m-numerik

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

2026-02-24

Inhaltsverzeichnis

Preamble

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Maik Poetzsch, und Sebastian Seipel. 2024. „Bausteine Computergestützter Datenanalyse. Methodenbaustein Numerik". https://github.com/bausteine-der-datenanalyse/m-numerik.

BibTeX-Vorlage

```
@misc{BCD-Styleguide-2024,
 title={Bausteine Computergestützter Datenanalyse. Methodenbaustein Numerik},
 author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Poetzsch,
 year={2024},
 url={https://github.com/bausteine-der-datenanalyse/m-numerik}}
```

Intro

Voraussetzungen

- Grundlagen Python
- Einbinden von zusätzlichen Paketen
- Plotten mit Matplotlib

Verwendete Pakete und Datensätze

Pakete

- NumPy
- Matplotlib

Datensätze

Bearbeitungszeit

Geschätzte Bearbeitungszeit: 2h

Lernziele

- Was ist Numerik
- Numerische Verfahren umsetzen
- Numerische Integration
- Numerische Differentiation

1 Integration

Die Bildung von Integralen findet beispielsweise bei der Bestimmung von Flächeninhalten oder von Gesamtkräften Anwendung. Formal wird das bestimmte Integral I der Funktion $f(x)$ auf dem Intervall $x \in [a, b]$ wie folgt dargestellt.

$$I = \int_a^b f(x)\ dx$$

Im Allgemeinen kann das Integral nicht analytisch gelöst werden, da die Stammfunktion $F(x)$ nicht leicht zu bestimmen ist. In solchen Fällen können numersiche Verfahren eingesetzt werden um den Integralwert zu approximieren. Die numerische Integration wird oft auch als numersiche Quadratur bezeichnet.

Dieses Kapitel bietet eine kurze Übersicht von numerischen Integrationsmethoden:

- Ober- und Untersumme
- Quadratur
- Monte-Carlo

1.0.1 Ober- und Untersumme

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

Eine der grundlegendsten Arten Integrale von Funktionen zu bestimmen sind die Ober- und Untersumme. Sie nähern den Integralwert durch eine Abschätzung nach oben bzw. unten an. Mit einer steigenden Anzahl von Stützstellen, d.h. Positionen an welchen die Funktion ausgewertet wird, konvergieren beide Abschätzungen gegen den Integralwert.

1.1 Definition

Für die Bildung der Ober- und Untersumme, werden gleichmäßig verteilte Stützstellen auf dem Intervall $[a, b]$ benötigt. Werden $n + 1$ Stützstellen gewählt, so gilt:

$$a = x_0 < x_1 < \cdots < x_n = b$$

Der Abstand der Stützstellen beträgt $x = (b - a)/(n - 1)$. Auf jedem der n Teilintervalle $[x_{i-1}, x_i]$ wird nun der maximale bzw. minimale Wert der Funktion $f(x)$ bestimmt und als O_i bzw. U_i definiert.

$$O_i = \max\left(f(x)|x \in [x_{i-1}, x_i]\right)$$
$$U_i = \min\left(f(x)|x \in [x_{i-1}, x_i]\right)$$

Die gesuchte Approximation des Integrals ist die Summe der O_i bzw. U_i mal der Breite des Teilintervalls, hier x:

$$\sum_{i=1}^{n} \Delta x U_i \lesssim I \lesssim \sum_{i=1}^{n} \Delta x O_i$$

1.2 Beispiel

Beispielhaft soll folgendes Integral bestimmt werden

$$I = \int_0^2 \sin(3x) + 2x \, dx$$

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

Als erstes werden die Stützstellen gleichmäßig im Intervall $[0, 2]$ verteilt.

```
n = 5

xi = np.linspace(0, 2, n)
yi = fkt(xi)
```

Die beiden Summen benötigen die Extremwerte der zu integrierenden Funktion in den Teilintervallen. Diese werden mit Hilfe einer Funktionsauswertung auf dem Teilintervall bestimmt. Für die nachfolgende Visualisierung hat die Menge der Summen ebenfalls `n` Elemente.

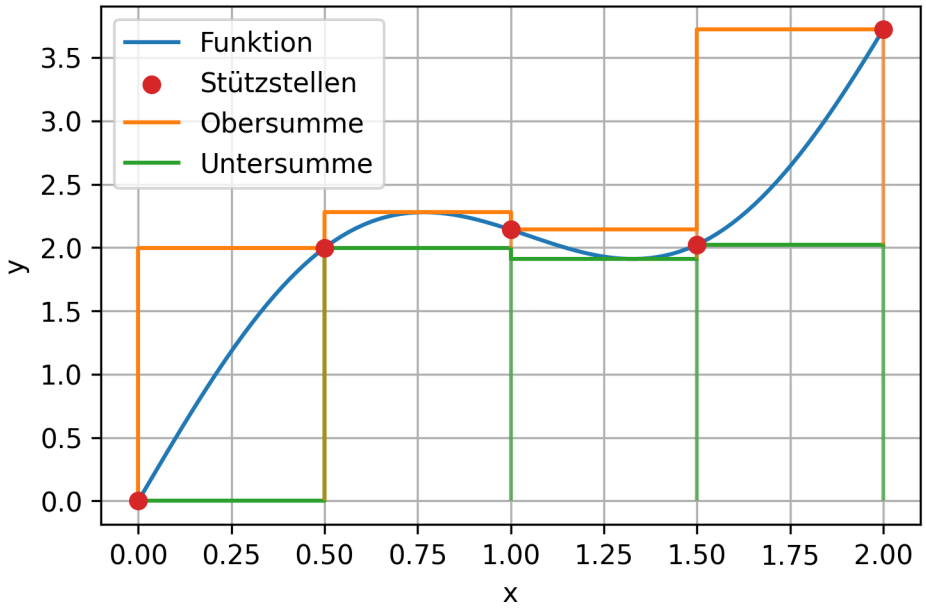
```
oben = np.zeros(n)
unten = np.zeros(n)

for i in range(len(oben)-1):
    cx = np.linspace(xi[i], xi[i+1], 50)
    cy = fkt(cx)
    oben[i+1] = np.max(cy)
    unten[i+1] = np.min(cy)
```

Die ersten Elemente der beiden Summenlisten werden auf die ersten Funktionswerte gesetzt, dies dient nur der folgenden Darstellung.

```
oben[0] = yi[0]
unten[0] = yi[0]
```

Visualisierung der einzelnen Funktionen.

```
plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3', zorder=3)
plt.plot(xi, oben, drawstyle='steps-pre', label='Obersumme')
plt.plot(xi, unten, drawstyle='steps-pre', label='Untersumme')

plt.vlines(xi, ymin=unten, ymax=oben, color='C1', alpha=0.6)
plt.vlines(xi, ymin=0, ymax=unten, color='C2', alpha=0.6)

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

plt.grid()
plt.legend();
```



Das obige Verfahren kann nun in einer Funktion zusammengefasst werden, welche die Summen der beiden Folgen zurückgibt.

