Der Python-Kurs für Ingenieure und Naturwissenschaftler

Das Praxisbuch für NumPy, SciPy und Matplotlib

DIE LESEPROBE

Kapitel 1
Einführung

Dieses Kapitel gibt Ihnen einen kurzen Überblick über die Erweiterungsmöglichkeiten, Einsatzgebiete und die Funktionalität der Programmiersprache Python.

Wenn Sie für Ihre wissenschaftliche Arbeit umfangreiche Berechnungen durchführen müssen und die Ergebnisse auch grafisch ansprechend präsentieren wollen, dann sollten Sie sich ernsthaft mit Python beschäftigen. Python ist eine Programmiersprache, die über eine ähnliche Funktionalität wie MATLAB verfügt, wenn sie durch entsprechende Module erweitert wird. Außerdem wird Python einschließlich aller Erweiterungsmodule kostenfrei zur Verfügung gestellt. Mit Python können Sie z. B. Gleichungssysteme lösen, Funktionsplots erstellen, differenzieren, integrieren und auch Differenzialgleichungen lösen. Auch das Erstellen grafischer Benutzeroberflächen ist möglich. Für fast jede Problemstellung in den Ingenieur- und Naturwissenschaften gibt es Lösungsangebote, die nicht nur ein breites Anwendungsgebiet abdecken, sondern zusätzlich auch noch durch Benutzerfreundlichkeit und Leistungsfähigkeit überzeugen.

Die Programmiersprache Python wurde Anfang der 1990er-Jahre von Guido van Rossum am *Centrum Wiskunde & Informatica* in Amsterdam entwickelt. Die Namensgebung hat nichts mit der gleichnamigen Schlange Python zu tun, sondern bezog sich auf die britische Komikergruppe Monty Python.

Die besonderen Vorteile und Leistungsmerkmale der Programmiersprache sind folgende:

- Python ist eine leicht zu erlernende und leistungsfähige Programmiersprache.
- Sie stellt effiziente Datenstrukturen bereit.
- Sie erlaubt auch objektorientierte Programmierung.
- Sie hat eine übersichtliche Syntax und eine dynamische Typisierung.
- Python-Programme werden mit einem Interpreter übersetzt und eignen sich deshalb für eine schnelle Entwicklung von Prototypen.
- Python steht für Linux, macOS und Windows zur Verfügung.
- Python kann durch Module erweitert werden.

Das Modulkonzept ist der Grundpfeiler und eine der herausragenden Stärken von Python. Ein Modul ist ein Baustein eines Softwaresystems, das eine funktional in sich abgeschlossene Einheit bildet und einen bestimmten Dienst bereitstellt. Für ein abgrenzbares wissenschaftliches Problem wird jeweils ein Modul zur Verfügung gestellt, das genau auf diese Problemstellung zugeschnitten ist. In diesem Buch stelle ich Ihnen die fünf Module NumPy, Matplotlib, SymPy, SciPy und VPython vor.

1.1 Entwicklungsumgebungen

Eine Entwicklungsumgebung ist ein Softwareprogramm, das aus einem Texteditor, einem Debugger und einem Interpreter besteht. Der Texteditor einer Entwicklungsumgebung unterstützt den Programmierer beim Schreiben von Programmen, z. B. durch Syntaxhervorhebung, automatisches Einrücken des Quelltextes usw. Der Debugger hilft dem Programmierer bei der Fehlersuche, und der Interpreter führt die Anweisungen des Programms aus. Von den vielen Entwicklungsumgebungen, mit denen Python-Programme entwickelt werden können, sollen hier nur die Entwicklungsumgebungen IDLE, Thonny und Spyder kurz vorgestellt werden.

1.1.1 IDLE

Die Abkürzung IDLE steht für **I**ntegrated **D**evelopment and **L**earning **E**nvironment. Abbildung 1.1 zeigt die Bedienoberfläche von IDLE.

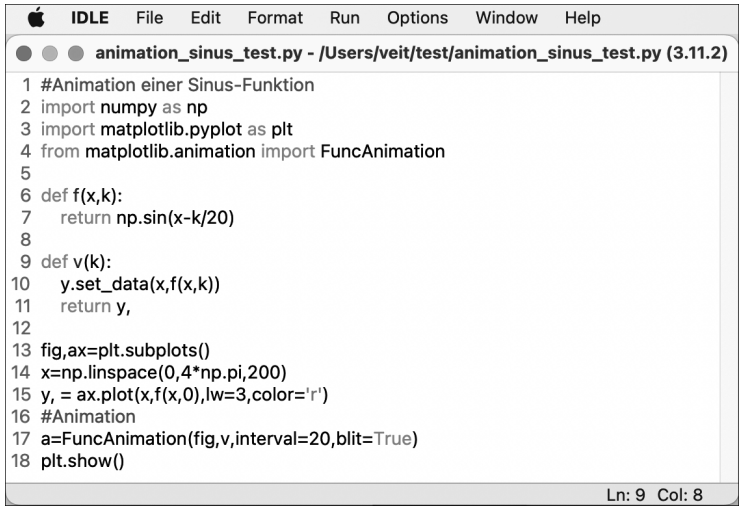

Abbildung 1.1 Die Entwicklungsumgebung IDLE

Die IDLE ist Bestandteil des Python-Downloads. Mit der Installation von Python wird sie gleichzeitig zusammen mit dem Paketmanager Pip installiert. Unter der URL

https://www.python.org/downloads/ können Sie die aktuelle Version von Python für die Betriebssysteme Linux, macOS und Windows herunterladen. Die einzelnen Module NumPy, Matplotlib, SymPy, SciPy und VPython müssen Sie mit dem Paketmanager Pip installieren (siehe Abschnitt 1.1.4). Dabei kann es zu Problemen kommen, wenn Sie eine neue Python-Version installieren: Die Module können mit der neuen IDLE-Version nicht mehr importiert werden und die Programme werden nicht mehr ausgeführt. Wie Sie dieses Problem eventuell beheben können, zeige ich Ihnen in Abschnitt 1.1.4. Falls Sie mit der Installation der Python-Module scheitern sollten, empfehle ich Ihnen die Entwicklungsumgebung Thonny.

Wenn Sie auf RUN • PYTHON SHELL klicken, öffnet sich die Python-Shell. Hinter dem Eingabeprompt >>> können Sie direkt Python-Befehle oder mathematische Ausdrücke eingeben, z. B. `2+3`, `3*5` oder `7/5`. Jede Eingabe müssen Sie mit der ↵-Taste abschließen.

1.1.2 Thonny

Thonny ist zwar, gemessen an den professionellen Angeboten, eine recht einfach gestaltete Entwicklungsumgebung mit einem vergleichsweise geringen Funktionsumfang, sie ist aber aufgrund der leichten Bedienbarkeit besonders für Programmieranfänger geeignet. Mit Thonny können Sie alle hier im Buch besprochenen Programmbeispiele ausführen und testen. Abbildung 1.2 zeigt die Bedienoberfläche.

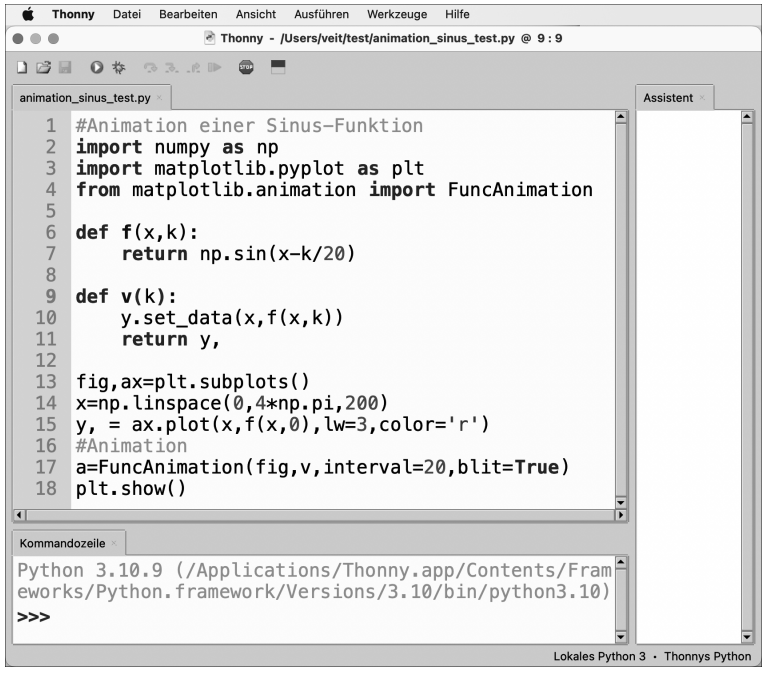

Abbildung 1.2 Die Entwicklungsumgebung Thonny

Thonny steht unter der URL *https://thonny.org* für die Betriebssysteme Linux, macOS und Windows als Download zur Verfügung.

Der Quelltext des Programms wird in den Texteditor (linker oberer Bereich) eingegeben. Nachdem das Programm mit der Funktionstaste F5 oder mit einem Mausklick auf den START-Button gestartet wurde, erscheint ein Dialogfenster, in das der Dateiname des Programms eingegeben werden muss. Das Ergebnis von numerischen Berechnungen wird dann in dem Fenster KOMANDOZEILE links unten in der Python-Shell ausgegeben. Funktionsplots von Matplotlib-Programmen werden in einem separaten Fenster ausgegeben. In der Shell, auch Python-Konsole genannt, können Sie auch direkt Python-Befehle eingeben. Der ASSISTENT auf der rechten Seite des Hauptfensters unterstützt Sie bei der Fehlersuche. Allerdings sollten Sie diesbezüglich keine zu hohen Erwartungen haben.

Ein besonders wichtiges Feature von Thonny ist, dass Sie die Module NumPy, Matplotlib, SymPy, SciPy und VPython einfach nachinstallieren und aktualisieren können. Dazu brauchen Sie nur den Dialog WERKZEUGE • VERWALTE PAKETE zu öffnen (siehe Abbildung 1.3). Dann geben Sie oben links im Textfeld den Namen des zu installierenden Moduls ein und klicken auf INSTALLIEREN oder AKTUALISIEREN.

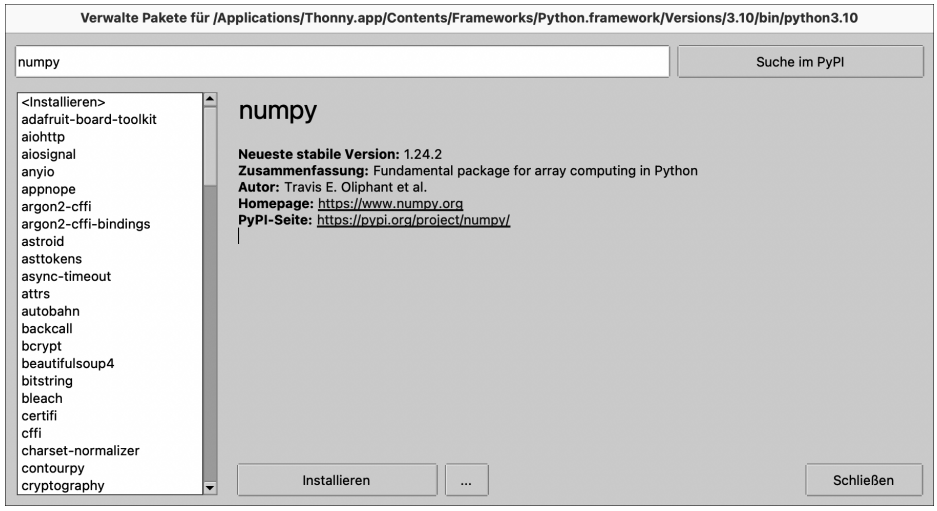

Abbildung 1.3 Installation von Modulen

Wenn Sie ein Modul deinstallieren wollen, dann müssen Sie das entsprechende Modul im linken Fenster auswählen. Dann erscheint rechts neben der Befehlsschaltfläche INSTALLIEREN die Befehlsschaltfläche DEINSTALLIEREN. Ein besonderer Vorteil des Paketmanagers von Thonny besteht darin, dass Sie auch ältere Versionen aller verfügbaren Module testen können. Dazu müssen Sie nur auf die Befehlsschaltfläche ... direkt rechts neben der Befehlsschaltfläche INSTALLIEREN klicken, und es öffnet sich ein Fenster, in dem Sie die gewünschte Version des Moduls auswählen können.

1.1.3 Spyder

Spyder ist die Entwicklungsumgebung der Anaconda-Distribution. Bis auf VPython sind die in diesem Buch behandelten Module NumPy, Matplotlib, SymPy und SciPy schon eingebaut.

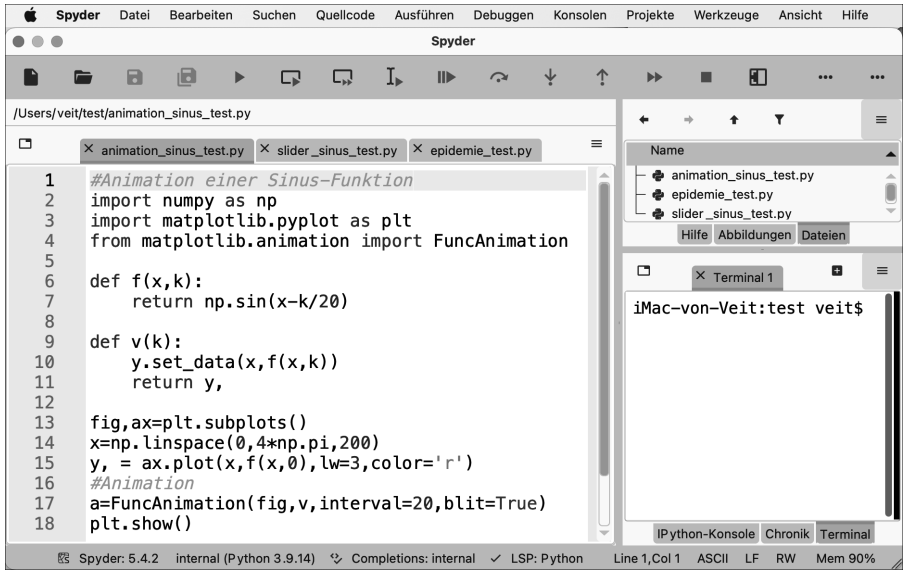

Abbildung 1.4 Die Entwicklungsumgebung Spyder

Spyder steht unter der URL *https://www.spyder-ide.org* für Linux, macOS und Windows als kostenfreier Download zur Verfügung.

