Dokumentation Matplotlib

March 2024

Contents

1 Grundlagen des Plotens

1.1 Graphische Darstellung von Plots

plt.plot(x,y) erstellt einen Liniengraphen, indem es die Werte aus x auf die x-Achse und y auf der y-Achse abbildet.

```
1  x_1 = np.linspace(0,5,10)
2  y_1 = x_1**2
3  plt.plot(x_1,y_1) # Einfaches Plotten von Dateien,
4  #in einem einzelnen Diagramm
```

Listing 1: Linien Graph ploten

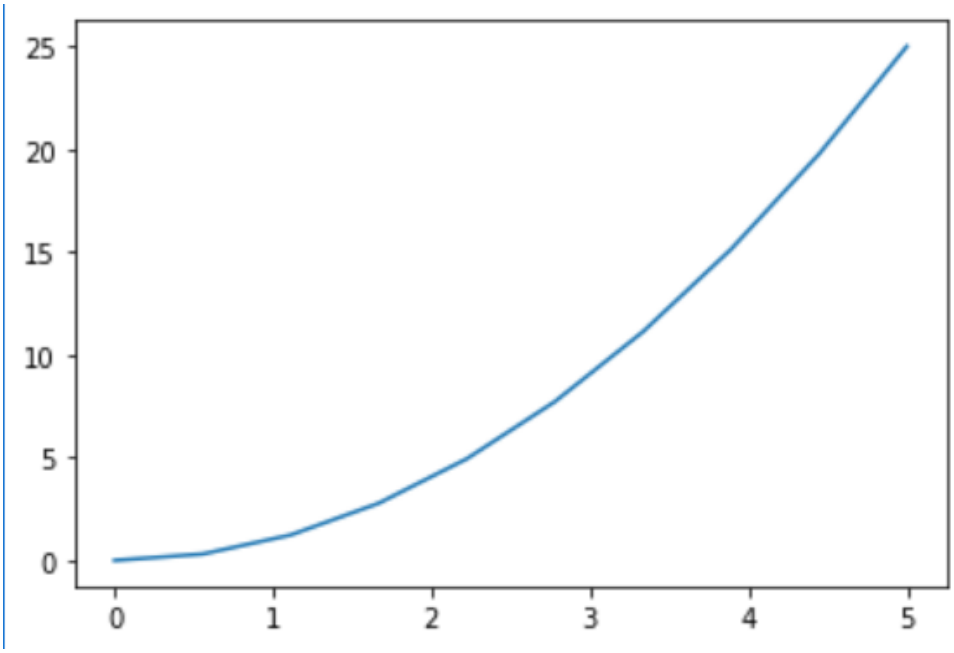

Figure 1: Linien Plot

1.2 Linienstil und Linientypen

Zeichen	Beschreibung
-	Durchgezogene Linie
–	Gestrichelte Line
-.	Strichpunkt-Linie
:	Puntierte Linie

```
1  fig_2, axes_2 = plt.subplots(figsize=(8,4), nrows=1,ncols=4)
2  plt.tight_layout()
3
4  axes_2[0].set_title('Durchgezogene Linie')
5  axes_2[0].plot(x_1,y_1,'-')
6  axes_2[1].set_title('Gestrichelte Linie')
7  axes_2[1].plot(x_1,y_1,'--')
8  axes_2[2].set_title('Strichpunkt-Linie')
9  axes_2[2].plot(x_1,y_1,'-.')
10 axes_2[3].set_title('Punktierte Linie')
11 axes_2[3].plot(x_1,y_1,':')
```

Listing 2: Linien Stile

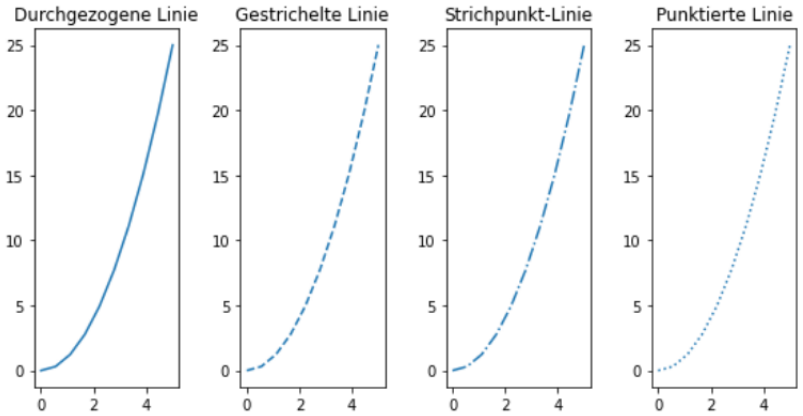

Figure 2: Linienstil und Linientypen

1.3 Linien-Style inderviduell anpassen

Der Parameter **dashes=[Linienlänge, Lückelänge, Linienlänge, ...]** ist ein Teil der Funktion "ax.plot", die verwendet wird, um den Linienstil individuell anzupassen, indem er die Länge der Linienabschnitte und Lücken festlegt.

```
1 fig = plt.figure(figsize=(5,4))
2 axes = fig.add_axes([0,0,1,1])
3 axes.grid(True)
4 axes.set_title("Linien-Style inderviduell")
5 axes.plot(x_1,y_1, dashes=[2, 2, 10, 2], lw=3)
```

Listing 3: Linien-Style

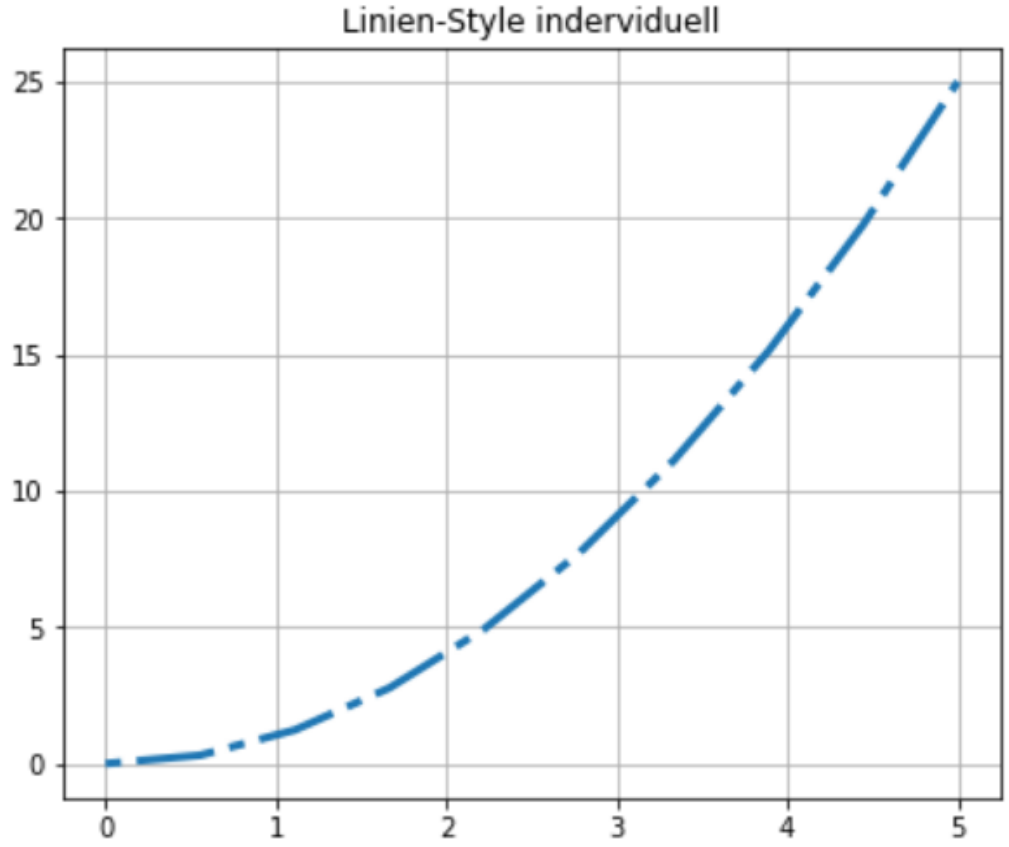

Figure 3: Linien-Style indervideuell anpassen

1.4 Linienstil und Kontur

Mit **linewidth** oder **lw** innerhalb der Plot Funktion, kann man die Linien-Breite ändern.

```
1  fig_2, axes_2 = plt.subplots(figsize=(8,4), nrows=1,ncols=4)
2  plt.tight_layout()
3
4  axes_2[0].set_title('linewidth=6')
5  axes_2[0].plot(x_1,y_1,linewidth=6)
6  axes_2[1].set_title('linewidth=12')
7  axes_2[1].plot(x_1,y_1,linewidth=12)
8  axes_2[2].set_title('linewidth=18')
9  axes_2[2].plot(x_1,y_1,linewidth=18)
10 axes_2[3].set_title("linewidth=24")
11 axes_2[3].plot(x_1,y_1,linewidth=24)
```

Listing 4: Linienstil und Kontur

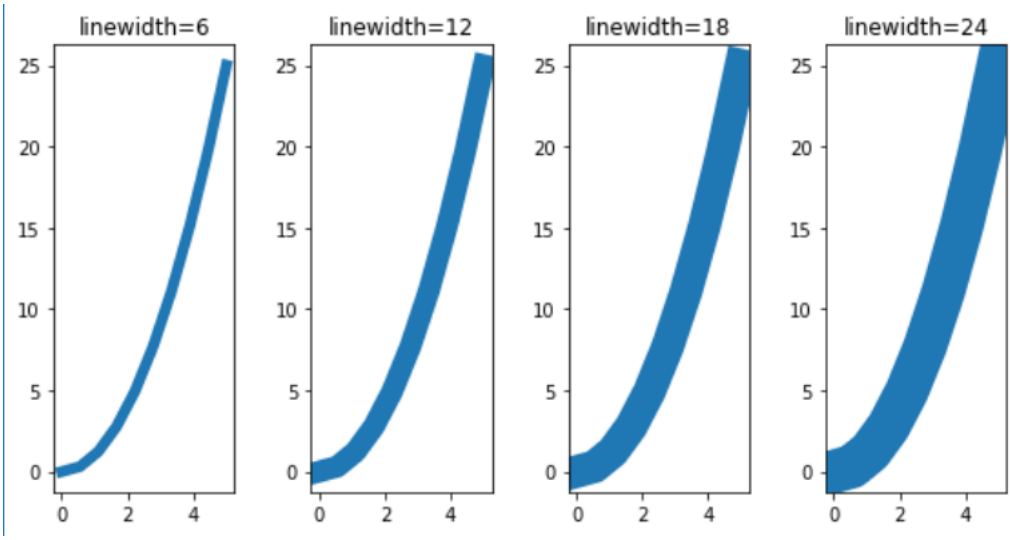

Figure 4: Linienstil und Kontur

1.5 Transparenz von Linien

Mit der Funktion **alpha=** innerhalb der Plot-Funktion kann man die Transparenz des Plots anpassen. Der Alpha-Wert ist eine Dezimalzahl zwischen 0 und 1, wobei 0 für vollständig transparent steht und 1 für vollständig undurchsichtig steht.

```
fig_1, axes_1 = plt.subplots(figsize=(8,4), nrows=1, ncols=4)
plt.tight_layout()
axes_1[0].set_title('25% Alpha')
axes_1[0].plot(x_1,y_1,alpha=0.25, lw=4)
axes_1[1].set_title('50% Alpha')
axes_1[1].plot(x_1,y_1,alpha=0.5, lw=4)
axes_1[2].set_title('75% Alpha')
axes_1[2].plot(x_1,y_1,alpha=0.75, lw=4)
axes_1[3].set_title('100% Alpha')
axes_1[3].plot(x_1,y_1,alpha=1,lw=4)
```

Listing 5: Transparenz von Linien

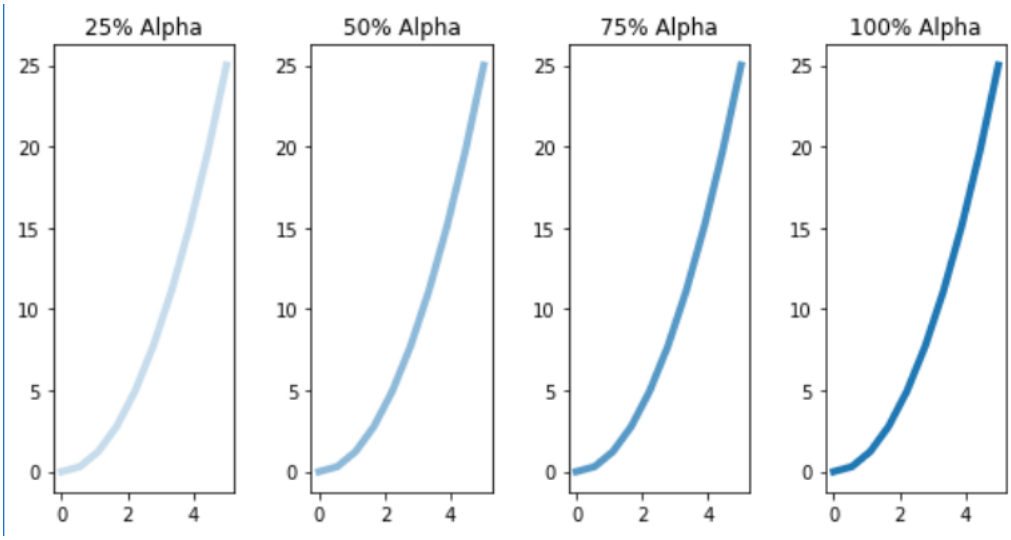

Figure 5: Transparenz von Linien

1.6 Verwendung von Marker

Nr.	Zeichen	Beschreibung
1.	.	Punkt-Marker
2.	,	Pixel-Marker
3.	o	Kreis-Marker
4.	v	Dreiecks-Marker 1
5.	^	Dreiecks-Marker 2
6.	¡	Dreiecks-Marker 3
7.	¿	Dreiecks-Marker 4
8.	1	Tri-Runter-Marker
9.	2	Tri-Hoch-Marker
10.	3	Tri-Links-Marker
11.	4	Tri-Rechts-Marker
12.	s	Quadratischer-Marker
13.	p	Fünfeckiger-Marker
14.	h	Sechseck-Marker 1
15.	H	Sechseck-Marker 2
16.	+	Plus-Marker
17.	x	x-Marker
18.	D	Rautenförmiger-Marker
19.	d	Dünner Rautenförmiger-Marker
20.	—	Vertikale Linien-Marker
21.	-	Horizontale Linien-Marker
22.	*	Stern-Marker