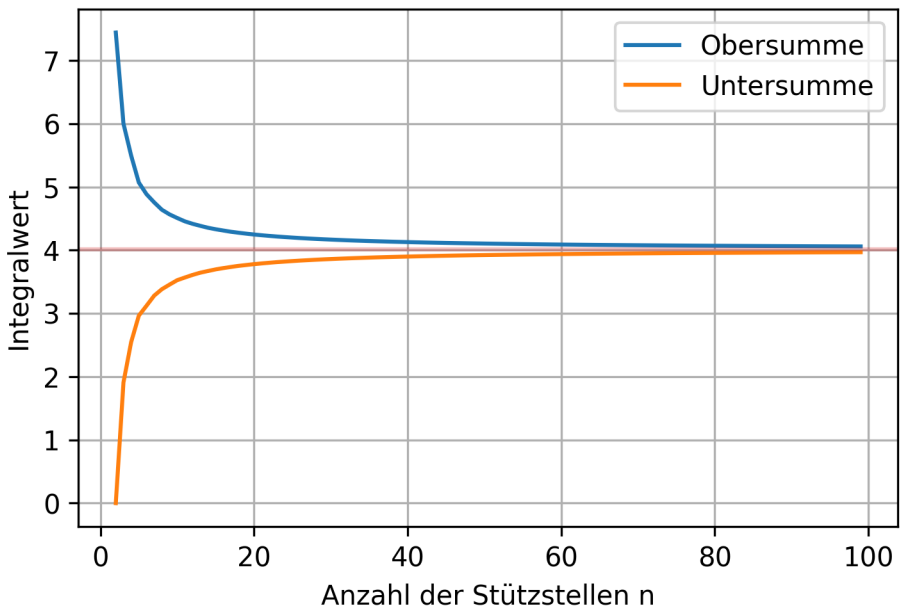
```
def ou_summe(n, a=0, b=2):
    xi = np.linspace(a, b, n)
    yi = fkt(xi)
    dx = xi[1] - xi[0]

    sum_oben = 0
    sum_unten = 0

    for i in range(n-1):
        cx = np.linspace(xi[i], xi[i+1], 50)
        cy = fkt(cx)
        oben = np.max(cy)
        unten = np.min(cy)
        sum_oben += dx * oben
        sum_unten += dx * unten

    return sum_oben, sum_unten
```

Für eine systematische Untersuchung des Konvergenzverhaltens, wird die Integrationsfunktion für verschiedene Anzahlen von Stützstellen aufgerufen.

```
n_max = 100
ns = np.arange(2, n_max, 1, dtype=int)
os = np.zeros(len(ns))
us = np.zeros(len(ns))


for i, n in enumerate(ns):
    o, u = ou_summe(n)
    os[i] = o
    us[i] = u
```

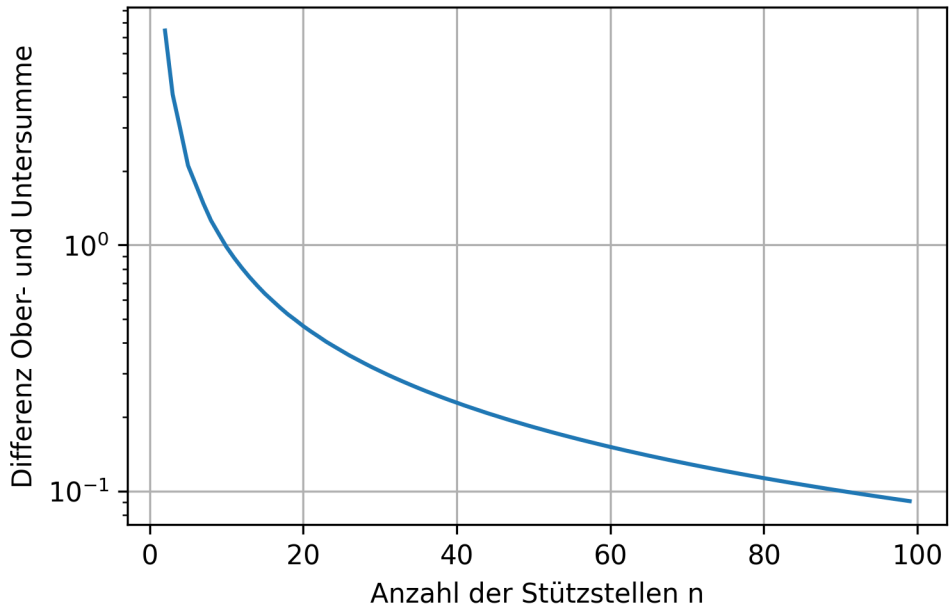
Die graphische Darstellung der beiden Summen zeigt eine kontinuierliche Annäherung dieser.

```
plt.plot(ns, os, label='Obersumme')
plt.plot(ns, us, label='Untersumme')

plt.axhline(y=I_exakt, color='C3', alpha=0.3)

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Integralwert')

plt.grid()
plt.legend();
```



Dies wird insbesondere deutlich, wenn die Differenz der beiden Summen aufgetragen wird. Mit einer logarithmischen Darstellung kann die kontinuierliche Annäherung auch quantitativ abgelesen werden.

```
plt.plot(ns, os-us)

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz Ober- und Untersumme')

# plt.xscale('log')
plt.yscale('log')

plt.grid();
```



1.3 Interpolation

Bei der Bildung der Ober- und Untersumme wurde die zu integrierende Funktion durch einen konstanten Wert in den Teilintervallen zwischen den Stützstellen angenähert. Eine genauere Berechnung des Integrals kann durch eine bessere Interpolation erfolgen. Dazu eignen sich Polynome, da diese leicht zu Integrieren sind.

1.3.1 Trapezregel

Die Trapezregel beruht auf der Annäherung der zu integrierenden Funktion durch Geraden, d.h. Polynome vom Grad 1, auf den Teilintervallen. Die Approximation des Integralwertes ergibt sich entsprechend aus den Flächeninhalten der so entstandenen Trapeze.

Wie im vorhergehenden Kapitel wird das Verfahren anhand folgender Funktion demonstriert

$$I = \int_0^2 \sin(3x) + 2x \, dx$$

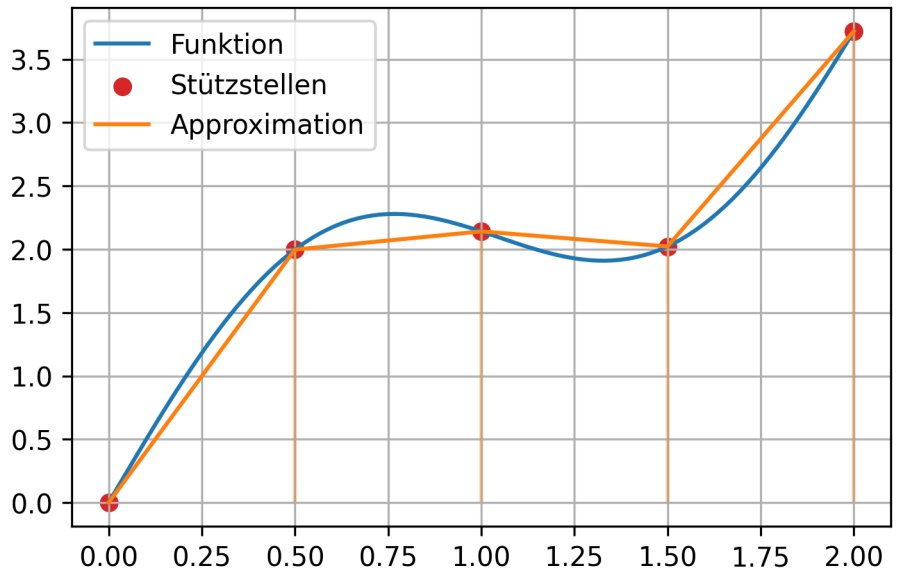
```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

Bildung der Stützpunkte:

```
n = 5

xi = np.linspace(0, 2, n)
yi = fkt(xi)
```

Zunächst erfolgt noch die Visualisierung des Verfahrens.