Wenn Sie eine Animation mit einem Matplotlib-Programm ausführen wollen, dann müssen Sie in den Einstellungen unter IPython-Konsole • Grafik als Backend Automatisch auswählen. Nach dem Programmstart erscheint dann ein separates Fenster, in dem die Animation ausgeführt wird. Auch Matplotlib-Programme, die Slider-Steuerelemente enthalten, können nur mit dieser Option interaktiv ausgeführt werden.

Spyder ist zwar eine sehr leistungsfähige Entwicklungsumgebung. Sie hat allerdings den Nachteil, dass die Nachinstallation von standardmäßig nicht mitinstallierten Modulen, wie z. B. VPython, für Anfänger nicht leicht zu handhaben ist. Nähere Hinweise zur Installation von Python-Modulen finden Sie in der Dokumentation zu Spyder (*https://www.spyder-ide.org*).

1.1.4 Pip

Wenn Sie andere Entwicklungsumgebungen als Thonny oder Spyder benutzen wollen, dann können Sie Python-Module mit Pip installieren. Pip ist keine Entwicklungs-

umgebung, sondern der Paketmanager von Python, der Module aus dem *Python Package Index* (*https://pypi.org/*) installiert. Über ihn können Sie sehr komfortabel Module herunterladen und aktualisieren – für die Arbeit mit Python ist Pip ein sehr wichtiges Werkzeug.

Wenn Sie Python installiert haben und beispielsweise nur das Modul NumPy hinzufügen wollen, nutzen Sie folgenden Befehl, den Sie in einem Terminal unter Windows, Linux oder macOS eingeben können:

```
pip install numpy
```

Eine bestehende NumPy-Installation aktualisieren Sie mit diesem Kommando:

```
pip install --upgrade numpy
```

Falls Sie IDLE (z. B. Version 3.9) nutzen und eine neue Version von Python (z. B. 3.11) installieren, dann werden die zuvor installierten Python-Module in die aktualisierte Version nicht mehr importiert. In diesem Fall versuchen Sie die Installation mit `pip3.11 install numpy`.

Wenn Ihnen die Installation oder Aktualisierung der Python-Module nicht gelingen sollte, empfehle ich Ihnen die Entwicklungsumgebung Thonny. Weitere Informationen zum Einsatz von Pip finden Sie auf der Seite *https://pypi.org/project/pip*.

1.2 Die Module von Python

Damit Sie sich einen ersten Überblick über die Möglichkeiten und Leistungsmerkmale des Modulkonzepts von Python verschaffen können, beschreibe ich die fünf Module zunächst einmal schlagwortartig. Statt *Modul* ist auch die Bezeichnung *Bibliothek* oder *Softwarebibliothek* üblich. Die Fähigkeiten von Python verdeutlicht man am besten anhand kurzer Programmbeispiele. Sie müssen selbstverständlich die hier gezeigten Quelltexte noch nicht verstehen. Dazu dienen ja die nachfolgenden Kapitel.

1.2.1 NumPy

Mit dem Modul NumPy (**num**erisches **Py**thon) können Sie umfangreiche numerische Berechnungen durchführen, z. B. lineare Gleichungssysteme lösen – auch mit komplexen Zahlen. Listing 1.1 zeigt ein Beispiel zur Vektorrechnung:

```
01  import numpy as np
02  A=np.array([1, 2, 3])
```

```
03  B=np.array([4, 5, 6])
04  print("Vektor      A:",A)
05  print("Vektor      B:",B)
06  print("Summe     A+B:",A+B)
07  print("Produkt   A*B:",A*B)
08  print("Kreuzprodukt :",np.cross(A,B))
09  print("Skalarprodukt:",np.dot(A,B))
```

Listing 1.1 NumPy-Programm

Ausgabe

```
Vektor      A: [1 2 3]
Vektor      B: [4 5 6]
Summe     A+B: [5 7 9]
Produkt   A*B: [ 4 10 18]
Kreuzprodukt : [-3  6 -3]
Skalarprodukt: 32
```

Das Modul NumPy wird in Kapitel 3, »Numerische Berechnungen mit NumPy«, behandelt.

1.2.2 Matplotlib

Mit dem Modul Matplotlib können Sie mathematische Funktionen, Histogramme und viele andere Diagrammtypen darstellen sowie physikalische Vorgänge simulieren und animieren. Die grafischen Gestaltungsmöglichkeiten sind sehr vielfältig und detailreich. Listing 1.2 zeigt ein einfaches Beispiel für den Funktionsplot eines Polynoms:

```
01  import numpy as np
02  import matplotlib.pyplot as plt
03  x=np.arange(-2,6,0.01)
04  y=x**3-7*x**2+7*x+15
05  plt.plot(x,y)
06  plt.show()
```

Listing 1.2 Funktionsplot mit Matplotlib

Ausgabe

In Abbildung 1.5 sehen Sie die Ausgabe des Funktionsplots.

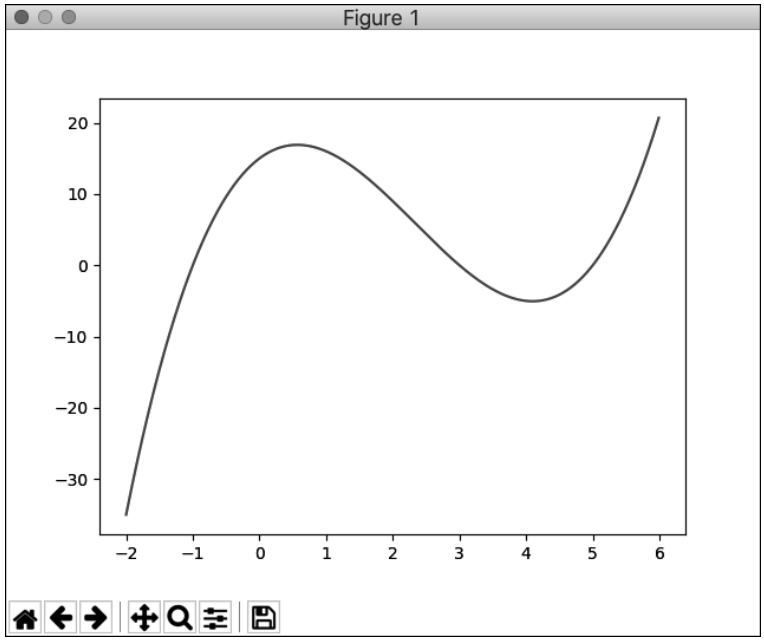

Abbildung 1.5 Ein mit Matplotlib erstellter Funktionsplot

Das Modul Matplotlib wird in Kapitel 4, »Funktionsdarstellungen und Animationen mit Matplotlib«, behandelt.

1.2.3 SymPy

Mit SymPy (**sym**bolisches **Py**thon) können Sie Integrale oder Ableitungen symbolisch berechnen oder Differenzialgleichungen symbolisch lösen. Auch die Vereinfachung mathematischer Terme ist möglich (und vieles mehr). Listing 1.3 zeigt ein einfaches Beispiel für die symbolische Differenziation und Integration:

```
01  from sympy import *
02  x=symbols("x")
03  y=x**3-7*x**2+7*x+15
04  y_1=diff(y,x,1)
05  y_2=diff(y,x,2)
06  y_3=diff(y,x,3)
07  Y=integrate(y,x)
08  print("1. Ableitung:",y_1)
09  print("2. Ableitung:",y_2)
10  print("3. Ableitung:",y_3)
11  print("   Integral :",Y)
```

Listing 1.3 Symbolisches Differenzieren und Integrieren mit SymPy

Ausgabe

```
1. Ableitung: 3*x**2 - 14*x + 7
2. Ableitung: 2*(3*x - 7)
3. Ableitung: 6
   Integral : x**4/4 - 7*x**3/3 + 7*x**2/2 + 15*x
```

Das Modul SymPy wird in Kapitel 5, »Symbolisches Rechnen mit SymPy«, behandelt.

1.2.4 SciPy

Mit SciPy (**sci**entific **Py**thon) können Sie numerisch differenzieren, integrieren und Systeme von Differenzialgleichungen numerisch lösen. SciPy ist ebenso umfangreich wie vielseitig. Die Möglichkeiten von SciPy können in diesem Buch nur ansatzweise beschrieben werden. Listing 1.4 zeigt ein einfaches Beispiel zur numerischen Integration:

```
01  import scipy.integrate as integral
02  def f(x):
03      return x**2
04  A=integral.quad(f,0,5)
05  print("Flächeninhalt A=",A[0])
```

Listing 1.4 Numerisches Integrieren mit SciPy

Ausgabe

```
Flächeninhalt A= 41.66666666666666
```

Das Modul SciPy wird in Kapitel 6, »Numerische Berechnungen und Simulationen mit SciPy«, behandelt.

1.2.5 VPython

Mit VPython können Sie Körper in einer 3D-Ansicht darstellen oder auch deren Bewegungen im 3D-Raum animieren. Ab Version 7 werden die Animationen nach dem Programmstart im Standardbrowser dargestellt. Listing 1.5 zeigt ein Beispiel, wie die Animation eines springenden Balls programmiert werden kann:

```
01  from vpython import *
02  r=1. #Radius
03  h=5. #Höhe
04  scene.background=color.white
05  scene.center=vector(0,h,0)
06  box(pos=vector(0,0,0),size=vector(2*h,r/2,h), color=color.green)
```

```
ball = sphere(radius=r, color=color.yellow)
ball.pos=vector(0,2*h,0) #Fallhöhe
ball.v = vector(0,0,0)   #Anfangsgeschwindigkeit
g=9.81
dt = 0.01
while True:
    rate(100)
    ball.pos = ball.pos + ball.v*dt
    if ball.pos.y < r:
        ball.v.y = -ball.v.y
    else:
        ball.v.y = ball.v.y - g*dt
```

Listing 1.5 Animation eines springenden Balls

Ausgabe

In Abbildung 1.6 sehen Sie eine Momentaufnahme der Animation.

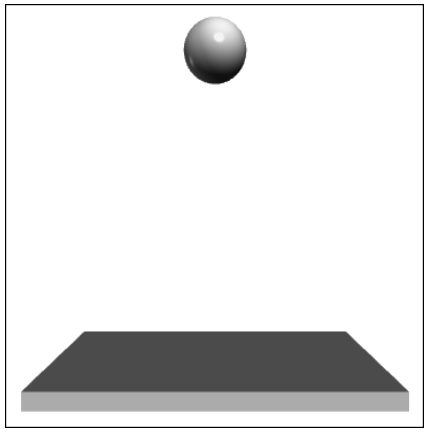

Abbildung 1.6 Eine mit VPython erstellte Animation (springender Ball)

Das Modul VPython wird in Kapitel 7, »3D-Grafik und Animationen mit VPython«, behandelt.

In diesem Buch können natürlich nicht alle Möglichkeiten der aufgezählten Python-Module erschöpfend behandelt werden. Sollten Sie ein bestimmtes Thema vermissen, empfehle ich Ihnen die Onlinedokumentation als ergänzende Informationsquelle. Auf den Internetseiten der Modulbetreuer finden Sie für jedes Modul Tutorials für den Einstieg und die Dokumentationen der vollständigen Modulbeschreibungen.

In den auf Kapitel 7 folgenden Kapiteln werden weitere Einsatzmöglichkeiten der Module vertieft. Im Fokus stehen hier mehr die praktischen Anwendungsmöglichkeiten.

In Kapitel 8, »Rechnen mit komplexen Zahlen«, zeige ich Ihnen, wie mit der symbolischen Methode elektrische Wechselstromnetzwerke berechnet werden. In der Projektaufgabe lernen Sie, wie ein elektrisches Energieübertragungssystem dimensioniert wird.

In Kapitel 9, »Statistische Berechnungen«, geht es hauptsächlich um die Simulation einer Qualitätsregelkarte. Sie lernen, wie Sie normalverteilte Zufallszahlen erzeugen und in einer Datei abspeichern. Diese Daten werden wieder ausgelesen, um ihre statistischen Kennwerte, den arithmetischen Mittelwert und die Standardabweichung zu berechnen.

In Kapitel 10, »Boolesche Algebra«, wird gezeigt, wie Sie Wahrheitstabellen aufstellen und komplexe logische Schaltungen mit SymPy vereinfachen können.

In Kapitel 11, »Interaktive Programmierung mit Tkinter«, lernen Sie, wie Sie mit Python grafische Benutzeroberflächen programmieren können. Die Projektaufgabe zeigt, wie Sie einfache Regelkreise simulieren können.

1.3 Die Schlüsselwörter von Python

Wenn Sie eine Programmiersprache erlernen wollen, dann müssen Sie als Erstes wissen, welche *Schlüsselwörter* in dieser Programmiersprache definiert sind. Schlüsselwörter sind die reservierten Wörter einer Programmiersprache. Sie haben in der Definition der Programmiersprache eine bestimmte Bedeutung und dürfen deshalb nicht als Variablennamen in einem Programm benutzt werden. Python hat 35 Schlüsselwörter. Wenn Sie in der Python-Konsole folgende Befehle eingeben

```
>>> import keyword
>>> a=keyword.kwlist
>>> print(a)
```

dann erhalten Sie die Ausgabe:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

und mit der Anweisung

```
>>> print(len(a))
35
```

wird die Anzahl der Schlüsselwörter ausgeben.