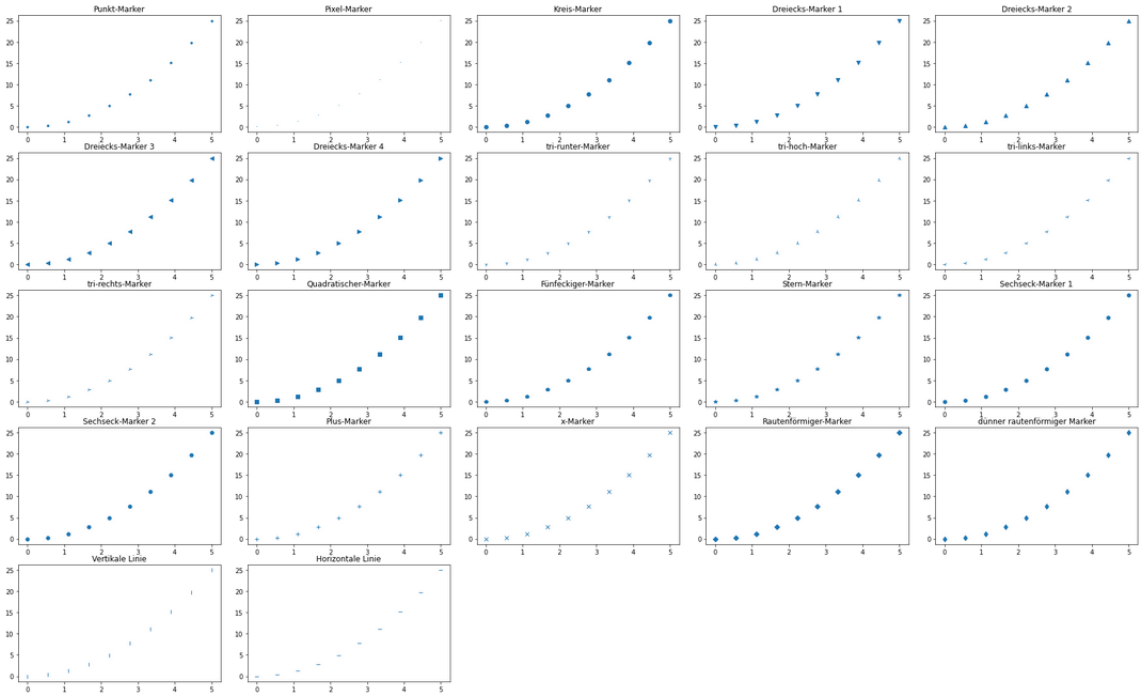

Figure 6: Verwendung von Markern

1.7 Anpassung der Markergröße

Mit **markersize** oder **ms** innerhalb der Plot-Funktion, kann man die Größe der Marker ändern.

```
1  fig_2, axes_2 = plt.subplots(figsize=(8,4), nrows=1,ncols=4)
2  plt.tight_layout()
3
4  axes_2[0].set_title('markersize=3')
5  axes_2[0].plot(x_1,y_1,'x',markersize=3)
6  axes_2[1].set_title('markersize=6')
7  axes_2[1].plot(x_1,y_1,'x',markersize=6)
8  axes_2[2].set_title('markersize=12')
9  axes_2[2].plot(x_1,y_1,'x',markersize=12)
10 axes_2[3].set_title("markersize=16")
11 axes_2[3].plot(x_1,y_1,'x',markersize=16)
```

Listing 6: Anpassung der Markergröße

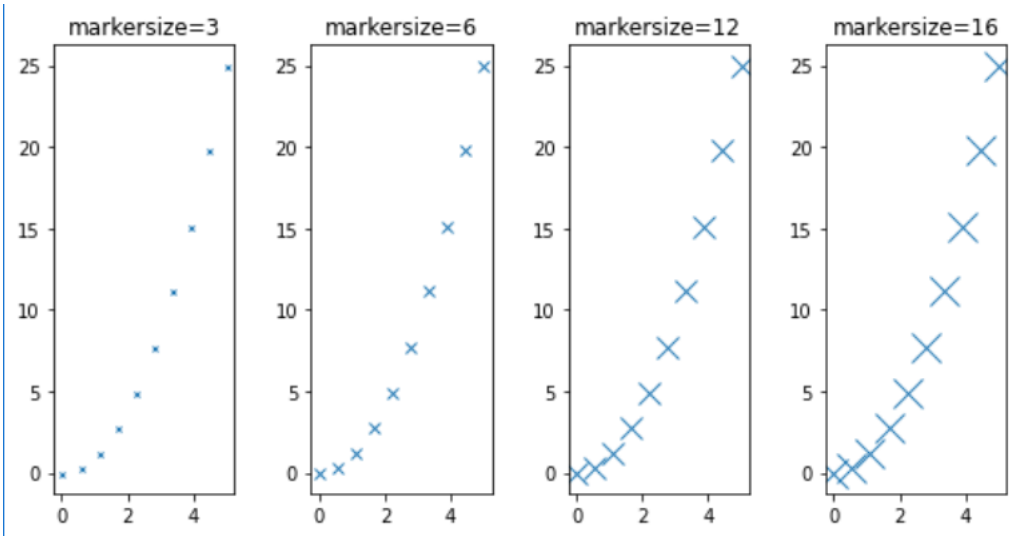

Figure 7: Anpassung der Marker größe

1.8 Farben

Zeichen	Farbe
b	Blau
g	Grün
r	Rot
c	Cyan
m	Magenta
y	Gelb
k	Schwarz
w	Weiß

2 Anpassung und Gestaltung

2.1 Raster einfügen

Mit der Funktion **grid(True, color=,dashes=())** wird in dem Diagramm ein Raster eingefügt.
Mit dashes wird das Muster des Rasters eingestellt. (Anzahl Linien, Anzahl Pausen, Anzahl Linien, Anzahl Pausen ....).

```
1 fig_1 = plt.figure(figsize=(5,4))
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Raster')
4 axes_1.grid(True, color='0.6', dashes=(5,2,1,2))
5 axes_1.plot(x_1,y_1)
```

Listing 7: Raster einfügen

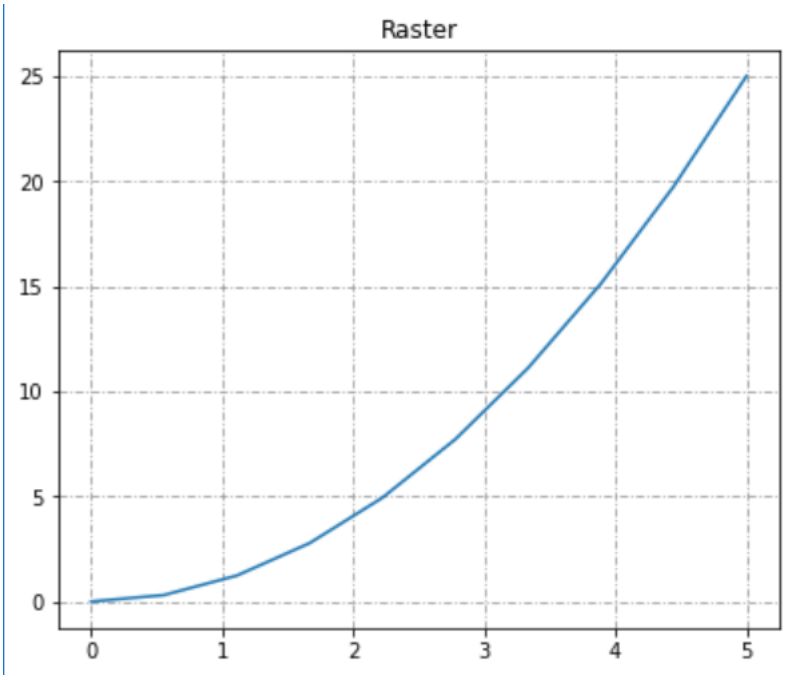

Figure 8: Raster einfügen

2.2 Anpassung der Hintergrundfarbe

Mit der Funktion **facecolor()** kann man beliebig die Hintergrundfarbe, des Plots anpassen.

```
1 fig_1 = plt.figure(figsize=(5,4))
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Hintergrundfarbe')
4 axes_1.set_facecolor('#FAEBD7')
5 axes_1.grid(True)
6 axes_1.plot(x_1,y_1)
```

Listing 8: Anpassung der Hintergrundfarbe

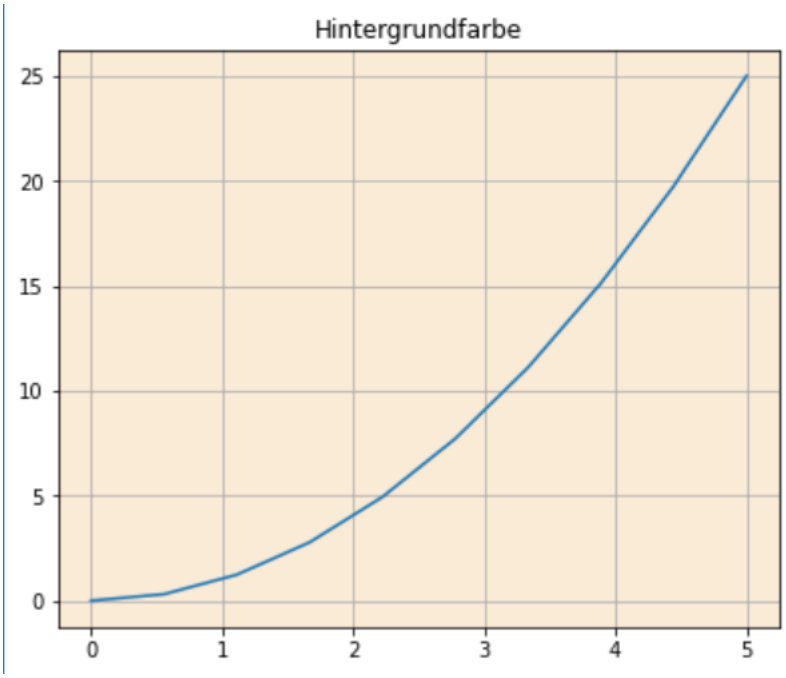

Figure 9: Anpassung der Hintergrundfarbe

2.3 Auf einen Punkt zeigen

Mit der Funktion **plt.annotate('Text', xy=(x,y),xytext=(x,y),arrowprops=dict(facecolor='black', shrink=0.05))** kann man mit einem Pfeil auf einen bestimmten Punkt verweisen. Mit den optimalen Parametern facecolor kann man die Farbe des Pfeils ändern und mit shrink bestimmen, um wie viel Prozent der Pfeil kleiner sein soll, damit er nicht direkt auf deinen Punkt zeigt.

```
1 fig_1 = plt.figure(figsize=(5,4),dpi=100)
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Auf einen Punkt verweisen')
4 axes_1.annotate('Ein Punkt',xy=(2,4),xytext=(3,3.5),arrowprops=dict(facecolor='b
5 axes_1.plot(x_1,y_1)
```

Listing 9: Auf einen Punkt zeigen

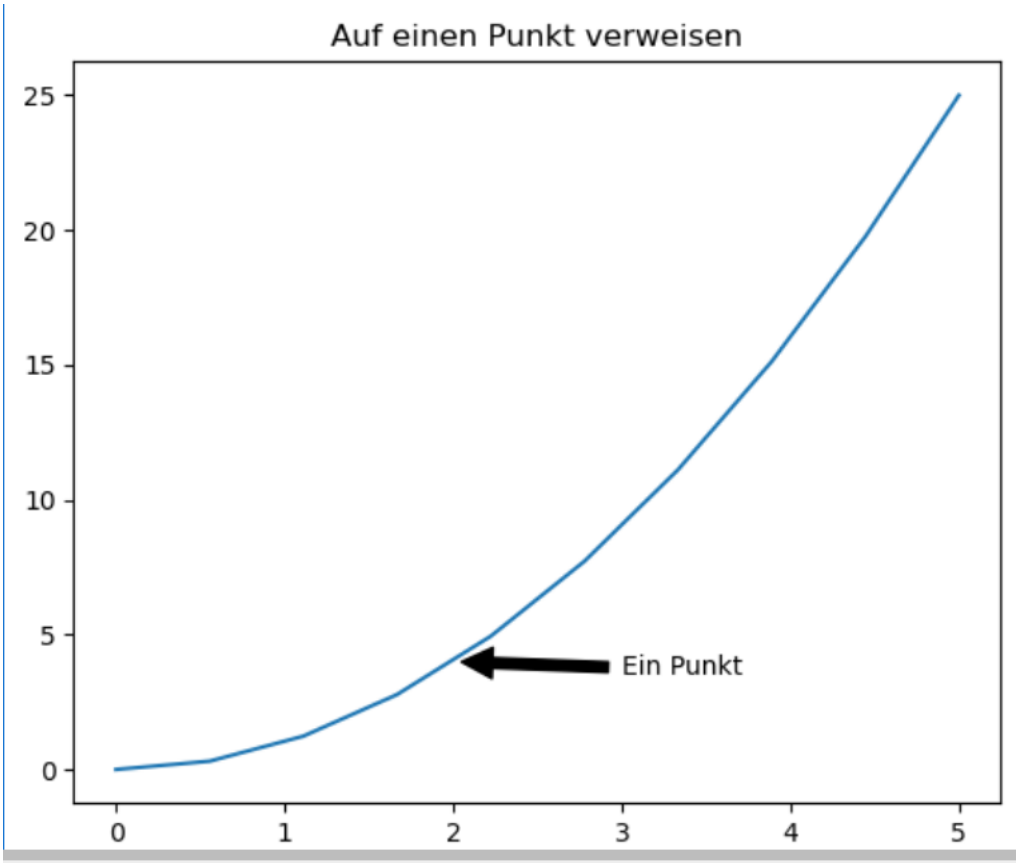

Figure 10: Auf einen Punkt verweisen

2.4 Einfügen von Pfeilen

Mit **plt.arrow(x=, y=, dx=, dy=)** kann ein Pfeil ins Koordinatensystem geplottet werden. Dabei steht x und y für die Koordinaten, an denen der Pfeil startet und dx und dy für die Länge des Pfeiles.

```
1 fig_1 = plt.figure(figsize=(5,4),dpi=100)
2 axes_1 = fig_1.add_axes([0.1,0.1,0.9,0.9])
3 axes_1.set_title('Pfeil im Plot')
4 plt.arrow(x=4, y=6, dx=2, dy=2, width=0.04)
5 plt.arrow(x=6, y=7.8, dx=-1.5, dy=-1.5, width=0.04)
```