```
plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3')
plt.plot(xi, yi, label='Approximation', c='C1')

plt.vlines(xi, ymin=0, ymax=yi, color='C1', alpha=0.3)

plt.grid()
plt.legend();
```



Die Integration selbst kann mittels der Funktion `scipy.integrate.trapezoid` ausgeführt werden.

```
res = scipy.integrate.trapezoid(yi, xi)
print(f"Integralwert mit {n} Stützstellen: {res:.4f}")
```

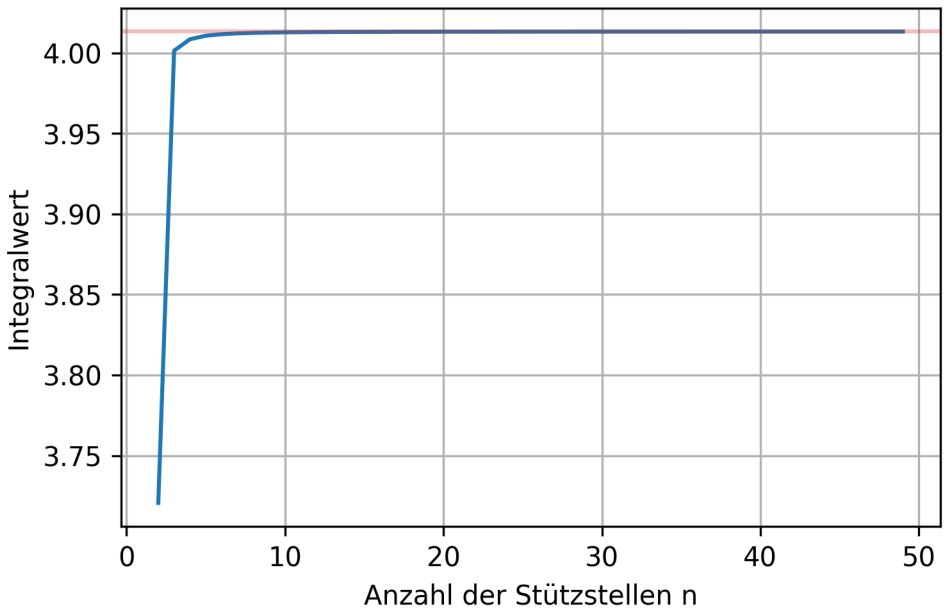
```
Integralwert mit 5 Stützstellen: 4.0107
```

Der so ermittelte Wert nähert sich dem exakten Wert mit zunehmender Anzahl der Stützstellen.

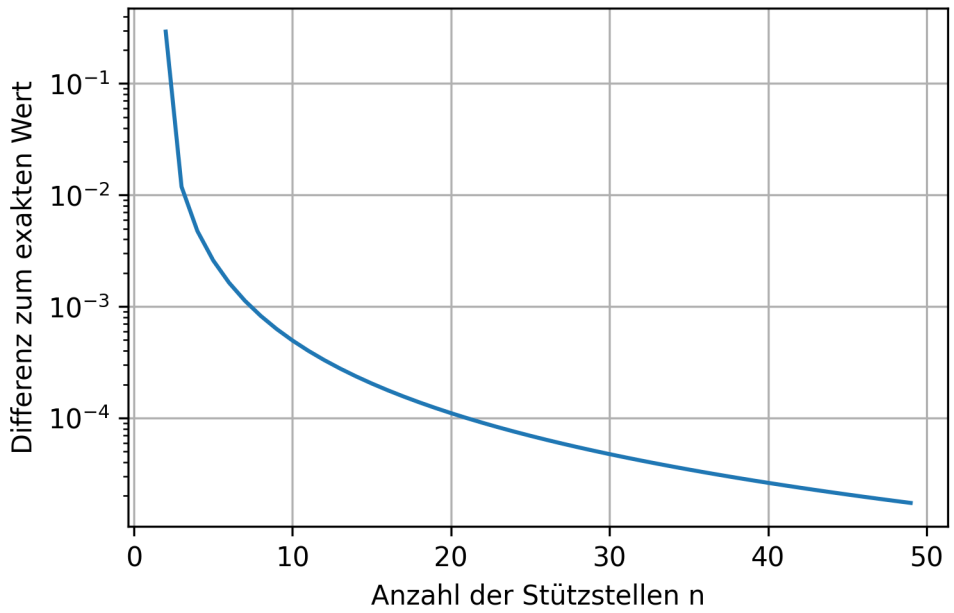
```
n_max = 50
ns = np.arange(2, n_max, 1, dtype=int)
tr = np.zeros(len(ns))

for i, n in enumerate(ns):
    xi = np.linspace(0, 2, n)
    yi = fkt(xi)
    tr[i] = scipy.integrate.trapezoid(yi, xi)
```

```
plt.plot(ns, tr)
plt.axhline(y=I_exakt, color='C3', alpha=0.3)

plt.xlabel('Anzahl der Stützstellen n')
```

```
plt.ylabel('Integralwert')

plt.grid();
```




```
plt.plot(ns, np.abs(tr-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
plt.yscale('log')

plt.grid();
```



1.3.2 Simpsonregel

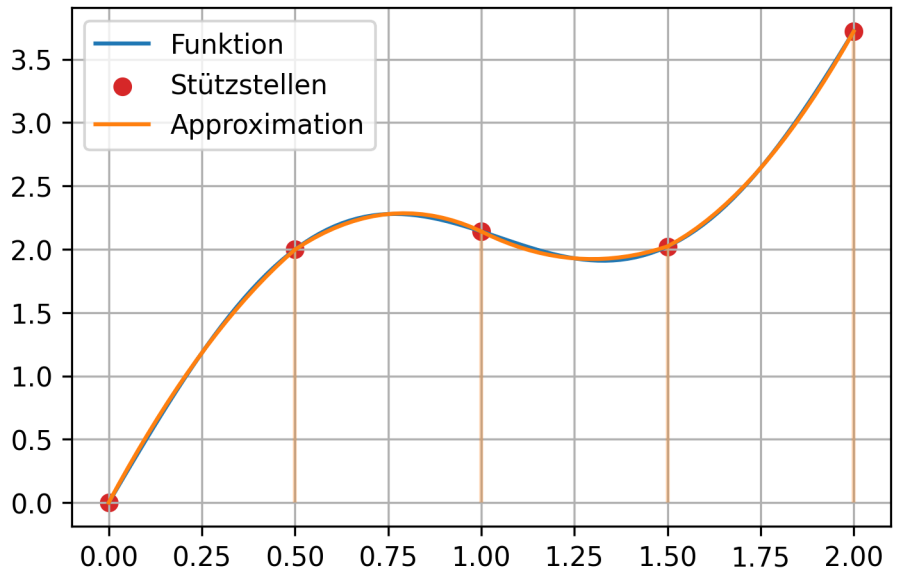
Die Verwendung eines Polynoms vom zweiten Grad führt zur Simpsonregel. Hierzu wird die Funktion an einem Zwischenwert, mittig im Teilintervall, ausgewertet und zusammen mit den Werten an den Stützstellen zur Bestimmung der Polynomkoeffizienten verwendet.

Anhand des obigen Beispiels wird die Simpsonregel visuell demonstriert.

