Python & MicroPython: Programmieren lernen für Einsteiger

Grafikkunst mit MatPlotLib

Teil 6

Die meisten Menschen sind visuelle Wesen – eine einzige Grafik sagt daher oft mehr als umfangreiche Datenlisten oder numerische Tabellen. Zudem können Zusammenhänge meist wesentlich rascher erfasst werden, wenn sie in grafische Darstellungen gefasst sind. Mit MatPlotLib steht für Python eine Bibliothek zur Verfügung, mit der die Arbeit mit grafischen Darstellungen zum Kinderspiel wird. Kein Wunder also, dass sich Python und die MatPlotLib zunehmender Belietbheit erfreuen. In Kombination mit NumPy und SciPy ist die MatPlotLib klassischen Mathematikprogrammen mindestens ebenbürtig, wenn nicht sogar überlegen. Zudem ist die MatPlotLib kostenlos erhältlich, quelloffen und sie kann objektorientiert programmiert werden.

Mittels MatPlotLib können Diagramme und Darstellungen in verschiedenen Formen und Formaten erzeugt werden. Die dabei erzielbare Qualität reicht problemlos auch an Erfordernisse für wissenschaftliche Veröffentlichungen heran.

Dennoch können selbst Einsteiger mit MatPlotLib schnell beachtliche Erfolge erzielen. Mit nur wenigen Codezeilen lassen sich einfache x/y-Plots, Histogramme, Leistungsspektren, Balkendiagramme, Fehlerdiagramme, Streudiagramme etc. erzeugen.

Importieren der MatPlotLib-Bibliothek in Thonny

Nach dem Starten von Thonny wird zunächst der Paketmanager (Werkzeuge (Tools) → Paketmanager) geöffnet. Im Menü „Werkzeuge" (Tools) wird dann nach „MatPlotLib" gesucht (Bild 1).

Nach einem Klick auf die gefundene Bibliothek kann diese im nächsten Fenster installiert werden.

Zum Abschluss kann man noch überprüfen, ob MatPlotLib korrekt installiert und verfügbar ist.

Hierzu kann man ein neues Programmfenster öffnen und „import MatPlotLib.pyplot as plt" eingeben. Wenn kein Fehler gemeldet wurde, ist alles in Ordnung.

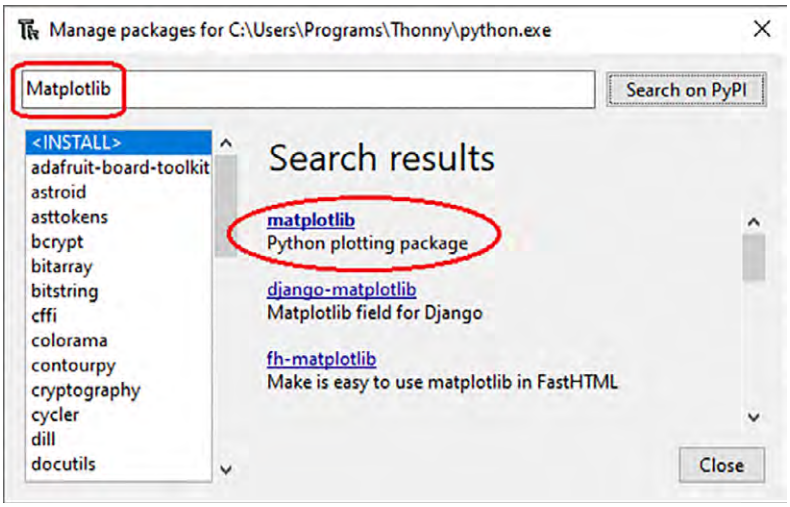

Bild 1: Installation der MatPlotLib-Bibliothek

Erste Schritte: Erstellen einfacher Diagramme

Nur einige wenige Befehle und Anweisungen bilden die Grundlage der MatPlotLib. Eines der zentralen Module der MatPlotLib ist Pyplot. Damit lassen sich einfache Funktionen wie Linien, Bilder, Texte mit wenigen Programmzeilen darstellen. Ein erstes Beispiel verdeutlicht, wie man Linien-Diagramme erstellt (siehe Line_Diagram.py):

```
import MatPlotLib.pyplot as plt

# Daten definieren
x = [1, 2, 3, 4, 5, 6]
y = [1, 4, 9, 16, 25, 36]

# Plot erstellen
plt.figure(figsize=(8,6))
plt.plot(x, y, label='squares')
plt.title('line diagram')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
plt.show()
```

Dieser Code erzeugt ein einfaches Liniendiagramm (Bild 2).

- plt.figure() erstellt eine neue Grafik („Figur")
- plt.plot() fügt dem Diagramm die definierten Daten hinzu
- plt.show() zeigt das Diagramm in einem eigenen Fenster

Mit den Funktionen plt.title(), plt.xlabel() und plt.ylabel() können dem Diagramm Titel, Beschriftungen und Achsenbezeichnungen hinzugefügt werden. Damit lassen sich bereits eine Vielzahl von Grafikaufgaben erledigen. Dennoch geben diese Befehle nur einen ersten Eindruck. Im Laufe dieses Artikels sollen die Funktionen und Möglichkeiten dieser mächtigen Bibliothek etwas detaillierter vorgestellt werden.

Zunächst besteht eine MatPlotLib-Grafik prinzipiell aus den folgenden Bestandteilen:

- **Figure**

Hierunter wird die gesamte Abbildung verstanden. Sie kann eine oder mehrere Diagramme enthalten. Eine „Figure" ist als eine Art Leinwand zu verstehen, die einzelne Diagramme enthält.

- **Axes**

Eine „Figure" kann mehrere Achsen enthalten. Jede Achse hat optional
- einen Titel
- ein X-Label
- ein Y-Label
- ein Z-Label (bei 3D-Darstellungen)

- **Axis**

Über Axis-Objekte werden die Achsen mit Maßstäben und Diagrammgrenzen versehen.

- **Artist**

Enthält Text-, Linen-, 2D- oder 3D-Objekte etc.

Mit diesen Grafikelementen lassen sich praktisch alle denkbaren Bild- und Darstellungsvariationen aufbauen.