Listing 10: Einfügen von Pfeilen

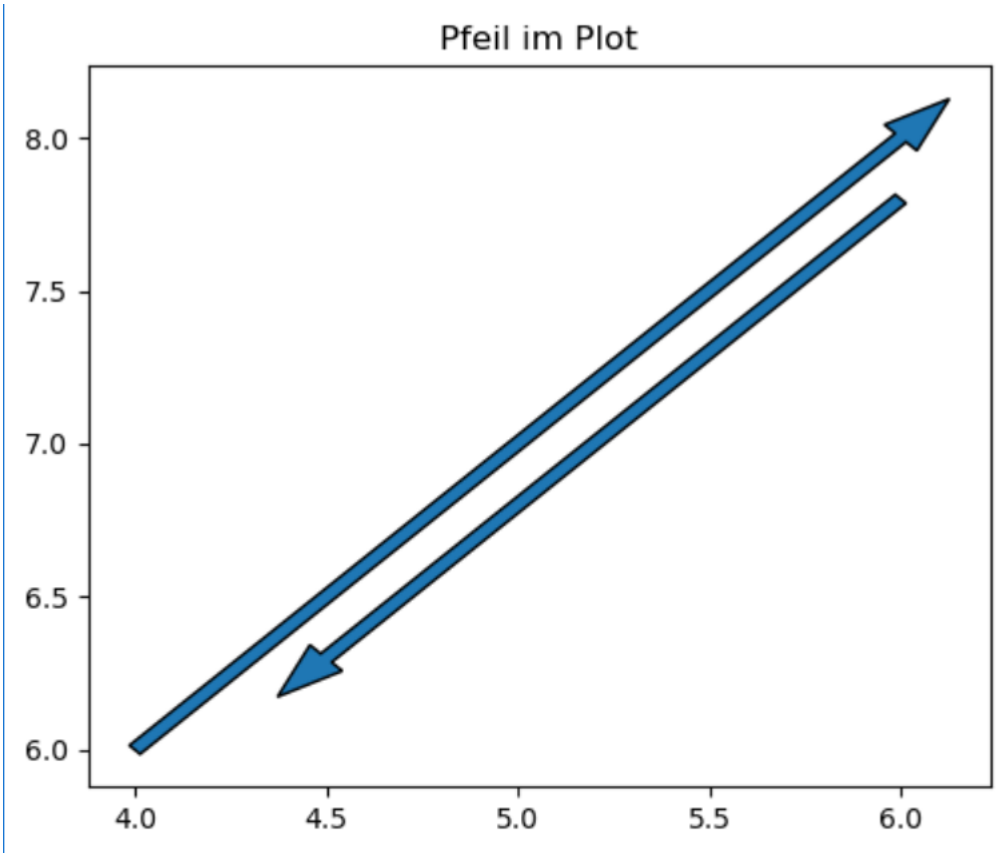

Figure 11: Einfügen von Pfeilen

2.5 Texteinbindung

Mit der Funktion **plt.text(x, y, Text)** kann man in einen Plot einen eigenen Text einfügen. Dabei definiert x und y die Position im Plot und Text, was an der gewählten Stelle stehen soll.

```
1  x = [3, 6, 8, 12, 14]
2  y = [4, 9, 14, 12, 9]
3
4  plt.title('Text im Plot')
5  plt.plot(x,y,'bo')
6  plt.text(6, 9.5, 'A piece of text')
```

Listing 11: Texteinbindung

Figure 12: Texteinbindung

2.6 Anpassung des Textstils

font = {'family': 'serif',
'color': 'red',
'weight': 'bold',
'size': 20
}
family - Gibt die Schriftart an.
color - Die Schriftfarbe
weight - Schrift Eigenschaft (normal, bold, light)
size - Schriftgröße in px

```
1  x = [3, 6, 8, 12, 14]
2  y = [4, 9, 14, 12, 9]
3
4  font = {'family': 'serif',
5          'color':  'red',
6          'weight': 'bold',
7          'size': 20
8          }
9
10 plt.title('Text im Plot')
11 plt.plot(x,y,'bo')
12 plt.text(6, 9.5, 'A piece of text', fontdict=font)
```

Listing 12: Anpassung des Textstils

Figure 13: Anpassung des Textstils

2.7 Rahmung von Text mit einer Box

box = {'facecolor': 'none',
'edgecolor': 'green',
'boxstyle': 'round'
}
facecolor - Hintergrundfarbe
edgecolor - Farbe des Rahmens

Boxstyle
circle
darrow
larrow
rarrow
round
round4
roundtooth
sawtooth
square

```
1 box_r = {'facecolor': 'none',
2          'edgecolor': 'green',
3          'boxstyle': 'round'
4         }
5 plt.title('Text-Boxen')
6 plt.plot(6, 9.5, 'bo', ms=2)
7
8 plt.text(6, 9.5, 'round', bbox=box_r)
```

Listing 13: Rahmung von Text mit einer Box

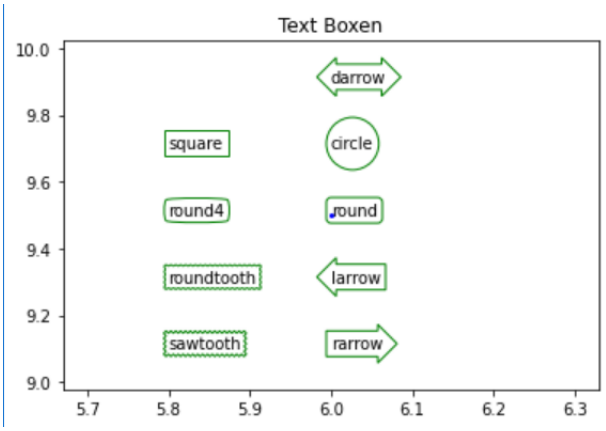

Figure 14: Box

2.8 Title

plt.title('Title') setzt den Title für den erstellten Diagrammplot fest.

```
1 x_1 = np.linspace(0,5,10)
2 y_1 = x_1**2
3 plt.title('Days Squared Chart') # Ein Diagramm mit Title
4 # versehen.
5 plt.plot(x_1,y_1)
```

Listing 14: Title

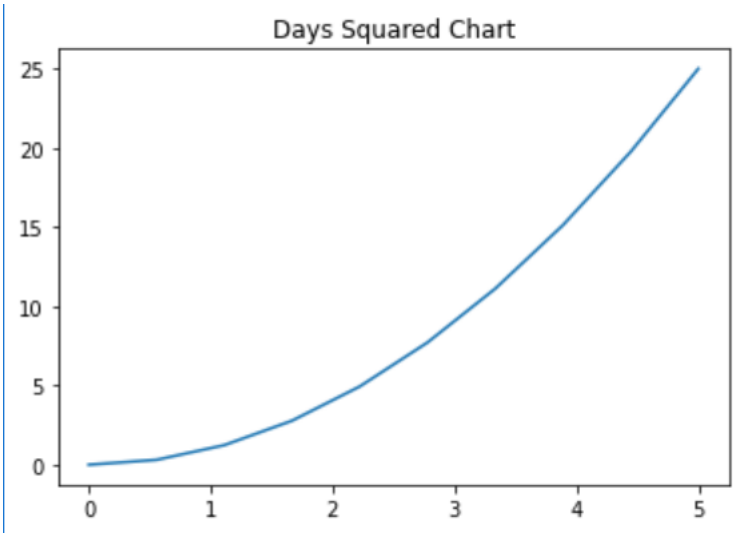

Figure 15: Tilte

2.9 TeX-Markup verwenden

In dem man innerhalb eines Rohe-Strings zwei $$ einfügt, kann man TeX Markup benutzten, innerhalb seiner Plots.

```
fig_1 = plt.figure(figsize=(5,4))
axes_1 = fig_1.add_axes([0,0,1,1])
axes_1.set_title('TeX-Markup')
axes_1.text(0, 23,
            r'$\delta_i\gamma^{ij}\sum_{i=0}^\infty x_1\frac{3}{4}$')
axes_1.text(0,18,
            r'$\frac{8-\frac{x}{5}}{8}\sqrt{\pi}\sin(\pi)\sqrt[3]{8}$')

axes_1.plot(x_1,y_1)
```

Listing 15: TeX-Markup verwenden

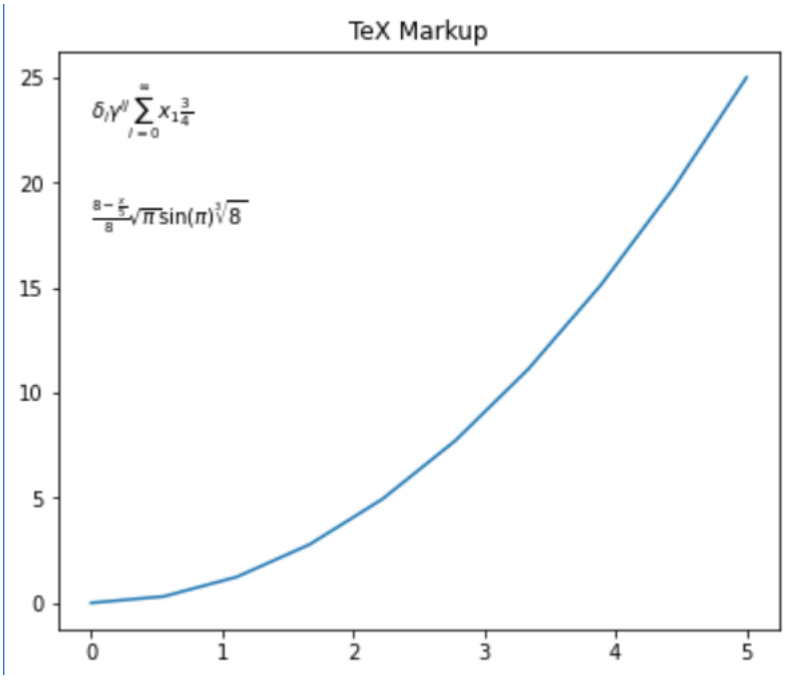

Figure 16: TeX-Markup verwenden

3 Achsen und Koordinatensysteme

3.1 Achsenunsichtbar machen

Mit **axes.get_xaxis().set_visible(False)** und **axes.get_yaxis().set_visible(False)**, kann jeweils die x-Achse, sowie auch die y-Achse unsichtbar gemacht werden.

```
1  x_1 = np.linspace(0,5,10)
2  y_1 = x_1**2
3  fig_1 = plt.figure(figsize=(5,4))
4  axes_1 = fig_1.add_axes([0,0,1,1])
5  axes_1.set_title('Achsen entfernen')
6  axes_1.get_xaxis().set_visible(False)
7  axes_1.get_yaxis().set_visible(False)
8  axes_1.plot(x_1,y_1)
```

Listing 16: Achsenunsichtbar machen

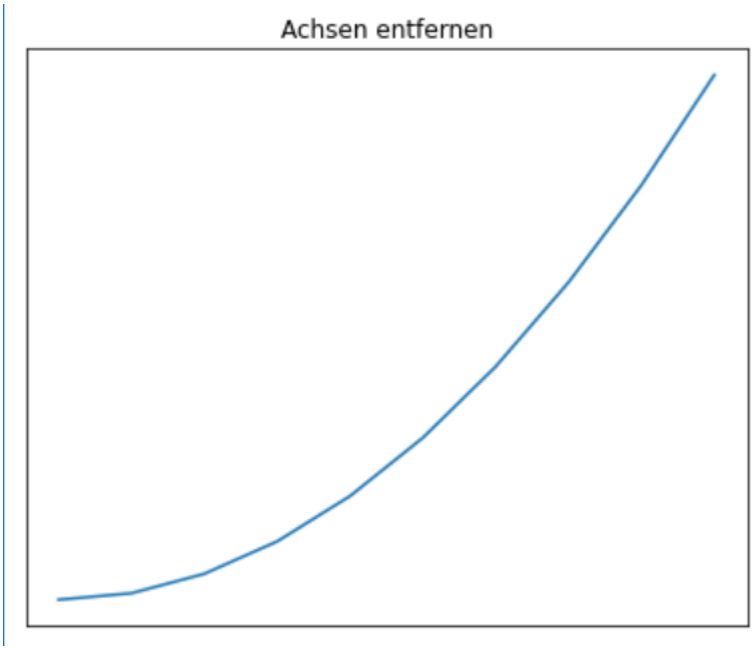

Figure 17: Achsen unsichtbar machen

3.2 Kartetisches Koordinatensystem

Mit **spines[”].set_position(”)** kann man die Position der Achsen verschieben. Es gibt 4 Achsen im Koordinatensystem: Top, bottom, right und left.

