Die Gefahr, dass der Computer so wird wie der Mensch, ist nicht so groß wie die Gefahr, dass der Mensch so wird wie der Computer.

Konrad Zuse

3 NumPy und Matplotlib

In der Physik werden die Naturgesetze häufig in Form von Vektorgleichungen formuliert. Ein klassisches Beispiel ist das newtonsche Gesetz $\vec{F} = m\vec{a}$, mit dem wir uns in Kapitel 7 näher beschäftigen werden, oder die Definition der mechanischen Arbeit W über das Skalarprodukt $W = \vec{F} \cdot \vec{s}$. Nehmen wir einmal an, wir wollen für zwei vorgegebenen Vektoren $\vec{F}$ und $\vec{s}$ dieses Skalarprodukt ausrechnen. In Python könnte das zum Beispiel wie folgt aussehen, wenn wir die Vektoren durch Tupel repräsentieren:

```
F = (1.0, -0.3, 0.7)
s = (-0.2, 0.4, -0.9)
W = 0
for Fi, si in zip(F, s):
    W += Fi * si
print(W)
```

Auch wenn dieser Code völlig korrekt funktioniert, so ist es doch unschön, dass man drei Zeilen Code benötigt, um die Berechnung eines Skalarproduktes umzusetzen.

Ein ähnliches Problem hatten wir bei der Berechnung des relativen Fehlers der Näherung $\sin(x) \approx x$ im vorherigen Kapitel zu lösen. Hier musste eine mathematische Funktion auf jedes Element einer Liste angewendet werden. Auch hierzu haben wir eine `for`-Schleife verwendet, und es wurde eine neue Liste mit dem Ergebnis produziert (siehe Programm 2.3).

Probleme dieser Art tauchen bei physikalischen Berechnungen sehr häufig auf. Um diese Art der Berechnung einfacher und schneller durchführen zu können, gibt es die Bibliothek NumPy [1]. Die Bibliothek NumPy bietet Datentypen und die benötigten Rechenoperationen an, mit denen beispielsweise Matrix- und Vektorrechnungen sehr effizient durchgeführt werden können. Auch das Erstellen von Wertetabellen, wie im oben erwähnten Programm 2.3 lässt sich sehr effizient mit NumPy lösen. Man kann daher NumPy ohne Zweifel als die wichtigste Bibliothek für das wissenschaftliche Rechnen mit Python bezeichnen.

Darüber hinaus gibt es noch die Bibliothek SciPy [2], die viele weitere spezielle Funktionen für das wissenschaftliche Rechnen zur Verfügung stellt. Die Funktionsvielfalt dieser Bibliothek ist so groß, dass es kaum möglich ist, diese sinnvoll an einem Stück vorzustellen. Stattdessen werden wir in den nachfolgenden Kapiteln anhand der

O. Natt, *Physik mit Python*, https://doi.org/10.1007/978-3-662-66454-4_3

Import von Modulen mit `import modul as name`

Die `as`-Klausel beim Import eines Moduls dient dazu, das Modul anschließend unter dem angegebenen Namen anzusprechen. Für einige häufig benutzte Module gibt es Empfehlungen, wie diese Module importiert werden sollten. Ich rate dringend dazu, diesen Empfehlungen zu folgen, da Sie dann Codebeispiele aus der Dokumentation der entsprechenden Module direkt übernehmen können und Ihr Code leichter zu verstehen ist.

physikalischen Beispiele immer wieder einzelne Funktionen aus der Bibliothek SciPy diskutieren.

Häufig möchte man das Ergebnis einer Berechnung oder die Auswertung eines Experiments grafisch darstellen. Dazu gibt es einige geeignete Bibliotheken für die Programmiersprache Python. Die am weitesten verbreitete Grafikbibliothek ist sicherlich Matplotlib [3]. Die Stärke von Matplotlib liegt darin, dass man nicht nur relativ einfach Grafiken auf dem Bildschirm darstellen kann, sondern es lassen sich auch qualitativ sehr hochwertige Grafiken für Veröffentlichungen erzeugen. Da Matplotlib an vielen Stellen auf Datentypen der Bibliothek NumPy zurückgreift, werden wir hier zunächst die grundlegenden Funktionen von NumPy diskutieren, bevor wir mit Abschn. 3.16 in Matplotlib einsteigen.

3.1 Eindimensionale Arrays

Den wichtigsten Datentyp in NumPy stellt das **Array** dar. Der Datentyp heißt dabei eigentlich `ndarray`, wobei sich die ersten beiden Buchstaben »nd« auf »*n*-dimensional« beziehen. Wir sprechen der Einfachheit halber aber einfach von Arrays. Ein 1-dimensionales Array ist ähnlich wie eine Liste. Der große Unterschied ist, dass die einzelnen Elemente einer Liste beliebige Python-Objekte sein können. Dagegen müssen bei einem Array alle Elemente vom gleichen Typ sein. Ein weiterer Unterschied zwischen Listen und Arrays besteht darin, dass man bei einem Array die Größe nicht mehr nachträglich ändern kann. Man kann insbesondere keine Elemente hinzufügen. Das sind zunächst einmal Einschränkungen gegenüber der Liste. Diese Einschränkungen erlauben es aber, mit Arrays viel effizienter zu arbeiten als mit Listen.

Es wird empfohlen, das Modul `numpy` stets in der Form

```
import numpy as np
```

zu importieren. Diese spezielle Art der Import-Anweisung erlaubt es, auf die Elemente des Moduls mit dem Namen `np` anstelle von `numpy` zuzugreifen. Der Bezeichner `np` ist im Prinzip beliebig wählbar. Es ist allerdings gängige Praxis, das Modul `numpy` immer unter dem Namen `np` zu importieren.

Um ein Array zu erzeugen, kann man mit `np.array` eine Liste oder ein Tupel in ein Array konvertieren.

```
>>> import numpy as np
>>> a = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
```

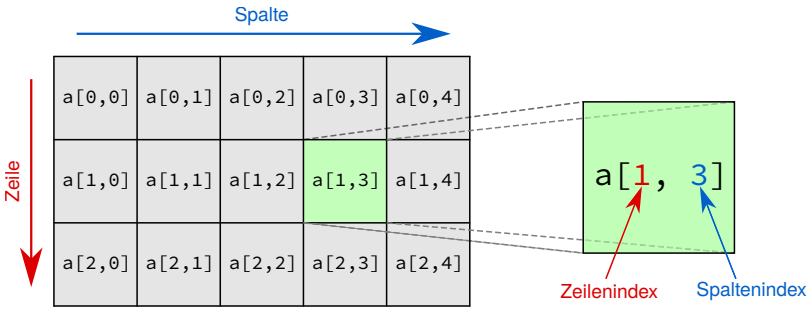

Abb. 3.1: Indizierung von 2-dimensionalen Arrays. *Die Array-Indizes folgen der Konvention der Matrixrechnung in der Mathematik. Der erste Index gibt die Zeile an und der zweite Index die Spalte.*

Anschließend kann man verschiedene Eigenschaften des Arrays abfragen:

```
>>> a.ndim
1
>>> a.size
6
```

Im obigen Beispiel handelt es sich um ein 1-dimensionales Array (`a.ndim` ergibt 1) mit sechs Elementen (`a.size` ergibt 6). Die einzelnen Elemente eines Arrays lassen sich genauso indizieren, wie die Einträge von Listen:

```
>>> a[4]
5.0
```

3.2 Mehrdimensionale Arrays

Mit NumPy kann man auch mehrdimensionale Arrays erzeugen. Das folgende Beispiel erzeugt ein 2-dimensionales Array.

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a.ndim
2
>>> a.size
12
>>> a.shape
(3, 4)
>>> len(a)
3
```

Ein 2-dimensionales Array kann man sich, wie in Abb. 3.1 dargestellt, als ein Raster von Zahlen vorstellen. In unserem Beispiel handelt es sich um zwölf Zahlen (`a.size`), die in einem 3×4-Raster angeordnet sind (`a.shape`). Dabei stellt die erste Zahl die Anzahl der Zeilen dar, und die zweite Zahl gibt die Anzahl der Spalten an. Die Python-Funktion

Tab. 3.1: Wertebereich der ganzzahligen Datentypen in NumPy.

Datentyp	Wertebereich von	bis
int8	$-2^7 = -128$	$2^7 - 1 = 127$
int16	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
int32	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
int64	$-2^{63} \approx -9{,}22 \cdot 10^{18}$	$2^{63} - 1 \approx 9{,}22 \cdot 10^{18}$
uint8	0	$2^8 - 1 = 255$
uint16	0	$2^{16} - 1 = 65535$
uint32	0	$2^{32} - 1 = 4294967295$
uint64	0	$2^{64} - 1 \approx 1{,}84 \cdot 10^{19}$

`len` liefert immer die Größe der ersten Dimension also die Anzahl der Zeilen zurück, es gilt also `len(a) == a.shape[0]`. Warum es sinnvoll ist, dass die Funktion `len` gerade die Größe der *ersten* Dimension zurückgibt, wird in Abschn. 3.9 deutlich. Auf ein bestimmtes Element des Arrays, kann man zugreifen, indem man den Zeilenindex und den Spaltenindex mit einem Komma getrennt in eckigen Klammern angibt. Wenn wir das Beispiel von oben fortsetzen, erhalten wir für das Element in der zweiten Zeile und vierten Spalte den Zahlenwert 8.

```
>>> a[1, 3]
8
```

Man kann in NumPy auch mit höherdimensionalen Arrays arbeiten. Unter einem dreidimensionalen Array kann man sich einen Block von Zahlen vorstellen, in dem mehrere 2-dimensionale Arrays gewissermaßen übereinander gestapelt sind. Bei vier oder mehr Dimensionen versagt dann allerdings die geometrische Veranschaulichung.

3.3 Datentypen in NumPy

Wir haben bereits erwähnt, dass bei einem Array alle Elemente den gleichen Datentyp haben müssen. Den Datentyp eines Arrays findet man über die Eigenschaft `dtype` heraus.

```
>>> import numpy as np
>>> a = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
>>> a.dtype
dtype('float64')
```

Das Array hat den Datentyp `np.float64` und verhält sich damit fast genauso wie die gewöhnlichen Gleitkommazahlen in Python. Eine Übersicht der verfügbaren Datentypen ist in Tab. 3.1 und 3.2 angegeben. Wenn Sie ein Array mit einem bestimmten Datentyp erstellen wollen, dann können Sie dies wie folgt machen:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4, 5, 6], dtype=np.int8)
```

Tab. 3.2: Wertebereich der Gleitkomma-Datentypen in NumPy.