Sie brauchen sich nicht gleich zu Beginn alle Schlüsselwörter zu merken. Um sich einen besseren Überblick über die Schlüsselwörter von Python zu verschaffen, ist es

sinnvoll, sie zunächst einmal nach ihrer Funktionalität zu ordnen. Tabelle 1.1 gibt Ihnen eine nach funktionalen Kriterien geordnete Übersicht der wichtigsten Schlüsselwörter.

Bedingte Anweisungen	Schleifen	Klassen, Module, Funktionen	Fehler-behandlung
`if`	`for`	`class`	`try`
`else`	`in`	`def`	`exept`
`elif`	`while`	`global`	`finally`
`not`	`break`	`lambda`	`raise`
`or`	`as`	`nonlocal`	`assert`
`and`	`continue`	`yield`	`with`
`is`		`import`	
`True`		`return`	
`False`		`from`	
`None`			

Tabelle 1.1 Übersicht über die wichtigsten Schlüsselwörter von Python (Auszug)

Mit einigen wenigen Schlüsselwörtern wie `if`, `else`, `for` und `while` können Sie zusammen mit der eingebauten Python-Fuktion `print()` schon einfache Python-Programme schreiben.

1.4 Ihr Weg durch dieses Buch

Wie sollten Sie dieses Buch lesen? Im Prinzip können Sie die einzelnen Kapitel unabhängig voneinander lesen. Wenn Sie schon die Grundstrukturen von Python kennen, dann können Sie Kapitel 2 überspringen. Ist das nicht der Fall, dann müssen Sie dieses Kapitel zuerst lesen, weil es Voraussetzung für das Verständnis der nachfolgenden Kapitel ist.

Das Konzept der Darstellung und Wissensvermittlung erfolgt nach einem einheitlichen Prinzip: Zu jedem Thema werden ein bis drei Beispiele aus der Elektrotechnik, der Maschinenbautechnik oder der Physik vorgestellt. Nach einer kurzen Beschreibung der Aufgabenstellung wird der vollständige Quelltext abgedruckt. Direkt da-

nach erfolgt die Ausgabe (Ergebnisse der Berechnungen). Anschließend wird der Quelltext besprochen und analysiert.

Zu einer Analyse eines Quelltextes gehört auch die Analyse der Ergebnisse (Ausgabe). Stimmen die Ergebnisse mit den Erwartungen überein? Löst das Programm die an es gestellte Aufgabe überhaupt? Oft versteht man den Quelltext eines Programms erst vollständig, wenn man die Ausgabe genauer betrachtet hat. Nach der Betrachtung der Ausgabe können Sie dann wieder den Quelltext analysieren.

Am Ende eines jeden Kapitels werden eine oder mehrere Projektaufgaben gestellt, besprochen und vollständig gelöst, um das zuvor Gelernte zu vertiefen und zu erweitern.

Kapitel 3
Numerische Berechnungen mit NumPy

In diesem Kapitel lernen Sie, wie Sie mit NumPy Operationen auf Vektoren und Matrizen durchführen sowie lineare Gleichungssysteme lösen können.

Das Akronym NumPy steht für **num**erisches **Py**thon. Dieses Modul stellt, wie der Name schon andeutet, Funktionen für numerische Berechnungen bereit. Neben der Anzahl der zur Verfügung gestellten Funktionen ist vor allem die kurze Laufzeit der NumPy-Funktionen hervorzuheben. Das Modul NumPy sollten Sie immer mit der Importanweisung `import numpy as np` importieren. Die Vergabe des Alias `np` hat sich als Konvention durchgesetzt. NumPy bildet die Basis für fast alle wissenschaftlichen Berechnungen und wird deshalb häufig zusammen mit den Modulen Matplotlib und SciPy genutzt.

3.1 NumPy-Funktionen

Die am häufigsten genutzten NumPy-Funktionen sind `arange()` und `linspace()`. Beide Funktionen erzeugen *eindimensionale* Arrays der Länge *n*. Während der Laufzeit lässt sich *n* nicht mehr ändern. Wenn Sie die Funktion `arange()` wählen, dann werden die Abstände zwischen den Array-Elementen festgelegt; wählen Sie dagegen die Funktion `linspace()`, dann wird die Anzahl der Array-Elemente festlegt. Eine der beiden Funktionen kommt in jedem Matplotlib-Programm für die Erzeugung der unabhängigen Variablen in Wertetabellen vor. Die NumPy-Funktion `array()` erzeugt ein *zweidimensionales* Array, wenn ihr eine verschachtelte Liste als Argument übergeben wird. Des Weiteren stellt NumPy auch noch die trigonometrischen Funktionen, die hyperbolischen, die logarithmischen und wichtige statistische Funktionen zur Verfügung.

3.1.1 Eindimensionale Arrays mit arange() und linspace() erzeugen

Mit den NumPy-Funktionen `arange()` und `linspace()` lassen sich eindimensionale Arrays mit vorgegebener Länge erzeugen. Der Datentyp muss einheitlich sein. Die allgemeine Syntax für `arange()` lautet:

```
np.arange(start,stop,step,dtype=None)
```

Der Datentyp muss nicht vorgegeben werden. Er wird von NumPy automatisch ermittelt. In der Regel werden Zahlen vom Typ Float verarbeitet. Drei unterschiedliche Float-Datentypen sind möglich:

- **Float16**: halbe Genauigkeit mit 10 Bit Mantisse und 5 Bit Exponent
- **Float32**: einfache Genauigkeit mit 23 Bit Mantisse und 8 Bit Exponent
- **Float64**: doppelte Genauigkeit mit 52 Bit Mantisse und 11 Bit Exponent

Die Funktion `linspace()` gibt statt der Schrittweite `step` die Anzahl der Elemente `num` vor. Die Voreinstellung ist 50. Die allgemeine Syntax für `linspace()` lautet:

```
linspace(start,stop,num=50,endpoint=True,retstep=False, dtype=None, axis=0)
```

Listing 3.1 vergleicht beide Funktionen miteinander:

```
01  #01_1dim_array.py
02  import numpy as np
03  x1=list(range(10))
04  x2=np.arange(10)
05  x3=np.arange(1,10,0.5)
06  x4=np.linspace(1,10,10)
07  x5=np.linspace(1,10,10,endpoint=False)
08  print("Python Liste:",type(x1) ,"\n",x1)
09  print("arange() Schrittweite 1:",type(x2),"\n",x2)
10  print("arange() Schrittweite 0.5:",type(x3),"\n",x3)
11  print("linspace() Schrittweite 1:",type(x4),"\n",x4)
12  print("linspace() Schrittweite 0.9:",type(x5),"\n",x5)
```

Listing 3.1 Arrays mit arange() und linspace()

Ausgabe

```
Python Liste: <class 'list'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
arange() Schrittweite 1: <class 'numpy.ndarray'>
[0 1 2 3 4 5 6 7 8 9]
arange() Schrittweite 0.5: <class 'numpy.ndarray'>
[1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. 5.5 6. 6.5 7. 7.5 8. 8.5 9. 9.5]
```

```
linspace() Schrittweite 1: <class 'numpy.ndarray'>
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
linspace() Schrittweite 0.9: <class 'numpy.ndarray'>
[1. 1.9 2.8 3.7 4.6 5.5 6.4 7.3 8.2 9.1]
```

Analyse

Zeile 03 erzeugt die Liste `x1` aus der Python-Funktion `range(10)`. Zeile 08 gibt für `x1` die Zahlen von 0 bis 9 aus. Voreingestellt ist die Schrittweite 1.

Zeile 04 erzeugt ein Array `x2` mit der NumPy-Funktion `arange()`. Zeile 09 gibt ebenfalls für `x2` die Zahlen von 0 bis 9 aus. Voreingestellt ist ebenfalls die Schrittweite von 1.

Zeile 05 erzeugt ein NumPy-Array `x3` mit der Schrittweite `0.5`. Der Endwert wird nicht mit ausgegeben (Zeile 10).

In den Zeilen 06 und 07 werden zwei Arrays mit der NumPy-Funktion `linspace()` erzeugt. Wenn die Eigenschaft `endpoint=False` gesetzt wird, dann wird das letzte Element nicht mit ausgegeben und die Schrittweite beträgt 0,9 (Zeile 12). Die Voreinstellung ist `endpoint=True`.

Die NumPy-Funktionen `arange()` und `linspace()` sind vom Typ `numpy.ndarray`. Das Präfix `nd` steht für mehrdimensionale Arrays (engl. *n-dimensional*).

Laufzeit von arange() und linspace()

Ein besonderer Vorteil von NumPy-Funktionen soll deren *kurze* Laufzeit sein. Listing 3.2 berechnet und vergleicht die Laufzeiten einer Python-Liste mit den Laufzeiten der NumPy-Funktionen `arange()` und `linspace()`. Alle drei Funktionen `version1(n)`, `version2(n)` und `version3(n)` addieren jeweils 1 Million Zahlen aus zwei Arrays elementweise. Die Laufzeit wird mit der Funktion `time()` aus dem Python-Modul `time` ermittelt.

```
01  #02_laufzeitvergleich.py
02  import time as t
03  import numpy as np
04  #Python Liste
05  def version1(n):
06      t1=t.time()
07      x1=list(range(n)) #Liste erzeugen
08      x2=list(range(n))
09      summe=[]
10      for i in range(n):
11          summe.append(x1[i]+x2[i])
12      return t.time() - t1
```

```
#NumPy arange()
def version2(n):
    t1=t.time()
    x1=np.arange(n)
    x2=np.arange(n)
    summe=x1+x2
    return t.time() - t1
#NumPy linspace()
def version3(n):
    t1=t.time()
    x1=np.linspace(0,n,n)
    x2=np.linspace(0,n,n)
    summe=x1+x2
    return t.time() - t1

nt=1000000
laufzeit1=version1(nt)
laufzeit2=version2(nt)
laufzeit3=version3(nt)
faktor1=laufzeit1/laufzeit2
faktor2=laufzeit1/laufzeit3
#Ausgabe
print("Laufzeit für Python range()...:",laufzeit1)
print("Laufzeit für NumPy  arange()..:",laufzeit2)
print("Laufzeit für NumPy  linspace():",laufzeit3)
print("arange()   ist%4d mal schneller als range()" %faktor1)
print("linspace() ist%4d mal schneller als range()" %faktor2)
```

Listing 3.2 Laufzeitvergleich

Ausgabe

```
Laufzeit für Python range()...: 0.1445789337158203
Laufzeit für NumPy  arange()..: 0.0028200149536132812
Laufzeit für NumPy  linspace(): 0.0020291805267333984
arange()   ist  51 mal schneller als range()
linspace() ist  71 mal schneller als range()
```

Analyse

Die NumPy-Funktion `arange()` ist etwa 51-mal schneller und die NumPy-Funktion `linspace()` ist etwa 71-mal schneller als die Python-Liste, die mit der Python-Funktion `range()` erzeugt wird. Bei den Zeitmessungen handelt es sich nur um grobe Einschät-

zungen. Bei jedem neuen Programmstart und mit einer anderen Hardware und fallen die Ergebnisse anders aus.

Fazit: Für die numerische Auswertung großer Datenmengen sollten Sie NumPy-Arrays nutzen.

3.1.2 Zweidimensionale Arrays mit array() erzeugen

Bisher wurden mit den NumPy-Funktionen `arange()` und `linspace()` nur eindimensionale Arrays erzeugt. In der Praxis werden, z. B. für die Berechnung elektrischer Netzwerke oder für die Lösung linearer Gleichungssysteme, auch zweidimensionale Arrays benötigt. Zweidimensionale Arrays werden mit der NumPy-Funktion `array()` aus verschachtelten Listen erzeugt. Sie sollten für Berechnungen mit Matrizen nur die Funktion `array()` verwenden, weil die Funktion `matrix()` aus dem Modul NumPy in Zukunft entfernt werden soll.

Matrizen

Matrizen werden durch NumPy-Arrays repräsentiert.

Das Konsolenbeispiel demonstriert den Unterschied zwischen einem eindimensionalen und einem zweidimensionalen Array:

```
>>> import numpy as np
>>> a=np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> b=np.array([[1,2,3],[4,5,6]])
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.ndim
1
>>> b.ndim
2
>>> type(b)
<class 'numpy.ndarray'>
```

Die `array`-Funktion ist ebenso wie die Numpy-Funktionen `arange()` und `linspace()` Bestandteil (engl. *member*) der Klasse `ndarray`. Um die Operationen auf Arrays zu testen, ist es zweckmäßig die Erzeugung zweidimensionaler Arrays zu automatisieren. Mit der NumPy-Methode `obj.reshape()` können Sie ein eindimensionales Array in ein zweidimensionales Array umwandeln. Listing 3.3 zeigt, wie Sie eine $m \times n$-Matrix

aus einem eindimensionalen Array erzeugen können. Außerdem ermittelt das Programm noch den *Typ* (die Gestalt) des Arrays mit der Eigenschaft `shape` und zeigt, wie eine Matrix transponiert wird:

```
01  #03_2dim_array.py
02  import numpy as np
03  m=3 #Zeilen
04  n=4 #Spalten
05  a=np.arange(m*n).reshape(m,n)
06  b=a.reshape(n*m,)
07  print("Typ des Arrays",a.shape,"\n",a)
08  print("Linearisieren\n",b)
09  print("Transponieren\n",a.T)
```

Listing 3.3 Erzeigen eines zweidimensionalen Arrays

Ausgabe

```
Typ des Arrays (3, 4)
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Linearisieren
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
Transponieren
 [[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

Analyse

In den Zeilen 03 und 04 können Sie die Zeilen- und Spaltenanzahl des Arrays ändern.

In Zeile 05 wandelt die NumPy-Methode `reshape(m,n)` das eindimensionale Array in ein zweidimensionales Array um.

In Zeile 06 linearisiert die Methode `a.reshape(n*m,)` das zweidimensionale Array `a`. Die Anweisung `a.reshape(m,n)` wird hier als Methode bezeichnet, weil `reshape()` zu seiner Ausführung ein Objekt benötigt. Die Objektnotation `a.reshape(m,n)` kann mit der Formulierung »erzeuge aus dem Array-Objekt `a` ein Array-Objekt mit `m` Zeilen und `n` Spalten« in die Alltagssprache übersetzt werden.

In Zeile 07 ermittelt die Eigenschaft `shape` den Typ des Arrays `a`.

In der Zeile 09 wird das Array `a` mit `a.T` transponiert. Mit der Anweisung `np.transpose(a)` können Sie ebenfalls ein Array transponieren.

3.1.3 Slicing

Mit *Slicing* können Sie ausgewählte Teilbereiche von Elementen aus einem zweidimensionalen Array auslesen. Mit der allgemeinen Syntax `a[start:stop:step,start:stop:step]` wird aus der *m*-ten Zeile und der *n*-ten Spalte ein durch die Parameter `start,stop,step` festgelegter Teilbereich eines Arrays `a` ausgelesen. Der voreingestellte Wert von `step` ist 1. Mit `a[m,:]` können Sie die *m*-te Zeile auslesen und mit `a[:,n]` können Sie die *n*-te Spalte des Arrays `a` auslesen. Listing 3.4 zeigt anhand einer 4×4-Matrix, wie Slicing für das Auslesen von Spalten funktioniert. Die Matrix wird mit der NumPy-Methode `reshape()` erzeugt.