Mit **spines[”].set_color(”)** kann man die Farbe der jeweiligen Achse anpassen und mit 'none' unsichtbar machen.

ax.plot(0,1, ”¿k”, transform=axes_1.get_yaxis_transform(), clip_on=False), erstellt die Pfeile an den Koordinatenachsen enden. Die 0,1 gibt dabei die x bzw. y Koordinate an, dabei ist 0 ganz Links vom Diagramm und 1 ganz Rechts von Diagramm. Die Funktion
transform=ax.get_yaxis_transform() macht, dass die y-Koordinate exakt an die y-Achse bezogen ist (für x genau so). clip_on gibt an, ob der Pfeil abgeschnitten werden soll, falls er außerhalb des Diagramms ist.

```
1  from mpl_toolkits.axisartist.axislines import AxesZero
2
3  fig_1 = plt.figure(figsize=(5,4))
4  axes_1 = fig_1.add_axes([0,0,1,1])
5  axes_1.set_title('Kartetisches Koordinatensystem')
6
7  axes_1.spines['bottom'].set_position('zero')
8  axes_1.spines['left'].set_position('zero')
9  axes_1.spines['top'].set_color('none')
10 axes_1.spines['right'].set_color('none')
11
12 axes_1.plot(1, 0, ">k", transform=axes_1.get_yaxis_transform(), clip_on=False)
13 axes_1.plot(0, 1, "^k", transform=axes_1.get_xaxis_transform(), clip_on=False)
14
15 axes_1.plot(x_1,y_1)
```

Listing 17: Kartetisches Koordinatensystem

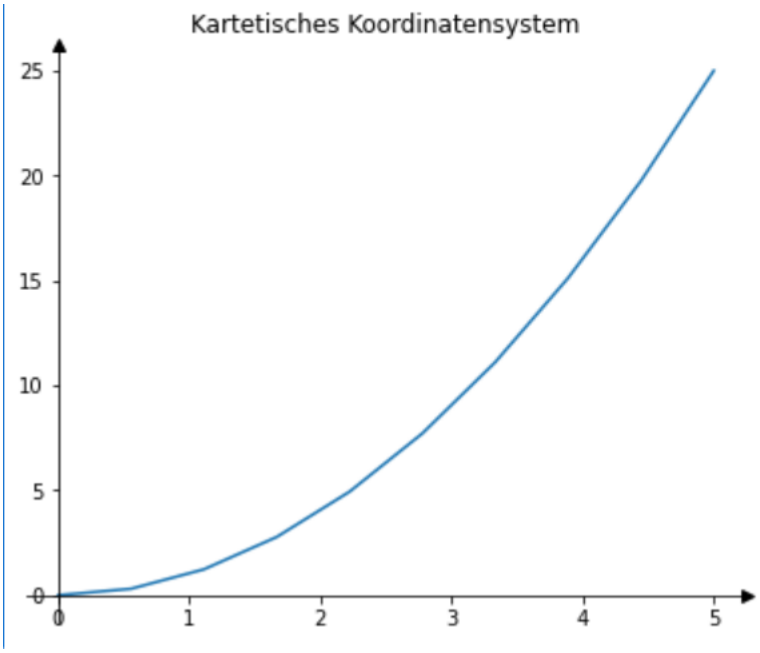

Figure 18: Kartetisches Koordinatensystem

3.3 Verwendung einer logarithmischen Skala

Mit den Funktionen **xscale('log')** oder **yscale('log')** kann man die jeweiligen Achsen in einen logarithmischen Maßtab skalieren.

```
1 fig_1 = plt.figure(figsize=(5,4))
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Log')
4 axes_1.set_yscale('log')
5 axes_1.plot(x_1,y_1)
```

Listing 18: Logarithmische Skalar

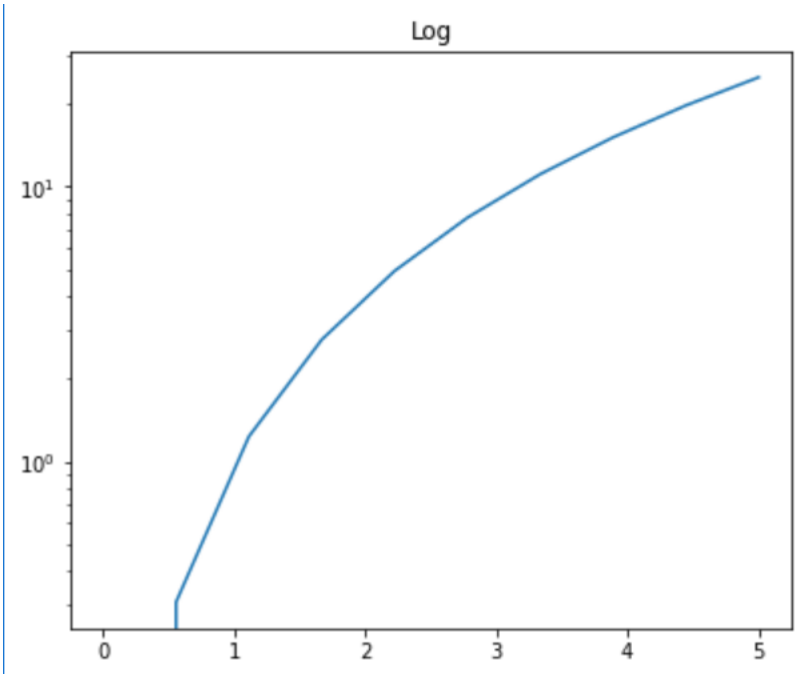

Figure 19: Logarithmische Skala

3.4 Achsenbegrenzung

Mit den Funktionen **xlin([x,y])** und **ylim([x,y])** kann man die Achsen beliebig begrenzen.

```
1 fig_1 = plt.figure(figsize=(5,4))
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Achsen begrenzen')
4 axes_1.set_xlim([0,3])
5 axes_1.set_ylim([0,8])
6 axes_1.plot(x_1,y_1)
```

Listing 19: Achsenbegrenzung

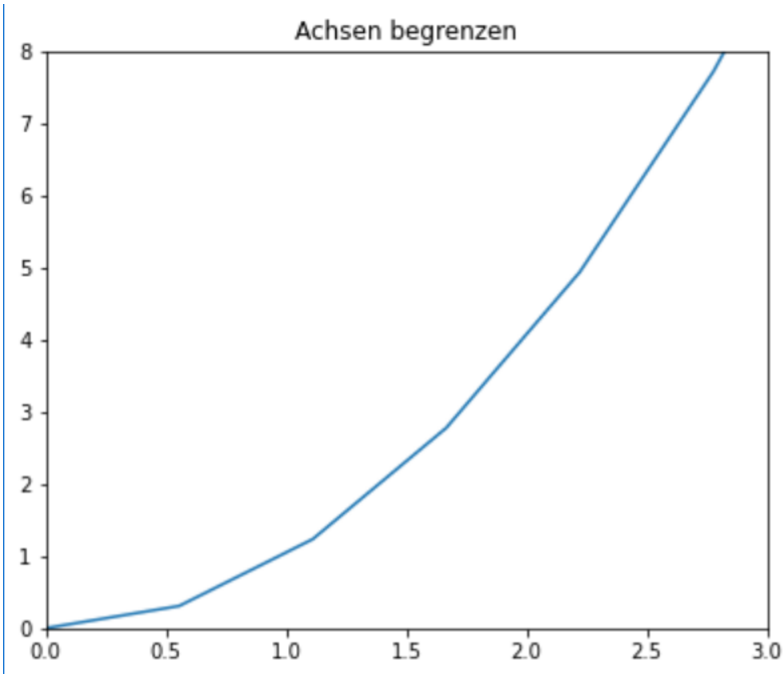

Figure 20: Achsenbegrenzung

4 Visualisierung von Daten

4.1 Legende

Mit **legend(loc=)** kann eine Legende zum Plot hinzugefügt werden.
upper right
upper left
lower left
lower right
oder eine tulpe aus (x, y) Koordinaten.

Um die Legende exakt zu positionieren, unter anderem auch außerhalb des Plots, kann folgender Befehl benutzt werden:
plt.legend(loc='right', bbox_to_anchor=(1, 0, 0.5, 1)) dabei steht **bbox_to_anchor=(x, y, width, height)** Alternative kann einfach nur **axes.legend(loc=)** benutzt werden, um eine Legende innerhalb des Plots zu erstellen.

```
1 fig_1 = plt.figure(figsize=(5,4))
2 axes_1 = fig_1.add_axes([0,0,1,1])
3 axes_1.set_title('Achsen begrenzen')
4 axes_1.set_xlim([0,3])
5 axes_1.set_ylim([0,8])
6 axes_1.plot(x_1,y_1, label='x^2 Werte')
7 axes_1.legend(loc='right', bbox_to_anchor=(0.8,0,0.5,1.7))
```

Listing 20: Legende

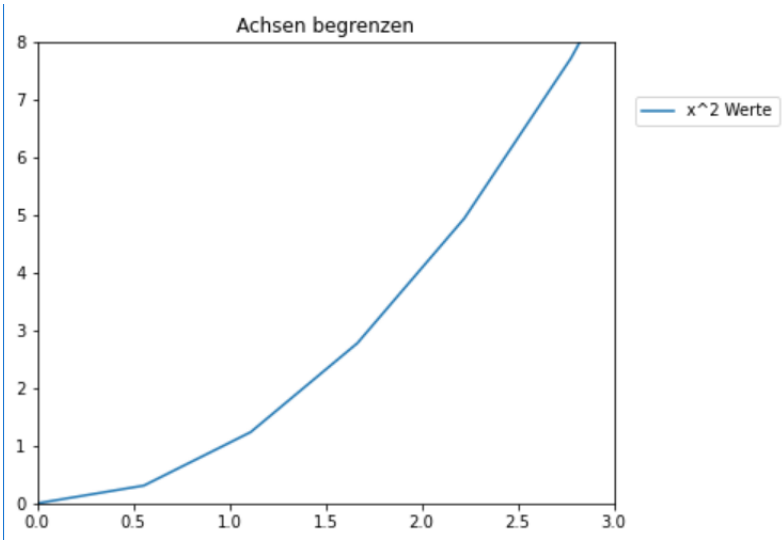

Figure 21: Legende

4.2 Mehrere Plots

plt.subplot(Spalte, Zeilen, Position) erzeugt eine Rasteranordnung von Plots, wobei die Reihenfolge Spalte, Zeile und dann die ausgewählte Position ist.

```
1  plt.subplot(1,2,1) # Erstellt ein Sub-Plot (Spalten, Zeilen, Adresse)
2  plt.plot(x_1,y_1,'r')
3  plt.subplot(1,2,2)
4  plt.plot(x_1,y_1,'b')
```

Listing 21: Mehrere Plots

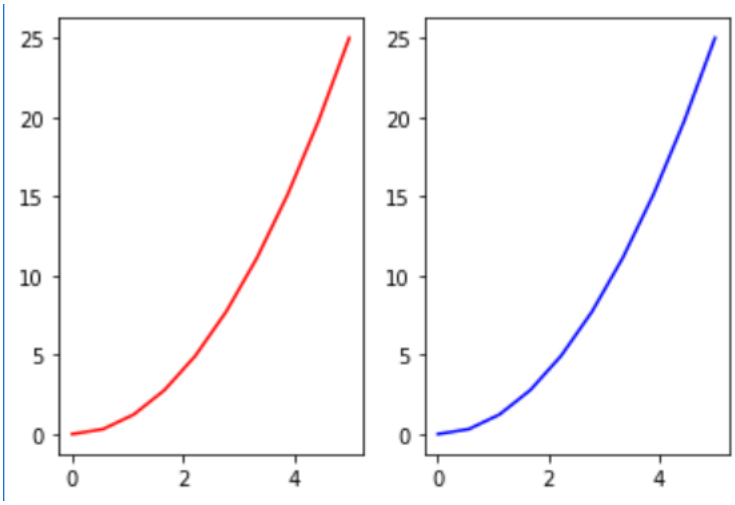

Figure 22: Mehrere Plots

4.3 Figures

Der Begriff "Figures" bezieht sich auf die oberste Ebene von Diagrammen, die als Leinwand für Plots dienen, auf denen mehrere Subplots oder Grafikelemente angeordnet werden können. Diese "Figures" sind wichtig, da sie die grundlegende Zeichenfläche für Diagramme bereitstellen, auf der Plots, Achsen, Beschriftungen und andere Elemente angeordnet werden können. Sie ermöglichen die Strukturierung und Organisation von Grafiken, was besonders nützlich ist, wenn mehrere Plots oder Subplots in einer einzigen Ausgabe benötigt werden.

plt.figure(figsize=(x,y),dip=100) erstellt eine neue Matplotlib-Figur mit einer Größe von 5X4 und einer Auflösung von 100 DPI (Dots per inch)

.legend(loc=) wird verwendet, um eine Legende zu einem Diagramm hinzuzufügen. Dabei gibt es mehrere Möglichkeiten, die Position zu wählen.

loc=	Position
0	System wählt automatisch
1	Oben Rechts
2	Oben Links
3	Unten Links
4	Unten Rechts
(x,y)	Frei im Diagramm wählbar

```
1  fig_1 = plt.figure(figsize=(5,4),dpi=100) # Erstellt eine Figure.
2  Figsize gibt die gr    e des Diagramms an
3  # dpi gibt den zoom an (100%)
4  axes_1 = fig_1.add_axes([0.1,0.1,0.9,0.9]) # fuegt die Achsen hinzu
5  (left, bottom, width, height)
6  axes_1.set_xlabel('Days') # X-Achsen beschrieftung
7  axes_1.set_ylabel('Days Squared') # Y-Achsen beschrieftung
8  axes_1.set_title('Days Squared Chart') # Title des Diagramms
9  axes_1.plot(x_1,y_1,label='x/x^2') # Plot mit label
10 axes_1.plot(y_1,x_1,label='x^2/x')
11 axes_1.legend(loc=0)
```

