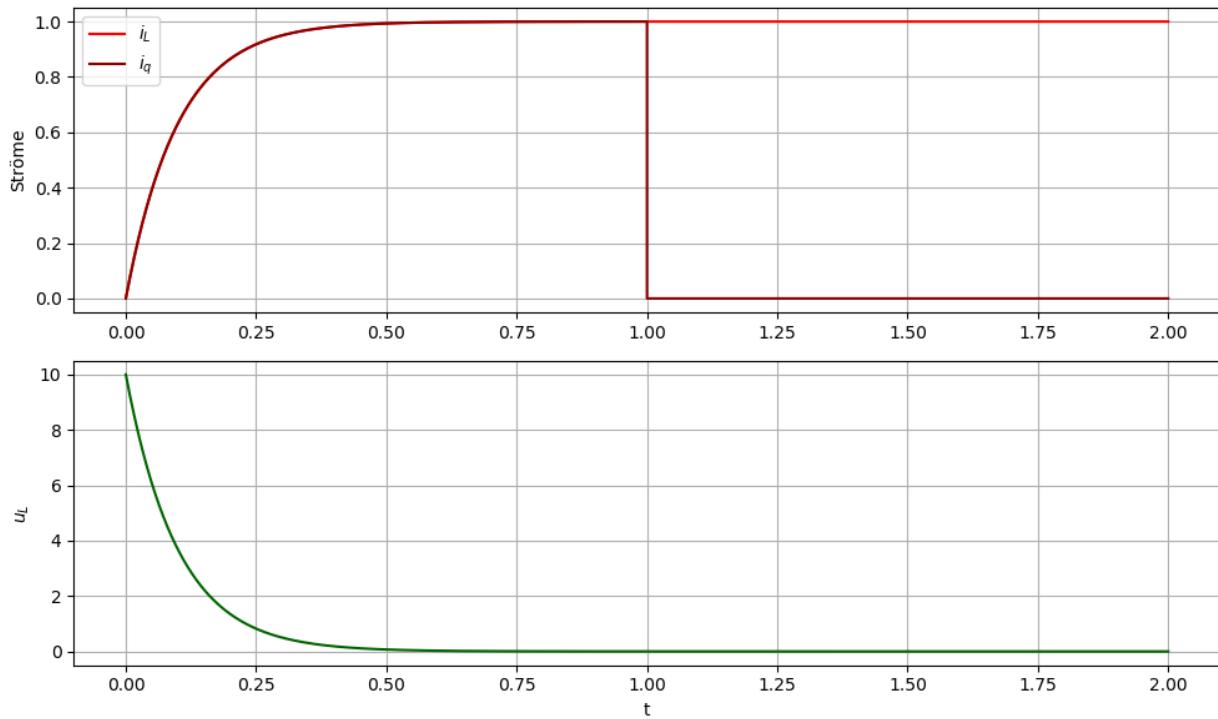
when-Befehl:

Die Anweisungen hinter dem `when`-Befehl werden ausgeführt, wenn die Bedingung falsch war und dann wahr wird. Damit sie noch einmal ausgeführt wird, muss die Bedingung zunächst wieder falsch werden (edge-detection)

## Simulation einer RL- Schaltung

Das Erstellen eigener Modelica-Blöcke soll geübt werden.

**Aufgabe 1:** Simulieren Sie das Ein- und Ausschaltverhalten einer ohmsch-induktiven Last mit Freilaufdiode mit den in der Modelica-Bibliothek vorhandenen Elementen ( $U_0=10\text{ V}$ ,  $R=10\Omega$ ,  $L=1\text{ H}$ ). Starten Sie die Simulation von einem Python-Script aus und stellen Sie Ihre Ergebnisse wie folgt dar:



**Aufgabe 2:** Duplizieren Sie das Modell aus Aufgabe 1 (Rechte Maustaste → Duplicate) und entwickeln Sie einen eigenen Widerstand eine eigene Diode und einen eigenen Schalter! Speichern Sie diese neuen Elemente in einer eigenen Bibliothek. (Die Bibliothek ModSimBib ist bereits vorhanden. Der Schalter kann direkt verwendet werden. Sie müssen lediglich den Widerstand und die Diode erweitern und in Ihr Modell einbinden.) Analysieren Sie den Aufbau der Bibliothek!