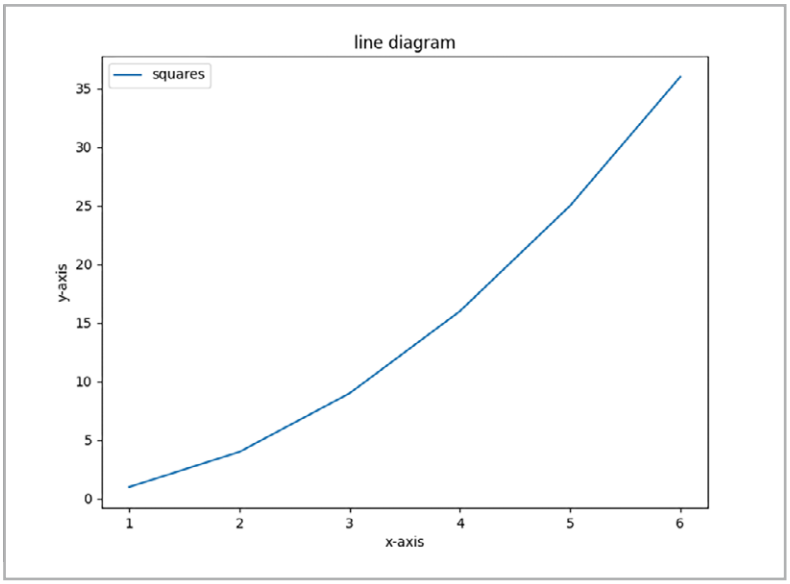

Bild 2: Liniendiagramm

Grundelemente der Visualisierung mit MatPlotLib

In MatPlotLib bezeichnet die „Figure" das gesamte Fenster oder die gesamte Seite, auf der alle Darstellungen erfolgen. Eine Figure kann eine oder mehrere „Axes" enthalten. Hierunter versteht man x/y-Diagramme, in denen Daten in Form von Linien, Balken, Punkten usw. visualisiert werden können. Eine „Axe" kann man sich als eine einzelne Plotfläche vorstellen, die ihre eigenen Achsen und Titel besitzt. Das Verständnis des Figure-Axes-Konzepts ist entscheidend für die effektive Nutzung von MatPlotLib. Das folgende Beispiel (SinCos.py) illustriert diese Methode:

```
import MatPlotLib.pyplot as plt
import numpy as np

# Beispiel-Daten
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Erstellen einer Figure mit 2 Subplots (1x2 Anordnung)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Plot im ersten Subplot
ax1.plot(x, y1, label='sin(x)')
ax1.set_title('Sinusfunktion')
ax1.set_xlabel('x')
ax1.set_ylabel('sin(x)')
ax1.legend()

# Plot im zweiten Subplot
ax2.plot(x, y2, label='cos(x)', color='orange')
ax2.set_title('Kosinusfunktion')
ax2.set_xlabel('x')
ax2.set_ylabel('cos(x)')
ax2.legend()

# Layout anpassen und anzeigen
plt.tight_layout()
plt.show()
```

Das Programm verwendet neben MatPlotLib auch die NumPy-Bibliothek. Diese ermöglicht die Einbindung wissenschaftlicher Rechenmethoden in Python. Sie stellt unter anderem Arrays, eine Vielzahl mathematischer Funktionen und Operationen sowie Funktionen zur Datenmanipulation zur Verfügung. Weitere Details zur NumPy-Bibliothek folgen in späteren Beiträgen.

In obigem Programm werden zunächst Beispieldaten für Sinus- und Kosinusfunktionen erzeugt. Dabei ist np.linspace eine Funktion aus der NumPy-Bibliothek, die ein Array von 100 gleichmäßig verteilten Punkten zwischen 0 und 10 zur Verfügung stellt. Dieses Array x wird als unabhängige Variable für die Funktionen sin und cos verwendet. Mit np.sin kommt eine Funktion aus der NumPy-Bibliothek zum Einsatz, die den Sinus jedes Elements im Array x berechnet. Das Ergebnis ist ein Array y1, das die Sinuswerte der entsprechenden x-Werte enthält.

Entsprechend wird mit np.cos ein Array y2 für die Kosinuswerte berechnet. Diese drei Zeilen generieren also die Daten, die in den Subplots geplottet werden. Die x-Werte sind die Eingaben (die horizontale Achse) und y1 und y2 sind die Ausgaben (die vertikale Achse) für die Sinus- und Kosinusfunktionen. Zugleich demonstrieren sie eindrucksvoll die Leistungsfähigkeit der NumPy-Bibliothek.

Nach der Erzeugung von Daten wird eine „Figure" mit zwei Subplots nebeneinander (1 Reihe, 2 Spalten) aufgebaut. Im ersten Subplot (ax1) wird die Sinusfunktion geplottet, im zweiten Subplot (ax2) die Kosinusfunktion dargestellt.

Die Methode plt.tight_layout() sorgt dafür, dass die Subplots nicht überlappen und alle Daten lesbar sind. Dies ist eine typische Python-Funktion, die versucht, alles so gut wie möglich (best effort) zu erledigen. Bild 3 zeigt das Ergebnis.

Farben, Linien und Marker

Die visuelle Gestaltung eines Diagramms ist entscheidend, um eine klare und verständliche Darstellung der Daten zu gewährleisten. MatPlotLib bietet eine breite Palette von Stil-Optionen, um Diagramme nach Wunsch anzupassen. Zunächst können Farben beliebig variiert werden. Linienfarben, Farbmarkierungen, Achsen- und Titelfarben sind frei wählbar. Sie können als HTML-Hex-Strings, HTML-Farbnamen oder als RGB-Tupel definiert werden, z. B. definiert

```
plt.plot(x, y, color="green")
```

die Linienfarbe als Grün.