Listing 22: Figures

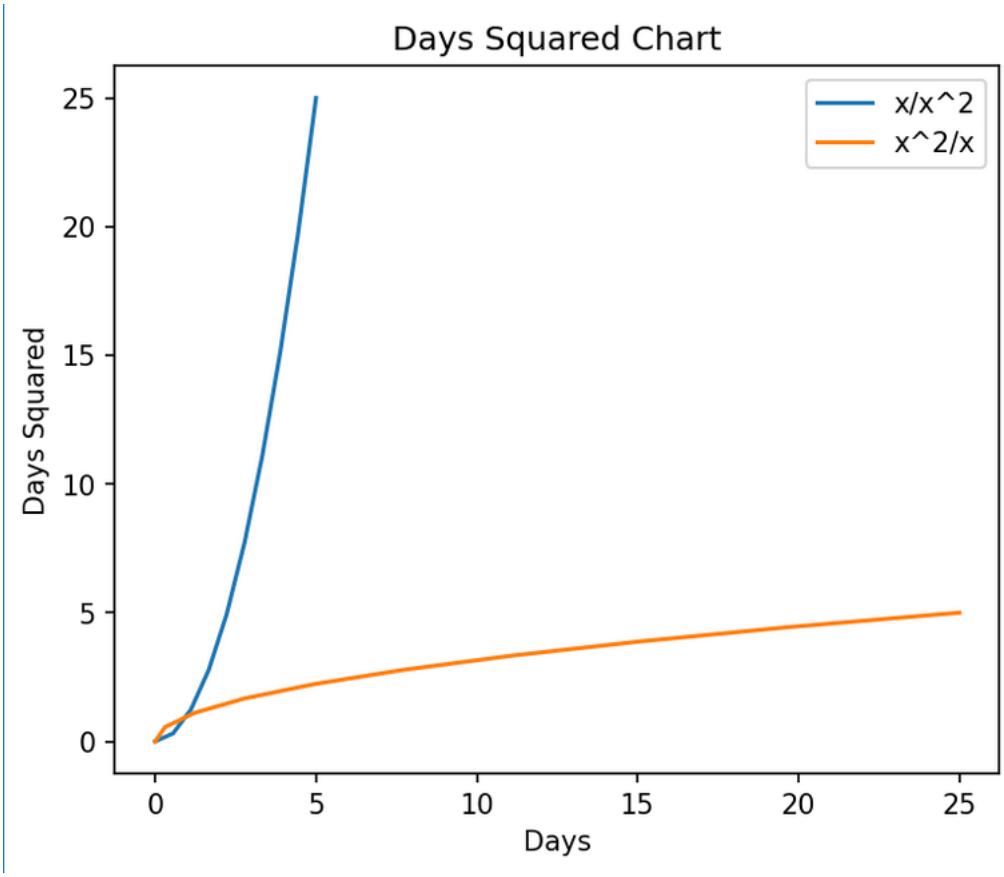

Figure 23: Figures

4.4 Flächen einfärben

Mittels Matplotlib ist es auch möglich, Flächen in einem Plot einzufärben und damit visuell hervorzuheben. Dafür wird die Funktion **ax.fill_between(x,y,where=,facecolor=,alpha=,legend=)** verwendet. Dabei kann mit where=, eine Bedingung eingegeben werden für die Einfärbung.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0,5,100)
5  y1 = (x-3)**2
6  y2 = -(x-2)**2+8
7
8  fig = plt.figure()
9  ax = fig.add_axes([0.1,0.1,0.9,0.9])
10 ax.plot(x,y1,x,y2, color='black')
11 ax.grid(True)
12 ax.fill_between(x,y1,y2,where=y2>=y1, facecolor='b',alpha=0.2)
```

Listing 23: Flächen einfärben 1

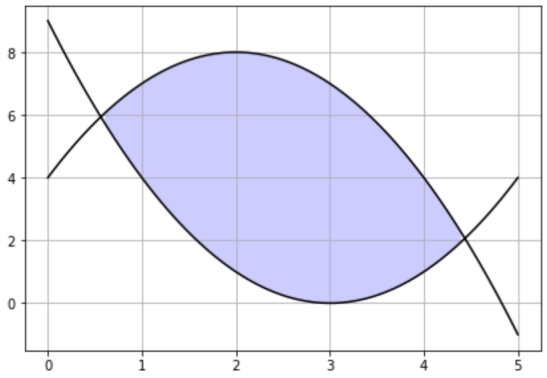

Figure 24: Flächen einfärben 1

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  t = np.linspace(0,40,500)
5  y = 20 * np.sin(2*np.pi*50*t*np.float_power(10,-3))
6
7  fig = plt.figure()
8  ax = fig.add_axes([0.1,0.1,0.9,0.9])
9  ax.plot(t,y,color='k')
10 ax.grid(True)
11 ax.fill_between(t,y,where=y>=0, facecolor='b', alpha=0.2,
12 label='Positiver Anteil')
13 ax.fill_between(t,y,where= y<=0, facecolor='g', alpha=0.2,
14 label='Negativer Anteil')
15 ax.legend(loc=0)
```

Listing 24: Flächen einfärben 2

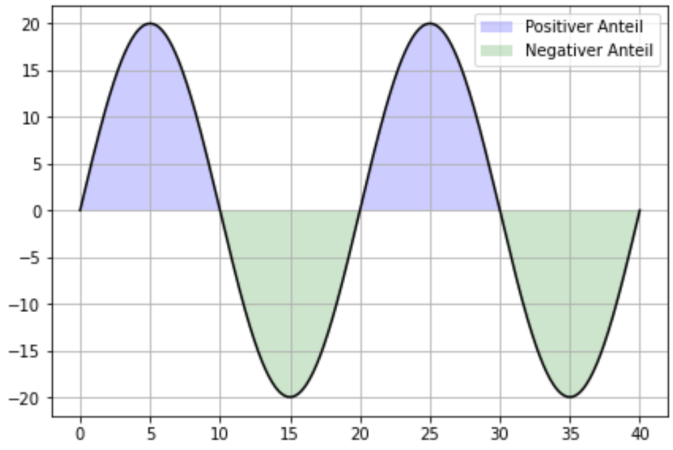

Figure 25: Flächen einfärben 2

4.5 Plot im Plot

```
1  fig_1 = plt.figure(figsize=(5,4),dpi=100)
2  axes_1 = fig_1.add_axes([0.1,0.1,0.9,0.9])
3  axes_1.set_xlabel('Days')
4  axes_1.set_ylabel('Days Squared')
5  axes_1.set_title('Days Squared Chart')
6  axes_1.plot(x_1,y_1,label='x/x^2')
7  axes_1.plot(y_1,x_1,label='x^2/x')
8  axes_1.legend(loc=1)
9
10 # Erstellt einen Plot, innerhalb eines Plots
11 axes_2 = fig_1.add_axes([0.45, 0.45, 0.4, 0.35])
12 axes_2.set_xlabel('Days')
13 axes_2.set_ylabel('Days Squared')
14 axes_2.set_title('Days Squared Chart')
15 axes_2.plot(x_1,y_1,'r')
16
17 axes_2.text(0,35,'Message') # fügt einen Text hinzu
```

Listing 25: Plot im Plot

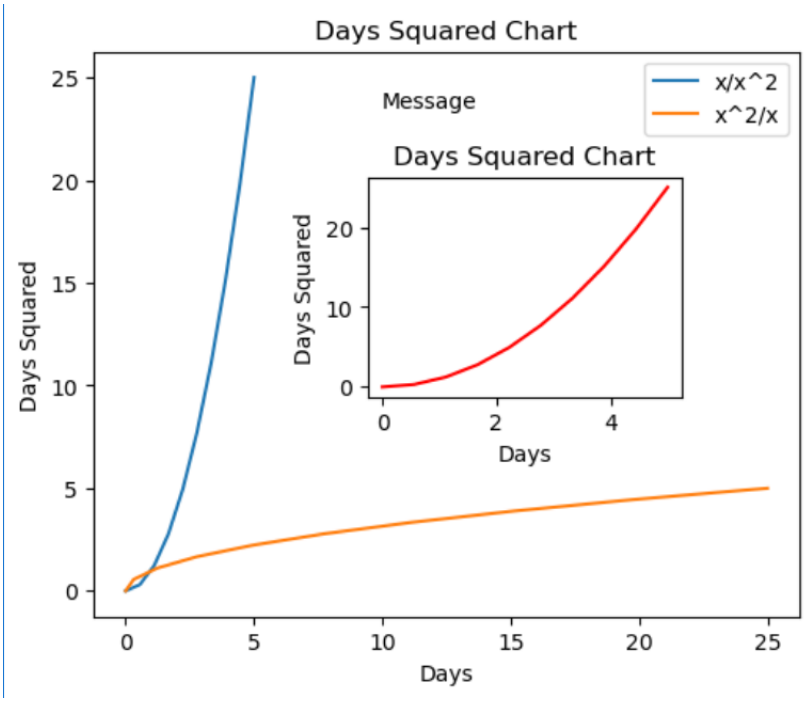

Figure 26: Plot im Plot

4.6 Sub-Plots

```
fig_2, axes_2 = plt.subplots(figsize=(8,4), nrows=1, ncols=3)
plt.tight_layout()
axes_2[1].set_title('Plot 2')
axes_2[1].set_xlabel('x')
axes_2[1].set_ylabel('x Squared')
axes_2[1].plot(x_1,y_1)
```

Listing 26: Sub-Plots

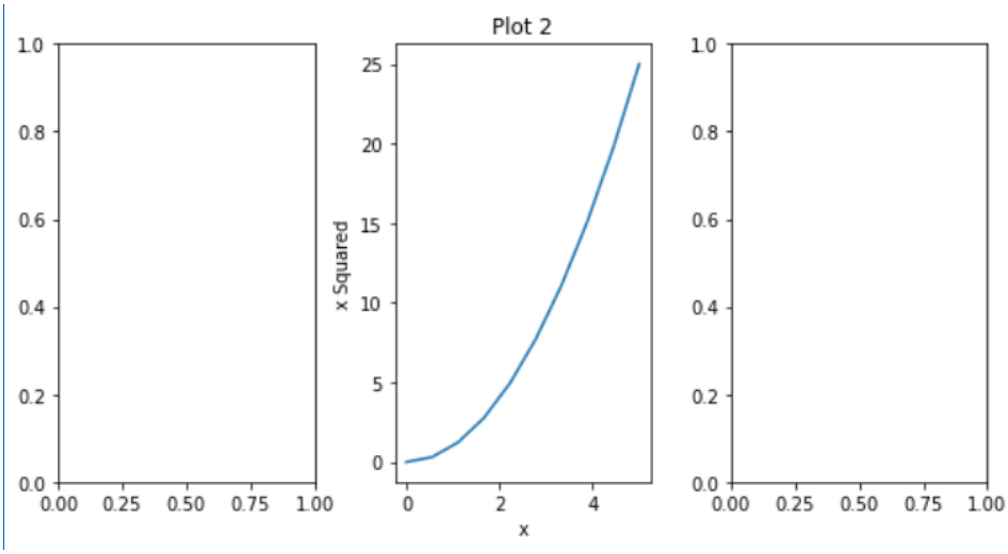

Figure 27: Sub-Plots

4.7 Ploten mit fehlenden Werten

Mit der NumPy-Funktion **np.nan** kann man fehlende Werte in seinen Datensatz hinzufügen; diese werden Lücken, kennzeichnen sich dadurch, dass sie nicht verbunden werden.

```
1  import numpy as np
2  x_2 = np.arange(-1.5, 1.75, 0.25)
3  y_2 = [0, 1, 1.5, 2, np.nan, 2.3, 2.5, np.nan,
4  2.3, 2, np.nan, 1.5, 1.0]
5  fig = plt.figure(figsize=(5,4))
6  axes2 = fig.add_axes([0,0,1,1])
7  axes2.grid(True)
8  axes2.plot(x_2, y_2)
```

Listing 27: Ploten mit fehlenden Werten

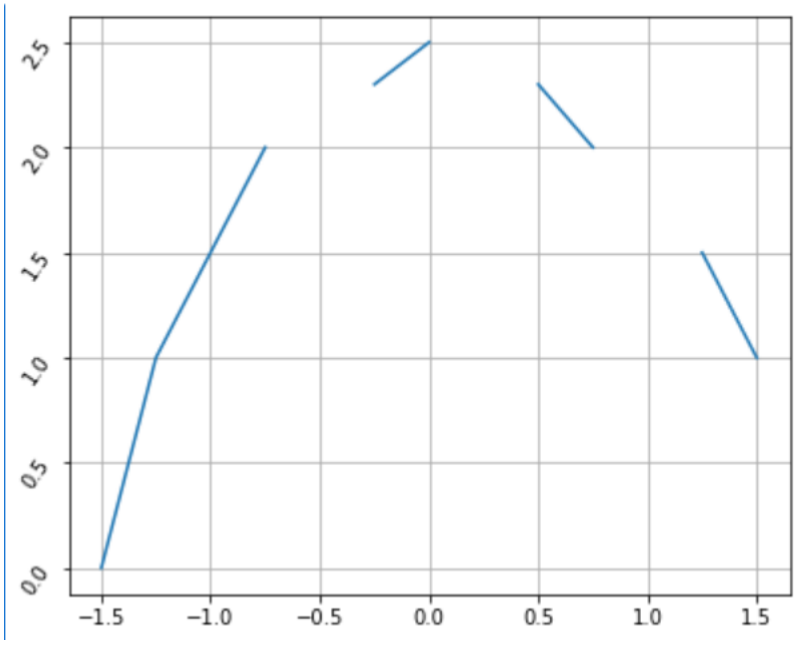

Figure 28: Ploten mit fehlenden Werten

5 Fortgeschrittene Visualisierungstechniken

5.1 Plots Interaktiv gestalten

In Matplotlib ist es möglich, direkt Plots interaktiv zu gestalten, allerdings ist diese Funktion eher begrenzt nutzbar und für "aufwändigere"-Programme sollte eher auf Tkinter oder PyQt5 zurückgegriffen werden.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.widgets import Slider, Button
4
5  t = np.linspace(0,1,200)
6  a0 = 5
7  f0 = 5
8  s = a0*np.sin(2*np.pi*f0*t)
9
10 fig = plt.figure(figsize=(10,6))
11 ax = fig.add_axes([0.25,0.30,0.9,0.9])
12 ax.axis([0,1,-10,10])
13 ax.grid(True)
14 kurve, = ax.plot(t,s,lw=2,color='k')
15
16 #Objekt f r die Plazierung der Steuerelemente
17 #Linker Rand, unterer Rand, l nge, h he
18 xyAmp = fig.add_axes([0.35,0.15,0.7,0.05])
19 xyFre = fig.add_axes([0.35,0.1,0.7,0.05])
20 xyRes = fig.add_axes([0.8,0.025,0.1,0.05])
21
22 #Objekte f r Steuerelemente erzeugen
23 #(Koordinaten, Text, Kleinster-Wert, Gr   ter-Wert, Startwert, Step-Gr    e)
24 sldAmp = Slider(xyAmp, 'Amplitude',1,10, valinit=a0, valstep=0.1)
25 sldFre = Slider(xyFre, 'Frequenz',1,10, valinit=f0, valstep=0.1)
26 bRes = Button(xyRes, 'Reset')
27
28 #Funktion um Variablen und Funktion updaten
29 def update(val):
30       A = sldAmp.val
31       F = sldFre.val
32       kurve.set_data(t,A*np.sin(2*np.pi*F*t))
33
34 #Funktion um den Plot zu Reseten
35 def reset(event):
36       sldAmp.reset()
37       sldFre.reset()
38
```

