Bausteine Computergestützter Datenanalyse

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Franca Hollmann
Maik Poetzsch Sebastian Seipel

2025-11-06

Inhaltsverzeichnis

Methodenbaustein Datenfitting und Datenoptimierung

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch, und Sebastian Seipel. 2025. “Bausteine Computergestützter Datenanalyse. Methodenbaustein Datenfitting und Datenoptimierung”. https://github.com/bausteine-der-datenanalyse/m-datenfitting-und-optimierunge.

BibTeX-Vorlage

```
@misc{BCD-m-datenfittung-und-optimierung-2025,
 title={Bausteine Computergestützter Datenanalyse.  Methodenbaustein Datenfitting und Datenop
 author={Arnold, Lukas and Arnold, Simone and Bagemihl, Florian and Baitsch, Matthias and Fe
 year={2025},
 url={https://github.com/bausteine-der-datenanalyse/m-datenfitting-und-optimierung}}
```

Voraussetzungen

Die Bearbeitungszeit dieses Bausteins beträgt circa **Platzhalter**. Für die Bearbeitung dieses Bausteins werden folgende Bausteine vorausgesetzt:

- Werkzeugbaustein Python
- **to do**

In diesem Baustein werden die folgenden Module und Pakete verwendet:

- numpy
  - numpy.polynomial
- matplotlib **to do**

Querverweis auf:

to do

Im Baustein werden folgende Daten verwendet:

Lernziele

In diesen Baustein lernen Sie ...

to do

1 Einleitung

Experiementell gewonnene Daten können stark verrauscht sein oder die Beziehung der Veriablen wird am besten durch einen nicht-linearen Zusammenhang beschrieben. Datenfitting ist der Prozess, ein Modell an einen Datensatz anzupassen, um die zugrundeliegende Beziehung zwischen den Variablen zu beschreiben, die Daten zu glätten oder Werte zwischen den vorhandenen Datenpunkten zu schätzen. Das Ziel dieses Prozesses ist es, eine Funktion zu finden, die den Datensatz so gut wie möglich beschreibt, indem die Abweichung zwischen dem Modell und den tatsächlichen Daten minimiert wird. Man nennt diesen Prozess auch Modellierung.

In diesem Baustein werden die folgenden Module verwendet:

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

Die Modellierung von Daten kann auf folgendes Problem verallgemeinert werden:

1. Gegeben sind n Messpunktpaare (x_i, y_i) mit $x_i, y_i \in \mathbb{R}$.
2. Gesucht ist eine Modellfunktion $y(x)$, welche die Messpunktpaare approximiert.

Ein möglicher Ansatz ist die Darstellung der Modellfunktion als Summe von m Basisfunktionen $\phi_i(x)$ mit den Koeffizienten β_i.

$$y(x) = \sum_{i=1}^{m} \beta_i \cdot \phi_i(x) = \beta_1 \cdot \phi_1(x) + \cdots + \beta_m \cdot \phi_m(x)$$

Die Koeffizienten β_i müssen dabei so bestimmt werden, dass $y(x)$ so gut wie möglich – oder gar exakt – die Messpunkte approximiert.

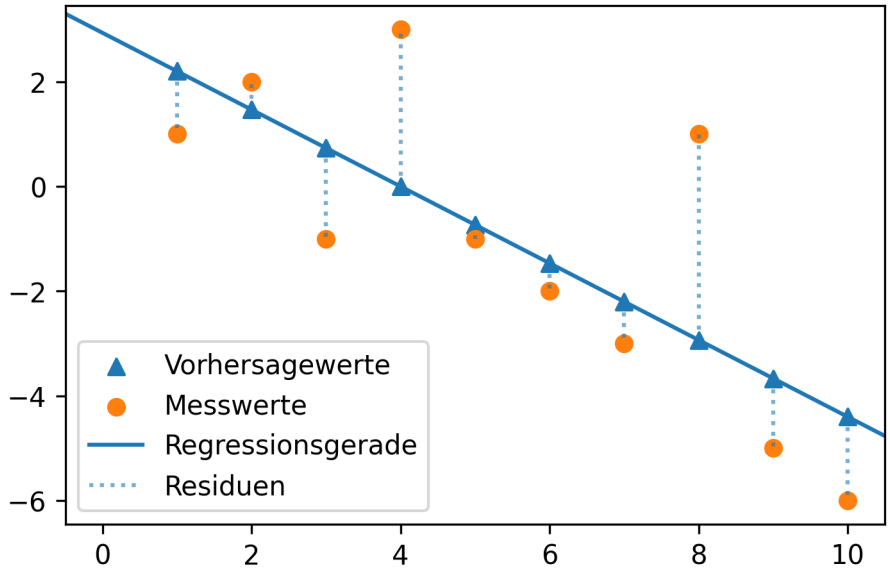
Als Abstandmaß zwischen einer Modellfunktion und den Messpunkten kann die Methode der kleinsten Quadrate genutzt werden.

> **Hinweis 1: Methode der kleinsten Quadrate**
>
> Mit der Methode der kleinsten Quadrate soll diejenige Gerade $\hat{y} = \beta_0 + \beta_1 \cdot x$ gefunden werden, die die quadrierten Abstände der Vorhersagewerte $\hat{y}$ von den tatsächlich gemessenen Werten y minimiert. Die Werte $y_i - \hat{y}_i$ sind die Residuen e_i. Es gilt also:

$$\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} e_i = \min$$

Grafisch kann man sich die Minimierung der quadrierten Abstände so vorstellen.

1.1 Grafik




```
Regressionskoeffizienten: [ 2.93333333 -0.73333333]
```

1.2 Code

```
x = np.arange(1, 11)
y = - x.copy() + 4
y[0] -= 2
y[2] -= 2
y[3] += 3
y[-3] += 5

lm = poly.polyfit(x, y, 1)
vorhersagewerte = poly.polyval(x, lm)

plt.scatter(x, vorhersagewerte, label = 'Vorhersagewerte', marker = "^", color = "tab:blue'
plt.scatter(x, y, label = 'Messwerte', marker = 'o', color = "tab:orange")
plt.axline(xy1 = (0, lm[0]), slope = lm[1], label = "Regressionsgerade", color = "tab:blue'
dotted = plt.vlines(x, ymin = vorhersagewerte, ymax = y, alpha = 0.6, ls = 'dotted', label

plt.legend()
plt.show()

print("Regressionskoeffizienten:", lm)
```

Die eingezeichnete Gerade entspricht der linearen Funktion $\hat{y} = \beta_0 + \beta_1 \cdot x + e_i$. Die Dreiecksmarker sind die Vorhersagewerte $\hat{y_i}$ des linearen Modells für die Werte $x_i = np.arange(1, 11)$. Die tatsächlichen Messwerte y sind mit Kreismarkern markiert. Die Länge der gestrichelten Linien entspricht der Größe der Abweichung zwischen den Mess- und Vorhersagewerten $y_i - \hat{y_i}$, also den Residuen e_i.
Gesucht wird diejenige Gerade, die die Summe der quadrierten Residuen minimiert. Die gesuchten Werte β_0 und β_1 sind die Kleinst-Quadrate-Schätzer.

$$\beta_0 = \bar{y} - \beta_1 \cdot \bar{x}$$

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Der Vollständigkeit halber leiten wir die Kleinst-Quadrate-Schätzer her. Gesucht werden Werte für β_0 und β_1, damit die Summe der Residuenquadrate $\sum_{i=1}^{n} e_i^2$ möglichst klein wird. Die Residuenquadratsumme ist die Summe der quadrierten Differenzen aus beobachteten Werten y_i und der durch die lineare Funktion vorhergesagten Werte.

$$\sum_{i=1}^{n} e_i^2 \;=\; \sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 \cdot x_i))^2$$

Wir untersuchen also eine Funktion, die von zwei Variablen abhängig ist.

$$f(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 \cdot x_i))^2$$

Das Summenzeichen ist die Kurzschreibweise für eine Summe.

$$f(\beta_0, \beta_1) = (y_1 - (\beta_0 + \beta_1 \cdot x_1))^2 \ + \ (y_2 - (\beta_0 + \beta_1 \cdot x_2))^2 \ + \ ... \ (y_n - (\beta_0 + \beta_1 \cdot x_n))^2$$

Im Minimum der Funktion müssen die beiden partiellen Ableitungen gleich Null sein (Warum das so ist, wird hier leicht verständlich erklärt.)

Hinweis 2: Partielle Ableitung

Die partielle Ableitung ist die Ableitung einer Funktion mit mehreren Variablen nach einer Variablen, wobei die übrigen Variablen als Konstanten behandelt werden. Für eine Funktion $f(x, y) = 2x + y^2$ wird die partielle Ableitung nach x so ausgedrückt:
$\frac{\partial f(x,y)}{\partial x}$

- Das Symbol ist die kursive Darstellung des kyrillischen Kleinbuchstaben (d) und wird als "del" gelesen. Es zeigt an, das eine partielle Ableitung durchgeführt wird.
- Im Zähler steht die Funktion, die abgeleitet werden soll. Im Nenner steht die Variable nach der abgeleitet wird. Der Term wird gelesen als "del f von x und y nach del x".

Die partielle Ableitung $\frac{\partial f(x,y)}{\partial x} = 2.\ y^2$ wird als Konstante behandelt (z. B. 5^2) und ist abgeleitet Null.
Die partielle Ableitung $\frac{\partial f(x,y)}{\partial y} = 2y.\ 2x$ wird als Konstante behandelt (z. B. $2 \cdot 3$) und ist abgeleitet Null.

In beiden partiellen Ableitungen sind x_i und y_i konstant. In der partiellen Ableitung nach β_0 ist außerdem β_1 konstant, in der partiellen Ableitung nach β_1 ist entsprechend β_0 konstant.