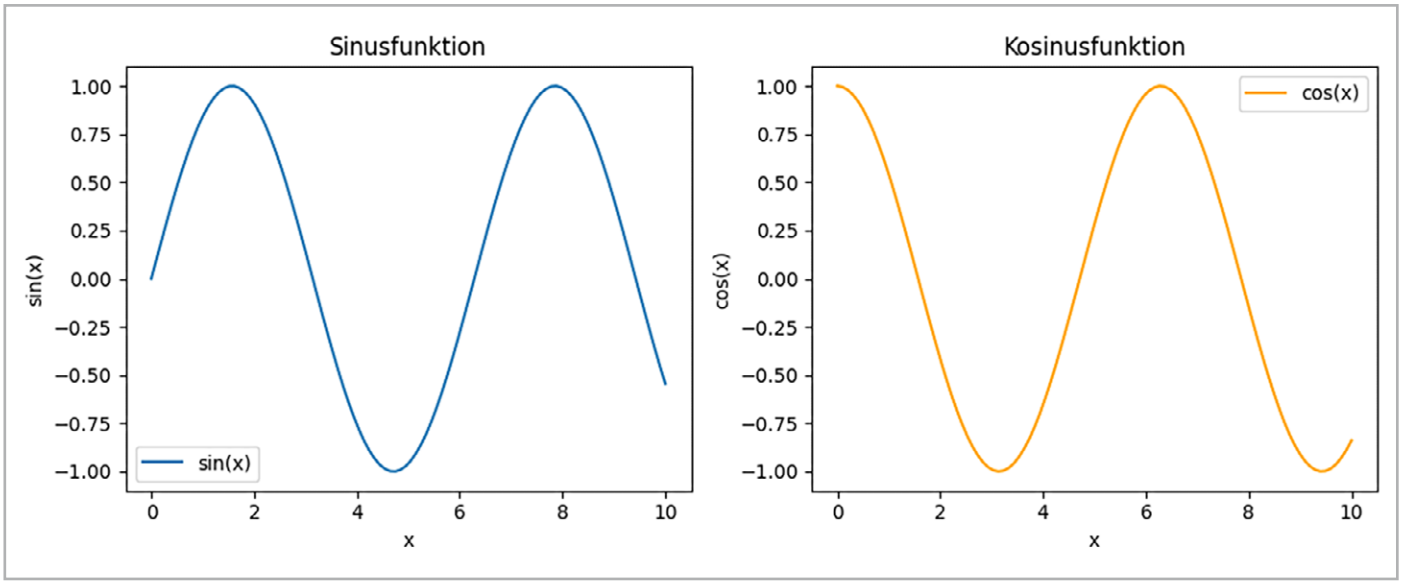

Bild 3: Subplots

Dann kann über Linienparameter und Marker-Anweisungen der Linientyp festgelegt werden. Mögliche Linienparameter sind durchgezogen, gestrichelt, gepunktet und andere Varianten.

Die Form der Datenpunkte (Kreise, Dreiecke, Quadrate usw.) kann ebenfalls angepasst werden. Das Beispiel

plt.plot(x, y, linestyle='--', marker='o', color='r')

liefert gestrichelte Linien mit Kreismarkern in rot (siehe Bild 4, Programm Styles_marker.py):

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

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3))

ax1.plot(x, y1, label='sin(x)', color='g', linestyle=':', marker='.')
ax2.plot(x, y2, label='cos(x)', color='r', linestyle='--', marker='o')

plt.tight_layout()
plt.show()
```

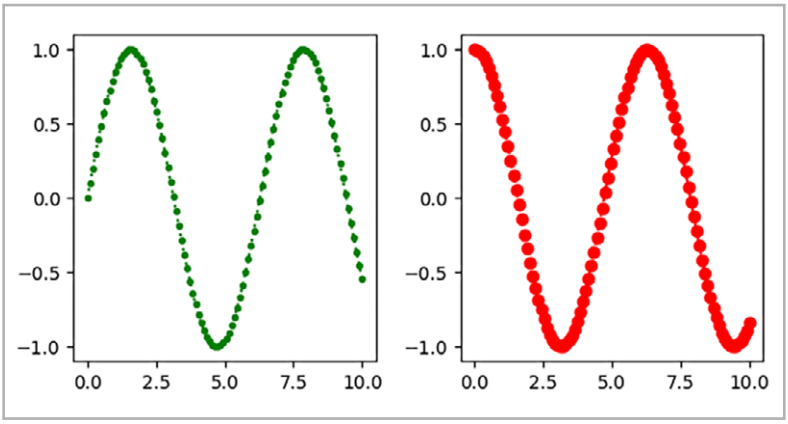

Bild 4: Festlegung von Linientypen und Markern

Text und Annotationen für Erklärungen und Beschriftungen

MatPlotLib ermöglicht es auch, Texte innerhalb des Diagramms zu platzieren. Zudem können Beschriftungen für Achsen, Titel für die Achsen und die Figur sowie Legenden hinzugefügt werden. Mit sogenannten Annotationen kann Text an einer bestimmten Position im Diagramm platziert werden.

Mit Pfeilen kann man auf Bereiche mit besonderer Bedeutung hinweisen:

ax.annotate('Wichtiger Punkt', xy=(2, 1), xytext=(3, 1.5), arrowprops=dict(facecolor='black', shrink=0.05))

Wenn Diagramme mit Texten, Beschriftungen und Annotationen versehen werden, kann dies die Aussagekraft und Klarheit von Visualisierungen entscheidend verbessern.

Einige Möglichkeiten, wie Text und Annotationen in MatPlotLib verwendet werden können, sind:

Titel und Überschriften: Sie helfen, den Inhalt auf den ersten Blick zu verstehen.
Achsenbeschriftungen erklären, welche Werte die Achsen darstellen und geben den Daten einen klaren Kontext.
Legenden helfen, verschiedene Datenreihen oder Kategorien innerhalb eines Diagramms zu identifizieren.
Annotationen können als spezifische Hinweise oder Markierungen an besonderen Datenpunkten angebracht werden.
Text kann an beliebigen Stellen im Diagramm platziert werden, um bestimmte Bereiche zu beschreiben oder zu kommentieren.

Bild 5 veranschaulicht dies für eine Anwendung in der Wechselstromtechnik.

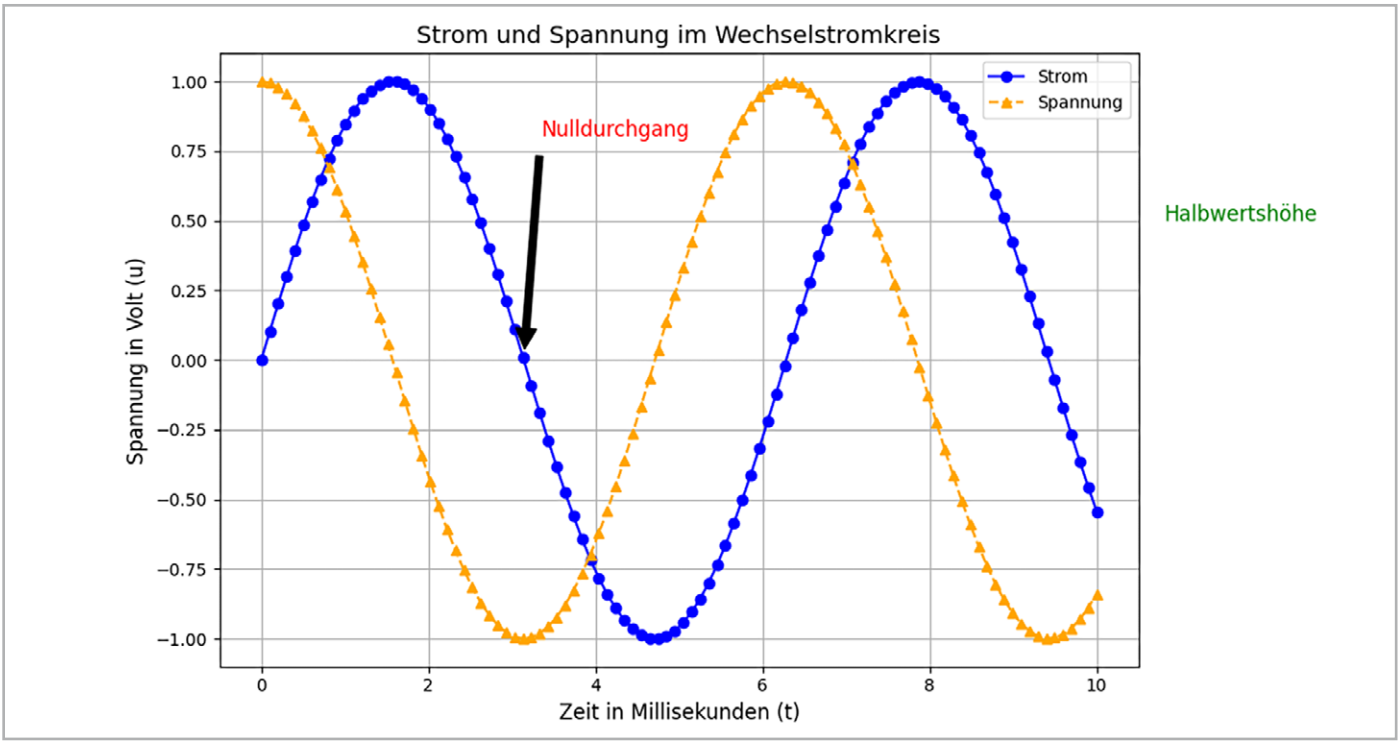

Bild 5: Festlegung von Linientypen und Markern in der Wechselstromtechnik

Das Programm (E_tech.py) zu dieser Grafik sieht so aus:

```
import MatPlotLib.pyplot as plt
import MatPlotLib.pyplot as plt
import numpy as np

# Beispiel-Daten
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.figure(figsize=(10, 6))

plt.plot(x, y1, label='Strom', color='blue', linestyle='-', marker='o')
plt.plot(x, y2, label='Spannung', color='orange', linestyle='--', marker='^')

plt.title('Strom und Spannung im Wechselstromkreis', fontsize=14)

plt.xlabel('Zeit in Millisekunden (t)', fontsize=12)
plt.ylabel('Spannung in Volt (u)', fontsize=12)

plt.legend()
plt.annotate('Nulldurchgang', xy=(np.pi, np.sin(np.pi)), xytext=(np.pi+0.2, np.sin(np.pi)+0.8),
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=12, color='red')

plt.text(10.8, 0.5, 'Halbwertshöhe', fontsize=12, color='green')

plt.grid(True)
plt.tight_layout()
plt.show()
```

Das Programm erstellt ein Diagramm mit einer Sinus- und einer Kosinusfunktion, die in „Strom“ und „Spannung“ im Wechselstromkreis bezeichnet werden. Zunächst werden hierzu die erforderlichen Bibliotheken importiert:
- MatPlotLib.pyplot wird für das Plotten von Grafiken verwendet.
- numpy wird genutzt, um mathematische Berechnungen durchzuführen und Arrays zu erzeugen.

Dann werden die Beispieldaten erzeugt. Dazu wird ein Array (x) von 100 gleichmäßig verteilten Punkten zwischen 0 und 10 erstellt. Die Sinusfunktion (y1) von x wird berechnet und gespeichert, sie repräsentiert den „Strom“. Dann wird die Kosinusfunktion (y2) von x berechnet, sie repräsentiert die „Spannung“.

Danach wird ein Plot-Fenster erstellt. Die Sinuskurve (y1) wird mit einer durchgezogenen blauen Linie und Kreismarkern dargestellt, die Cosinuskurve (y2) mit einer gestrichelten orangen Linie und Dreieckmarkern.

Anschließend wird der Titel „Strom und Spannung im Wechselstromkreis“ hinzugefügt.
Zudem wird die x-Achse mit „Zeit in Millisekunden (t)“ und die y-Achse mit „Spannung in Volt (u)“ beschriftet.

Die Legende kennzeichnet die beiden Kurven als „Strom“ und „Spannung“. Eine Annotation weist auf den „Nulldurchgang“ der Sinusfunktion hin. Dies wird durch einen roten Text und einen Pfeil markiert, der auf den Punkt bei t = → zeigt, an dem der Sinuswert 0 ist.

Dann wird ein grüner Text „Halbwertshöhe“ als zusätzliche Information am oberen rechten Rand hinzugefügt. Zudem wird ein x/y-Raster eingeblendet (plt.grid(True)), um das Ablesen der Kurven zu erleichtern. Das Layout wird so angepasst, dass alles gut sichtbar und nicht überlappend dargestellt wird (plt.tight_layout()).

Zum Schluss wird das Diagramm mit plt.show() angezeigt. Dieses Programm veranschaulicht die zeitabhängige Veränderung von Strom und Spannung in einem Wechselstromkreis und nutzt dabei grundlegende mathematische Funktionen (Sinus und Kosinus) zur Modellierung.

Diagrammtypen in allen Variationen

Neben den oben vorgestellten Liniendiagrammen bietet MatPlotLib noch eine Vielzahl weiterer Darstellungsmöglichkeiten.

Balkendiagramme eignen sich hervorragend, um Daten zu vergleichen. Sie können verwendet werden, um auch große Gruppen von Datenpunkten schnell zu erfassen. Die Anweisung

plt.bar(x, y)

erstellt ein vertikales Balkendiagramm. Balkendiagramme eignen sich zudem bestens, um Unterschiede und Trends zwischen verschiedenen Kategorien klar und übersichtlich darzustellen.

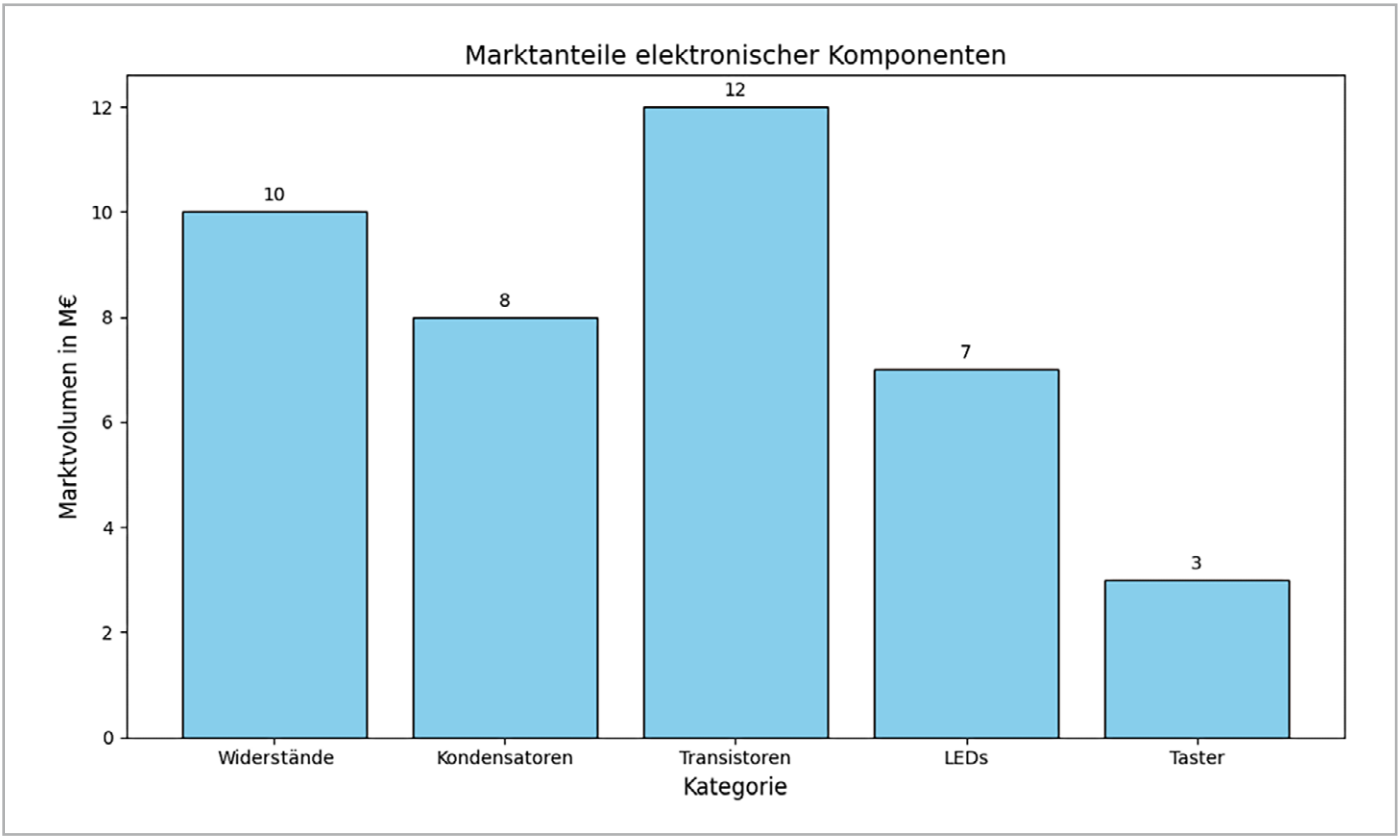

Bild 6: Balkendiagramm

Sie werden häufig im Bereich Wirtschaft und Finanzen verwendet. Hier dienen sie zum Vergleich von Umsätzen, Gewinnen oder Kosten in verschiedenen Zeiträumen oder Unternehmen. Auch beim Vergleich von Umfrage- oder Testergebnissen sind sie oft zu finden.

In der Politik dienen sie häufig der Darstellung von Bevölkerungsdaten wie Altersgruppen, Geschlecht, Bildung oder Einkommen, der Visualisierung von Wahlergebnissen oder politischen Präferenzen.

Bild 6 zeigt ein typisches Balkendiagramm aus dem Bereich der Marktanalyse. Das zugehörige Programm (BarGraphics.py) fällt recht kompakt aus:

```
import MatPlotLib.pyplot as plt
import numpy as np
# Beispiel-Daten
kategorien = ['Widerstände', 'Kondensatoren', 'Transistoren', 'LEDs', 'Taster']
werte = [10, 8, 12, 7, 3]

plt.figure(figsize=(10, 6))

plt.bar(kategorien, werte, color='skyblue', edgecolor='black')

plt.title('Marktanteile elektronischer Komponenten', fontsize=14)
plt.xlabel('Kategorie', fontsize=12)
plt.ylabel('Marktvolumen in M€', fontsize=12)

for i, value in enumerate(werte):
    plt.text(i, value + 0.2, str(value), ha='center', fontsize=10)

plt.tight_layout()
plt.show()
```

Der Python-Code erstellt ein Balkendiagramm, das den Marktanteil verschiedener elektronischer Komponenten darstellt. Hierzu wird eine Liste mit Kategorien verschiedener elektronischer Komponenten (Widerstände, Kondensatoren, Transistoren, LEDs und Taster) erzeugt. Die Werte-Liste enthält die entsprechenden Marktvolumen dieser Komponenten in Millionen Euro (M€).

Die Kategorien werden auf der x-Achse angezeigt, während die Marktvolumen auf der y-Achse dargestellt werden. Die Balken werden in Hellblau (color=skyblue) mit schwarzen Rändern (edgecolor='black') gezeichnet.

Der Titel des Diagramms lautet „Marktanteile elektronischer Komponenten“ und wird mit plt.title() angezeigt.

Dann wird die x-Achse mit „Kategorie“ und die y-Achse mit „Marktvolumen in M€“ beschriftet. Zudem werden die Werte (Marktvolumen) direkt oberhalb der Balken als Text angezeigt, um den visuellen Vergleich zu ergänzen. Abschließend wird wieder das Layout des Diagramms angepasst, um Überlappungen zu vermeiden (tight_layout()).

Kreis- oder Tortendiagramme werden verwendet, um Proportionen und Anteile zu zeigen. Sie sind sehr effektiv, wenn es darum geht, den Beitrag eines Teils zum Ganzen darzustellen. Bild 7 zeigt ein entsprechendes Beispiel (PiePlot.py).

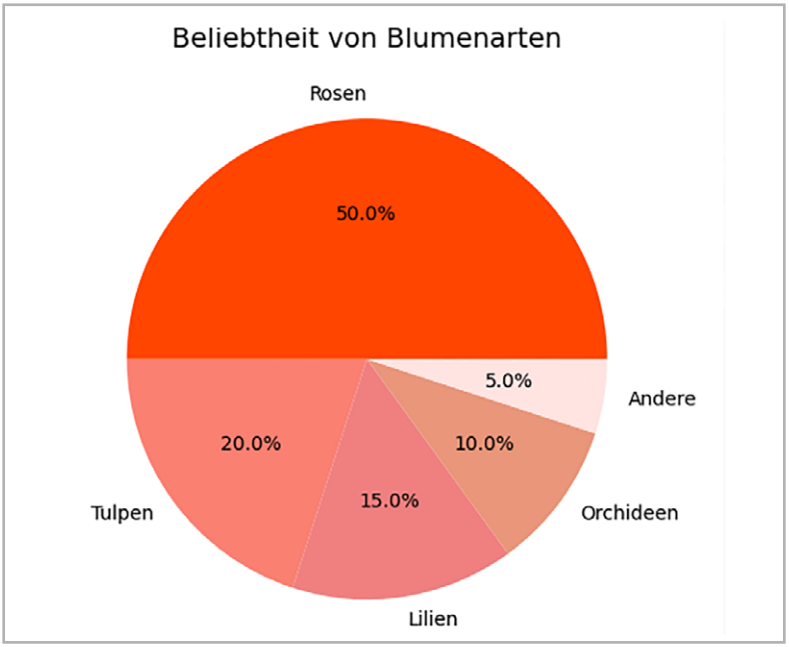

Bild 7: Kreisdiagramm

Das Programm dazu sieht so aus:

```
import matplotlib.pyplot as plt

# Beispiel-Daten
kategorien = ['Rosen', 'Tulpen', 'Lilien', 'Orchideen', 'Andere']
werte = [50, 20, 15, 10, 5]

plt.figure(figsize=(8, 8))

plt.pie(werte, labels=kategorien, autopct='%1.1f%%', colors=['orangered','salmon','lightcoral','darksalmon','mistyrose'], startangle=0)

plt.title('Beliebtheit von Blumenarten', fontsize=14)
plt.tight_layout()
plt.show()
```

Die Anweisung

plt.pie(sizes, labels=labels, autopct='%1.1f%%')

sorgt hier dafür, dass die Daten in einem klassischen Tortendiagramm angezeigt werden. Der Rest des Programms erstellt in bekannter Weise die Überschrift und die Bezeichnungen.

Violinplots sind eine weniger häufig genutzte Variante von Boxplots und Dichtediagrammen. Sie sollen hier jedoch ebenfalls angesprochen werden, da sie zeigen, wie vielfältig Python eingesetzt werden kann. Violinplots ermöglichen es, die Dichte der Daten bei verschiedenen Werten anschaulich darzustellen.

Das folgende Beispiel (Violinplot.py) erzeugt den Violinplot in Bild 8.

```
import MatPlotLib.pyplot as plt
import numpy as np

# Daten erstellen
np.random.seed(10)
data = [np.random.normal(loc=0, scale=1, size=100),
        np.random.normal(loc=1, scale=2, size=100),
        np.random.normal(loc=-1, scale=1.5, size=100),
        np.random.normal(loc=2, scale=1, size=100)]

plt.figure(figsize=(10, 6))
plt.violinplot(data, showmeans=False, showmedians=True, showextrema=True)

plt.title('Ausfallwahrscheinlichkeiten von Bauelementen')
plt.xlabel('Gruppe')
plt.ylabel('Wert')
plt.xticks([1, 2, 3, 4], ['Elko', 'Tantals', 'Ceramics', 'Styroflex'])  # x-Achsen-Beschriftungen

plt.show()
```

Hier wird wieder NumPy verwendet, um einen Testdatensatz zu erstellen. Mit

```
np.random.seed(10)
```

wird der Startpunkt für den Zufallszahlengenerator gesetzt.

Über data = [...] werden vier Datengruppen erstellt, jede mit 100 Werten („size“). Diese Gruppen repräsentieren Daten mit unterschiedlichen Mittelwerten (loc) und Standardabweichungen (scale):

Gruppe 1: Mittelwert = 0, Standardabweichung = 1
Gruppe 2: Mittelwert = 1, Standardabweichung = 2
Gruppe 3: Mittelwert = -1, Standardabweichung = 1,5
Gruppe 4: Mittelwert = 2, Standardabweichung = 1

Der Code visualisiert dann die Verteilung von vier Datengruppen. Jede Gruppe besteht aus den 100 zufälligen Werten mit unterschiedlichen Mittelwerten und Standardabweichungen. Der Plot zeigt die Verteilung jeder Gruppe, einschließlich der Mediane und Extremwerte, und verwendet spezifische Beschriftungen für die x-Achse („Gruppe“).

Fortgeschrittene Diagrammtypen: Konturdiagramme, 3D-Plots und mehr

Um den Rahmen dieses Beitrags nicht zu sprengen, soll die Darstellung von 3D- oder Konturdiagrammen hier nur kurz angesprochen werden.

Konturdiagramme werden verwendet, um dreidimensionale Daten zweidimensional darzustellen. Sie sind besonders nützlich, um Höhenlinien in topografischen Darstellungen zu zeigen oder um Optimierungsschwerpunkte z. B. bei KI-Anwendungen zu visualisieren.

3D-Plots sind hilfreich, um Beziehungen in Daten zu visualisieren, die drei Dimensionen haben. MatPlotLib bietet Werkzeuge, um 3D-Linien-, Balken- und Streudiagramme zu erstellen.

Diese fortgeschrittenen Diagrammtypen eröffnen neue Möglichkeiten für die Datenanalyse und -visualisierung, indem sie es ermöglichen, komplexe Muster und Beziehungen auf einfache und intuitive Weise darzustellen. Im folgenden Kapitel werden verschiedene Verfahren der Datenbearbeitung und -verarbeitung sowie der Visualisierung vorgestellt, um die Grundlage für effektive und aussagekräftige Diagramme zu legen.

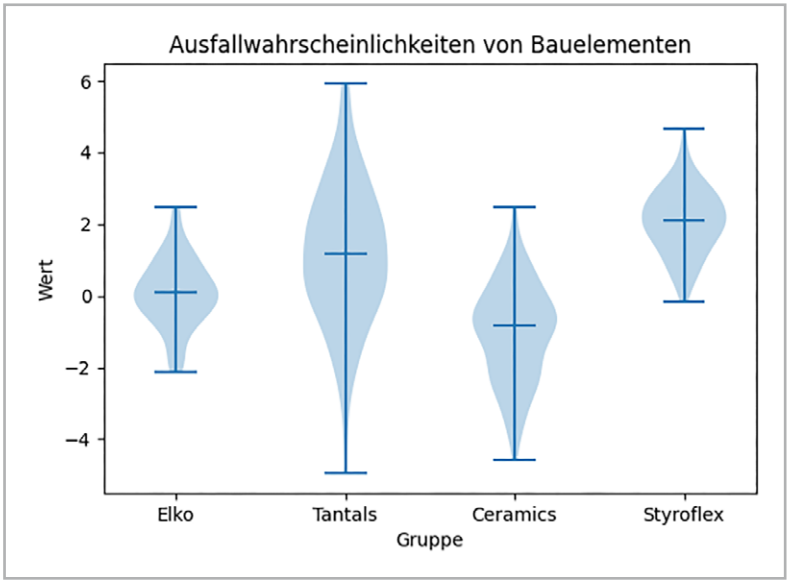

Bild 8: Violinplot

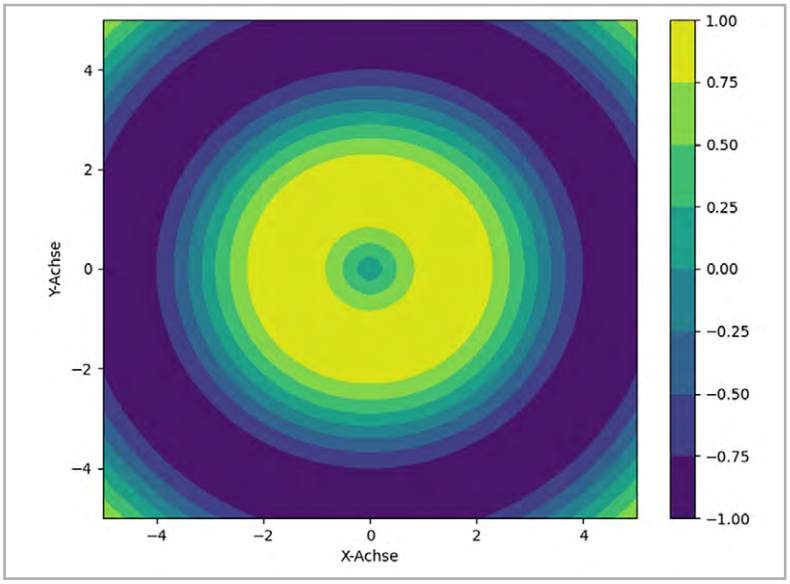

Bild 9: Konturplot

Bild 9 und Bild 10 zeigen jeweils ein Beispiel für einen Kontur- und einen 3D-Plot.

Im Konturplot-Programm (Kontur.py):

