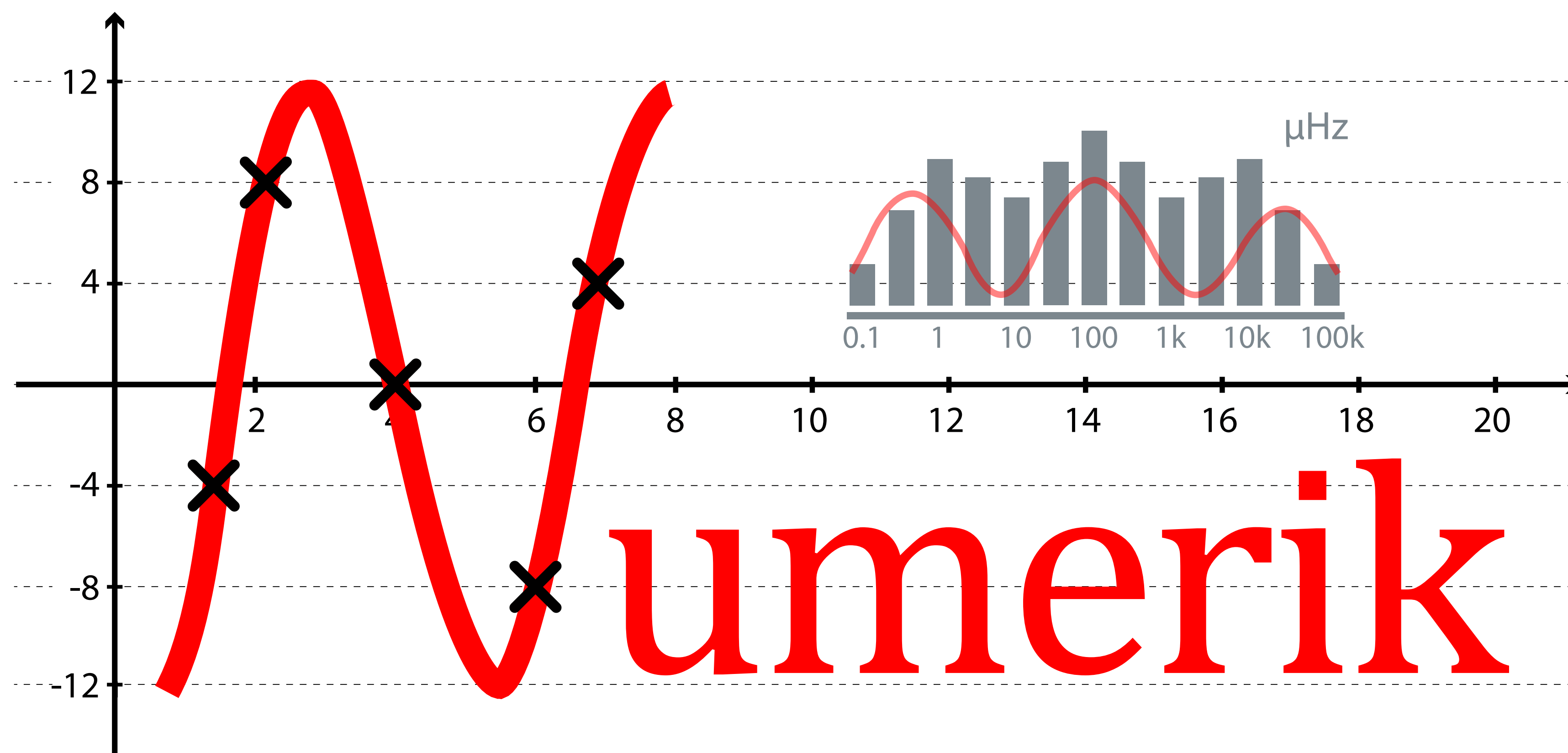




DHBW

Duale Hochschule
Baden-Württemberg
Ravensburg



umerik

Vorwort

Dieser Foliensatz wird durch das Zentrum für Angewandte Ökonomik (kurz ZAÖ) der DHBW Ravensburg bereitgestellt.

Autoren: Prof. Dr. Daniel Blochinger
Illustration: Prof. Dr. Daniel Blochinger
Stand: 28. Mai 2025
Lizenz: [CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)

Weitere Lehr- und Lernmaterialien finden Sie auf unserer [Webseite](#).

Fehler gefunden? E-Mail an blochinger@dhbw-ravensburg.de!



Was erwartet uns?

<u>Grundlagen & Brute Force</u>	4 - 31
<u>Newton- & Sekantenverfahren</u>	32 - 50
<u>DGLs mit Eulerverfahren</u>	51 - 57
<u>DGLs mit Runge-Kutta-Verfahren</u>	58 - 71
<u>Integration mit Trapezregel</u>	72 - 77
<u>Newton-Cotes & Interpolation</u>	78 - 95
<u>Monte Carlo Integration</u>	96 - 102

<u>Fourier Transformation mit FFT</u>	103 - 111
---------------------------------------	-----------

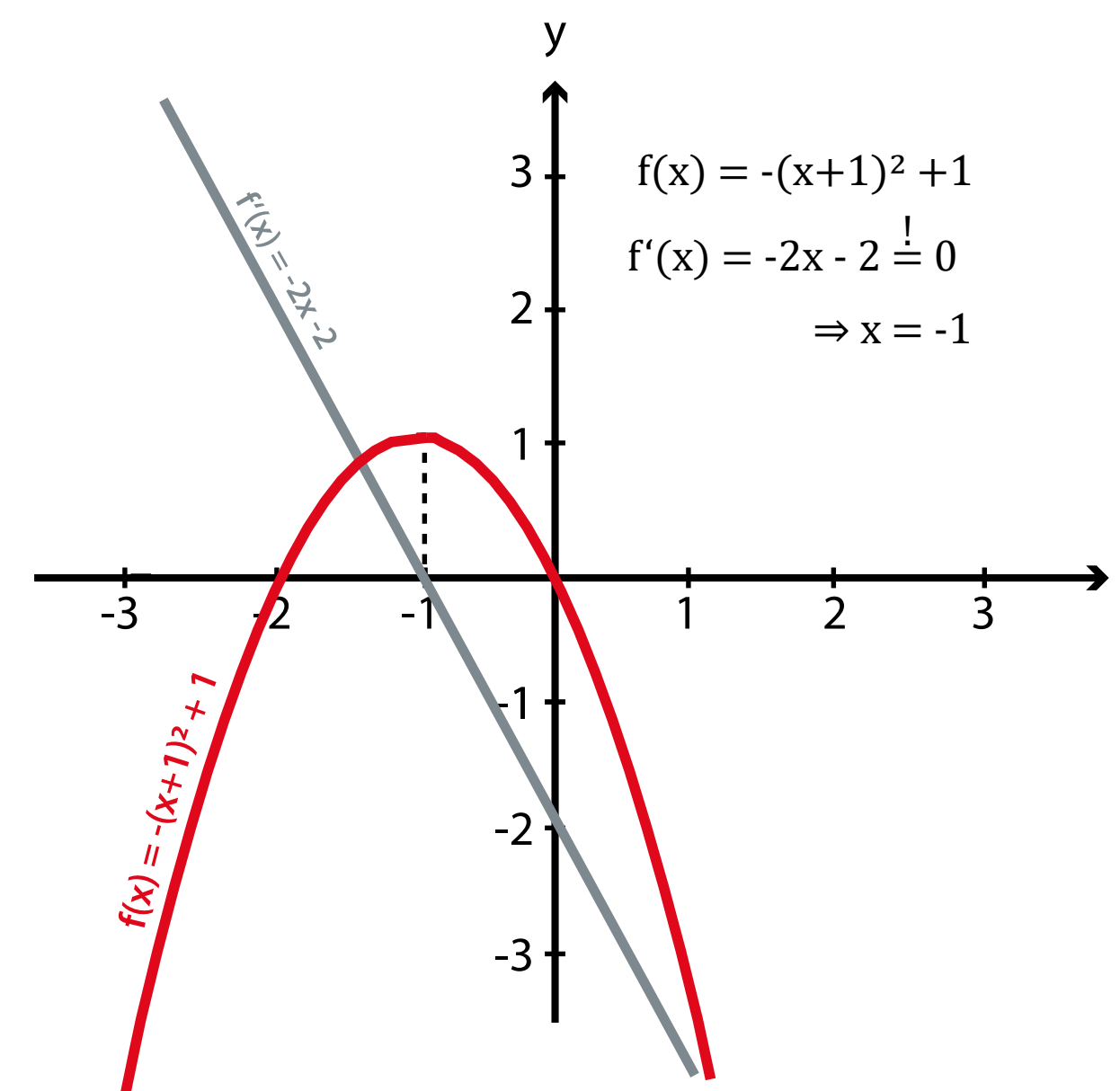
Numerik

Die Numerik ist ein Teilgebiet der Mathematik in der wir unsere Lösungen nicht **analytisch** exakt, sondern **näherungsweise** berechnen.

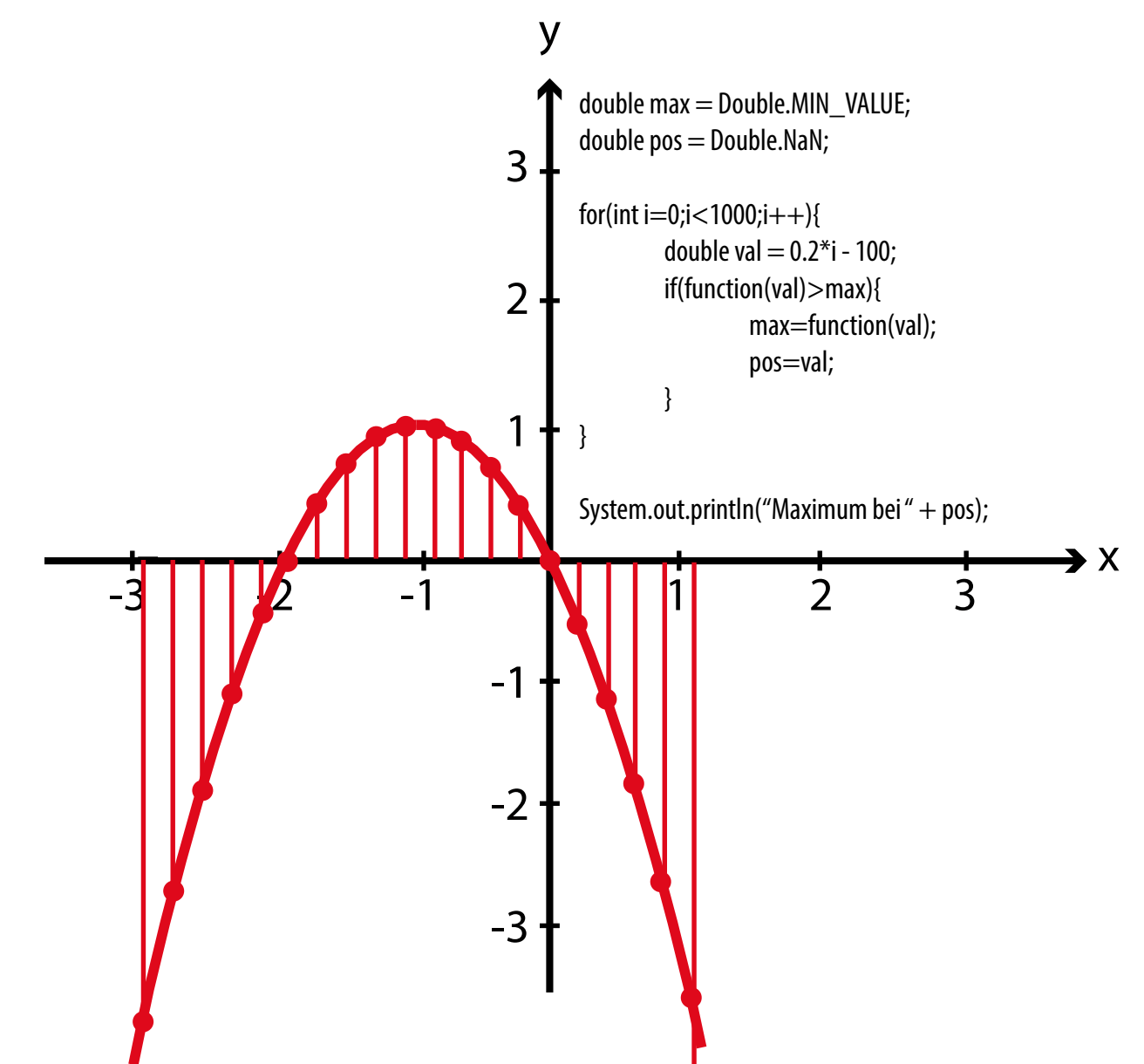
Vorteil Wir können auf die Rechenleistung von Computern zurückgreifen und Probleme angehen, die analytisch nicht lösbar sind.

Nachteil Die Lösungen sind nicht exakt und wir müssen uns über die Größe des Fehlers Gedanken machen.

Analytische Lösung



Numerische Lösung

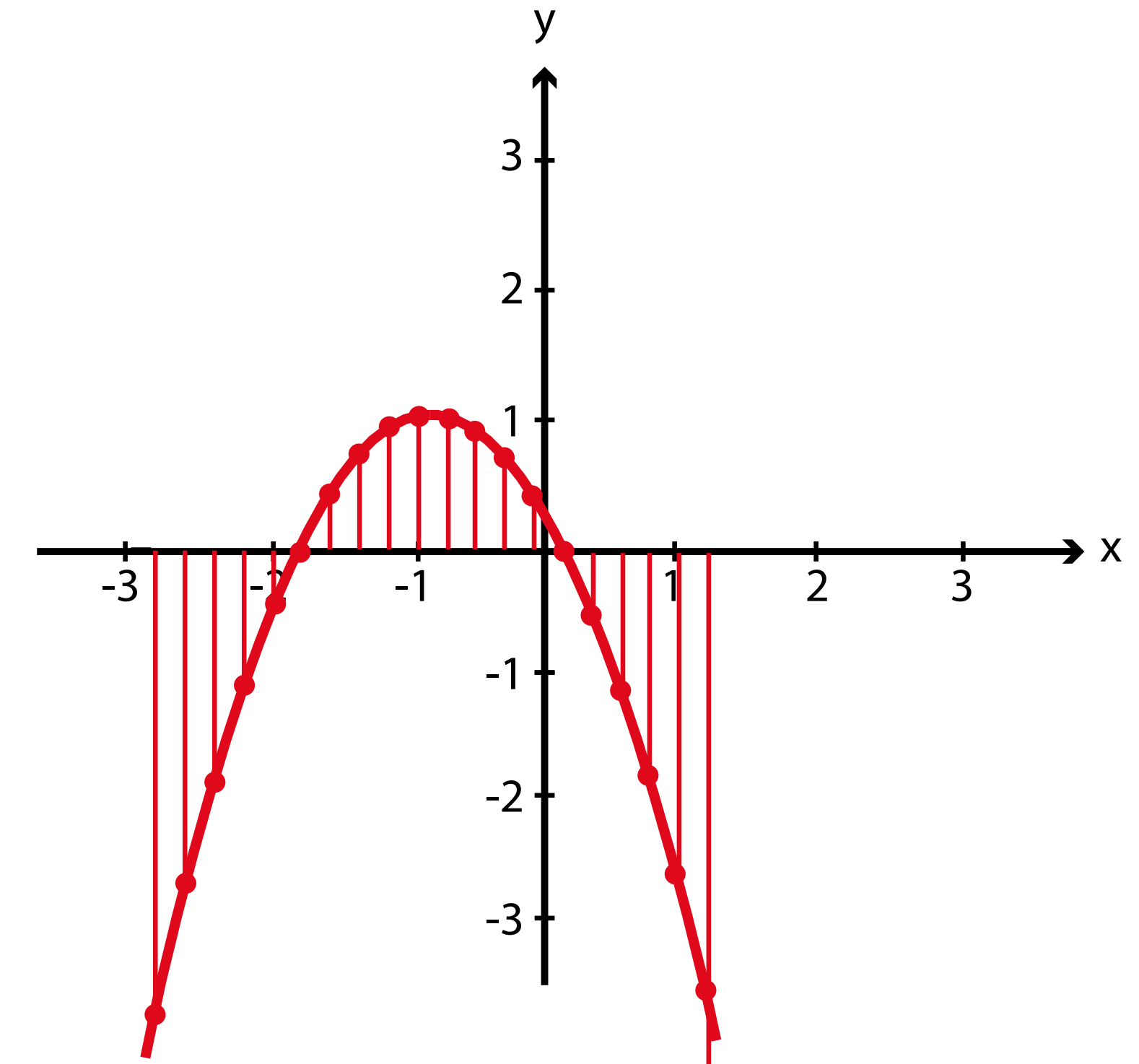


Diskretisierung

Zentraler Ansatz ist die **Diskretisierung**. In der Numerik übersetzen wir kontinuierliche Probleme in diskrete Probleme!

Schauen wir uns dazu das rechts gezeigte Beispiel an. Was wird wie diskretisiert?

```
def gs_max(lower, upper, res, fun):  
  
    span = upper - lower  
    steps = int(span/res)  
  
    maxVal = float('-inf')  
    maxPos = float('-inf')  
  
    for i in range(steps):  
        x = lower + i*res  
        val = fun(x)  
  
        if(val > maxVal):  
            maxVal = val  
            maxPos = x  
  
    return(maxPos)
```



Diskretisierung

Definitionsbereich Der Definitionsbereich der Funktion wird auf das Intervall $[-3,1]$ begrenzt und in Schritte von 0.2 aufgelöst.

Funktionsauswertung Argument und Rückgabewert der Funktion werden mit der **Genauigkeit** einer Double-Variable berechnet.

```
def gs_max(lower,upper,res,fun):
```

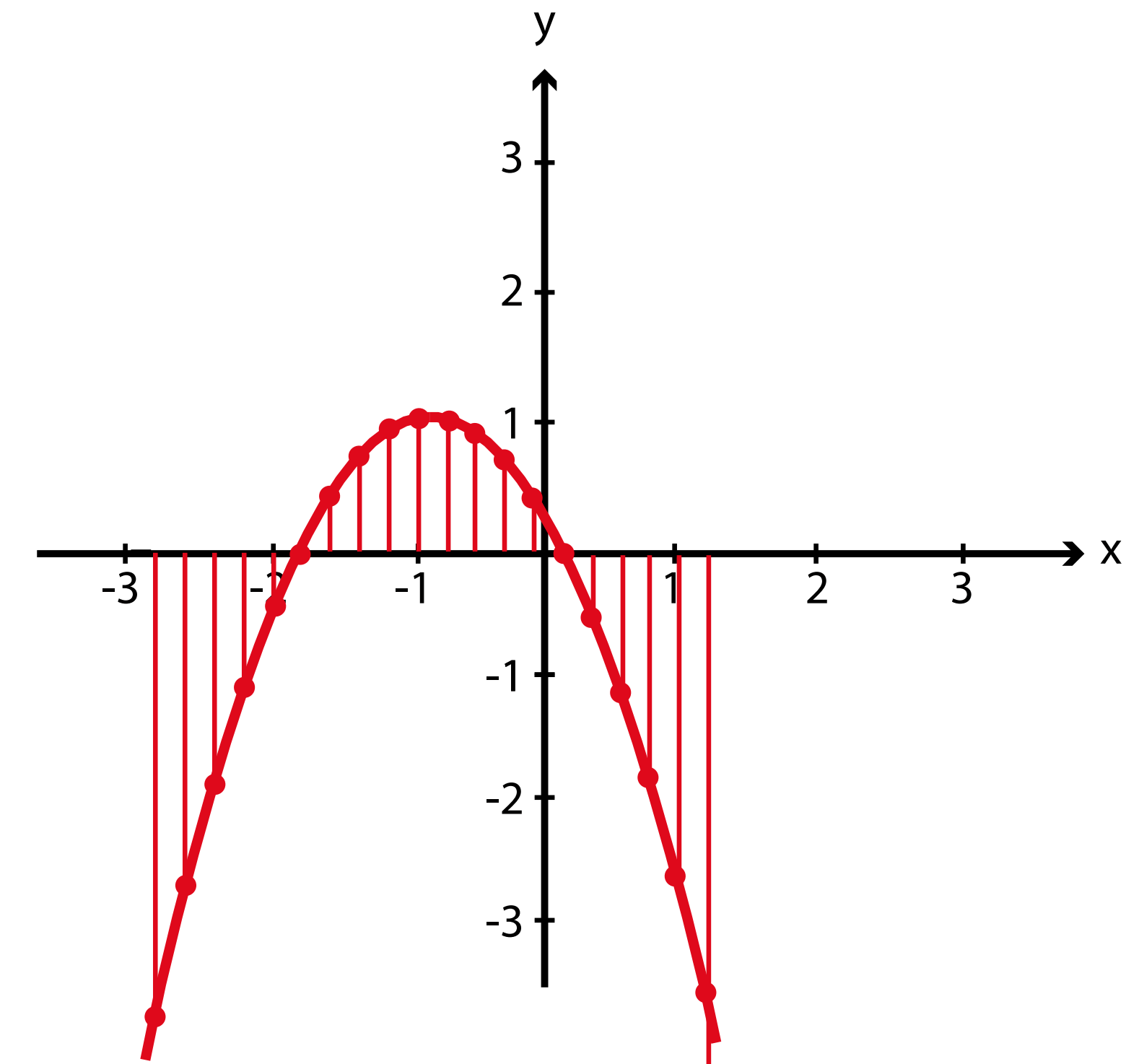
```
    span = upper-lower  
    steps = int(span/res)
```

```
    maxVal = float('-inf')  
    maxPos = float('-inf')
```

```
    for i in range(steps):  
        x = lower + i*res  
        val = fun(x)
```

```
        if(val > maxVal):  
            maxVal = val  
            maxPos = x
```

```
    return(maxPos)
```

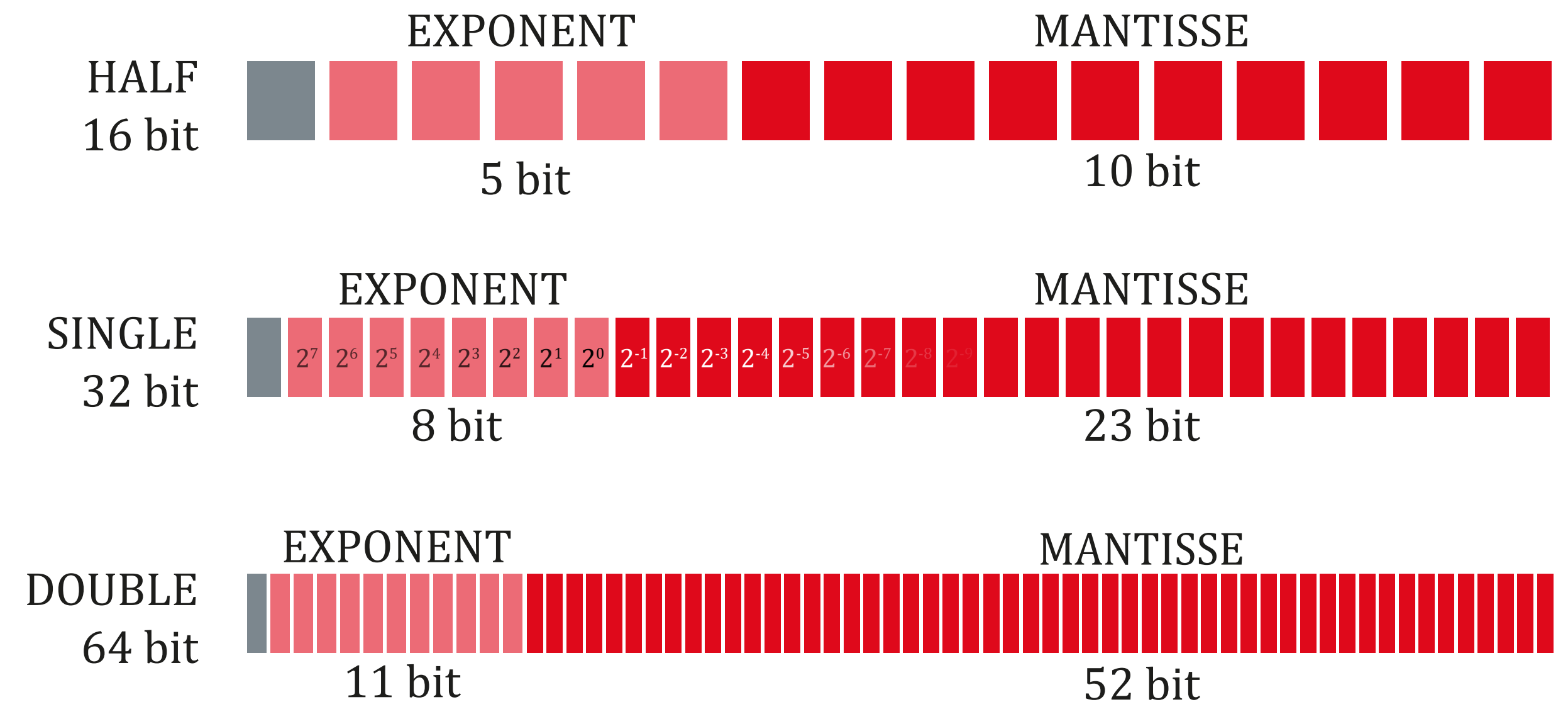


Genauigkeit

Zahlenwerte können in Computern nur mit begrenzter Genauigkeit gespeichert werden.

Im Falle von **Gleitkommazahlen** werden Zahlen in folgender Form gespeichert:

$$x = [\text{VORZEICHEN}] [\text{MANTISSE}] \times 2^{[\text{EXPONENT}]}$$



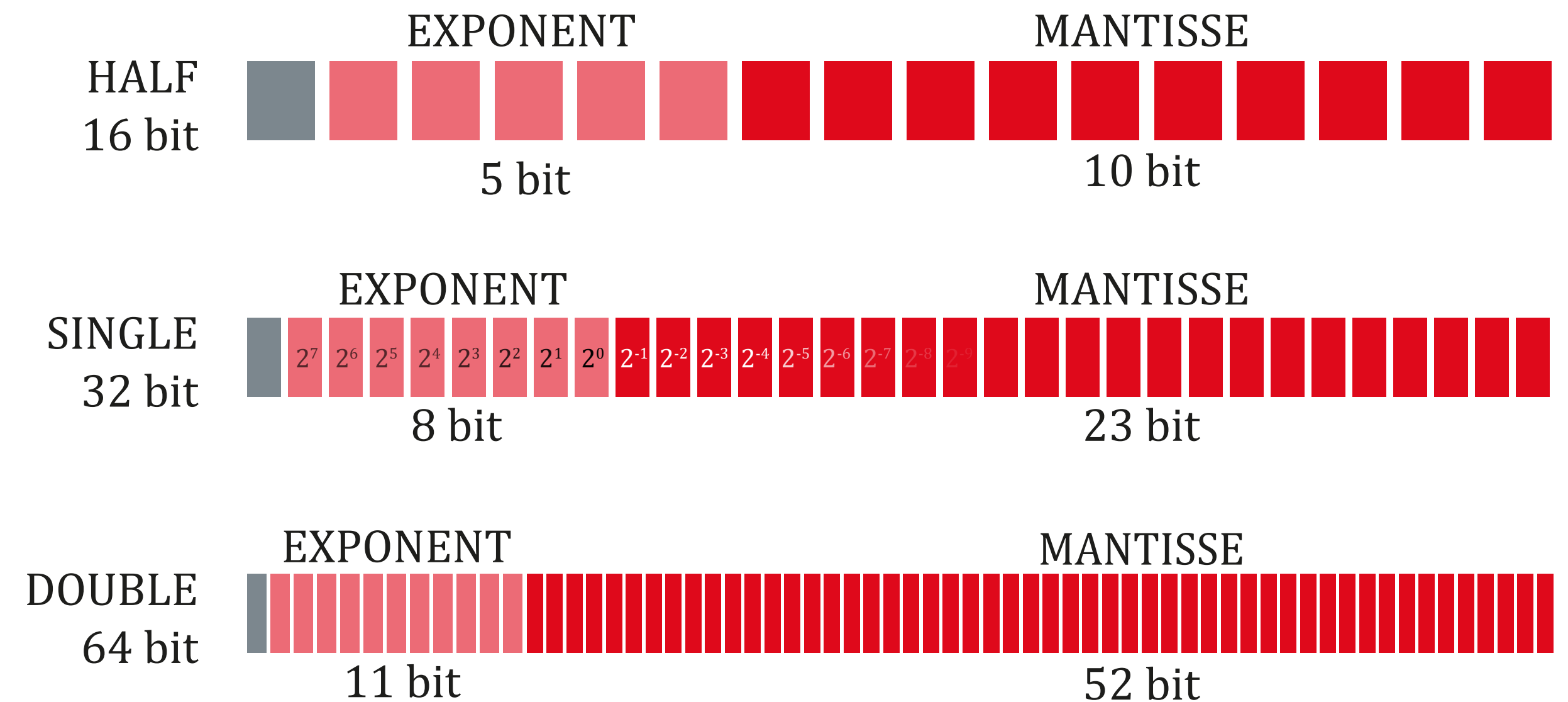
Genauigkeit

Nach IEEE-754 werden Exponent und Mantisse bei einfacher Präzision folgendermaßen gebildet:

Wenn Exponent nicht 0000 0000 ist, gilt ...

Exponent = 8-bit Binärzahl minus 127

$$\text{Mantisse} = 1 + \sum_{k=0}^{23} 2^{-k}$$



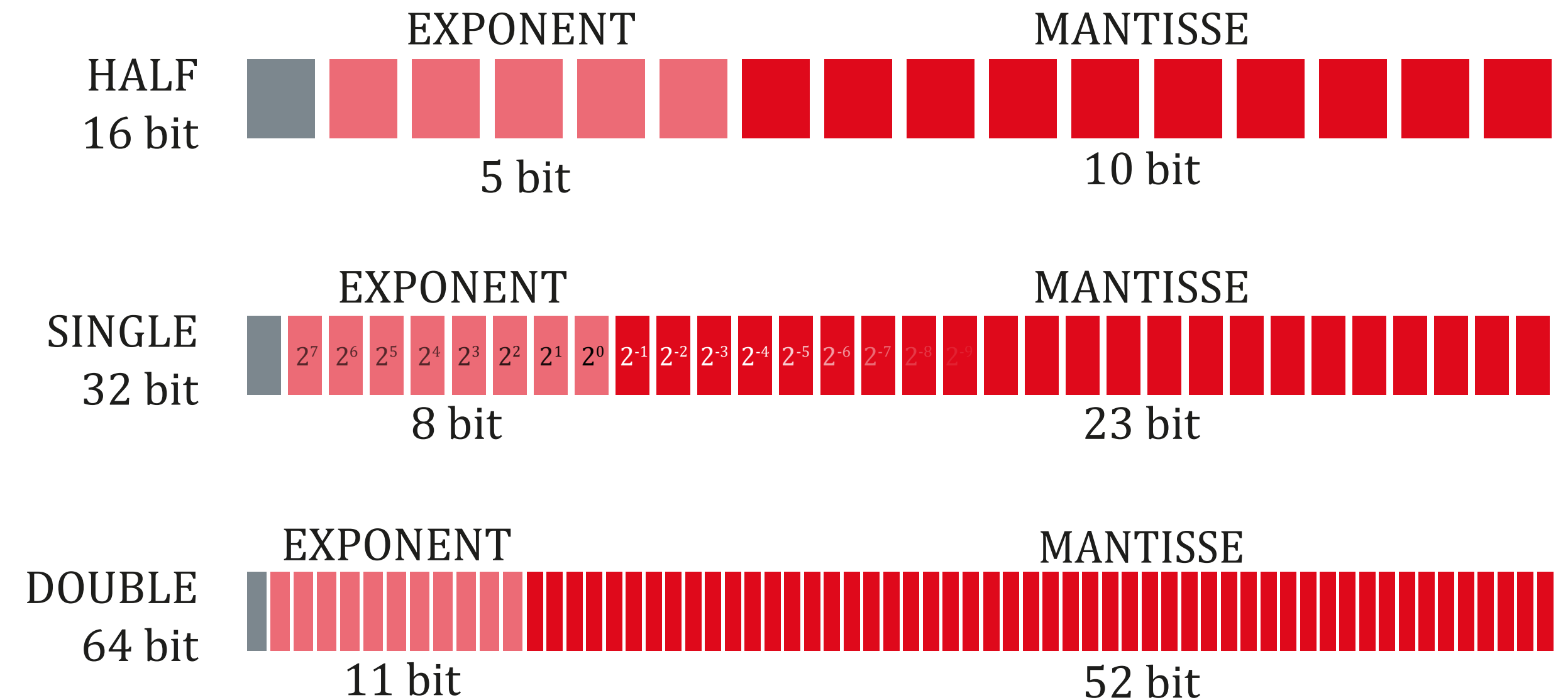
Genauigkeit

Nach IEEE-754 werden Exponent und Mantisse bei einfacher Präzision folgendermaßen gebildet:

Wenn Exponent genau 0000 0000 ist, gilt ...

$$\text{Exponent} = 2^{-126}$$

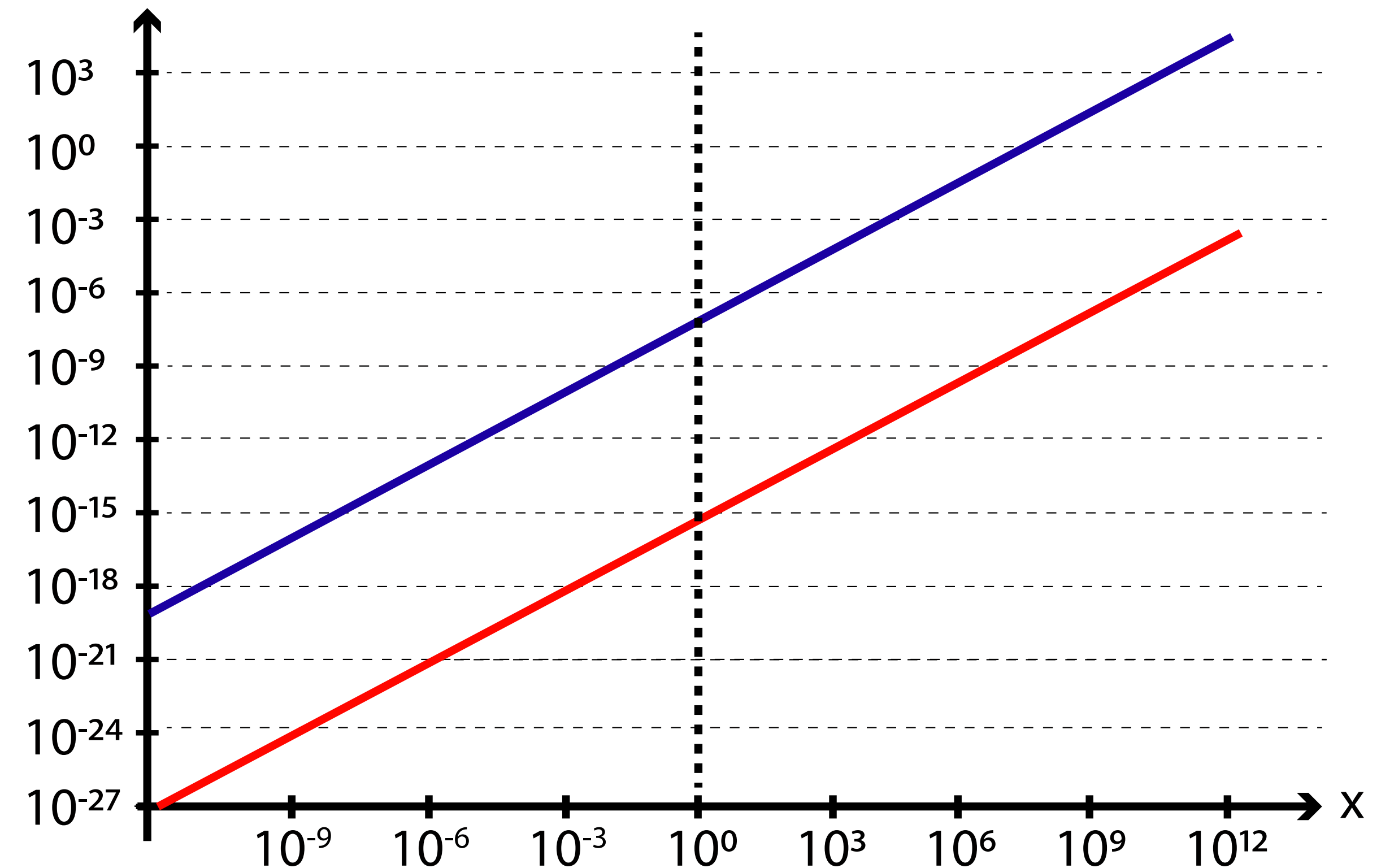
$$\text{Mantisse} = \sum_{k=0}^{23} 2^{-k}$$



Genauigkeit

Die Genauigkeit von Gleitkommazahlen variiert über den Wertebereich.