1.3 partielle Ableitung nach dem y-Achsenschnittpunkt

Für die partielle Ableitung nach β_0 gilt also nach der Kettenregel für die äußere Funktion (oben) und die innere Funktion (Mitte):

$$\frac{\partial f(\beta_0, \beta_1)}{\partial \beta_0} = 2 \cdot (y_1 - (\beta_0 + \beta_1 \cdot x_1)) \ + \ ... \ (y_n - (\beta_0 + \beta_1 \cdot x_n)) = 2 \cdot \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 \cdot x_i)) \ \cdot (0 - (1 + 0 \cdot 0)$$

Für die partielle Ableitung nach β_0 gilt also:

$$\frac{\partial f(\beta_0, \beta_1)}{\partial \beta_0} = -2 \cdot \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 \cdot x_i)) = 0$$

Diese kann vereinfacht werden, indem der Vorfaktor -2 entfällt (weil $-2 \cdot 0 = 0$ gelten muss) und die Vorzeichen aufgelöst werden. Sodass:

$$\sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \cdot x_i) = 0$$

Man kann auch schreiben:

$$\sum_{i=1}^{n} y_i - \sum_{i=1}^{n} \beta_0 - \sum_{i=1}^{n} \beta_1 \cdot x_i = 0$$

β_0 und β_1 sind Konstanten, sodass gilt $\sum_{i=1}^{n} \beta_0 = \beta_0 \cdot \sum_{i=1}^{n} 1 = \beta_0 \cdot n$ und $\sum_{i=1}^{n} \beta_1 \cdot x_i = \beta_1 \cdot \sum_{i=1}^{n} 1 \cdot x_i$. So gilt:

$$\sum_{i=1}^{n} y_i - n \cdot \beta_0 - \beta_1 \cdot \sum_{i=1}^{n} x_i = 0$$

Jetzt kann man durch n teilen. Dabei entspricht $\frac{\sum_{i=1}^{n} y_i}{n}$ dem arithmetischen Mittelwert von y und $\frac{\sum_{i=1}^{n} x_i}{n}$ dem arithmetischen Mittelwert von x. Somit steht:

$$\bar{y} - \beta_0 - \beta_1 \cdot \bar{x} = 0$$

Umgestellt:

$$\beta_0 = \bar{y} - \beta_1 \cdot \bar{x}$$

1.4 partielle Ableitung nach dem Anstieg

Für die partielle Ableitung nach β_1 ist ebenfalls die Kettenregel anzuwenden, sodass die äußere Funktion (oben) identisch abgeleitet wird:

$$\frac{\partial f(\beta_0, \beta_1)}{\partial \beta_1} = 2 \cdot (y_1 - (\beta_0 + \beta_1 \cdot x_1)) \ + \ ... \ (y_n - (\beta_0 + \beta_1 \cdot x_n)) = 2 \cdot \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 \cdot x_i)) \ \cdot (0 - (0 + 1 \cdot x$$

Für die partielle Ableitung nach β_1 gilt also:

$$\frac{\partial f(\beta_0, \beta_1)}{\partial \beta_1} = -2 \sum_{i=1}^{n} x_i \cdot (y_i - (\beta_0 + \beta_1 \cdot x_i)) = 0$$

Auch diese kann vereinfacht werden, indem der Vorfaktor -2 entfällt (weil $-2 \cdot 0 = 0$ gelten muss) und die Vorzeichen aufgelöst werden. Außerdem kann ausmultipliziert werden:

$$\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} \beta_0 \cdot x_i - \sum_{i=1}^{n} \beta_1 \cdot x_i x_i = 0$$

Wieder können die Konstanten herausgezogen werden:

$$\sum_{i=1}^{n} x_i y_i - \beta_0 \cdot \sum_{i=1}^{n} x_i - \beta_1 \cdot \sum_{i=1}^{n} x_i x_i = 0$$

Jetzt kann man $\beta_0 = \bar{y} - \beta_1 \cdot \bar{x}$ und $\sum_{i=1}^{n} x_i = n \cdot \bar{x}$ einsetzen:

$$\sum_{i=1}^{n} x_i y_i - (\bar{y} - \beta_1 \cdot \bar{x}) \cdot n \cdot \bar{x} - \beta_1 \cdot \sum_{i=1}^{n} x_i x_i = 0$$

Der mittlere Term wird ausmultipliziert und $x_i x_i$ im letzten Term als x_i^2 geschrieben:

$$\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y} - \beta_1 \cdot n\bar{x}\bar{x} - \beta_1 \cdot \sum_{i=1}^{n} x_i^2 = 0$$

Die letzten beiden Terme werden unter Anwendung des Distributivgesetzes $a - b = -(b - a)$ zusammengefasst.

$$\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y} - \beta_1 \cdot (\sum_{i=1}^{n} x_i^2 - n\bar{x}^2) = 0$$

Jetzt kann nach β_1 umgestellt werden. Erst:

$$\beta_1 \cdot (\sum_{i=1}^{n} x_i^2 - n\bar{x}^2) = \sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}$$

Dann:

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}$$

Nun kann zuerst mit $\sum_{i=1}^{n} x_i^2 - n\bar{x}^2 = \sum_{i=1}^{n} (x_i - \bar{x})^2$ umgeformt werden.

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Dann - und das wird gleich gezeigt - mit $\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y} = \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$. Sodass steht:

$$\beta_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Der letzte Schritt wird ausgehend vom Ergebnis gezeigt und beginnt mit dem Ausmultiplizieren:

$$\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = \sum_{i=1}^{n}(x_i y_i - x_i\bar{y} - \bar{x}y_i + \bar{x}\bar{y})$$

Man kann auch schreiben:

$$\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i\bar{y} - \sum_{i=1}^{n} \bar{x}y_i + \sum_{i=1}^{n} \bar{x}\bar{y}$$

$\bar{x}$ und $\bar{y}$ sind Konstanten, sodass $\bar{x} \cdot \sum_{i=1}^{n} y_i$ und $\bar{y} \cdot \sum_{i=1}^{n} x_i$ geschrieben werden kann. $\sum_{i=1}^{n} x_i$ ist gleich $n \cdot \bar{x}$ (analog für y). So ergibt sich:

$$\sum_{i=1}^{n} x_i y_i - \bar{y} \cdot n \cdot \bar{x} - \bar{x} \cdot n \cdot \bar{y} + \sum_{i=1}^{n} \bar{x}\bar{y}$$

Sortieren:

$$\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y} - n\bar{x}\bar{y} + \sum_{i=1}^{n} \bar{x}\bar{y}$$

Der letzte Term $\sum_{i=1}^{n} \bar{x}\bar{y}$ kann auch $n \cdot \bar{x}\bar{y}$ geschrieben werden, sodass sich ergibt:

$$\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y} - n\bar{x}\bar{y} + n\bar{x}\bar{y}$$

Die letzten beiden Terme entfallen somit und es bleibt:

$$\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}$$

(Baitsch, Matthias 2019, S. 73–74)

In diesem Kapitel werden folgende Verfahren für die Modellierung von Daten vorgestellt:

1. die Polynominterpolation,
2. Datenfitting durch Polynome und
3. Datenfitting durch Splines.

2 Interpolation – Lücken schließen

Was tun, wenn Werte fehlen? In vielen Datensätzen gibt es Lücken – zum Beispiel, weil Messungen nur an bestimmten Punkten vorgenommen wurden. Interpolation ist eine Methode, mit der wir Zwischenwerte schätzen können, also Werte innerhalb eines bekannten Wertebereichs.

Im Gegensatz dazu versucht Extrapolation, Werte außerhalb des bekannten Bereichs vorherzusagen – was in der Regel mit größerer Unsicherheit verbunden ist.

Bei der Interpolation wird eine Modellfunktion gesucht, welche die Messdaten exakt abbildet.

i Theorie - Modellierung

Die Modellierung von Daten hat das Ziel, eine Menge von Daten durch einen funktionalen Zusammenhang abzubilden. Daten aus Experimenten oder Simulationen können stark verrauscht und so für eine Weiterverarbeitung nicht geeignet sein. Eine mittelnde Funktion kann den Datensatz stark vereinfachen. Oder es existieren nur wenige Datenpunkte und die Zwischenstellen müssen durch eine Funktion bestimmt werden.
Generell kann die Modellierung von Daten auf folgendes Problem verallgemeinert werden:

1. Gegeben sind n Messpunktpaare (x_i, y_i) mit $x_i, y_i \in \mathbb{R}$
2. Gesucht ist eine Modellfunktion $y(x)$, welche die Messpunktpaare approximiert

Ein möglicher Ansatz ist die Darstellung der Modellfunktion als Summe von m Basisfunktionen $\phi_i(x)$ mit den Koeffizienten β_i.

$$y(x) = \sum_{i=1}^{m} \beta_i \cdot \phi_i(x) = \beta_1 \cdot \phi_1(x) + \cdots + \beta_m \cdot \phi_m(x)$$

Die Koeffizienten β_i müssen dabei so bestimmt werden, dass $y(x)$ so gut wie möglich – oder gar exakt – die Messpunkte approximieren.
Als Abstandmaß zwischen einer Modellfunktion und den Messpunkten kann die L2-Norm verwendet werden. Diese ist definiert als

$$||y(x) - (x_i, y_i)||_2 = \sum_{i=1}^{n} \left(y(x_i) - y_i\right)^2 \quad .$$

Eine solche Norm gibt ein Maß für die Qualität einer Approximation: je kleiner der Abstand, desto besser die Qualität. Dies ermöglicht das Finden optimaler Koeffizienten

und wird beispielsweise in der Methode der kleinsten Quadrate genutzt, in der ein Satz an Koeffizienten gesucht wird, der die L2-Norm minimiert.

2.1 Übersicht

In vielen praktischen Anwendungen werden Polynome als Basisfunktionen der Modellfunktion angenommen. Vorteile von Polynomen:

- Polynome sind leicht zu differenzieren und integrieren.
- Mit Polynomen können beliebige Funktionen angenommen werden, siehe Taylor-Entwicklung.
- Die Auswertung benötigt nur wenige arithmetische Operationen (Addition und Multiplikation) und ist dadurch einfach und schnell anzuwenden.

Ein Beispiel für eine Basis aus Polynomen:

$$\phi_1(x) = 1, \quad \phi_2(x) = x, \quad \phi_3(x) = x^2, \quad \cdots, \quad \phi_m = x^{m-1}$$

2.2 Polynome

Polynome $P(x)$ sind Funktionen in Form einer Summe von Potenzfunktionen mit natürlichzahligen Exponenten $(x^i, i \in \mathbb{N})$ mit den entsprechenden Koeffzienten a_i:

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0, \quad i, n \in \mathbb{N}, a_i \in \mathbb{R}$$

- Als Grad eines Polynoms wird der Term mit dem höchsten Exponenten und nichtverschwindenden Koeffizienten (der sogenannte Leitkoeffizient) bezeichnet.
- Ein Polynom mit Grad n hat n, teilweise komplexe, Nullstellen.