```
39 #Funktion der Steurelemente implementieren
40 sldAmp.on_changed(update)
41 sldFre.on_changed(update)
42 bRes.on_clicked(reset)
```

Listing 28: Plots Interaktiv gestalten

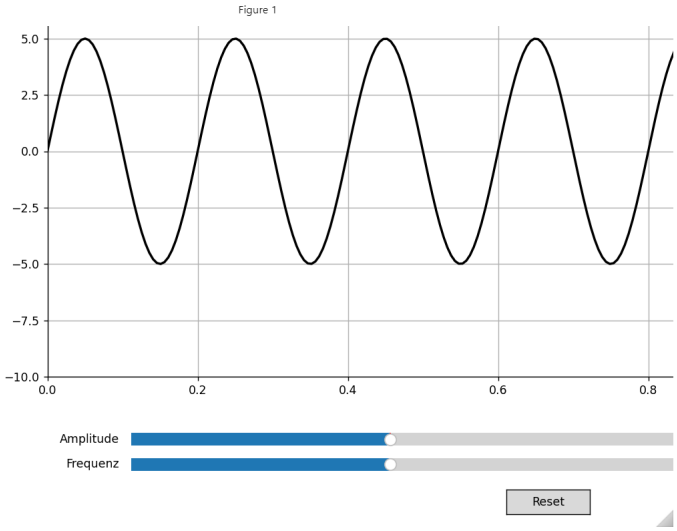

Figure 29: Plots Interaktiv

5.2 Histogramm

Histogramme zeigen die Verteilung von Daten in verschiedenen Kategorien oder Intervallen gut an. Es besteht aus Balken, die die Häufigkeit oder Anzahl der Datenpunkte in jedem Intervall darstellen, wodurch man schnell Einsicht in die Datenverteilung gewinnt.

Mit der Funktion **plt.hist(Array, bins=, density=, stacked=)** kann man ein Histogramm ploten.

Dabei bedeutet **bins=** die Anzahl, in wie viele Intervalle, der Datensatz eingeteilt werden soll.

Wenn, **density=** auf True gesetzt ist, gibt die y-Achse die Wahrscheinlichkeitsdichte an, in Bezug, wie hüfig der Datenpunkt aufgetreten ist. Falls density auf False gesetzt ist, repräsentieren die Balken die absolute Häufigkeit der Daten.

Ist **stacked=** True, werden mehrere Datensätze innerhalb eines Histogramms, übereinander gelegt; falls auf false, sind die verschiedenen Datensätze nebeneinander.

```
1 arr_1 = np.random.randint(1,7,5000)
2 arr_2 = np.random.randint(1,7,5000)
3 arr_3 = arr_1 + arr_2
4
5 fig_1 = plt.figure(figsize=(5,4))
6 axes_1 = fig_1.add_axes([0,0,1,1])
7 axes_1.hist(arr_3, bins=11, density=True, stacked=True)
```

Listing 29: Histogramm

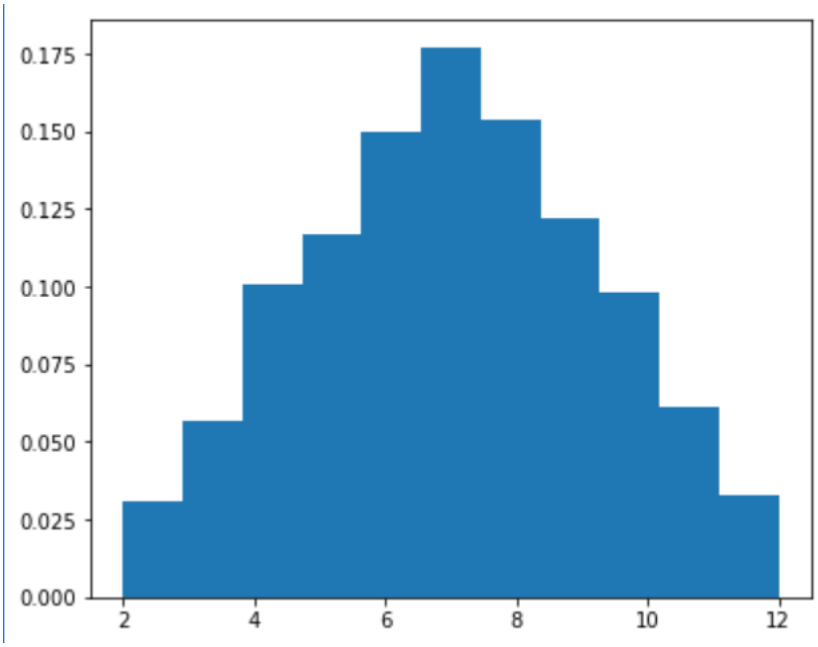

Figure 30: Histogramm

5.3 Bar-Charts

```
x = ['Nuclear', 'Hydro', 'Coal', 'Gas' , 'Solar', 'Wind',
'Other']
per_1 = [71, 10, 3, 7, 2, 4, 3]
variance = [8, 3, 1, 3, 1, 2, 1]

fig_1 = plt.figure(figsize=(5,4))
axes_1 = fig_1.add_axes([0,0,1,1])
axes_1.bar(x, per_1, color='purple', yerr=variance)
```

Listing 30: Bar-Charts

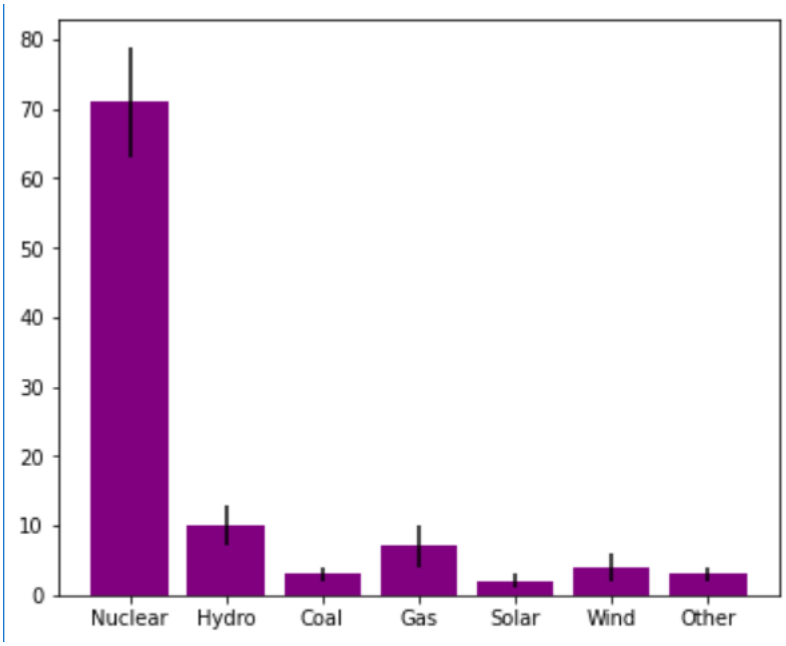

Figure 31: Bar-Charts

5.3.1 Balken Horizontal

```
x = ['Nuclear', 'Hydro', 'Coal', 'Gas' , 'Solar',
'Wind', 'Other']
per_1 = [71, 10, 3, 7, 2, 4, 3]
variance = [8, 3, 1, 3, 1, 2, 1]
a
fig_1 = plt.figure(figsize=(5,4))
axes_1 = fig_1.add_axes([0,0,1,1])
axes_1.barh(x, per_1, color='purple')
```

Listing 31: Bar-Charts

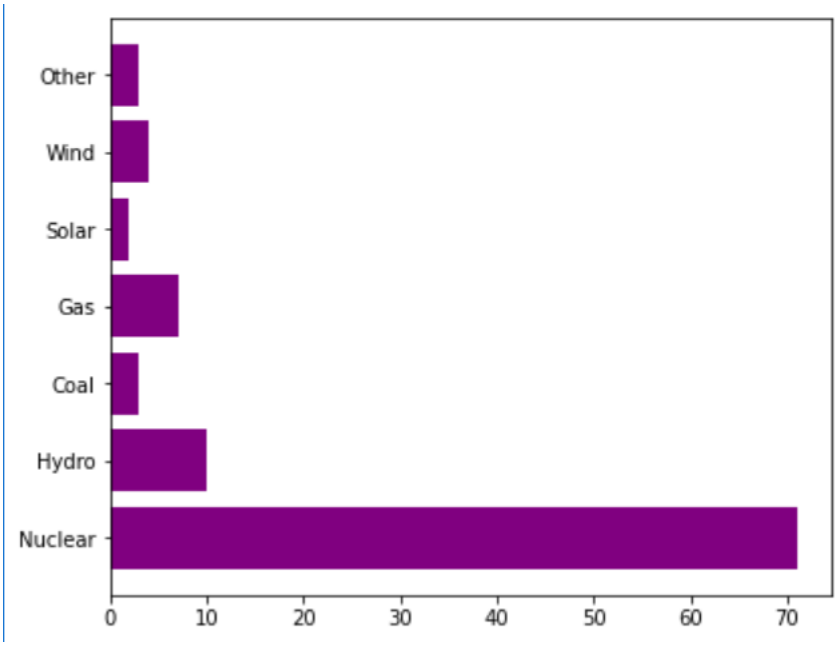

Figure 32: Balken Horizontal

5.3.2 Gruppierte Bar-Charts

axes_1.bar(x, y, width=, edgecolor=") erstellt ein Balkendiagramm. width gibt die Breite der jeweiligen Balken an. edgecolor, die Umrandungsfarbe der Balken.

```
m_eng = (76, 85, 86, 88, 93)
f_eng = (24, 15, 14, 12, 7)

fig_1 = plt.figure(figsize=(5,4))
axes_1 = fig_1.add_axes([0,0,1,1])

spc = np.arange(5)

axes_1.bar(spc, m_eng, width=0.45, label='Male',
edgecolor='k')
axes_1.bar(spc + 0.45, f_eng, width=0.45, label='Female',
edgecolor='k')
axes_1.set_xticks(spc + 0.45/2, ('Aero', 'Chem', 'Civil',
'Elec', 'Mech'))
```

Listing 32: Gruppierte Bar-Charts

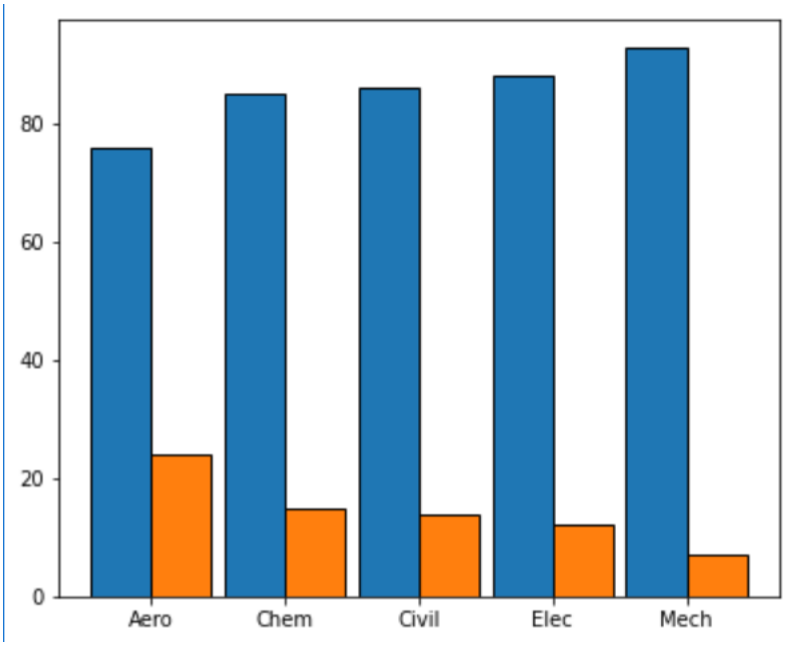

Figure 33: gruppierung von Bar-Charts

5.3.3 Gestapelte Bar-Charts

Indem **bottom=** benutzt wird, wird an jeder Balken Position auch die neuen Balken des zweiten Datensatzes eingefügt.