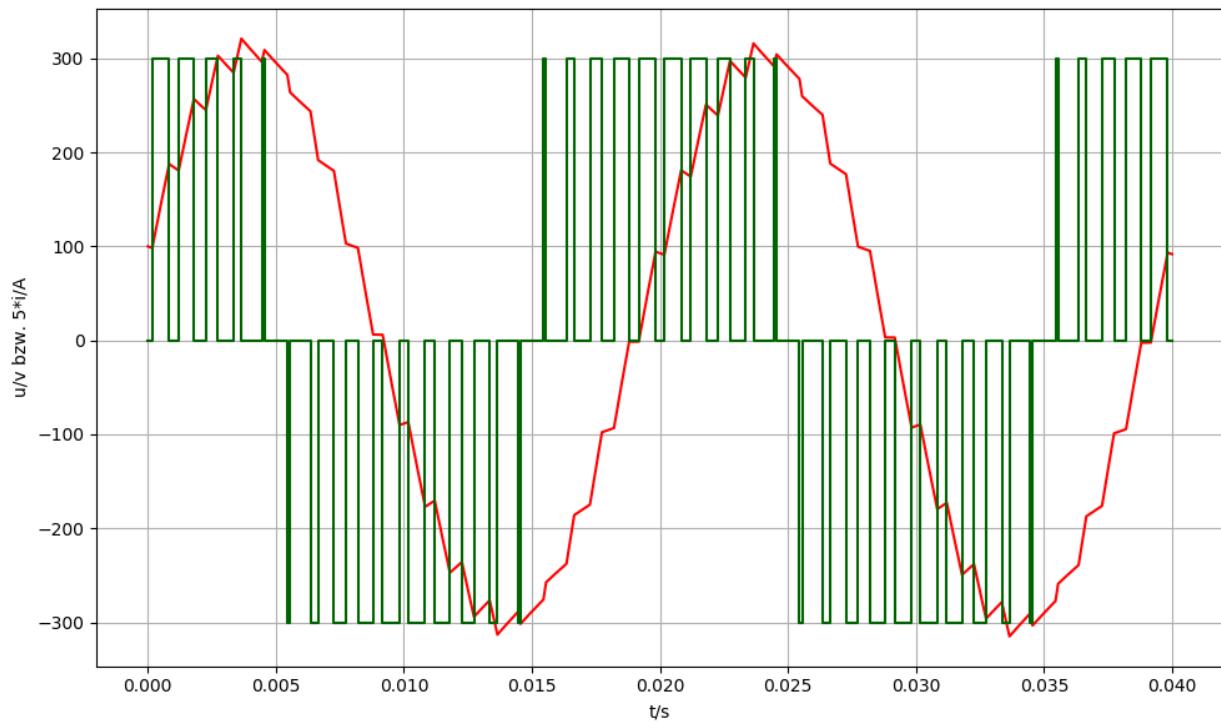
Tipp: Das Python Skript aus Aufgabe 1 können Sie fast identisch übernehmen, lediglich die zusätzliche Bibliothek muß eingebunden werden:

```
mod=ModelicaSystem(os.getcwd()  
+' / '+modelname+'.mo',modelname,[os.getcwd()+' /  
ModSimBib/package.mo'])
```

**Aufgabe 3:** Analysieren Sie die Dateien A3.py und A3.mo und das Modell des Vierquadrantenstellers in der ModSimBib.

----- Selbständige Weiterarbeit-----

**Aufgabe 4:** Simulieren Sie den Vierquadrantensteller mit Python/Modelica und stellen Sie Ihre Ergebnisse wie folgt dar (siehe Kapitel „Vierquadrantensteller“):



# Modellbildung und Simulation

## Parameteridentifikation

### Lernziele

Am Ende des Kapitels ...

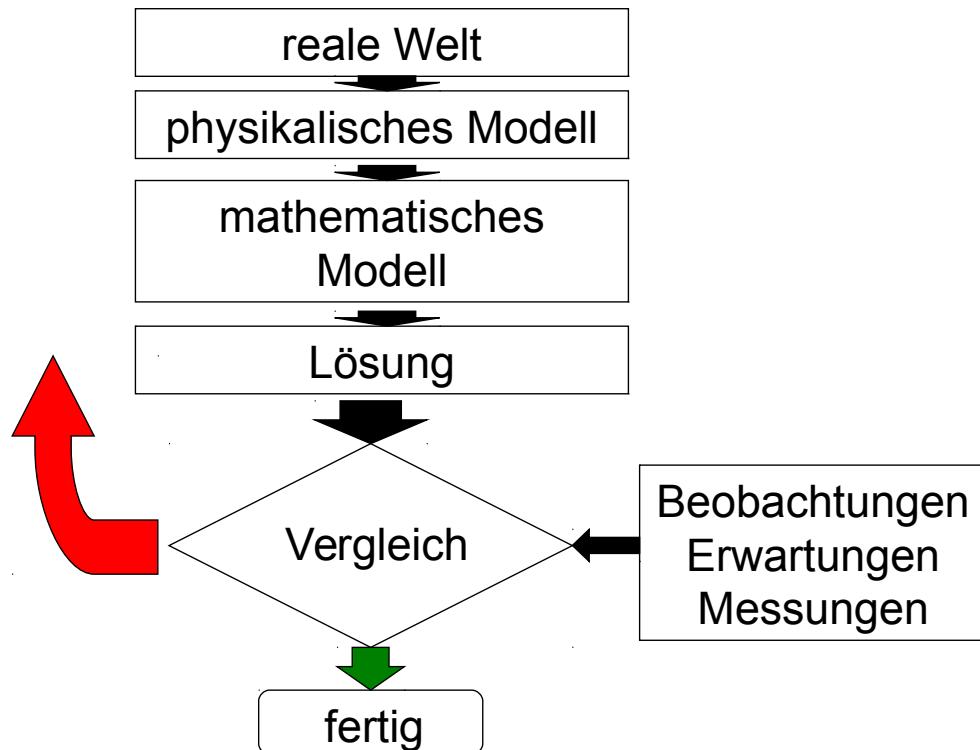
- ✓ ... wissen Sie, was man unter Parameteridentifikation versteht
- ✓ ... haben Sie das Prinzip der kleinsten Fehlerquadrate verstanden
- ✓ ... haben Sie das Gradientenverfahren kennengelernt
- ✓ ... haben Sie das Plotten zweidimensionaler Funktionen geübt