Im NumPy-Modul werden Polynome durch ihre Koeffizienten repräsentiert. Im Allgemeinen wird ein Polynom mit dem Grad n durch folgendes Array dargestellt

```
[a0, a1, a2, ..., an]
```

Beispielsweise für $P(x) = 3 - 2x + 5x^2 + x^3$:

```
P = np.array([3, -2, 5, 1])
print(P)
```

```
[ 3 -2  5  1]
```

Funktionen für die Auswertung von Polynomfunktionen stellt das Paket `numpy.polynomial` bzw. dessen Modul `numpy.polynomial.polynomial` bereit.

```
import numpy.polynomial.polynomial as poly
```

Die Auswertung des Polynoms an einem Punkt oder einem Array erfolgt mit der `poly.polyval(x, c)`-Funktion. Diese berechnet die Funktionswerte für in `x` übergebene Werte mit den Funktionsparametern `c`.

```
x = 1
y = poly.polyval(x, P)
print(f"P(x={x}) = {y}")
```

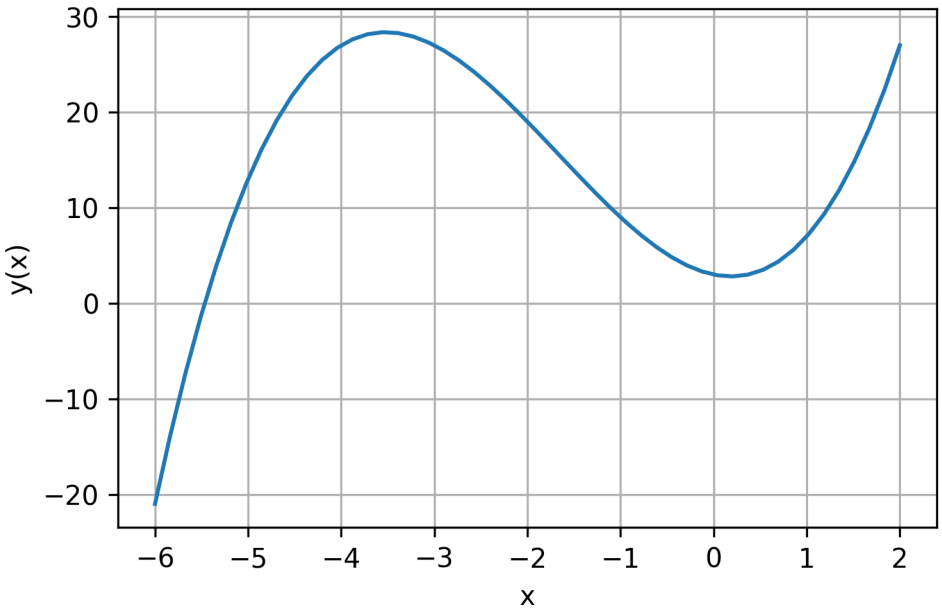
```
P(x=1) = 7.0
```

```
x = np.array([-1, 0, 1])
y = poly.polyval(x, P)
print(f"P(x={x}) = {y}")
```

```
P(x=[-1  0  1]) = [9. 3. 7.]
```

Für die graphische Darstellung im Bereich $x \in [-6, 2]$ kann das Paket Matplotlib verwendet werden.

```
x = np.linspace(-6, 2, 50)
y = poly.polyval(x, P)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y(x)')
plt.grid()

plt.show()
```



Um die Nullstellen eines Polynoms zu finden, kann die Funktion `poly.polyroots()` genutzt werden. Für das obige Polynom können folgende Nullstellen bestimmt werden.

```
nullstellen = poly.polyroots(P)

# direkte Ausgabe des Arrays
print("Nullstellen: ")
print(nullstellen)
```

```
Nullstellen:
[-5.46628038+0.j         0.23314019-0.703182j  0.23314019+0.703182j]
```

```
print("Nullstellen: ")
# schönere Ausgabe des Arrays
for i, z in enumerate(nullstellen):
    if z.imag == 0:
        print(f"  x_{i+1} = {z.real:.2}")
    else:
        print(f"  x_{i+1} = {z.real:.2} {z.imag:+.2}i")
```

```
Nullstellen:
```

```
x_1 = -5.5
x_2 = 0.23 -0.7i
x_3 = 0.23 +0.7i
```

In diesem Beispiel sind zwei der Nullstellen komplex. Eine komplexe Zahl z wird in Python als Summe des Realteils (Re) und Imaginärteils (Im) dargestellt. Letzterer wird durch ein nachfolgendes `j`, die imaginäre Einheit, gekennzeichnet.

$$z = Re(z) + Im(z)j$$

Die Nullstellen können auch zur alternativen Darstellung des Polynoms verwendet werden. Sind x_i die n Nullstellen, so ist das Polynom n-ten Grades durch folgendes Produkt beschrieben:

$$P(x) = \prod_{i=1}^{n} (x - x_i) = (x - x_1) \cdot (x - x_2) \cdot \cdots \cdot (x - x_n)$$

Seien beispielsweise 1 und 2 die Nullstellen eines Polynoms, so lautet dieses:

$$P(x) = (x - 1)(x - 2) = 2 - 3x + x^2$$

Die Funktion `poly.polyfromroots(nullstellen)` kann aus den Nullstellen die Polynomkoeffizienten bestimmen. Anhand des obigen Beispiels lautet der Funktionsaufruf:

```
nullstellen = [1, 2]
koeffizienten = poly.polyfromroots(nullstellen)

print("Nullstellen:", nullstellen)
print("Koeffizienten:", koeffizienten)
```

```
Nullstellen: [1, 2]
Koeffizienten: [ 2. -3.  1.]
```

Das Modul `numpy.polynomial.polynomial` stellt viele praktische Funktionen zum Umgang mit Polynomen zur Verfügung. So existieren weitere Funktionen, um Polynome zu addieren, zu multiplizieren, abzuleiten oder zu integrieren. Eine Übersicht findet sich in der numpy-Dokumentation.

2.3 Polynominterpolation

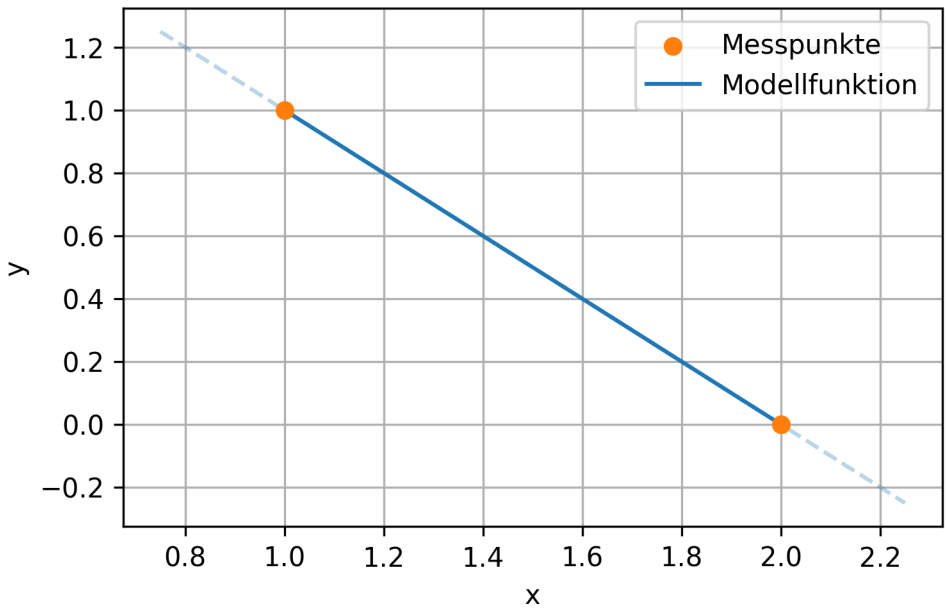
Interpolation ist eine Methode, um Datenpunkte zwischen gegebenen Messpunkten zu konstruieren. Dazu wird eine Funktion gesucht, die alle Messpunkte exakt abbildet, was gleichbedeutend damit ist, dass die L2-Norm zwischen Funktion und Punkten Null ist.

Zwei Punkte können z. B. mit einer Geraden interpoliert werden. Das heißt, für zwei Messpunktpaare (x_1, y_1) und (x_2, y_2) mit $x_1 \neq x_2$ existiert ein Koeffizientensatz, sodass die L2-Norm zwischen den Messpunkten und der Modellfunktion

$$y(x) = \beta_0 + \beta_1 x$$

verschwindet.

```
# Beispieldaten aus y(x) = 2 - x

N = 50
dx = 0.25

def fnk(x):
    return 2 - x

x = np.array([1, 2])
y = fnk(x)

plt.scatter(x, y, color='C1', label="Messpunkte", zorder=3)

x_modell = np.linspace(np.min(x), np.max(x), N)
plt.plot(x_modell, fnk(x_modell), color='C0', label="Modellfunktion")

x_linie = np.linspace(np.min(x)-dx, np.max(x)+dx, N)
plt.plot(x_linie, fnk(x_linie), '--', alpha=0.3, color='C0')

plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()

plt.show()
```

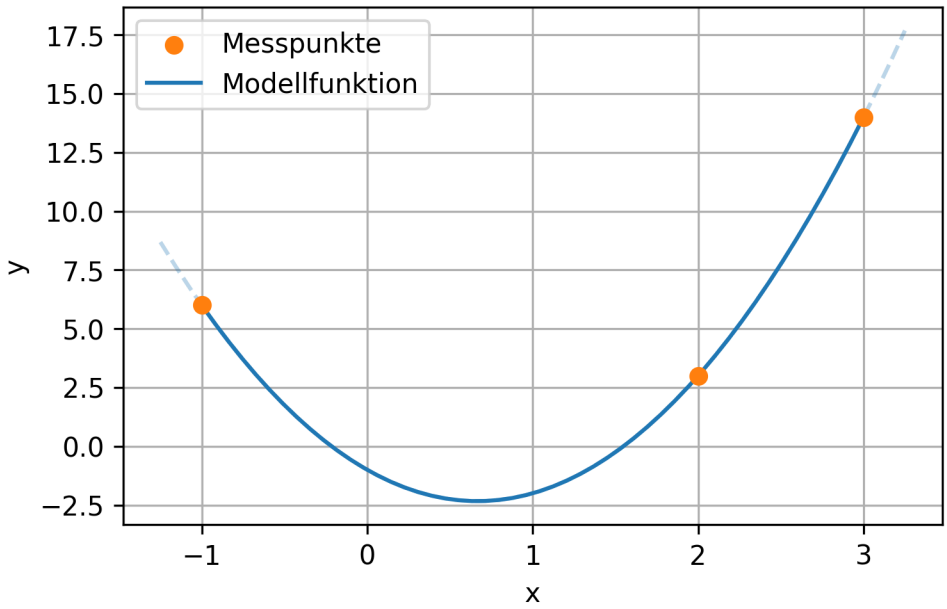


Für drei Messpunkte muss ein Polynom zweiten Grades verwendet werden, um die Punkte exakt zu erfassen.