```
n = 5

xi = np.linspace(0, 2, n)
yi = fkt(xi)
```

```
plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3')

# Bestimmung und Plotten der Polynome
for i in range(n-1):
    dx = xi[i+1] - xi[i]
    cx = (xi[i] + xi[i+1]) / 2
    cy = fkt(cx)

    P = np.polyfit([xi[i], cx, xi[i+1]], [yi[i], cy, yi[i+1]], 2)
```

```
    Px = np.linspace(xi[i], xi[i+1], 20)
    Py = np.polyval(P, Px)

    label=None
    if i==0:
        label='Approximation'

    plt.plot(Px, Py, color='C1', label=label)

plt.vlines(xi, ymin=0, ymax=yi, color='C1', alpha=0.3)

plt.grid()
plt.legend();
```

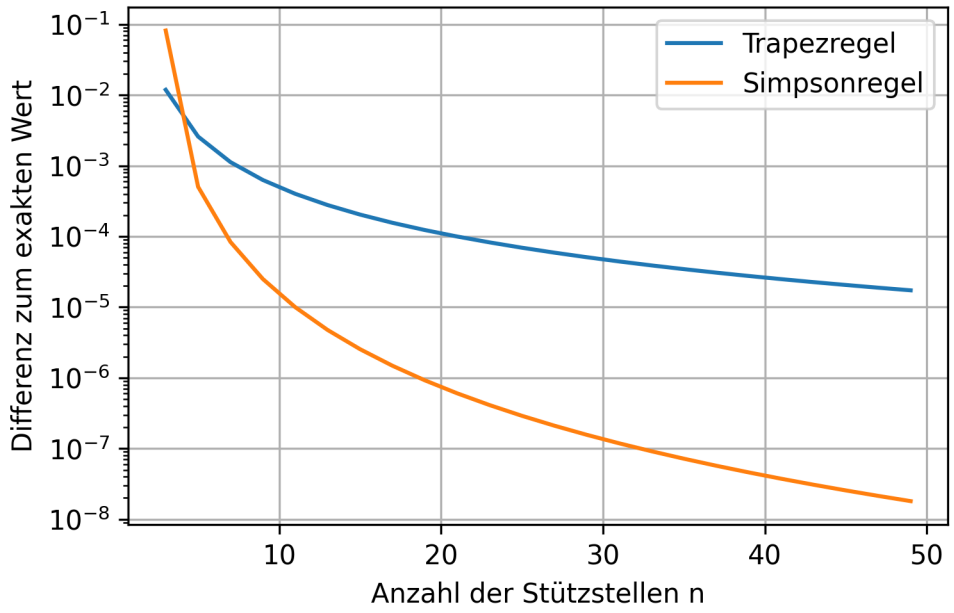


Die Simpsonregel ist bereits in der Funktion `scipy.integrate.simpson` implementiert. Im Folgenden wird nur die Differenz zur Trapezregel demonstriert.

```
n_max = 50
ns = np.arange(3, n_max, 2, dtype=int)
si = np.zeros(len(ns))
tr = np.zeros(len(ns))

for i, n in enumerate(ns):
    xi = np.linspace(0, 2, n)
```

```
    yi = fkt(xi)
    si[i] = scipy.integrate.simpson(yi, xi)
    tr[i] = scipy.integrate.trapezoid(yi, xi)
```

```
plt.plot(ns, np.abs(tr-I_exakt), label='Trapezregel')
plt.plot(ns, np.abs(si-I_exakt), label='Simpsonregel')

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();
```



1.4 Monte-Carlo

Ein ganz anderer Ansatz zur Integration wird mit dem Monte-Carlo-Ansatz verfolgt. Hierbei werden Zufallspunkte x_i innerhalb der gesuchten Integralbereichs generiert. Der Mittelwert der dazugehörigen Summe der Funktionswerte $f(x_i)$ nähert das Integral an. Insbesondere für

eine kleine Anzahl von Zufallswerten kann das Ergebnis deutlich vom exakten Wert abweichen. Der Vorteil des Verfahrens wird bei hochdimensionalen Integralen deutlich.

Für $n \gg 1$ zufällige Stützstellen $x_i \in [a, b]$ gilt folgende Näherung

$$I = \int_a^b f(x)\, dx \approx \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$$

Für das Beispiel aus den vorhergehenden Kapiteln gilt

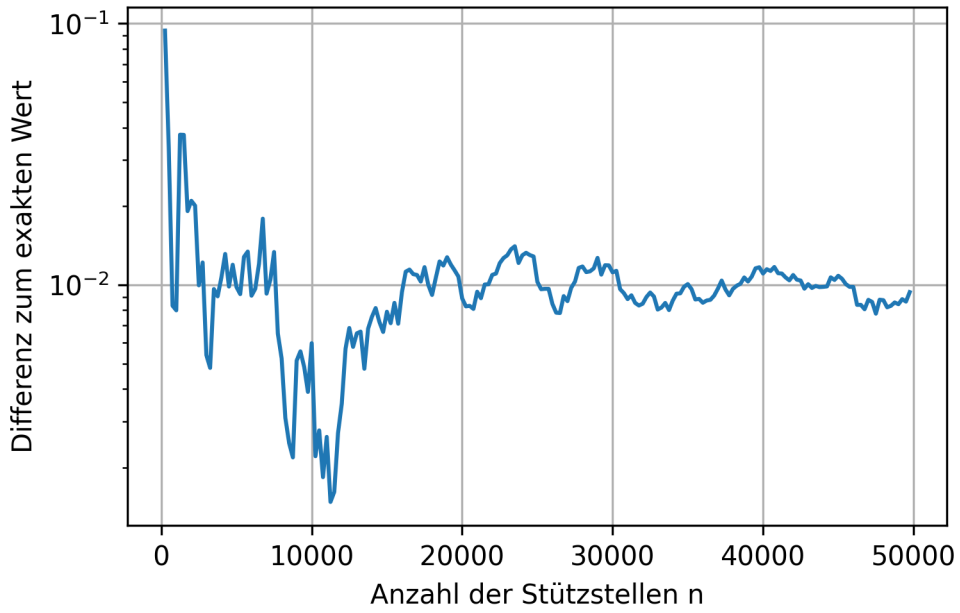
```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

```
n = 2000
xi = np.random.random(n) * 2
yi = fkt(xi)
I = 2 * 1/n * np.sum(yi)
print(f"Integralwert für {n} Stützstellen: {I:.4f}")
```

```
Integralwert für 2000 Stützstellen: 3.9620
```

```
n_max = 50000
dn = 250
ns = np.arange(dn, n_max, dn, dtype=int)
mc = np.zeros(len(ns))

xi = np.zeros(n_max)

for i, n in enumerate(ns):
    xi[n-dn:n] = np.random.random(dn) * 2
    yi = fkt(xi[:n])
    mc[i] = 2 * 1/n * np.sum(yi)
```

```
plt.plot(ns, np.abs(mc-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
plt.yscale('log')

plt.grid();
```



Alternativ kann auch das Flächenverhältnis zwischen der zu integrierenden Funktion und einer Referenzfläche A_r gebildet werden. Hierzu werden n Zufallszahlenpaare (x_i, y_i) generiert und gezählt wieviele davon in der gesuchten Fläche liegen. Die Annahme ist, dass sich beide Verhältnisse für große n annähern. Im einfachsten Fall, wenn $f(x) \geq 0$, gilt folgende Abschätzung

$$I \approx \frac{A_r \cdot |\{y_i \mid y_i < f(x_i)\}|}{n}$$

Im obigen Beispiel kann die Fläche $[0, 2] \times [0, 4] = 8$ als Referenzfläche verwendet werden.