# Lernziele

Am Ende des Kapitels ...

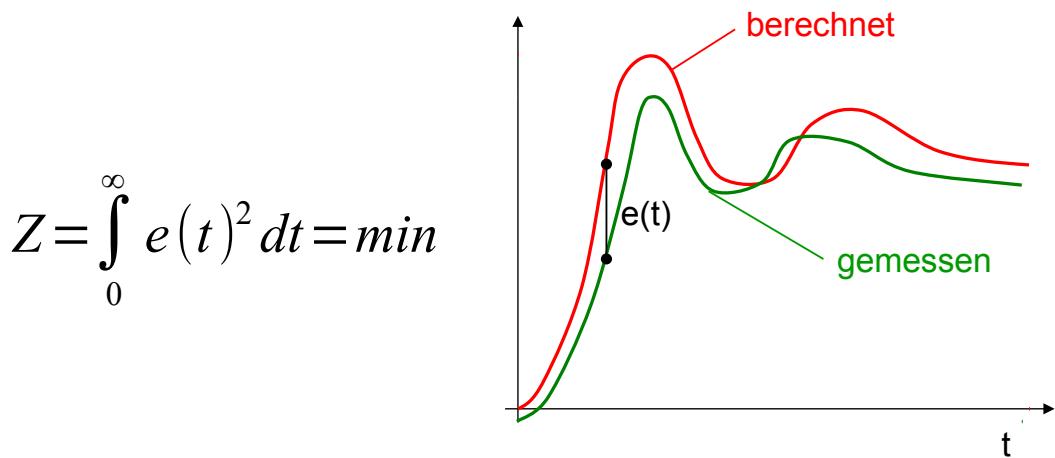
- ✓ ... kennen Sie die Schwierigkeit, das globale Minimum zu erkennen
- ✓ ... wissen Sie, dass die Parameteridentifikation in der Praxis häufig verwendet wird um schwer messbare Größen zu bestimmen
- ✓ ... haben Sie das `optimize`-Modul von SciPy kennen gelernt