$$y(x) = \beta_0 + \beta_1 x + \beta_2 x^2$$

```
# Beispieldaten aus y(x) = -1 - 4x + 3x^2

N = 50
dx = 0.25

def fnk(x):
    return -1 - 4*x + 3*x**2

x = np.array([-1, 2, 3])
y = fnk(x)

plt.scatter(x, y, color='C1', label="Messpunkte", zorder=3)

x_modell = np.linspace(np.min(x), np.max(x), N)
plt.plot(x_modell, fnk(x_modell), color='C0', label="Modellfunktion")

x_linie = np.linspace(np.min(x)-dx, np.max(x)+dx, N)
```

```
plt.plot(x_linie, fnk(x_linie), '--', alpha=0.3, color='C0')

plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()

plt.show()
```



Dies kann verallgemeinert werden: n Messpunkte können exakt mit einem Polynom $(n-1)$-ten Grades abgebildet werden. Die Suche nach den passenden Koeffizienten ist das Lagrangesche Interpolationsproblem. Für das gesuchte Polynom $P(x)$ gilt:

$$P(x_i) = y_i \quad i \in 1, \ldots, n$$

Die Existenz und Eindeutigkeit eines solchen Polynoms kann gezeigt werden. Das gesuchte Polynom lautet:

$$P(x) = \sum_{i=1}^{n} y_i I_i(x)$$

$$\text{mit} \quad I_i(x) = \prod_{j=1, i \neq j}^{n} \frac{x - x_j}{x_i - x_j}$$

Alternativ kann auch ein Gleichungssystem, welches durch die Polynomialbasis $\phi_i(x)$ gegeben ist, gelöst werden. Für die n Punktepaare gilt jeweils:

$$y(x_i) = \sum_{i=1}^{m} \beta_i \cdot \phi_i(x_i) = \beta_1 \cdot \phi_1(x_i) + \cdots + \beta_m \cdot \phi_m(x_i) = y_i$$

Das allgemeine Geleichungssystem lautet

$$\begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_m(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_m(x_n) \end{pmatrix} \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

und mit der Polynomialbasis

$$\underbrace{\begin{pmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{pmatrix}}_{V} \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Die Matrix V heisst Vandermonde-Matrix und kann exakt gelöst werden, für $m = n$ und wenn für alle $i, j; i \neq j$ gilt $x_i \neq x_j$.

In Python kann das Interpolationsproblem mit der Funktion `numpy.polynomial.polynomial.polyfit()` gelöst werden, das wir durch durch das Importieren des Moduls mit dem Schlüsselwort `poly` mit `poly.polyfit()` aufrufen können. Das folgende Beispiel demonstriert deren Anwendung.

Die Messtellen folgen in dem Beispiel der Funktion $f(x)$, welche nur zur Generierung der Datenpunkte verwendet wird.

$$f(x) = \frac{1}{2} + \frac{1}{1+x^2}$$

Zunächst werden die Messpunkte generiert.

```
def fnk(x):
    return 0.5 + 1/(1+x**2)
```

```
xmin = -5
xmax =  5
x = np.linspace(xmin, xmax, 100)
y = fnk(x)
```

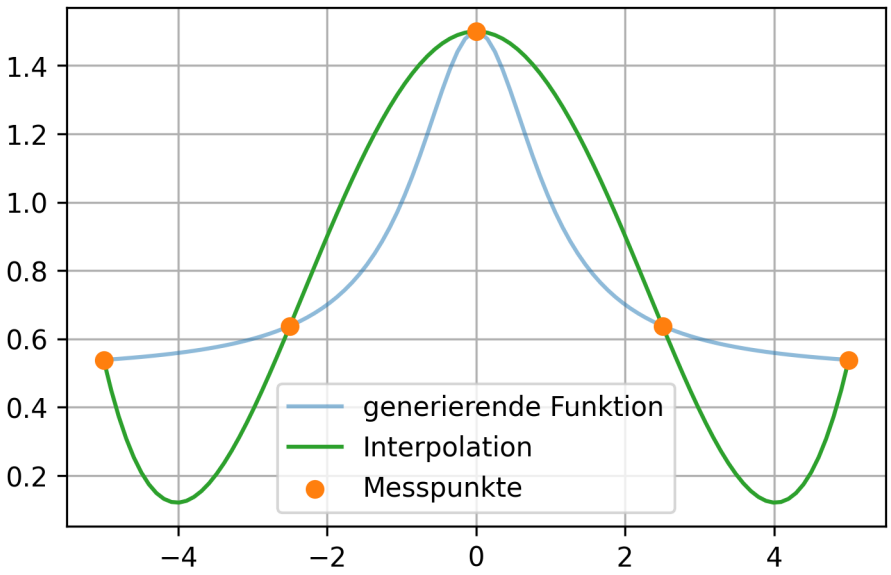
```
n = 5
xi = np.linspace(xmin, xmax, n)
yi = fnk(xi)
```

Nun folgt die Interpolation für 5 und 15 Messpunkte.

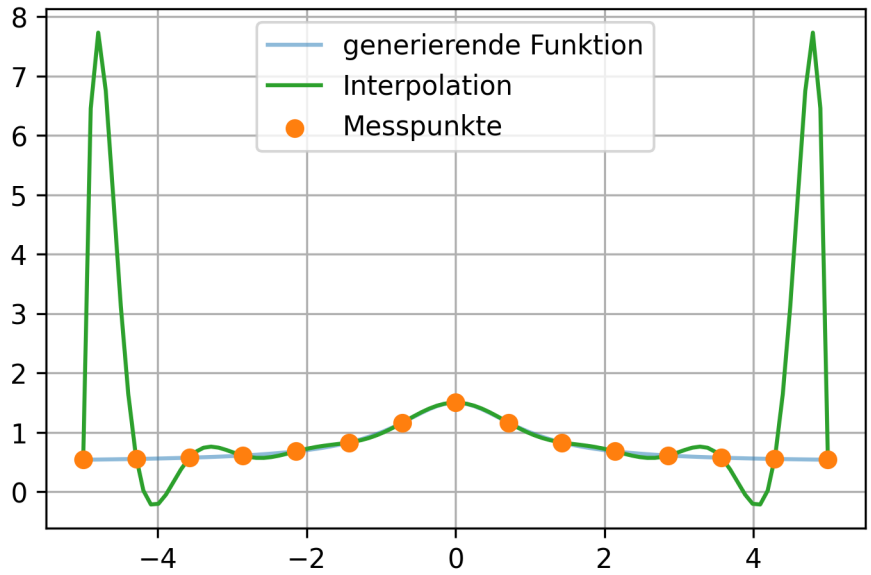
```
P = poly.polyfit(xi, yi, n-1)
print("Interpolationskoeffizienten:")
print(P)
```

```
Interpolationskoeffizienten:
[ 1.50000000e+00 -1.12346671e-16 -1.71087533e-01  6.98020740e-18
  5.30503979e-03]
```

```
plt.plot(x, y, color='C0', alpha=0.5, label='generierende Funktion')
plt.plot(x, poly.polyval(x, P), color='C2', label='Interpolation')
plt.scatter(xi, yi, color='C1', label='Messpunkte', zorder=3)
plt.legend()
plt.grid()

plt.show()
```




```
n = 15
xi = np.linspace(xmin, xmax, n)
yi = fnk(xi)

P = poly.polyfit(xi, yi, n-1)

plt.plot(x, y, color='C0', alpha=0.5, label='generierende Funktion')
plt.plot(x, poly.polyval(x, P), color='C2', label='Interpolation')
plt.scatter(xi, yi, color='C1', label='Messpunkte', zorder=3)
plt.legend()
plt.grid()

plt.show()
```



Die Interpolation erfüllt immer die geforderte Bedingung $y(x_i) = y_i$. Jedoch führen Polynome mit einem hohen Grad oft zu nicht sinnvollen Ergebnissen. Es entstehen starke Überschwinger, welche mit zunehmendem Grad immer stärker werden.

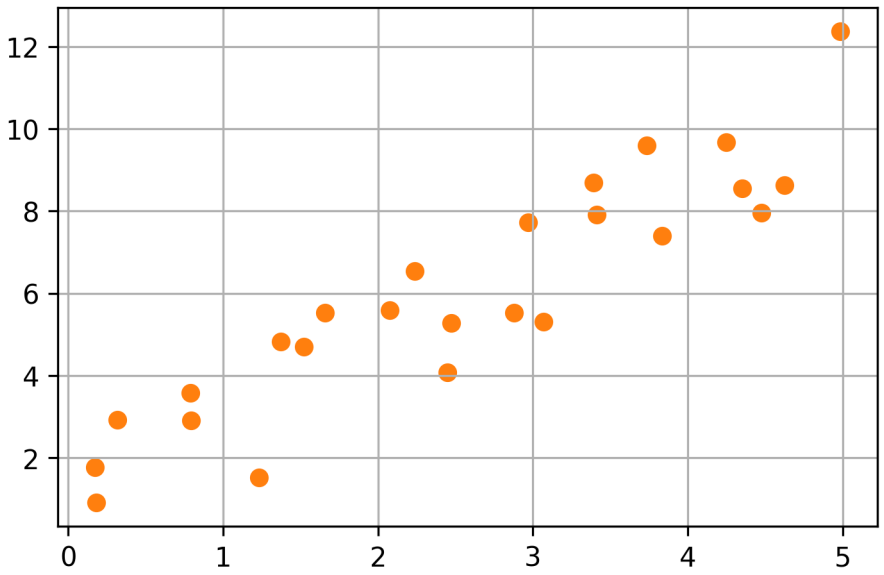
3 Fitting

Beim Fitting wird eine Modellfunktion gesucht, welche die Messdaten nicht unbedingt exakt abbildet. Wird ein Polynom verwendet, so hat es einen Grad, welcher deutlich kleiner ist, als die Anzahl der Messpunkte. Lineare Regression ist ein Beispiel für ein Fitting durch ein Polynom mit dem Grad Eins.