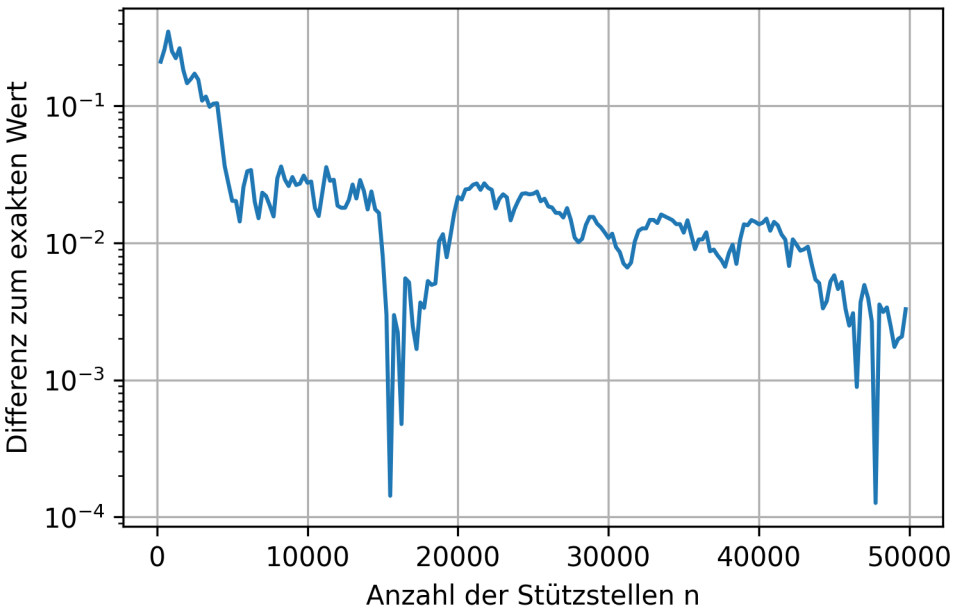
```
n = 2000
xi = np.random.random(n) * 2
```

```
yi = np.random.random(n) * 4

z = np.sum(yi < fkt(xi))

I = z / n * 8
print(f"Integralwert für {n} Stützstellen: {I}")
```

```
Integralwert für 2000 Stützstellen: 4.044
```

```
n_max = 50000
dn = 250
ns = np.arange(dn, n_max, dn, dtype=int)
mc = np.zeros(len(ns))

xi = np.zeros(n_max)
yi = np.zeros(n_max)

for i, n in enumerate(ns):
    xi[n-dn:n] = np.random.random(dn) * 2
    yi[n-dn:n] = np.random.random(dn) * 4
    z = np.sum(yi[:n] < fkt(xi[:n]))
    mc[i] = z / n * 8
```

```
plt.plot(ns, np.abs(mc-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
plt.yscale('log')

plt.grid();
```


Differenz zum exakten Wert
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
10000
20000
30000
40000
50000
Anzahl der Stützstellen n

2 Differentiation

Die numerische Bestimmung von Ableitungen wird hier anhand von zwei Ansätzen demonstriert. Zum Einen als Differenzenquotienten und zum Anderen über das Polynomfitting. Angewendet werden diese Verfahren z.B. beim Suchen von Extrema in Experimental- oder Simulationsdaten, beim Lösen von Differentialgleichungen oder bei Optimierungsverfahren.

Obwohl die analytische Bildung einer Ableitung oft viel einfacher ist als die Integration, ist dies in den oben genannten Fällen nicht direkt möglich. Gesucht ist hierbei immer die Ableitung $f'(x)$ einer Funktion $f(x)$ oder einer diskreten Punktmenge (x_i, y_i) an einer bestimmten Stelle $x = x_0$ oder auf einem Intervall.

Die Grundidee bei den hier vorgestellten Differenzenquotienten bzw. Differenzenformeln ist die Annäherung der abzuleitenden Funktion mit einer Taylor-Entwicklung an mehreren Stellen. Damit kann nach der gesuchte Ableitung an der entsprechenden Entwicklungsstelle aufgelöst werden.

2.1 Taylor-Entwicklung

Mittels der Taylor-Entwicklung kann jede beliebig oft stetig differenzierbare Funktion $f(x)$ um einem Entwicklungspunkt x_0 beliebig genau angenähert werden. Die funktionale Abhängigkeit bezieht sich nun auf die Variable h, welche nur in direkter Umgebung um x_0 betrachtet wird. Die Taylor-Entwicklung lautet:

$$f(x_0 + h) = \sum_{i=0}^{\infty} \frac{1}{i!} f^{(i)}(x_0) \cdot h^i$$

$$= f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \frac{1}{6} f'''(x_0) \cdot h^3 + \cdots$$

Diese Entwicklung kann auch nur bis zu einer vorgegebenen Ordnung betrachtet werden. So nimmt die Entwicklung bis zur Ordnung $\mathcal{O}(h^3)$ folgende Form an:

$$f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$

Hierbei deutet das Landau-Symbol $\mathcal{O}$ die Ordnung an, welche die vernachlässigten Terme, hier ab h^3, als Approximationsfehler zusammenfasst. Die Ordnung gibt an wie schnell bzw. mit welchem funktionalem Zusammenhang der Approximationsfehler gegen Null läuft für $h \to 0$.

Eine graphische Darstellung der ersten Elemente der Reihe verdeutlichen nochmals die Grundidee. Das folgende Beispiel entwickelt die Funktion

$$f(x) = \sin(3x) + 2x$$

am Punkt $x_0 = 0.85$.