## Iterationsschleife bei der Modellbildung



# Parameteridentifikation

Methode der kleinsten Fehlerquadrate



## Extremwertbestimmung

Vektor der Einflußgrößen:  $\vec{x} = [x_0, x_1, \dots, x_n]$

Zielfunktion:  $Z(\vec{x}) = \min$

Restriktionen:  $R_j(\vec{x}) \leq 0$

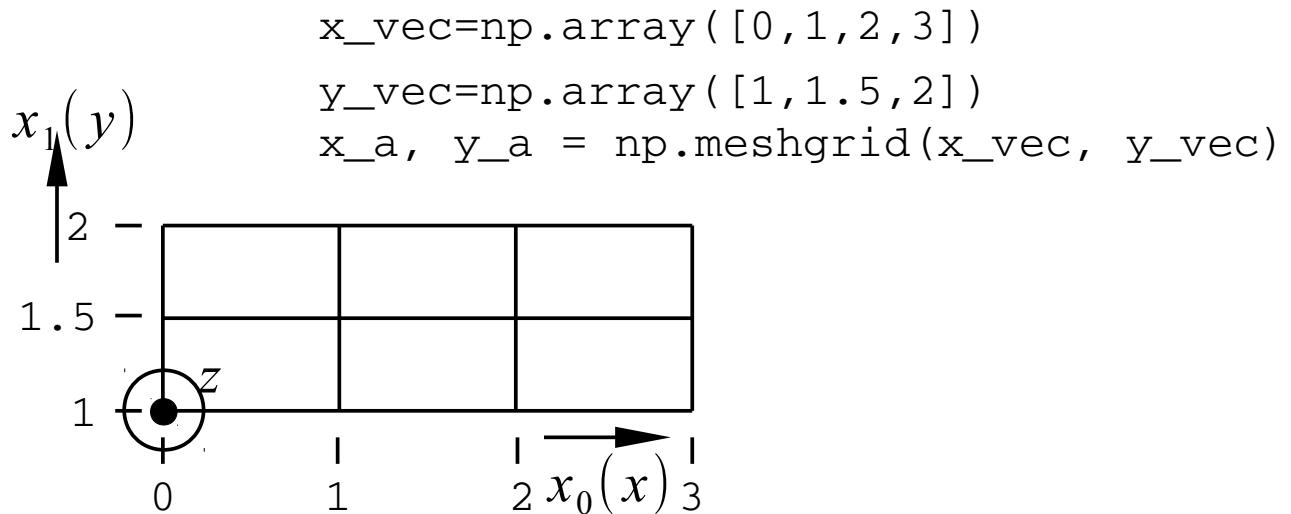
Gesucht wird der Vektor  $\underline{x}$ , der  $Z(\underline{x})$  zum globalen Minimum führt.