Gerade bei sehr großen Zahlen lässt die Genauigkeit massiv nach und wir sehen einen immensen Unterschied zwischen Single und Double.



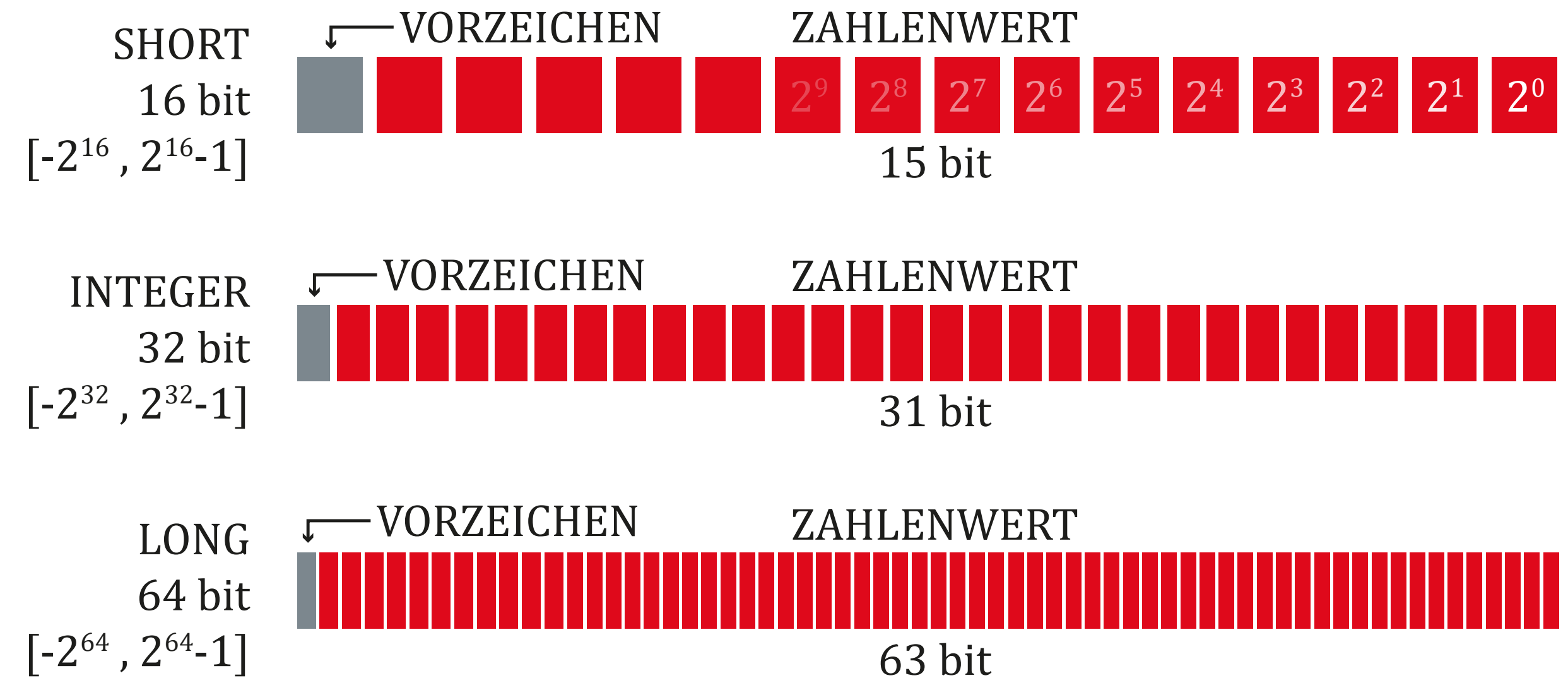
Datenquelle: CC BY-SA 4.0 von Autor Ghennessy auf Wikipedia (<https://commons.wikimedia.org/wiki/File:IEEE754.png>)

Genauigkeit

Im Falle von **Festkommazahlen** werden Zahlen in folgender Form gespeichert:

$$x = [\text{VORZEICHEN}] \sum_{k=0}^{n-2} 2^k \delta_{x_k,1}$$

Dadurch ergeben sich zunächst ganze Zahlen. Die Position des Kommas ist entweder hard-coded oder wird in einer separaten Variable angegeben.

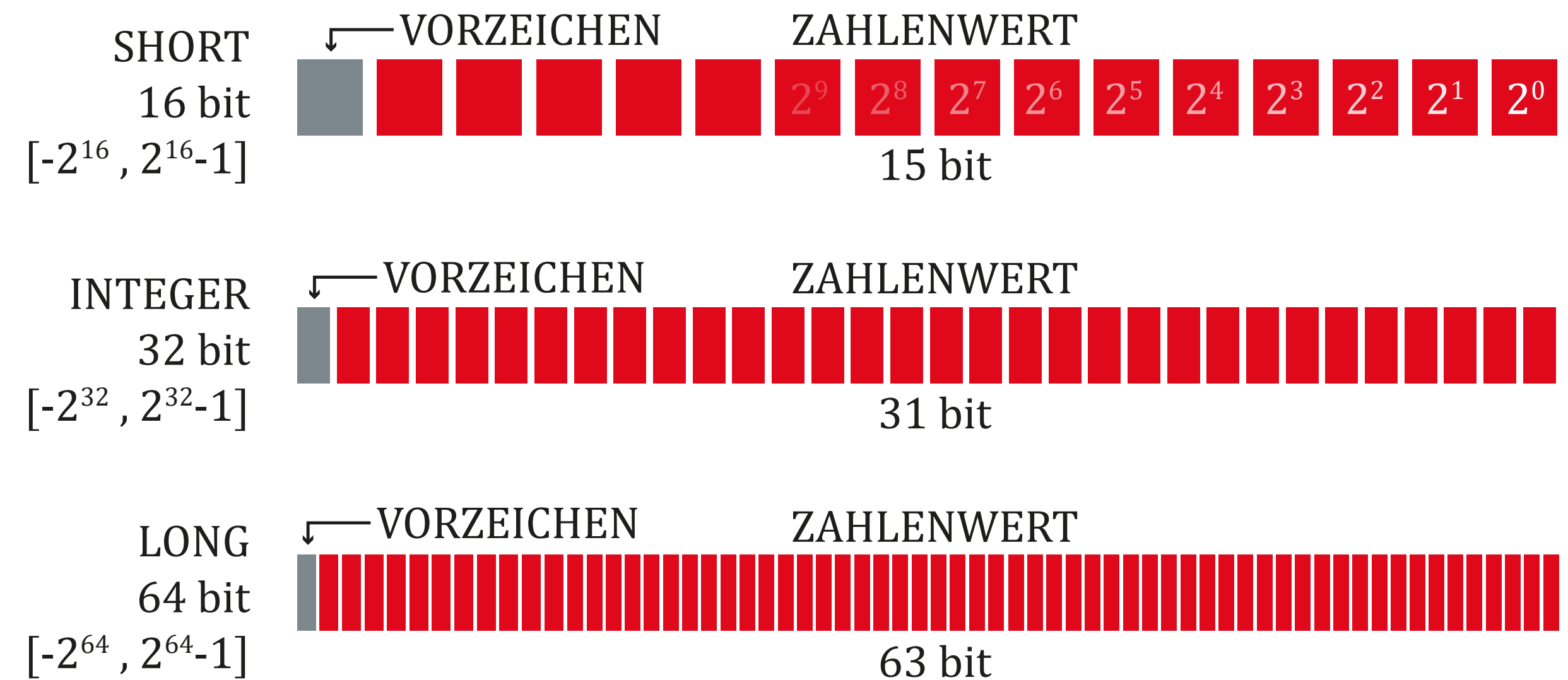


Auflistung zeigt Datentypen in C für einen modernen Compiler auf einem PC-System. Im Bereich von Mikrocontrollern kann z. B. ein Integer abweichend auch nur 16 Bit haben.

Genauigkeit

Die Genauigkeit ist bei **Festkommazahlen** über den ganzen Wertebereich konstant.

Beispiel: Wir wollen einen Wert im Bereich $[-1000, 1000]$ in einem 32-bit Integer speichern.



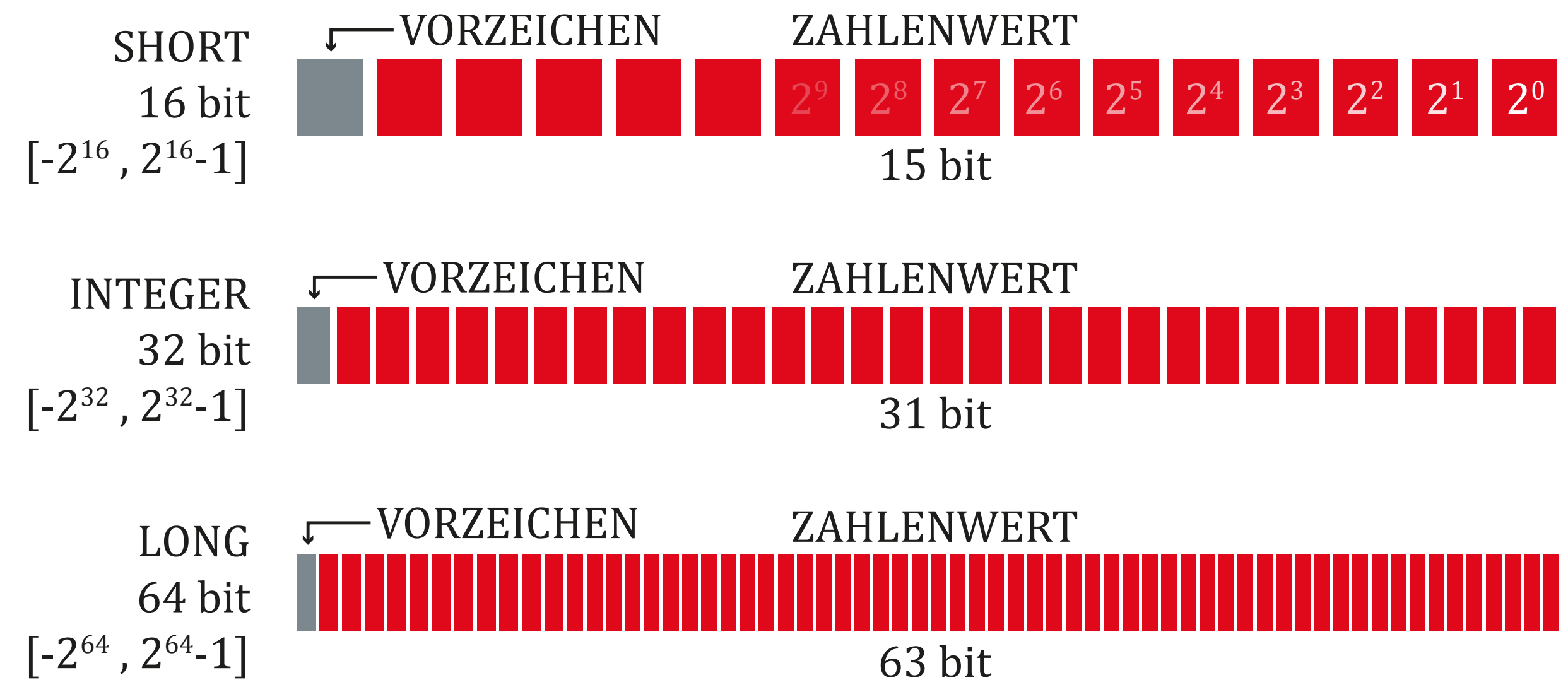
Auflistung zeigt Datentypen in C für einen modernen Compiler auf einem PC-System. Im Bereich von Mikrocontrollern kann z. B. ein Integer abweichend auch nur 16 Bit haben.

Genauigkeit

Ohne Skalierung können wir in der Variable Werte im Bereich ± 4.29 Mrd. speichern.

Da wir nur den Wertebereich $[-1000, 1000]$ benötigen wählen wir den Skalierungsfaktor als 10^{-6} .

Damit haben wir eine Genauigkeit von 10^{-6} bzw. einen maximalen Rundungsfehler von $0.5 \cdot 10^{-6}$.



Auflistung zeigt Datentypen in C für einen modernen Compiler auf einem PC-System. Im Bereich von Mikrocontrollern kann z. B. ein Integer abweichend auch nur 16 Bit haben.

Fehleranalyse

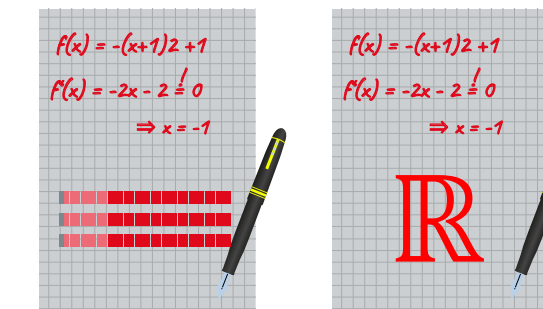
In der Numerik unterscheiden wir folgende Kriterien für die Bewertung unserer Ergebnisse: **Kondition**, **Stabilität**, **Konsistenz** und **Konvergenz**.

Das Zusammenspiel der ersten drei Kriterien entscheidet über die **Konvergenz** und damit über die Gesamtqualität unseres Ergebnisses.

Sei $f(\dots)$ die analytische Lösung und $\tilde{f}(\dots)$ der numerische Algorithmus. Außerdem stehe x für die exakten Eingabedaten und $\tilde{x}$ für die dem Algorithmus gegebenen Eingabedaten.

Kondition

$$| f(\tilde{x}) - f(x) |$$



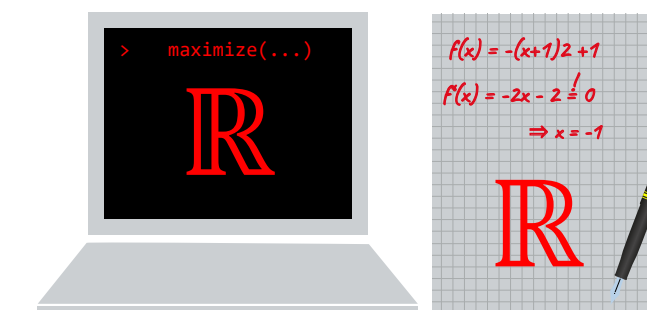
Stabilität

$$| \tilde{f}(\tilde{x}) - \tilde{f}(x) |$$



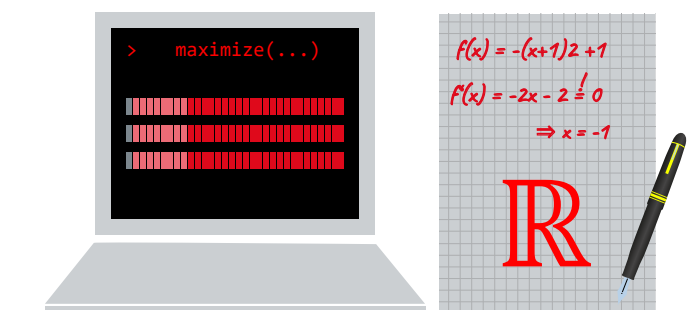
Konsistenz

$$| \tilde{f}(x) - f(x) |$$



Konvergenz

$$| \tilde{f}(\tilde{x}) - f(x) |$$



Fehleranalyse

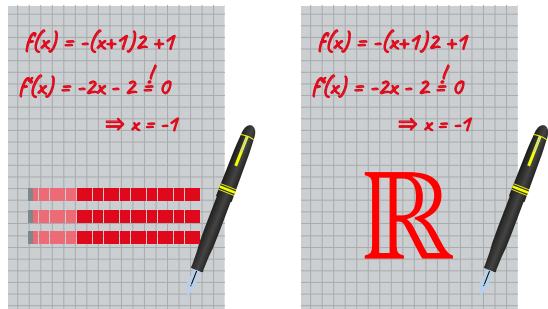
Wir suchen das Maximum der folgenden Funktion mit unserem R-Skript:

$$f(x) = 1 - e^{(x^2 - \pi)^2}$$

Das Ergebnis wollen wir mit den rechts gezeigten Maßen bewerten.

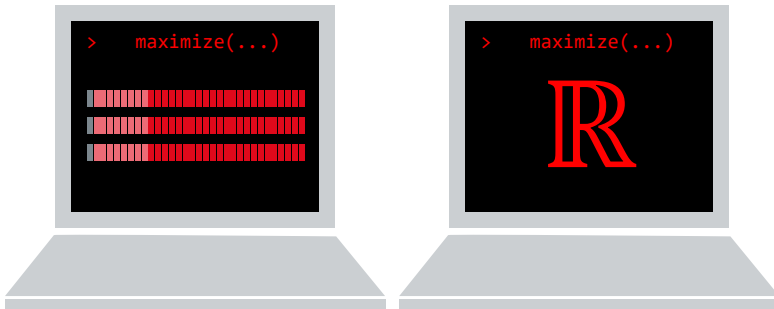
Kondition

$$| f(\tilde{x}) - f(x) |$$



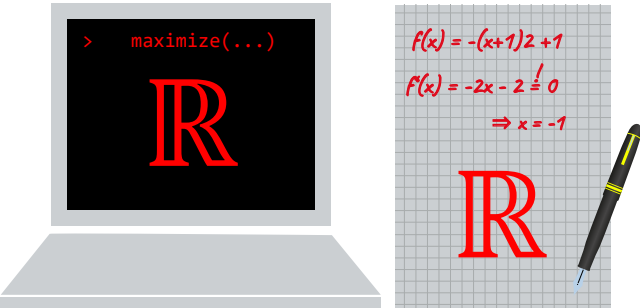
Stabilität

$$| \tilde{f}(\tilde{x}) - \tilde{f}(x) |$$



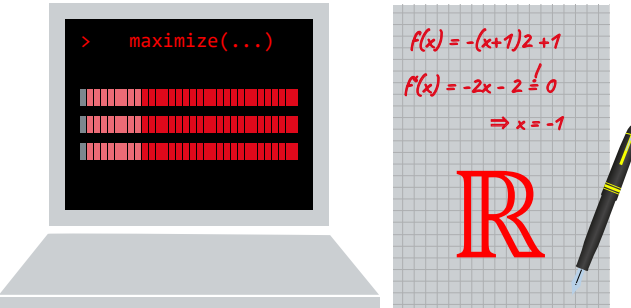
Konsistenz

$$| \tilde{f}(x) - f(x) |$$



Konvergenz

$$| \tilde{f}(\tilde{x}) - f(x) |$$



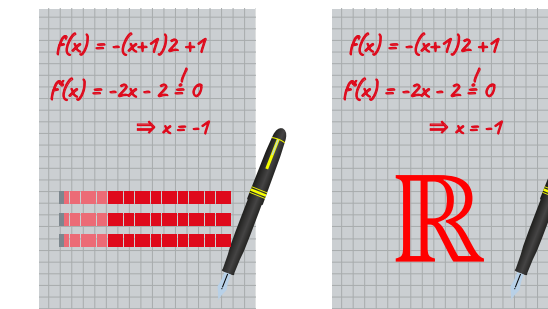
Fehleranalyse

Die **Konsistenz** hängt von der **Schrittweite** bzw. **Abtaste** ab, die wir dem Algorithmus vorgeben. Im Beispielcode beträgt die Schrittweite 0.2 was einer Abtaste von 5 entspricht.

Lassen wir die Abtaste unseres Algorithmus gegen unendlich gehen, ist die Konsistenz perfekt gegeben.

Kondition

$$| f(\tilde{x}) - f(x) |$$



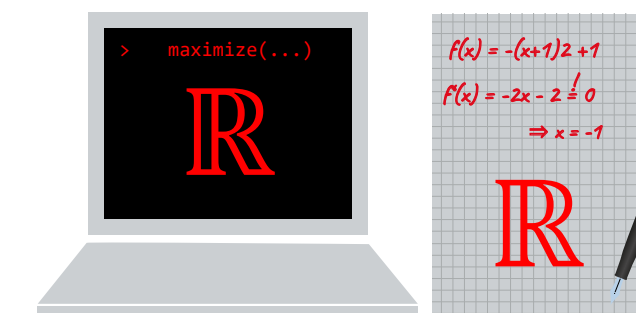
Stabilität

$$| \tilde{f}(\tilde{x}) - \tilde{f}(x) |$$



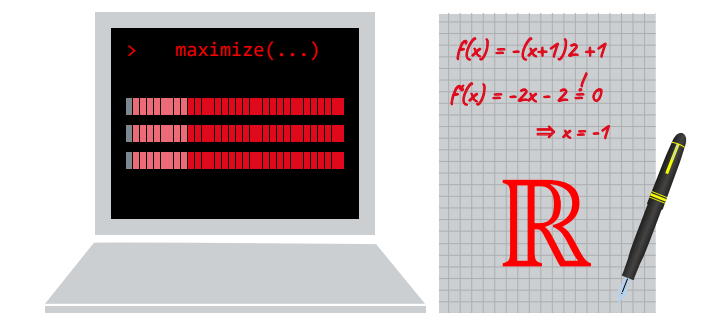
Konsistenz

$$| \tilde{f}(x) - f(x) |$$



Konvergenz

$$| \tilde{f}(\tilde{x}) - f(x) |$$



Fehleranalyse

Die **Kondition** hängt von der Genauigkeit ab, mit der wir die Zahlenwerte 0, 1, 2, 4, e, π sowie die Wurzel von π abbilden können.

$$f(x) = 1 - e^{(x^2 - \pi)^2}$$

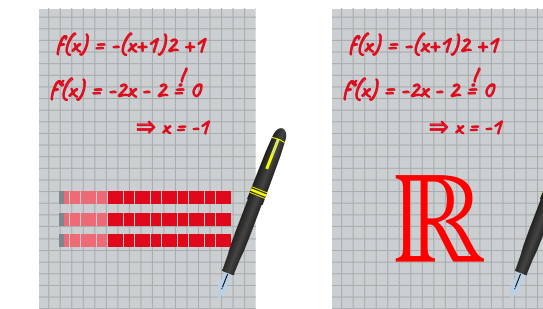
$$f'(x) = -4x(x^2 - \pi)e^{(x^2 - \pi)^2} \stackrel{!}{=} 0$$

$$\Rightarrow -4x(x^2 - \pi) = 0 \Rightarrow x_1 = 0 \quad x_{2,3} = \pm\sqrt{\pi}$$

R und Python arbeiten mit 64-bit Gleitkommazahlen. Die oben genannten Zahlen sind exakt bzw. auf die 15-17te Stelle genau. Der Genauigkeitsverlust ist sehr gering.

Kondition

$$|f(\tilde{x}) - f(x)|$$



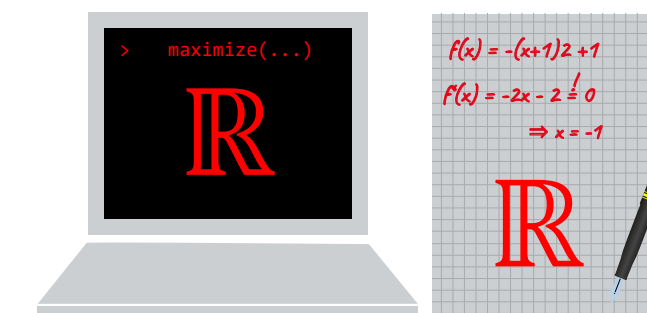
Stabilität

$$|\tilde{f}(\tilde{x}) - \tilde{f}(x)|$$



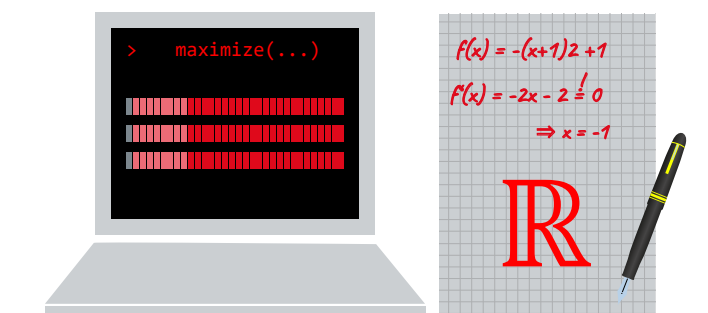
Konsistenz

$$|\tilde{f}(x) - f(x)|$$



Konvergenz

$$|\tilde{f}(\tilde{x}) - f(x)|$$



Fehleranalyse

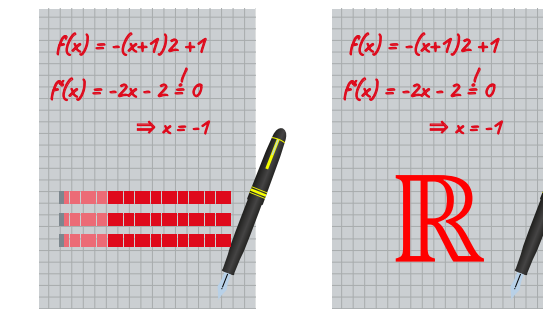
Die **Stabilität** zeigt an, wie viel Genauigkeit wir durch die Darstellung von reellen Zahlen als Gleit- oder Festkommazahlen verlieren.

Bei einer hohen Schrittweite bzw. niedrigen Abtastrate kann der dadurch verursachte Fehler gegen 0 gehen.

Lassen wir dagegen die Abtastrate unseres Algorithmus gegen unendlich gehen, entspricht die Stabilität der Kondition.

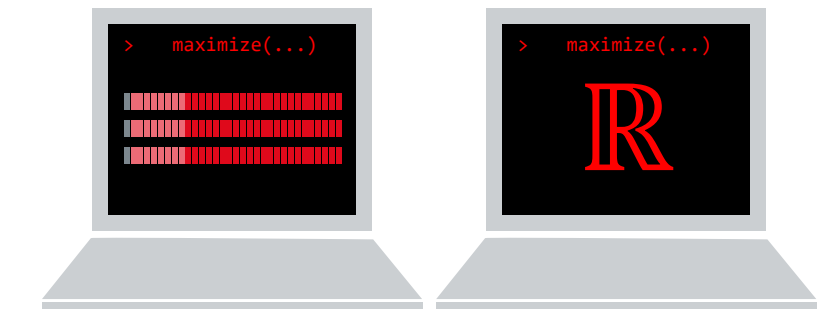
Kondition

$$| f(\tilde{x}) - f(x) |$$



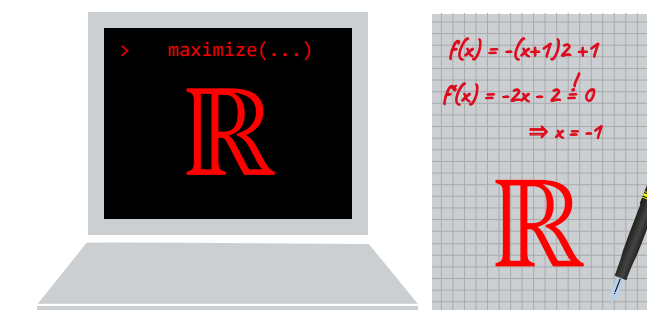
Stabilität

$$| \tilde{f}(\tilde{x}) - \tilde{f}(x) |$$



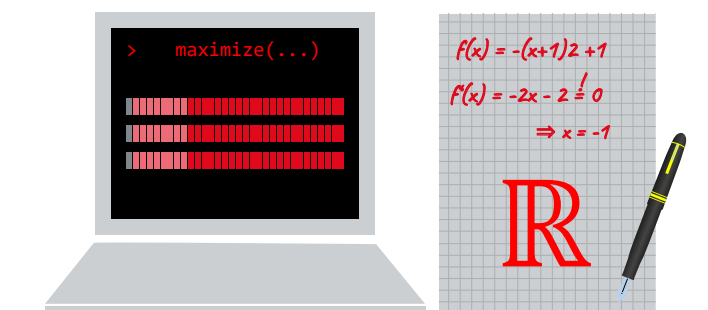
Konsistenz

$$| \tilde{f}(x) - f(x) |$$



Konvergenz

$$| \tilde{f}(\tilde{x}) - f(x) |$$



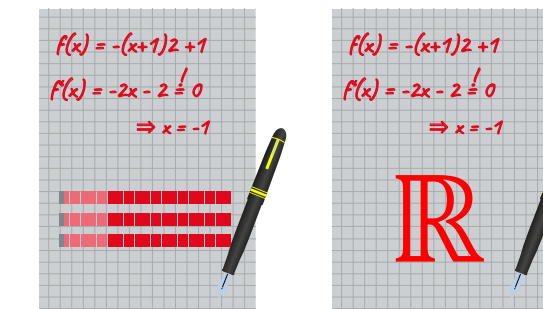
Fehleranalyse

Konsistenz, Kondition und Stabilität sind in ausreichendem Maße gegeben und damit auch die Konvergenz.

Einwurf Sind Kondition und Stabilität bei der Verwendung von 64-bit Gleitkommazahlen nie ein nennenswertes Problem?

Kondition

$$| f(\tilde{x}) - f(x) |$$



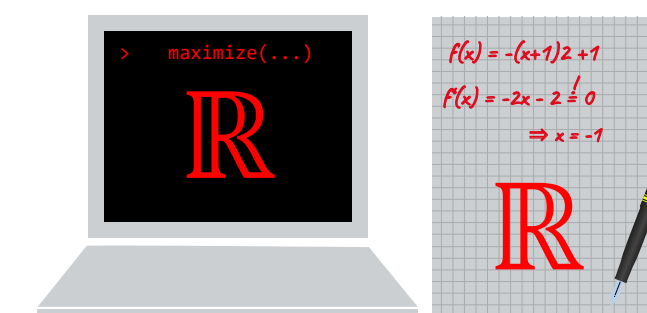
Stabilität

$$| \tilde{f}(\tilde{x}) - \tilde{f}(x) |$$



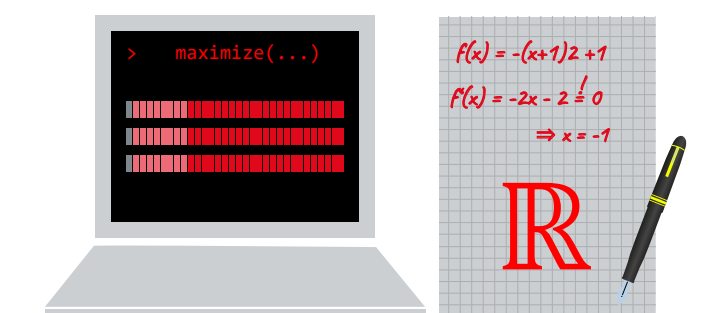
Konsistenz

$$| \tilde{f}(x) - f(x) |$$



Konvergenz

$$| \tilde{f}(\tilde{x}) - f(x) |$$



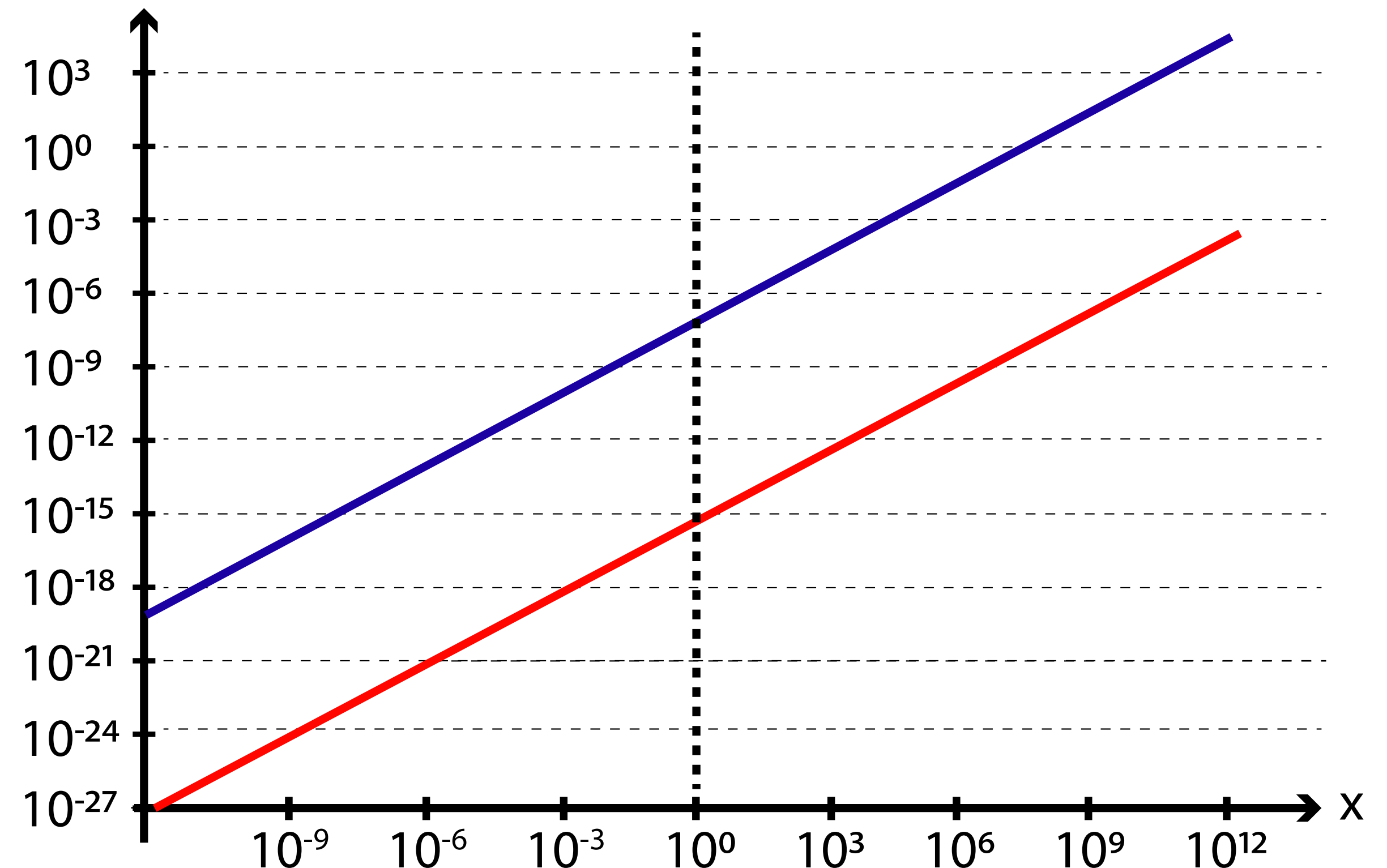
Doppelt ist genug?!