```
#04_slicing.py
import numpy as np
m=4 #Zeilen
n=4 #Spalten
a=np.arange(m*n).reshape(m,n)
#Ausgabe
print(a)
print("erste  Spalte\n",a[:,0])
print("zweite Spalte\n",a[:,1])
print("erste Zeile\n",  a[0,:])
print("zweite Zeile\n", a[1,:])
print("a[1:3,0:2]\n",   a[1:3,0:2])
```

Listing 3.4 Slicing

Ausgabe

```
[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
erste  Spalte
 [ 0  4  8 12]
zweite Spalte
 [ 1  5  9 13]
erste Zeile
 [0 1 2 3]
zweite Zeile
 [4 5 6 7]
a[1:3,0:2]
 [[4 5]
 [8 9]]
```

Analyse

In den Zeilen 03 und 04 wird die Anzahl der Zeilen und der Spalten für die in Zeile 05 erzeugte Matrix festgelegt. Dort erzeugt die NumPy-Methode `reshape(m,n)` eine 4×4-Matrix aus einer Sequenz von 16 ganzen Zahlen.

In den Zeilen von 08 bis 11 werden einzelne Spalten und Zeilen ausgelesen. Zu beachten ist, dass die Zählung bei dem Index 0 beginnt.

In Zeile 12 wird ein Bereich der Matrix ausgelesen.

3.1.4 Mathematische NumPy-Funktionen

NumPy stellt die gleichen mathematischen Funktionen zur Verfügung, die auch aus dem Python-Modul `math` bekannt sind. Warum aber nur mathematische Funktionen aus dem NumPy-Modul benutzt werden dürfen, wenn diesen Funktionen Argumente aus einem NumPy-Array übergeben werden, zeigt Listing 3.5:

```
01  #05_numpy_funktionen.py
02  import numpy as np
03  #import math
04  x=np.arange(-3,4,1)
05  #y1=math.sin(x)
06  y1=np.sin(x)
07  y2=np.exp(x)
08  y3=np.sinh(x)
09  y4=np.cosh(x)
10  y5=np.hypot(3,4)#Diagonale
11  y1,y2,y3,y4,y5=np.round((y1,y2,y3,y4,y5),decimals=3)
12  #Ausgabe
13  print("x-Werte:\n",x)
14  print("sin-Funktion:\n",y1)
15  print("e-Funktion:\n",y2)
16  print("sinh-Funktion:\n",y3)
17  print("cosh-Funktion:\n",y4)
18  print("Hypotenuse:",y5)
```

Listing 3.5 Ausgewählte mathematische NumPy-Funktionen

Ausgabe

```
x-Werte:
 [-3 -2 -1 0 1 2 3]
```

```
sin-Funktion:
 [-0.141 -0.909 -0.841 0. 0.841 0.909 0.141]
e-Funktion:
 [0.05 0.135 0.368 1. 2.718 7.389 20.086]
sinh-Funktion:
 [-10.018 -3.627 -1.175 0. 1.175 3.627 10.018]
cosh-Funktion:
 [10.068 3.762 1.543 1. 1.543 3.762 10.068]
Hypotenuse: 5.0
```

Analyse

Das Programm berechnet Wertetabellen für eine `sin`-, eine `e`-, eine `sinh`- und eine `cosh`-Funktion. Der Wertebereich liegt zwischen –3 und +3 (Zeile 04). Die Schrittweite beträgt 1. Dass die obere Grenze für die x-Werte bei +3 abbricht, obwohl im Quelltext als obere Grenze 4 festgelegt wurde, mag zunächst einmal irritieren. Die Dokumentation von NumPy liefert die Erklärung: Bei ganzzahliger und auch bei nicht ganzzahliger Schrittweite ist das Intervallende des Wertebereichs nicht mit enthalten. Es gilt also stets $x < 4$. In Ausnahmefällen kann es durch Rundungseffekte dazu kommen, dass das Intervallende mit eingeschlossen wird.

Auf die NumPy-Funktionen wird über den Punktoperator mit dem Alias `np` zugegriffen. Wenn in den Zeilen 03 und 05 die Kommentare entfernt werden, erscheint nach dem Programmstart folgende Fehlermeldung:

```
y1=math.sin(x)
TypeError: only size-1 arrays can be converted to Python scalars
```

Das heißt, Wertetabellen für die mathematischen Funktionen aus dem Modul `math` dürfen nur mit Schleifenkonstrukten erstellt werden. Für jede neue Berechnung eines Funktionswertes für `math.sin(x)` muss die Schleife erneut durchlaufen werden. Wenn Wertetabellen dagegen mit den NumPy-Funktionen `arange()` oder `linspace()` und den vordefinierten mathematischen Funktionen aus dem Modul NumPy erstellt werden, dann wird keine `for`- oder `while`-Schleife mehr benötigt. Alle mathematischen NumPy-Funktionen liefern ein `ndarray` zurück. Auf jeden diskreten Wert der Variablen `y1` bis `y4` kann also über den Indexoperator zugegriffen werden. Die Ausgaben in den Zeilen 14 bis 17 belegen das: Für jedes `x`-Argument wird der zugehörige Funktionswert ausgegeben.

Interessant ist die Anweisung in Zeile 11. An dieser Stelle rundet die NumPy-Funktion `round()` die Ausgaben für alle vier Funktionswerte auf drei Stellen, indem ihr ein Tupel aus vier Elementen übergeben wird. Die Funktion `round()` gibt ein Tupel mit ebenfalls vier Elementen zurück.

3.1.5 Statistische NumPy-Funktionen

NumPy stellt auch Funktionen für die Erzeugung von gleich- und normalverteilten Zufallszahlen bereit. Aus diesen Zahlen können mit NumPy-Statistikfunktionen der arithmetische Mittelwert, der Median, die Varianz und die Standardabweichung berechnet werden. Diese statistischen Funktionen stellt standardmäßig auch das Python-Modul `statistics` zur Verfügung. Die NumPy-Funktionen sind aber wesentlich leistungsfähiger. Deshalb sollten Sie bei der statistischen Auswertung großer Datenmengen die NumPy-Funktionen bevorzugt einsetzen. Listing 3.6 zeigt die Anwendung dieser Funktionen:

```
01 #06_numpy_statistik.py
02 import numpy as np
03 zeilen=5
04 spalten=10
05 np.random.seed(1)
06 x=np.random.normal(8,4,size=(zeilen,spalten))
07 mw=np.mean(x)
08 md=np.median(x)
09 v=np.var(x)
10 staw=np.std(x)
11 minimum=np.amin(x)
12 maximum=np.amax(x)
13 min_index=np.where(x==np.amin(x))
14 max_index=np.where(x==np.amax(x))
15 #min_index=np.argmin(x)
16 #max_index=np.argmax(x)
17 #Ausgabe
18 print("Zufallszahlen\n",np.round(x,decimals=2),"\n")
19 print("kleinste Zahl...........:",minimum)
20 print("größte Zahl.............:",maximum)
21 print("Index der kleinsten Zahl:",min_index)
22 print("Index der größten Zahl..:",max_index)
23 print("Mittelwert.............:",mw)
24 print("Median.................:",md)
25 print("Varianz................:",v)
26 print("Standardabweichung.....:",staw)
27 print("Typ von  x:",type(x))
28 print("Typ von mw:",type(mw))
```

Listing 3.6 NumPy-Statistikfunktionen

Ausgabe

```
Zufallszahlen
 [[14.5   5.55  5.89  3.71 11.46 -1.21 14.98  4.96  9.28  7. ]
 [13.85 -0.24  6.71  6.46 12.54  3.6   7.31  4.49  8.17 10.33]
 [ 3.6  12.58 11.61 10.01 11.6   5.27  7.51  4.26  6.93 10.12]
 [ 5.23  6.41  5.25  4.62  5.32  7.95  3.53  8.94 14.64 10.97]
 [ 7.23  4.45  5.01 14.77  8.2   5.45  8.76 16.4   8.48 10.47]]

kleinste Zahl...........: -1.2061547875211307
größte Zahl.............: 16.40102054591537
Index der kleinsten Zahl: (array([0]), array([5]))
Index der größten Zahl..: (array([4]), array([7]))
Mittelwert..............: 7.8979406079693995
Median..................: 7.271472480175898
Varianz.................: 15.041654042036352
Standardabweichung......: 3.8783571318325434
Typ von  x: <class 'numpy.ndarray'>
Typ von mw: <class 'numpy.float64'>
```

Analyse

In Zeile 06 erzeugt die NumPy-Funktion `random.normal(8,4,size=(zeilen,spalten))` 50 normalverteilte Zufallszahlen als Matrix mit fünf Zeilen und zehn Spalten. Als erstes Argument erwartet diese Funktion das Zentrum der Verteilung, als zweites Argument eine grobe Angabe für die Streuung der zu erzeugenden Zufallszahlen und als drittes Argument ein Tupel für die Anzahl der Zeilen und Spalten.

Damit bei jedem Neustart des Programms die gleichen Zufallszahlen erzeugt werden, steht in Zeile 05 die Funktion `random.seed()`. Wenn bei jedem neuen Programmstart auch neue Zufallszahlen erzeugt werden sollen, muss diese Funktion auskommentiert oder gelöscht werden.

Die Zeilen 07 bis 10 berechnen die erwähnten statistischen Maßzahlen: den Mittelwert `mw`, den Median `md`, die Varianz `v` und die Standardabweichung `staw`.

Interessant ist die Ermittlung des Array-Index für die kleinste und die größte Zufallszahl in den Zeilen 13 und 14. Die Funktion `where(x==np.amin(x))` ermittelt in Zeile 13 die Stelle im Array mit der kleinsten Zufallszahl: `[0,5]` (Ausgabe in Zeile 21). Für die Ermittlung des Index der größten Zufallszahl gilt Entsprechendes. Das Programm gibt hierfür in Zeile 22 den Index `[4,7]` aus. Eine Überprüfung anhand der in Zeile 18 ausgegebenen Zufallszahlen bestätigt die Ergebnisse. Einfacher kann die Stelle im Array, wo sich die kleinste bzw. größte Zufallszahl befindet, mit den Funktionen in den auskommentierten Zeilen 15 und 16 ermittelt werden.

Alle berechneten statistischen Maßzahlen sind vom Typ Float64 (Zeile 28). Sie haben also eine doppelte Genauigkeit mit 52 Bit Mantisse und 11 Bit Exponent.

3.2 Vektoren

Vektoren (lat. *vector*, dt. *Träger, Fahrer*) sind physikalische Größen, die im Gegensatz zu den skalaren Größen neben dem Betrag auch noch durch eine Richtung gekennzeichnet sind. Gerichtete Größen sind beispielsweise Geschwindigkeiten, Kräfte oder Feldstärken. In der Physik und Mathematik werden Vektoren anschaulich als Pfeile dargestellt (siehe Abbildung 3.1).

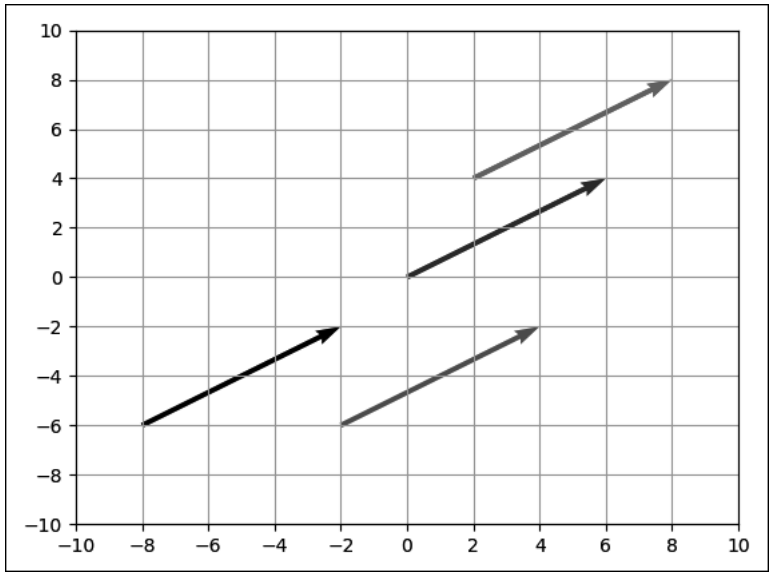

Abbildung 3.1 Verschiebung von Vektoren

Wie Abbildung 3.1 zeigt, können Vektoren in der Ebene beliebig verschoben werden – unter der Voraussetzung, dass ihre Beträge und Richtungen sich nicht ändern. Die gleiche Aussage gilt auch im dreidimensionalen Raum. Die dargestellten Vektoren haben die gleiche x- und y-Komponente von $x = 6$ und $y = 4$. In der Mathematik ist die Formulierung (6,4) üblich. Der Winkel beträgt jeweils etwa 33,7°.

3.2.1 Addition von Vektoren

Vektoren werden komponentenweise addiert. Abbildung 3.2 veranschaulicht diese Operation.

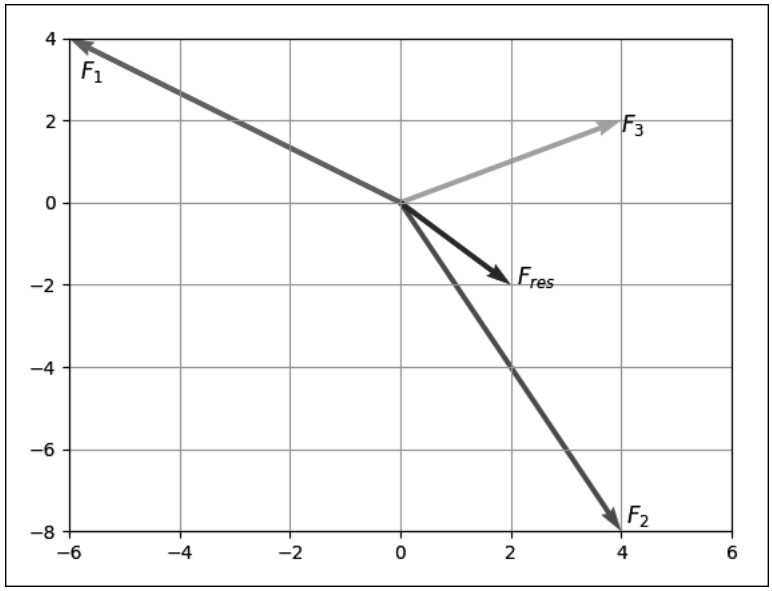

Abbildung 3.2 Addition von drei Vektoren

Wenn Sie den Vektor $\mathbf{F_1}$ = (–6,4), den Vektor $\mathbf{F_2}$ = (4,–8) und den Vektor $\mathbf{F_3}$ = (4,2) addieren, erhalten Sie den resultierenden Vektor $\mathbf{F}_{res}$ = (2,–2). In der Sprache der Mathematik formuliert:

$$\vec{F}_{res} = \begin{pmatrix} F_{x1} \\ F_{y1} \end{pmatrix} + \begin{pmatrix} F_{x2} \\ F_{y2} \end{pmatrix} + \begin{pmatrix} F_{x3} \\ F_{y3} \end{pmatrix} = \begin{pmatrix} -6 \\ 4 \end{pmatrix} + \begin{pmatrix} 4 \\ -8 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \end{pmatrix}$$

Vektoren können mit der `array`-Funktion aus Tupeln oder Listen erzeugt werden. Listing 3.7 zeigt die Umsetzung der Vektoraddition mit Tupeln:

```
#07_vektoraddition.py
import numpy as np
F1=-6,4
F2=4,-8
F3=4,2
F1=np.array(F1)
F2=np.array(F2)
F3=np.array(F3)
Fres=F1+F2+F3
F_1=np.sqrt(F1[0]**2+F1[1]**2)
F_2=np.sqrt(F2[0]**2+F2[1]**2)
F_3=np.sqrt(F3[0]**2+F3[1]**2)
F_res=np.sqrt(Fres[0]**2+Fres[1]**2)
```

```
14  winkel=np.arctan(Fres[0]/Fres[1])
15  winkel=np.degrees(winkel)
16  #Ausgabe
17  print("Koordinaten von F1:",F1)
18  print("Koordinaten von F2:",F2)
19  print("Koordinaten von F3:",F3)
20  print("Betrag von F1     :",F_1)
21  print("Betrag von F2     :",F_2)
22  print("Betrag von F3     :",F_3)
23  print("resultierende Kraft:",Fres)
24  print("Betrag von Fres   :",F_res)
25  print("Winkel von Fres   :",winkel,"°")
```

Listing 3.7 Addition von drei Vektoren

Ausgabe

```
Koordinaten von F1: [-6  4]
Koordinaten von F2: [ 4 -8]
Koordinaten von F3: [4 2]
Betrag von F1     : 7.211102550927978
Betrag von F2     : 8.94427190999916
Betrag von F3     : 4.47213595499958
resultierende Kraft: [ 2 -2]
Betrag von Fres   : 2.8284271247461903
Winkel von Fres   : -45.0 °
```

Analyse

In den Zeilen 03 bis 05 werden die x-y-Komponenten der drei Kräfte den Variablen `F1` bis `F3` als Tupel übergeben. Die Anweisungen in den Zeilen 06 bis 08 erstellen jeweils ein eindimensionales NumPy-Array aus den Kräftekomponenten.

In Zeile 09 erfolgt die Vektoraddition. Die Kräfte werden elementweise addiert. Die internen Vorgänge bleiben dem Anwender verborgen. Das Programm rechnet intern `Fres[0]=F1[0]+F2[0]+F3[0]` und `Fres[1]=F1[1]+F2[1]+F3[1]`. Zeile 23 gibt das Ergebnis aus. Das Programm berechnet die Beträge der drei Kräfte mit dem Satz des Pythagoras aus (Zeile 10 bis 12). Zeile 14 berechnet den Winkel der resultierenden Kraft mit der NumPy-Funktion `arctan(Fres[0]/Fres[1])`. Die NumPy-Funktion `degrees(winkel)` in Zeile 15 sorgt dafür, dass der Winkel in Grad umgerechnet wird.

Die Ausgaben des Programms in den Zeilen 17 bis 25 lassen sich anhand von Abbildung 3.2 leicht überprüfen. Auf die Angaben der Einheiten wurde bewusst verzichtet.

3.2.2 Skalarprodukt

In der Mechanik wird die Arbeit als Produkt aus der Kraft F mal dem Weg s mal dem Kosinus des Winkels α zwischen den beiden Größen definiert:

$$W = F \cdot s \cdot \cos\alpha$$

Aus dieser Definition kann mit dem Kosinussatz die Koordinatenform des Skalarprodukts hergeleitet werden:

$$W = F_x s_x + F_y s_y + F_z s_z = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \cdot \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix}$$

Das *Skalarprodukt* wird als Summe der Produkte aus Kraft- und Wegkomponenten berechnet.

In abgekürzter Schreibweise gilt für die Definition des Skalarprodukts:

$$W = \vec{F} \cdot \vec{s}$$

Der Betrag der Kraft wird aus der Wurzel des Skalarprodukts des Kraftvektors mit sich selbst berechnet:

$$F = \sqrt{\vec{F} \cdot \vec{F}}$$

Und der Betrag des Weges wird aus der Wurzel des Skalarprodukts des Wegvektors mit sich selbst berechnet:

$$s = \sqrt{\vec{s} \cdot \vec{s}}$$

Für den Winkel zwischen Kraftvektor und Wegvektor gilt:

$$\alpha = \arccos\left(\frac{\vec{F} \cdot \vec{s}}{Fs}\right)$$

An einem Beispiel für $\vec{F} = (2{,}7, -3)$ N und $\vec{s} = (-2{,}3{,}4)$ m soll gezeigt werden, wie das Skalarprodukt für dreidimensionale Vektoren berechnet wird. Es gilt:

$$W = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \cdot \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ -3 \end{pmatrix} \text{N} \cdot \begin{pmatrix} -2 \\ 3 \\ 4 \end{pmatrix} \text{m} = 5 \text{ Nm}$$

Für die angegebenen Komponenten des Vektors der Kraft und des Vektors des Weges wird also eine Arbeit von 5 Nm verrichtet. Die NumPy-Funktion `dot(F,s)` berechnet das Skalarprodukt. Listing 3.8 zeigt, wie die mechanische Arbeit aus dem Skalarprodukt des Kraft- und Wegvektors berechnet wird:

```
01  #08_skalarprodukt.py
02  import numpy as np
```

```
03  F=2,7,-3
04  s=-2,3,4
05  F_B=np.sqrt(np.dot(F,F))
06  s_B=np.sqrt(np.dot(s,s))
07  cos_Fs=np.dot(F,s)/(F_B*s_B)
08  winkel=np.degrees(np.arccos(cos_Fs))
09  W=np.dot(F,s)
10  #Ausgabe
11  print("Betrag der Kraft:",F_B,"N")
12  print("Betrag des Weges:",s_B,"m")
13  print("Winkel zwischen F und s:",winkel,"°")
14  print("Arbeit:",W,"Nm")
```

Listing 3.8 Skalarprodukt

Ausgabe

```
Betrag der Kraft: 7.874007874011811 N
Betrag des Weges: 5.385164807134504 m
Winkel zwischen F und s: 83.22811782220313 °
Arbeit: 5 Nm
```

Analyse

In den Zeilen 03 und 04 werden den Variablen `F` und `s` drei Kraft- bzw. drei Wegkomponenten als Tupel übergeben. Das erste Element eines Tupels enthält die x-Komponente, das zweite die y-Komponente und das dritte die z-Komponente der Vektoren Kraft `F` und Weg `s`.

Die Zeilen 05 und 06 berechnen die Beträge der Vektoren mit dem Skalarprodukt der NumPy-Funktion `dot(F,F)` bzw. `dot(s,s)`. Der Winkel zwischen dem Kraft- und dem Wegvektor wird ebenfalls mit der `dot`-Funktion berechnet (Zeile 07). Zeile 09 berechnet die mechanische Arbeit `W` mit dem Skalarprodukt `W=np.dot(F,s)`. Das Programm berechnet intern die mechanische Arbeit durch die elementweise Multiplikation, wie es die Definition des Skalarprodukts verlangt: `W=F[0]s[0]+F[1]s[1]+F[2]s[2]`.

Zeile 14 gibt die an einem im Raum bewegten Massepunkt verrichtete mechanische Arbeit `W` aus. Das Ergebnis von `5 Nm` stimmt mit dem zuvor ermittelten Wert überein.

3.2.3 Kreuzprodukt

Der Betrag des Drehmoments M wird definiert als Produkt aus der Kraft F mal dem Hebelarm l mal dem Sinus des Winkels α zwischen den beiden Größen:

$$M = F \cdot l \cdot \sin \alpha$$

Aus dieser Definition kann die Koordinatenform des *Kreuzprodukts* hergeleitet werden:

$$\vec{M} = \begin{pmatrix} F_y l_z - F_z l_y \\ F_z l_x - F_x l_z \\ F_x l_y - F_y l_x \end{pmatrix} = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} \times \begin{pmatrix} l_x \\ l_y \\ l_z \end{pmatrix}$$

Abgekürzt formuliert, gilt für die Definition des Kreuzprodukts:

$$\vec{M} = \vec{F} \times \vec{l}$$

Für $\vec{F} = (2{,}7, -3)$ und $\vec{l} = (-2{,}3{,}4)$ ergibt sich der Drehmomentvektor:

$$\vec{M} = \begin{pmatrix} F_y l_z - F_z l_y \\ F_z l_x - F_x l_z \\ F_x l_y - F_y l_x \end{pmatrix} = \begin{pmatrix} 7 \cdot 4 & -(-3) \cdot 3 \\ -3 \cdot -2 & -2 \cdot 4 \\ 2 \cdot 3 & -7 \cdot -2 \end{pmatrix} \text{Nm} = \begin{pmatrix} 37 \\ -2 \\ 20 \end{pmatrix} \text{Nm}$$

Listing 3.9 berechnet das Drehmoment aus dem Kraft- und dem Hebelvektor im dreidimensionalen Raum mit der NumPy-Funktion `cross(F,l)`:

```
01 #09_kreuzprodukt.py
02 import numpy as np
03 F=2,7,-3
04 l=-2,3,4
05 F_B=np.sqrt(np.dot(F,F))
06 l_B=np.sqrt(np.dot(l,l))
07 cos_Fl=np.dot(F,l)/(F_B*l_B)
08 winkel=np.degrees(np.arccos(cos_Fl))
09 M=np.cross(F,l)
10 M_B=np.sqrt(np.dot(M,M))
11 #Ausgabe
12 print("Betrag der Kraft       :",F_B,"N")
13 print("Betrag des Hebelarms   :",l_B,"m")
14 print("Winkel zwischen F und l:",winkel,"°")
15 print("Drehmoment M           :",M,"Nm")
16 print("Betrag des Drehmoments :",M_B,"Nm")
```

Listing 3.9 Kreuzprodukt

Ausgabe

```
Betrag der Kraft       : 7.874007874011811 N
Betrag des Hebelarms   : 5.385164807134504 m
Winkel zwischen F und l: 83.22811782220313 °
Drehmoment M           : [37 -2 20] Nm
Betrag des Drehmoments : 42.1070065428546 Nm
```

Analyse

Der Kraftvektor `F` und der Vektor des Hebelarms `l` werden in den Zeilen 03 und 04 wieder als Tupel festgelegt.

In Zeile 09 berechnet das Programm das Kreuzprodukt mit der NumPy-Funktion `M=np.cross(F,l)`. Das Ergebnis ist wieder ein Vektor `[37 -2 20] Nm` (Ausgabe in Zeile 15). Der Betrag des Drehmoments von `42.1 Nm` entspricht dem Flächeninhalt des Parallelogramms, das durch den Kraftvektor `F` und den Vektor des Hebelarms `l` umspannt wird.

3.2.4 Spatprodukt

Das *Spatprodukt* berechnet das Volumen eines Spats (Parallelepiped) aus dem Kreuzprodukt und dem Skalarprodukt:

$$V = \vec{c} \cdot (\vec{a} \times \vec{b})$$

Listing 3.10 berechnet das Volumen eines Quaders mithilfe des Spatprodukts `dot(c, np.cross(a,b))`:

```
01  #10_spatprodukt.py
02  import numpy as np
03  a=2,0,0
04  b=0,3,0
05  c=0,0,4
06  a_B=np.sqrt(np.dot(a,a))
07  b_B=np.sqrt(np.dot(b,b))
08  c_B=np.sqrt(np.dot(c,c))
09  V=np.dot(c,np.cross(a,b))
10  #Ausgabe
11  print("Betrag von a:",a_B)
12  print("Betrag von b:",b_B)
13  print("Betrag von c:",c_B)
14  print("Spatprodukt :",V)
```

Listing 3.10 Spatprodukt

Ausgabe

```
Betrag von a: 2.0
Betrag von b: 3.0
Betrag von c: 4.0
Spatprodukt : 24
```

Analyse

Die Komponenten der drei Vektoren `a`, `b` und `c` wurden so gewählt, dass sie einen Quader mit den Seiten `a=2`, `b=3` und `c=4` bilden.

Zeile 09 berechnet das Spatprodukt aus den NumPy-Funktionen `dot()` und `cross()`. Der `dot`-Funktion werden die Variable `c` für die Höhe des Quaders und die `cross(a,b)`-Funktion für die Berechnung der Grundfläche als Argumente übergeben.

Die Ausgabe in Zeile 14 liefert das richtige Ergebnis von 24 Raumeinheiten.

3.2.5 Dyadisches Produkt

Beim *dyadischen Produkt*, auch äußeres Produkt genannt, werden die Zeilenvektoren mit dem Spaltenvektoren multipliziert:

$$(1 \quad 2 \quad 3) \otimes \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{pmatrix}$$

Listing 3.11 berechnet das dyadische Produkt für die angegebenen Matrizen.