```
1  t_type = ['Kind', 'Elem', 'Sec', 'Spec']
2  m_teach = np.array([2, 20, 44, 14])
3  f_teach = np.array([98, 80, 56, 86])
4  ind = [x for x, _ in enumerate(t_type)]
5
6  fig_1 = plt.figure(figsize=(5,4))
7  axes_1 = fig_1.add_axes([0,0,1,1])
8  axes_1.bar(ind, m_teach, width=0.45, label='Male',
9  bottom=f_teach)
10 axes_1.bar(ind, f_teach, width=0.45, label='Female')
11 axes_1.legend(loc='lower right')
```

Listing 33: Gestapelte Bar-Charts

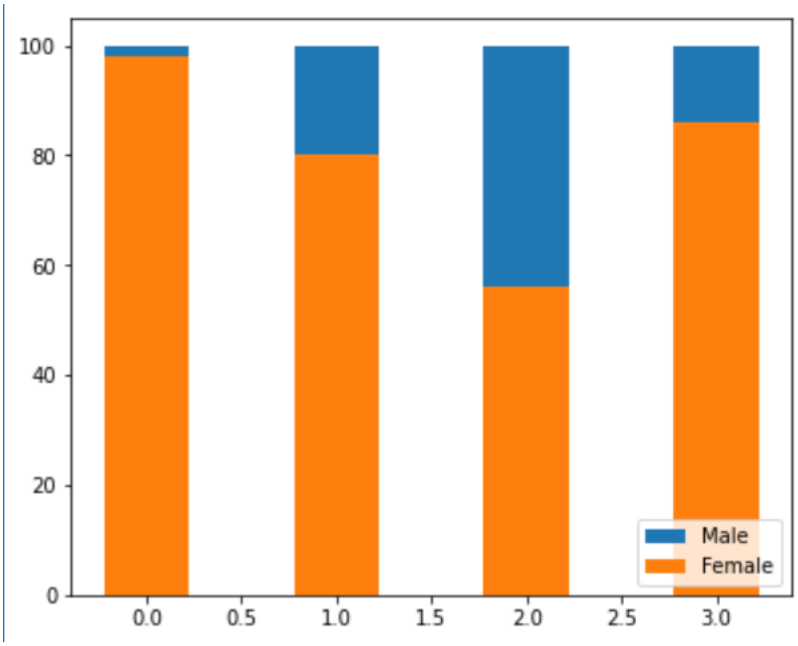

Figure 34: Gestapelte Bar-Charts

5.4 Kreis-Diagramm

Mit **plt.pi(Anzahl,explode=,labels=,colors=,autpct='%1.of% %startangle=,textprops=dict(color=""))** erstellt ein Kreis-Diagramm.
Anzahl gibt die Daten an, die das Diagramm repräsentiert. **explode** definiert, welches Stück herausgezogen wird und um wie viel Prozent, dazu muss eine Liste übergeben werden, mit so vielen Elementen wie Tortenstücke im Diagramm.
labels sind die Beschriftungen der Tortenstücke.
colors gibt die Farbe der Tortenstücke an.
autopct='%1.0f%
%' gibt die Prozentwerte ohne Dezimalstellen aus.
shadow=True fügt Schatten zu den Tortenstücken hinzu.
startangle= legt den Winkel fest, unter dem das Torten-Diagramm beginnt.
textprops=dict(color=' ') legt die Textfarbe für die Tortenstücke fest.

```
1  import random
2  fig_6 = plt.figure(figsize=(5,4))
3  axes_6 = fig_6.add_axes([0.1,0.1,0.9,0.9])
4
5  types = ['Water', 'Normal', 'Flying', 'Grass',
6           'Psychic', 'Bug', 'Fire', 'Posion',
7           'Ground', 'Rock', 'Fighting', 'Dark',
8           'Steel', 'Electric', 'Dragon', 'Fairy',
9           'Ghost', 'Ice']
10 poke_num = [133, 109, 101, 98, 85, 77, 68, 66, 65,
11             60, 57, 54, 53, 51, 50, 50, 46, 40]
12
13 colors = []
14 for i in range(18):
15     rgb = (random.uniform(0,.5),
16     random.uniform(0,.5),
17     random.uniform(0,.5))
18     colors.append(rgb)
19
20 explode = [0] * 18 # Liste mit 18 Nullen
21 explode[0] = 0.2
22
23 wedges, texts, autotexts = plt.pie(poke_num,
24 explode=explode, labels=types, colors=colors,
25     autopct='%1.0f%%', shadow=True,
26     startangle=140,
27     textprops=dict(color='w'))
28
29 plt.legend(wedges, types, loc='right',
```

```
30  bbox_to_anchor=(1, 0, 0.5, 1))
```

Listing 34: Kreis-Diagramm

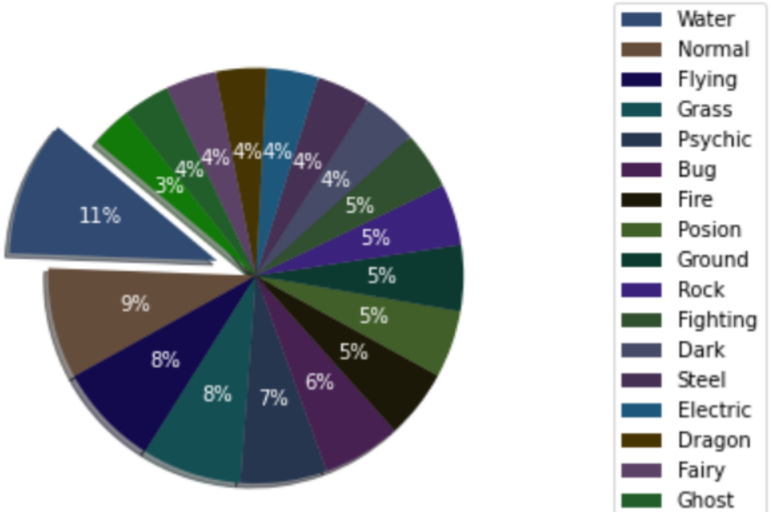

Figure 35: Kreis-Diagramm

5.5 Tabellarische Darstellung

Mit Matplotlib ist es auch möglich Tabellen zu erstellen, mit der Funktion **table_1 = ax_1.table(cellText=data, loc='center', cellColours=)**

CellText= Daten die innerhalb der Tabelle stehen sollen. **loc=** Ausrichtung der Zellen **CellColours=** Hintergrund der Tabellenfarbe, dabei muss eine Liste übergeben werden, in der jede Zelle Seperat die Farbe angepasst wird [rows * 'g'] * cols.

```
1  import matplotlib.pyplot as plt
2
3  data = [
4        ['Name', 'Alter', 'Stadt', 'Land', 'Beruf'],
5        ['Alice', 30, 'New-York', 'USA', 'Ingenieur'],
6        ['Bob', 25, 'London', 'UK', 'Student'],
7        ['', '', '', '', ''],   # Leere Zeile
8        ['Charlie', 35, 'Berlin', 'Deutschland', 'K nstler']
9  ]
10
11 fig = plt.figure(figsize=(8,8))
12 gs = fig.add_gridspec(7, 2)
13
14 ax_1 = fig.add_subplot(gs[0, 0:1])
15 ax_1.set_xticks([])
16 ax_1.set_yticks([])
17 ax_1.set_frame_on(False)
18 table_1 = ax_1.table(cellText=data, loc='center',
19 cellColours=[5*'g']*5)
20 table_1.set_fontsize(20)
21 table_1.scale(8,3)
```

Listing 35: Tabelle 1

Name	Alter	Stadt	Land	Beruf
Alice	30	New York	USA	Ingenieur
Bob	25	London	UK	Student
Charlie	35	Berlin	Deutschland	Künstler

Figure 36: Tabelle 1

Mit dem zusätzlichen Parameter **colLabels=** kann man eine Liste mit Zeilen Überschrieften übergeben und diese mit **colColours=** einfärben.

```
data2 = [
      ['Alice', 30, 'New York', 'USA', 'Ingenieur'],
      ['Bob', 25, 'London', 'UK', 'Student'],
      ['', '', '', '', ''],   # Leere Zeile
      ['Charlie', 35, 'Berlin', 'Deutschland', 'K nstler']
]

head = ['Name', 'Alter', 'Stadt', 'Land', 'Beruf']

ax_2 = fig.add_subplot(gs[2, 0:1])
ax_2.set_xticks([])
ax_2.set_yticks([])
ax_2.set_frame_on(False)
table_2 = ax_2.table(cellText=data2, colLabels=head,
cellColours=[5*'g']*4, colColours=['r']*5)
table_2.set_fontsize(20)
table_2.scale(8,3)
```

Listing 36: Tabelle 2

Name	Alter	Stadt	Land	Beruf
Alice	30	New York	USA	Ingenieur
Bob	25	London	UK	Student
Charlie	35	Berlin	Deutschland	Künstler

Figure 37: Tabelle 2

Dies geht auch mit **rowLabels=** einer zusätzlichen Spalte für Überschrieften erstellen und mit **rowColours=** die Farbe anpassen.

```
data3 = [
        ['Alice', 30, 'New York', 'USA', 'Ingenieur'],
        ['Bob', 25, 'London', 'UK', 'Student'],
        ['', '', '', '', ''],   # Leere Zeile
        ['Charlie', 35, 'Berlin', 'Deutschland', 'K nstler']
]

rowhead = ['Person 1.', 'Person 2.', 'Person 3.', 'Person 4.']

colhead = ['Name', 'Alter', 'Stadt', 'Land', 'Beruf']

ax_3 = fig.add_subplot(gs[6, 0:1])
ax_3.set_xticks([])
ax_3.set_yticks([])
ax_3.set_frame_on(False)
table_3 = ax_3.table(cellText=data3, cellColours=[5*'g']*4,
rowLabels=rowhead, rowColours=['b']*4, colLabels=colhead,
colColours=['r']*5)
table_3.set_fontsize(20)
table_3.scale(8,3)
```

Listing 37: Tabelle 3

	Name	Alter	Stadt	Land	Beruf
Person 1.	Alice	30	New York	USA	Ingenieur
Person 2.	Bob	25	London	UK	Student
Person 3.					
Person 4.	Charlie	35	Berlin	Deutschland	Künstler

Figure 38: Tabelle 3

5.6 Höhenlinien

Um in Python Höhenlinien oder magnetische Felder darstellen zu können, muss ein Netz aus relevanten Punkten über eine Ebene erzeugt werden. Dies kann durch die Numpy-Funktion **np.meshgrid(x,y)** erzeugt werden.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x=y=np.linspace(1,6,6)
5 x,y=np.meshgrid(x,y) # Erzeugt ein Netz in der Ebene
6
7 fig, ax = plt.subplots(figsize=(10,8))
8 ax.plot(x,y,marker='x',color='r',ls='none', ms=9)
```

Listing 38: Meshgrid

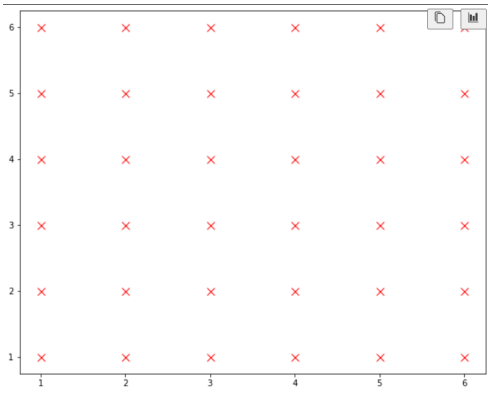

Figure 39: Meshgrid

Um eine Kontur bzw. Höhenliniendiagramm zu erstellen, wird die Funktion **ax.contour(x,y,H,levels=,colors=)**.
Dabei sind x und y die Koordinaten des Netzes, das durch np.meshgrid erzeugt wurde.
H sind die Werte für jeden Punkt in der Ebene. Levels geben an, welche Höhenlinien angezeigt werden sollen.

Zusätzlich wird die Funktion **ax.clabel(cp)** verwendet, um die Höhenlinien anzupassen.
cp ist dabei der Graph und inline gibt an.

Durch die Funktion **ax.set_aspect('equal')** kommt es nicht durch den Bildschirm zu Verzerrungen.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  I = 62.8
5  rmax = 10
6  n=100
7  lvl=[1,2,4,8,16]
8
9  x=y=np.linspace(-rmax,+rmax,n)
10 x,y=np.meshgrid(x,y)
11
12 H = I/(2*np.pi*np.hypot(x,y))
13
14 fig,ax = plt.subplots(figsize=(10,8))
15 cp = ax.contour(x,y,H,levels=lvl,colors='red')
16 ax.clabel(cp)
17 ax.set_aspect('equal')
```

Listing 39: Höhenlinien

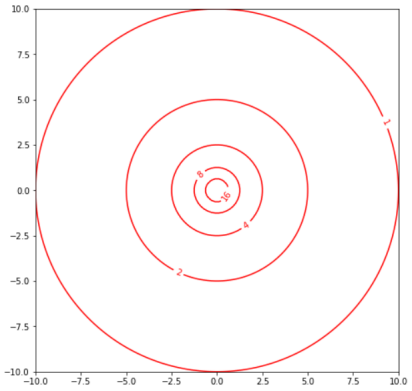

Figure 40: Höhenlinien mit equal

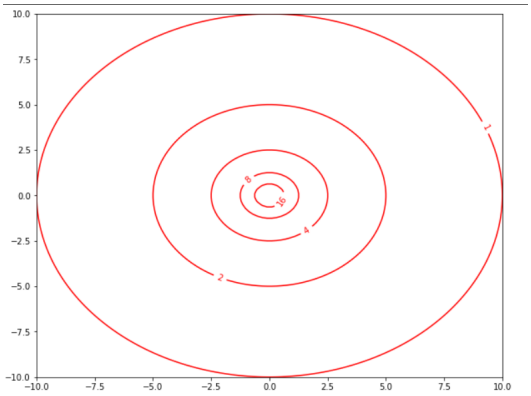

Figure 41: Höhenlinien ohne equal

5.7 Entfernen von Koordinatenachsen und Rändern

Um Felder oder Formen nicht in einen Koordinatensystem zu veranschaulichen, wie zum Beispiel Feldlinien, können die Ränder und Achsen leicht wie folgt entfernt werden.
ax.set_xticks([])
ax.set_yticks([])
ax.set_frame_on(False)

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x=y=np.linspace(1,6,6)
5  x,y=np.meshgrid(x,y) # Erzeugt ein Netz in der Ebene
6
7  fig, ax = plt.subplots(figsize=(10,8))
8  ax.plot(x,y,marker='x',color='r',ls='none', ms=9)
9  ax.set_xticks([])
10 ax.set_yticks([])
11 ax.set_frame_on(False)
```