Datentyp	Größte positive Zahl	Gültige Dezimalstellen
float16	$6{,}5 \cdot 10^4$	≈ 3
float32	$3{,}4 \cdot 10^{34}$	≈ 6
float64	$1{,}8 \cdot 10^{308}$	≈ 15
complex64	Real- und Imaginärteil jeweils wie float32	
complex128	Real- und Imaginärteil jeweils wie float64	

```
>>> a.dtype
dtype('int8')
```

Bei der Wahl des Datentyps muss man sehr vorsichtig sein und den erlaubten Wertebereich beachten. Das sieht man an dem folgenden Beispiel:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4, 5, 6], dtype=np.uint8)
>>> a[2] = 300
>>> print(a)
[ 1  2 44  4  5  6]
```

Bei dem Versuch, die Zahl 300 in einem Array vom Typ `uint8` zu speichern, kommt es zu einem sogenannten Überlauf, und es wird stattdessen die Zahl $300 - 256 = 44$ gespeichert. Beachten Sie bitte die folgende Regel, die eine wichtige Konsequenz hat, die gerne beim Programmieren übersehen wird und zu schwer auffindbaren Fehlern führt:

Achtung!

Wenn Sie aus einer Liste mit der Funktion `np.array` ein Array erzeugen und keinen Datentyp mit `dtype=` angeben, so versucht NumPy, den Datentyp des Arrays aus den in der Liste enthaltenen Daten abzuleiten.

Wir demonstrieren dies an einem Beispiel:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = np.array([1.0, 2, 3])
>>> a[2] = 42.7
>>> b[2] = 42.7
>>> a
array([ 1,  2, 42])
>>> b
array([ 1. ,  2. , 42.7])
```

Das Array `a` wird aus einer Liste erzeugt, die nur aus ganzen Zahlen besteht, und erhält darum einen ganzzahligen Datentyp. Das Array `b` wird aus einer Liste erzeugt, die eine Gleitkommazahl enthält, und erhält darum einen Gleitkommatyp. Beim Versuch, einer Zelle des Arrays `a` die Zahl 42,7 zuzuweisen, werden die Nachkommastellen einfach

abgeschnitten. Wir überprüfen den Datentyp der beiden Arrays, indem wir das Beispiel fortsetzen:

```
>>> a.dtype
dtype('int64')
>>> b.dtype
dtype('float64')
```

Sie erkennen, dass NumPy für ganze Zahlen den Datentyp `int64` wählt.

Wenn Sie einfach nur sicherstellen möchten, dass ein Array einen ganzzahligen Datentyp besitzt, dann können Sie als Datentyp einfach `int` angeben. Es wird dann normalerweise der größte ganzzahlige Datentyp genommen, der vom Prozessor des Rechners direkt unterstützt wird. Auf neueren PCs ist das meistens der Datentyp `np.int64` bei etwas älteren 32-Bit-Computern ist es der Datentyp `np.int32`. Analog können Sie die Datentypen `float` oder `complex` verwenden, um sicherzustellen, dass es sich um Gleitkommazahlen beziehungsweise um komplexe Zahlen handeln soll, ohne explizit die Speicherbreiten, also die Anzahl der pro Zahl verwendeten Bits anzugeben.

3.4 Rechnen mit Arrays

Mit Arrays kann man Rechenoperationen ausführen. Wir besprechen diese Rechenoperationen hier nur an 1-dimensionalen Arrays, sie lassen sich aber auf mehrdimensionale Arrays völlig analog übertragen. Im Modul `numpy` ist eine Reihe Funktionen enthalten, die den gleichen Namen haben wie die entsprechenden Funktionen aus dem Modul `math`. Dies sind beispielsweise die trigonometrischen Funktionen, Wurzeln, Logarithmen und die Exponentialfunktion. Wenn man diesen Funktionen ein Array übergibt, dann wird die Funktion für jedes Element des Arrays ausgewertet, und es entsteht ein neues Array, das die gleiche Größe hat wie das ursprüngliche Array:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> b = np.sqrt(a)
>>> print(b)
[1.         1.41421356 1.73205081 2.        ]
```

Ein Array kann man mit einer Zahl multiplizieren. Dabei wird jedes Element mit der Zahl multipliziert.[1]

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> b = 5 * a
>>> print(b)
[ 5 10 15 20]
```

Man kann zu einem Array eine Zahl addieren. Auch hierbei wird die Zahl zu jedem Element des Arrays addiert:

[1] Hier unterscheiden sich Arrays ganz wesentlich von Listen. Wenn `a` eine Liste ist, dann erzeugt `5 * a` eine Liste, die aus einer 5-fachen Wiederholung der Liste `a` besteht.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> b = a + 10
>>> print(b)
[11 12 13 14]
```

Interessant ist, was passiert, wenn man versucht, zwei Arrays der gleichen Größe zu addieren oder miteinander zu multiplizieren. Wir probieren dies an einem Beispiel aus:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([10, 20, 30, 40])
>>> c = a + b
>>> print(c)
[11 22 33 44]
>>> d = a * b
>>> print(d)
[ 10  40  90 160]
```

Sie können erkennen, dass auch diese Operationen elementweise ausgeführt werden.

3.5 Erzeugen von Arrays

In den bisherigen Beispielen wurde ein Array immer aus einer Liste erzeugt. Wir werden im Folgenden einige Funktionen kennenlernen, mit denen man spezielle Sorten von Arrays erzeugen kann.

Die Funktion `np.arange` erzeugt ein 1-dimensionales Array, das aus einer Folge von Zahlen mit fester Schrittweite besteht.[2] Die Funktion wird ähnlich aufgerufen wie die `range`-Funktion:

```
np.arange(Anfang, Ende, Schrittweite)
```

Wir demonstrieren das wieder an einem Beispiel:

```
>>> import numpy as np
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]
```

In manchen Fällen möchte man einen Start- und einen Endwert vorgeben und die Anzahl der Punkte. Das geht am einfachsten mit der Funktion `np.linspace`:

```
np.linspace(Anfang, Ende, Anzahl)
```

Im Unterschied zu `np.arange` wird bei `np.linspace` der Endwert mit in das Array aufgenommen, wie das folgende Beispiel zeigt:

[2] Wenn alle Argumente ganzzahlig sind, erzeugt `np.arange` ein Array mit dem Datentyp `int`, ansonsten wird der Datentyp `float` gewählt. Alle anderen Funktionen, die im Folgenden diskutiert werden, erzeugen immer Arrays mit dem Datentyp `float`, wenn kein anderer Datentyp explizit mit `dtype=` angefordert wird.

```
>>> import numpy as np
>>> a = np.linspace(5, 20, 6)
>>> a
array([ 5.,  8., 11., 14., 17., 20.])
```

Manchmal benötigt man auch ein Array, das komplett mit Nullen initialisiert ist:

```
>>> import numpy as np
>>> a = np.zeros(5)
>>> print(a)
[0. 0. 0. 0. 0.]
>>> a = np.zeros((3, 4))
>>> print(a)
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Sie erkennen an diesem Beispiel, dass man der Funktion `np.zeros` auch ein Tupel übergeben kann, um damit ein mehrdimensionales Array zu erzeugen. Eine ähnliche Funktion zum Erzeugen von Arrays ist `np.empty`, die das Array aber nicht mit Nullen füllt, sodass es im Allgemeinen irgendwelche unsinnigen Werte enthält. Verwenden Sie `np.empty` also nur, wenn anschließend die Array-Elemente anderweitig initialisiert werden.

Wir können nun die Berechnung des prozentualen Fehlers der Näherung $\sin(x) \approx x$ sehr elegant formulieren. Das Programm 3.1 ist nicht nur kürzer, sondern wird auch deutlich schneller ausgeführt als das Programm 2.3, das mit Listen gearbeitet hat.

Programm 3.1: *NumpyMatplotlib/naeherung_sin3.py*

```
 9 import numpy as np
10
11 # Lege ein Array der Winkel in Grad an.
12 winkel = np.arange(5, 95, 5)
13
14 # Wandle die Winkel in das Bogenmaß um.
15 x = np.radians(winkel)
16
17 # Berechne die relativen Fehler.
18 fehler = 100 * (x - np.sin(x)) / np.sin(x)
19
20 # Gib das Ergebnis aus.
21 print(winkel)
22 print(fehler)
```

3.6 Indizierung von Array-Ausschnitten (Array-Slices)

In Abschn. 2.12 haben wir bereits besprochen, wie man beispielsweise aus einer Liste eine bestimmte Teilliste auswählen kann. Das gleiche Verfahren funktioniert auch bei Arrays. So kann man zum Beispiel mit

```
a[:10]
```

die ersten zehn Elemente des Arrays `a` auswählen. Bei mehrdimensionalen Arrays kann man die Slice-Operation auf jede der Dimensionen anwenden. So erzeugt beispielsweise

```
a[:10, 5:8]
```

aus einem 2-dimensionalen Array `a` ein Teil-Array, das aus den ersten zehn Zeilen und aus den Spalten 5 bis 7 besteht. Im Gegensatz zu den Listen wird bei Arrays aber nur eine neue **Ansicht** (engl. view) des bestehenden Arrays erzeugt. Was damit gemeint ist, können Sie an dem folgenden Beispiel erkennen: Wir erzeugen zunächst ein 3×4-Array. Aus diesem Array wird ein Ausschnitt ausgewählt, der aus zwei Zeilen und drei Spalten besteht. Diesen Ausschnitt nennen wir `b`.

```
1  >>> import numpy as np
2  >>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
3  >>> b = a[:2, 1:]
4  >>> print(b)
5  [[2 3 4]
6   [6 7 8]]
```

Nun weisen wir einem Element dieses Teil-Arrays einen neuen Wert zu.