Anschauung bei:  $\mathbb{R}^2 \rightarrow \mathbb{R}$

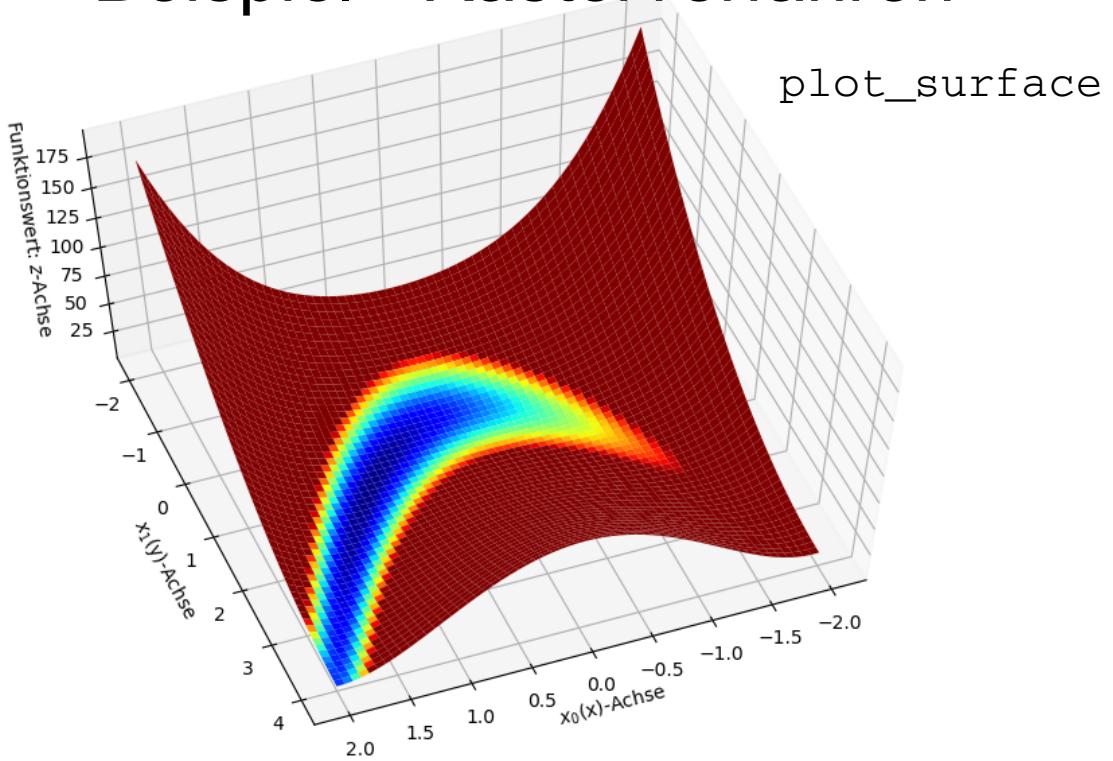
Skifahrer sucht das Tal

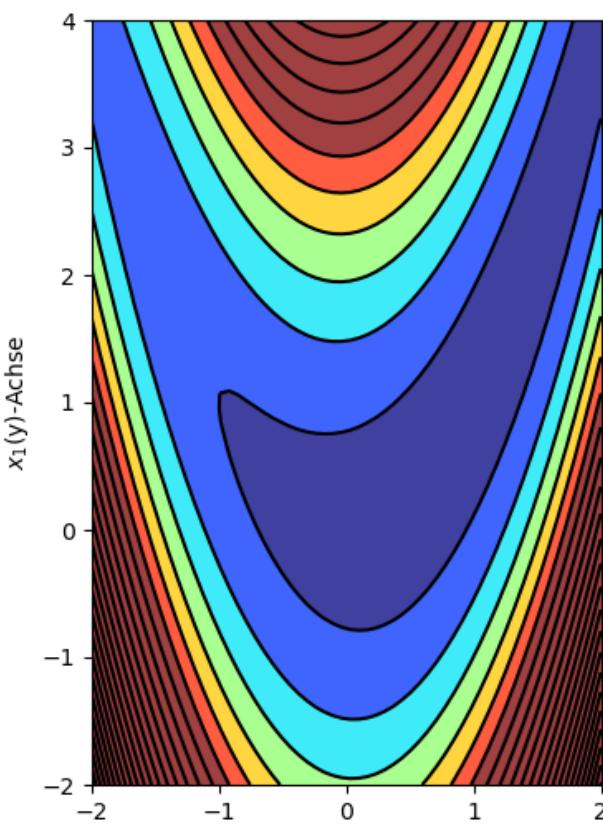
# Auswertung von Funktionen mit zwei Einflußgrößen

Rosenbrock - Funktion  $Z = 5x_0^4 - 10x_0^2x_1 + x_0^2 + 5x_1^2 - 2x_0 + 5$



## Beispiel - Rasterverfahren

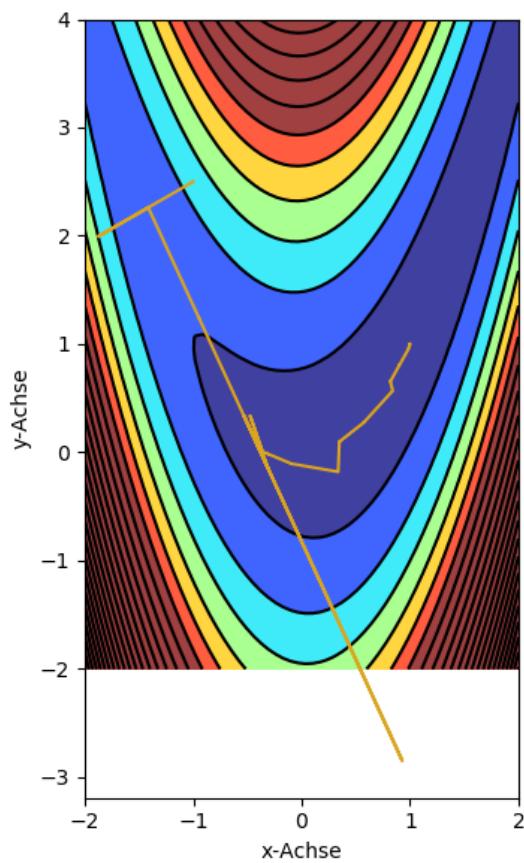




Niveaulinien

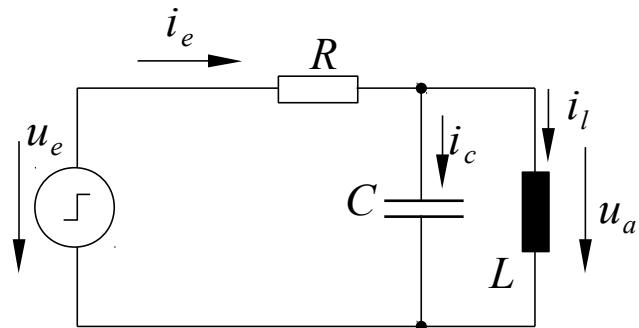
contour

contourfc



Minimum bei  $(1, 1)$ .  
Die Funktionsaufrufe  
lassen den Suchweg  
erkennen.