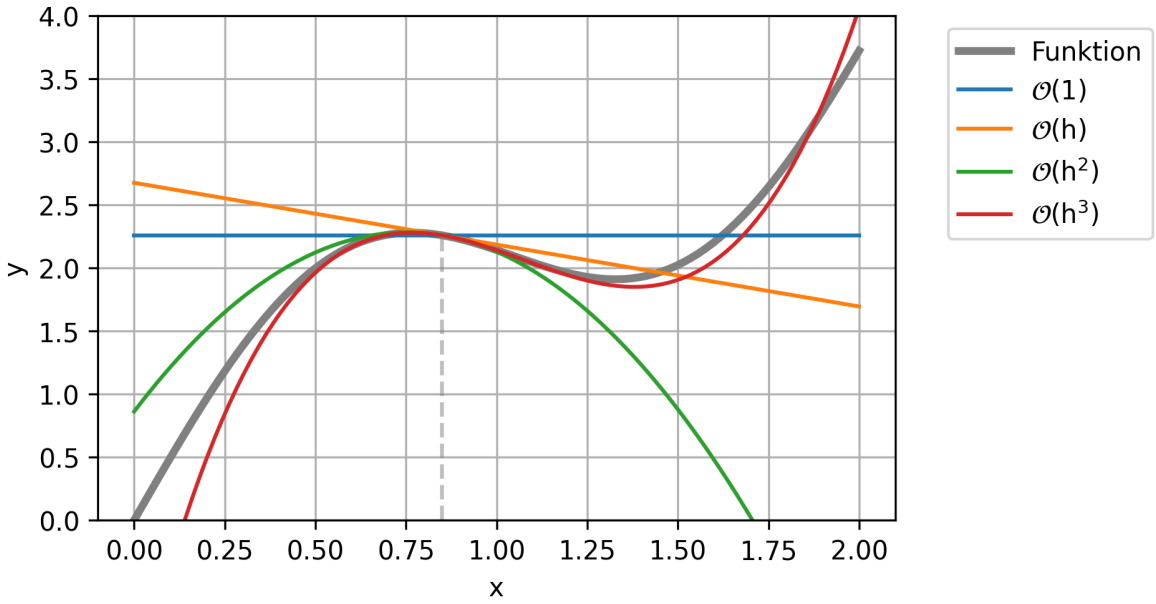
```
def fkt(x, p=0):
    if p==0:
        return np.sin(3*x) + 2*x
    if p==1:
        return 3*np.cos(3*x) + 2
    if p==2:
        return -9*np.sin(3*x)
    if p==3:
        return -27*np.cos(3*x)
    return None

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x, p=0)
```

```
x0 = 0.85

# Taylor-Elemente
te = []
te.append(0*(x-x0) + fkt(x0, p=0))
te.append((x-x0) * fkt(x0, p=1))
te.append((x-x0)**2 * fkt(x0, p=2) * 1/2)
te.append((x-x0)**3 * fkt(x0, p=3) * 1/6)
```

```
plt.plot(x, y, color='Grey', lw=3, label="Funktion")
plt.plot(x, te[0], label="$\mathsf{\mathcal{O}(1)}$")
plt.plot(x, te[0] + te[1], label="$\mathsf{\mathcal{O}(h)}$")
plt.plot(x, te[0] + te[1] + te[2], label="$\mathsf{\mathcal{O}(h^2)}$")
plt.plot(x, te[0] + te[1] + te[2] + te[3], label="$\mathsf{\mathcal{O}(h^3)}$")

plt.vlines(x0, ymin=0, ymax=fkt(x0), color='Grey', ls='--', alpha=0.5)
```

```
plt.ylim([0,4])

plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.grid()
plt.xlabel('x')
plt.ylabel('y');
```

```
<>:2: SyntaxWarning: invalid escape sequence '\m'
<>:3: SyntaxWarning: invalid escape sequence '\m'
<>:4: SyntaxWarning: invalid escape sequence '\m'
<>:5: SyntaxWarning: invalid escape sequence '\m'
<>:2: SyntaxWarning: invalid escape sequence '\m'
<>:3: SyntaxWarning: invalid escape sequence '\m'
<>:4: SyntaxWarning: invalid escape sequence '\m'
<>:5: SyntaxWarning: invalid escape sequence '\m'
/var/folders/p_/ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_60998/2784086881.py:2: SyntaxWarni
  plt.plot(x, te[0], label="$\mathsf{\mathcal{O}(1)}$")
/var/folders/p_/ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_60998/2784086881.py:3: SyntaxWarni
  plt.plot(x, te[0] + te[1], label="$\mathsf{\mathcal{O}(h)}$")
/var/folders/p_/ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_60998/2784086881.py:4: SyntaxWarni
  plt.plot(x, te[0] + te[1] + te[2], label="$\mathsf{\mathcal{O}(h^2)}$")
/var/folders/p_/ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_60998/2784086881.py:5: SyntaxWarni
  plt.plot(x, te[0] + te[1] + te[2] + te[3], label="$\mathsf{\mathcal{O}(h^3)}$")
```



2.2 Differenzenformeln

In diesem Abschnitt werden Berechnungsformeln für die Approximation von Ableitungen durch Bildung von Funktionswertdifferenzen vorgestellt. Diese beruhen alle auf der Taylor-Entwicklung und können für beliebige Ableitungen und Ordnungen formuliert werden. Die einfachsten davon werden hier vorgestellt.

2.2.1 Erste Ableitung erster Ordnung

Die einfachste Differenzenformel ergibt sich aus der Taylor-Reihe bis $\mathcal{O}(h^2)$. Hier kann die Reihe direkt nach der gesuchten Ableitung an der Stelle x_0 umgeformt werden.

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \mathcal{O}(h^2)$$
$$\Rightarrow \quad f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h)$$

Dies ist die vorwärtsgerichtete Differenzformel erster Ordnung für die erste Ableitung. Erste Ordnung bedeutet hierbei, dass im Grenzwert $h \to 0$ der Approximationsfehler linear mit der Schrittweite abnimmt.

Nach dieser Formel muss die abzuleitende Funktion an zwei Stellen $f(x_0)$ und $f(x_0 + h)$ ausgewertet werden, um die Ableitung numerisch zu bestimmen. Im Grenzwert für eine beliebig kleine Schrittweite, d.h. $h \to 0$, nähert sich dieser Quotient der exakten Ableitung an der Stelle x_0 an.

Das folgende Beispiel demonstriert die Näherung anhand der Funktion

$$f(x) = \sin(3x) + 2x$$

Die Ableitung wird an der Stelle $x_0 = 0.85$ angenähert.

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung bei x=0.85
fp_exakt = 3*np.cos(3*0.85) + 2
```

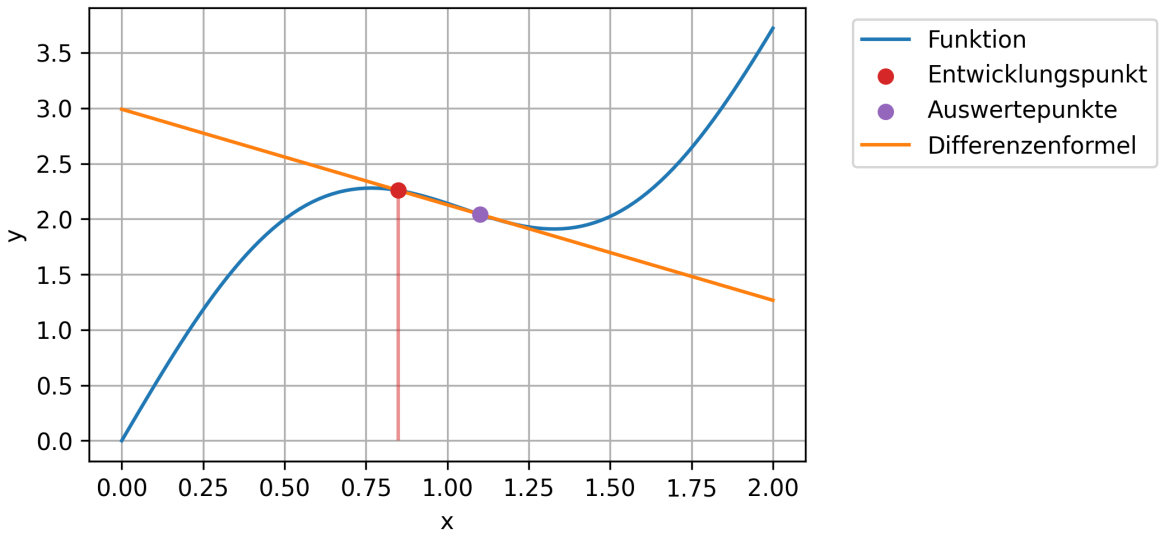
```
# Entwicklungspunkt und Schrittweite
h = 0.25
x0 = 0.85