```
7  >>> b[0, 1] = 50
```

Was passiert, wenn wir nun das ursprüngliche Array `a` ausgeben, zeigt der folgende Code:

```
8  >>> print(a)
9  [[ 1  2 50  4]
10  [ 5  6  7  8]
11  [ 9 10 11 12]]
```

Sie können erkennen, dass sich das ursprüngliche Array ebenfalls verändert hat.

Warum verhalten sich Arrays auf diese Art? Die Bibliothek NumPy wurde dazu konzipiert, mit großen Datenmengen umzugehen. Wenn man mit großen Datenmengen hantiert, ist es unbedingt zu vermeiden, diese Daten unnötig zu kopieren. Aus diesem Grund wird versucht, alle Operationen so durchzuführen, dass keine Daten im Speicher des Computers hin und her kopiert werden müssen. Ein Array besteht grob gesagt aus einem Bereich des Hauptspeichers des Computers, in dem die Zahlen abgelegt sind, und einer Zuordnungsvorschrift, wie die Indizes den Speicherzellen zugeordnet sind. Wenn man einen Ausschnitt eines Arrays auf die oben gezeigte Art auswählt, verweist das neue Array auf den gleichen Speicherbereich wie das alte, nur dass die Zuordnungsvorschrift der Indizes an die Speicherzellen angepasst wird. Wenn Sie wirklich eine Kopie der Daten wollen, dann müssen Sie das mit der Methode `copy` explizit anfordern. Wenn Sie im obigen Beispiel die Zeile 3 durch

```
3  >>> b = a[:2, 1:].copy()
```

ersetzen, dann wirken sich Änderungen von `b` nicht mehr auf `a` aus.

3.7 Indizierung mit ganzzahligen Arrays

Anstelle eines Ausschnitts lassen sich auch einzelne Elemente adressieren, indem man ein zweites Array benutzt, das die entsprechenden Indizes gespeichert hat. Das wird an dem folgenden Beispiel deutlich:

```
>>> import numpy as np
>>> a = np.arange(0, 40, 5)
>>> idx = np.array([2, 3, 5, 7])
>>> b = a[idx]
>>> print(b)
[10 15 25 35]
>>> print(a[2], a[3], a[5], a[7])
10 15 25 35
```

Sie können erkennen, dass das Array `b` nun die Einträge des Arrays `a` enthält, dessen Indizes im Array `idx` gespeichert waren. Dabei ist wichtig, dass das indizierende Array, hier also `idx` unbedingt einen ganzzahligen Datentyp haben muss.

Bei der Indizierung mit einem ganzzahligen Array werden die Elemente des ursprünglichen Arrays immer kopiert, während bei den Array-Slices eine Ansicht des ursprünglichen Arrays erzeugt wird. Unabhängig davon kann man aber die Indizierung mit ganzzahligen Arrays dazu benutzen, um einzelne Elemente eines Arrays zu verändern. Wir setzen das vorherige Beispiel fort:

```
>>> a[idx] = 0
>>> print(a)
[ 0  5  0  0 20  0 30  0]
```

Hier sind also die Elemente von `a`, deren Indizes im Array `idx` enthalten sind, auf null gesetzt worden.

3.8 Indizierung mit booleschen Arrays

Es gibt noch eine weitere sehr ähnliche Methode, mit der man auf bestimmte Teile eines Arrays zugreifen kann. Dazu betrachten wir zuerst, was passiert, wenn man die Vergleichsoperatoren auf Arrays anwendet.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4, 5])
>>> b = np.array([2, 1, 6, 2, 4])
>>> b < a
array([False,  True, False,  True,  True])
```

Sie sehen, dass ein Vergleich zweier Arrays gleicher Größe ein neues Array erzeugt, das mit den entsprechenden Wahrheitswerten gefüllt ist. Man kann auch ein Array mit einer einzelnen Zahl vergleichen und erhält ein entsprechendes Array.

```
>>> import numpy as np
>>> b = np.array([2, 1, 6, 2, 4])
>>> b < 3
array([ True,  True, False,  True, False])
```

Die so erzeugten booleschen Arrays haben eine interessante Anwendung. Man kann ein Array mit einem booleschen Array indizieren und erhält alle Elemente des Arrays, bei denen das boolesche Array den Wert True enthält.

```
>>> import numpy as np
>>> b = np.array([2, 1, 6, 2, 4])
>>> idx = b < 3
>>> b[idx]
array([2, 1, 2])
```

Diese Art der Indizierung wird häufig verwendet, wenn man Arrays verändern möchte. Nehmen wir das folgende Beispiel: Sie haben ein Array mit Messdaten und möchten für eine Auswertung alle Messwerte, die kleiner als ein bestimmter Wert sind, auf null setzen. Um dies zu erreichen, können Sie den folgenden Code verwenden:

```
1 >>> import numpy as np
2 >>> a = np.array([1.5, 0.2, 33.7, 0.02, 5.2, 7.4])
3 >>> a[a < 0.5] = 0
4 >>> print(a)
5 [ 1.5  0.  33.7  0.   5.2  7.4]
```

Das Schöne an dieser Art der Indizierung ist, dass der Programmcode fast intuitiv verständlich ist: In Zeile 3 werden alle Elemente von `a`, deren Zahlenwert kleiner als 0,5 ist, auf null gesetzt.

Bitte beachten Sie, dass auch hier, genau wie bei der Indizierung mit ganzzahligen Arrays, die Werte in das neue Array kopiert werden. Es wird also auch hier keine Ansicht erzeugt.

3.9 Ausgelassene Indizes

Bei der Indizierung eines mehrdimensionalen Arrays kann man auch weniger Indizes angeben, als das Array Dimensionen hat. In diesem Fall wird für die verbleibenden Indizes stillschweigend der Doppelpunkt `:` angenommen und damit alle Elemente der verbleibenden Dimensionen ausgewählt. Für ein 2-dimensionales Array `a` ist der Ausdruck `a[1]` also gleichbedeutend mit `a[1, :]`, wie das folgende Beispiel zeigt:

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[1]
array([4, 5, 6])
>>> a[1, :]
array([4, 5, 6])
```

Die Konvention für ausgelassene Indizes erklärt auch das in Abschn. 3.2 erwähnte Verhalten der Funktion `len`. Die Funktion `len` gibt die Anzahl der möglichen Werte

für den Index zurück, wenn man das mehrdimensionale Array mit nur einem Index indiziert.

Diese Art der Indizierung ist sehr hilfreich, um gut lesbaren Programmcode zu erzeugen. Stellen Sie sich vor, dass wir in einem Programm die Bewegung eines Körpers im 3-dimensionalen Raum beschreiben möchten. Dazu haben wir die Koordinaten des Körpers zu 100 verschiedenen Zeitpunkten gegeben. Wir können nun diese Koordinaten in einem 100×3-Array speichern, dem wir den Namen `ortsvektoren` geben. In diesem Fall können wir beispielsweise mit `ortsvektoren[0]` direkt auf den nullten Ortsvektor zugreifen und wir können mit `ortsvektoren[:, 0]` für alle Ortsvektoren auf die nullte Komponente zugreifen. Wir könnten uns aber auch dafür entscheiden, dieselben Daten in einem 3×100-Array zu speichern. Dann sollte man das Array vielleicht besser `koordinaten` nennen, denn mit `koordinaten[0]` erhält man den Zeitverlauf der nullten Koordinate und mit `koordinaten[:, 0]` erhält man alle Koordinaten zum nullten Zeitpunkt.

3.10 Logische Operationen auf Arrays

Wir haben in Abschn. 3.8 bereits boolesche Arrays kennen gelernt. Mit den folgenden Zeilen erzeugen wir aus einem Array `a` ein boolesches Array `b`, das bei den Einträgen, den Wert `True` enthält, wo die angegebene Bedingung `a < 20` erfüllt ist.

```
>>> import numpy as np
>>> a = np.arange(0, 30, 5)
>>> b = a < 20
>>> print(b)
[ True  True  True  True False False]
```

Nehmen wir an, wir möchten ein dir Bedingung $a < 20$ durch die Bedingung $5 < a < 20$ ersetzen. Die folgende Umsetzung erscheint naheliegend, führt jedoch zu einer Fehlermeldung:

```
>>> b = a > 5 and a < 20
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

Warum erhalten wir hier diese Fehlermeldung? Dazu erinnern wir uns an die Regeln für Wahrheitswerte von Objekten aus Abschn. 2.14. Die Funktion `bool` muss für ein Objekt genau einen Wahrheitswert zurückliefern. Welcher sollte das sein? Ein Array könnte `True` sein, wenn *alle* Einträge des Arrays `True` sind oder wenn *mindestens ein* Eintrag `True` ist. Man könnte das Array auch genauso behandeln, wie die anderen Aufzählungstypen und ihm den Wahrheitswert `True` genau dann zuordnen, wenn das Array nicht leer ist. Die Entwickler von NumPy haben sich dafür entschieden, mögliche Fehlinterpretationen zu vermeiden, indem die Umwandlung eines Arrays in einen booleschen Wert einen Fehler verursacht. Deswegen funktionieren die logischen Operatoren `and`, `or` und `not` nicht mit Arrays.

Stattdessen werden für Arrays Operatoren benutzt, die in Python normalerweise bitweise logische Operationen ausführen. Diese wurden in Kapitel 2 nicht behandelt, weil sie vergleichsweise selten benötigt werden. Bei Arrays bewirken diese Operatoren,

Tab. 3.3: Elementweise logische Operatoren. *Anstelle der üblichen booleschen Operatoren in Python können die folgenden Operatoren verwenden, um logische Operationen elementweise auf Arrays anzuwenden.*

Elementweiser Op.	Boolescher Op.	Bedeutung
`&`	`and`	und
`|`	`or`	oder
`~`	`not`	nicht
`^`		exklusives oder

die in Tab. 3.3 aufgeführt sind, eine elementweise logische Verknüpfung Das Beispiel von oben kann mit diesen Operatoren wie folgt korrekt umgesetzt werden:

```
>>> import numpy as np
>>> a = np.arange(0, 30, 5)
>>> b = (a > 5) & (a < 20)
>>> print(b)
[False False  True  True False False]
```

Es ist an dieser Stelle wichtig, dass die Ausdrücke `a > 5` und `a < 20` in Klammern gesetzt werden. Das liegt daran, dass die Operatoren `&`, `|`, `~` und `^` in der Operatorrangfolge vor den Vergleichsoperatoren kommen. Ohne die Klammern wäre der Ausdruck also gleichbedeutend mit `a > (5 & a) < 20`, was offensichtlich keinen Sinn ergibt. Die vollständige Auflistung der Rangfolge aller Operatoren in Python finden Sie in der Python-Dokumentation [4].

3.11 Mehrfache Zuweisungen mit Arrays (Unpacking)

Wir haben bereits in Abschn. 2.11 mehrfache Zuweisungen mit Tupeln und Listen kennengelernt. Völlig analog kann man auch mehrfache Zuweisungen mit Arrays durchführen. Dazu muss die Anzahl der Variablen mit der Größe der ersten Dimension des Arrays übereinstimmen. Das folgende Beispiel zeigt, wie man auf diese Art die Zeilen eines 2-dimensionalen Arrays auf unterschiedliche Variablen verteilt.