Wir wissen bereits, dass die Präzision bei großen Zahlen nachlässt.

Extrem kleine Zahlen können ebenfalls ein Problem darstellen! Die kleinste Zahl > 0 die wir in einer Double-Variable speichern können ist:

$$4.940 \cdot 10^{-324}$$

Die Nächste ist bereits doppelt so groß.



Datenquelle: CC BY-SA 4.0 von Autor Ghennessy auf Wikipedia (<https://commons.wikimedia.org/wiki/File:IEEE754.png>)

Doppelt ist genug?!

Kondition und Stabilität sind problematisch, wenn wir mit höheren Potenzen arbeiten.

Nehmen wir mal an, wir würden in unserem Algorithmus π als Integervariable speichern und daher $\pi=3$ verwenden.

Der Fehler verstärkt sich deutlich, wenn wir unser π zu einer hohen Potenz nehmen oder als Exponent verwenden.

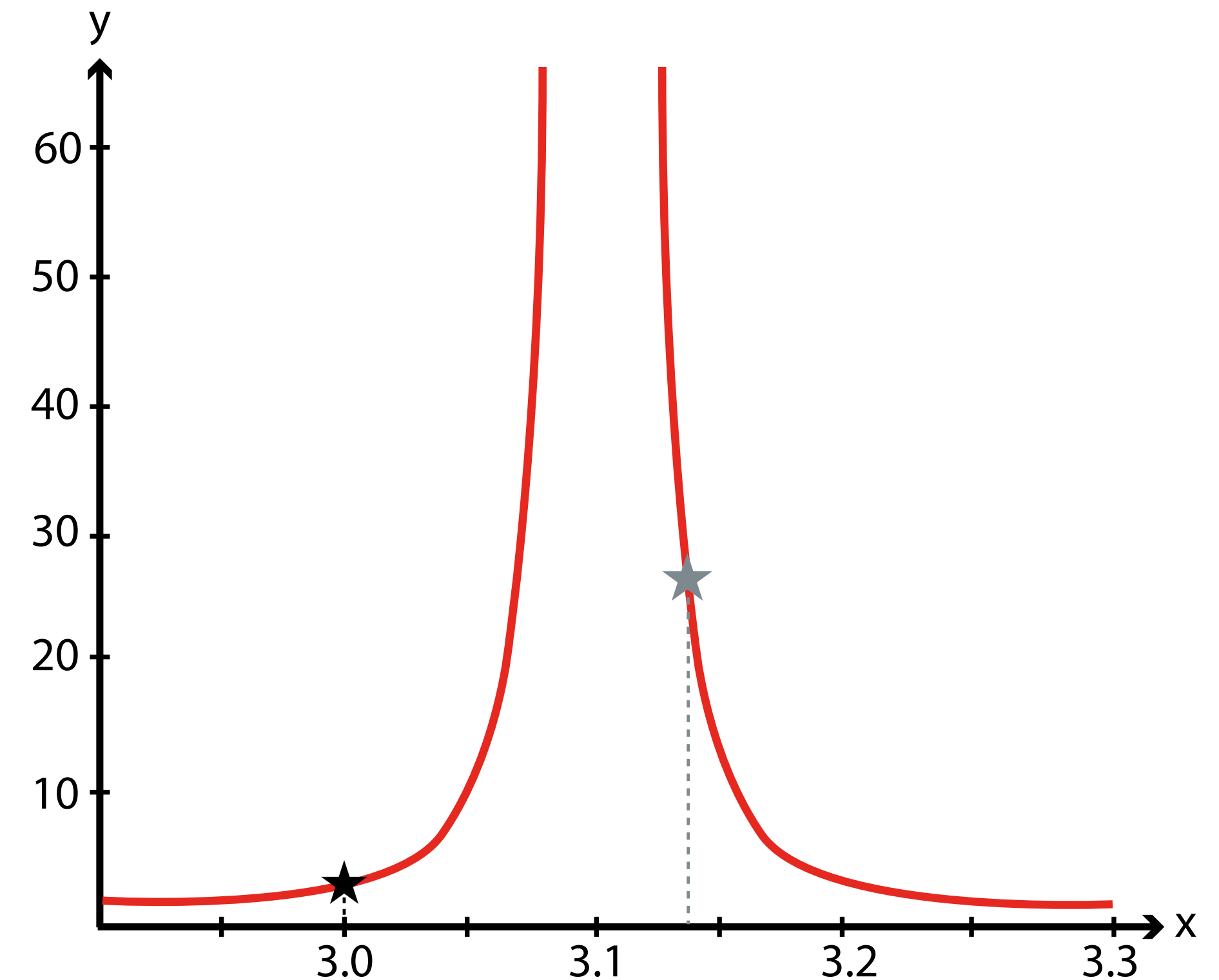
Berechnung	Ergebnis Soll	Ergebnis Ist	Fehler
$\log(\pi)$	1.1447	1.0986	0.0461
$\sqrt{\pi}$	1.7724	1.7320	0.0404
π	3.1415	3.0000	0.1415
π^2	9.8696	9.0000	0.8696
2^π	8.8249	8.0000	0.8249

Doppelt ist genug?!

Kondition und Stabilität sind problematisch, wenn wir uns in der Nähe von Singularitäten bewegen.

Nehmen wir erneut an, wir würden in unserem Algorithmus π als Integervariable speichern und daher $\pi=3$ verwenden.

Bei der Auswertung der rechts gezeigten Funktion wäre der dadurch entstehende Fehler extrem.



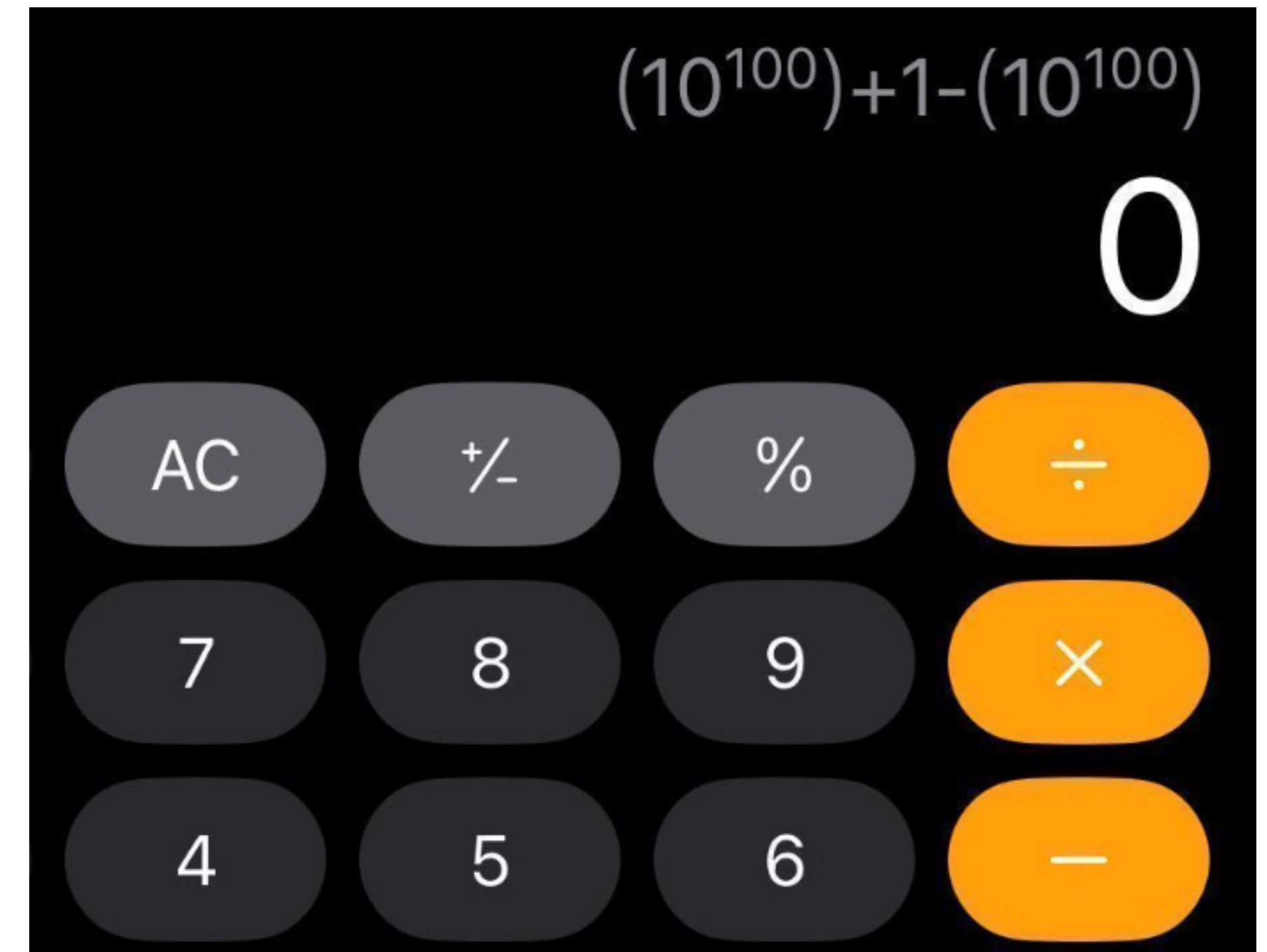
Doppelt ist genug?!

Was tun, wenn 64-bit Gleitkommazahlen nicht ausreichen?

Höher aufgelöste Gleitkommazahlen wie z. B. `numpy.float128` oder `long double` verwenden.

Festkommazahlen verwenden. In Python stehen neben beliebig großen Integers auch der `Decimal`-Typ zur Verfügung.

Das Problem zunächst analytisch umformen bzw. vereinfachen. Idealerweise nicht händisch, sondern mit einem CAS.



Ein sehr tiefes Rabbit-Hole zum Thema numerische Genauigkeit bei Taschenrechnern (<https://chadnauseam.com/coding/random/calculator-app>)

Zeitkomplexität

Unser erstes Beispiel ist ein eindimensionaler grid search und verwendet eine **brute force Vorgehensweise**.

Im Grenzwert Abtastrate gegen unendlich und Suchbereich gegen $[-\infty, \infty]$ testen wir alle potenziellen Lösungen durch.

Für viele Anwendungen im Bereich Data Science ist die hohe **Zeitkomplexität** nicht praktikabel.

$O(2^n)$	Exponentieller Zeitaufwand
$O(n^2)$	Quadratisch Zeitaufwand
$O(n \log n)$	Log-Linear Zeitaufwand
$O(n)$	Linear Zeitaufwand
$O(\log n)$	Logarithmisch Zeitaufwand
1	Konstanter Zeitaufwand

Zeitkomplexität

Die **Laufzeit** unseres einfachen Algorithmus für Extremstellen skaliert für Funktionen ...

$$f: \mathbb{R}^m \rightarrow \mathbb{R}$$

... mit $O(n^m)$ in der Abtaste n und der Anzahl an Dimensionen m .

$O(2^n)$	Exponentieller Zeitaufwand
$O(n^2)$	Quadratisch Zeitaufwand
$O(n \log n)$	Log-Linear Zeitaufwand
$O(n)$	Linear Zeitaufwand
$O(\log n)$	Logarithmisch Zeitaufwand
1	Konstanter Zeitaufwand

Zeitkomplexität

Bei kleineren Datensätzen kann ein Algorithmus mit höherer Zeitkomplexität nicht nur praktikabel, sondern sogar schneller sein, wenn die Zeit pro Durchlauf kürzer ist.

Für ausreichend große Datenmengen ist eine geringere Zeitkomplexität immer von Vorteil. Zum Beispiel gilt:

$$\forall a, b \in \mathbb{R}^+ \exists n \in \mathbb{R}^+ : a \cdot n^2 < b \cdot 2^n$$

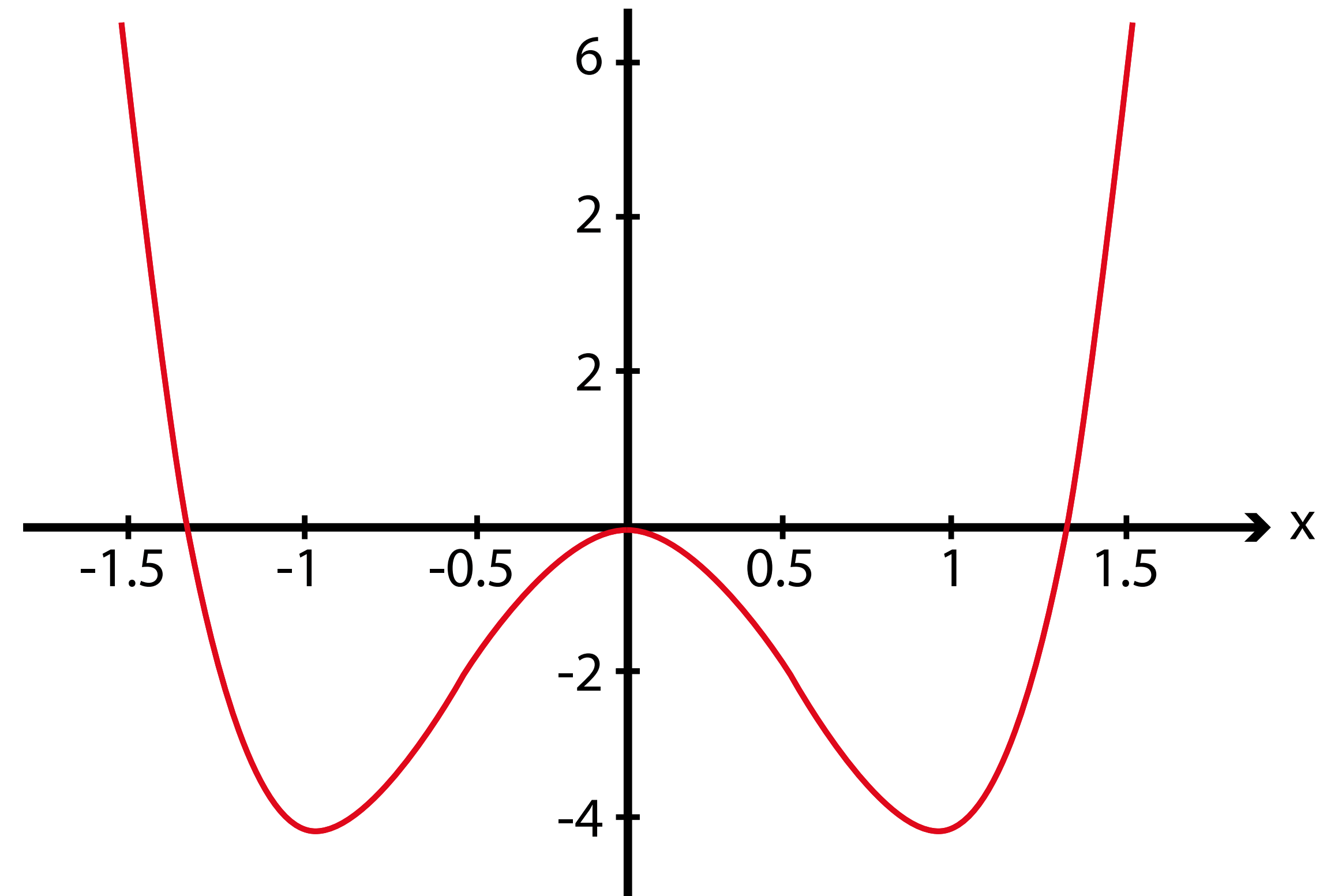
$O(2^n)$	Exponentieller Zeitaufwand
$O(n^2)$	Quadratisch Zeitaufwand
$O(n \log n)$	Log-Linear Zeitaufwand
$O(n)$	Linear Zeitaufwand
$O(\log n)$	Logarithmisch Zeitaufwand
1	Konstanter Zeitaufwand

Extremstellen Brute-Force

Entwickle die Funktion "gs_max" in "Grid Search Maximum" zu "gs_extreme" weiter.

Die neue Funktion soll alle Extremstellen einer Funktion ausgeben und dabei zwischen lokalen/globalen Minima und Maxima unterscheidet.

Implementiere außerdem eine Zeitmessung.

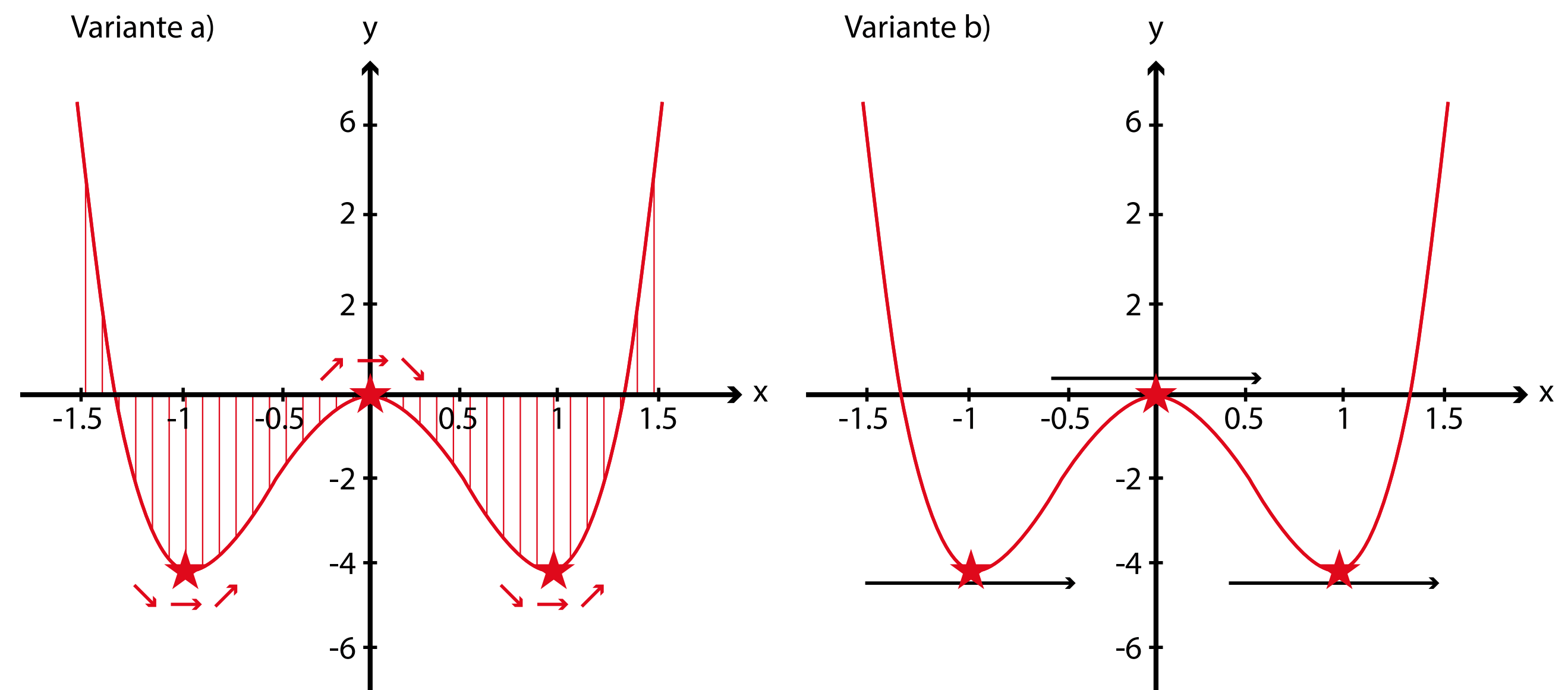


Extremstellen Brute-Force

Es gibt mehrere Algorithmen, mit denen die Aufgabe gelöst werden kann:

a) Wir vergleichen den Funktionswert einer Stelle x_n mit den beiden Stellen x_{n-1} und x_{n-2} davor. Ist x_{n-1} der größte/kleinste Wert, dann haben wir ein Maximum/Minimum bei x_{n-1} .

b) Wir berechnen die erste und zweite Ableitung numerisch über ein Steigungsdreieck und prüfen die üblichen notwendigen und hinreichenden Kriterien.

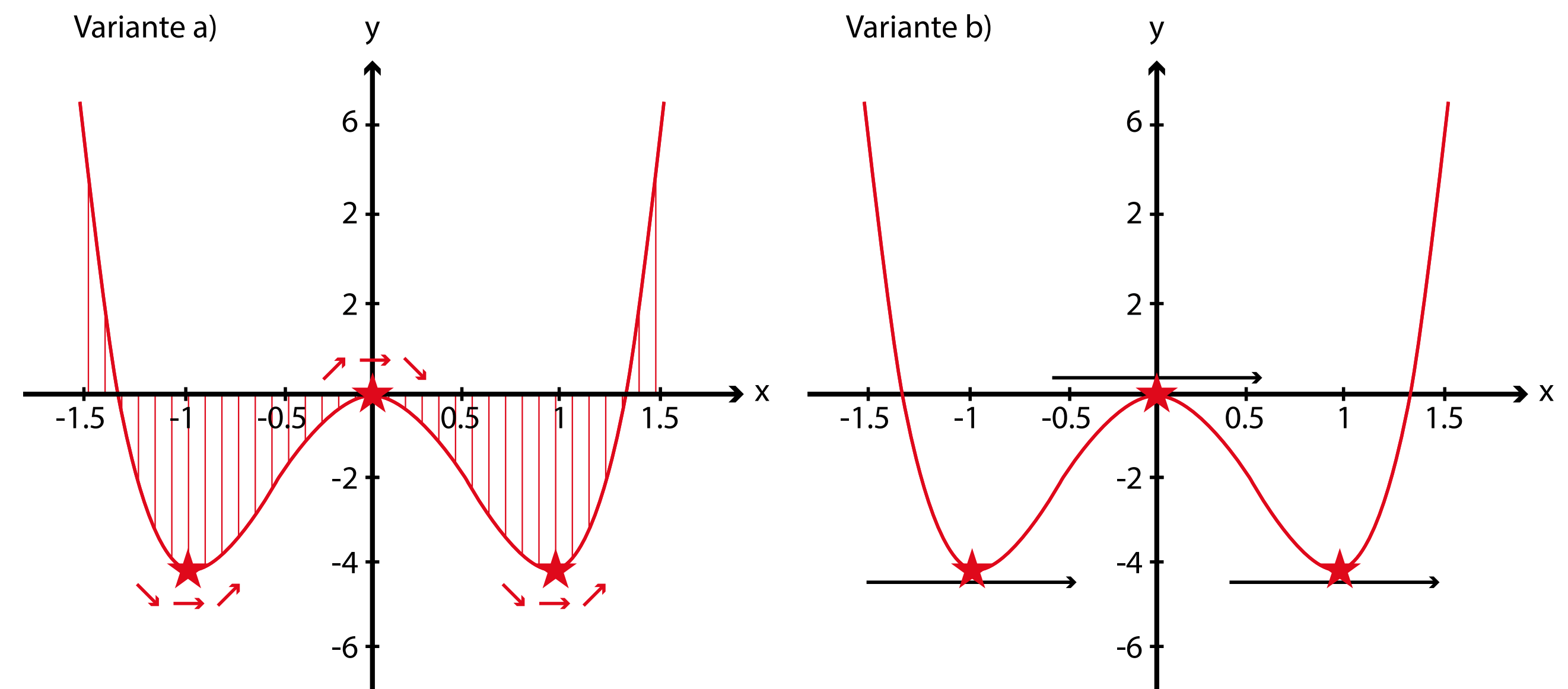


Nachtrag zu Brute Force

Die brute force Vorgehensweise hat einige Vorteile.

Der Algorithmus ist einfach und nachvollziehbar. Er besteht aus einer Schleife über die potenziellen Lösungen und einer Bedingung, welche korrekte Lösungen identifiziert.

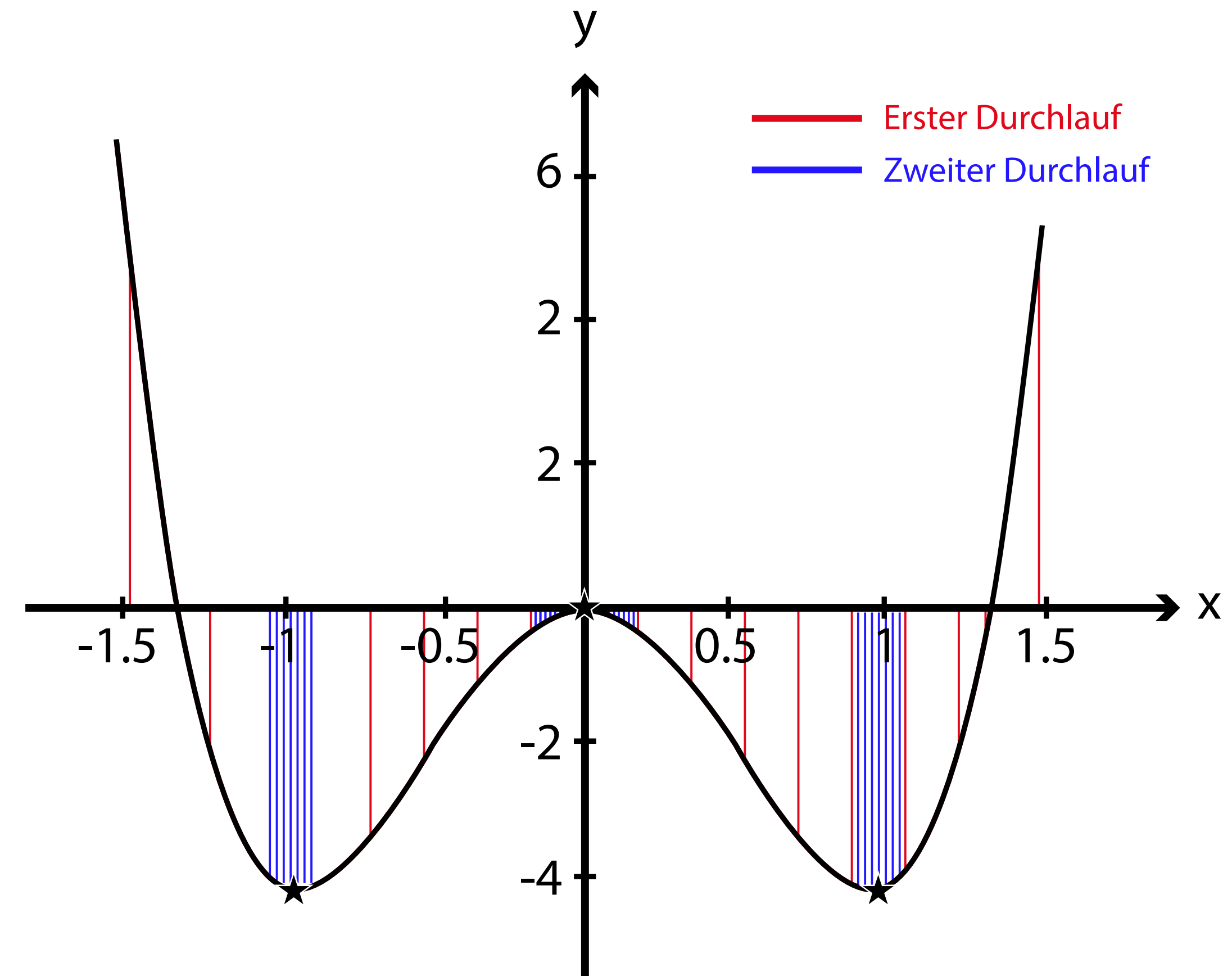
Die Konsistenz ist meistens perfekt gegeben.



Nachtrag zu Brute Force

Wir könnten den grid search sogar noch durch eine adaptive Schrittweite verbessern.

Wir tasten die Funktion zunächst mit einem groben Gitter ab und tasten in einem zweiten Durchgang die "vielversprechenden Stellen" mit einem feineren Gitter ab.

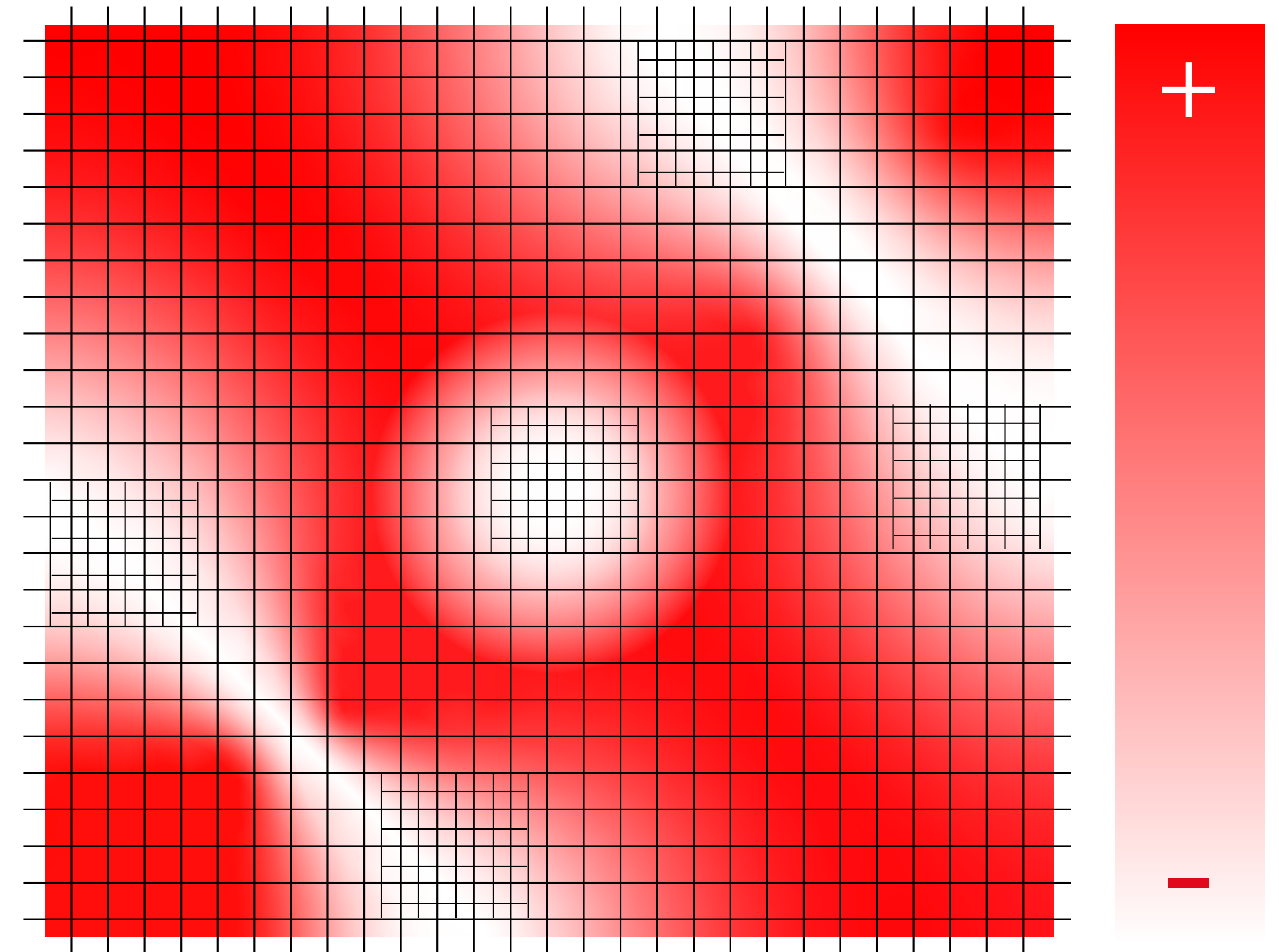


Nachtrag zu Brute Force

Auch die Verallgemeinerung auf mehrdimensionale Funktionen, d. h. Skalarfelder und Vektorfelder, ist sehr einfach.