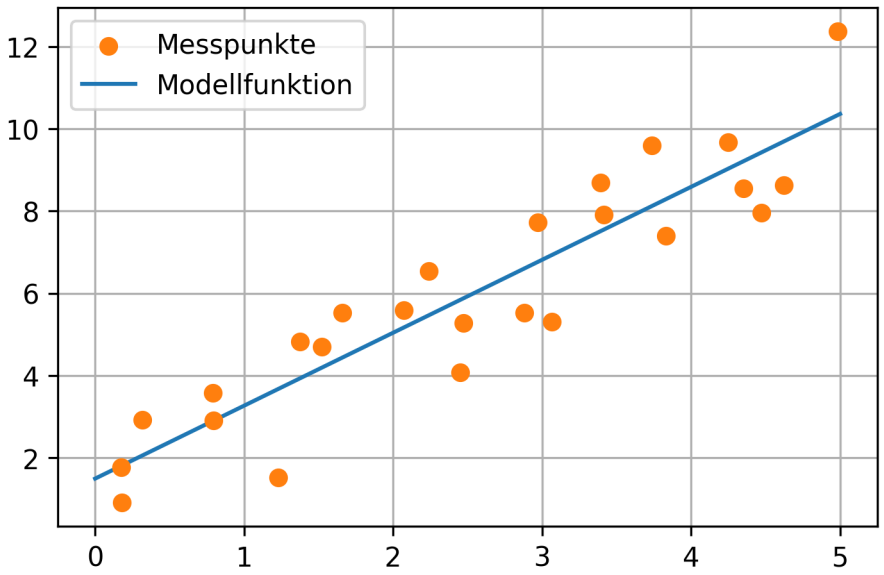
Zum Fitten durch ein Polynom kann die Funktion `numpy.polynomial.polynomial.polyfit()` verwendet werden, genauso wie bei der Polynominterpolation. Diesmal jedoch mit einem kleineren Polynomgrad.

Im folgenden Beispiel werden zunächst Modelldaten generiert und dann mit entsprechenden Polynomen gefittet.

```
xmin = 0
xmax = 5
x = np.linspace(xmin, xmax, 100)

ni = 25

# x-Werte mit leichtem Rauschen
xi = np.linspace(xmin, xmax, ni) + 0.2*(2 * np.random.random(ni) -1)

# y(x) = 2x+0.5 mit leichtem Rauschen
yi = 2*xi + 0.5 + 2*(2 * np.random.random(ni) -1)

plt.scatter(xi, yi, color='C1')
plt.grid()

plt.show()
```



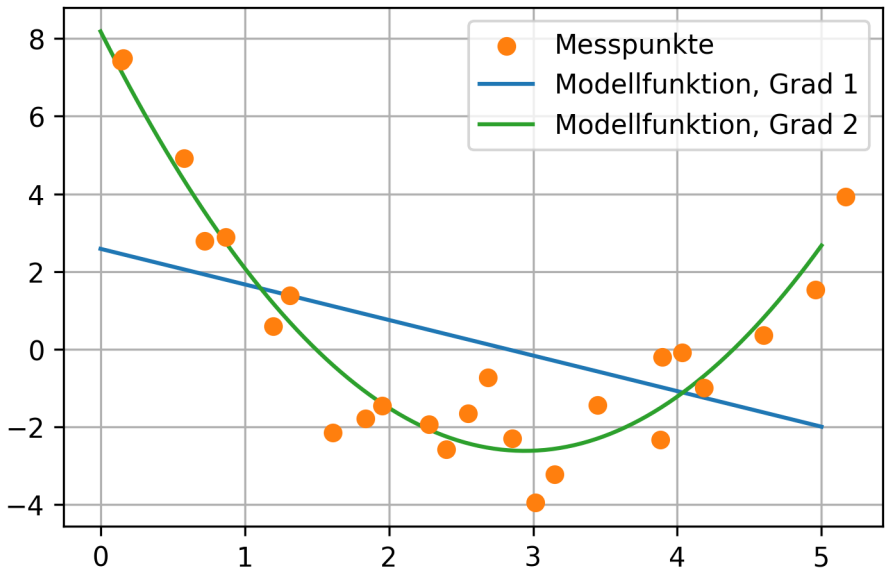

```
P1 = poly.polyfit(xi, yi, 1)

plt.scatter(xi, yi, color='C1', zorder=3, label='Messpunkte')
plt.plot(x, poly.polyval(x, P1), color='C0', label="Modellfunktion")
plt.grid()
plt.legend()

plt.show()
```



In diesem Beispiel werden ein Polynom ersten Grades und ein Polynom zweiten Grades bestimmt.

```
# x-Werte mit leichtem Rauschen
xi = np.linspace(xmin, xmax, ni) + 0.2*(2 * np.random.random(ni) -1)

# y(x) = 2x+0.5 mit leichtem Rauschen
yi = (xi - 2)**2 -2*xi + 2.5 + 2*(2 * np.random.random(ni) -1)
```

```
P1 = poly.polyfit(xi, yi, 1)
P2 = poly.polyfit(xi, yi, 2)

plt.scatter(xi, yi, color='C1', zorder=3, label='Messpunkte')
plt.plot(x, poly.polyval(x, P1), color='C0', label="Modellfunktion, Grad 1")
plt.plot(x, poly.polyval(x, P2), color='C2', label="Modellfunktion, Grad 2")
plt.grid()
plt.legend()

plt.show()
```


Messpunkte
Modellfunktion, Grad 1
Modellfunktion, Grad 2
8
6
4
2
0
−2
−4
0
1
2
3
4
5

4 Splines

Polynominterpolation versucht eine globale Modellfunktion zu finden. Jedoch eignen sich Polynome mit hohen Graden im Allgemeinen nicht für eine Interpolation vieler Punkte. Einen anderen Ansatz verfolgen Splines (auch: Polynomzug), welche mehrere, niedrige Polynome zur Interpolation vieler Punkte verwenden. Die Polynome haben typischerweise Grade zwischen eins und drei.

4.1 Definition

Für $n+1$ Messpunkte (x_i, y_i) kann eine Splinefunktion s_k, hier ein Polynomspline, wie folgt definiert werden:

- Vorausgesetzt ist, dass die Messpunkte sortiert sind, d. h. $x_0 < x_1 < \cdots < x_n$.
- Für jedes $i = 0 \ldots n-1$ ist s_k ein Polynom vom Grad k auf dem Intervall $[x_i, x_{i+1}]$
- s_k ist auf $[x_0, x_n]$ $(k-1)$-mal stetig differenzierbar

Beispiele:

- $k = 1$: Polygonzug
- $k = 3$: kubische Polynomsplines (B-Splines)

4.2 Kubische Splines

Die in der Praxis häufig eingesetzten kubischen Polynomsplines s_3 $(k = 3)$ haben folgende Eigenschaften:

- $s_3 | [x_i, x_{i+1}] = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$
- s_3 ist zweimal stetig differenzierbar auf $[x_0, x_n]$, also insbesondere an den Stützpunkten x_i der Messpunkte.

Die Koeffizienten β_i werden wie folgt bestimmt:

- Aus den $n+1$ Messpunkten ergeben sich n Intervalle, d. h. mit jeweils vier Koeffizienten sind es insgesamt $4n$ Koeffizienten.

- Exakte Darstellung der Messpunkte ($n+1$ Gleichungen), d. h.: $s_3(x_i) = y_i$
- Glattheitsbedingungen an den inneren Messpunkten ($i = 1 \ldots n-1$), mit jeweils ($n-1$ Gleichungen):

$$s_3'(x_i)_- = s_3'(x_i)_+$$
$$s_3''(x_i)_- = s_3''(x_i)_+$$
$$s_3'''(x_i)_- = s_3'''(x_i)_+$$

- Damit sind es $4n-2$ Gleichungen für $4n$ Koeffizienten.

Um die beiden fehlenden Gleichungen zu finden bzw. zu bestimmen, werden Randbedingungen oder Abschlussbedingungen benötigt. Die gängigsten Bedingungen sind:

- natürliche Splines: die Krümung am Rand verschwindet, d. h.:

$$s_3''(x_0) = s_3''(x_n) = 0$$

- periodische Splines: die Steigung und Krümung ist an beiden Rändern gleich

$$s_3'(x_0) = s_3'(x_n)$$

$$s_3''(x_0) = s_3''(x_n)$$

- Hermite Splines: die Steigungen am Rand werden explizit vorgegeben (hier durch u und v)

$$s_3'(x_0) = u$$
$$s_3'(x_n) = v$$

4.3 Anwendung

Im Folgenden werden zwei Beispiele, s_1 und s_3, für die Erstellung von Splines mit Python vorgestellt.