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x, y, z = a
>>> x
array([1, 2, 3])
>>> y
array([4, 5, 6])
>>> z
array([7, 8, 9])
```

Bitte beachten Sie auch hier, dass die Variablen `x`, `y` und `z` Ansichten des ursprünglichen Arrays `a` sind. Wenn Sie nachträglich einen Eintrag beispielsweise mit `z[1] = 10` verändern, so ändern Sie damit auch das Array `a`.

3.12 Broadcasting

Vielleicht haben Sie sich schon gewundert, dass Operatoren wie `+` oder `*` sowohl mit zwei Arrays gleicher Größe als auch mit einem Array und einer Zahl funktionieren. Der dahinterstehende Mechanismus wird als **Broadcasting** bezeichnet. Broadcasting beschreibt, wie NumPy bei arithmetischen Operationen mit Arrays unterschiedlicher Größe umgeht. Der Broadcasting-Mechanismus wird immer dann angewendet, wenn eine arithmetische Operation auf zwei Arrays mit unterschiedlicher Größe wirkt.

Den Broadcasting-Mechanismus macht man sich am besten an einem Beispiel klar. Nehmen wir an, dass wir ein 5-dimensionales $7 \times 2 \times 1 \times 6 \times 4$-Array `A` und ein 3-dimensionales $3 \times 1 \times 4$-Array `B` addieren. Um den Mechanismus des Broadcastings zu verstehen, schreiben wir die Größe der Arrays rechtsbündig untereinander.

A	:	7	×	2	×	1	×	6	×	4
B	:					3	×	1	×	4

Im ersten Schritt ergänzt NumPy intern das Array, das weniger Dimensionen hat, von links durch zusätzliche Dimensionen der Größe 1. Die Anzahl der Elemente des Arrays ändert sich dadurch nicht. In unserem Beispiel sieht das dann wie folgt aus:

A	:	7	×	2	×	1	×	6	×	4
B	:	1	×	1	×	3	×	1	×	4

Nun wird die Größe der beiden Arrays spaltenweise verglichen. Wenn beide Arrays in einer Dimension gleich groß sind, ist nichts weiter zu unternehmen. Wenn eines der beiden Arrays in einer Dimension die Größe 1 hat, wird dieses Array durch Wiederholung in dieser Dimension auf die Größe des anderen Arrays vergrößert. Falls sich die Größen der Arrays in einer Dimension unterscheiden, aber keine der Größen gleich 1 ist, dann sind die Arrays nicht kompatibel miteinander und es kommt zu einer Fehlermeldung. Falls kein solcher Fehler aufgetreten ist, dann haben die beiden Arrays am Ende des Vorgangs die gleiche Größe und werden elementweise addiert. In unserem Beispiel ergibt sich ein Array der folgenden Größe:

A	:	7	×	2	×	1	×	6	×	4
B	:					3	×	1	×	4
A + B	:	7	×	2	×	3	×	6	×	4

Wir wollen das Broadcasting an einigen Beispielen demonstrieren. In NumPy kann man ein 3×4-Array und ein 1-dimensionales Array der Größe 4 oder ein 2-dimensionales Array der Größe 1×4 addieren:

$$\underset{(3\times 4)}{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}} + \underset{(1\times 4)\text{ bzw. }(4)}{\begin{bmatrix} 10 & 20 & 30 & 40 \end{bmatrix}} = \underset{(3\times 4)}{\begin{bmatrix} 11 & 22 & 33 & 44 \\ 15 & 26 & 37 & 48 \\ 19 & 30 & 41 & 52 \end{bmatrix}} \tag{3.1}$$

Zu dem 3×4-Array kann man auch ein 3×1-Array addieren:

$$\underset{(3\times 4)}{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}} + \underset{(3\times 1)}{\begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}} = \underset{(3\times 4)}{\begin{bmatrix} 11 & 12 & 13 & 14 \\ 25 & 26 & 27 & 28 \\ 39 & 40 & 41 & 42 \end{bmatrix}} \tag{3.2}$$

Dagegen kann man zu einem 3 × 4-Array kein 1-dimensionales Array der Größe 3 addieren und auch kein 1 × 3-Array. Diese Addition scheitert, weil die jeweils letzte Dimension mit vier bzw. drei Elementen nicht kompatibel ist.

$$\underset{(3\times 4)}{\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\end{bmatrix}} + \underset{(1\times 3)\text{ bzw. }(3)}{\begin{bmatrix}10 & 20 & 30\end{bmatrix}} \longrightarrow \texttt{ValueError} \tag{3.3}$$

Das nächste Beispiel zeigt ein Broadcasting, das häufig benutzt wird, wenn man die Elemente zweier Arrays paarweise miteinander kombinieren will:

$$\underset{(1\times 4)\text{ bzw. }(4)}{\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}} + \underset{(3\times 1)}{\begin{bmatrix}10\\ 20\\ 30\end{bmatrix}} = \underset{(3\times 4)}{\begin{bmatrix}11 & 12 & 13 & 14\\ 21 & 22 & 23 & 24\\ 31 & 32 & 33 & 34\end{bmatrix}} \tag{3.4}$$

Wir werden im Laufe der folgenden Kapitel viele Beispiele kennenlernen, in denen man mithilfe des Broadcastings physikalische Probleme nicht nur sehr effizient, sondern auch sehr übersichtlich darstellen kann.

Achtung!

Folgende Punkte sollten beim Broadcasting von Arrays beachtet werden:

- Bei der Addition und Multiplikation von Arrays spielt die Reihenfolge der Operanden keine Rolle.
- In NumPy sind 2-dimensionale Arrays der Größe $1 \times n$, der Größe $n \times 1$ und 1-dimensionale Arrays der Größe n unterschiedliche Datentypen.
- Beim Broadcasting werden 1-dimensionale Arrays genauso behandelt wie 2-dimensionale Arrays der Größe $1 \times n$.

3.13 Matrixmultiplikationen mit @

Vielleicht haben Sie die Beispiele des vorherigen Abschnitts auch einmal mit dem Operator `*` anstelle von `+` ausprobiert und sich gefragt, ob es denn auch eine Möglichkeit gibt, eine klassische Matrixmultiplikation durchzuführen.[3] Die Matrixmultiplikation

[3] Wenn Sie mit dem mathematischen Konzept einer Matrixmultiplikation noch nicht vertraut sein sollten, dann können Sie diesen Abschnitt und den folgenden über das Lösen linearer Gleichungssysteme zunächst überspringen. Sie können diese Abschnitte dann nachholen, wenn Sie bei der Lektüre mit Kap. 6 beginnen. Dort werden wir von der Matrixmultiplikation Gebrauch machen. Die formale Definition der Matrixmultiplikation und viele nützliche Eigenschaften finden Sie in jedem einführenden Mathematikbuch über lineare Algebra [5, 6].

wird in der mathematischen Notation meist mit einem Punkt · gekennzeichnet. In Python wird stattdessen der Operator `@` verwendet. Bitte rechnen Sie als Beispiel einmal

$$\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\end{bmatrix} \cdot \begin{bmatrix}13\\14\\15\\16\end{bmatrix} = \begin{bmatrix}1\cdot 13 + 2\cdot 14 + 3\cdot 15 + 4\cdot 16\\ 5\cdot 13 + 6\cdot 14 + 7\cdot 15 + 8\cdot 16\\ 9\cdot 13 + 10\cdot 14 + 11\cdot 15 + 12\cdot 16\end{bmatrix} = \begin{bmatrix}150\\382\\614\end{bmatrix} \tag{3.5}$$

in Python mithilfe des Operators `@` nach. In dem folgenden Beispiel wird eine Matrixmultiplikation einer 1 × 3-Matrix mit einer 3 × 1-Matrix durchgeführt:

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} \cdot \begin{bmatrix}4\\5\\6\end{bmatrix} = \begin{bmatrix}1\cdot 4 + 2\cdot 5 + 3\cdot 6\end{bmatrix} = \begin{bmatrix}32\end{bmatrix} \tag{3.6}$$

Der Python-Code für die entsprechende Rechnung ist unten abgedruckt. Man kann erkennen, dass hier tatsächlich ein 2-dimensionales Array der Größe 1 × 1 erzeugt wird.

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3]])
>>> b = np.array([[4], [5], [6]])
>>> a @ b
array([[32]])
```

Eine häufige Anwendung des `@`-Operators besteht darin, das Skalarprodukt zweier Vektoren zu berechnen, wenn man die Vektoren durch 1-dimensionale Arrays darstellt.

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> a @ b
32
```

An den vorangegangenen Beispielen konnten Sie erkennen, dass der Operator `@` immer eine Multiplikation durchführt, bei der über einen gemeinsamen Index summiert wird. Da wir später diesen Operator auch auf höherdimensionale Arrays anwenden werden, wollen wir hier kurz die Logik des Operators diskutieren. Zu jeder der folgenden Regeln sind Beispiele gegeben, wobei nur die jeweilige Array-Größe angegeben ist und die Indizes, über die summiert wird, rot markiert sind.

- Summiert wird immer über den letzten Index des ersten Faktors und den vorletzten Index des zweiten Faktors:

$$\begin{aligned}(3\times 4) \;\; @ \;\; (4\times 6) &\;\longrightarrow\; (3\times 6)\\ (3\times 4) \;\; @ \;\; (4\times 1) &\;\longrightarrow\; (3\times 1)\\ (1\times 4) \;\; @ \;\; (4\times 3) &\;\longrightarrow\; (1\times 3)\\ (3\times 4) \;\; @ \;\; (5\times 6) &\;\longrightarrow\; \texttt{valueError}\end{aligned}$$

- Wenn der erste Faktor ein 1-dimensionales Array ist, wird dieses wie ein 1 × n-Array behandelt. Die zusätzlich eingeführte Dimension wird nach der Multiplikation wieder entfernt:

$$(4) \;\; @ \;\; (4\times 3) \;\longrightarrow\; (3)$$

Arrays und Matrizen in NumPy

In NumPy gibt es neben der Klasse `np.ndarray` noch die Klasse `np.matrix`. Ein `matrix`-Objekt ist immer 2-dimensional und verhält sich ähnlich wie ein 2-dimensionales Array. Der Unterschied zum Array besteht darin, dass der Operator `*` bei Matrizen die Matrizenmultiplikation ausführt und nicht die elementweise Multiplikation. Der Potenzoperator bewirkt bei Matrizen eine Matrixpotenz. Wenn man im selben Programm sowohl Arrays als auch Matrizen einsetzt, ist der Code sehr schwer verständlich, weil man dem Zeichen `*` dann nicht ohne Weiteres ansieht, ob eine elementweise Multiplikation oder eine Matrixmultiplikation ausgeführt wird. Empfehlung: Verwenden Sie immer nur Arrays und keine Matrizen!

- Wenn der zweite Faktor ein 1-dimensionales Array ist, wird dieses wie ein $n \times 1$-Array behandelt. Die zusätzlich eingeführte Dimension wird nach der Multiplikation wieder entfernt:

$$(3 \times 4) \quad @ \quad (4) \quad \longrightarrow \quad (3)$$

- Alle anderen Dimensionen der beiden Faktoren werden gemäß der Regel für das Broadcasting behandelt:

$$\begin{array}{rcrcl}
(2 \times 6 \times 3 \times 4) & @ & (4 \times 5) & \longrightarrow & (2 \times 6 \times 3 \times 5) \\
(2 \times 6 \times 3 \times 4) & @ & (4) & \longrightarrow & (2 \times 6 \times 3) \\
(4) & @ & (2 \times 6 \times 3 \times 4) & \longrightarrow & \texttt{valueError} \\
(4) & @ & (2 \times 6 \times 4 \times 3) & \longrightarrow & (2 \times 6 \times 3) \\
(1 \times 4) & @ & (2 \times 6 \times 4 \times 3) & \longrightarrow & (2 \times 6 \times 1 \times 3)
\end{array}$$

3.14 Lösen von linearen Gleichungssystemen

Wir werden im Verlauf des Buches einige physikalische Systeme kennenlernen, deren Behandlung das Lösen von linearen Gleichungssystemen erfordert oder die sich in einer Näherung durch lineare Gleichungssysteme beschreiben lassen. Ein Beispiel für ein lineares Gleichungssystem ist durch

$$\begin{array}{rcrcrcr}
7x & + & 4{,}5y & + & 3z & = & 5 \\
2{,}4x & + & 3y & + & 9{,}5z & = & -12{,}3 \\
7x & + & 2{,}1y & + & 4{,}6z & = & 17{,}4
\end{array} \tag{3.7}$$

gegeben, wobei x, y, und z die gesuchten Größen sind. Wenn das Gleichungssystem genauso viele Gleichungen wie Unbekannte hat und die Gleichungen linear unabhängig sind, dann gibt es eine eindeutige Lösung des Gleichungssystems. Man kann dieses Gleichungssystem auch in der folgenden Art aufschreiben, wobei man die Koeffizienten in einer Matrix zusammenfasst:

$$\begin{pmatrix} 7 & 4{,}5 & 3 \\ 2{,}4 & 3 & 9{,}5 \\ 7 & 2{,}1 & 4{,}6 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 5 \\ -12{,}3 \\ 17{,}4 \end{pmatrix} \tag{3.8}$$

In NumPy ist mit der Funktion `np.linalg.solve` ein effizientes Verfahren implementiert, das solche Gleichungssysteme lösen kann. Wir müssen dazu zunächst die Koeffizientenmatrix `A` als 3 × 3-Array und den Vektor `b` auf der rechten Seite von Gleichung (3.8) als 1-dimensionales Array definieren.