# Zurück zur Parameteridentifikation: Beispiel: RLC-Schaltung

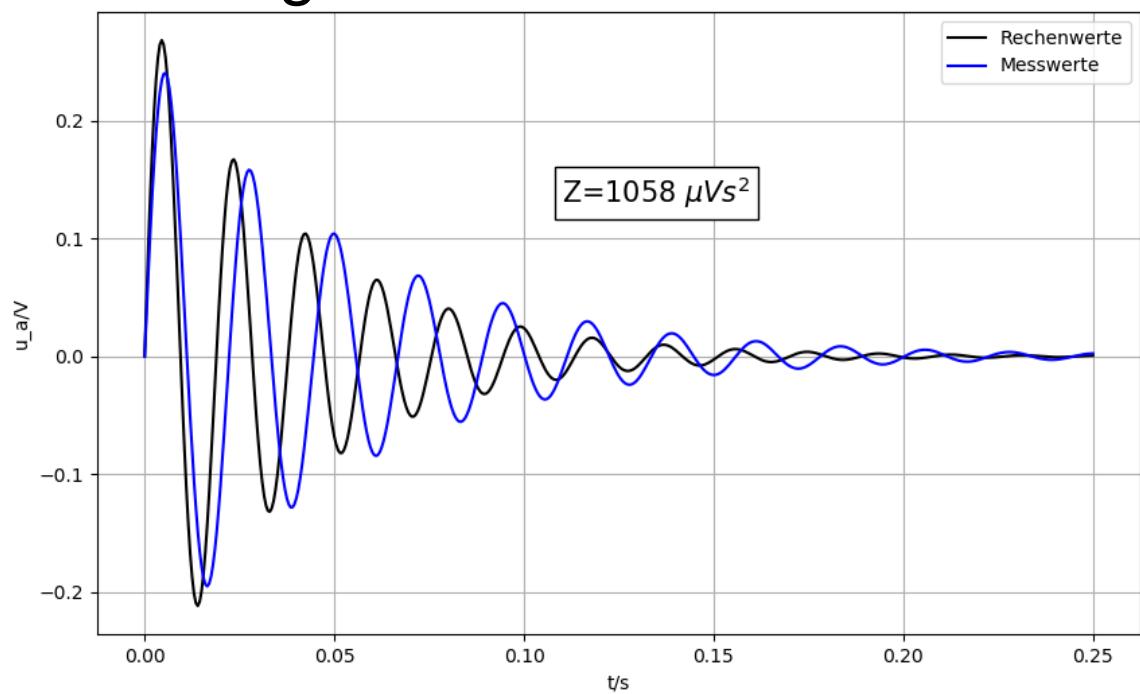


Die genauen Werte von L und C sollen durch Messung der Sprungantwort bestimmt werden