Um mit größeren Datenmengen klar zu kommen, müssen wir uns dennoch von der brute force Vorgehensweise verabschieden.

Im nächsten zwei Kapiteln lernen wir zwei Algorithmen kennen, mit denen wir Null- und Extremstellen ohne brute force berechnen können.



Es gibt viele numerische Verfahren, mit denen wir Gleichungen lösen können.

Ein besonders bekanntes ist das Newtonverfahren für Gleichungen der Form:

$$f(x) = 0, \text{ mit } f: \mathbb{R} \rightarrow \mathbb{R}$$

Das Verfahren findet die Nullstellen einer gegebenen eindimensionalen Funktion.

$$\begin{aligned} f(x) &= x^4 - 2x^2 = 0 && | x^2(\dots) \\ \Leftrightarrow & \underbrace{x^2(x^2 - 2)} = 0 \\ & \quad \quad \quad \downarrow x^2 = 2 && | (\dots)^{0.5} \\ & \quad \Rightarrow x_{1,2} = \pm \sqrt{2} \\ & \quad \quad \downarrow \\ & \Rightarrow x_3 = 0 \end{aligned}$$

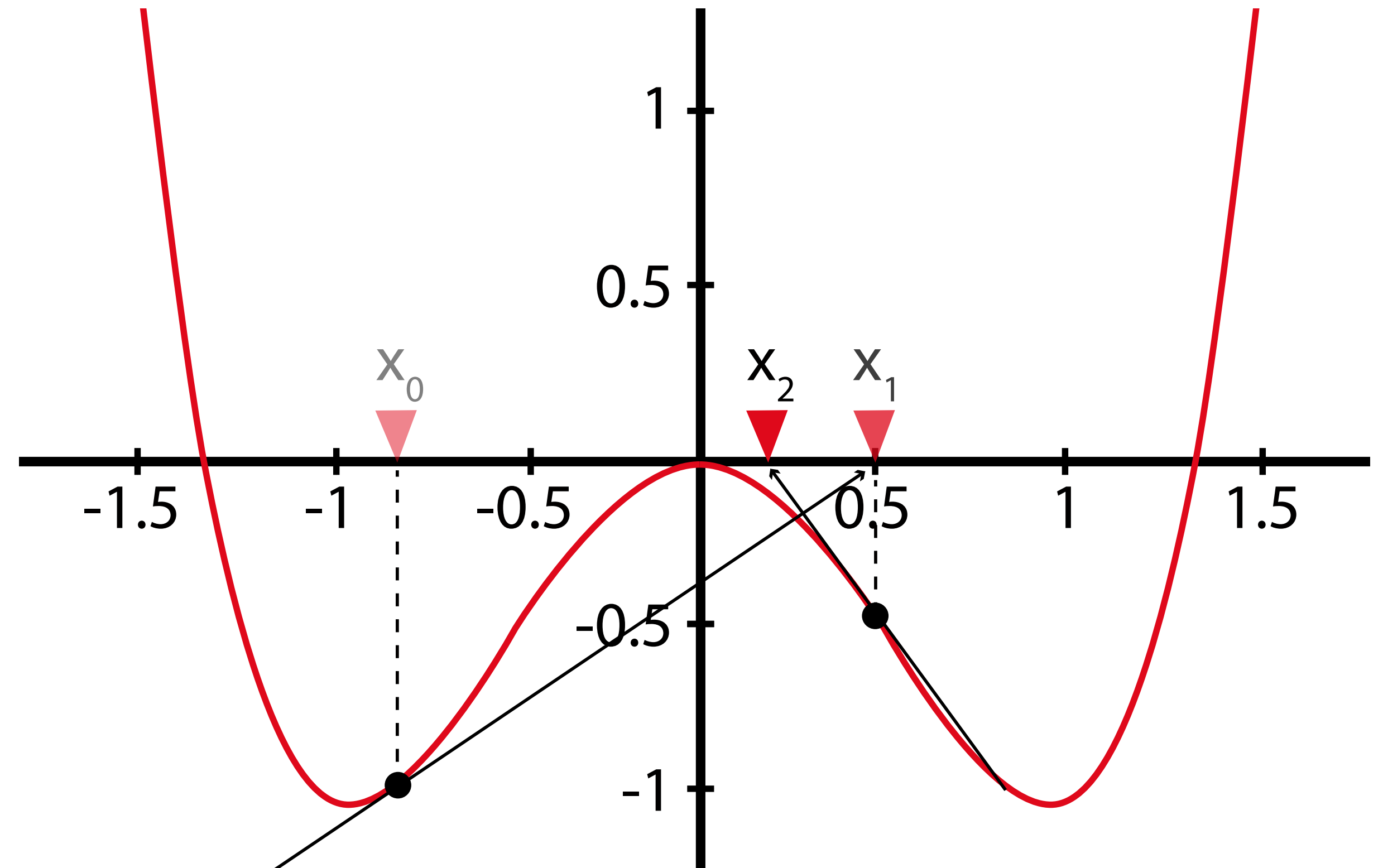
A graph of a function $f(x)$ (red curve) on a Cartesian coordinate system. The x-axis ranges from -1.5 to 1.5, and the y-axis ranges from -1.5 to 1.5. The function has three real roots. Three points are marked on the curve: x_0 at approximately $x = -0.8$, x_1 at approximately $x = 0.4$, and x_2 at approximately $x = 0.1$. Tangent lines are drawn from each point to the x-axis, with the next iteration point marked on the x-axis: x_1 is the root of the tangent at x_0 , x_2 is the root of the tangent at x_1 , and x_0 is the root of the tangent at x_2 . This illustrates the convergence of the Newton-Raphson method towards the root x_0 .

Newtonverfahren

Grundidee Wir beginnen bei einem Startpunkt x_0 und bilden den Schnittpunkt einer dort anliegenden Tangente mit der x-Achse, um den nächsten Punkt x_1 zu finden, usw.

Mathematisch betrachtet führen wir folgenden Schritt wiederholt durch, bis eine Genauigkeit $|f(x_n)| < \varepsilon$ erreicht wird.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



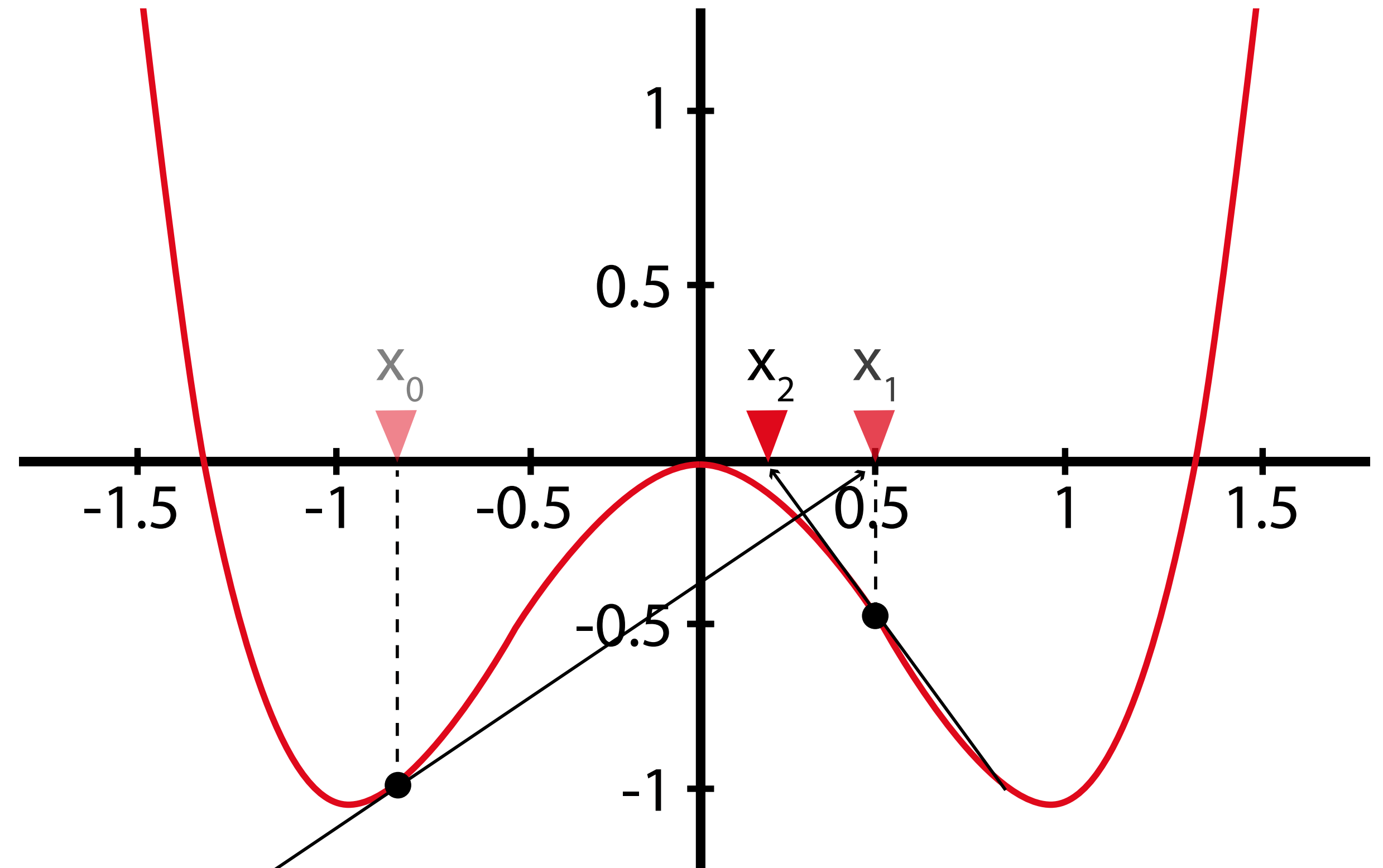
Newtonverfahren

$$f(x) = x^4 - 2x^2 \quad f'(x) = 4x^3 - 4x$$

Startpunkt $f(-0.9) = -0.9639$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = -0.9 - \frac{-0.9639}{0.684} = 0.509$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 0.509 - \dots$$



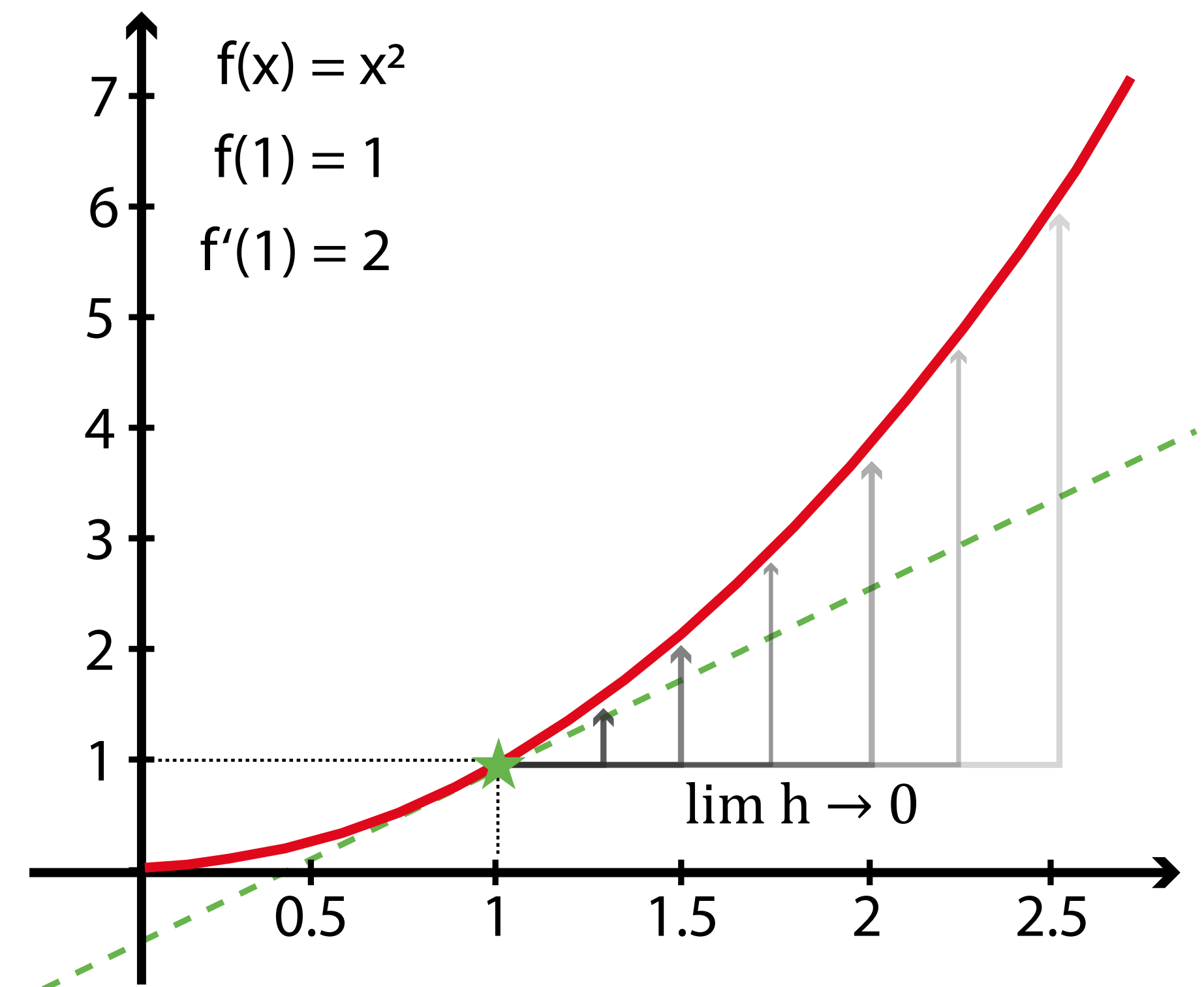
Newtonverfahren

Die Iteration über die Gleichung von Thomas Simpson ...

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

... ist sehr einfach zu implementieren, aber woher erhalten wir die Werte für die Ableitung? Auch diese müssen wir ggf. numerisch bestimmen!

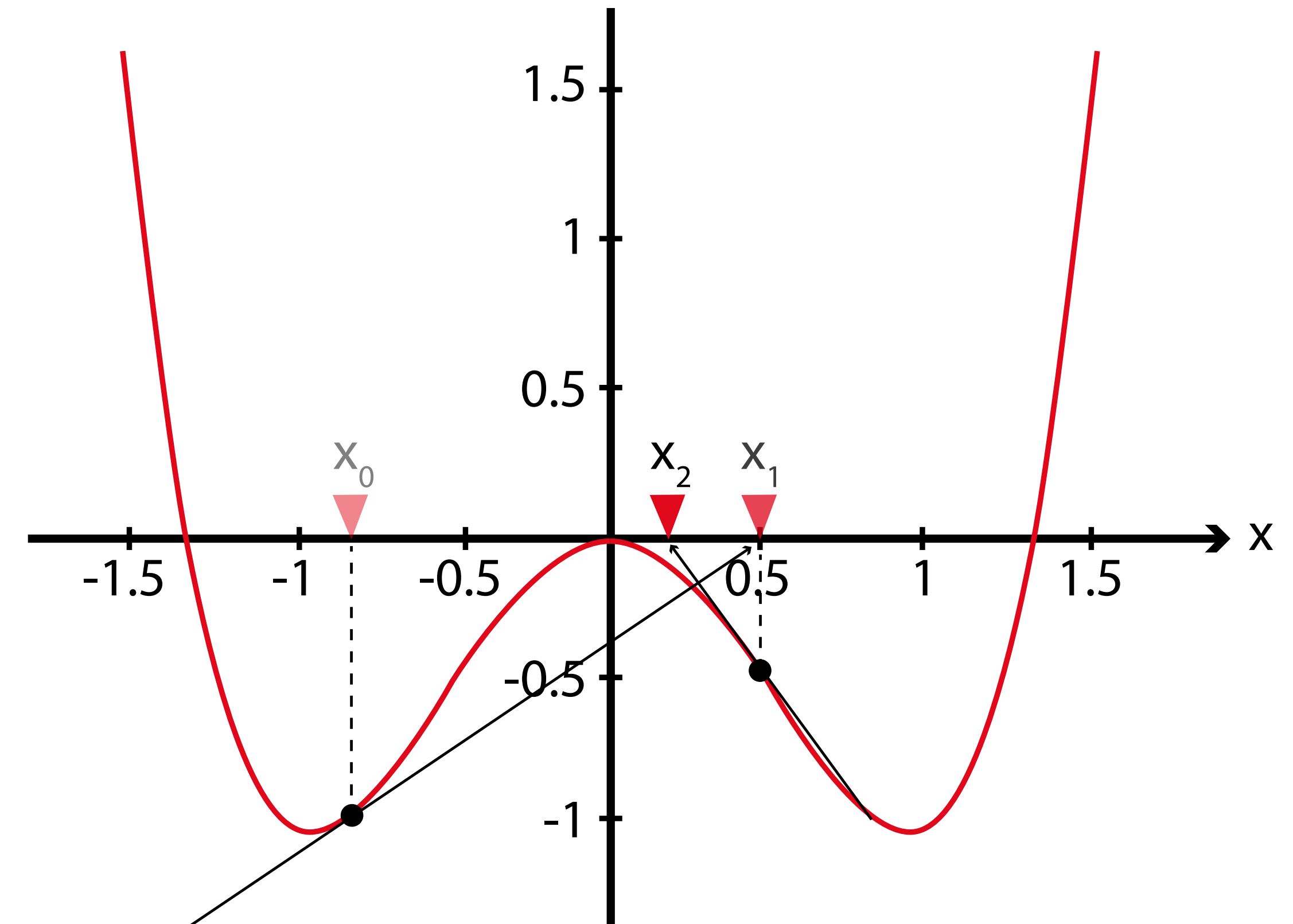
Ein Verfahren dazu kennen wir aus der Schule: das Steigungsdreieck mit endlicher Breite.



Newtonverfahren

Schau dir den Beispielcode "Newton Roots Hardcoded" zum Newtonverfahren an.

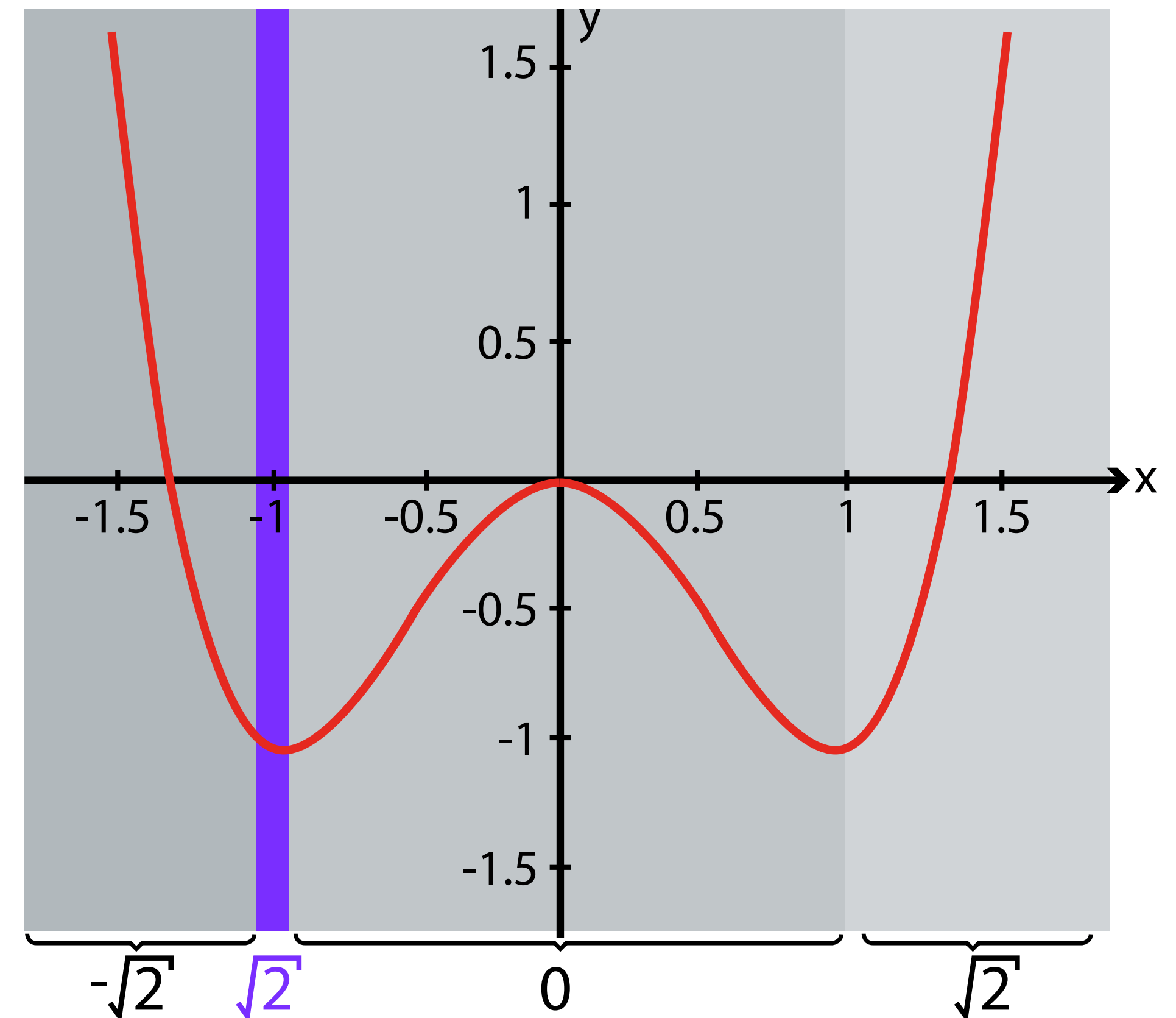
- Entwickle das Skript weiter, sodass es die Ableitung numerisch berechnen kann und nicht mehr auf eine hard-coded Ableitung angewiesen ist.
- Entwickle das Skript weiter, sodass es in der Lage ist mehrere Nullstellen zu finden.
- Implementiere einen Sicherheitsmechanismus, der die Suche nach spätestens $n=100$ Iterationen abbricht.



Newtonverfahren

Wir stellen fest, dass das Konvergenzverhalten nicht für alle Eingabeparameter intuitiv ist.

Im rechts gezeigten Schaubild sehen wir eine Konstellation, bei der ausgehend vom Startwert $x_0 = -1$ eine Nullstelle bei plus Wurzel 2 gefunden wird, obwohl der Startwert von zwei anderen Nullstellen "umzingelt" ist.



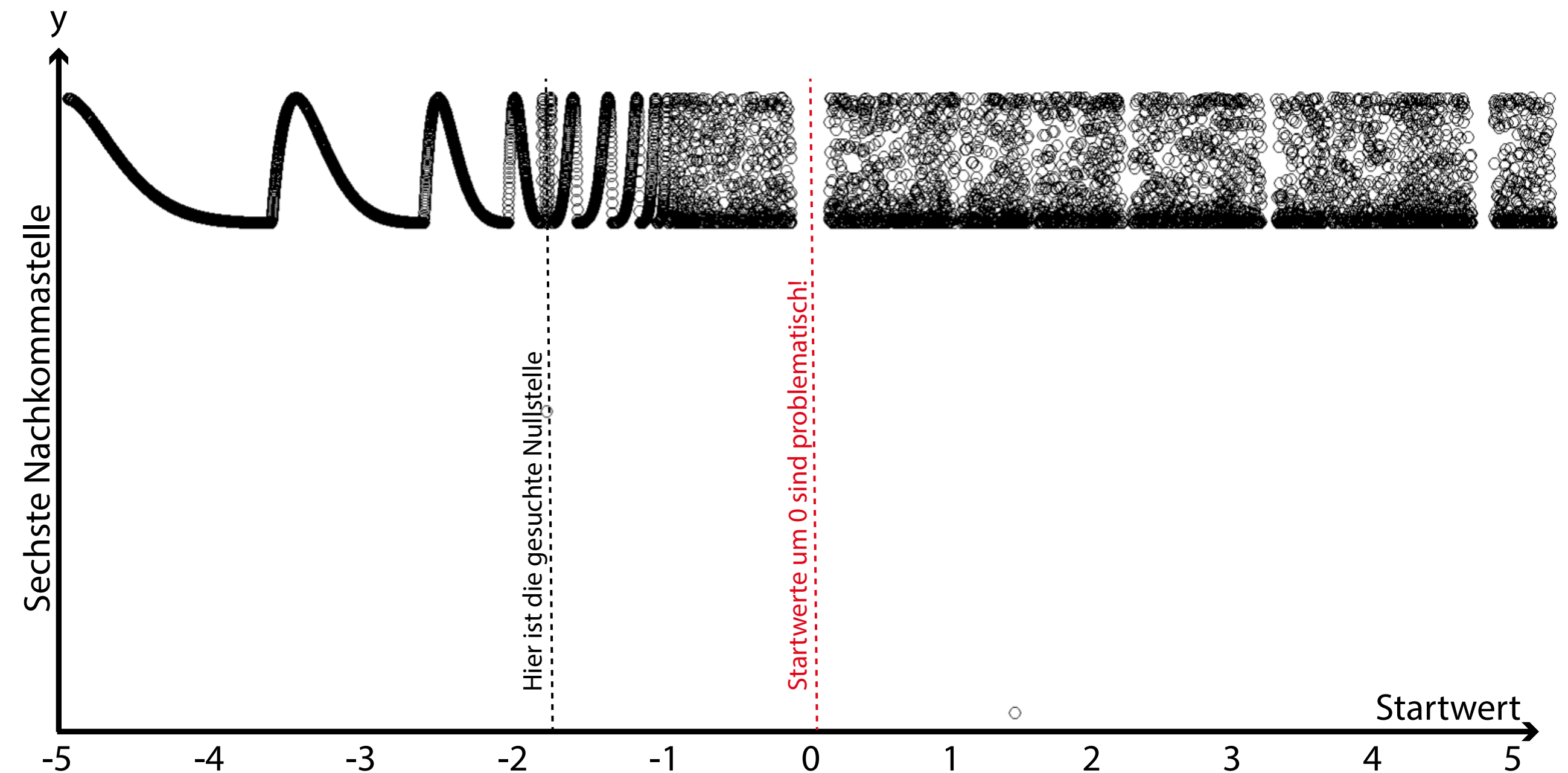
Newtonverfahren

Es sind auch Konstellationen denkbar, bei denen keine Nullstelle gefunden wird.

Das Newtonverfahren versagt, wenn es Stellen mit $f'(x)=0$ auswertet oder in eine Endlosschleife von Stellen verfällt.

Rechts sehen wir als Beispiel dazu das Konvergenzverhalten für die Funktion:

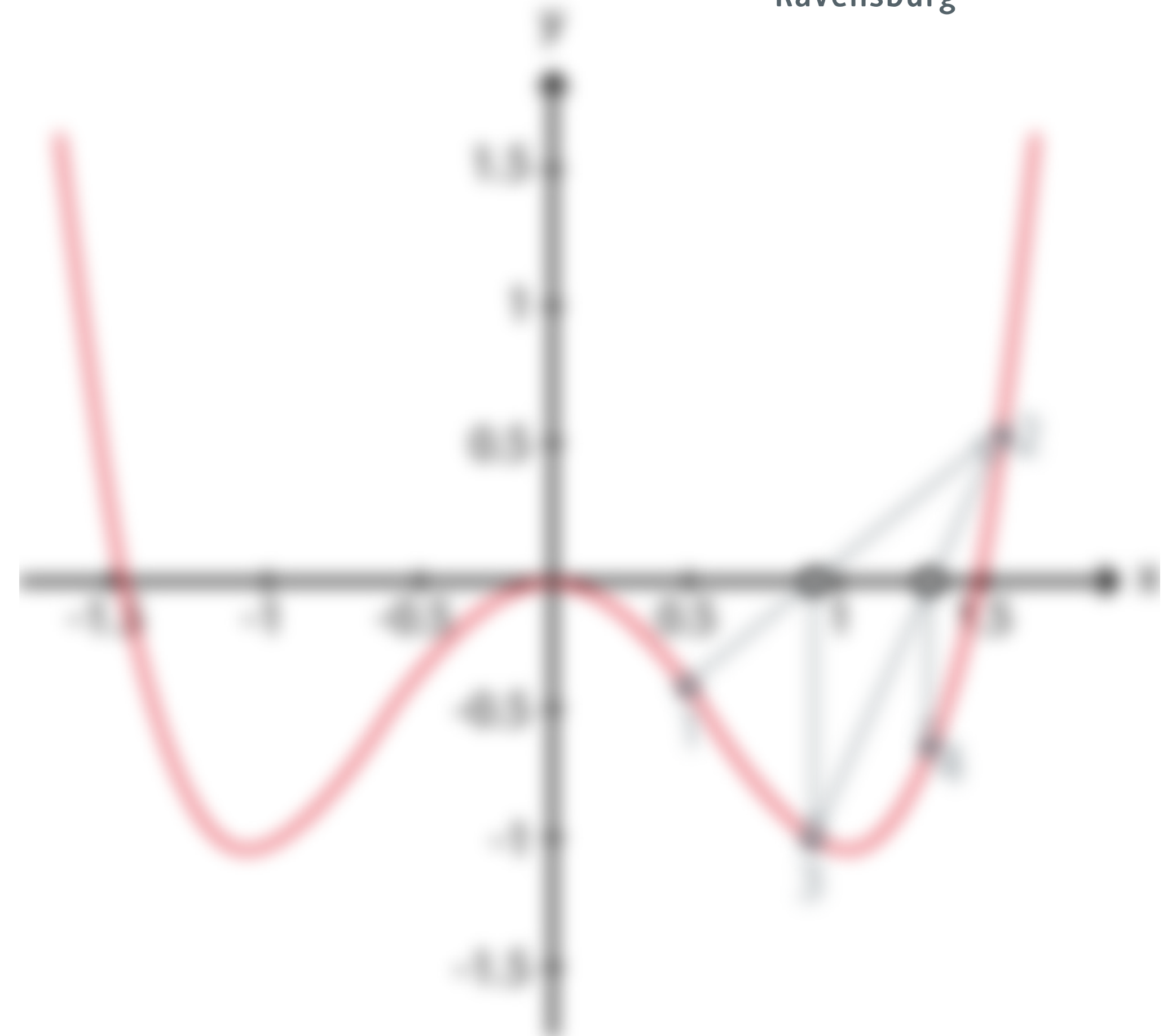
$$f(x) = x^3 - 2x + 2$$



Sekantenverfahren

Das Sekantenverfahren ist eine Alternative zum Newtonverfahren.

Schaue dir den Beispielcode "Secant Roots" an und versuche herauszufinden, wie das Sekantenverfahren vorgeht!



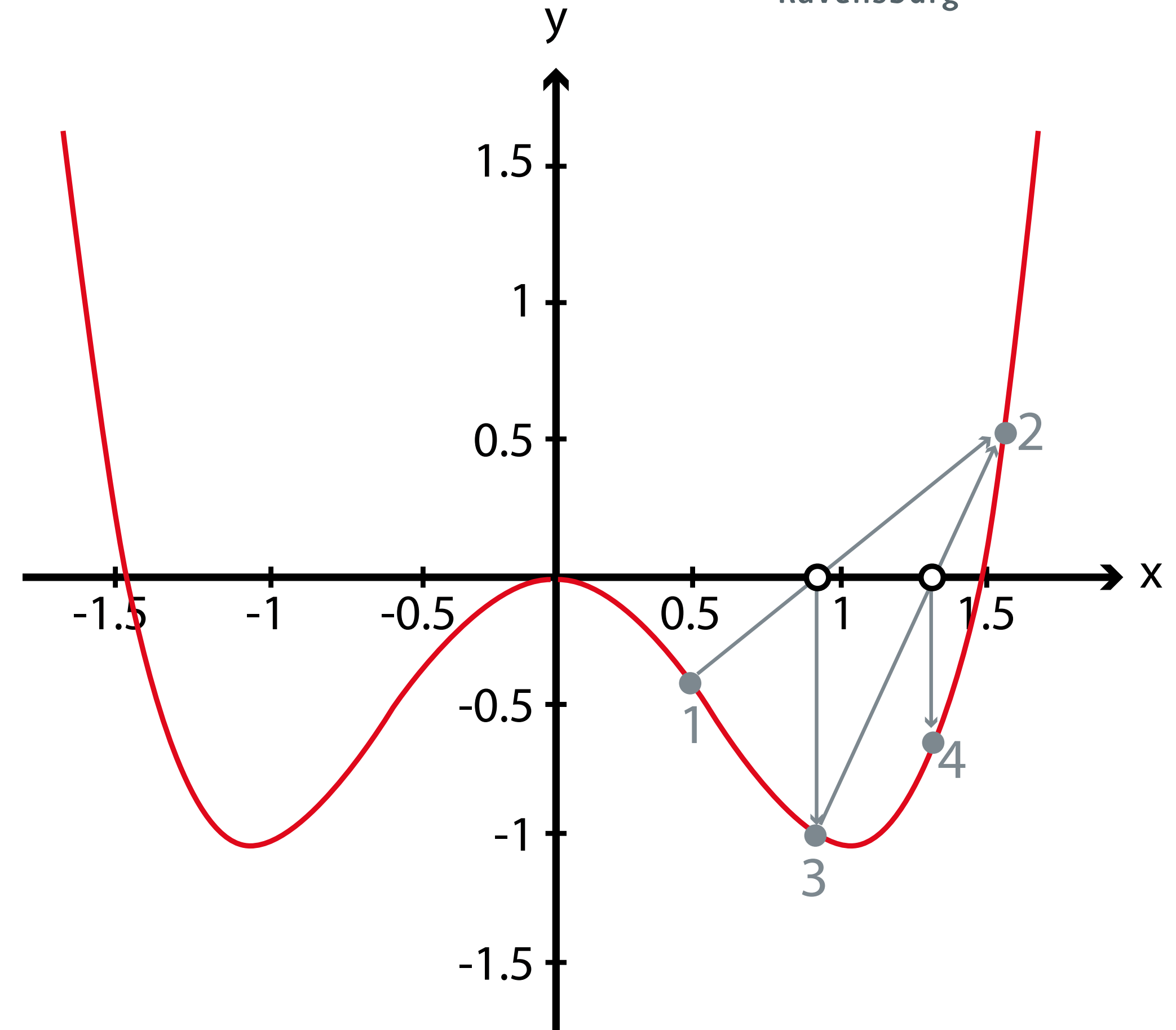
Sekantenverfahren

Die Grundidee ist dem Newtonverfahren recht ähnlich. Anstelle von einem Punkt arbeiten wir aber mit zwei Punkten und deren Verbindung.