```
>>> import numpy as np
>>> A = np.array([[7, 4.5, 3], [2.4, 3, 9.5], [7, 2.1, 4.6]])
>>> b = np.array([5, -12.3, 17.4])
```

Anschließend erhalten wir durch den Aufruf der Funktion `np.linalg.solve` den Ergebnisvektor `x` als 1-dimensionales Array.

```
>>> x = np.linalg.solve(A, b)
>>> print(x)
[ 4.62443268 -5.62455875 -0.68683812]
```

Überprüfen Sie durch Einsetzen dieser Werte in die Gleichungen (3.7), dass dies tatsächlich die Lösung dieses Gleichungssystems ist. Wenn Sie das in Python direkt überprüfen wollen, dann können Sie das vorherige Beispiel wie folgt fortsetzen, indem Sie die Matrixmultiplikation mit dem `@`-Operator verwenden:

```
 >>> print(A @ x)
[  5.  -12.3  17.4]
```

Wenn das Gleichungssystem nicht eindeutig lösbar ist, weil die Gleichungen nicht linear unabhängig sind, dann liefert Ihnen `np.linalg.solve` eine Fehlermeldung, dass die Matrix singulär ist. Sie erhalten ebenfalls eine Fehlermeldung, wenn das Array, das die Koeffizientenmatrix enthält, nicht quadratisch ist, also nicht genauso viele Zeilen wie Spalten hat.

3.15 Änderung der Form von Arrays

Es kommt häufiger vor, dass man aus einem 2-dimensionalen Array ein 1-dimensionales Array machen möchte oder umgekehrt. Dazu gibt es die Funktion `np.reshape` bzw. die Methode `reshape` der Array-Objekte.[4] Wir demonstrieren an einem Beispiel, wie man ein 1-dimensionales Array mit zehn Elementen in ein 2-dimensionales 2 × 5-Array umwandelt:

```
1 >>> import numpy as np
2 >>> a = np.linspace(0, 9, 10)
3 >>> print(a)
4 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
5 >>> a.shape
6 (10,)
7 >>> a = a.reshape(2, 5)
8 >>> a.shape
9 (2, 5)
```

[4] Es ist eine Frage des persönlichen Geschmacks, ob man lieber mit der Methode oder mit der Funktion arbeitet.

```
>>> print(a)
[[0. 1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]
```

Sie erkennen, dass die Elemente des 2-dimensionalen Arrays zeilenweise mit den Elementen des 1-dimensionalen Arrays gefüllt sind. Bei jeder Verwendung von `reshape` muss die Gesamtzahl der Elemente erhalten bleiben. Gewissermaßen ist also eine der beiden Angaben »2-Zeilen« und »5-Spalten« überflüssig, da sie sich aus der Gesamtzahl der Elemente ergibt. Aus diesem Grund gibt es die Möglichkeit, für eine Dimension des Ziel-Arrays die Größe -1 anzugeben, damit NumPy die Größe dieser Dimension automatisch festlegt. Dies kann man dazu benutzen, um aus einem 1-dimensionalen Array mit n Elementen ein 2-dimensionales Array der Größe $n \times 1$ zu erzeugen, wie das folgende Beispiel zeigt:

```
>>> import numpy as np
>>> a = np.linspace(0, 9, 10)
>>> b = a.reshape(-1, 1)
>>> b.shape
(10, 1)
```

Eine andere Art der Formänderung von Arrays ist das **Transponieren**, bei dem die Dimensionen eines Arrays vertauscht werden. Am häufigsten wird dies für 2-dimensionale Arrays angewendet. Um die Transponierte eines Arrays `a` zu erhalten, kann man entweder den Ausdruck `a.T` verwenden oder die Funktion `np.transpose`. Das Transponieren entspricht dabei der gleichnamigen Operation auf Matrizen in der linearen Algebra, wie das folgende Beispiel zeigt:

```
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4], [5, 6]]))
  >>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a.T
array([[1, 3, 5],
       [2, 4, 6]])
```

Aus den Zeilen werden also die Spalten und umgekehrt. Bei höherdimensionalen Arrays kann man mithilfe der Funktion `np.transpose` die Dimensionen beliebig umsortieren. Wenn man bei einem höherdimensionalen Array zwei Dimensionen vertauschen möchte, kann man stattdessen die Funktion `np.swapaxes` verwenden. Die Details hierzu finden Sie in der Hilfe-Funktion der entsprechenden Funktion.

3.16 Grafische Ausgaben mit Matplotlib

Ein Bild sagt mehr als tausend Worte, und erst recht sagt ein Bild mehr als eine lange Liste von Zahlen. In der Physik und in den Naturwissenschaften allgemein werden sehr häufig Diagramme verwendet, die den Zusammenhang von Größen veranschaulichen.

Die häufigste Variante ist eine Auftragung einer Größe als Funktion einer anderen Größe.

Die Bibliothek Matplotlib bietet ein riesiges Reservoir an grafischen Darstellungsmöglichkeiten, von denen hier nur ganz wenige vorgestellt werden können. Viele weitere werden Sie später bei den physikalischen Anwendungen kennenlernen. Die Bibliothek Matplotlib besteht aus vielen Teilmodulen. Es wird empfohlen, die zwei wichtigsten Module immer in der folgenden Form zu importieren:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Als ein einfaches Beispiel wollen wir den Funktionsgraphen der Sinusfunktion darstellen. Wenn Sie das Programm 3.2 ausführen, öffnet sich ein neues Fenster, in dem ein Plot dargestellt ist, der ungefähr wie Abb. 3.2 aussehen sollte. Um diese Grafik zu erzeugen, importieren wir zunächst die benötigten Module und erzeugen eine Wertetabelle der Sinusfunktion. Dabei müssen wir darauf achten, dass wir die Winkel vom Gradmaß in das Bogenmaß umwandeln. Die Wertetabelle besteht aus 500 Datenpunkten im Winkelbereich von 0° bis 360°. Die große Anzahl der Punkte stellt sicher, dass der Funktionsgraph schön glatt aussieht.

Programm 3.2: *NumpyMatplotlib/plot_sinus.py*

```
1 """Funktionsgraph der Sinusfunktion."""
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # Erzeuge ein Array für die x-Werte in Grad und für die y-Werte.
7 x = np.linspace(0, 360, 500)
8 y = np.sin(np.radians(x))
```

Nun erzeugen wir eine **Figure**. Normalerweise besteht eine Figure aus einem neuen Fenster. Dieses Fenster beinhaltet einige Schaltflächen, mit denen man einzelne Teile der Figure vergrößern oder den Inhalt der Figure ausdrucken bzw. in einer Datei abspeichern kann. Aus der Perspektive der Programmiersprache ist eine Figure gewissermaßen die Fläche, in die etwas gezeichnet werden kann.

```
11 fig = plt.figure()
```

Die Funktion `plt.figure` akzeptiert eine ganze Reihe optionaler Argumente, mit denen man beispielsweise die Größe der Figure beeinflussen kann. Wir haben es hier aber bei den Standardeinstellungen belassen. Anschließend muss man innerhalb der Figure eine **Axes** erstellen. Ein Axes-Objekt legt einen bestimmten Bereich innerhalb der Figure fest. Darüber hinaus beinhaltet das Axes-Objekt ein Koordinatensystem. In einer Figure können mehrere Axes-Objekte positioniert werden. Am häufigsten ordnet man diese mit der Funktion `fig.add_subplot` in einem Raster an. Das erste Argument ist dabei die Anzahl der Zeilen, das zweite Argument ist die Anzahl der Spalten, und das dritte Argument ist eine fortlaufende Nummer. Leider weicht hier die Konvention von der üblichen Indizierung in Python ab, da diese fortlaufende Nummer mit 1 beginnt

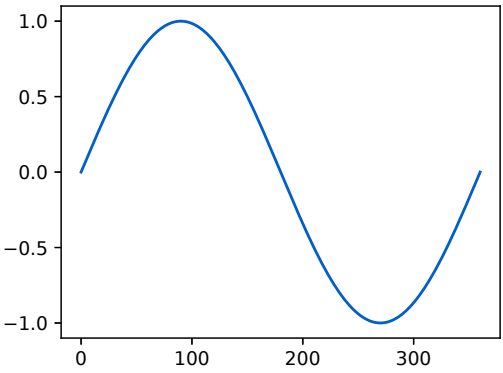

Abb. 3.2: Einfacher Plot. *Für eine Abbildung, die wissenschaftlichen Standards genügt, fehlen noch die Beschriftungen.*

und nicht mit 0. Da wir nur einen Graphen darstellen wollen, legen wir ein 1 × 1-Raster an und erzeugen das Axes-Objekt an der ersten Position:

```
12 ax = fig.add_subplot(1, 1, 1)
```

Als Nächstes wollen wir einen Funktionsgraphen in das eben definierte Axes-Objekt einzeichnen:

```
15 ax.plot(x, y)
```

Mithilfe der Funktion `plt.show` wird das in den Zeilen davor angelegte Bild angezeigt, und Python wartet, bis das Grafikfenster geschlossen wird.