$$u_e = 2 \text{ V} \quad R = 20 \Omega$$

$$L \approx 9 \text{ mH} \quad C \approx 1000 \mu \text{F}$$

## Vergleich zwischen Simulation und Messung

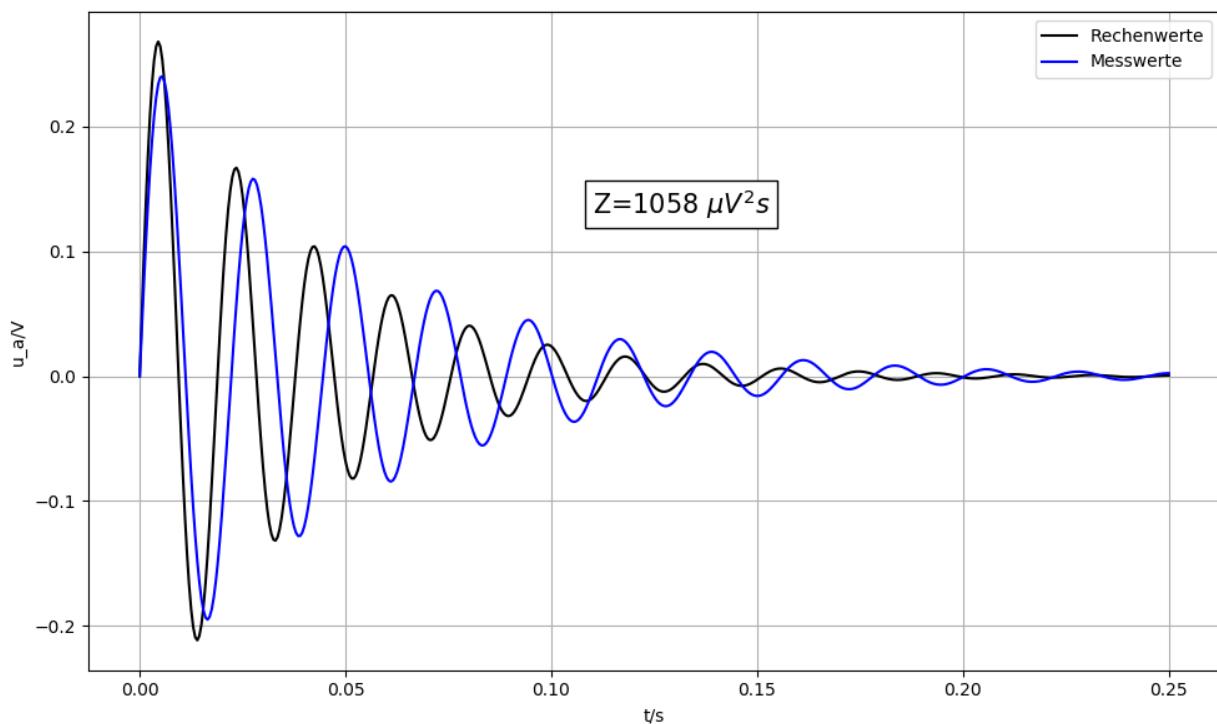


# Optimierung mit Python

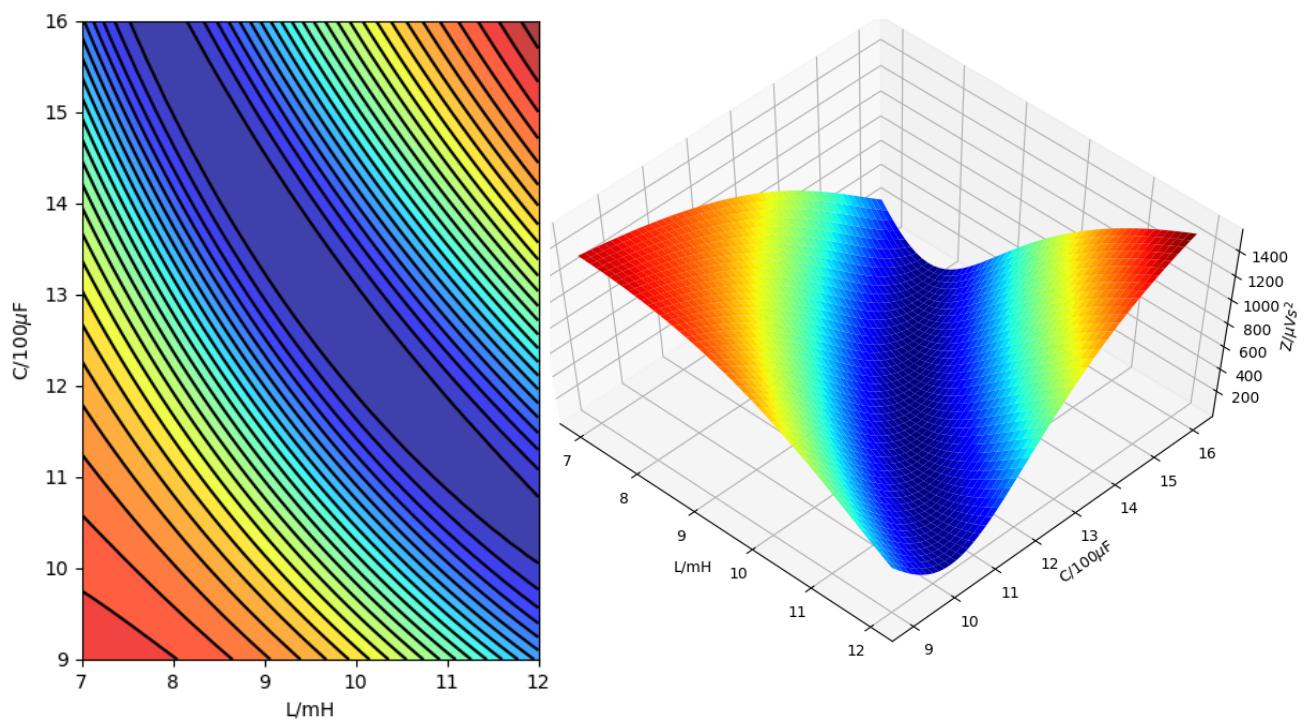
- Die numerische Minimalwertsuche einer mehrdimensionalen Funktion stellt eine anspruchsvolle mathematische Aufgabe dar.
- SciPy stellt leistungsfähige Funktionen zur Minimalwertsuche zur Verfügung. Für das vorliegende Beispiel ist die Scilabfunktion `optimize.minimize` besonders geeignet.

Die Parameter der bereits bekannten RLC-Schaltung sollen bestimmt werden. Dazu wird die Schaltung an eine Spannungsquelle von 2 V gelegt und so die Sprungantwort  $u_a(t)$  gemessen. Von den Nennwerten der Bauelemente  $R=20\Omega$ ,  $L=9\text{mH}$ ,  $C=1000\mu\text{F}$  konnte der Widerstand vorab mit einem Widerstandsmessgerät bestätigt werden.  $L$  und  $C$  haben aufgrund der Toleranz der Bauelemente jedoch andere Werte als die Nenndaten. Die korrekten Werte für  $L$  und  $C$  sollen durch eine Parameteridentifikation bestimmt werden.

**Aufgabe 1:** Stellen Sie einen Vergleich zwischen den Messwerten und der Berechnung mit den Nennwerten dar. Wie groß ist das Zielfunktional  $Z$  im Zeitbereich (quadratische Fehlersumme)?  
Erstellen Sie die dazu die Dateien `A1.py` ausgehend von der gegebenen Datei `A1_vorlage.py` und erweitern Sie diese. Stellen Sie Ihre Ergebnisse wie folgt dar.