Listing 40: Entfernung von Koordinatenachsen

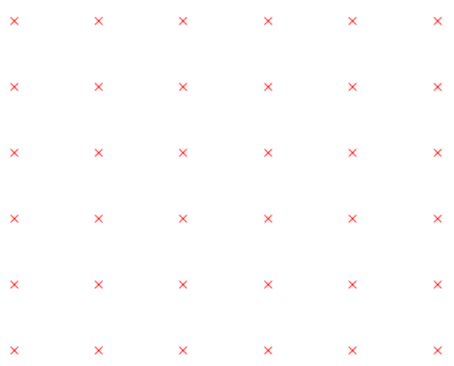

Figure 42: Entfernen von Koordinantenachsen

5.8 3D-Plots

5.8.1 3D-Linien

Die Funktion **plt.figure(figsize=).add_subplot(projection='3d')** ermöglicht die Visualisierung von dreidimensionalen Objekten, die neben den Achsen x und y auch eine z-Komponente aufweisen. Diese Objekte sind typischerweise keine Körper, sondern können beispielsweise Kurven oder punktuelle Datenrepräsentationen darstellen.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  A = 6
5  h = 5
6  omega = 3
7
8  t = np.linspace(0,2*np.pi,500)
9  x = A * np.cos(omega * t)
10 y = A * np.sin(omega * t)
11 z = h * t
12
13 ax = plt.figure(figsize=(6,6)).add_subplot(projection='3d')
14 ax.plot(x,y,z,lw=2)
```

Listing 41: 3D-Linien

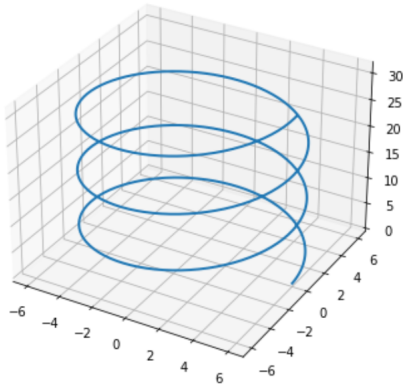

Figure 43: 3D-Linie

5.8.2 3D-Körper

Für das Erstellen von 3D-Körpern wird die Methode **ax.plot_surface(x, y, z, rstride=, cstride=, color=, edgecolors=)** verwendet. Hierbei legt der Parameter rstride die Schrittweite der horizontalen Linien und cstride die Schrittweite der vertikalen Linien fest. Die Option color definiert die Farbe der Flächenabschnitte, während edgecolors die Farben der horizontalen und vertikalen Linien festlegt.

Zur Erzeugung des Gitters für die Koordinaten ist die Numpy-Funktion np.meshgrid() erforderlich.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  n = 100
5  R = 2
6  r = 1
7
8  p = np.linspace(0,2*np.pi,n)
9  t = np.linspace(0,2*np.pi,n)
10 p,t = np.meshgrid(p, t)
11
12 x = (R + r * np.cos(p)) * np.cos(t)
13 y = (R + r * np.cos(p)) * np.sin(t)
14 z = r * np.sin(p)
15
16 ax = plt.figure(figsize=(6,6)).add_subplot(projection='3d')
17 ax.plot_surface(x,y,z,rstride=5,cstride=5,color='y',edgecolors='r')
18 ax.set_zlim(-3,3)
```

Listing 42: 3D-Körper

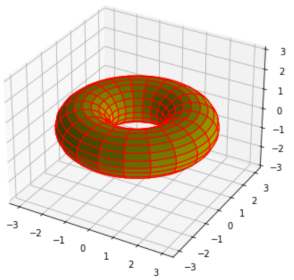

Figure 44: 3D-Körper

5.9 Vektoren

5.9.1 2D-Darstellung

Die Visualisierung eines 2D-Vektors wird mit der Methode ax.quiver(Fx, Fy, angles="xy", scale_units="xy", scale=1, color=) erreicht. Hierbei gibt angles="xy" den Ausgangspunkt des Vektors an. scale_units="xy" verwendet die Einheit der Achsen für die Skalierung. Der Parameter scale=1 definiert den Skalierungsfaktor des Vektors. Die Farbe des Vektors kann mit dem Parameter color= festgelegt werden.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  F1 = np.array((-6,4))
5  F2 = np.array((4,-8))
6  F3 = np.array((4,2))
7
8  Fres = F1 + F2 + F3
9
10 fig, ax = plt.subplots()
11 ax.axis([-8,8,-9,8])
12 ax.quiver(F1[0], F1[1], angles="xy", scale_units="xy", scale=1, color='m')
13 ax.quiver(F2[0], F2[1], angles="xy", scale_units='xy', scale=1, color='g')
14 ax.quiver(F3[0], F3[1], angles="xy", scale_units='xy', scale=1, color='b')
15 ax.quiver(Fres[1], Fres[1], angles="xy",
16 scale_units='xy', scale=1, color='r', label="Fres")
17 ax.legend(loc=0)
```

Listing 43: 2D-Vektoren

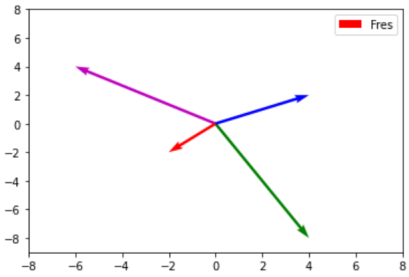

Figure 45: Vektoren 2D

5.9.2 3D-Darstellung

Für die Visualisierung von Vektoren im 3D-Raum kann dieselbe Funktion wie im 2D-Raum verwendet werden, jedoch ohne die Parameter angles="xy", scale_units="xy" und scale=1.

ax.quiver(0, 0, 0, Fx[0], Fy[1], Fz[2], color='r', lw=3)

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  F1 = np.array((3, 4, 2))
5  F2 = np.array((5, 2, -4))
6  Fres = F1 + F2
7
8  ax = plt.figure(figsize=(6,6)).add_subplot(projection='3d')
9  ax.set_xlim(-8,8)
10 ax.set_ylim(-8,8)
11 ax.set_zlim(-8,8)
12 ax.quiver(0, 0, 0, F1[0], F1[1], F1[2], color='r', lw=3)
13 ax.quiver(0, 0, 0, F2[0], F2[1], F2[2], color='g', lw=3)
14 ax.quiver(0, 0, 0, Fres[0], Fres[1], Fres[2], color='b', lw=3, label='Fres')
15 ax.legend(loc=0)
```

Listing 44: 3D-Vektoren

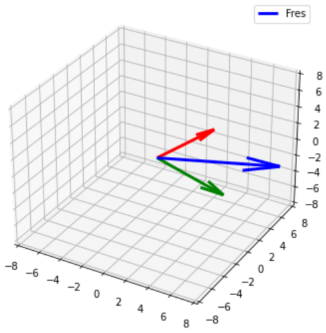

Figure 46: Vektoren 3D

5.9.3 Vektorfelder

Wenn man np.meshgrid() für x und y verwendet, wird ein Gitternetz erzeugt, das die Basis für die Darstellung eines Vektorfelds bildet.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(0, 12, 10)
5  y = np.linspace(0, 12, 10)
6
7  u = 2 # Länge
8  v = 0 # Richtung
9
10 x,y = np.meshgrid(x,y)
11
12 fig, ax = plt.subplots()
13 ax.quiver(x,y,u,v,units='xy',scale=2, color='b')
```

Listing 45: Vektorfelder

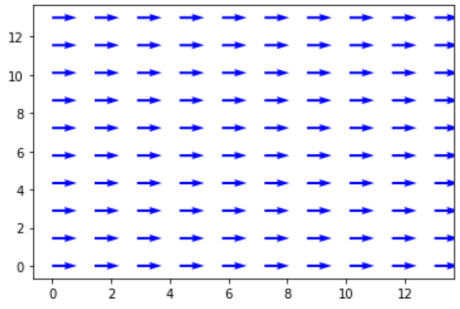

Figure 47: Vektorfeld

5.10 Geometrie

5.10.1 Rechteck

Um ein Rechteck mit Matplotlib zu erstellen, verwendet man die Funktion **mlt.patches.Rectangle((x1, y1), b, h, fill=, lw=, edgecolor=, angle=)**. Die Parameter (x1, y1) geben die Koordinaten der linken unteren Ecke an, b ist die Breite, h ist die Höhe.
Mit fill kann festgelegt werden, ob das Rechteck farbig ausgefüllt sein soll,
lw skaliert die Linienstärke,
edgecolor bestimmt die Randfarbe,
und angle ermöglicht es, das Rechteck um einen bestimmten Winkel zu drehen.

Um das Rechteck zur Zeichenebene hinzuzufügen, wird **ax.add_patch(rechteck)** verwendet.

```
1  import matplotlib as mlt
2  import matplotlib.pyplot as plt
3
4  fig, ax = plt.subplots()
5  ax.axis([-8,8,-8,8])
6  ax.set_xticks([])
7  ax.set_yticks([])
8  ax.set_frame_on(False)
9
10 r = mlt.patches.Rectangle((3,3),4,2, fill=False,lw=3,edgecolor='r')
11 r2 = mlt.patches.Rectangle((-3,-3),4,2, fill=True,color='b',lw=3,edgecolor='b')
12
13 ax.add_patch(r)
14 ax.add_patch(r2)
```

Listing 46: Rechteck

Figure 48: Rechteck

5.10.2 Kreis

Die Funktion **plt.patches.Circle((x, y), Radius, fill=, lw=, edgecolor=)** dient dazu, einen Kreis zu erstellen und zu zeichnen. Anschließend kann dieser Kreis mithilfe der Funktion **ax.add_patch(circle)** in den Plot eingefügt werden.

(x,y) - Die Koordinaten des Mittelpunktes des Kreises.
Radius - Der Radius des Kreises.
fill - Einstellung, ob der Kreis farbig ausgefüllt sein soll
lw - Linienbreite
edgecolor - Einstellung der Linienfarbe

```
1  import matplotlib as mlt
2  import matplotlib.pyplot as plt
3
4  r1 = 10
5  r2 = 5
6
7  fig, ax = plt.subplots()
8  kreis1 = mlt.patches.Circle((0,0), r1, fill=False, lw=3, edgecolor='b')
9  kreis2 = mlt.patches.Circle((0,0), r2, fill=True, lw=3, edgecolor='r')
10 ax.add_patch(kreis1)
11 ax.add_patch(kreis2)
12 ax.set_xticks([])
13 ax.set_yticks([])
14 ax.set_frame_on(False)
15 ax.axis([-15,15,-15,15])
16 ax.set_aspect('equal')
```

Listing 47: Kreis

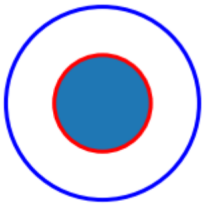

Figure 49: Kreis

5.11 Animationen

5.11.1 Animation: Sinus

Um Animationen in Matplotlib zu erstellen, benötigt man die Methode **FuncAnimation**, die mittels **from matplotlib.animation import FuncAnimation** importiert wird.

Zuerst definiert man die zu animierende Funktion und die Variable, in die Richtung, in der sich die Animation bewegen soll, mittels eines weiteren Parameters k, der die Animation steuert.

```
1 def f(x, frames):
2     return np.sin(x - frames / 20)
```

Listing 48: Funktion zum Animieren

Anschließend wird die Funktion definiert, die die Animation updatet.

```
1 def ani_update(frame):
2      line   .set_ydata(sinus(x,frame))
3      return    line ,
```

Listing 49: Update Funktion

Dabei tut die Funktion line.set_ydata() die y-Werte nach jedem Frame. Die Funktion muss einen Tupel zurückgeben, daher muss line, benutzt werden.

Zum Schluss wird die FuncAnimation benutzt mit **ani = FuncAnimation(fig, ani_update, interval=20)**, dabei ist repräsentiert fig den Plot, ani_pdate die Funktion für die Animation und Intervall gibt die Pause zwischen den zu generierenden Bildern in Millisekunden an.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.animation import FuncAnimation
4
5  def sinus(x,frame):
6        return np.sin(x-frame/20) # -frame, damit sich der
7      Winkel der Sin-funktion mit jeden Frame verschiebt
8  def ani_update(frame):
9        line.set_ydata(sinus(x,frame))
10       return line,
11
12 fig, ax = plt.subplots()
13 x = np.linspace(0,2*np.pi,200)
14 line, = ax.plot(x, sinus(x,0), 'r-', lw=3)
15 ani = FuncAnimation(fig, ani_update, interval=20)
16 ani.save('ani.gif', writer='pillow', fps=50, dpi=100)
```

Listing 50: Sinus Animation

5.11.2 Animation: Punkt

Um einen Punkt zu animieren, wird die Variable "Lauf" als Frames definiert, wobei für jeden Datenpunkt ein eigener Frame erstellt und berechnet wird. Dies erfolgt durch das Hinzufügen des Parameters "frames=" in die FuncAnimation-Funktion.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.animation import FuncAnimation
4
5  # Funktion Definieren f r die Animation
6  def sinus(x):
7          y = np.sin(x)
8          pl.set_data(x, y)
9          return pl,
10
11 # Laufvariable bzw. Anzahl der Frames
12 x = np.linspace(0,4*np.pi,100)
13
14 fig = plt.figure()
15 ax = fig.add_subplot()
16 pl, = ax.plot([], [], 'ro', ms=5)
17 ax.axis([0,4*np.pi,-1.5,1.5])
18 ax.grid(True)
19 ani = FuncAnimation(fig, sinus, frames=x, interval=20)
```

Listing 51: Punkt Animation

5.11.3 Animation: Funktion schritt für schritt

Um eine Funktion schrittweise zu animieren, geht man ähnlich vor wie im vorherigen Kapitel zur Animation eines Punktes. Allerdings muss hier innerhalb der Funktion eine zweite Laufvariable erstellt werden, damit nicht nur der Punkt zum Zeitpunkt "t" berechnet wird, sondern auch alle Punkte, die vor diesem Zeitpunkt liegen.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.animation import FuncAnimation
4
5  def sinus(x):
6          x2 = np.linspace(0, x, int(25*x))
7          y = np.sin(x2)
8          pl.set_data(x2, y)
9          return pl,
10
11 x = np.linspace(0,4*np.pi,100)
12 fig = plt.figure()
13 ax = fig.add_subplot()
14 pl, = ax.plot([], [], 'r', lw=3)
15 ax.grid(True)
16 ax.axis([0,4*np.pi,-1.5,1.5])
17 ani = FuncAnimation(fig, sinus, frames=x, interval=20)
18 plt.show()
```

Listing 52: Funktion schritt für schritt

List of Figures

Codeverzeichnis