Wir benötigen zwei Startwerte und die Iterationsformel ist gegeben durch:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

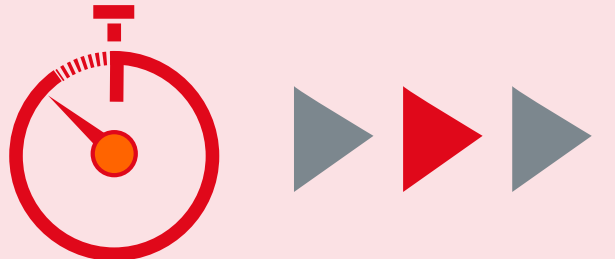
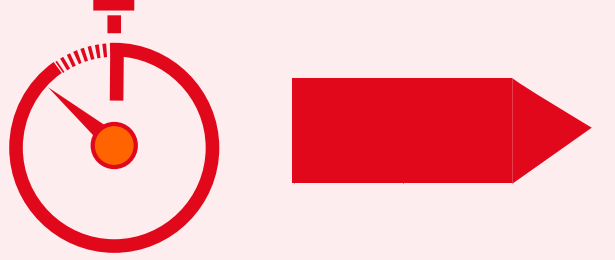
Wir vermeiden die Berechnung/Näherung der Ableitung!



Performancekennzahlen

Wir haben bei unserem brute force Algorithmus bereits die Laufzeit und die Zeitkomplexität kennengelernt.

Bei den iterativen Verfahren können wir auch die **Konvergenzordnung** und die **Zeit pro Iteration** untersuchen.

	Zeitkomplexität Wie stark skaliert die Laufzeit mit der Menge der Eingabedaten?
	Konvergenzgeschwindigkeit Wie schnell erhöht sich die Genauigkeit pro Iterationsschritt
	Laufzeit einer Iteration Wie lange benötigt der Algorithmus für eine Iteration?
	Tatsächliche Laufzeit Wie lange benötigt der Algorithmus bis zur Konvergenz?

Performancekennzahlen

Für die Konvergenzordnung betrachten wir den Grenzwert des rechts gezeigten Quotienten.

Ist $q > 1$ und $\mu \in (0, \infty)$ dann konvergiert der Algorithmus überlinear mit der Konvergenzordnung q . Jeder Schritt erhöht die Anzahl genauer Stellen um den Faktor q .

Ist $q = 1$ und $\mu \in (0, 1)$ dann konvergiert der Algorithmus linear mit Konvergenzordnung 1. Jeder Schritt erhöht die Anzahl genauer Stellen um einen gewissen Betrag.

Wie weit liegt der Algorithmus daneben?

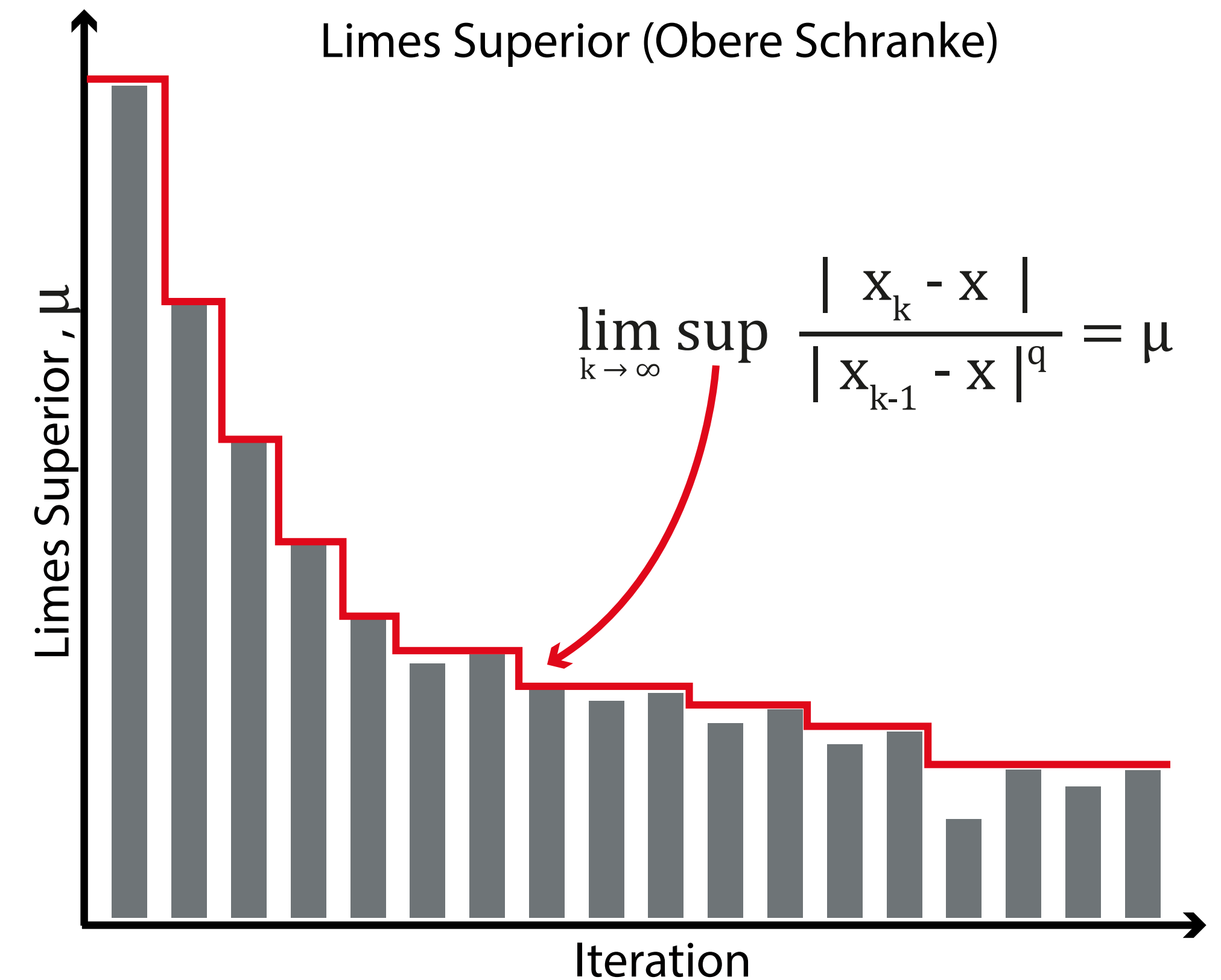
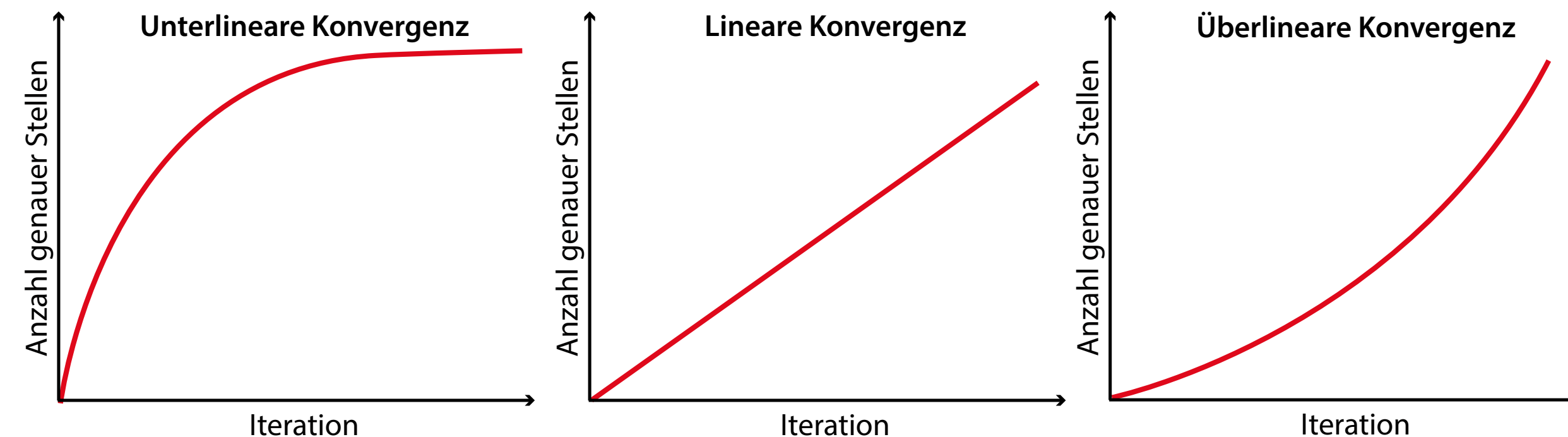
$$\lim_{k \rightarrow \infty} \sup \frac{|x_k - x|}{|x_{k-1} - x|^q} = \mu$$

Wie weit lag er einen Schritt vorher daneben?

Performancekennzahlen

Den Limes superior trägt der Sache Rechnung, dass der Quotient nicht streng monoton fallend ist.

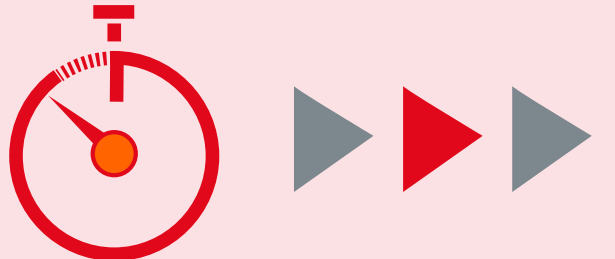
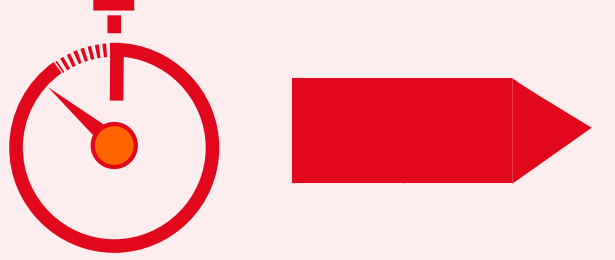
Jeg größer die Konvergenzordnung, umso weniger Schritte werden bis zum Erreichen einer gewissen Genauigkeit benötigt.



Performancekennzahlen

Vorsicht! Eine höhere Konvergenzordnung deutet nur auf eine kleinere Anzahl an Iterationsschritten hin.

Je nachdem wie lange diese Schritte dann dauern, kann ein Algorithmus mit höherer Konvergenzordnung trotzdem langsamer sein als ein Algorithmus mit niedrigerer Konvergenzordnung!

	Zeitkomplexität Wie stark skaliert die Laufzeit mit der Menge der Eingabedaten?
	Konvergenzgeschwindigkeit Wie schnell erhöht sich die Genauigkeit pro Iterationsschritt
	Laufzeit einer Iteration Wie lange benötigt der Algorithmus für eine Iteration?
	Tatsächliche Laufzeit Wie lange benötigt der Algorithmus bis zur Konvergenz?



Suche nach Extremstellen

Mit dem Newton- und dem Sekantenverfahren können auch Extremstellen berechnet werden.

Dazu berechnen wir die erste und zweite Ableitung der gewünschten Funktion numerisch und führen die Iteration auf der ersten Ableitung durch.

	Newtonverfahren	Sekantenverfahren
Nullstellen	$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}$	$X_{n+1} = X_n - \frac{X_n - X_{n-1}}{f(X_n) - f(X_{n-1})} f(X_n)$
Extremstellen	$X_{n+1} = X_n - \frac{f'(X_n)}{f''(X_n)}$	$X_{n+1} = X_n - \frac{X_n - X_{n-1}}{f'(X_n) - f'(X_{n-1})} f'(X_n)$

Suche nach Extremstellen

Wir können unsere Verfahren auf Skalarfelder generalisieren.

$$f: \mathbb{R}^n \rightarrow \mathbb{R}$$

Für das Newtonverfahren erhalten wir die rechts gezeigten Iterationsvorschriften.

Nullstellen

$$\vec{X}_{n+1} = \vec{X}_n - \frac{f(\vec{X})}{|\nabla f|^2} \nabla f$$

Extremstellen

$$\vec{X}_{n+1} = \vec{X}_n - H^{-1} \cdot \nabla f$$

Newtonverfahren

Wir benötigen ...

$$\nabla f = \left(\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \quad \dots \right)^T$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \dots \\ \frac{\partial^2 f}{\partial x_1 \partial x_3} & \dots & \dots \end{pmatrix}$$

Suche nach Extremstellen

Die numerische Berechnung des Gradienten von f ist einfach.

Wir nähern alle partiellen Ableitungen numerisch mit einem endlich breiten Steigungsdreieck für die entsprechende Dimension.

Bei der Hesse-Matrix wird es dagegen deutlich schwieriger! Das wollen wir eigentlich vermeiden.

Nullstellen

$$\vec{X}_{n+1} = \vec{X}_n - \frac{f(\vec{X})}{|\nabla f|^2} \nabla f$$

Extremstellen

$$\vec{X}_{n+1} = \vec{X}_n - H^{-1} \cdot \nabla f$$

Newtonverfahren

Wir benötigen ...

$$\nabla f = \left(\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \quad \dots \right)^T$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \dots \\ \frac{\partial^2 f}{\partial x_1 \partial x_3} & \dots & \dots \end{pmatrix}$$

Suche nach Extremstellen

Für die Suche nach Extremstellen von Skalarfeldern können wir das Newton- oder auch Sekantenverfahren verwenden.

Gängiger im mehrdimensionalen Raum ist der gradient descent mit Iterationsvorschrift:

$$\vec{x}_{n+1} = \vec{x}_n - \alpha \nabla f$$

Nullstellen

$$\vec{x}_{n+1} = \vec{x}_n - \frac{f(\vec{x})}{|\nabla f|^2} \nabla f$$

Extremstellen

$$\vec{x}_{n+1} = \vec{x}_n - H^{-1} \cdot \nabla f$$

Newtonverfahren

Wir benötigen ...

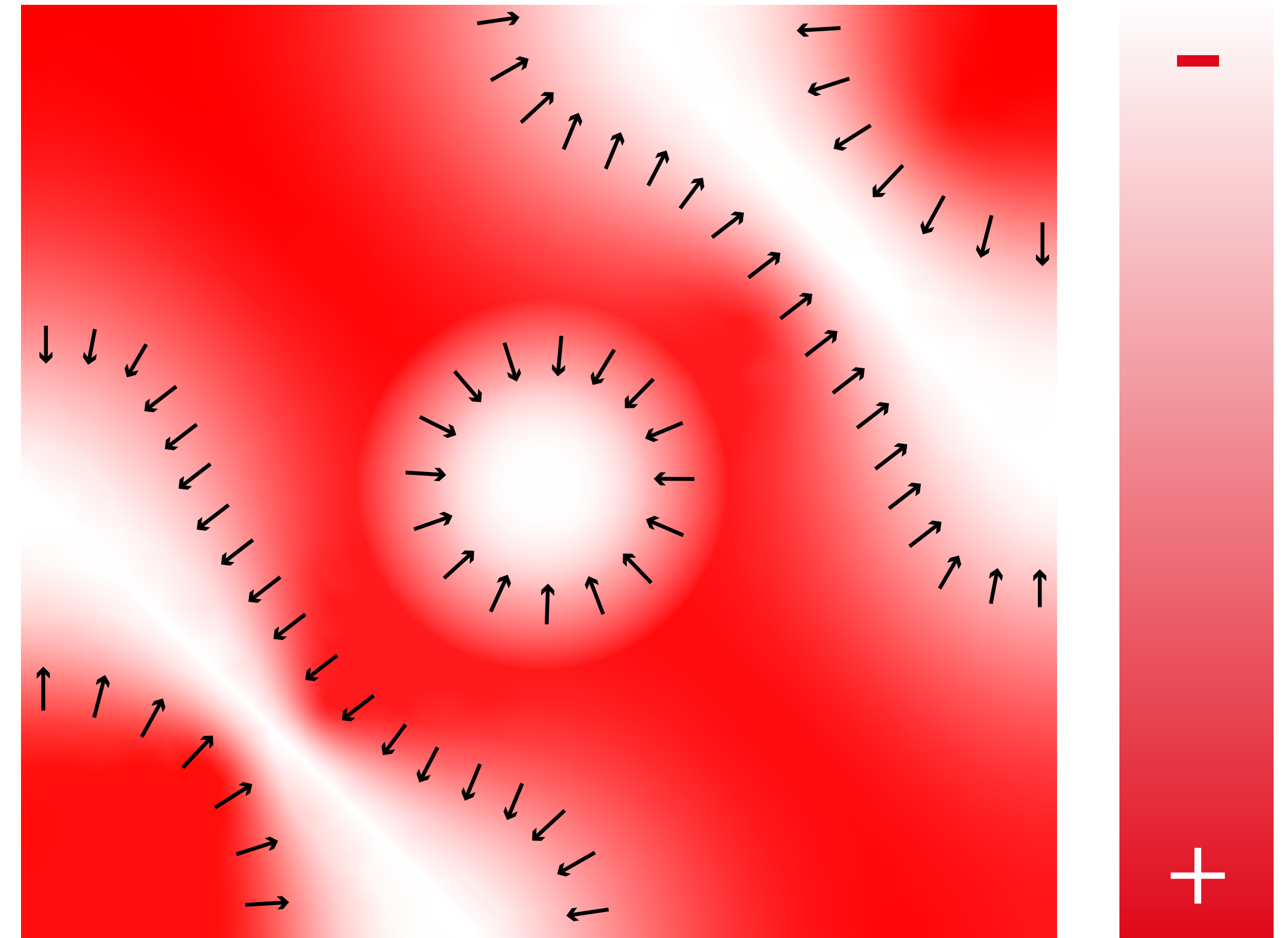
$$\nabla f = \left(\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \quad \dots \right)^T$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \dots \\ \frac{\partial^2 f}{\partial x_1 \partial x_3} & \dots & \dots \end{pmatrix}$$

Suche nach Extremstellen

Hier schließt sich der Kreis von Analysis, linearer Algebra, Numerik und Data Science.

Die Schätzung eines Modells mit gradient descent ist nichts anderes als eine numerische Suche nach dem Minimum einer Funktion, die von mehreren Variablen abhängt!



Suche nach Extremstellen

Schau dir den halb fertigen Beispielcode zum gradient descent Verfahren an.

- Nähere den Gradienten über die numerische Berechnung der partiellen Ableitungen
- Implementiere die gradient descent Iterationsvorschrift mit einer variablen Learning rate.
- Nutze die bereitgestellte Visualisierungsfunktion, um die Bedeutung der Learning rate herauszufinden.

$$f(x,y) = | 2 \sin(x) | + | 2 \sin(y) | + \sqrt{x^2 + y^2}$$

Differenzialgleichungen

Als Nächstes wollen wir gewöhnliche Differenzialgleichungen numerisch lösen. Zur Erinnerung:

Differenzialgleichungen sind Gleichungen, die eine Funktion mit mindestens einer ihrer Ableitungen in Verbindung setzt.

Die Lösungen sind keine Zahlenwerte, sondern Funktionen $y(x)$ für welche die Differenzialgleichung erfüllt ist.



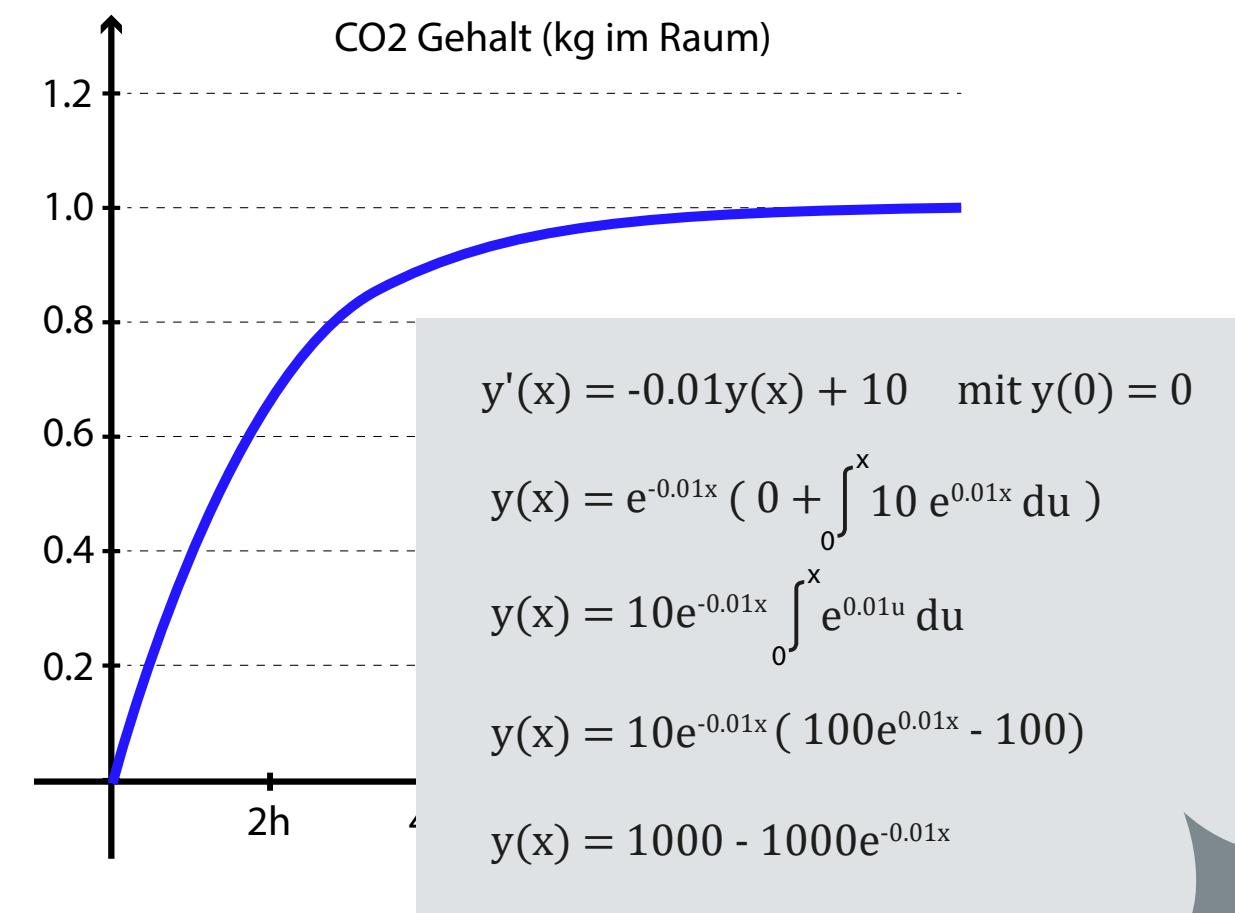
Differenzialgleichungen

Für lineare DGL erster Ordnung haben wir in der Analysis Lösungsformeln kennengelernt.

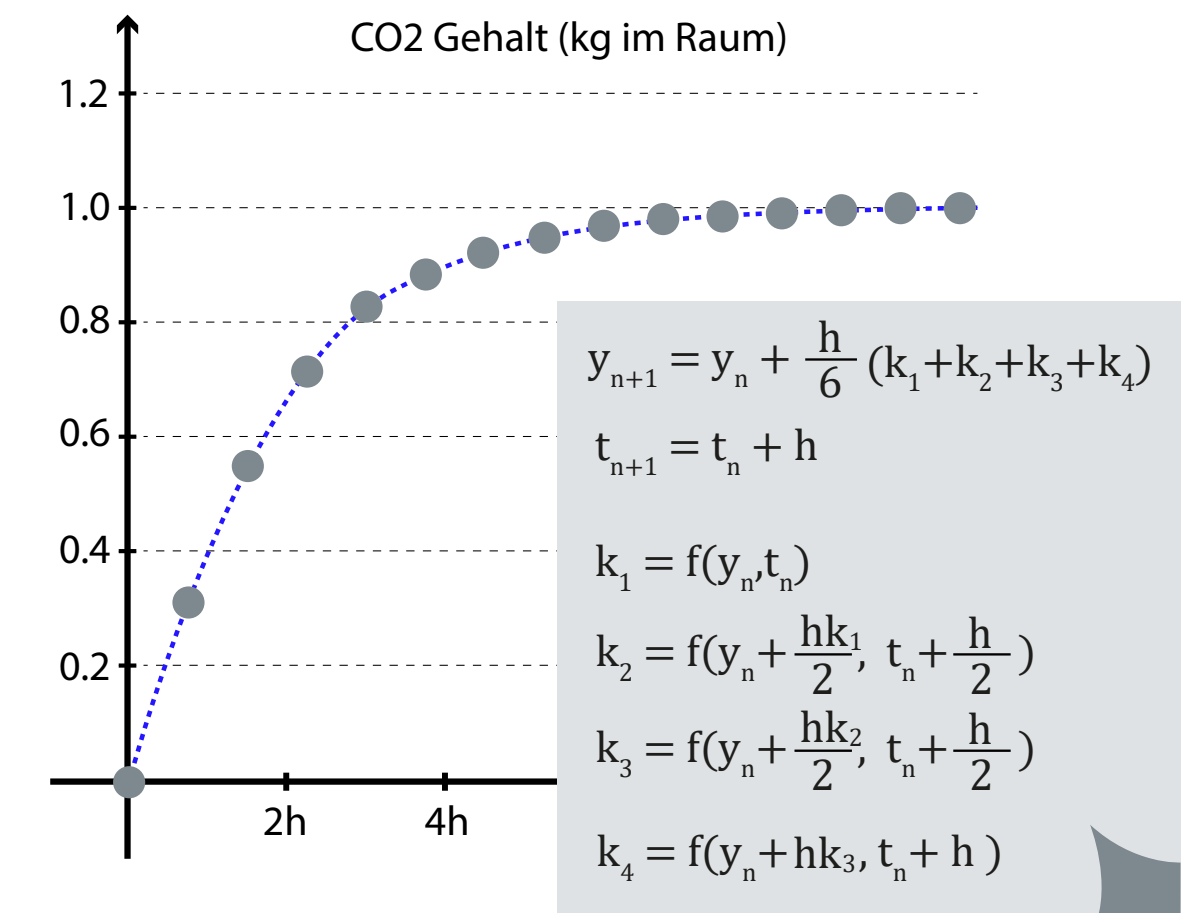
Bei nicht linearen DGL haben wir versucht, die Veränderlichen zu trennen und dann zu integrieren.

In der Numerik lernen wir das Eulerverfahren und die Familie der Runge-Kutta-Verfahren kennen.

Analytische Lösung



Numerische Lösung



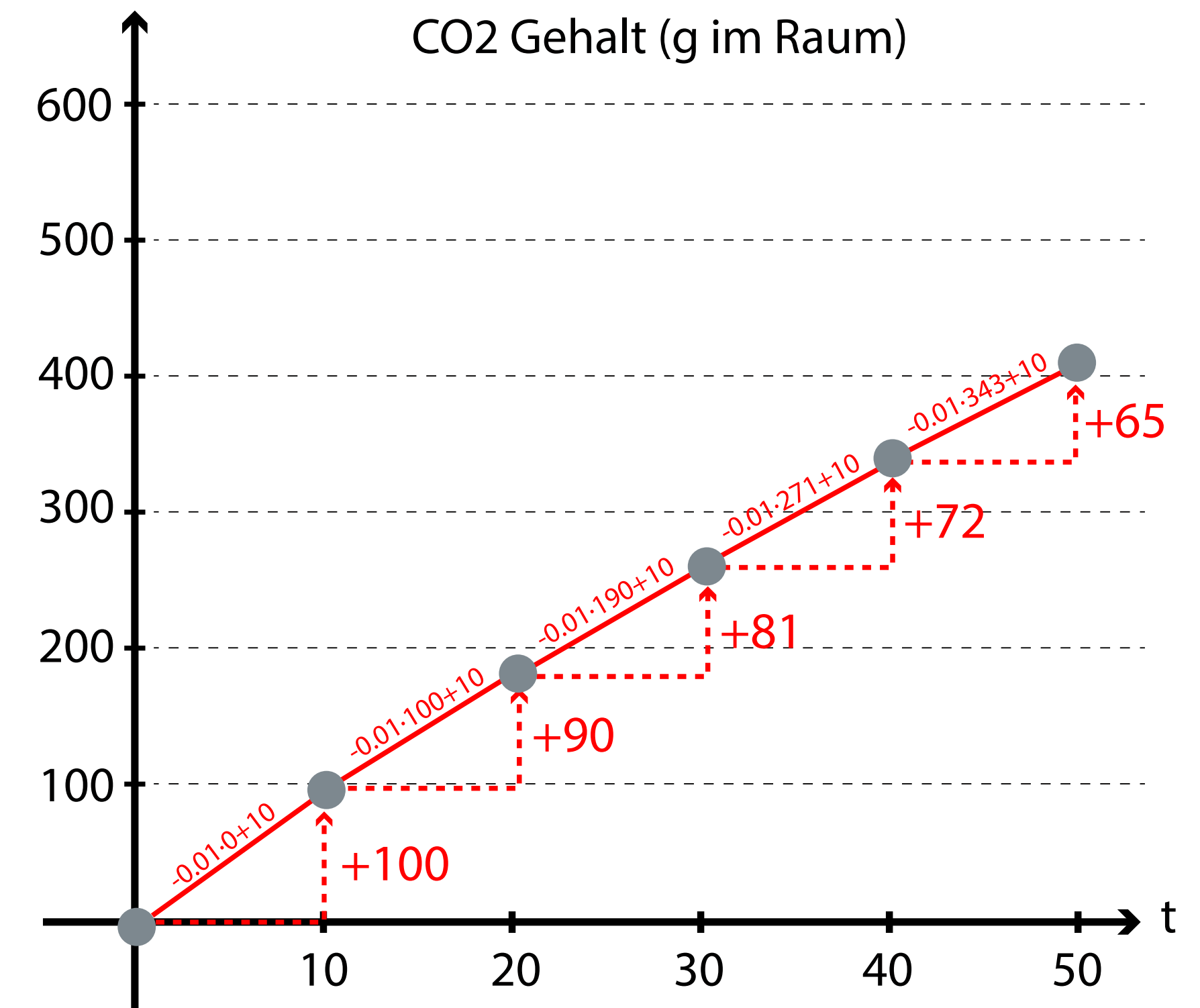
Eulerverfahren

Um das Eulerverfahren intuitiv zu verstehen, betrachten wir folgende Differenzialgleichung:

$$y'(x) = -0.01 \cdot y(x) + 10 \text{ mit } y(0) = 0$$

Wir kennen von Anfang an einen Punkt der gesuchten Funktion und wissen auch, dass sie dort eine Steigung von 10 hat!

Idee Konstruiere die gesuchte Funktion Schritt für Schritt aus der bekannten Steigung, ausgehend vom bekannten Punkt.



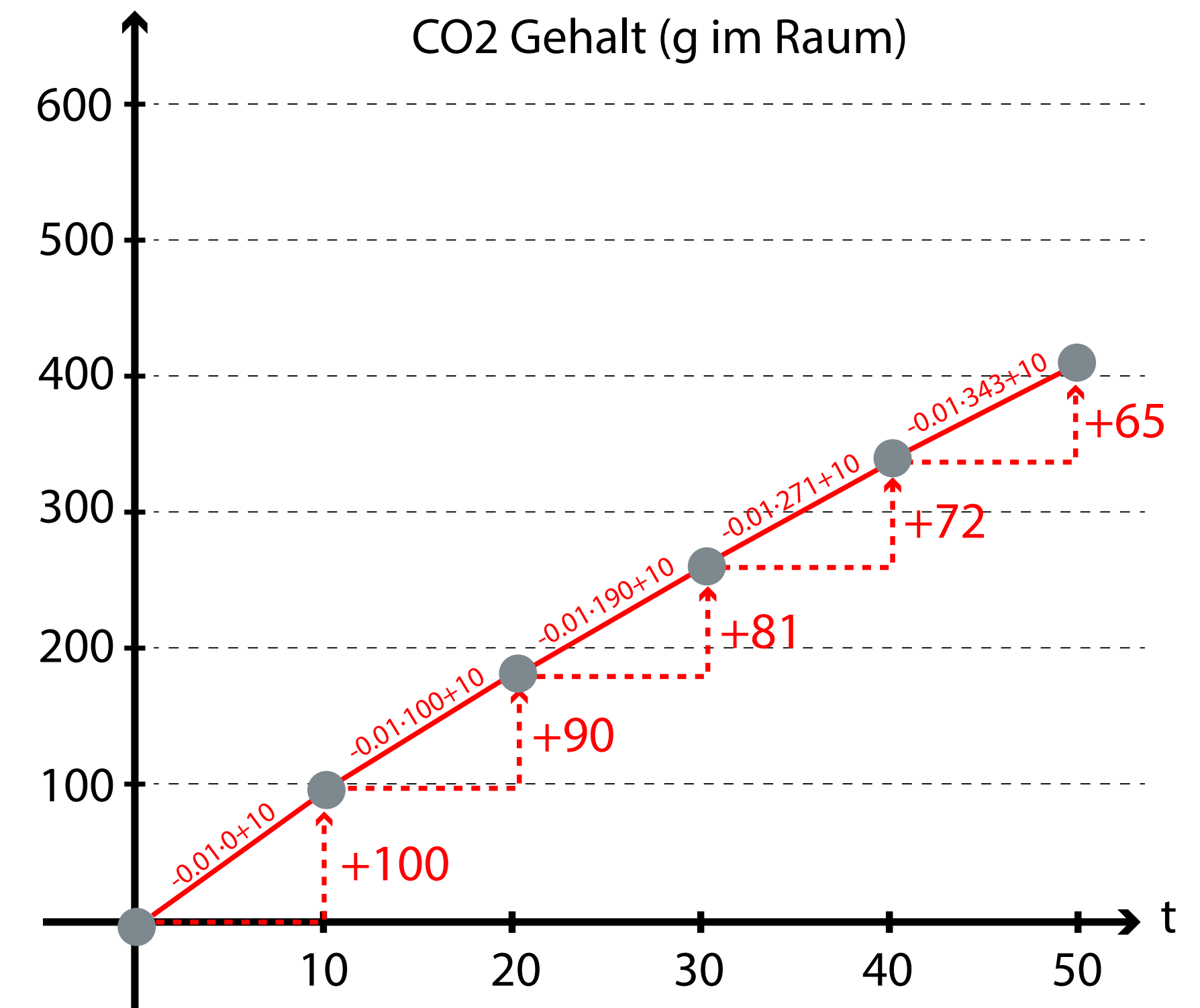
Eulerverfahren

Mit 10er Schritten erhalten wir das Schaubild rechts.

$$y'(x) = -0.01 \cdot y(x) + 10 \text{ mit } y(0) = 0$$

Der zweite Punkt liegt bei $x=10$ und hat den Wert 100, da wir von $y(0)=0$ bis $y(10)=100$ mit der Steigung 10 rechnen.