```
01  #11_outer.py
02  import numpy as np
03  A=np.array([[1,2,3]])
04  B=np.array([[4],[5],[6]])
05  C=np.outer(A,B)
06  print("Matrix A")
07  print(A)
08  print("Matrix B")
09  print(B)
10  print("dyadisches Produkt")
11  print(C)
```

Listing 3.11 Dyadisches Produkt

Ausgabe

```
Matrix A
[[1 2 3]]
Matrix B
[[4]
 [5]
 [6]]
dyadisches Produkt
[[ 4  5  6]
 [ 8 10 12]
 [12 15 18]]
```

Analyse

In Zeile 03 wird ein Zeilenvektor `A` und in Zeile 04 wird ein Spaltenvektor `B` definiert. Das dyadische Produkt berechnet die NumPy-Funktion `outer(A,B)` in Zeile 05. Das Ergebnis stimmt mit dem manuell berechneten Wert überein.

3.3 Matrizenmultiplikation

Die *Matrizenmultiplikation* wird beispielsweise bei der Berechnung von elektrischen Netzwerken benötigt. Wenn mehrere Zweitore hintereinandergeschaltet werden (Kettenschaltung), dann können mit der Kettenform (A-Parameter) für eine vorgegebene Ausgangsspannung und einen vorgegebenen Ausgangsstrom die erforderliche Eingangsspannung und der erforderliche Eingangsstrom durch Matrizenmultiplikation berechnet werden.

Zwei Matrizen werden miteinander multipliziert, indem die Zeilen der ersten Matrix mit den Spalten der zweiten Matrix elementweise multipliziert und die einzelnen Produkte addiert (*falksches Schema*) werden:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}.b_{11} + a_{12} \cdot b_{21} & a_{11}.b_{12} + a_{12} \cdot b_{22} \\ a_{21}.b_{11} + a_{22} \cdot b_{21} & a_{21}.b_{12} + a_{22} \cdot b_{22} \end{pmatrix}$$

Ein einfaches Beispiel soll die Matrizenmultiplikation anhand eines Schemas demonstrieren (siehe Tabelle 3.1). Die beiden folgenden Matrizen sollen multipliziert werden:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Die erste Matrix wird in der ersten und zweiten Spalte und der dritten und vierten Zeile einer Tabelle eingetragen. Die zweite Matrix wird in der dritten und vierten Spalte und in der ersten und zweiten Zeile einer Tabelle eingetragen.

		5	6
		7	8
1	2	1 · 5 + 2 · 7 = **19**	1 · 6 + 2 · 8 = **22**
3	4	3 · 5 + 4 · 7 = **43**	3 · 6 + 4 · 8 = **50**

Tabelle 3.1 Schema für die Matrizenmultiplikation

Die erste Zeile der ersten Matrix wird mit der ersten Spalte der zweiten Matrix elementweise multipliziert. Die beiden Produkte werden addiert. Die zweite Zeile der ersten Matrix wird mit der ersten Spalte der zweiten Matrix multipliziert. Die beiden

Produkte werden wiederum addiert. Die zweite Spalte wird nach dem gleichen Schema berechnet.

NumPy stellt die Funktion `array([[a11,a12],[a21,a22]])` für die Erzeugung der Matrizen zur Verfügung. Die Anzahl der Zeilen und Spalten können Sie bei Bedarf anpassen.

Die Matrizenmultiplikation kann am einfachsten mit dem Infixoperator `@` durchgeführt werden. Alternativen sind `matmul(A,B)` oder `multi_dot([A,B,C,...])`.

Listing 3.12 zeigt die Durchführung der Matrizenmultiplikation anhand der Zahlen aus dem obigen Beispiel:

```
01  #12_mulmatrix1.py
02  import numpy as np
03  A=np.array ([[1, 2],
04              [3, 4]])
05  B=np.array ([[5, 6],
06              [7, 8]])
07  C=A@B
08  D=B@A
09  #Ausgabe
10  print(type(A))
11  print("Matrix A\n",A)
12  print("matrix B\n",B)
13  print("Produkt A*B\n",C)
14  print("Produkt B*A\n",D)
```

Listing 3.12 Matrizenmultiplikation

Ausgabe

```
<class 'numpy.ndarray'>
Matrix A
 [[1 2]
 [3 4]]
Matrix B
 [[5 6]
 [7 8]]
Produkt A*B
 [[19 22]
 [43 50]]
Produkt B*A
 [[23 34]
 [31 46]]
```

Analyse

Die Zeilen 03 bis 06 definieren Matrizen mit jeweils zwei Zeilen und zwei Spalten. Die Werte der einzelnen Koeffizienten werden in die Variablen `A` und `B` gespeichert.

Zeile 07 führt die Matrizenmultiplikation `C=A@B` durch und Zeile 08 führt die Multiplikation mit vertauschter Reihenfolge der Faktoren `D=B@A` durch.

Das Produkt für `C` gibt Zeile 13 korrekt so aus, wie es manuell in Tabelle 3.1 berechnet wurde. Das Ergebnis aus Zeile 14 weicht dagegen hiervon ab. Dieses Ergebnis ist ebenfalls korrekt, wie Sie leicht durch Nachrechnen überprüfen können. Sie lernen daraus, dass für das Matrizenprodukt das Kommutativgesetz nicht gilt.

Anwendungsbeispiel: Analyse einer π-Ersatzschaltung

Ein Python-Programm soll für eine π-Ersatzschaltung aus Abbildung 3.3 die Matrix der Kettenparameter und die erforderlichen Eingangsgrößen U_1 und I_1 für gegebene Ausgangsgrößen U_2 und I_2 berechnen.

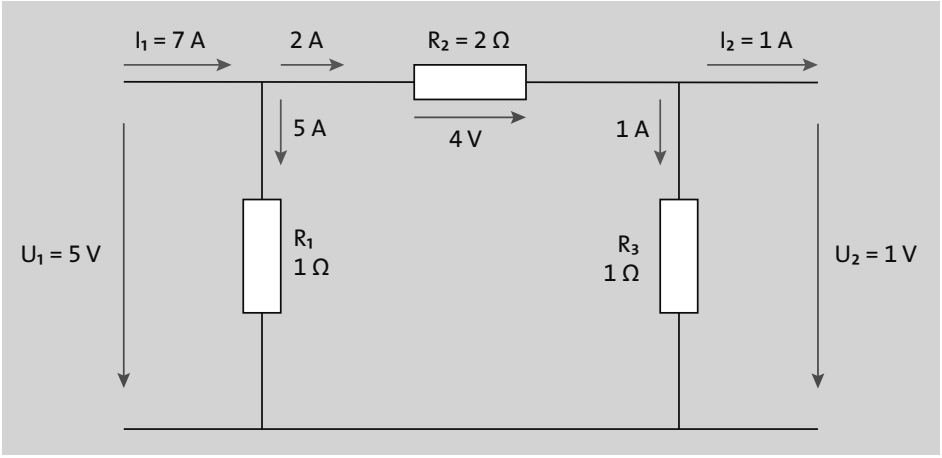

Abbildung 3.3 π-Ersatzschaltung

Jedes passive Zweitor lässt sich allgemein durch ein lineares Gleichungssystem mit einer Matrix aus vier Parametern und den Spaltenvektoren aus Spannungen oder Strömen beschreiben.

Kettenform mit A-Parametern

Für das gestellte Problem muss die sogenannte Kettenform gewählt werden. Auf der linken Seite des Gleichungssystems steht der Spaltenvektor der gesuchten Eingangsgrößen. Auf der rechten Seite steht die Kettenmatrix mit den vier Koeffizienten A_{11} bis A_{22}. Die Kettenmatrix wird mit dem Spaltenvektor der Ausgangsgrößen multipliziert. Wenn die Koeffizienten der Kettenmatrix und der Spaltenvektor der Ausgangsgrößen bekannt sind, können die Eingangsgrößen U_1 und I_1 berechnet werden.

$$\begin{pmatrix} U_1 \\ I_1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} U_2 \\ I_2 \end{pmatrix}$$

Für die Querwiderstände R_1 und R_3 lassen sich folgende A-Parameter aus der Schaltung aus Abbildung 3.3 ermitteln:

$$A_{1q} = \begin{pmatrix} 1 & 0 \\ \frac{1}{R_1} & 1 \end{pmatrix}, A_{2q} = \begin{pmatrix} 1 & 0 \\ \frac{1}{R_3} & 1 \end{pmatrix}$$

Für den Längswiderstand R_2 ergibt sich folgende Matrix:

$$A_l = \begin{pmatrix} 1 & R_2 \\ 0 & 1 \end{pmatrix}$$

Um die Systemmatrix der gesamten Schaltung aus Abbildung 3.3 zu erhalten, müssen Sie alle drei Teilmatrizen miteinander multiplizieren.

Listing 3.13 führt die Matrizenmultiplikation aus den drei Teilmatrizen für die π-Ersatzschaltung durch. Die Werte für die Widerstände können Sie natürlich für weitere Testzwecke ändern.

```
01  #13_mulmatrix2.py
02  import numpy as np
03  R1=1
04  R2=2
05  R3=1
06  U2=1
07  I2=1
08  A1q=np.array([[1, 0],
09                [1/R1, 1]])
10  Al=np.array([[1, R2],
11               [0, 1]])
12  A2q=np.array([[1, 0],
13                [1/R3, 1]])
14  A=A1q@Al@A2q
15  b=np.array([[U2],[I2]])
16  E=A@b
17  print("Kettenform A\n",A)
18  print("Eingangsgrößen\n",E)
19  print("Eingangsspannung U1=%3.2f V" %E[0])
20  print("Eingangsstrom    I1=%3.2f A" %E[1])
```

Listing 3.13 Matrizenmultiplikation mit A-Kettenparametern

Ausgabe

```
Kettenform A
 [[3. 2.]
 [4. 3.]]
Eingangsgrößen
 [[5.]
 [7.]]
Eingangsspannung U1=5.00 V
Eingangsstrom    I1=7.00 A
```

Analyse

Die Werte für die Ausgangsspannung U_2, die Ausgangsstromstärke I_2 und die drei Widerstände R_1, R_2, R_3 wurden aus den Vorgaben der Schaltung aus Abbildung 3.3 übernommen.

In den Zeilen 08 bis 13 werden die drei Teilmatrizen `A1q`, `Al` und `A2q` für die Querwiderstände R_1 und R_3 sowie den Längswiderstand R_2 definiert. Zeile 14 führt die Matrizenmultiplikation `A=A1q@Al@A2q` durch. Dabei ist auf die richtige Reihenfolge der Faktoren zu achten. Wie in Listing 3.12 gezeigt wurde, gilt das Kommutativgesetz für die Matrizenmultiplikation nicht! Eine veränderte Anordnung der Reihenfolge der Teilmatrizen würde auch eine andere Schaltungsstruktur repräsentieren.

Zeile 15 erzeugt den Spaltenvektor `b=np.array([[U2],[I2]])` für die Ausgangsgrößen. In Zeile 16 wird die Systemmatrix `A` mit dem Spaltenvektor `b` multipliziert. Das Ergebnis der Matrizenmultiplikation wird dem Spaltenvektor `E` zugewiesen.

Die Eingangsspannung muss 5 V betragen, damit am Ausgang der π-Ersatzschaltung eine Spannung von $U_2 = 1$ V anliegt. Am Eingang der Schaltung muss ein Strom von $I_1 = 7$ A fließen, damit am Ausgang ein Strom von $I_2 = 1$ A fließt. Die Ergebnisse können Sie anhand der Schaltung aus Abbildung 3.3 überprüfen.

Kettenform mit B-Parametern

Die Ausgangsgrößen U_2 und I_2 eines Zweitors werden mit den *B*-Kettenparametern berechnet. Für eine π-Ersatzschaltung ergeben sich dann folgende Zweitorgleichungen:

$$\begin{pmatrix} U_2 \\ I_2 \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \cdot \begin{pmatrix} U_1 \\ I_1 \end{pmatrix}$$

Für die Querwiderstände R_1 und R_3 lassen sich die *B*-Parameter aus der Schaltung aus Abbildung 3.3 wie folgt ermitteln:

$$B_{1q} = \begin{pmatrix} 1 & 0 \\ -\frac{1}{R_1} & 1 \end{pmatrix}, \qquad B_{2q} = \begin{pmatrix} 1 & 0 \\ -\frac{1}{R_3} & 1 \end{pmatrix}$$

Für den Längswiderstand R_2 ergibt sich folgende Matrix:

$$B_l = \begin{pmatrix} 1 & -R_2 \\ 0 & 1 \end{pmatrix}$$

Allgemein lassen sich die *B*-Parameter aus der inversen Matrix von *A* ermitteln. Es gilt:

$$B = A^{-1}$$

Listing 3.14 berechnet die Ausgangsspannung U_2 und den Ausgangsstrom I_2 einer π-Ersatzschaltung mit den B-Kettenparametern:

```
01  #14_mulmatrix3.py
02  import numpy as np
03  R1=1
04  R2=2
05  R3=1
06  U1=5
07  I1=7
08  B1q=np.array([[1, 0],
09              [-1/R1, 1]])
10  B2l=np.array([[1, -R2],
11              [0, 1]])
12  B3q=np.array([[1, 0],
13              [-1/R3, 1]])
14  B=B1q@B2l@B3q
15  b=np.array([[U1],[I1]])
16  E=B@b
17  print("Kettenform B\n",B)
18  print("Ausgangsgrößen\n",E)
19  print("Ausgangsspannung U2=%3.2fV" %E[0])
20  print("Ausgangsstrom    I2=%3.2fA" %E[1])
```

Listing 3.14 Matrizenmultiplikation mit B-Kettenparametern

Ausgabe

```
Kettenform B
 [[ 3. -2.]
 [-4.  3.]]
Ausgangsgrößen
 [[1.]
 [1.]]
Ausgangsspannung U2=1.00V
Ausgangsstrom    I2=1.00A
```

Analyse

Das Programm ist im Prinzip genauso aufgebaut wie in Listing 3.13, nur haben die Parameter in der Nebendiagonale ein negatives Vorzeichen.

Das Ergebnis für die Ausgangsspannung U_2 und den Ausgangsstrom I_2 stimmt mit den Werten überein, die anhand der kirchhoffschen Gesetze in der Schaltung in Abbildung 3.3 ermittelt wurden.

Anwendungsbeispiel: Die Energie eines rotierenden starren Körpers im Raum berechnen

Das nächste Beispiel zeigt die Multiplikation des Zeilenvektors einer Winkelgeschwindigkeit mit einem Trägheitstensor *I* (3×3-Matrix) und dem Spaltenvektor einer Winkelgeschwindigkeit.

Für die Rotationsenergie gilt:

$$E_{\text{rot}} = \frac{1}{2}\vec{\omega}^T \cdot I \cdot \vec{\omega}$$

Das hochgestellte *T* bedeutet, dass der Vektor der Winkelgeschwindigkeit *transponiert* werden muss, das heißt, der Spaltenvektor wird in einen Zeilenvektor umgewandelt. In Komponentenschreibweise erhält man:

$$E_{\text{rot}} = \frac{1}{2}(\omega_x \quad \omega_y \quad \omega_z) \cdot \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix} \cdot \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

Für den Trägheitstensor einer Punktmasse *m* gilt:

$$I = m \cdot \begin{pmatrix} y^2 + z^2 & -xy & -xz \\ -yx & x^2 + z^2 & -yz \\ -zx & -zy & x^2 + y^2 \end{pmatrix}$$

Das Produkt aus der Masse *m* und der Matrix mit den Ortskoordinaten wird als Trägheitstensor bezeichnet. Wenn Sie die Matrizenmultiplikation durchführen, erhalten Sie die Rotationsenergie, die in dem rotierenden Körper gespeichert ist.

Für den Fall, dass die Punktmasse *m* mit dem Radius $x = r$ um die z-Achse in der x-y-Ebene rotiert, gilt vereinfacht:

$$I = m \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \end{pmatrix}$$

Listing 3.15 berechnet die Rotationsenergie einer Punktmasse mit der Masse von $m = 6$ kg, die mit einer Winkelgeschwindigkeit von $\vec{\omega} = (0,0,1)s^{-1}$ im Raum um die z-Achse rotiert:

```
01 #15_mulmatrix4.py
02 import numpy as np
03 x=1 #Abstand in m
04 y=0
05 z=0
06 wx=0
07 wy=0
08 wz=1 #Winkelgeschwindigkeit
09 m=6  #Masse in kg
10 w_Z=np.array([wx,wy,wz])
11 I=m*np.array([[y**2+z**2, -x*y, -x*z],
12                [-x*y, x**2+z**2, -y*z],
13                [-x*z, -y*z, x**2+y**2]])
14 w_S=np.array([[wx],
15                [wy],
16                [wz]])
17 #Berechnung der Rotationsenergie
18 Erot=0.5*w_Z@I@w_S
19 #Erot=0.5*w_S.T@I@w_S
20 #Ausgabe
21 print("Rotationsenergie: %3.2f Joule" %Erot)
```

Listing 3.15 Matrixmultiplikation mit drei Vektoren

Ausgabe

```
Rotationsenergie: 3.00 Joule
```

Analyse

Die Rotationsenergie wird nach der Vorschrift »Zeilenvektor mal 3×3-Matrix mal Spaltenvektor« berechnet. Diese Reihenfolge muss zwingend eingehalten werden. Hier gilt das Kommutativgesetz nicht! In Zeile 10 steht der Zeilenvektor der Winkelgeschwindigkeit, die Zeilen 11 bis 13 enthalten die 3×3-Matrix des Trägheitstensors, und in den Zeilen 14 bis 16 steht der Spaltenvektor der Winkelgeschwindigkeit.

Die Anweisung in Zeile 18 führt die Matrizenmultiplikation durch und speichert das Ergebnis in die Variable `Erot`. Alternativ können Sie die Zeilen 10 und 18 auskommentieren und den Kommentar in Zeile 19 entfernen. In dieser Zeile wird der Spaltenvektor aus Zeile 14 mit der Eigenschaft `T` in einen Zeilenvektor transponiert.

3.4 Lineare Gleichungssysteme

Die Elektrotechnik verwendet lineare Gleichungssysteme, um Maschenströme und Knotenspannungen in Netzwerken zu berechnen. Die Statik verwendet für die Berechnung von Stabkräften in Fachwerken ebenfalls lineare Gleichungssysteme. Das Lösen von linearen Gleichungssystemen mit *n*-Unbekannten ist also ein wichtiges und unverzichtbares Instrument der Ingenieurwissenschaften. Mit der NumPy-Funktion `solve()` lassen sich Gleichungssysteme mit reellen und komplexen Koeffizienten genauso einfach und ohne großen Aufwand lösen wie beispielsweise mit dem Programm MATLAB.

3.4.1 Gleichungssysteme mit reellen Koeffizienten

Ein lineares Gleichungssystem kann allgemein als Matrixprodukt aus Koeffizientenmatrix (Systemmatrix) a_{11} bis a_{mn} und Lösungsvektor x als Gleichung dargestellt werden. Auf der rechten Seite des Gleichungssystems steht der Inhomogenitätsvektor b.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Abgekürzt ist folgende Schreibweise üblich:

$$A \cdot x = b$$

Um den Lösungsvektor x zu bestimmen, muss die inverse Matrix A^{-1} gebildet und mit dem Inhomogenitätsvektor b multipliziert werden:

$$x = A^{-1} \cdot b$$

Anhand eines einfachen Beispiels soll die Lösung eines einfachen Gleichungssystems mit drei Unbekannten gezeigt werden:

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & -2 & 3 \\ 3 & -4 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 7 \\ 1 \end{pmatrix}$$

Listing 3.16 löst ein lineares Gleichungssystem für drei Unbekannte mit der NumPy-Funktion `solve(A,b)`:

```
01  #16_gleichungssystem.py
02  import numpy as np
03  from numpy.linalg import solve
04  #Koeffizientenmatrix
05  A = np.array([[1,  1, 1],
06                [2, -2, 3],
07                [3, -4, 2]])
```

```
08 #Inhomogenitätsvektor
09 b = np.array([6, 7, 1])
10 #Lösung
11 loesung=solve(A,b)
12 #Ausgabe
13 print("Lösung eines linearen Gleichungssystems")
14 print("Koeffizientenmatrix\n",A)
15 print("Inhomogenitätsvektor\n",b)
16 print("Lösung:\n",loesung)
```

Listing 3.16 Lösung eines linearen Gleichungssystems

Ausgabe

```
Lösung eines linearen Gleichungssystems
Koeffizientenmatrix
 [[ 1  1  1]
 [ 2 -2  3]
 [ 3 -4  2]]
Inhomogenitätsvektor
 [6 7 1]
Lösung:
 [1. 2. 3.]
```

Analyse

Zeile 03 importiert das Untermodul `linalg` mit der Funktion `solve`.

In den Zeilen 05 bis 07 wird die Koeffizientenmatrix `A` des Gleichungssystems als zweidimensionales NumPy-Array erzeugt.

In Zeile 09 wird der Inhomogenitätsvektor `array([6,7,1])` der Variablen `b` zugewiesen.

In Zeile 11 berechnet die NumPy-Funktion `solve(A,b)` die Lösung des linearen Gleichungssystems. Der Lösungsvektor ist in der Variablen `loesung` gespeichert.

Der Lösungsvektor enthält Gleitpunktzahlen, obwohl die Koeffizientenmatrix und der Inhomogenitätsvektor aus Ganzzahlen bestehen. Wenn Sie sich mit

```
print(type(A[0,0]))
print(type(b[0]))
print(type(loesung[0]))
```

die Datentypen ausgeben lassen, erhalten Sie diese Ausgabe:

```
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.float64'>
```

Wenn bei einer mathematischen Operation auf Arrays Gleitpunktzahlen entstehen, werden alle Ganzzahlen des Arrays in Gleitpunktzahlen umgewandelt. Wenn Sie eine einzige beliebige Ganzzahl eines Arrays als Gleitpunktzahl deklarieren (z. B. `2.` statt `2`), dann werden alle anderen Elemente des Arrays automatisch in Gleitpunktzahlen umgewandelt.

3.4.2 Gleichungssysteme mit komplexen Koeffizienten

In einem Wechselstromnetzwerk besteht ein komplexer Widerstand entweder aus einer induktiv oder kapazitiv wirkenden Impedanz:

$$Z_L = R + \mathrm{j}\omega L \quad Z_C = R - \mathrm{j}\frac{1}{\omega C}$$

Für ein Netzwerk mit vier Maschen gilt allgemein:

$$\begin{pmatrix} Z_{11} & Z_{12} & Z_{13} & Z_{14} \\ Z_{21} & Z_{22} & Z_{23} & Z_{24} \\ Z_{31} & Z_{32} & Z_{33} & Z_{34} \\ Z_{41} & Z_{42} & Z_{43} & Z_{44} \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{pmatrix} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{pmatrix}$$

Am Beispiel des Netzwerks aus Abbildung 3.4 wird gezeigt, wie ein Gleichungssystem nach dem Maschenstromverfahren direkt aus der Schaltung abgelesen wird.

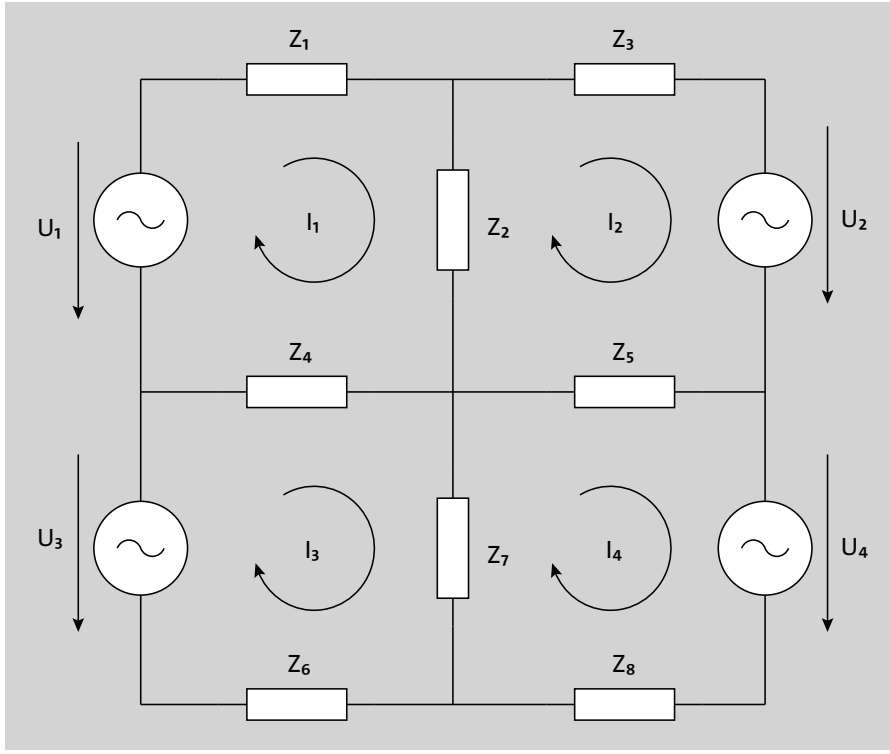

Abbildung 3.4 Wechselstromnetzwerk mit vier Maschen

Die Koeffizientenmatrix wird in Tabelle 3.2 eingetragen. Diese Tabelle besteht aus vier Zeilen und fünf Spalten. Die fünfte Spalte ist für den Vektor der Quellspannungen vorgesehen.

	I_1	I_2	I_3	I_4	U
1	$Z_1 + Z_2 + Z_4$	$-Z_2$	$-Z_4$	0	U_1
2	$-Z_2$	$Z_2 + Z_3 + Z_5$	0	$-Z_5$	$-U_2$
3	$-Z_4$	0	$Z_4 + Z_6 + Z_7$	$-Z_7$	U_3
4	0	$-Z_5$	$-Z_7$	$Z_5 + Z_7 + Z_8$	$-U_4$

Tabelle 3.2 Gleichungssystem nach dem Maschenstromverfahren

In der Hauptdiagonalen stehen jeweils die Summen der Impedanzen aus den einzelnen Maschen. In den Nebendiagonalen stehen die gemeinsamen Impedanzen von zwei Maschen. Wenn zwei Maschen keine gemeinsamen Impedanzen haben, wird in die Tabelle eine 0 eingetragen. Alle Koeffizienten der Nebendiagonalen haben ein negatives Vorzeichen und spiegeln sich an der Hauptdiagonalen der Impedanzmatrix. In Listing 3.17 wird die Koeffizientenmatrix der Zeilen 1 bis 4 und der Spalten 1 bis 4 aus Tabelle 3.2 direkt in ein NumPy-Array übertragen.