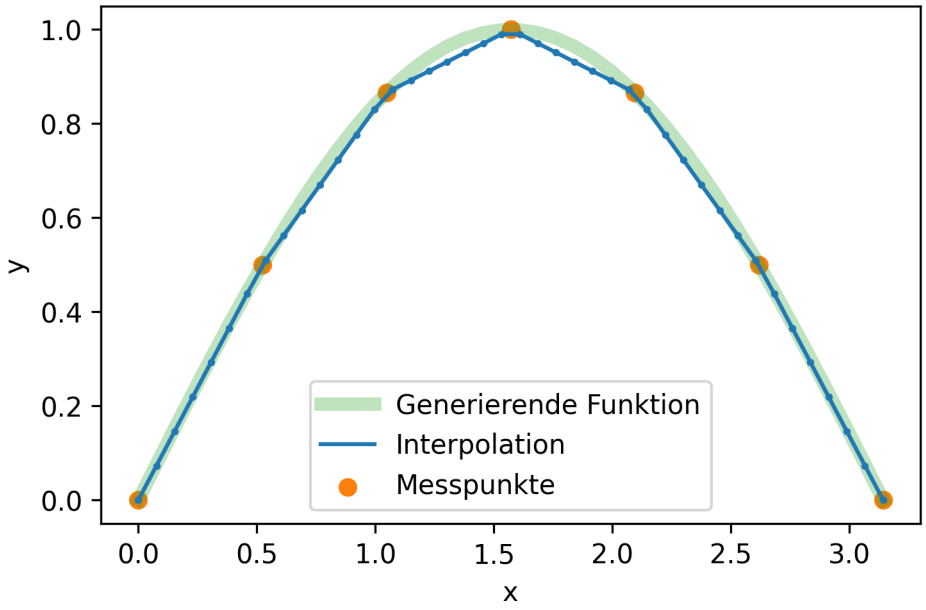
```
# Erzeugung von Messpunkten
n = 7
xi = np.linspace(0, np.pi, n)
yi = np.sin(xi)
```

Für die s_1 Splines, kann die Funktion `np.interp()` verwendet werden. Sie führt eine lineare Interpolation zwischen gegebenen Wertepaaren durch.

```
# Wertebereich für die Visualisierung der Interpolation
x = np.linspace(0, np.pi, n*6)
y = np.sin(x)
```

```
# Interpolation
y_s1 = np.interp(x, xi, yi)
```

```
plt.plot(x,y, alpha=0.3, color='C2', lw=5,
         label='Generierende Funktion')
plt.plot(x, y_s1, color='C0', label='Interpolation')
plt.scatter(x, y_s1, s=3, zorder=3, color='C0')
plt.scatter(xi, yi, color='C1', label='Messpunkte')

plt.xlabel('x')
plt.ylabel('y')
plt.legend();

plt.show()
```



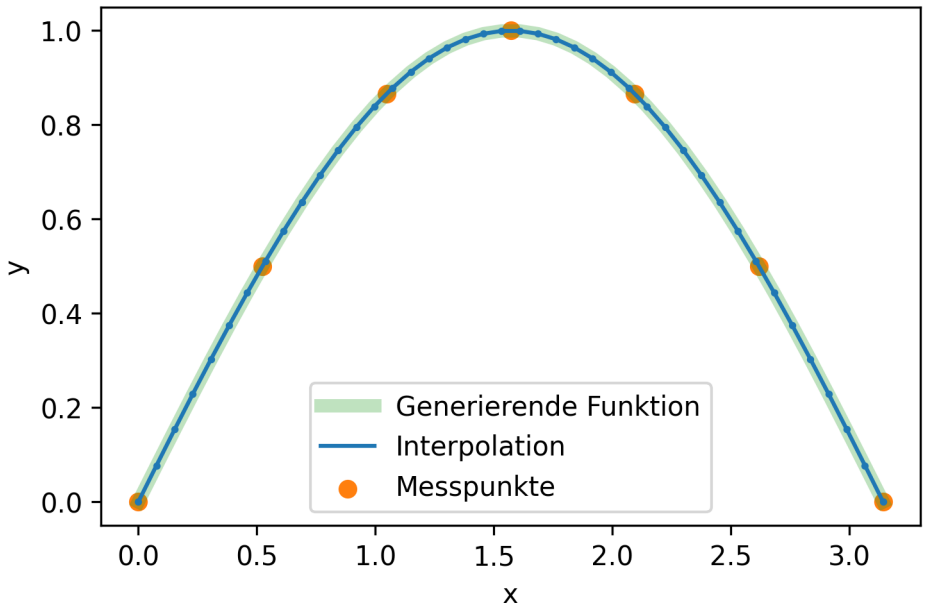
Die s_3 Splines können mit Funktionen aus dem scipy-Modul berechnet werden. Dazu werden zunächst die Koeffizienten bestimmt (`scipy.interpolate.splrep`) und diese ermöglichen die

gewünschte Auswertung, welche mit der Funktion `scipy.interpolate.splev` vorgenommen werden kann.

```
import scipy.interpolate as si
```

```
s3 = si.splrep(xi, yi)
y_s3 = si.splev(x, s3)
```

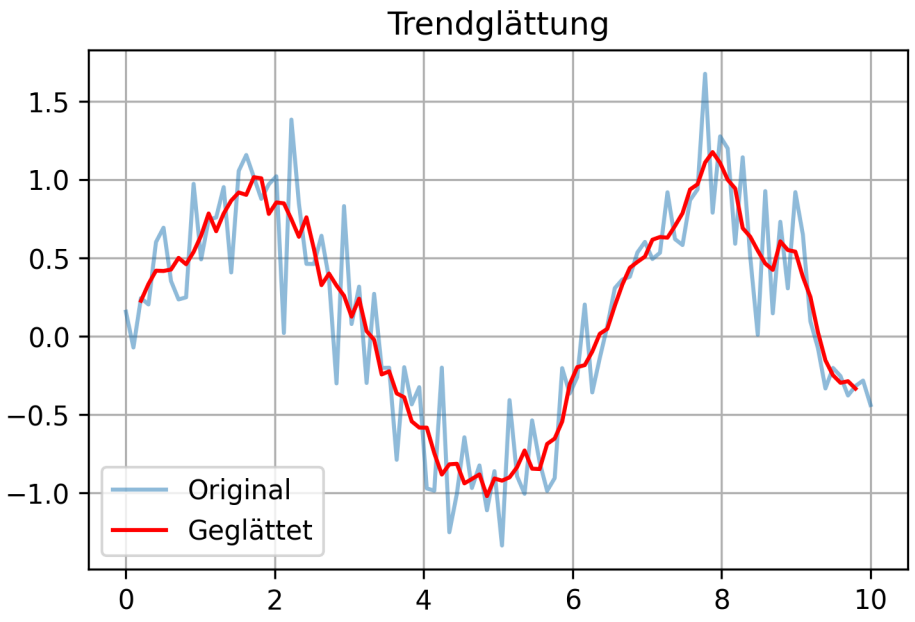
```
plt.plot(x,y, alpha=0.3, color='C2', lw=5,
         label='Generierende Funktion')
plt.plot(x, y_s3, color='C0', label='Interpolation')
plt.scatter(x, y_s3, s=3, zorder=3, color='C0')
plt.scatter(xi, yi, color='C1', label='Messpunkte')

plt.xlabel('x')
plt.ylabel('y')
plt.legend();

plt.show()
```



5 Trendglättung – Rauschen reduzieren

Verrauschte Daten? Ein **gleitender Mittelwert** glättet Kurven:

```
data = np.genfromtxt("01-daten/trenddaten-mit-rauschen.csv", delimiter=",", skip_header=1)
x = data[:, 0]
y = data[:, 1]

window = 5
weights = np.ones(window) / window
y_smooth = np.convolve(y, weights, mode='valid')

plt.plot(x, y, label="Original", alpha=0.5)
plt.plot(x[(window-1)//2:-(window//2)], y_smooth, label="Geglättet", color='red')
plt.legend()
plt.grid(True)
plt.title("Trendglättung")

plt.show()
```


Trendglättung
1.5
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
Original
Geglättet

6 Übungen

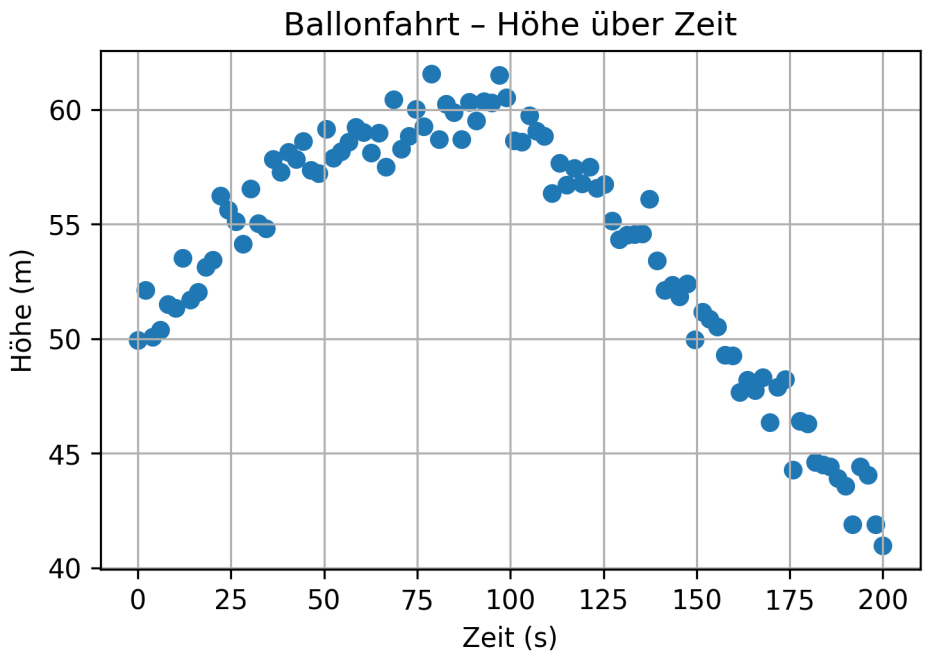
6.1 Übung: Ballonfahrt-Daten analysieren

In der Datei '01-daten/messdaten-ballonfahrt.txt' sind Messdaten zur Höhe eines Ballons während einer Ballonfahrt gespeichert. In der ersten Zeile der Datei sind die gemessenen Größen und ihre Einheit notiert.

```
# Dateikopf als String einlesen
ballon = np.genfromtxt("01-daten/messdaten-ballonfahrt.txt", delimiter=",", max_rows= 4, dtyp

print("Der Kopf der Datei:")
print(ballon)
```

```
Der Kopf der Datei:
[['Zeit (s)' 'Hoehe (m)']
 ['0.0' '49.93175839467537']
 ['2.0202020202020203' '52.11727320291297']
 ['4.040404040404041' '50.0624468191306']]
```

Schauen wir uns die Daten einmal an:



Gesucht ist ein Polynom, das die Steiggeschwindigkeit, also die Änderung der Höhe mit der Zeit, des Heißluftballons beschreibt.

Schreiben Sie ein Programm, das:

- die Höhe aus den Daten ausliest und die Steiggeschwindigkeit berechnet,
- einen Fit der Steiggeschwindigkeit mit einem Polynom dritten Grades durchführt.