Der dritte Punkt liegt bei $x=20$ und hat den Wert 190, da wir von $y(10)=100$ bis $y(20)=190$ mit der Steigung 9 rechnen.



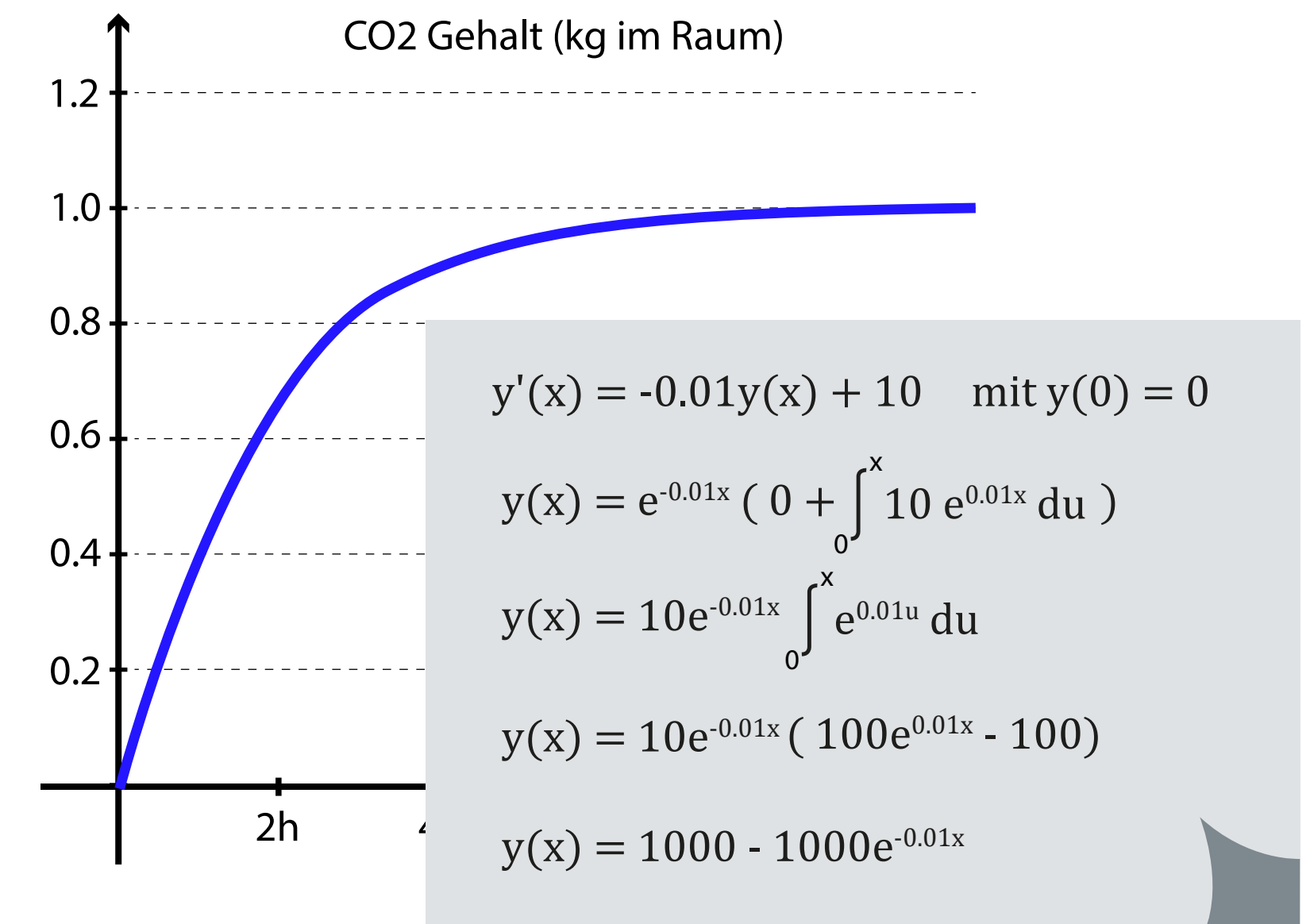
Eulerverfahren

Wie genau ist das Eulerverfahren? Wie stark ist die Abweichung zur analytischen Lösung?

Schau dir die Implementierung des Euler Verfahrens in "Euler Basic" an und entwickle den Code weiter!

Ergänze eine Messung des absoluten und relativen Fehlers, der durch die Verwendung des Eulerverfahrens entsteht. Verwende dabei die rechts gezeigte analytische Lösung!

Analytische Lösung

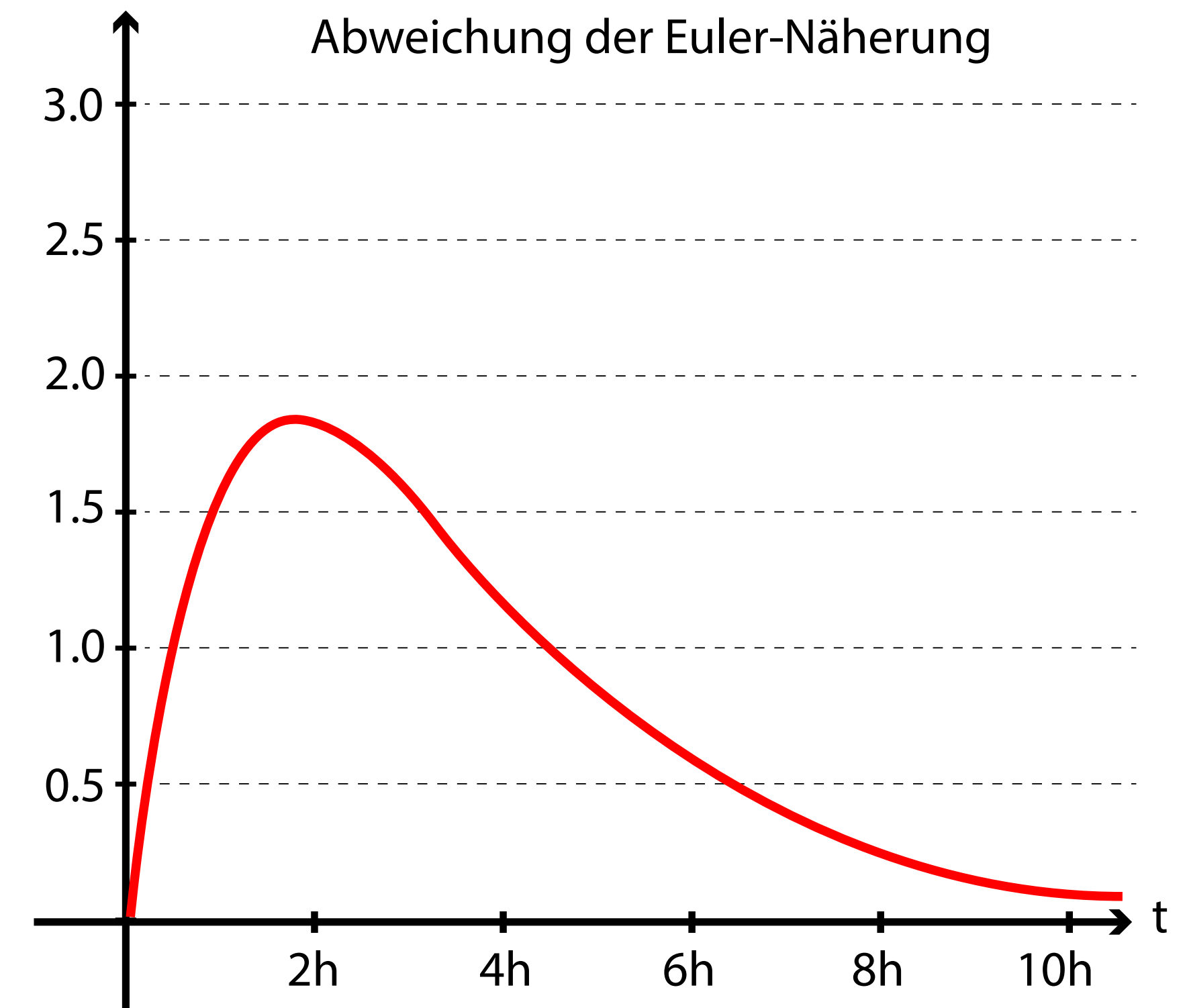


Eulerverfahren

Die Abweichung zwischen den Werten des Eulerverfahrens und den exakten Werten steigt zunächst auf maximal 0.184 an und fällt danach wieder ab.

Der Wert, gegen den die Funktion für t gegen unendlich konvergiert wird um 0.007 verfehlt.

Da die Funktion eine CO₂ Menge im Raum von bis zu 1000g beschreibt, sind diese Abweichung irrelevant. Perfekt ist die Näherung allerdings auch nicht.

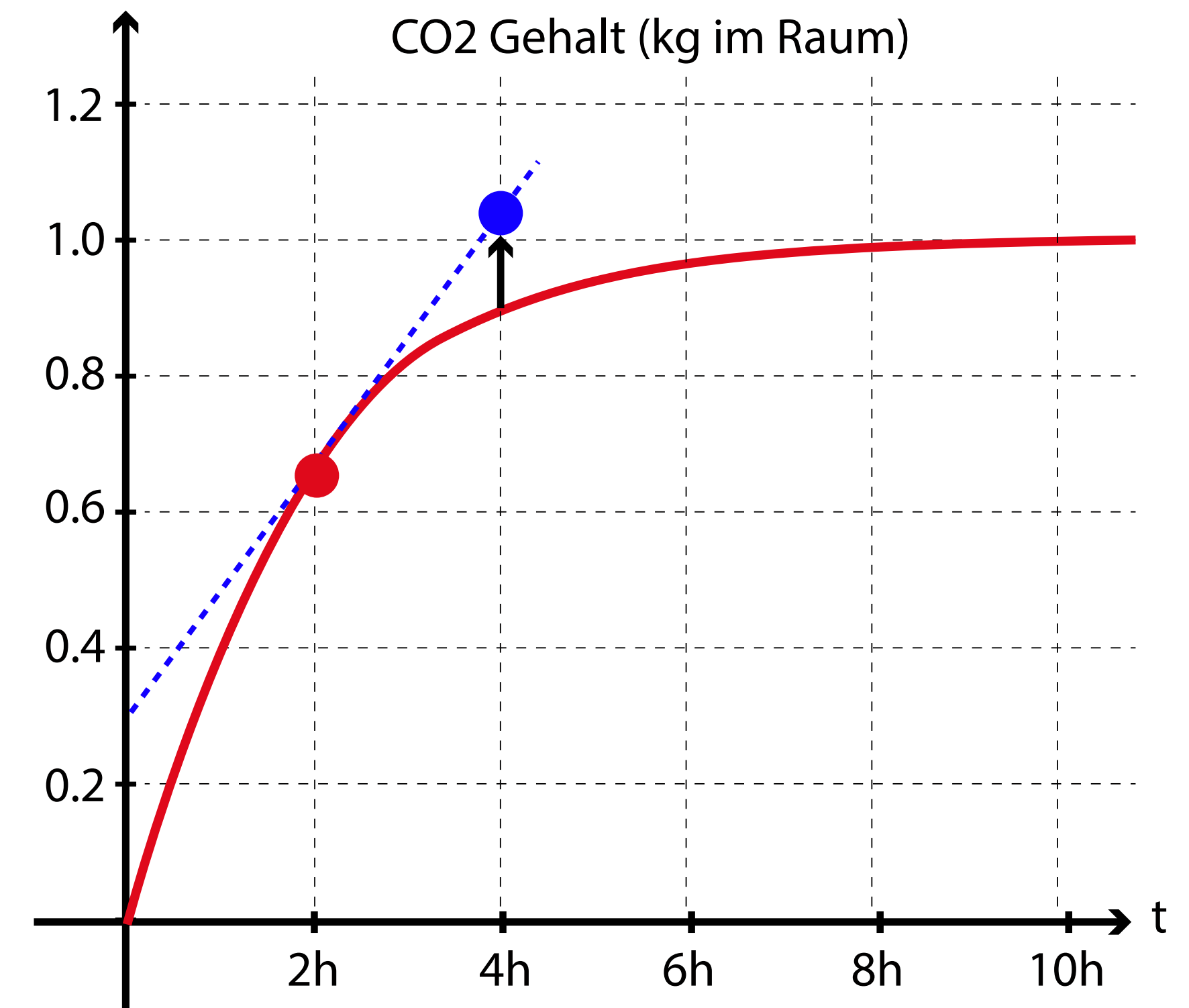


Eulerverfahren

Der Grund für die Abweichung ist die Annahme, dass die Steigung innerhalb der Schrittweite konstant bleibt.

Je größer die Schrittweite umso größer der Fehler, der entsteht.

Je größer die Änderung der Ableitung innerhalb der Schrittweite, umso größer der Fehler, der entsteht.

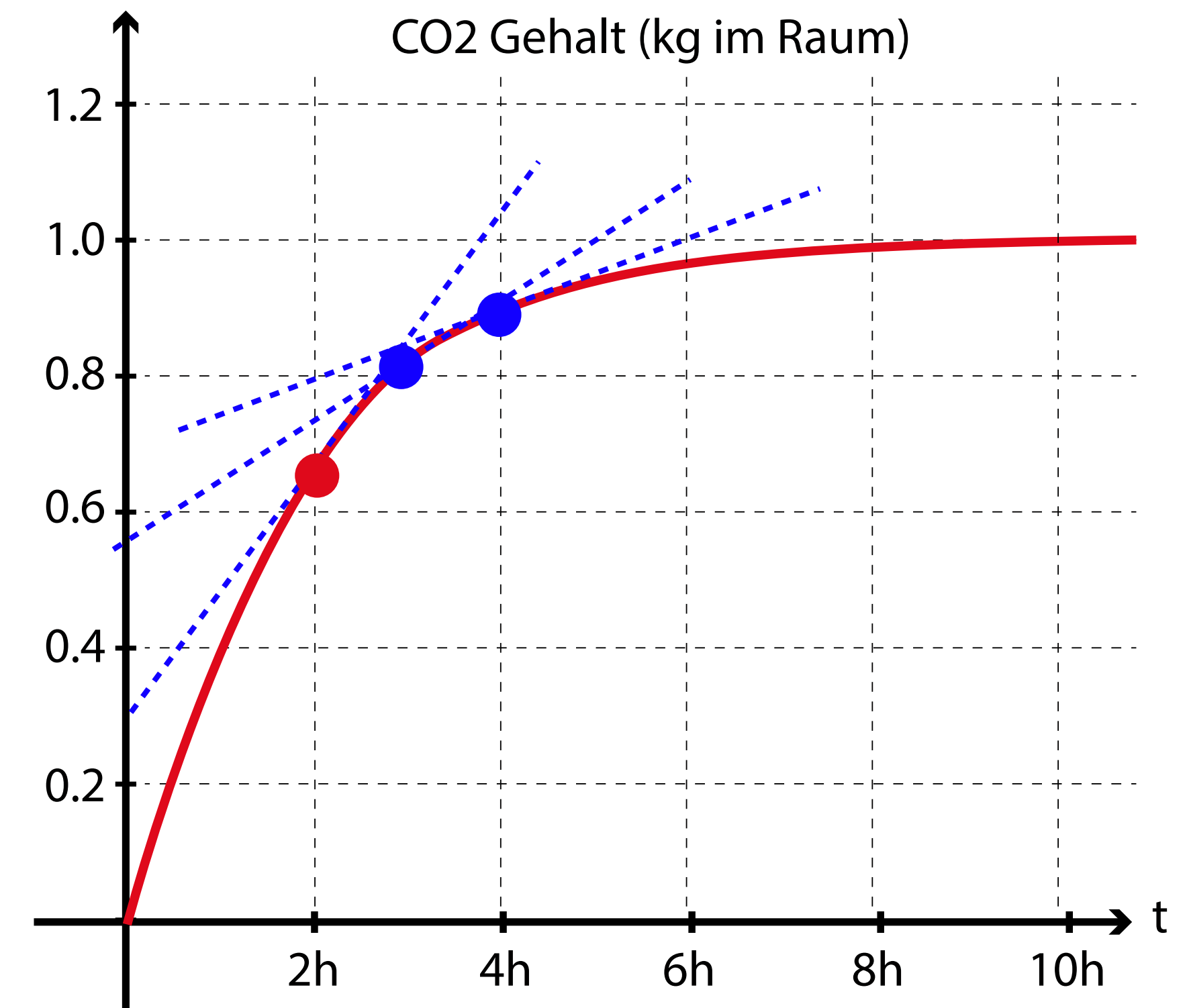


Runge-Kutta Verfahren

Die Runge-Kutta Verfahren sind eine Weiterentwicklung des Euler-Verfahrens mit dem Ziel, diesen Nachteil abzuschwächen.

Das Grundprinzip des iterativen Vorgehens bleibt unangetastet.

Neu ist die Verwendung der Steigung an mehreren Punkten. Wir verwenden nicht nur die Steigung am Anfang eines Schrittes, sondern auch die in der Mitte und am Ende!



Runge-Kutta Verfahren

Hinter den Runge-Kutta Verfahren verstecken sich eine ganze Reihe an Algorithmen, mit denen wir gewöhnliche DGL lösen können.

Der RK1 Algorithmus entspricht dem Eulerverfahren.

Der RK2 Algorithmus ist auch als Mittelpunktregel bekannt.

Der RK4 Algorithmus basiert auf den rechts gezeigten Gleichungen. Würden wir die Koeffizienten k_2 bis k_4 entfernen und den Vorfaktor $h/6$ auf $h/1$ setzen wären wir zurück beim Eulerverfahren.

Runge-Kutta 4

$$y_{n+1} = y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = t_n + h$$

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + \frac{hk_1}{2}, t_n + \frac{h}{2}\right)$$

$$k_3 = f\left(y_n + \frac{hk_2}{2}, t_n + \frac{h}{2}\right)$$

$$k_4 = f(y_n + hk_3, t_n + h)$$

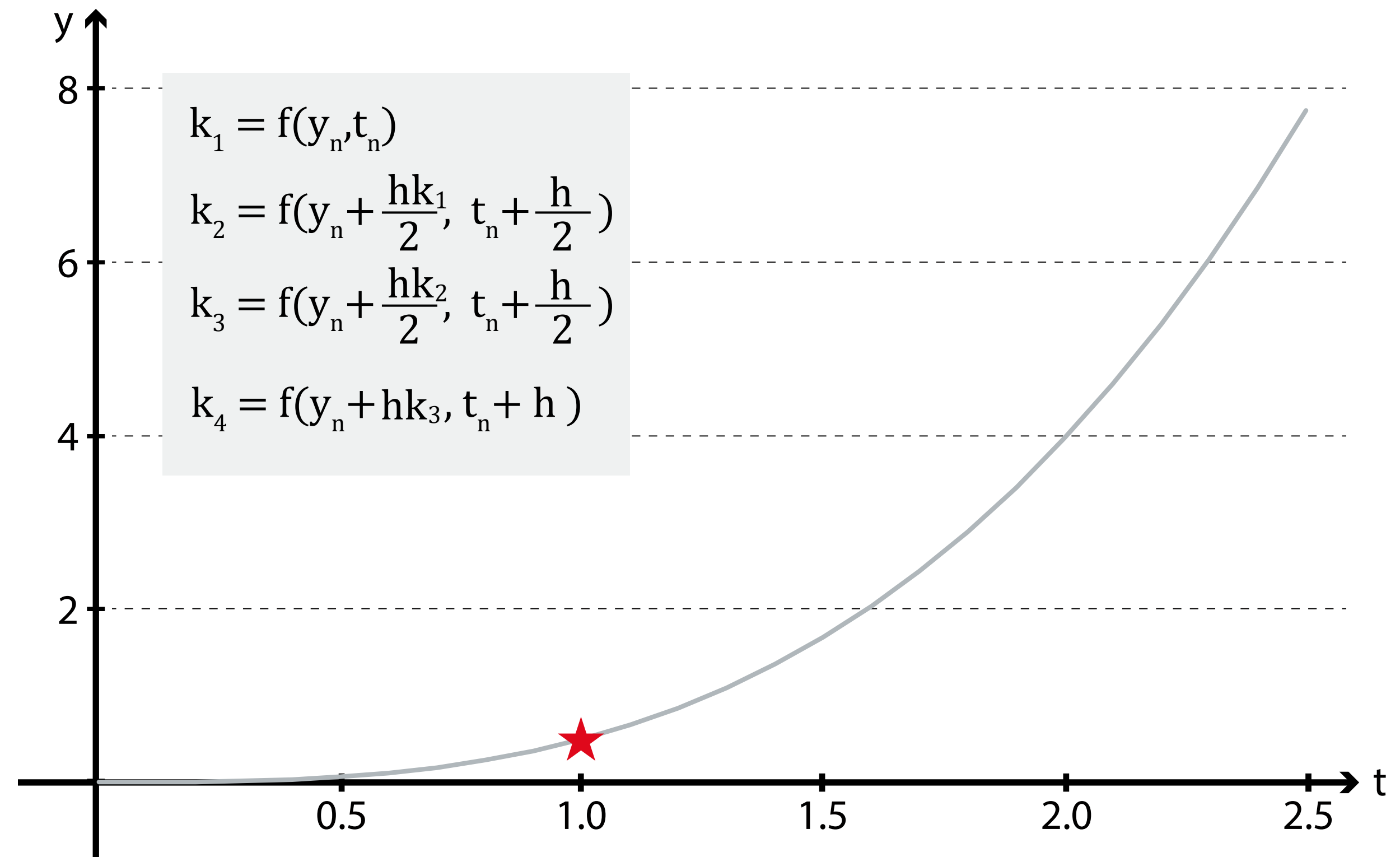
Runge-Kutta Verfahren

Wir schauen uns die Vorgehensweise des RK4 Verfahrens an folgendem Beispiel an:

$$y'(y,t) = \frac{y(t)}{t} + t^2, \quad y(1) = 0.5$$

Die analytische Lösung ist:

$$y(t) = 0.5t^3$$



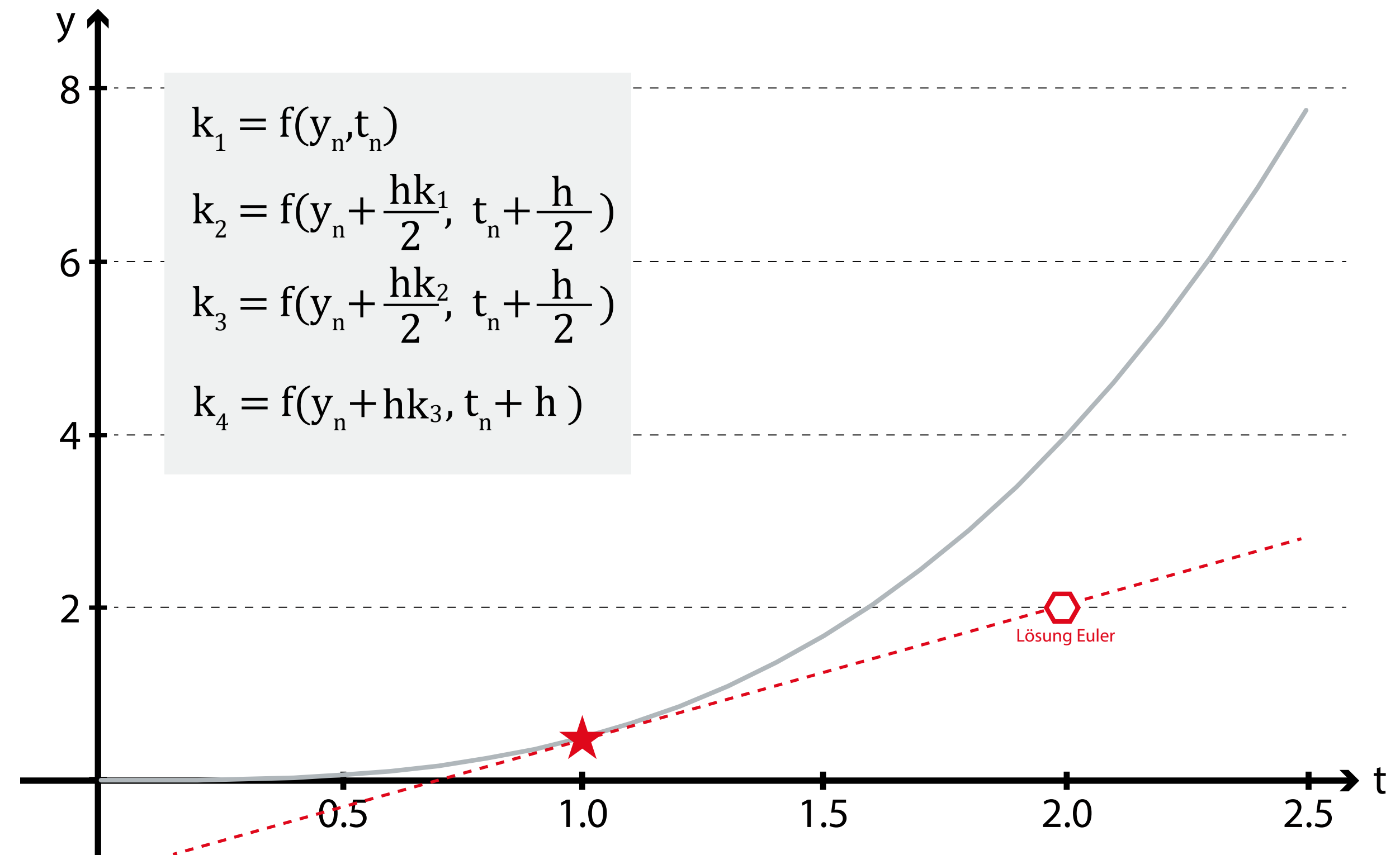
Runge-Kutta Verfahren

Wir wählen eine Schrittweite von 1 und berechnen die Koeffizienten k_1 bis k_4 .

Der erste Koeffizient entspricht der Steigung direkt am gegebenen Punkt $y(1)=0.5$.

$$k_1 = y'(0.5, 1) = \frac{0.5}{1} + 1^2 = 1.5$$

Beim Eulerverfahren würden wir diese Steigung über die Schrittweite anwenden und wären fertig mit dem ersten Schritt.

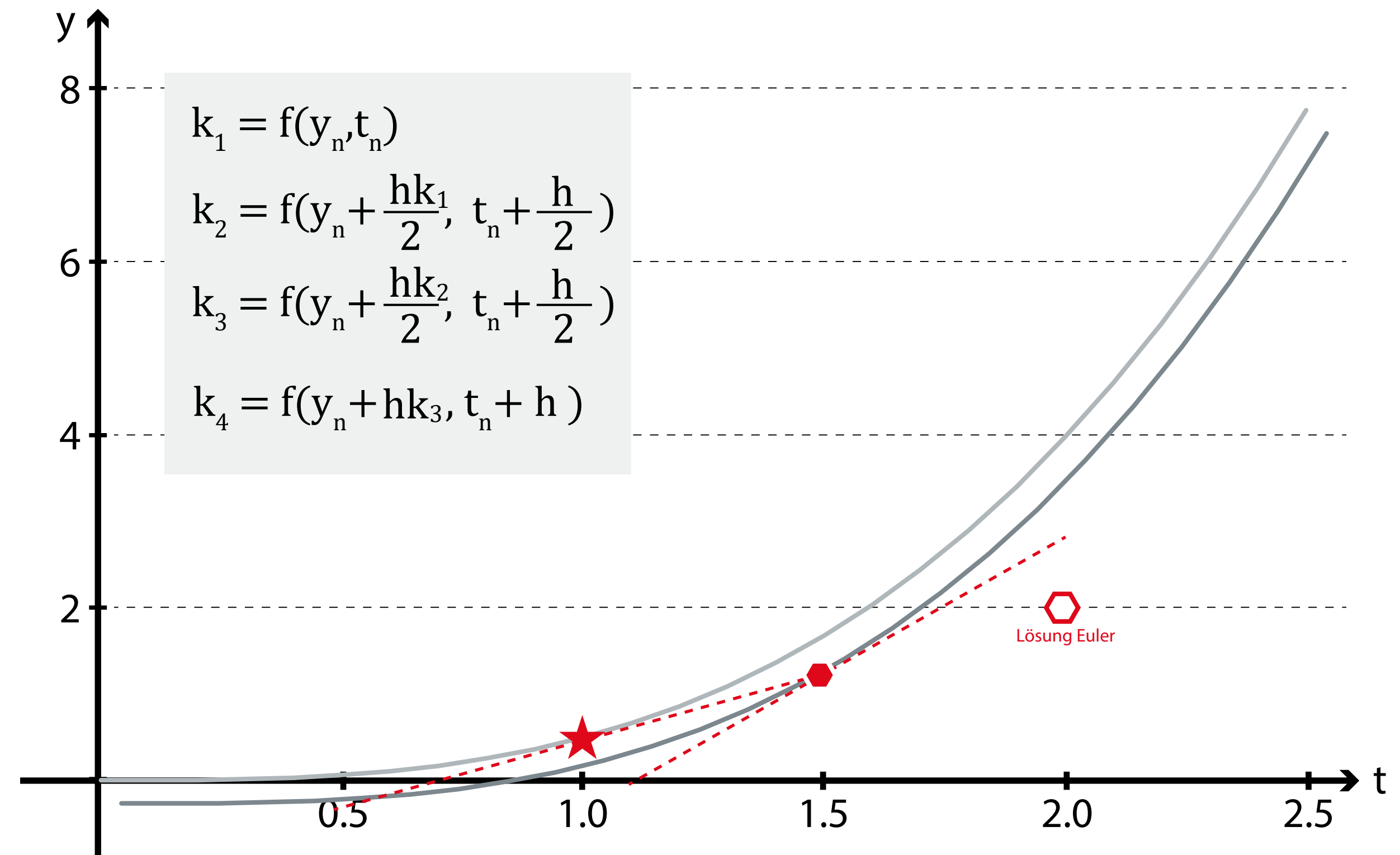


Runge-Kutta Verfahren

RK4 berechnet mit k_1 den Koeffizienten k_2 .

$$\begin{aligned}k_2 &= y(0.5 + 0.5k_1, 1 + 0.5) \\&= y(1.25, 1.5) \\&= 3.083\end{aligned}$$

Wir wenden die Steigung nur über die Hälfte der Schrittweite an und berechnen die Steigung an dieser Stelle.

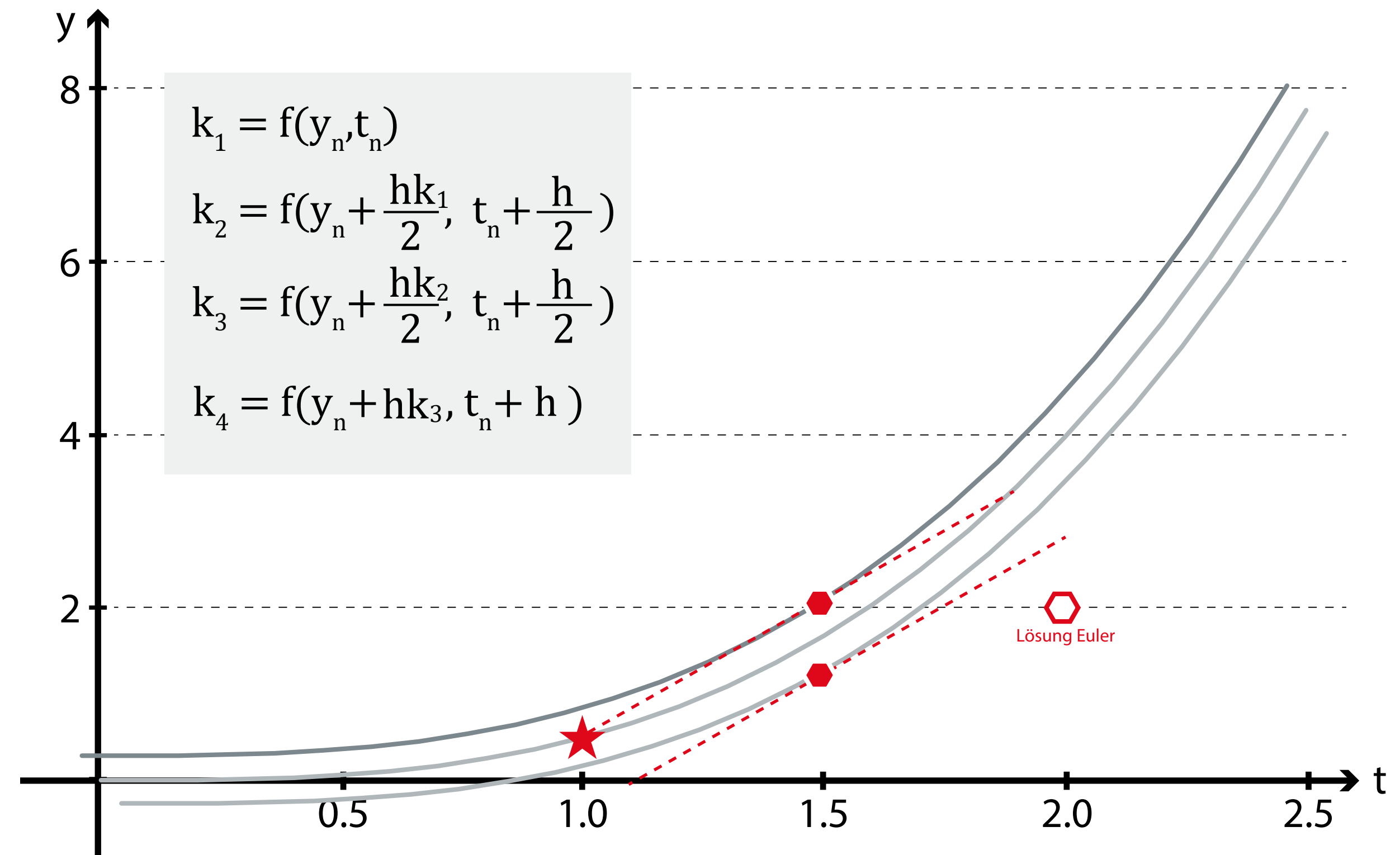


Runge-Kutta Verfahren

Die mit k_2 gefundene Steigung wird nun rückwirkend auf die erste Hälfte der Schrittweite angewendet!