# Auswertung an den beiden Stellen
f0 = fkt(x0)
fh = fkt(x0 + h)

# Bestimmung der Ableitungsnäherung
fp = (fh - f0) / h
```

```
print(f"Die numerische Näherung der Ableitung an der Stelle {x0:.2f}:")
print(f"Näherung mit Schrittweite {h:.2f}: {fp:.2f}")
print(f"Exakter Wert: {fp_exakt:.2f}")
```

```
Die numerische Näherung der Ableitung an der Stelle 0.85:
Näherung mit Schrittweite 0.25: -0.86
Exakter Wert: -0.49
```

Die Methode kann auch graphisch dargestellt werden. Die gesuchte Steigung ist die Steigung der eingezeichneten Geraden.

```
plt.plot(x, y, label="Funktion")
plt.scatter([x0], [f0], color='C3', label='Entwicklungspunkt', zorder=3)
plt.scatter([x0+h], [fh], color='C4', label='Auswertepunkte', zorder=3)

plt.vlines(x0, ymin=0, ymax=f0, color='C3', alpha=0.5)

plt.plot(x, f0 + fp*(x-x0), label='Differenzenformel')

plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left');
```



2.2.2 Erste Ableitung zweiter Ordnung

Mit dem gleichen Ansatz kann auch eine Differenzenformel zweiter Ordnung gefunden werden. Dazu wird die Funktion an den Stellen $x_0 - h$ und $x_0 + h$ mit der Taylor-Reihe bis zur Ordnung $\mathcal{O}(h^3)$ approximiert.

$$f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$
$$f(x_0 - h) = f(x_0) - f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$

Die Differenz dieser beiden Gleichungen führt zu

$$f(x_0 + h) - f(x_0 - h) = 2f'(x_0) \cdot h + \mathcal{O}(h^3)$$

Und die Umformung nach der gesuchten Ableitung an der Stelle x_0 ergibt

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + \mathcal{O}(h^2)$$

Dies ist die zentrale Differenzenformel für die erste Ableitung zweiter Ordnung. Wie bei der vorwärtsgerichteten Formel muss hier die Funktion an zwei Stellen ausgewertet werden, jedoch nicht mehr am Entwicklungspunkt selbst. Durch diese Symmetrie bzgl. des Entwicklungspunkts ergibt sich ein besseres, hier quadratisches, Konvergenzverhalten.

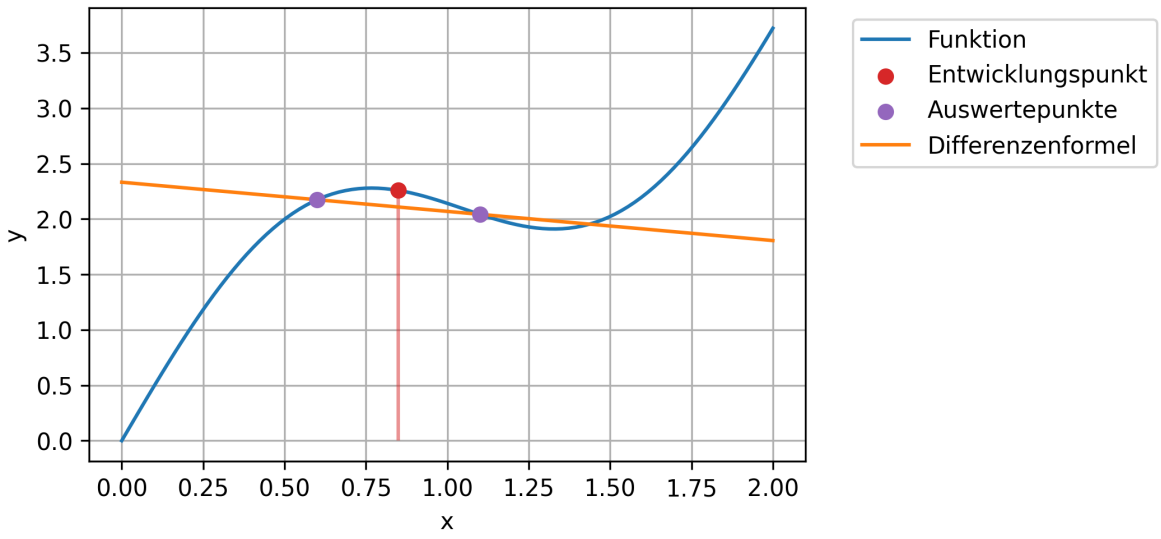
```
# Auswertung an den beiden Stellen
fnh = fkt(x0 - h)
fph = fkt(x0 + h)

# Bestimmung der Ableitungsnäherung
fp = (fph - fnh) / (2*h)
```

```
print(f"Die numerische Näherung der Ableitung an der Stelle {x0:.2f}:")
print(f"Näherung mit Schrittweite {h:.2f}: {fp:.2f}")
print(f"Exakter Wert: {fp_exakt:.2f}")
```

```
Die numerische Näherung der Ableitung an der Stelle 0.85:
Näherung mit Schrittweite 0.25: -0.26
Exakter Wert: -0.49
```

Die Methode kann auch graphisch dargestellt werden. Die gesuchte Steigung ist die Steigung der eingezeichneten Geraden.

```
plt.plot(x, y, label="Funktion")
plt.scatter([x0], [f0], color='C3', label='Entwicklungspunkt', zorder=3)
plt.scatter([x0-h, x0+h], [fnh, fph], color='C4', label='Auswertepunkte', zorder=3)

plt.vlines(x0, ymin=0, ymax=f0, color='C3', alpha=0.5)

plt.plot(x, fnh + fp*(x-x0+h), label='Differenzenformel')

plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left');
```



2.3 Zweite Ableitung zweiter Ordnung

Mit dem gleichen Schema wie oben, kann auch die Differenzenformel für die zweite Ableitung bestimmt werden. Diese lautet

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + \mathcal{O}(h^2)$$

2.4 Fehlerbetrachtung

In diesem Abschnitt werden die Approximationsfehler, d.h. Fehler aus der Differenzenformeln, und Rundungsfehler, d.h. Fehler durch die endliche Genauigkeit der digitalen Darstellung von Zahlen, betrachtet.

2.4.1 Approximationsfehler

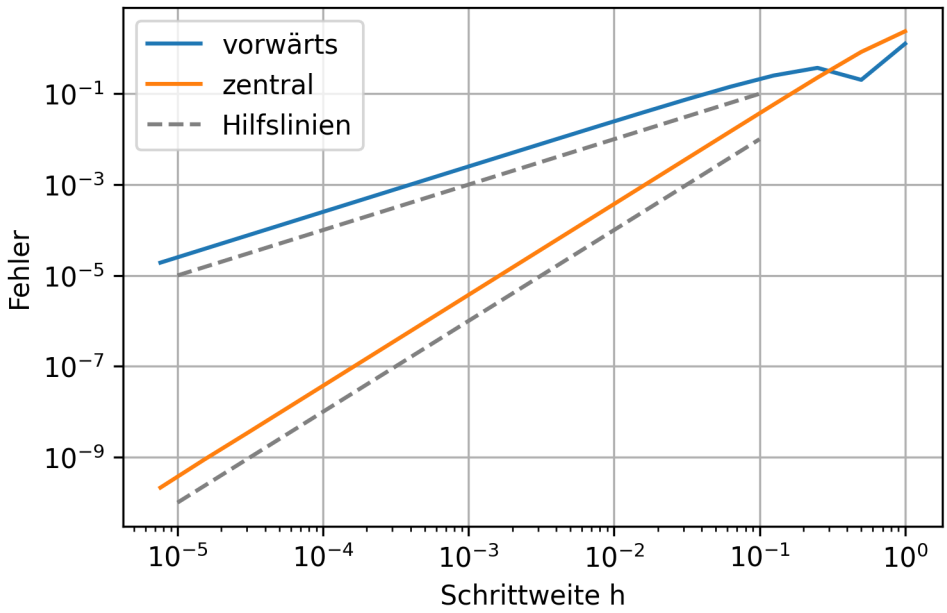
Die Ordnung des Verfahrens kann durch die Betrachtung des Fehlers, hier zum bekannten exakten Wert, bestimmt werden. Dazu wird die Schrittweite kontinuierlich verkleinert.

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
```

```
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung bei x=1
fp_exakt = 3*np.cos(3*0.85) + 2
```

```
x0 = 0.85

hs = []
fpfs = []
fpcs = []


h0 = 1
for i in range(18):
    h = h0 / 2**i

    f0 = fkt(x0)
    fnh = fkt(x0 - h)
    fph = fkt(x0 + h)

    fpf = (fph - f0) / h
    fpc = (fph - fnh) / (2*h)

    hs.append(h)
    fpfs.append(fpf)
    fpcs.append(fpc)
```

```
plt.plot(hs, np.abs(fpfs - fp_exakt), label='vorwärts')
plt.plot(hs, np.abs(fpcs - fp_exakt), label='zentral')

plt.plot([1e-5, 1e-1], [1e-5, 1e-1], '--', color='grey', label='Hilfslinien')
plt.plot([1e-5, 1e-1], [1e-10, 1e-2], '--', color='grey')

plt.xlabel('Schrittweite h')
plt.ylabel('Fehler')

plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();
```



In der logiarithmischen Darstellung beider Achsen werden Potenzfunktionen zu Graden mit dem Potenzgrad als Steigung. Das bedeutet, dass der Fehler im obigen Plot sich wie eine Potenzfunktion mit dem Grad eins bzw. zwei verhält. Die eingezeichneten Hilfslinien haben eine Steigung von eins bzw. zwei. Dies entspricht auch der Ordnung $\mathcal{O}(h)$ bzw. $\mathcal{O}(h^2)$ aus der Differenzenformel.

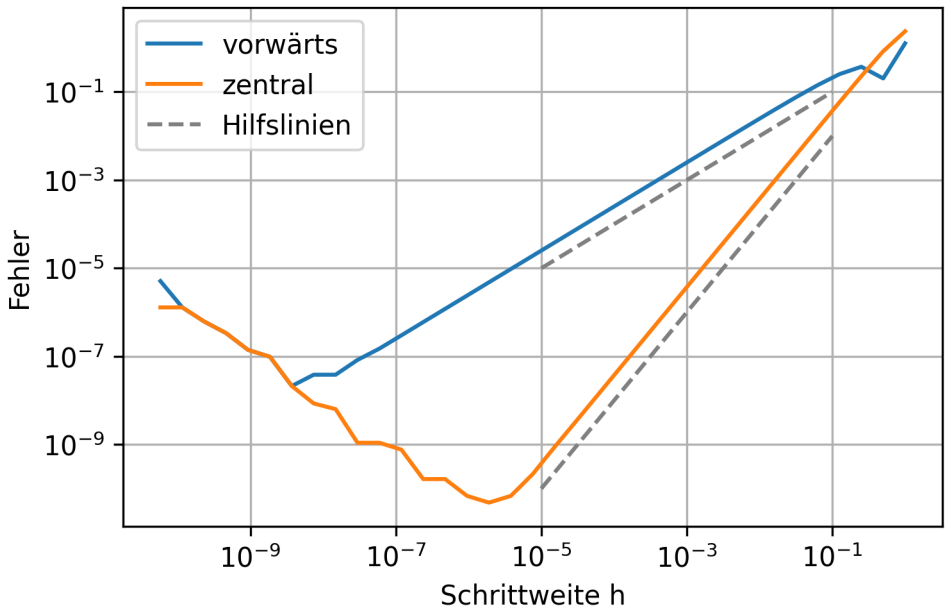
2.4.2 Rundungsfehler

Wird nun die Schrittweiter noch weiter verkleinert, wirkt sich die Genauigkeit der Darstellung von Zahlen bzw. Rundungsfehler auf die Approximation aus.

```
x0 = 0.85

hs = []
fpfs = []
fpcs = []


h0 = 1
for i in range(35):
    h = h0 / 2**i

    f0 = fkt(x0)
```

```
    fnh = fkt(x0 - h)
    fph = fkt(x0 + h)

    fpf = (fph - f0) / h
    fpc = (fph - fnh) / (2*h)

    hs.append(h)
    fpfs.append(fpf)
    fpcs.append(fpc)
```

```
plt.plot(hs, np.abs(fpfs - fp_exakt), label='vorwärts')
plt.plot(hs, np.abs(fpcs - fp_exakt), label='zentral')

plt.plot([1e-5, 1e-1], [1e-5, 1e-1], '--', color='grey', label='Hilfslinien')
plt.plot([1e-5, 1e-1], [1e-10, 1e-2], '--', color='grey')

plt.xlabel('Schrittweite h')
plt.ylabel('Fehler')

plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();
```



Wie bereits vorgestellt, können 64-Bit-Zahlen nur mit einer Genauigkeit von etwa $\approx 10^{-16}$ dargestellt werden. Das bedeutet, dass z.B. die Differenz von zwei Zahlen nicht genauer als berechnet werden kann. Dies ist der sogenannte Rundungsfehler.

Im konkreten Fall der Vorwärtsdifferenzenformel bedeutet dies:

$$
\begin{aligned}
f'(x_0) &= \frac{f(x_0+h)-f(x_0)}{h} + \mathcal{O}(h) \\
&\overset{\text{Rundungsfehler}}{\Rightarrow} \quad \frac{f(x_0+h)-f(x_0)+\mathcal{O}(\ )}{h} + \mathcal{O}(h) \\
&= \frac{f(x_0+h)-f(x_0)}{h} + \mathcal{O}\left(\frac{}{h}\right) + \mathcal{O}(h)
\end{aligned}
$$

Damit macht eine Verkleinerung von h nur Sinn, solange der Rundungsfehler klein gegenüber h ist. Genauer:

$$
\begin{aligned}
\frac{}{h} &\leq h \\
\Rightarrow \quad h &\geq \sqrt{}
\end{aligned}
$$

Mit $\approx 10^{-16}$ ist für diese Differenzenformel ein h nur bis etwa 10^{-8} angemessen.