```
01 #17_masche4c.py
02 import numpy as np
03 import numpy.linalg
04 U1=230
05 U2=-230
06 U3=230
07 U4=-230
08 Z1=1+2j
09 Z2=2-4j
10 Z3=3+4j
11 Z4=2+5j
12 Z5=1+5j
13 Z6=2+5j
14 Z7=4-5j
15 Z8=1+5j
16 Z=np.array([[Z1+Z2+Z4,-Z2,-Z4, 0],
17              [-Z2,Z2+Z3+Z5, 0,-Z5],
18              [-Z4, 0,Z4+Z6+Z7,-Z7],
19              [0,-Z5,-Z7,Z5+Z7+Z8]])
20 U=np.array([U1,-U2,U3,-U4])
```

```
21  strom=np.linalg.solve(Z,U) #numpy.ndarray
22  for k, I in enumerate(strom,start=1):
23      print("I%d = (%0.2f, %0.2fj)A" %(k,I.real,I.imag))
```

Listing 3.17 Netzwerk mit komplexen Widerständen

Ausgabe

```
I1 = (33.16, -52.04j)A
I2 = (19.63, -49.35j)A
I3 = (20.09, -41.98j)A
I4 = (18.09, -51.66j)A
```

Analyse

In den Zeilen 04 bis 15 stehen die Werte für die Spannungen und Impedanzen des Netzwerks. Die Koeffizientenmatrix `Z` wird in den Zeilen 16 bis 19 definiert. Die Zeilen und Spalten der Matrix sind entsprechend Tabelle 3.2 in einem NumPy-Array `array([[],...,[]])` angeordnet. Für die Berechnung des Lösungsvektors `I` muss noch in Zeile 20 der Inhomogenitätsvektor `U` definiert werden. Die Lösung wird mit der NumPy-Funktion `linalg.solve(Z,U)` in Zeile 21 berechnet. Der Lösungsvektor `strom` enthält die vier Maschenströme `I[0]`, `I[1]`, `I[2]` und `I[3]`.

Mit der Python-Funktion `enumerate(strom)` ist es möglich, die einzelnen Maschenströme innerhalb einer `for`-Schleife auszugeben (Zeile 23). Jeder einzelne Maschenstrom `I` wird mit dem Index `k` gekennzeichnet. Mit jeder Iteration gibt die Funktion `enumerate(strom)` ein Tupel zurück, das den Index `k` und das entsprechende Element `I` des Arrays `strom` enthält.

3.5 Projektaufgabe: Blitzschutzsystem

Für ein quaderförmiges Gebäude sollen die Stromstärken in den Fang- und Ableitungen berechnet werden. Das Gebäude hat eine Länge von 10 m, eine Breite von 5 m und eine Höhe von 3 m. Die Fang- und Ableitungen aus Stahl mit einem Leiterquerschnitt von A = 50 mm^2 werden an den Kanten des Gebäudes angebracht. Für die Draufsicht ergibt sich somit ein Netzwerk mit vier Knoten und acht Widerständen (siehe Abbildung 3.5). Der zeitliche Verlauf eines Blitzstroms kann annähernd durch ein Dreieck mit einer Anstiegszeit von etwa 10 µs und einer Abfallzeit von etwa 1 ms beschrieben werden. Die Maxima der Stromstärken liegen etwa zwischen 10.000 A und 300.000 A.

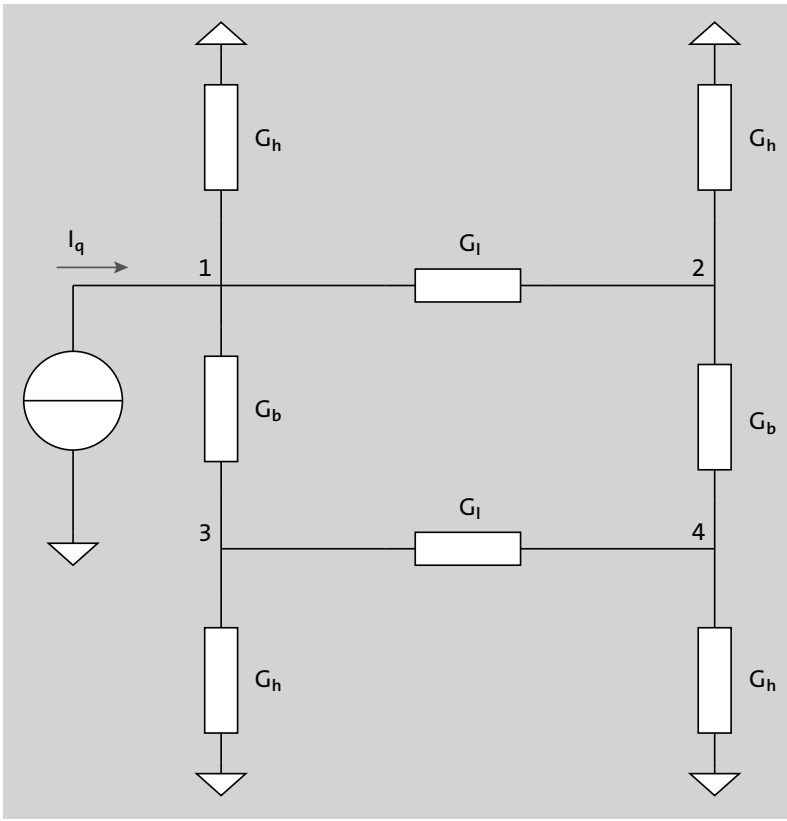

Abbildung 3.5 Ersatzschaltung für ein Blitzschutzsystem

Die Spannungsfälle zwischen den Knoten werden mit dem Knotenpotenzialverfahren berechnet. Das Gleichungssystem können Sie direkt aus der Schaltung ablesen und als Matrixform darstellen:

$$\begin{pmatrix} G_b+G_h+G_l & -G_l & -G_b & 0 \\ -G_l & G_b+G_h+G_l & 0 & -G_b \\ -G_b & 0 & G_b+G_h+G_l & -G_l \\ 0 & -G_b & -G_l & G_b+G_h+G_l \end{pmatrix} \cdot \begin{pmatrix} U_{10} \\ U_{20} \\ U_{30} \\ U_{40} \end{pmatrix} = \begin{pmatrix} I_q \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Für die Leitwerte der Fang- und Ableitungen gilt:

$$G = \frac{\gamma A}{l}$$

Die Stromstärken in den Fang- und Ableitungen berechnen Sie mit dem ohmschen Gesetz aus den Potenzialdifferenzen der Knotenspannungen und den Leitwerten der Fang- und Ableitungen.

Listing 3.18 löst das Gleichungssystem für die vier unbekannten Knotenspannungen mit der NumPy-Funktion `U=linalg.solve(G,I)`:

```
#18_projekt_blitzschutz.py
import numpy as np
Iq=1e5 #Stromstärke des Blitzes in A
g=10   #Leitwert für Stahl S*m/mm^2
A=50   #Leiterquerschnitt in mm^2
l=10   #Länge in m
b=5    #Breite in m
h=3    #Höhe in m
Gh=g*A/h   #Leitwert für Höhe in S
Gl=g*A/l   #Leitwert für Länge in S
Gb=g*A/b   #Leitwert für Breite in S
G=np.array([[Gb+Gh+Gl, -Gl, -Gb, 0],
            [-Gl, Gb+Gh+Gl, 0,-Gb],
            [-Gb, 0, Gb+Gh+Gl,-Gl],
            [ 0,-Gb,-Gl, Gb+Gh+Gl]])
I=np.array([Iq,0,0,0])
U=np.linalg.solve(G,I)
I10=U[0]*Gh
I20=U[1]*Gh
I30=U[2]*Gh
I40=U[3]*Gh
I12=(U[0]-U[1])*Gl
I13=(U[0]-U[2])*Gb
I34=(U[2]-U[3])*Gl
I24=(U[1]-U[3])*Gb
print("--Spannungsfälle der Ableitungen--")
print("Spannung U10: %3.2f V" %U[0])
print("Spannung U20: %3.2f V" %U[1])
print("Spannung U30: %3.2f V" %U[2])
print("Spannung U40: %3.2f V" %U[3])
print("--Stromstärken in den Ableitungen--")
print("Stromstärke I10: %3.2f A" %I10)
print("Stromstärke I20: %3.2f A" %I20)
print("Stromstärke I30: %3.2f A" %I30)
print("Stromstärke I40: %3.2f A" %I40)
print("--Stromstärken in den Fangleitungen--")
print("Stromstärke I12: %3.2f A" %I12)
print("Stromstärke I13: %3.2f A" %I13)
print("Stromstärke I34: %3.2f A" %I34)
print("Stromstärke I24: %3.2f A" %I24)
```

Listing 3.18 Stromverteilung in den Fang- und Ableitungen

Ausgabe

```
--Spannungsfälle der Ableitungen--
Spannung U10: 365.50 V
Spannung U20: 70.86 V
Spannung U30: 122.00 V
Spannung U40: 41.64 V
--Stromstärken in den Ableitungen--
Stromstärke I10: 60917.21 A
Stromstärke I20: 11810.06 A
Stromstärke I30: 20332.79 A
Stromstärke I40: 6939.94 A
--Stromstärken in den Fangleitungen--
Stromstärke I12: 14732.14 A
Stromstärke I13: 24350.65 A
Stromstärke I34: 4017.86 A
Stromstärke I24: 2922.08 A
```

Analyse

Zeile 03 gibt den Spitzenwert des Blitzstroms von 100.000 A vor. Der Querschnitt der Fang- und Ableitungen beträgt in der Regel 50 mm^2 (Zeile 05). Die Zeilen 06 bis 08 legen die Länge, Breite und Höhe des Gebäudes in Metern fest.

In den Zeilen 09 bis 11 werden die Leitwerte der Fang- und Ableitungen berechnet. Die Koeffizientenmatrix der Leitwerte steht in den Zeilen 12 bis 15. In Zeile 16 wird mit dem Inhomogenitätsvektor festgelegt, dass der Blitz in den Knoten 1 einschlägt. Der Lösungsvektor für die Spannungsfälle wird in Zeile 17 mit der NumPy-Funktion `linalg.solve(G,I)` berechnet und der Variablen `U` zugewiesen. Die Berechnung der Ströme in den Ableitungen erfolgt in den Zeilen 18 bis 21. Die Zeilen 22 bis 25 berechnen die Ströme in den Fangleitungen aus den Potenzialdifferenzen.

Die Ausgaben in den Zeilen 26 bis 40 zeigen, dass sehr hohe Ströme mit einer maximalen Stromdichte von etwa 1.218 A/mm^2 fließen können. Diese hohen Stromdichten sind deshalb noch akzeptabel, weil der Strom nur wenige Millisekunden fließt.

3.6 Aufgaben

1. Das Volumen eines Spats (*Parallelepiped*) soll mit einer Determinante und der Funktion `dot(cross(a,b),c)` berechnet werden.
2. Eine Kettenschaltung setzt sich aus drei Spannungsteilern (Längsglied R_1, Querglied R_2) zusammen. Alle Widerstände haben den gleichen Wert von 1 Ω. Die Ausgangsspannung beträgt $U_2 = 1$ V. Berechnen Sie mit der Methode der Kettenparameter die Eingangsspannung U_1 und den Eingangsstrom I_1.

3. Berechnen Sie das dyadische Produkt für:

$$(1 \quad 2 \quad 3 \quad 4) \otimes \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

4. Gegeben ist die erweiterte Koeffizientenmatrix eines linearen Gleichungssystems:

$$\left(\begin{array}{ccc|c} 3 & 2 & 1 & 10 \\ 1 & 2 & 3 & 14 \\ 2 & 1 & 4 & 16 \end{array}\right)$$

   Lösen Sie dieses Gleichungssystem mit der NumPy-Funktion `solve()`. Die Koeffizientenmatrix und der Inhomogenitätsvektor sollen durch Slicing aus der erweiterten Koeffizientenmatrix ermittelt werden.

5. Ein lineares Gleichungssystem mit sehr vielen Unbekannten (50 bis 1000) soll mit der NumPy-Funktion `solve()` gelöst werden. Erzeugen Sie die Koeffizientenmatrix und den Inhomogenitätsvektor mit der NumPy-Funktion `random.normal()`. Testen Sie die Grenzen von `solve()`, indem Sie die Anzahl der Unbekannten schrittweise erhöhen.

6. Berechnen Sie alle Maschenströme für die Kettenschaltung aus Aufgabe 2 mit dem Maschenstromverfahren. Die Eingangsspannung beträgt 13 V.

7. Berechnen Sie alle Knotenspannungen für die Kettenschaltung aus Aufgabe 2 mit dem Knotenpotenzialverfahren. Der Eingangsstrom hat einen Wert von 8 A.