Wir finden einen neuen Punkt und die Steigung dort ist der dritte Koeffizient.

$$\begin{aligned}
 k_3 &= y(0.5 + 0.5k_2, 1 + 0.5) \\
 &= y(2.0415, 1.5) \\
 &= 3.611
 \end{aligned}$$

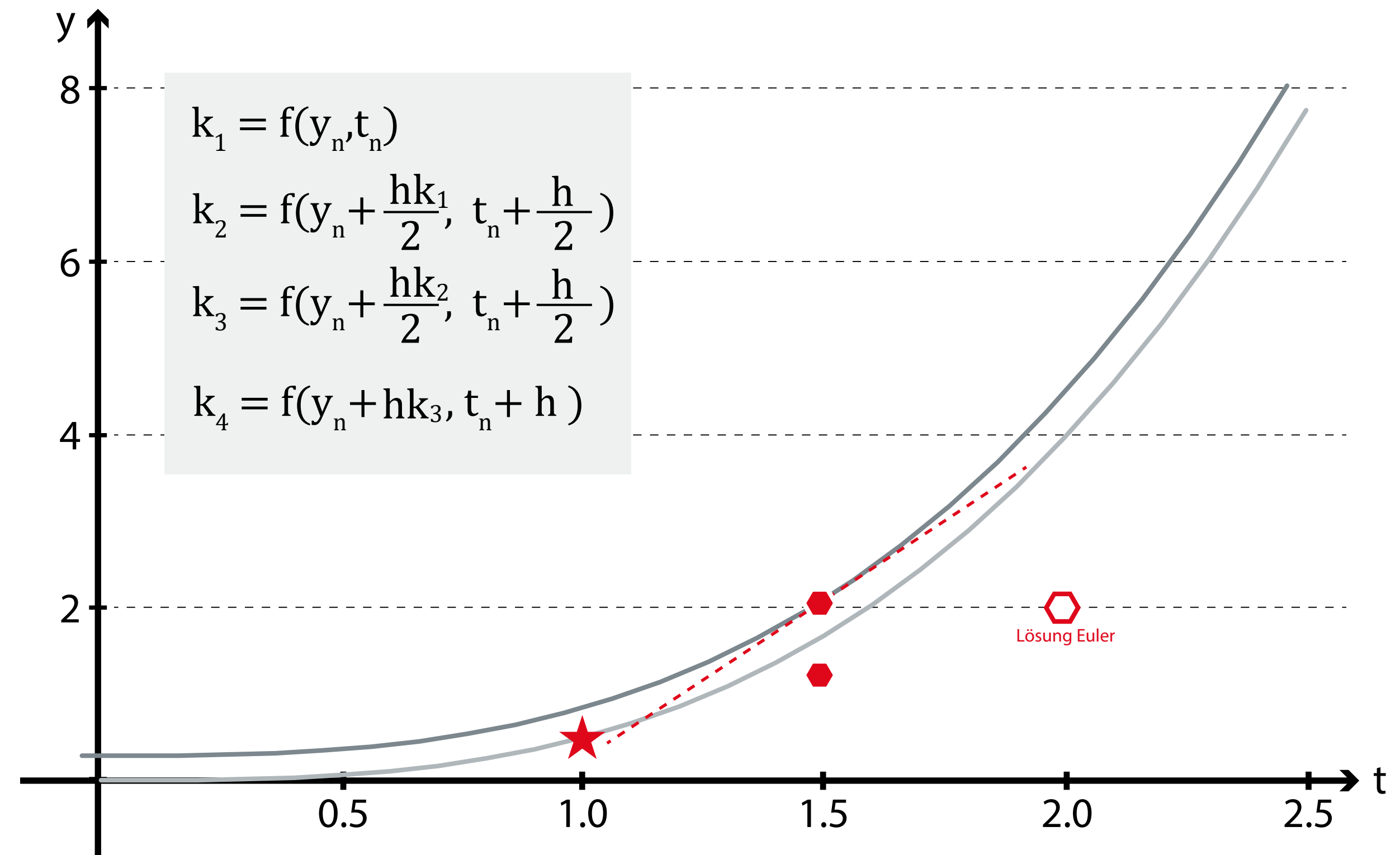


Runge-Kutta Verfahren

Die mit k_2 gefundene Steigung wird nun rückwirkend auf die erste Hälfte der Schrittweite angewendet!

Wir finden einen neuen Punkt und die Steigung dort ist der dritte Koeffizient.

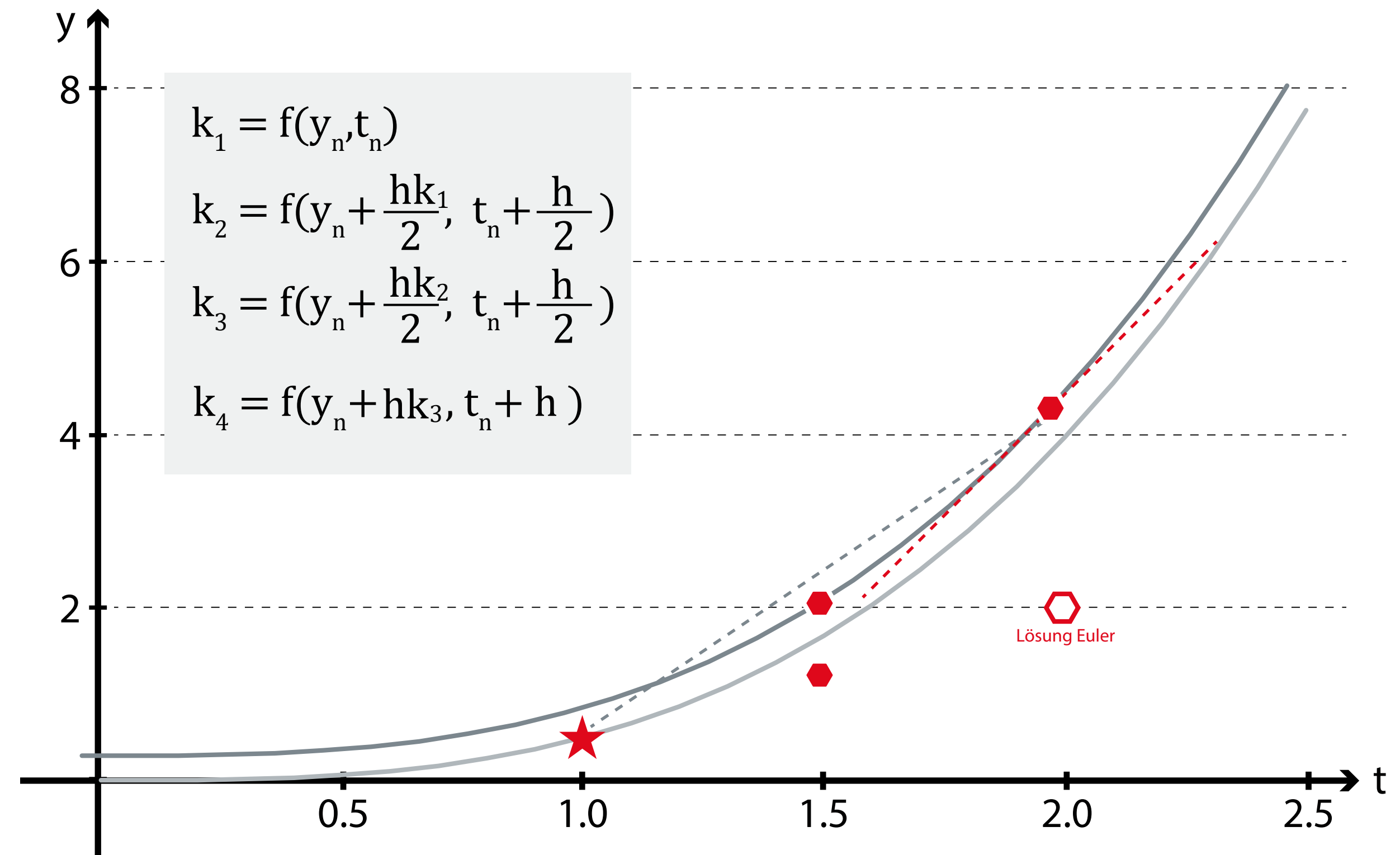
$$\begin{aligned}
 k_3 &= y(0.5 + 0.5k_2, 1 + 0.5) \\
 &= y(2.0415, 1.5) \\
 &= 3.611
 \end{aligned}$$



Runge-Kutta Verfahren

Diese Steigung k_3 wird auf die ganze Schrittweite angewendet, um einen finalen Punkt zu finden, an dem erneut die Steigung gemessen wird!

$$\begin{aligned} k_4 &= y(0.5 + k_3, 1 + 1) \\ &= y(4.111, 2) \\ &= 6.0555 \end{aligned}$$



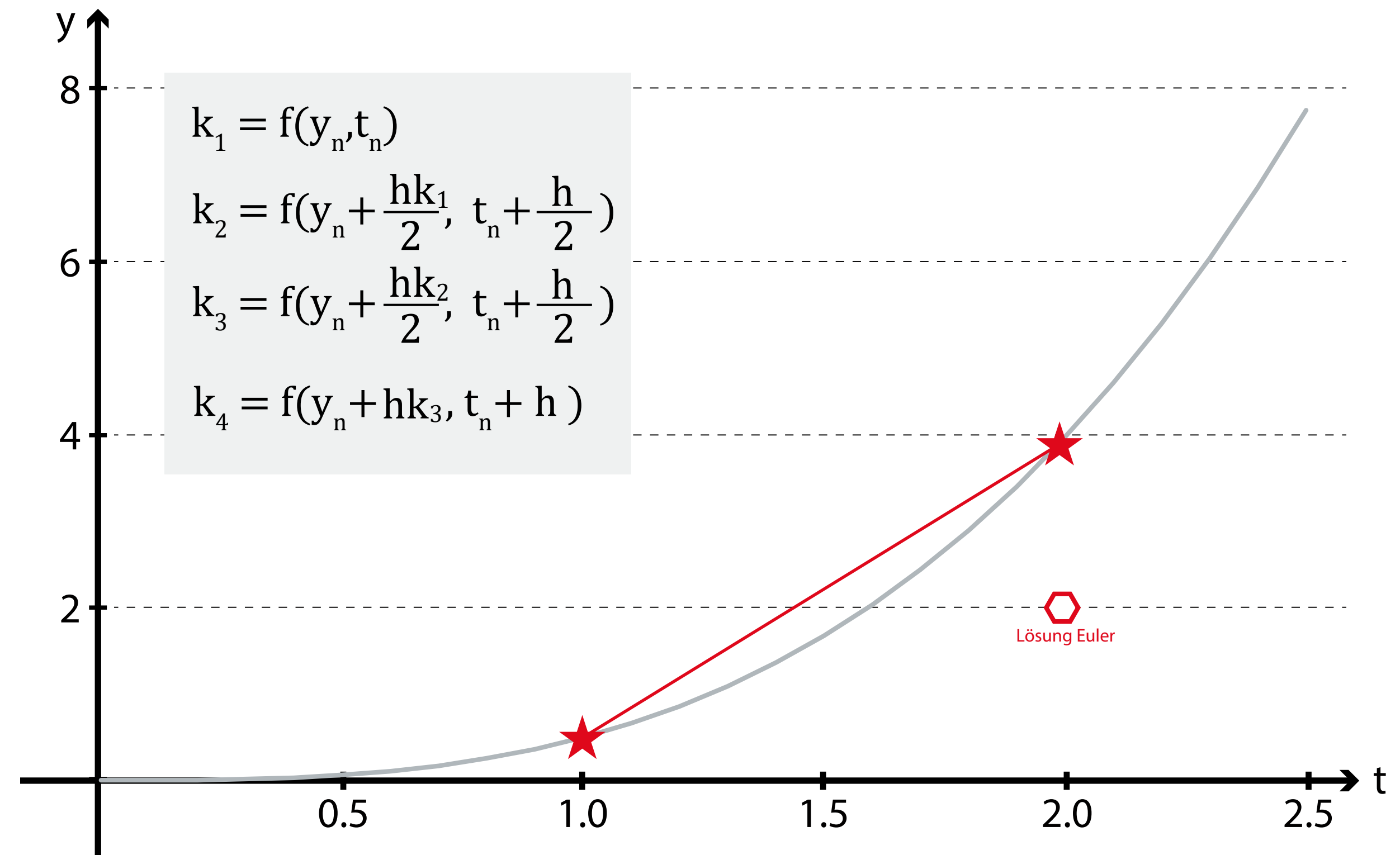
Runge-Kutta Verfahren

Die beim tatsächlich vorgenommenen Schritt verwendete Steigung ist ein gewichteter Durchschnitt aus den Koeffizienten:

$$(k_1 + 2k_2 + 2k_3 + k_4)/6 = 3.491$$

Der neue Punkt ist:

$$y(2) = 0.5 + 3.491 = 3.991$$



Runge-Kutta Verfahren

Die verschiedenen Runge-Kutta-Verfahren können kompakt als Butcher-Tablett dargestellt werden.

Die Spalte links gibt den Inkrement von t als Vielfaches von h an.

Die Matrix rechts oben gibt den Inkrement von y als Vielfaches von h mal den Koeffizienten k_1 , k_2 , k_3 und k_4 an.

Die Zeile unten gibt die Gewichtung von k_1 , k_2 , k_3 und k_4 beim Iterationsschritt an.

		Schrittweite für y			
Schrittweite für t	0				
	$\frac{1}{2}$	$\frac{1}{2}$			
	$\frac{1}{2}$	0	$\frac{1}{2}$		
	1	0	0	1	
		$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$
		Gewichtung im Iterationsschritt			

Runge-Kutta Verfahren

Beispiel: dritter Koeffizient des R4 Verfahrens:

Der Inkrement von t ist 0.5h

Der Inkrement von y ist $0.5hk_2$

Die Gewichtung dieses Koeffizienten bei der Berechnung von y_{n+1} ist 2/6

		Schrittweite für y			
Schrittweite für t	0				
	$\frac{1}{2}$	$\frac{1}{2}$			
	$\frac{1}{2}$	0	$\frac{1}{2}$		
	1	0	0	1	
		Gewichtung im Iterationsschritt			
		$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

Runge-Kutta 4

$$y_{n+1} = y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = t_n + h$$

$$k_1 = f(y_n, t_n)$$

$$k_2 = f(y_n + \frac{hk_1}{2}, t_n + \frac{h}{2})$$

$$k_3 = f(y_n + \frac{hk_2}{2}, t_n + \frac{h}{2})$$

$$k_4 = f(y_n + hk_3, t_n + h)$$

Runge-Kutta Verfahren

Es gibt **adaptive Runge-Kutte-Verfahren**, bei denen wir im Butcher Tablett mehr als eine Zeile haben.

Dadurch werden zwei RK Varianten mit unterschiedlichen Konvergenzordnungen gleichzeitig angewendet.

Die Idee: Durch den Vergleich der beiden Varianten kann der Fehler abgeschätzt werden. Ist er zu groß, wird die Schrittweite verringert.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

Runge-Kutta Verfahren

Mache dir den Zusammenhang zwischen Gleichungen, Butcher Tableau und dem Quellcode in "Runge Kutta" klar.

Implementiere mit der Mittelpunkregel (RK2) ein zusätzliches Verfahren.

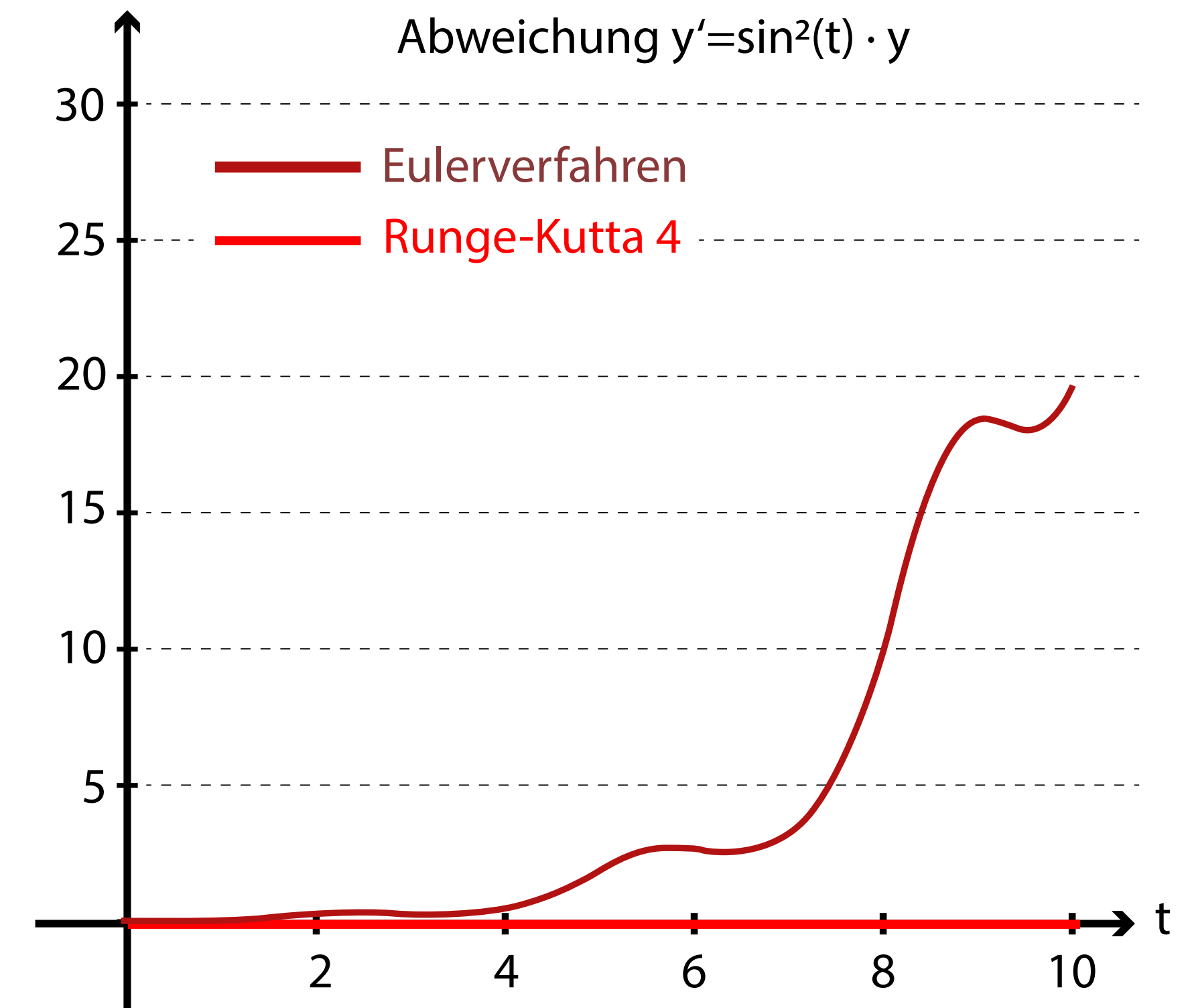
Teste mit den drei gegebenen Beispielen, welches Verfahren die bessere Näherung erzeugt.

0		
$\frac{1}{2}$	$\frac{1}{2}$	
	0	1

Runge-Kutta Verfahren

In allen drei Beispielen ist der RK4 Algorithmus bei gleicher Parametrisierung deutlich genauer.

Allerdings muss er die Funktionen auch mehrfach pro Iteration auswerten. Aber selbst wenn wir dem Eulerverfahren daher eine höhere Auflösung zugestehen ist RK4 genauer.



Numerische Integration

Ähnlich wie DGLs sind Integrale oft nur mit diversen Tricks oder sogar grundsätzlich nicht analytisch lösbar.

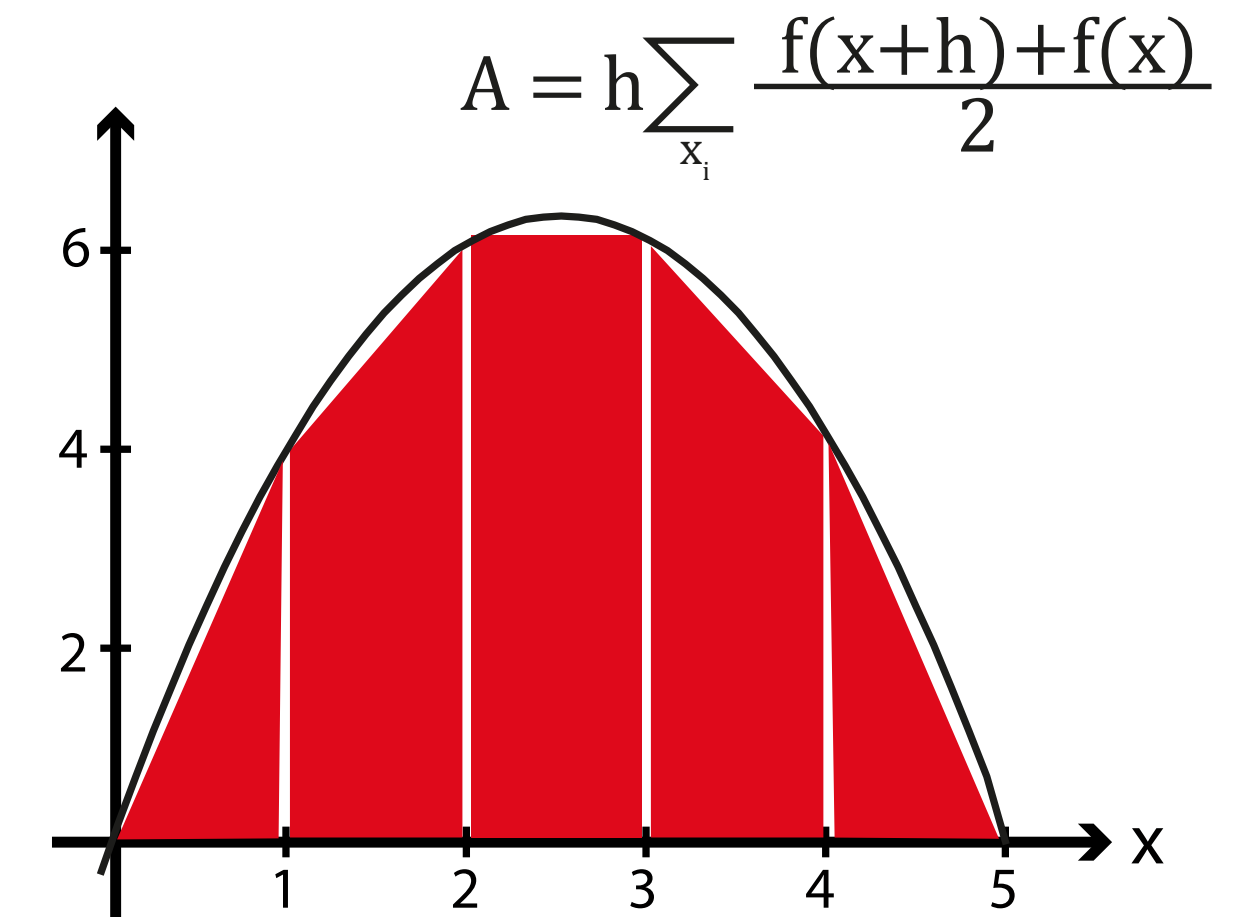
In diesem Kapitel lernen wir, bestimmte Integrale numerisch zu berechnen.

Wir erhalten dabei zwar keine Stammfunktion, aber eine Näherung für den Wert des bestimmten Integrals!

Analytische Lösung

$$\begin{aligned}
 \int_0^5 -x^2 + 5x &= \left[-\frac{1}{3}x^3 + 2.5x^2 \right]_0^5 \\
 &= -\frac{1}{3}5^3 + 2.5 \cdot 5^2 - 0 \\
 &= \frac{62}{3} \\
 &\approx 20.833
 \end{aligned}$$

Numerische Lösung

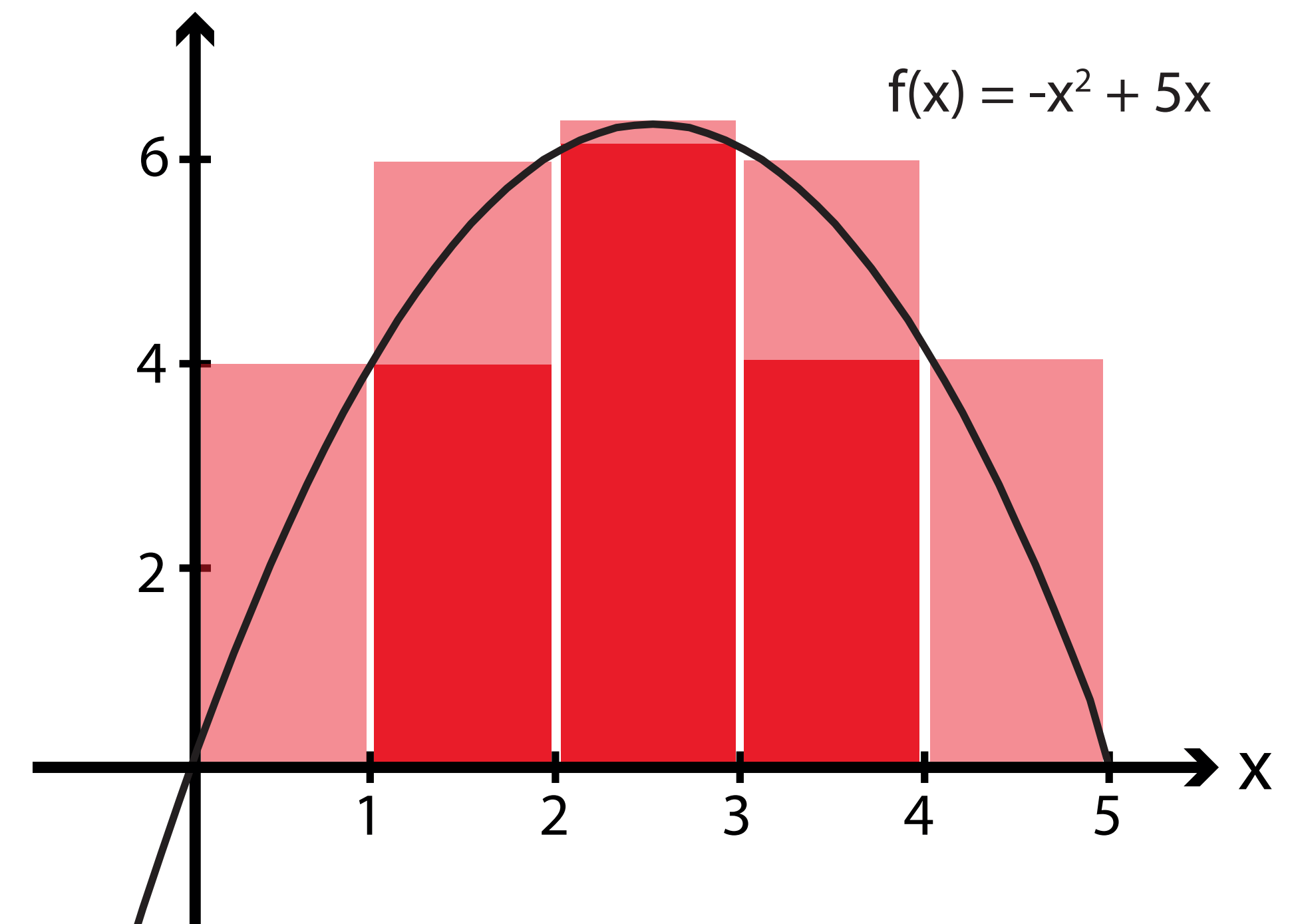


Trapezregel

Mit den Unter- und Obersummen haben wir bei der Definition des Integrals bereits eine Näherung kennengelernt.

Wir nähern die Fläche unter der Kurve durch Rechtecke einer festen Breite. Als Höhe wählen wir:

- Maximum der Funktion im Bereich (hell- & dunkelrote Flächen)
- Minimum der Funktion im Bereich (nur dunkelrot Flächen)



Trapezregel

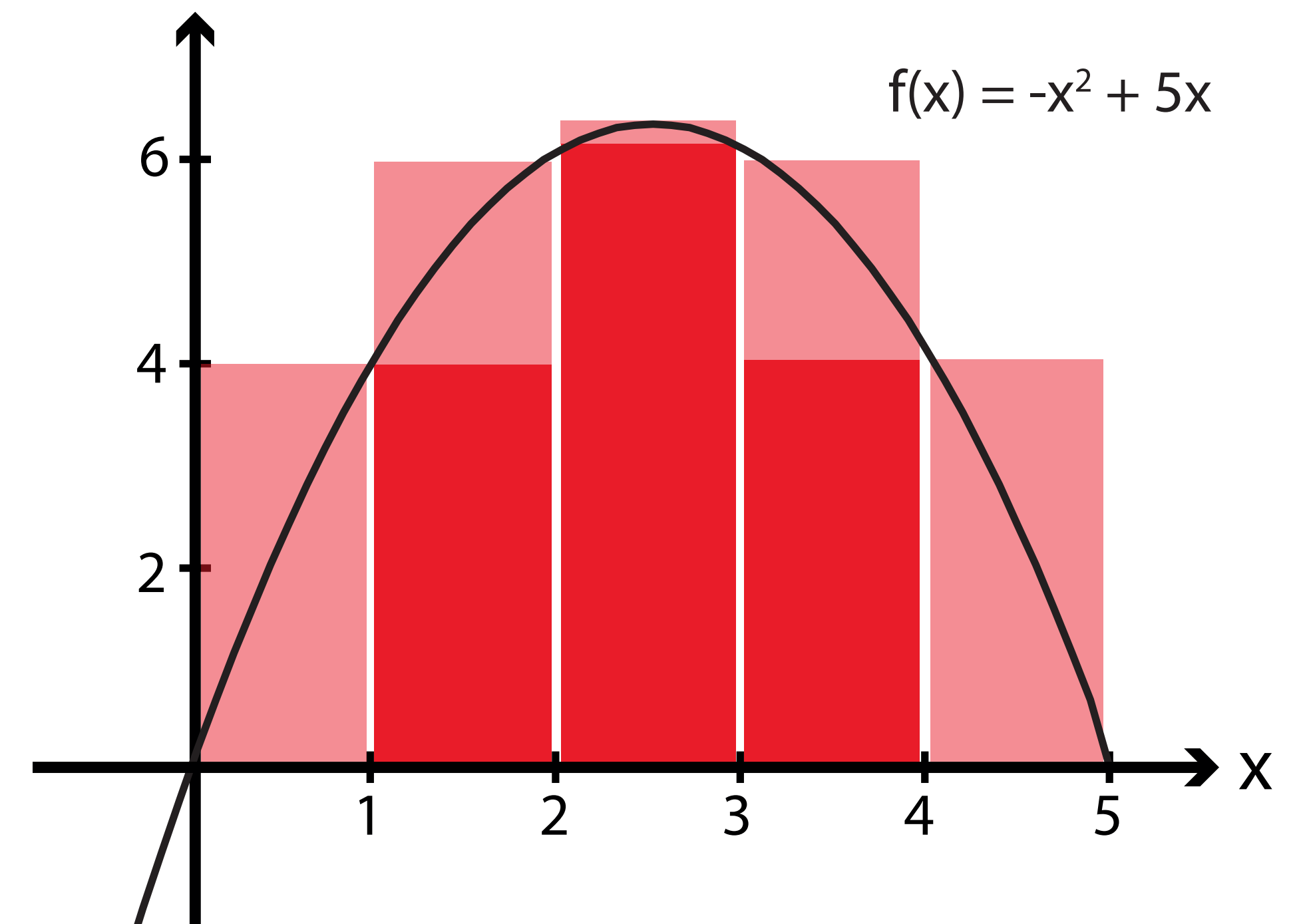
Wir erhalten die folgenden Näherungen:

$$\text{Obersumme } A \approx 4 + 6 + 6.25 + 6 + 4 = 26.25$$

$$\text{Untersumme } A \approx 0 + 4 + 6 + 4 + 0 = 14$$

Die Obersumme überschätzt die wahre Fläche tendenziell, da ihre Flächen über die Kurve hinausragen.

Die Untersumme unterschätzt die wahre Fläche tendenziell, da ihre Flächen die Kurve nicht vollständig ausfüllen.