Stellen Sie die Höhe, die Geschwindigkeit und den Fit jeweils gegen die Zeit dar. Beschriften Sie die Achsen.

Tipp 1: Musterlösung Ballonfahrt

Zuerst werden die Daten ausgelesen.

```
# Daten auslesen
ballon = np.genfromtxt("01-daten/messdaten-ballonfahrt.txt", delimiter=",", skip_header=1)
zeit = ballon[:, 0]
hoehe = ballon[:, 1]
```

Anschließend berechnen wir die Steiggeschwindigkeit $\frac{\Delta Hhe}{\Delta Zeit}$. Dazu verwenden wir die Funktion `np.diff()`. Diese berechnet die Differenz jedes Werts zu seinem Vorgänger `ergebnis[i] = wert[i+1] - wert[i]`. Für den i-ten Wert wird also keine Differenz be-

rechnet und das Ergebnis ist um ein Element kürzer. Die Steiggeschwindigkeit in $\frac{m}{s}$ ergibt sich aus dem Quotienten beider Reihen.

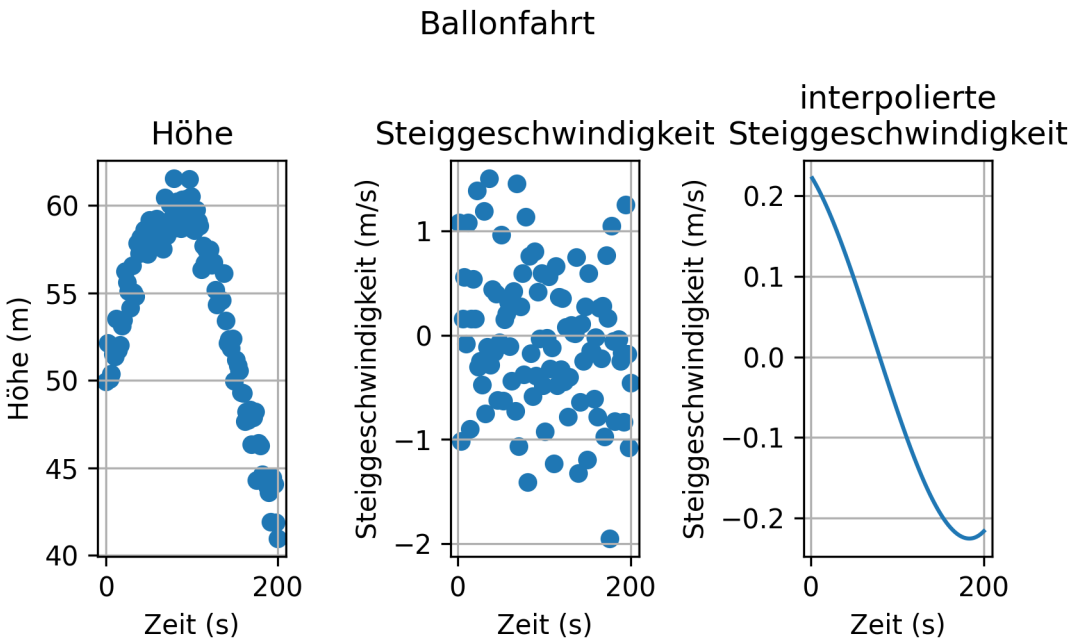
```
# Differenzen berechnen
delta_hoehe = np.diff(hoehe)
print("Veränderung der Höhe: ", delta_hoehe[0:4])

delta_zeit = np.diff(zeit)
print("Veränderung der Zeit: ", delta_zeit[0:4])

# Steiggeschwindigkeit berechnen
steiggeschwindigkeit = delta_hoehe / delta_zeit
print("Steiggeschwindigkeit: ", steiggeschwindigkeit[0:4])
```

```
Veränderung der Höhe:  [ 2.18551481 -2.05482638  0.32026988  1.12796595]
Veränderung der Zeit:  [2.02020202 2.02020202 2.02020202 2.02020202]
Steiggeschwindigkeit:  [ 1.08182983 -1.01713906  0.15853359  0.55834314]
```

Mit `numpy.polynomial.polynomial.polyfit()` berechnen wir erst ein Polynom dritten Grades und dann mit `numpy.polynomial.polynomial.polyval()` die gefitteten Daten.

```
polynom3_steiggeschwindigkeit = poly.polyfit(zeit[1:], steiggeschwindigkeit, deg = 3)

fit_steiggeschwindigkeit = poly.polyval(x = zeit[1:], c = polynom3_steiggeschwindigkeit)
```

```
# plotten
plt.suptitle("Ballonfahrt")

## subplot Höhe über Zeit
plt.subplot(1, 3, 1)
plt.scatter(zeit, hoehe)
plt.title("Höhe")
plt.xlabel("Zeit (s)")
plt.ylabel("Höhe (m)")
plt.grid(True)

## subplot Steiggeschwindigkeit über Zeit
## Länge des Arrays Steiggeschwindigkeit ist n-1
plt.subplot(1, 3, 2)
plt.scatter(zeit[1:], steiggeschwindigkeit)
plt.title("Steiggeschwindigkeit")
plt.xlabel("Zeit (s)")
plt.ylabel("Steiggeschwindigkeit (m/s)")
plt.grid(True)

## subplot Polynom 3. Grades Steiggeschwinkdigkeit über Zeit
## Länge des Arrays Steiggeschwindigkeit ist n-1
plt.subplot(1, 3, 3)
plt.plot(zeit[1:], fit_steiggeschwindigkeit)
plt.title("interpolierte\nSteiggeschwindigkeit")
plt.xlabel("Zeit (s)")
plt.ylabel("Steiggeschwindigkeit (m/s)")
plt.grid(True)

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



6.2 Übung: Balkenverformung im Bauingenieurwesen

Ein Träger wird in der Mitte belastet.

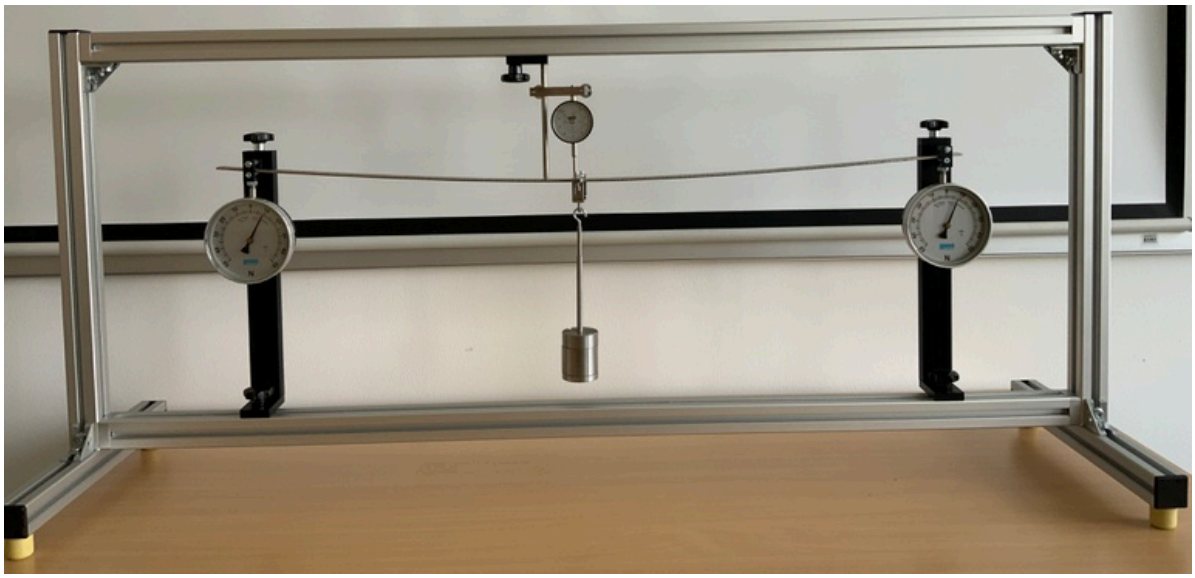


Abbildung 6.1: Beispiel für einen Versuchsaufbau Balkenverformung

Balkenverformung von Universität der Bundeswehr München. Das Werk ist abrufbar auf der Internetseite der UniBw M. ohne Jahr

Schauen wir uns die Struktur der Daten einmal an.

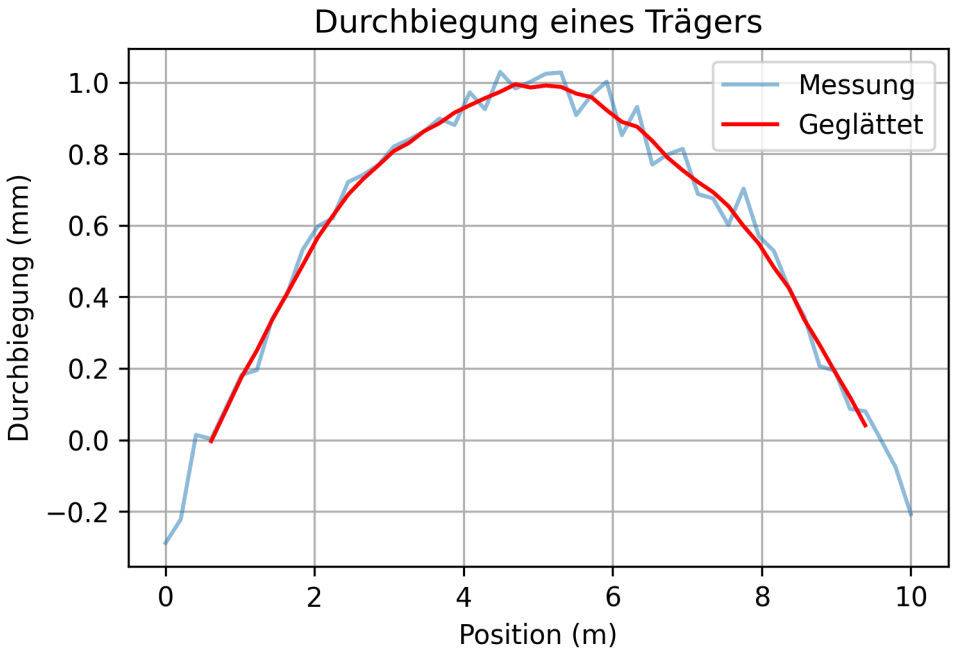
```
balken = np.genfromtxt("01-daten/balken-durchbiegung.csv", delimiter=",", max_rows= 4, dtype=

print("Der Kopf der Datei:")
print(balken)
```

```
Der Kopf der Datei:
[['Position (m)' 'Durchbiegung (mm)']
 ['0.0' '-0.28820719619532215']
 ['0.20408163265306123' '-0.22193122300286544']
 ['0.40816326530612246' '0.013978362500916705']]
```

Die Durchbiegung wird an 50 Punkten gemessen. Glätten Sie die Daten und stellen Sie die gemessenen und die geglätteten Daten gemeinsam dar.

Tipp 2: Musterlösung Balken