```
import numpy as np
import MatPlotLib.pyplot as plt

# Erstellen von Beispiel-Daten
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

plt.figure(figsize=(8, 6))
contour = plt.contourf(X, Y, Z, cmap='viridis')
plt.colorbar(contour)  # Farbskala hinzufügen

plt.xlabel('X-Achse')
plt.ylabel('Y-Achse')

plt.show()
```

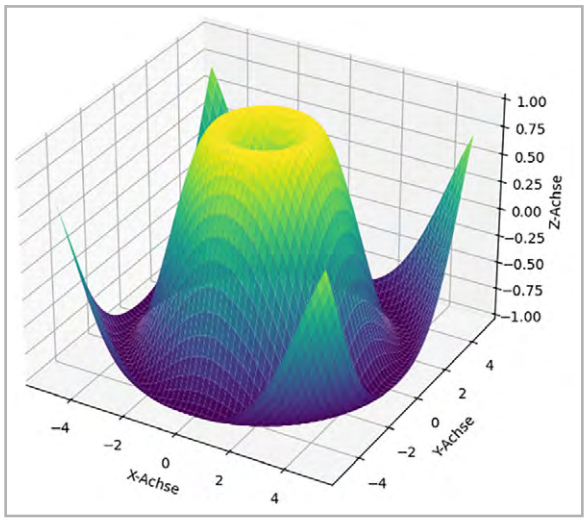

Bild 10: 3D-Plot

werden wieder zunächst Daten erstellt. Die Ausführung des Plots erfolgt mit plt.contourf(), mit cmap='viridis' wird die Farbkarte festlegt. Plt.colorbar fügt eine Farbskala hinzu. Es folgen die Beschriftungen und schließlich wird der Plot wieder mit plt.show() angezeigt.

Das 3D-Programm (3D-Plot.py) arbeitet ähnlich wie der Konturplot:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Erstellen von Beispiel-Daten
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Erstellen des 3D-Plots
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='coolwarm')

# Achsenbeschriftungen und Titel
ax.set_title('3D-Plot')
ax.set_xlabel('X-Achse')
ax.set_ylabel('Y-Achse')
ax.set_zlabel('Z-Achse')

# Plot anzeigen
plt.show()
```

Die Daten werden durch np.linspace erzeugt. Die Funktion np.meshgrid erstellt daraus 2D-Gitter für X und Y. Z enthält die Funktionswerte, hier durch eine abfallende Sinusfunktion definiert. Die Anweisung

ax = fig.add_subplot(111, projection='3d')

führt den eigentlichen 3D-Plot aus.

Mit ax.plot_surface(X, Y, Z, cmap='viridis') wird die gleiche Farbkarte (viridis) wie im Konturplot verwendet.

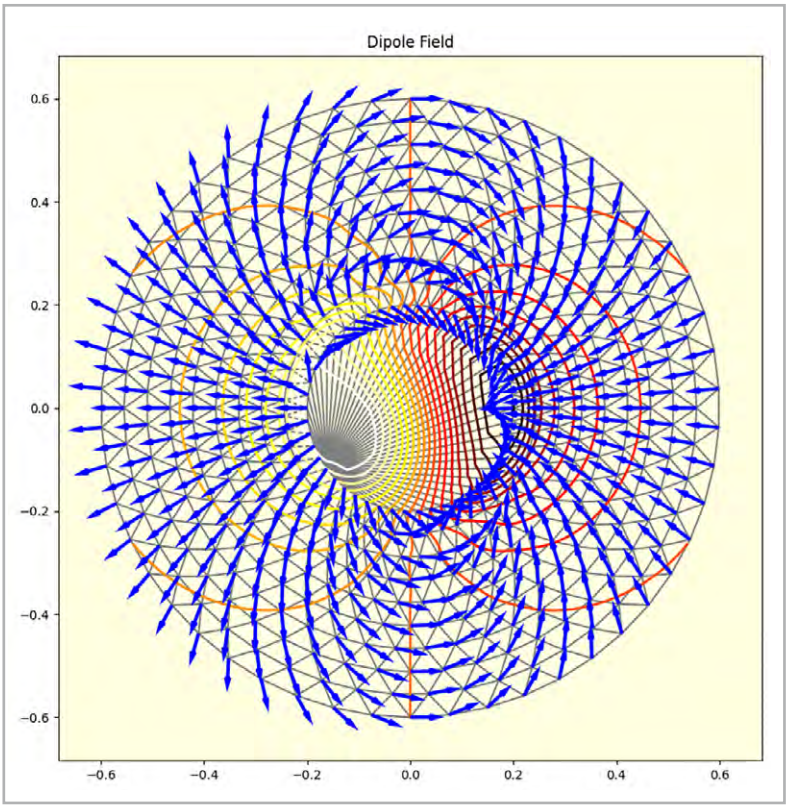

Bild 11: Komplexe Gradientengrafik

Komplexe Grafiken

Einfache 2D/3D-, Balken- oder Tortengrafiken können auch mit anderen Programmen wie Excel oder LibreOffice erstellt werden. Bei komplexeren Grafikaufgaben, die eine Vielzahl von Elementen enthalten, stoßen Excel oder LibreOffice jedoch schnell an Grenzen. Hier kann MatPlotLib seine volle Stärke ausspielen.