```
18 plt.show()
```

Geben Sie die Befehle aus dem obigen Beispiel bitte einmal direkt in der Python-Shell ein. Wenn Sie das tun, fällt Ihnen auf, dass bis zur Zeile 15 überhaupt keine Ausgabe des Programms erfolgt.[5] Erst in Zeile 18 wird das zuvor angelegte Bild angezeigt, und Python wartet, bis das Grafikfenster geschlossen wird. Wahrscheinlich wünschen Sie sich stattdessen, dass die Befehle sofort ein Resultat auf dem Bildschirm erzeugen. Das können Sie mit der Funktion `plt.ion` erreichen. Der Name `ion` steht für »interactive mode on« und schaltet in den sogenannten interaktiven Modus. Geben Sie bitte einmal die Befehle aus Programm 3.2 in einer Python-Shell ein und fügen Sie nach dem Import von `matplotlib.pyplot` den Funktionsaufruf `plt.ion()` ein. Sie werden sehen, dass die Befehle nun sofort eine Grafikausgabe bewirken.

Vielleicht fragen Sie sich, warum der interaktive Modus nicht einfach immer eingeschaltet ist. Der Grund liegt in der Ausführungsgeschwindigkeit. Wenn für aufwendige Grafiken sehr viele unterschiedliche Grafikobjekte erzeugt werden, ist es deutlich schneller, wenn das Bild nur einmal am Ende auf dem Bildschirm dargestellt wird.

Die Ausgabe in Abb. 3.2 ist nicht völlig befriedigend. Für eine nach wissenschaftlichen Maßstäben korrekte Abbildung fehlen die Achsenbeschriftungen, Gitternetzlinien, ein Titel und eine Legende. Das Programm 3.3 erzeugt eine etwas aufwendigere Darstellung, in der nicht nur die Sinusfunktion, sondern auch die Kosinusfunktion dargestellt

[5] Wenn Sie anstelle der gewöhnlichen Python-Shell die IPython-Shell verwenden, kann es sein, dass die Grafikbefehle hier sofort eine Bildschirmausgabe nach sich ziehen. Das liegt daran, dass einige Versionen von IPython den interaktiven Modus automatisch aktivieren.

ist und in der die üblichen Beschriftungen hinzugefügt wurden. Das Ergebnis ist in Abb. 3.3 dargestellt. Die Bedeutung der einzelnen Befehle in diesem Beispiel, die bisher nicht erklärt wurden, können Sie sich anhand der Ausgabe verdeutlichen. Es gibt eine nahezu unüberschaubare Anzahl von Optionen, mit denen man das Verhalten und Aussehen der Grafik beeinflussen kann. In diesem Buch kann aufgrund dieser Vielzahl keine auch nur annähernd vollständige Übersicht gegeben werden. Eine ausführliche Dokumentation aller Funktionen und Optionen finden Sie online [7].

Programm 3.3: *NumpyMatplotlib/plot_sin_cos_beschriftet.py*

```
"""Graph der Sinus- und Kosinusfunktion mit Beschriftung."""

import numpy as np
import matplotlib.pyplot as plt

# Erzeuge ein Array für die x-Werte in Grad.
x = np.linspace(0, 360, 500)

# Erzeuge je ein Array für die zugehörigen Funktionswerte.
y_sin = np.sin(np.radians(x))
y_cos = np.cos(np.radians(x))

# Erzeuge eine Figure und eine Axes.
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

# Beschrifte die Achsen. Lege den Wertebereich fest und erzeuge
# ein Gitternetz.
ax.set_title('Sinus- und Kosinusfunktion')
ax.set_xlabel('Winkel [Grad]')
ax.set_ylabel('Funktionswert')
ax.set_xlim(0, 360)
ax.set_ylim(-1.1, 1.1)
ax.grid()

# Plotte die Funktionsgraphen und erzeuge eine Legende.
ax.plot(x, y_sin, label='Sinus')
ax.plot(x, y_cos, label='Kosinus')
ax.legend()

# Zeige die Grafik an.
plt.show()
```

3.17 Animationen mit Matplotlib

Die Physik beschäftigt sich in nahezu all ihren Bereichen mit der Frage, wie sich der Zustand eines physikalischen Systems im Laufe der Zeit verändert. Zeitliche Veränderungen lassen sich oft nur schwer in statischen Abbildungen darstellen, und aus

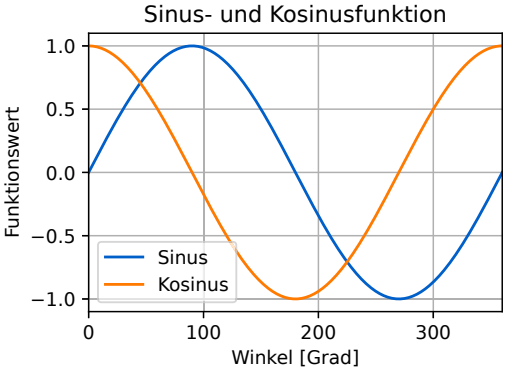

Abb. 3.3: Plot mit Beschriftung. *Die üblichen Beschriftungen von Diagrammen lassen sich mit Matplotlib sehr einfach darstellen.*

genau diesem Grund ist es besonders interessant, mit dem Computer bewegte Bilder zu erzeugen. Im Folgenden wollen wir die Funktion

$$u(x,t) = \cos(kx - \omega t) \tag{3.9}$$

animiert darstellen, wobei k und ω zwei vorgegebene Zahlen sind. Es handelt sich hierbei um die Darstellung einer ebenen Welle, die in Kap. 11 genauer besprochen wird. Die Konstante k bezeichnet man als die Wellenzahl, und die Konstante ω ist die sogenannte Kreisfrequenz. Um die Funktion (3.9) animiert darzustellen, müssen wir zu jedem Zeitpunkt t die Größe u als Funktion des Ortes x plotten. Dazu importieren wir zunächst die benötigten Module:

Programm 3.4: `NumpyMatplotlib/animation.py`

```
1  """Animierte Darstellung einer ebenen Welle."""
2
3  import numpy as np
4  import matplotlib as mpl
5  import matplotlib.pyplot as plt
6  import matplotlib.animation
```

Neu hinzugekommen ist hier das Modul `matplotlib.animation`, das wir später im Programm benutzen werden. Im Prinzip könnte man das Modul auch erst später importieren. Es ist aber die gängige und empfohlene Praxis, alle Importe am Anfang eines Programms durchzuführen.

Als Nächstes definieren wir ein Array mit x-Werten, die dargestellt werden sollen.

```
9  x = np.linspace(0, 20, 500)
```

Weiterhin weisen wir den Konstanten k und ω sinnvolle Werte zu und definieren eine Zeitschrittweite, die die Zeitdifferenz zwischen zwei Bildern der Animation angibt.

```
13 omega = 1.0
14 k = 1.0
15 delta_t = 0.04
```

Nun erstellen wir ein Figure- und ein Axes-Objekt und erzeugen die Achsenbeschriftungen. Da der Plot erst im Verlauf der Animation vervollständigt wird, muss man zu

Animationen in Spyder

Viele der Programme in diesem Buch erzeugen animierte grafische Ausgaben. Wenn Sie Spyder verwenden, werden die Grafiken standardmäßig innerhalb von Spyder dargestellt und nicht in einem separaten Grafikfenster. Leider werden von Spyder nur statische Grafiken und keine Animationen unterstützt. Damit alle Programme auch mit Spyder funktionieren, klicken Sie im Menü »Werkzeuge« bitte den Punkt »Voreinstellungen« an. Dort wählen Sie bitte unter dem Punkt »IPython-Konsole« im Reiter »Grafik« für »Backend« den Wert »automatisch« an.

Beginn bereits die Skalierung der Achsen festlegen. Dies geschieht mit `ax.set_xlim` und `ax.set_ylim`, indem man den kleinsten und den größten Wert für die entsprechende Achse angibt. Für den horizontalen Bereich verwenden wir die Funktionen `np.min` und `np.max`, um den kleinsten und größten Wert im Array `x` zu bestimmen. In der vertikalen Richtung erwarten wir Funktionswerte zwischen -1 und $+1$. Wir setzen den Grenzen des Plotbereichs hier aber bewusst etwas größer.

```
18 fig = plt.figure()
19 ax = fig.add_subplot(1, 1, 1)
20 ax.set_xlim(np.min(x), np.max(x))
21 ax.set_ylim(-1.2, 1.2)
22 ax.set_xlabel('Ort x')
23 ax.set_ylabel('u(x, t)')
24 ax.grid()
```

Wir erzeugen nun einen leeren Linienplot, indem wir der Funktion `ax.plot` zwei leere Listen übergeben. Genauer gesagt, erzeugen wir ein Objekt vom Typ `matplotlib.lines.Line2D`.

```
27 plot, = ax.plot([], [])
```

Später müssen wir während der Animation diesem Linienplot die zu plottenden Daten übergeben und speichern dazu den Plot in der Variablen `plot`. Das Komma hinter `plot` ist wichtig. Mit der Funktion `ax.plot` können nämlich auch mehrere Plots auf einmal erstellt werden. Aus diesem Grund liefert die Funktion eine Liste zurück, die die erzeugten Plots enthält. Da wir nur einen Plot erzeugen, wird eine Liste mit nur einem Element zurückgeliefert. Das Komma hinter `plot` sorgt dafür, dass diese Liste entpackt wird (siehe Abschn. 2.11).

In der Animation soll nicht nur der Funktionsgraph dargestellt werden, sondern auch noch der aktuelle Wert der Zeitkoordinate t. Wir erzeugen daher noch ein Textfeld mit der Methode `text` des Axes-Objekts. Das Textfeld wird an den Koordinaten $x = 0{,}5$ und $y = 1{,}05$ positioniert und enthält zunächst nur einen leeren String.

```
28 text = ax.text(0.5, 1.05, '')
```

Für die Animation müssen wir nun eine Funktion definieren, die wir `update` nennen. Diese Funktion erhält als Argument die Nummer des Bildes innerhalb der Animation, das gerade dargestellt werden soll.

```
31 def update(n):
32     """Aktualisiere die Grafik zum n-ten Zeitschritt."""
```

Innerhalb dieser Funktion berechnen wir die neuen Funktionswerte `u` zum aktuellen Zeitpunkt `t`.

```
34     t = n * delta_t
35     u = np.cos(k * x - omega * t)
```

Anschließend aktualisieren wir den Plot und den im Textfeld angezeigten String.

```
38     plot.set_data(x, u)
39     text.set_text(f't = {t:5.1f}')
```

Am Ende der Funktion geben wir ein Tupel zurück, das aus den beiden Grafikelementen besteht, die neu gezeichnet werden müssen.