```
balken = np.genfromtxt("01-daten/balken-durchbiegung.csv", delimiter=",", skip_header=1)
x = balken[:, 0]
y = balken[:, 1]

window = 7
weights = np.ones(window) / window
y_smooth = np.convolve(y, weights, mode='valid')

plt.plot(x, y, label="Messung", alpha=0.5)
plt.plot(x[(window-1)//2:-(window//2)], y_smooth, label="Geglättet", color='red')
plt.title("Durchbiegung eines Trägers")
plt.xlabel("Position (m)")
plt.ylabel("Durchbiegung (mm)")
plt.legend()
plt.grid(True)

plt.show()
```



6.3 Übung: Neutronenstreuung

Die Datei '01-daten/neutronen.txt' enthält Daten eines Neutronenstreuexperiments. Der Dateikopf:

```
# Dateikopf als String einlesen
neutronen = np.genfromtxt("01-daten/neutronen.txt", delimiter="\t", max_rows= 4, dtype='str')
print(neutronen)
```

```
[['E(MeV)' 'sigma(mb)' 'Delta sigma(mb)']
 ['0.00' '10.60' ' 1.39']
 ['25.00' '16.00' ' 2.09']
 ['50.00' '45.00' ' 3.85']]
```

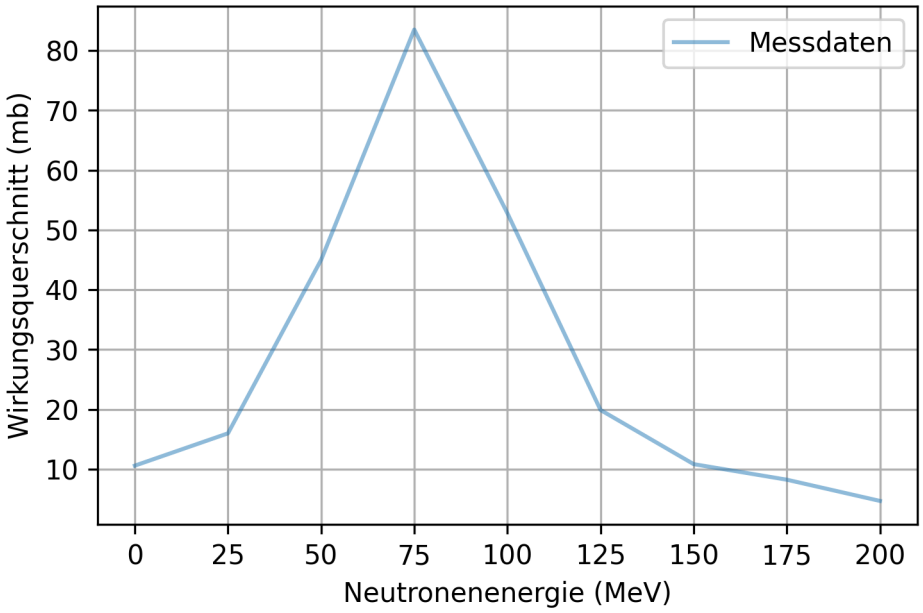
In der Spalte 'E(MeV)' ist die Energie eines Neutrons, das auf ein Ziel geschossen wird, in Megaelektronenvolt eingetragen. In der Spalte 'sigma(mb)' ist der Wirkungsquerschnitt, gemessen in Millibarn (mb), eingetragen. In der Spalte 'Delta sigma(mb)' ist die Unsicherheit des gemessenen Wirkungsquerschnitts, ebenfalls in Millibarn (mb), eingetragen.

1. Lesen Sie die Daten ein
2. Versuchen Sie die ersten zwei Spalten mit einem Polynom eines geeigneten Grades zu fitten und stellen Sie das Ergebnis graphisch dar.
3. Fitten Sie die Daten erneut, aber nutzen Sie dieses Mal Splines.
4. Stellen Sie das Ergebnis ebenfalls graphisch dar und vergleichen Sie mit den Polynomfit.
5. Was sind die Vor- und Nachteile der jeweiligen Varianten?

Tipp 3: Musterlösung Neutronenstreuung

1. Daten einlesen

```
neutronen = np.genfromtxt("01-daten/neutronen.txt", delimiter="\t", skip_header=1)
neutronenenergie = neutronen[:, 0]
wirkungsquerschnitt = neutronen[:, 1]

plt.plot(neutronenenergie, wirkungsquerschnitt, label="Messdaten", alpha=0.5)
plt.xlabel("Neutronenenergie (MeV)")
plt.ylabel("Wirkungsquerschnitt (mb)")
plt.legend()
plt.grid(True)

plt.show()
```

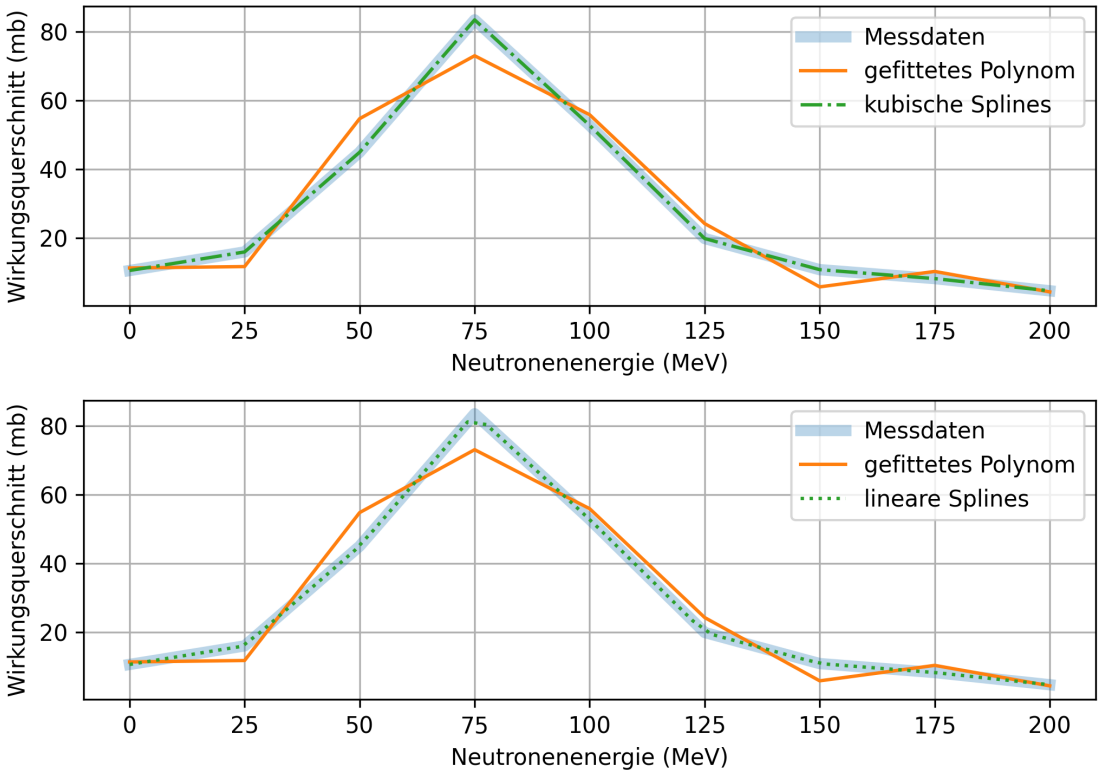


2. • 3. Polynom und Splines fitten:

```
# Polynom
polynom5 = poly.polyfit(neutronenenergie, wirkungsquerschnitt, deg = 5)
fit =  poly.polyval(x = neutronenenergie, c = polynom5)

# Splines mit scipy
splines3 = si.splrep(neutronenenergie, wirkungsquerschnitt)
y_splines3 = si.splev(neutronenenergie, splines3)

# Splines mit NumPy
x_neu = np.linspace(neutronenenergie.min(), neutronenenergie.max(), num=50)
y_splines1 = np.interp(x_neu, neutronenenergie, wirkungsquerschnitt) # linear interpolieren
```

4. Grafische Darstellung:

```
plt.figure(figsize = [7, 5])

# Vergleich mit kubischen Splines
plt.subplot(2, 1, 1)
plt.plot(neutronenenergie, wirkungsquerschnitt, linewidth = 5, alpha=0.3, label="Messdaten"
plt.plot(neutronenenergie, fit, label = "gefittetes Polynom")
plt.plot(neutronenenergie, y_splines3, linestyle = 'dashdot', label = "kubische Splines")
plt.xlabel("Neutronenenergie (MeV)")
plt.ylabel("Wirkungsquerschnitt (mb)")
plt.legend()
plt.grid(True)

# Vergleich mit linearen Splines
plt.subplot(2, 1, 2)
plt.plot(neutronenenergie, wirkungsquerschnitt, linewidth = 5, alpha=0.3, label="Messdaten"
plt.plot(neutronenenergie, fit, label = "gefittetes Polynom")

plt.plot(x_neu, y_splines1, linestyle = 'dotted', label = 'lineare Splines')
plt.xlabel("Neutronenenergie (MeV)")
plt.ylabel("Wirkungsquerschnitt (mb)")
plt.legend()
plt.grid(True)

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



5.: Die Splines bilden die Daten exakt nach. Das kann ein Vorteil sein, wenn keine Glättung der Daten gewünscht ist. Das Polynom ist dagegen nicht exakt an die Daten angepasst. Dafür können alle Datenpunkte mit einer einzigen Modellgleichung approximiert werden.

Quellen

Baitsch, Matthias (2019). *Vorlesungsskript Mathematik B (Master) Stochastik*.