**Aufgabe 2:** Programmieren Sie das Rasterverfahren um einen grafischen Eindruck vom Verlauf des Zielfunktionalen zu bekommen ( $7\text{mH} < L < 12\text{ mH}$  ;  $9\cdot100\mu\text{F} < C < 16\cdot100\mu\text{F}$ ).  
 $1\text{mH}$  und  $100\mu\text{F}$  sollen mit gleich vielen Bildpunkten dargestellt werden.  
Stellen Sie Ihre Ergebnisse wie folgt dar.:



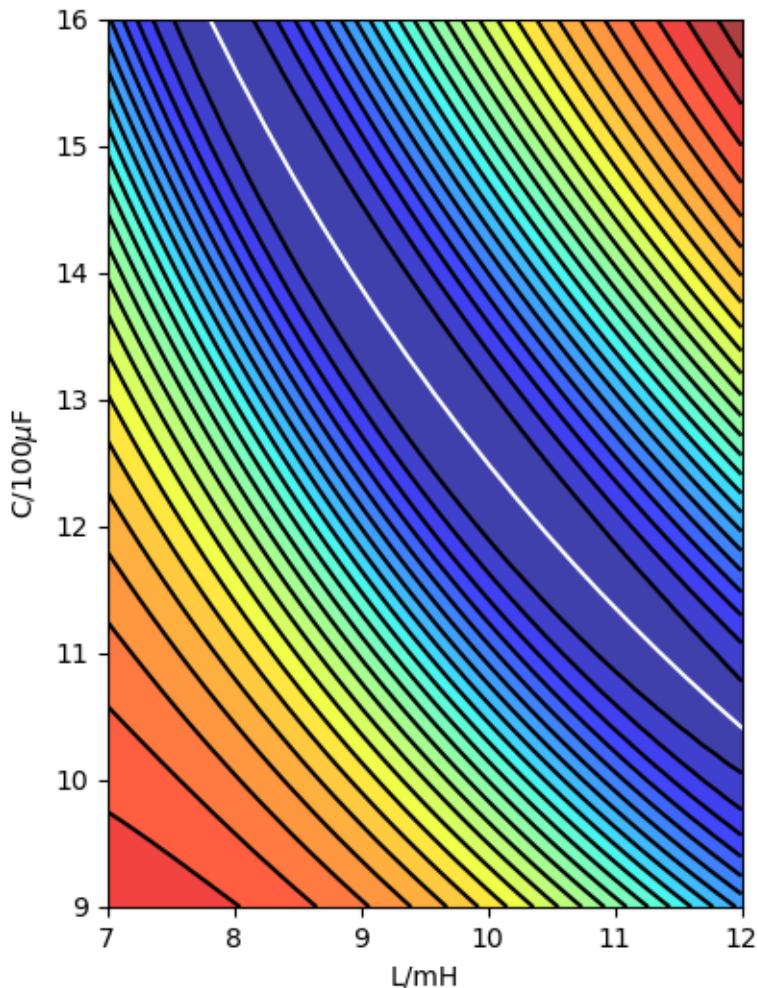
**Tipp 1:** Analysieren Sie zunächst die Datei: `rosenbrock.py`!

**Tipp 2:** Leider gestattet es die Funktion `signal.TransferFunction` nicht, dass man Zähler- und Nennerterm als mehrdimensionale arrays vorgibt. Sie müssen die Funktion zur Berechnung des Zielfunktionalen folglich mit zwei for-Schleifen aufbauen (siehe `rosenbrock.py`!).

**Aufgabe 3:** Berechnen Sie die gesuchten Parameter. Erweitern Sie dazu die Lösung Ihrer Aufgabe 2.

----- Selbständige Weiterarbeit -----

**Aufgabe 4:** Überlegen Sie, wie das stark ausgeprägte Tal zustande kommt und stellen Sie analytisch eine Funktion für den Verlauf des Tales auf. Markieren Sie den Verlauf des Tales wie in folgender Abbildung:



# Literaturverzeichnis

- Kral, C.: 'Modelica – Objektorientierte Modellbildung von Drehfeldmaschinen', Hanser Verlag
- Otter, M.; Elmquist, H.; Mattsson, S.: "Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle", CASD'99, Hawaii, USA, 1999
- Beater, P.: "Regelungstechnik und Simulationstechnik mit Scilab und Modelica", Verlag Books on Demand
- Feldmann: "Repetitorium der Ingenieurmathematik – Teil 2 – Numerische Mathematik", Binomi Verlag
- Preuß, Wenisch: "Lehr- und Übungsbuch Numerische Mathematik", Fachbuchverlag Leipzig

## Online:

- <http://scipy-lectures.org/>
- <http://book.xogeny.com/>