```
43     return plot, text
```

Beachten Sie bitte, dass wir innerhalb der Funktion `update` keine neuen Grafikelemente erzeugen, sondern lediglich die vorhandenen Elemente `plot` und `text` aktualisieren. Dazu hat jedes Grafikobjekt eine oder mehrere entsprechende Methoden. So kann man beispielsweise mit `text.set_text` den Text eines Textfeldes aktualisieren. Für das Objekt `plot` werden in unserem Beispiel mit `plot.set_data` die Koordinaten der Datenpunkte aktualisiert. Darüber hinaus gibt es auch noch die Methode `plot.set_xdata`, mit der man die x-Koordinaten der Datenpunkte aktualisieren kann, und eine Methode `plot.set_ydata`, die die y-Koordinaten aktualisiert. Die einzelnen Grafikobjekte haben meistens viele Methoden, deren Namen mit `set_` beginnen. Mit diesen Methoden kann man die meisten Eigenschaften der Grafikobjekte, wie beispielsweise Farben, Linienstärken und Punktgrößen, nachträglich verändern.

Wir erzeugen nun ein Animationsobjekt `ani`. Dieses Objekt kümmert sich um die gesamte Animation. Als erstes Argument wird die Figure übergeben, die animiert werden soll. Als zweites Argument wird die Funktion `update` übergeben.[6]

```
47 ani = mpl.animation.FuncAnimation(fig, update,
48                                    interval=30, blit=True)
```

Das Argument `interval=30` bewirkt, dass zwischen zwei Bildern eine Pause von 30 ms gelassen wird. Das letzte Argument `blit=True` bewirkt, dass nur die Teile des Bildes neu gezeichnet werden, die sich auch verändert haben. Dadurch laufen die meisten Animationen flüssiger. Wenn wir den Plot auf dem Bildschirm anzeigen, startet automatisch die Animation.

```
51 plt.show()
```

[6] Python ist hier unkompliziert: Funktionen sind ganz normale Objekte und können daher genauso wie andere Variablen als Argumente einer Funktion verwendet werden.

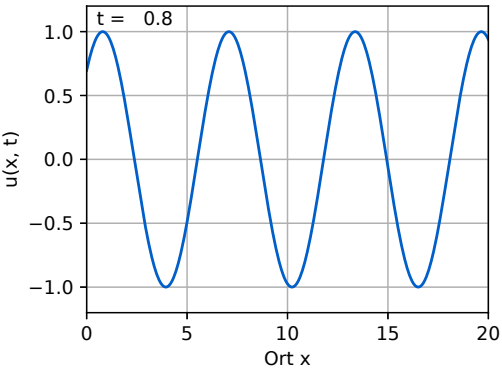

Abb. 3.4: Animation eines Plots. *Das Programm 3.4 erzeugt eine animierte Darstellung einer ebenen Welle nach Gleichung (3.9).*

Das Animationsobjekt ruft jetzt alle 30 ms die Funktion `update` auf und übergibt als Argument eine laufende Nummer des Bildes. Die Funktion `update` verändert den Plot und das Textfeld. Anschließend gibt die Funktion `update` ein Tupel zurück, das die Grafikobjekte enthält, die neu dargestellt werden müssen. Das Aktualisieren der Darstellung übernimmt dann wieder das Animationsobjekt.

Achtung!

Ein Animationsobjekt muss immer einer Variablen zugewiesen werden, auch wenn man auf diese Variable nicht explizit zugreift. Andernfalls wird das Animationsobjekt vom Garbage Collector entsorgt, und die Animation bleibt stehen.

Wenn Sie das Programm 3.4 laufen lassen, sehen Sie eine nach rechts, also in positive x-Richtung laufende, sinusförmige Welle, die in Abb. 3.4 dargestellt ist.

Wenn Sie die Animation nicht direkt auf dem Bildschirm anzeigen lassen möchten, sondern lieber eine Videodatei erzeugen möchten, so können Sie die letzten Zeilen durch die folgenden ersetzen:

```
47 ani = mpl.animation.FuncAnimation(fig, update, frames=1000,
48                                    blit=True)
49 ani.save('wellen.mp4', fps=30)
```

Diese beiden Befehle sorgen dafür, dass 1000 Bilder erzeugt werden, die in einer `.mp4`-Datei mit einer Bildrate von 30 Bildern pro Sekunde abgespeichert werden. Damit das Erzeugen der Videodatei funktioniert, müssen Sie allerdings zunächst das Programm FFmpeg installieren (siehe Kasten).

3.18 Positionierung von Grafikelementen

Im vorherigen Abschnitt haben wir eine Animation einer ebenen Welle dargestellt, in der der jeweilige Zeitpunkt mithilfe eines Textfeldes angezeigt wurde. Dieses Textfeld wurde mit dem Befehl

```
ax.text(0.5, 1.05, '')
```

Installation von FFmpeg

Das Programm FFmpeg ist ein frei verfügbares Programmpaket zur Bearbeitung und Konvertierung von digitalen Video- und Audiodaten [8]. Wenn Sie mit Matplotlib Videodateien erzeugen wollen, sollten Sie dieses Programm installieren.

- Wenn Sie die Python-Distribution Anaconda verwenden, dann können Sie das Programm FFmpeg in einem Anaconda-Prompt mit dem Befehl `conda install ffmpeg` installieren.
- Wenn Sie Linux verwenden, sollten Sie in der Paketverwaltung Ihres Linux-Systems nach einem Paket mit dem Namen `ffmpeg` suchen und dieses installieren.
- Sie können FFmpeg auch direkt von der Webseite des FFmpeg-Projekts [8] herunterladen. Die Installation erfordert jedoch einige Handarbeit.

erzeugt, der das Textfeld an dem Punkt mit den Koordinaten $x = 0{,}5$ und $y = 1{,}05$ positioniert hat. Das Textfeld wird also an einer Stelle positioniert, die durch die **Datenkoordinaten** gegeben ist. Wenn Sie die Animation mit dem Programm 3.4 laufen lassen und beispielsweise in die Animation mit der Maus hereinzoomen, dann stellen Sie fest, dass der Text unter Umständen nicht mehr sichtbar ist, weil der Punkt mit den angegebenen Koordinaten nicht mehr im Koordinatensystem dargestellt wird. Je nach Anwendungszweck mag das gewünscht sein oder auch nicht.

Matplotlib bietet die Möglichkeit, Grafikobjekte auch mithilfe anderer Koordinatensysteme zu positionieren. Die drei häufigsten Koordinatensysteme sind in Abb. 3.5 dargestellt. Wenn Sie möchten, dass ein Text immer in der linken unteren Ecke der Figure dargestellt wird, dann können Sie dies mit

```
ax.text(0.0, 0.0, 'Text', transform=fig.transFigure)
```

erreichen.[7] Wenn Sie den Text dagegen immer in der unteren linken Ecke des Koordinatensystems positionieren möchten, können Sie den folgenden Befehl verwenden:

```
ax.text(0.0, 0.0, 'Text', transform=ax.transAxes)
```

Ein weiterer Aspekt bei der Positionierung von Grafikobjekten in Matplotlib ist die sogenannte **zorder**. Durch die Angabe der `zorder` lassen sich Grafikobjekte in unterschiedlichen Zeichenebenen darstellen. Wir demonstrieren dies am folgenden Beispiel:

Programm 3.5: *NumpyMatplotlib/zorder.py*

```
1 """Demonstration der zorder in Matplotlib."""
2
3 import numpy as np
4 import matplotlib.pyplot as plt
```

[7] Achtung: Falls Sie den Text damit außerhalb der Axes positionieren, wird dieser in einer Animation nur dann korrekt aktualisiert, wenn Sie auf die Option `blit=True` von `FuncAnimation` verzichten.

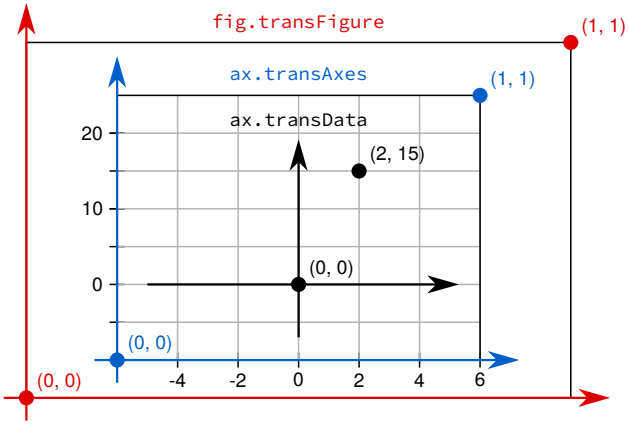

Abb. 3.5: Koordinatensysteme in Matplotlib. *Grafikobjekte lassen sich mithilfe des Axes-Koordinatensystems (`ax.transAxes`) oder des Figure-Koordinatensystems (`fig.transFigure`) unabhängig von den Koordinatenachsen der darzustellenden Daten positionieren.*