Bild 11 zeigt die Darstellung eines elektrischen Dipols sowohl mit Feldlinien als auch mit Äquipotentialflächen. Zudem wurde hier nicht die klassische X/Y-Darstellung, sondern ein Polarkoordinatensystem gewählt.

Das Programm hierzu ist naturgemäß bereits etwas umfangreicher (Field.py). Wenn dieser Artikel sorgfältig durchgearbeitet wurde, sollte das Verständnis des Codes allerdings kein Problem mehr darstellen, da alle verwendeten Bild- sowie Codeelemente bereits eingesetzt und erläutert wurden.

Unbegrenzte Möglichkeiten: Animationen

Mit Python ist es nicht nur möglich, statische Grafiken darzustellen, sondern auch bewegte Animationen umzusetzen. Das folgende Programm (QuantumAnimation.py) erzeugt eine animierte, durchlaufende Wellenstörung. Bild 12 zeigt einen stehenden Ausschnitt. Die durchlaufende Animation kann natürlich nur nach dem Starten des Python-Programms dargestellt werden.

Die Erstellung animierter Grafiken erfordert dabei ein tiefergehendes Verständnis der Grafikprogrammierung. Eine ausführliche Diskussion des zugehörigen Programms wird hier nicht beschrieben, da dies den Rahmen dieses Artikels sprengen würde. Das Beispiel soll lediglich zeigen, dass auch der Raspberry Pi in der Lage ist, animierte Grafiken flüssig darzustellen.

Ergänzungen und Übungen

- Erstellen Sie ein Liniendiagramm, das den Bremsweg eines Fahrzeugs darstellt.
- Verändern Sie die Farben und Symboldarstellungen in den Bildern 2 und 3.
- Erstellen Sie ein Balkendiagramm für den Ausgang einer Bundestagswahl.
- Experimentieren Sie mit verschiedenen Farbkarten und färben Sie die Bilder 9 und 10 mit verschiedenen Colormaps wie z. B.
  - plasma: von dunkelviolett zu gelb
  - inferno: von schwarz zu gelb
  - magma: von schwarz zu weiß
  - cividis: von dunkelblau zu gelb

Hinweis

Die im Text grün markierten Programme sind im Download-Paket enthalten.

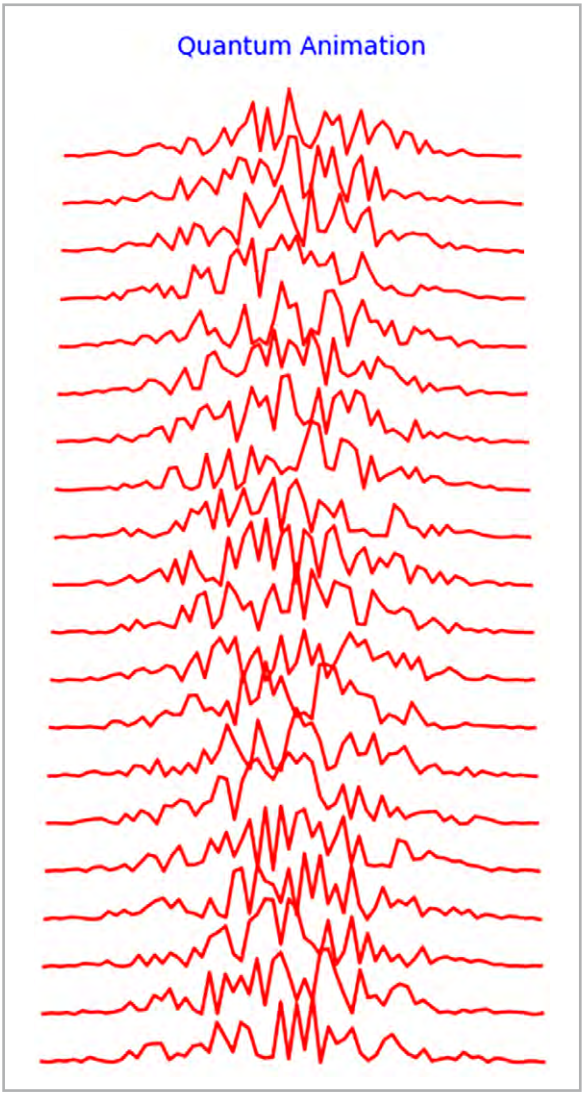

Bild 12: Simulation einer Wellenstörung

Ausblick

In diesem Beitrag stand die Erstellung von Grafiken mit Python im Vordergrund. Es wurde gezeigt, dass Python ein leistungsfähiges Werkzeug zur Erstellung verschiedenster Visualisierungen ist. Dabei können nicht nur einfache XY- oder Balkendiagramme erstellt werden, sondern auch durchaus komplexe Grafiken, die mit anderen Programmen wie z. B. Excel oder LibreOffice kaum bzw. gar nicht machbar wären. Selbst animierte Grafiken stellen für Python und den Raspberry Pi kein Problem dar.

Im nächsten Artikel wird es wieder etwas technischer werden, insbesondere werden dann Eingabemethoden genauer betrachtet.

Beginnend mit einfachen Tastern spannt sich der Bogen dabei von Tastern über Dreh-Encoder bis hin zu Matrix-Tastaturen. Dabei wird immer auch wieder auf grafische Ausgaben mit der MatPlotLib zurückgegriffen, um z. B. Eingabedaten visuell ansprechend darzustellen. ELV

Material

Raspberry Pi und Netzteil

z. B. Raspberry Pi 4 Model B, 8 GB RAM	Artikel-Nr. 250567
z. B. Raspberry Pi 4 USB-Netzteil Typ C	Artikel-Nr. 250962

Zum Download-Paket

Python & MicroPython: Alle bisherigen Teile im Überblick

Teil 1: Erste Schritte
Zum Beitrag

Teil 2: GPIOs steuern die Welt
Zum Beitrag

Teil 3: Digitale Logik
Zum Beitrag

Teil 4: Ablaufsteuerung und Programmstrukturen
Zum Beitrag

Teil 5: Erfassung analoger Werte
Zum Beitrag