```
5
6  x_grob = np.array([0, 2, 4, 8])
7  x_fein = np.linspace(0, 10, 500)
8
9  fig = plt.figure(figsize=(6, 3))
10 ax1 = fig.add_subplot(1, 2, 1)
11 ax1.plot(x_grob, x_grob ** 2, 'ro')
12 ax1.plot(x_fein, x_fein ** 2, 'b-', linewidth=2)
13
14 ax2 = fig.add_subplot(1, 2, 2)
15 ax2.plot(x_grob, x_grob ** 2, 'ro', zorder=2)
16 ax2.plot(x_fein, x_fein ** 2, 'b-', linewidth=2, zorder=1)
17
18 plt.show()
```

In Zeile 11 werden vier Datenpunkte geplottet, wobei das zusätzliche Argument `'ro'` dafür sorgt, dass die Datenpunkte als rote Kreise dargestellt werden.[8] In Zeile 12 werden 500 Datenpunkte geplottet, wobei das zusätzliche Argument `'b-'` für eine blaue durchgezogene Linie sorgt und das Argument `linewidth=2` die Linie etwas breiter als normal zeichnet. Da die Linie nach den Punkten gezeichnet wird, erhalten wir die Situation in Abb. 3.6 links, bei der die Linie die Punkte teilweise verdeckt. Wenn man ein Aussehen wie in Abb. 3.6 rechts erhalten möchte, bei dem die Punkte die Linie verdecken, dann kann man entweder die Reihenfolge der Plotbefehle vertauschen oder man kann angeben, dass die Punkte in einer Zeichenebene oberhalb der Linie liegen. Diese Zeichenebene legt man mit dem Argument `zorder` fest, wobei ein höherer Wert bedeutet, dass das Grafikobjekt weiter vorne liegt. In den Zeilen 15 und 16 wurde für die Punkte `zorder=2` und für die Linie `zorder=1` gesetzt, sodass die Punkte vor der Linie dargestellt werden.

[8] In der Dokumentation zu Matplotlib finden Sie zu diesen Formatangaben eine ausführliche Dokumentation.

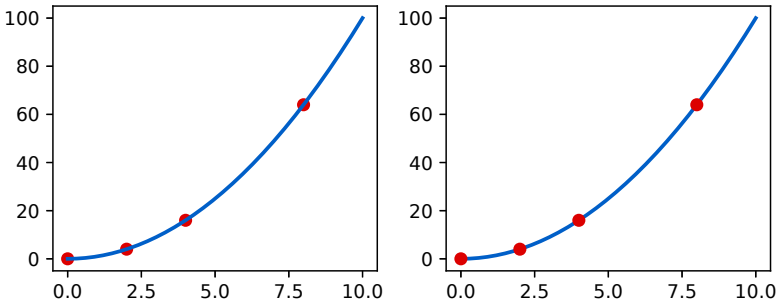

Abb. 3.6: Zorder in Matplotlib. *(Links) Da die blaue Linie nach den roten Punkten geplottet wurde, werden die Punkte von der Linie teilweise verdeckt. (Rechts) Durch Angabe einer höheren* `zorder` *für die Punkte werden die Punkte im Vordergrund dargestellt.*

Zusammenfassung

NumPy: Die Bibliothek NumPy ist für das wissenschaftliche Rechnen mit Python essenziell. Die wichtigsten Grundfunktionen der Bibliothek wurden vorgestellt. Einen ausführlichen Einstieg in die Arbeit mit NumPy gibt das Buch von Johansson [9].

Indizierung von Arrays: Ganz wesentlich für das Arbeiten mit NumPy ist die Art und Weise, wie Arrays indiziert werden können. Wir haben besprochen, dass Arrays ähnlich wie Listen oder Tupel indiziert werden können. Darüber hinaus haben wir diskutiert, wie man boolesche Arrays sehr effizient zur Indizierung von Arrays benutzen kann.

Broadcasting: Die Boadcasting-Regeln erscheinen am Anfang sicherlich verwirrend und es benötigt einige Zeit, bis man verinnerlicht hat, wie man mithilfe des Broadcastings und des `@`-Operators komplexe Operationen sehr effizient ausdrücken kann. Versuchen Sie bitte in den folgenden Kapiteln immer wieder einmal, zu den entsprechenden Regeln zurückzublättern und sich genau klar zu machen, wie die entsprechenden Ausdrücke funktionieren.

Grafiken mit Matplotlib: Wir haben an zwei Beispielen gezeigt, wie man mit der Bibliothek Matplotlib Grafiken erstellen kann. In dem Buch von Johansson [9] findet sich ebenfalls eine ausführliche Einführung hierzu. Darüber hinaus empfehle ich die Webseite von Matplotlib [7]. Dort finden Sie neben der Dokumentation aller Funktionen viele Beispielgrafiken zusammen mit dem entsprechenden Python-Code. Im Laufe des Buches werden wir an den konkreten physikalischen Beispielen viele weitere Möglichkeiten der Bibliothek Matplotlib kennen lernen.

Animationen: An einem einfachen Beispiel wurde gezeigt, wie man mit der Bibliothek Matplotlib auch Animationen erstellen kann. Viele weitere Animationen, die auf einem ähnlichen Grundgerüst aufbauen, werden Sie in den nachfolgenden Kapiteln kennen lernen.

Dokumentation: Wie auch bereits in Kap. 2 wurden nur einige wenige Aspekte der Funktionen von NumPy und Matplotlib erklärt, da eine vollumfängliche Behand-

lung den Umfang des Buches sprengen würde. Schauen Sie sich bitte unbedingt einmal die Online-Dokumentation zu NumPy und SciPy [10] sowie zu Matplotlib [7] an. Auch wenn der Umfang im ersten Moment überwältigend ist, so werden Sie sich schnell daran gewöhnen, sich in solchen Dokumentationen zurecht zu finden.

Aufgaben

Aufgabe 3.1: Erstellen Sie in Python 2-dimensionale Arrays, die den Beispielen in den Gleichungen (3.1) bis (3.4) entsprechen, und überprüfen Sie, ob Sie das gleiche Ergebnis erhalten. Was passiert, wenn Sie anstelle des 1×4-Arrays in Gleichung (3.1) ein 1-dimensionales Array mit vier Elementen verwenden? Was passiert, wenn Sie das 3×1-Array in Gleichung (3.2) durch ein 1-dimensionales Array mit drei Elementen ersetzen? Was passiert, wenn Sie die Reihenfolge der Summanden vertauschen?

Aufgabe 3.2: Erzeugen Sie ein 2-dimensionales Array, das eine Multiplikationstabelle für das kleine Einmaleins enthält:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 \\ 3 & 6 & 9 & 12 & \dots & & & & \\ \vdots & & & & & & & & \\ 9 & 18 & \dots & & & & & & 81 \end{pmatrix}$$

Verwenden Sie dabei keine `for`-Schleife, sondern benutzen Sie den Broadcasting-Mechanismus von NumPy und setzen Sie die Funktion `np.reshape` geschickt ein.

Aufgabe 3.3: Modifizieren Sie Ihre Lösung oder die Musterlösung der Aufgabe 2.7 zum Sieb des Eratosthenes so, dass anstelle der Listen boolesche Arrays verwendet werden. Ein mit `True` gefülltes boolesches Array können Sie einfach mit `np.ones(n, dtype=bool)` erzeugen. Mithilfe von Slices können Sie nun sehr effizient die Vielfachen von bereits gefundenen Primzahlen markieren.

Aufgabe 3.4: Stellen Sie den prozentualen Fehler der Näherung $\sin(x) \approx x$ im Bereich von 1° bis 45° grafisch dar. Achten Sie auf eine sinnvolle Beschriftung der Achsen.

Aufgabe 3.5: Der Body-Mass-Index (BMI) setzt Körpermasse m eines Menschen in Relation zur Körpergröße h. Er ist definiert durch:

$$\mathrm{BMI} = \frac{m}{h^2}$$

Häufig wird der Normalbereich für einen erwachsenen Mann zwischen $20\,\mathrm{kg/m^2}$ und $25\,\mathrm{kg/m^2}$ festgelegt. Darüber hinaus wird oft das Normalgewicht in kg als »Körpergröße in cm minus 100« angegeben und das Idealgewicht als »Normalgewicht minus 10 Prozent«.

x [m]	y [m]
0,0	1,0
0,0	2,8
1,0	3,3
2,2	3,5
2,8	3,4
3,8	2,7
4,6	2,4
5,7	2,3
6,4	2,1
7,1	1,6
7,6	0,9
7,9	0,5
7,9	0,0
0,0	1,0

Tab. 3.4: Blumenbeet. *Die Koordinaten geben die Eckpunkte eines Blumenbeetes, dessen Fläche berechnet werden soll, in kartesischen Koordinaten an (siehe Aufgabe 3.8).*

Erstellen Sie eine grafische Auftragung des so definierten Normalgewichts und des Idealgewichts als Funktion der Körpergröße im Bereich zwischen $h = 1{,}70\,\mathrm{m}$ und $h = 2{,}10\,\mathrm{m}$. Stellen Sie in der Grafik auch die Körpermasse für die Werte $\mathrm{BMI} = 20\,\mathrm{kg/m^2}$, $\mathrm{BMI} = 22{,}5\,\mathrm{kg/m^2}$ und $\mathrm{BMI} = 25\,\mathrm{kg/m^2}$ dar. Beschriften Sie die Grafik mit Achsenbeschriftungen und einer Legende.

Aufgabe 3.6: Eine rechteckförmige Funktion $f(x)$ der Periode 2π lässt sich durch eine Fourier-Reihe ausdrücken:

$$f(x) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{\sin((2k+1)x)}{2k+1} \tag{3.10}$$

Erstellen Sie ein Python-Programm, das diese Fourier-Reihe animiert darstellt, indem Sie die Funktion im Bereich von $x = 0 \ldots 2\pi$ plotten und im n-ten Bild der Animation die ersten n Summanden der Reihe zeigen.

Aufgabe 3.7: Die Takagi-Funktion ist für reelle Zahlen $x \in [0, 1]$ wie folgt definiert

$$f(x) = \sum_{k=0}^{\infty} \frac{s(2^k x)}{2^k}\,, \tag{3.11}$$

wobei $s(x)$ den Abstand der Zahl x von der nächsten ganzen Zahl bezeichnet. Man kann zeigen, dass diese Funktion auf dem gesamten Definitionsbereich überall stetig aber nirgendwo differenzierbar ist.

Visualisieren Sie diese Funktion mit einer Animation, indem Sie im n-ten Bild der Animation die Summe der ersten n-Summanden auswerten. Beschränken Sie sich dabei zur Vermeidung von Überläufen auf $n \leq 50$.

Aufgabe 3.8: Eine befreundete Staudengärtnerin, nennen wir sie Konnie, legt ein Blumenbeet an. Konnie hat einige Punkte der Berandung markiert und möchte für die Auswahl der Pflanzen die Fläche des Beetes bestimmen. Sie hat dazu in Tab. 3.4 die Eckpunkte des Beetes in kartesischen Koordinaten eingetragen. Schreiben Sie ein Python-Programm, das die Fläche des Beetes berechnet und die Umrandung des Beetes grafisch darstellt. Die Fläche eines Polygons kann man mit der gaußschen Trapezformel

$$A = \frac{1}{2} \left| \sum_{i=1}^{n} (y_i + y_{i+1})(x_i - x_{i+1}) \right| \tag{3.12}$$

berechnen, wobei x_i und y_i die Koordinaten der Eckpunkte des Polygons sind und man den Index $n + 1$ mit dem Index 1 gleichsetzen muss. Die Gleichung (3.12) kann man in Python sehr effizient mithilfe der Funktion `np.roll` implementieren. Informieren Sie sich dazu in der Hilfe über die Arbeitsweise dieser Funktion.

Literatur

[1] NumPy: The fundamental package for scientific computing with Python. https://numpy.org.

[2] SciPy: Fundamental algorithms for scientific computing in Python. https://scipy.org.

[3] Matplotlib: Visualization with Python. https://matplotlib.org.

[4] The Python Language Reference: Operator precedence. Python Software Foundation. https://docs.python.org/3/reference/expressions.html#operator-precedence.

[5] Stry Y und Rainer S. Mathematik Kompakt für Ingenieure und Informatiker. Berlin, Heidelberg: Springer Vieweg, 2013. DOI:10.1007/978-3-642-24327-1.

[6] Papula L. Mathematik für Ingenieure und Naturwissenschaftler Band 1. Wiesbaden: Springer Vieweg, 2018. DOI:10.1007/978-3-658-21746-4.

[7] Matplotlib Users Guide. https://matplotlib.org/stable/users/index.html.

[8] FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://www.ffmpeg.org.

[9] Johansson R. Numerical Python. Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib. Berkeley: Apress, 2019. DOI:10.1007/978-1-4842-4246-9.

[10] Numpy and Scipy Documentation. Python Software Foundation. https://docs.scipy.org.