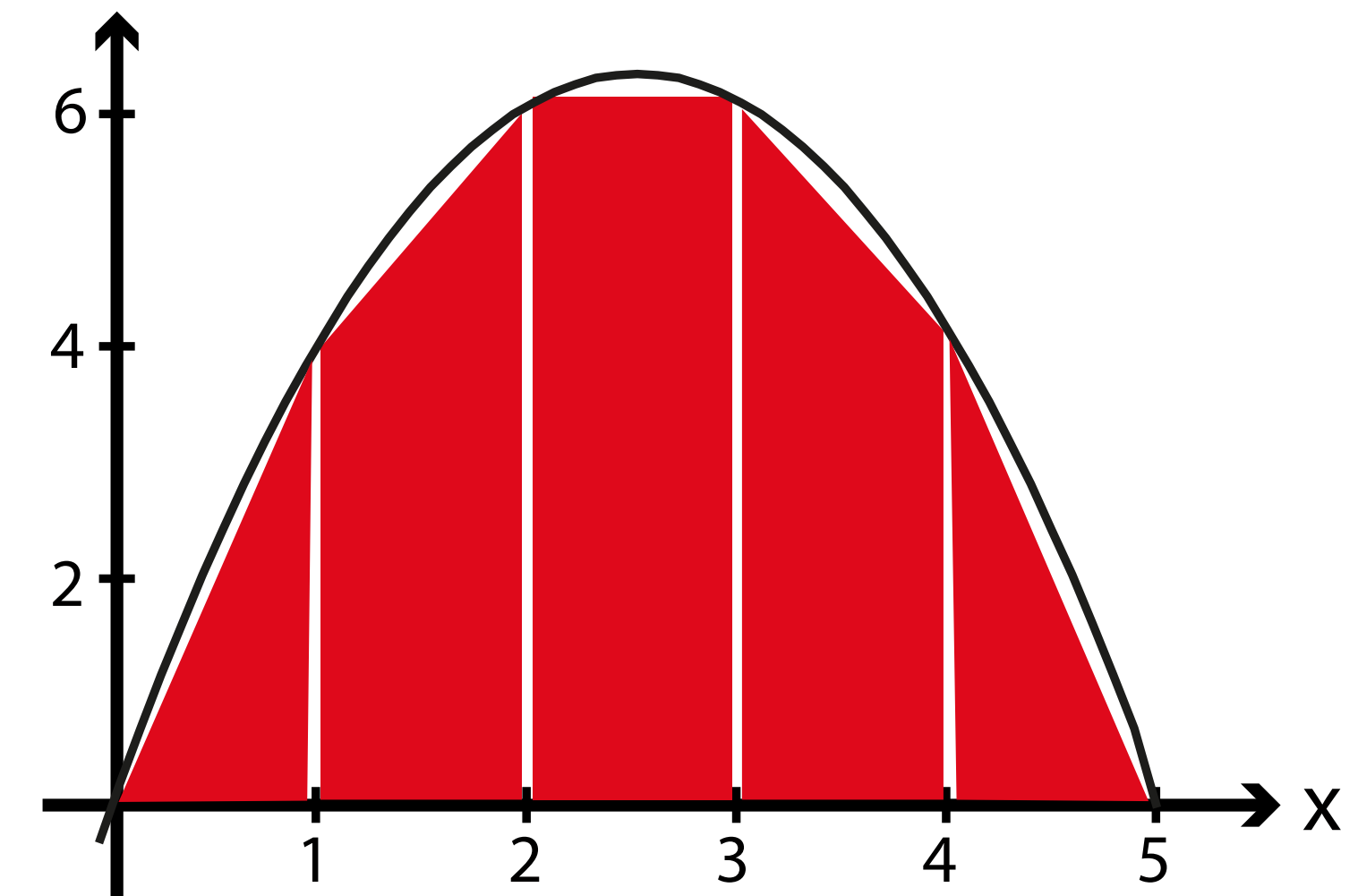


Trapezregel

Warum nicht den Mittelwert von beiden nehmen? Folgen wir diesem Ansatz, nähern wir die Fläche unter der Kurve grafisch gesehen durch Dreiecke und Trapeze.

Bei einer Breite von 1 gibt die Trapezregel eine Fläche von 20 an. Das ist gerade mal 4.2% daneben, obwohl die Breite doch recht groß gewählt wurde.

Je kleiner die Breite gewählt wird, umso besser ist die Näherung, aber umso länger ist die Rechenzeit.



$$\left. \begin{aligned} A_1 &= 0.5 \cdot (0+4) \cdot 1 = 2 \\ A_2 &= 0.5 \cdot (4+6) \cdot 1 = 5 \\ A_3 &= 0.5 \cdot (6+6) \cdot 1 = 6 \\ A_4 &= 0.5 \cdot (6+4) \cdot 1 = 5 \\ A_5 &= 0.5 \cdot (4+0) \cdot 1 = 2 \end{aligned} \right\} A = 20$$

Trapezregel

Die Implementierung der Trapezregel in R oder Python ist einfach. Probiert es mit "Trapez Template" zunächst mal selbst und schaut bei Bedarf in die Lösung.

Verwendet die rechts gezeigten Beispiele, um die Größe des Fehlers in Prozent zu messen.

Verwende jeweils eine Auflösung von 20 Trapezen.

Integral	Sollwert	Trapez	Simpson	Boole
$\int_0^1 e^x dx$	1.71828182			
$\int_0^1 \frac{dx}{1+x^2}$	0.78539816			
$\int_0^1 \sqrt{\ln(x+1)} dx$	0.59259042			
$\int_0^1 1 + \sin(2\pi x) dx$	1.00000000			

Trapezregel

Die Näherung durch die Trapezregel ist bereits ziemlich gut! Können wir die Genauigkeit erhöhen, ohne den Rechenaufwand zu steigern?

Gibt es Alternativen zur Trapezregel?

Integral	Sollwert	Trapez	Simpson	Boole
$\int_0^1 e^x dx$	1.71828182	0.02083%		
$\int_0^1 \frac{dx}{1+x^2}$	0.78539816	0.01326%		
$\int_0^1 \sqrt{\ln(x+1)} dx$	0.59259042	0.38105%		
$\int_0^1 1 + \sin(2\pi x) dx$	1.00000000	Exakt		

Newton-Cotes Formeln

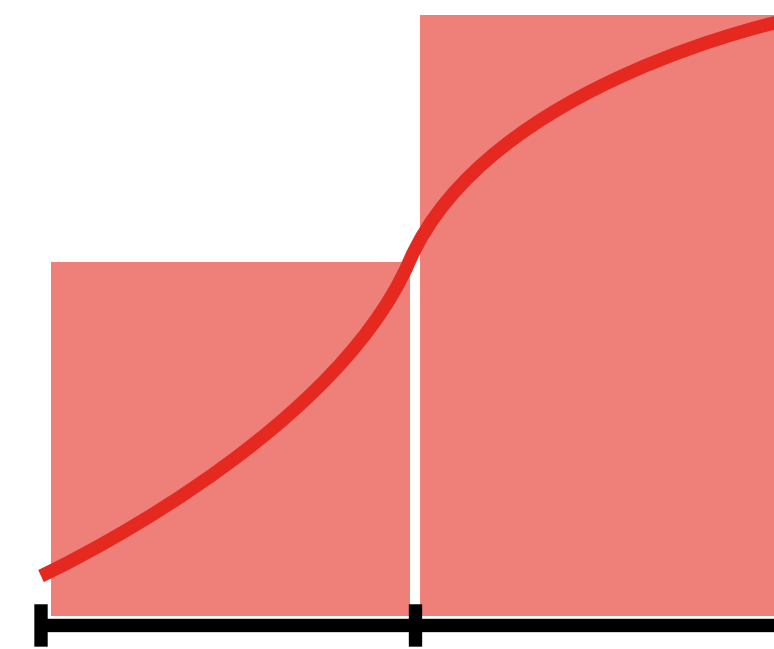
Die Newton-Cotes Formeln bieten eine ganze Sammlung an Näherungsformeln. Die Trapezregel ist eine von diesen Näherungsformeln. Es gibt aber auch noch:

Simpson-Regel

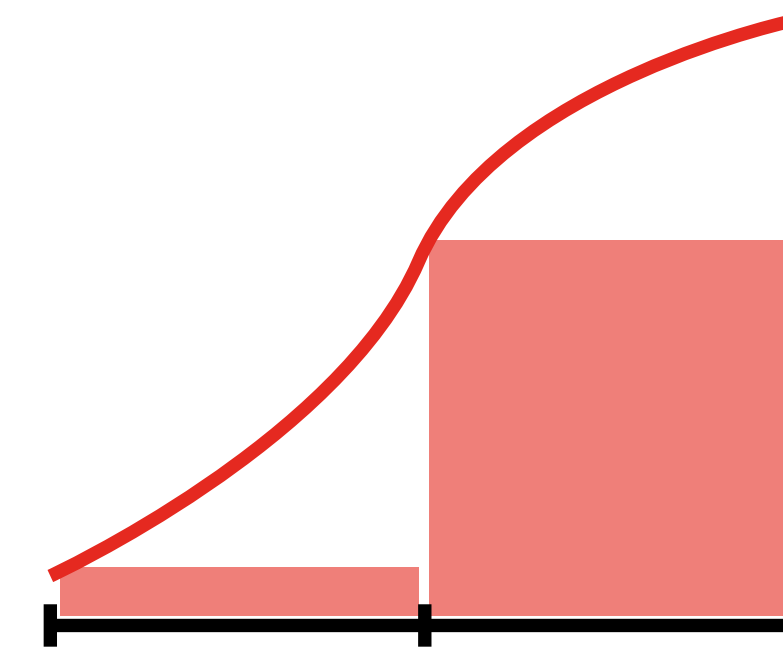
Simpson-Regel (3/8)

Boole-Regel

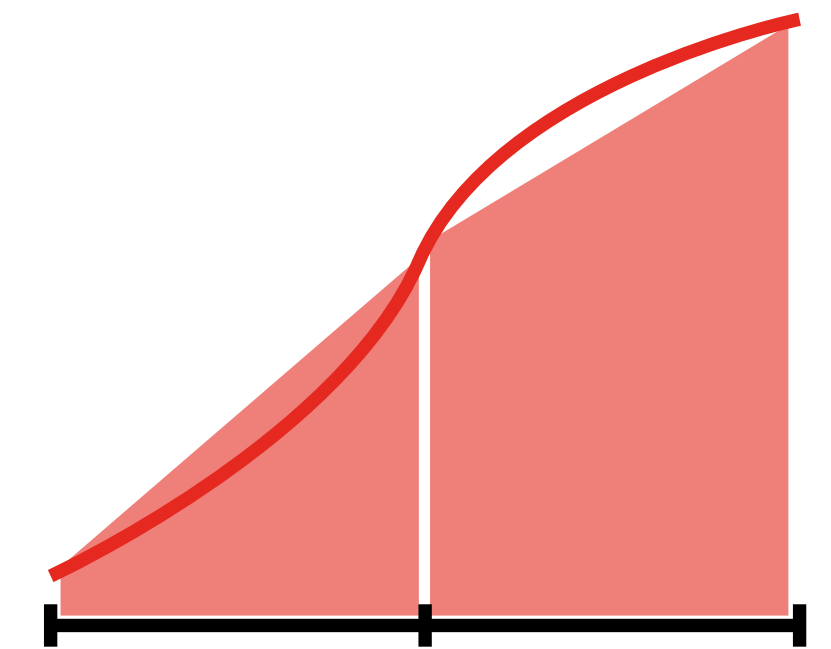
... und viele mehr!



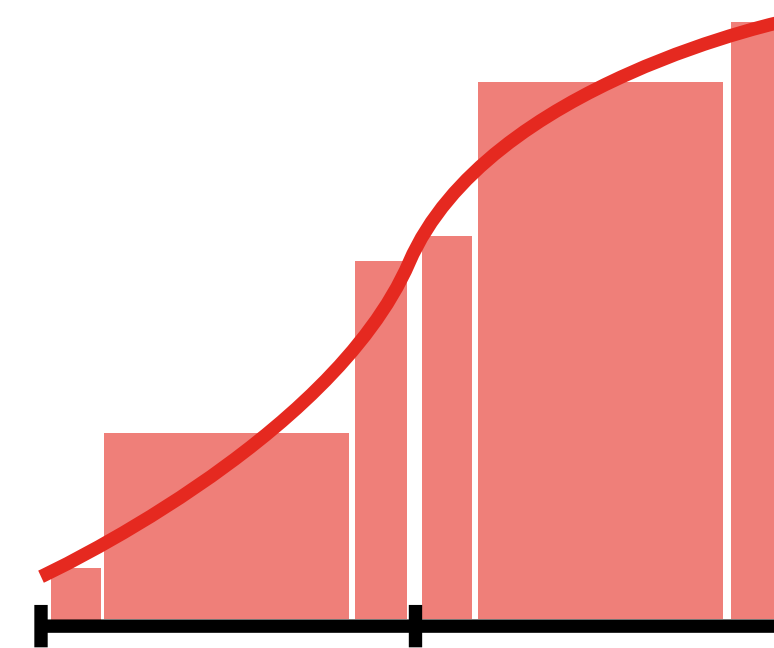
Obersumme



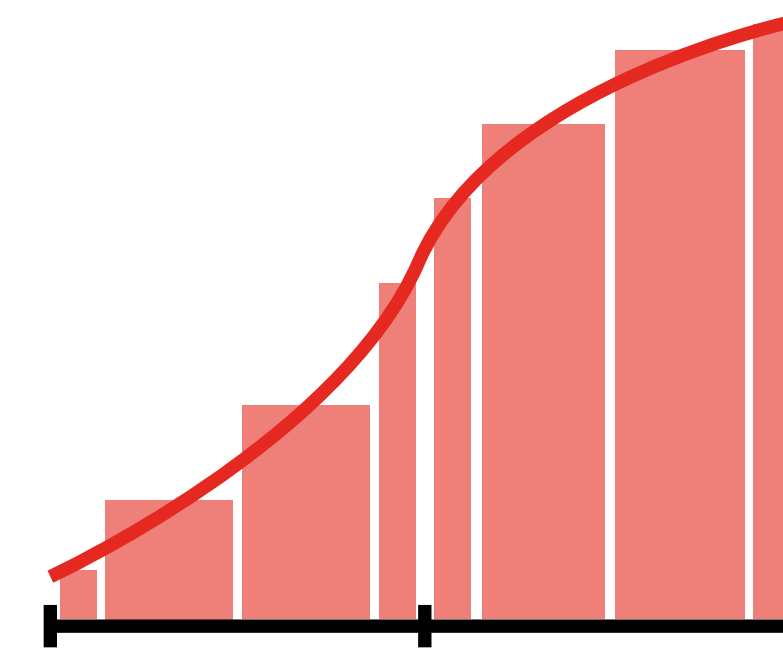
Untersumme



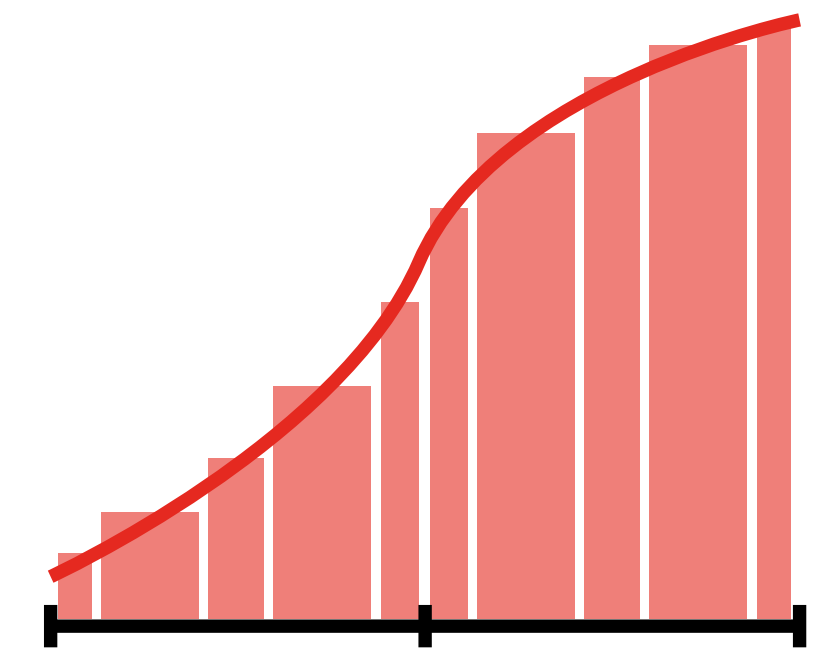
Trapezregel



Simpsonregel



3/8 Regel



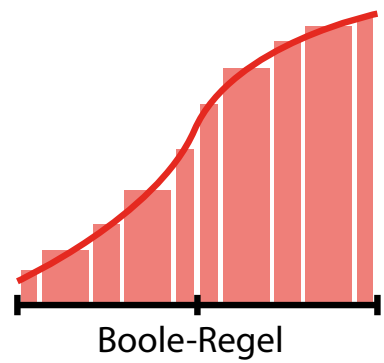
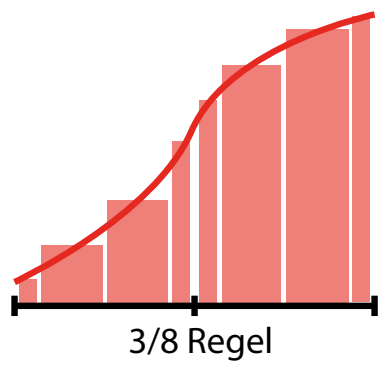
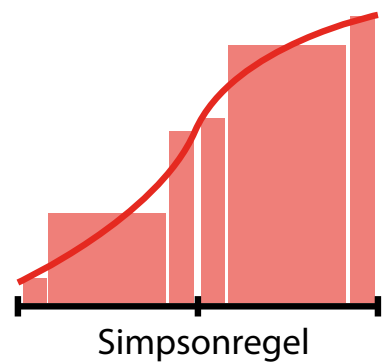
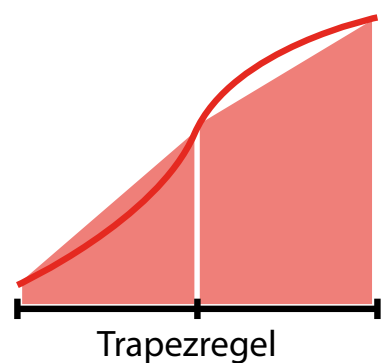
Boole-Regel

Newton-Cotes Formeln

Die Idee ist bei allen diesen Formeln gleich: Wir teilen das Intervall über das wir integrieren in mehrere Bereiche auf.

Innerhalb dieser Bereiche wird die Funktion an mehreren Stellen ausgewertet.

Meistens werden die Stellen in der Mitte der Bereiche höher gewichtet, aber es gibt Ausnahmen wie z. B. die Boole-Regel.



Stützstelle	0	1
Gewichtung	1/2	1/2

Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

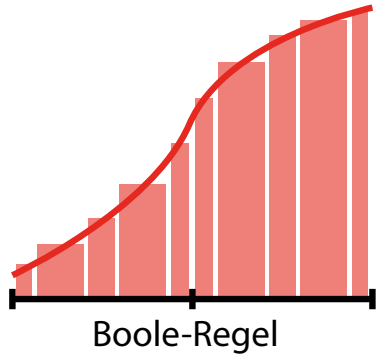
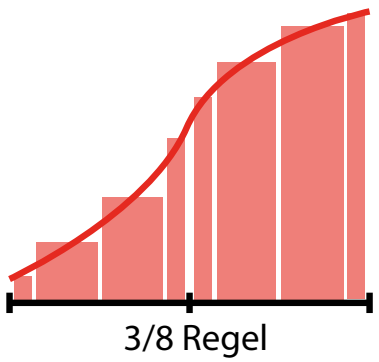
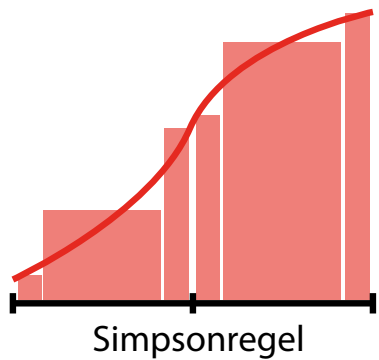
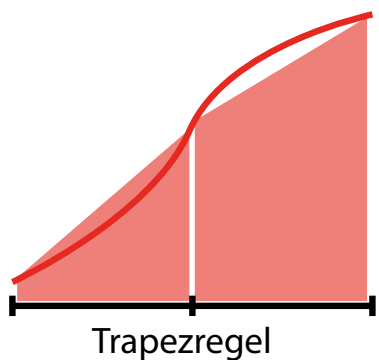
Newton-Cotes Formeln

Das Aufteilen des Intervalls in mehrere Bereiche entspricht der Summenregel der Integration:

$$\int_a^c f(x) \, dx = \int_a^b f(x) \, dx + \int_b^c f(x) \, dx$$

Das Vorgehen innerhalb der Bereiche wird als numerische Quadratur bezeichnet.

$$\int_a^b f(x) \, dx \approx \sum_{i=0}^n w_i f(x_i)$$



Stützstelle	0	1
Gewichtung	1/2	1/2

Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

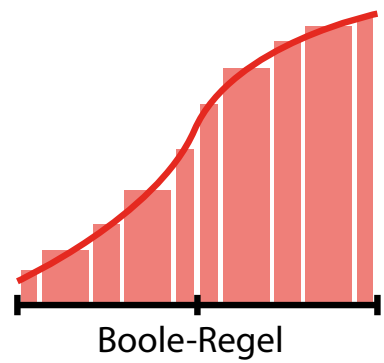
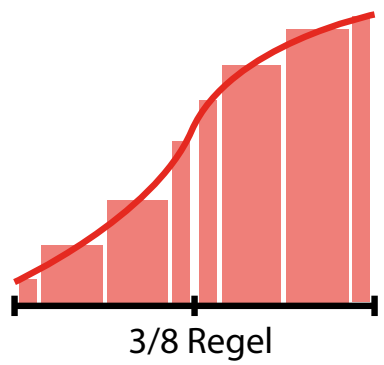
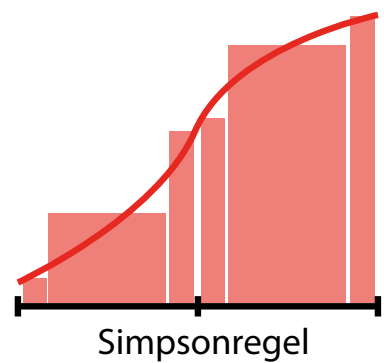
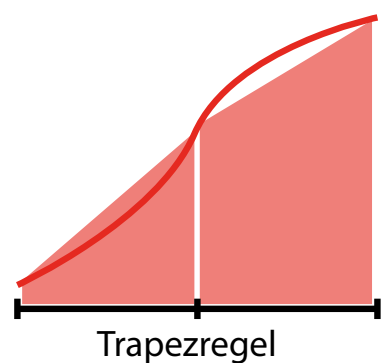
Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

Newton-Cotes Formeln

Die Auswertung der Funktion an n-Stellen führt zusammen mit gut gewählten Gewichtungen zu einer Approximation der Funktion als Polynom n-ter oder (n-1)ter Ordnung.

Aber wo kommen die Gewichtungen her?

Wir machen einen Exkurs zum Thema Inter- und Extrapolation!



Stützstelle	0	1
Gewichtung	1/2	1/2

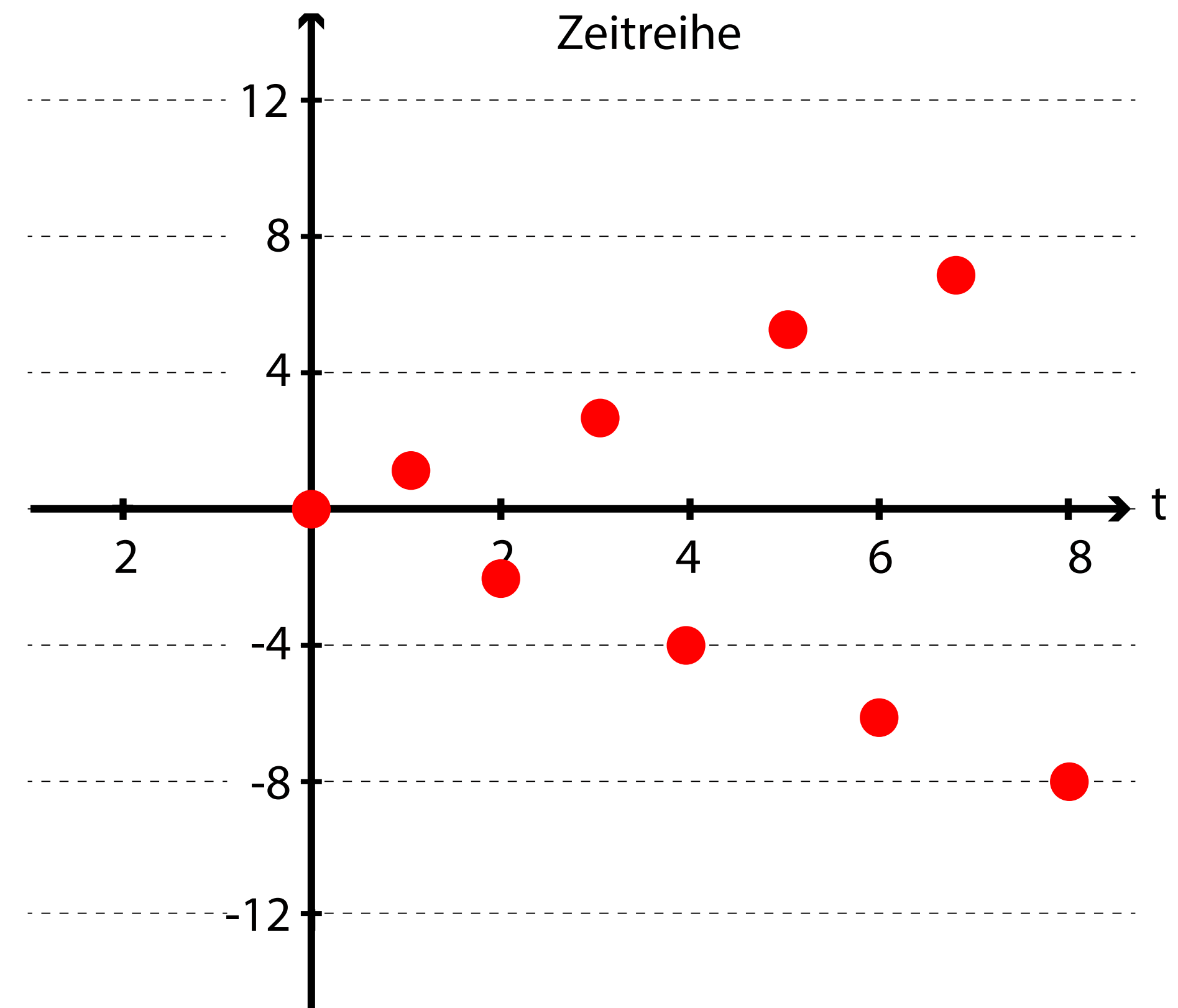
Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

Inter- und Extrapolation

Wie finden wir eine Funktion, die durch die Punkte der Zeitreihe geht und uns auch Werte für andere Stellen liefert?

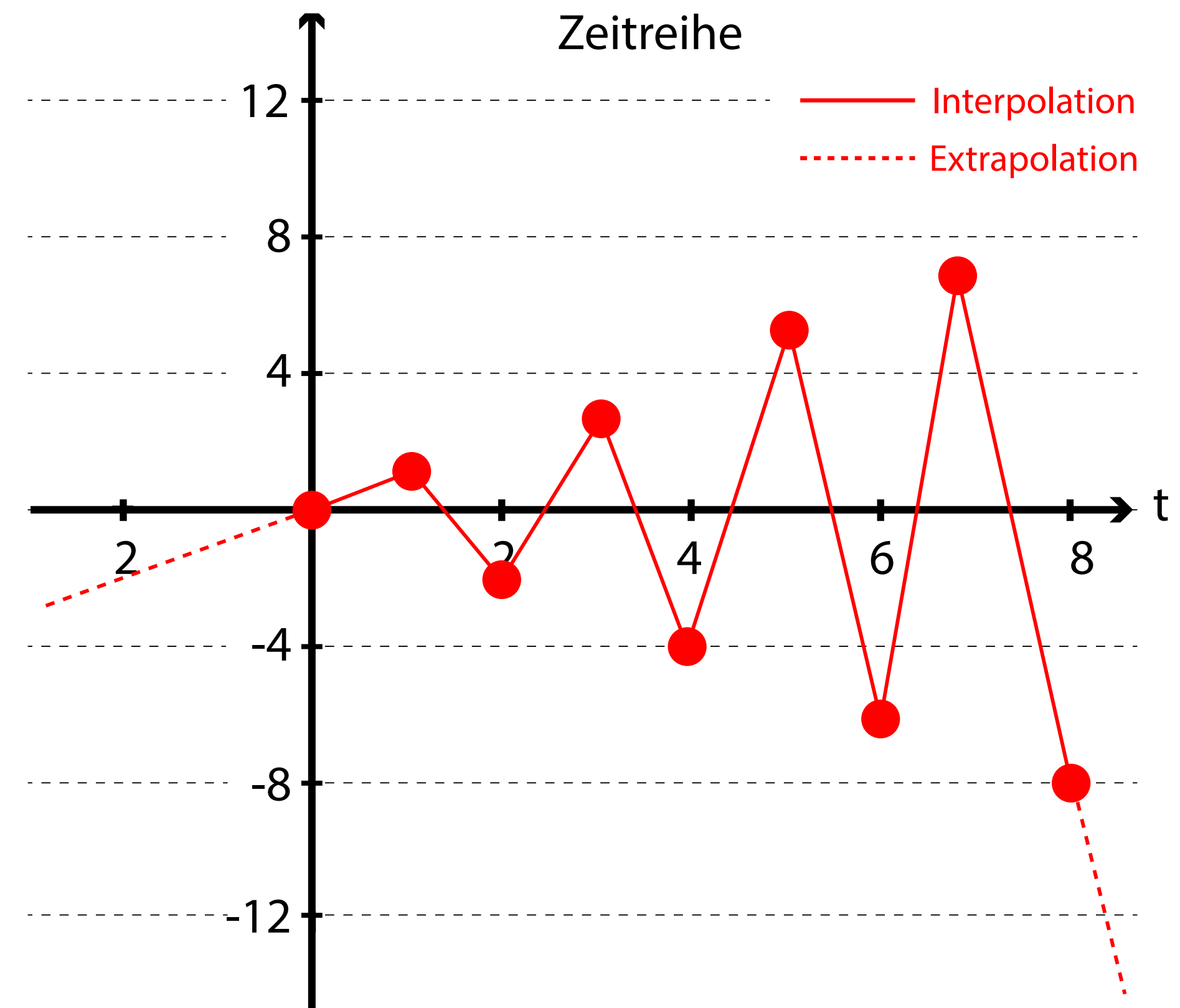


Inter- und Extrapolation

Eine einfache Variante ist die **lineare Interpolation**, bei der wir zwischen den Datenpunkten von einem linearen Verlauf ausgehen.

Dadurch erhalten wir eine stückweise definierte Funktion mit einer zusätzlichen Teilfunktion pro zusätzlichem Punkt.

Das Vorgehen entspricht der Trapezregel. Einfach, aber ungenau und krude in der Darstellung.

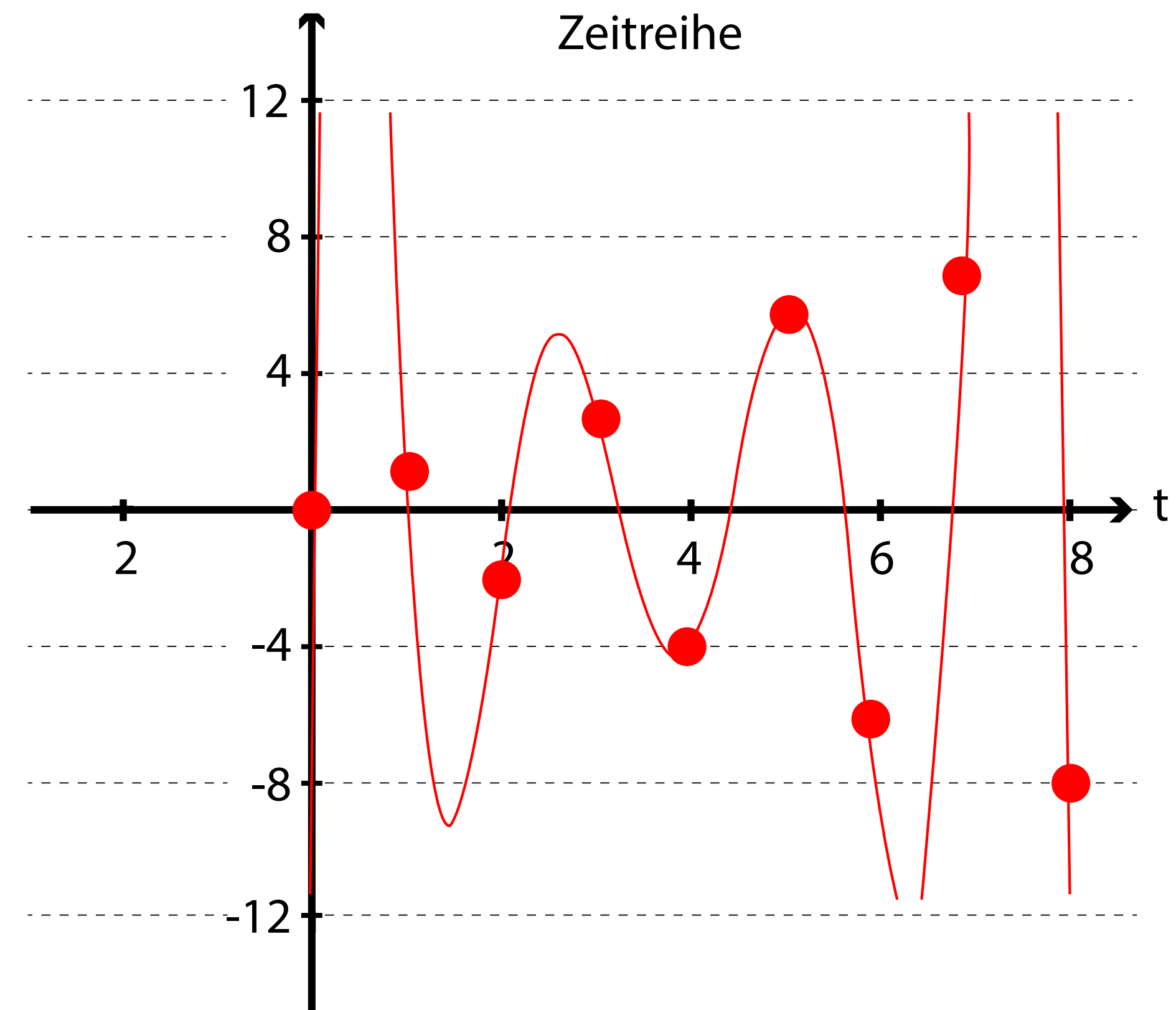


Inter- und Extrapolation

Eine elegantere Variante ist die **Polynominterpolation**.
Bei dieser suchen wir ein Polynom n-ten Grades ...

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

...das durch alle gegebenen Punkte geht. Tatsächlich existiert zu jeder Zeitreihe mit $n+1$ Datenpunkten ein passendes Polynom n-ten Grades oder niedriger.



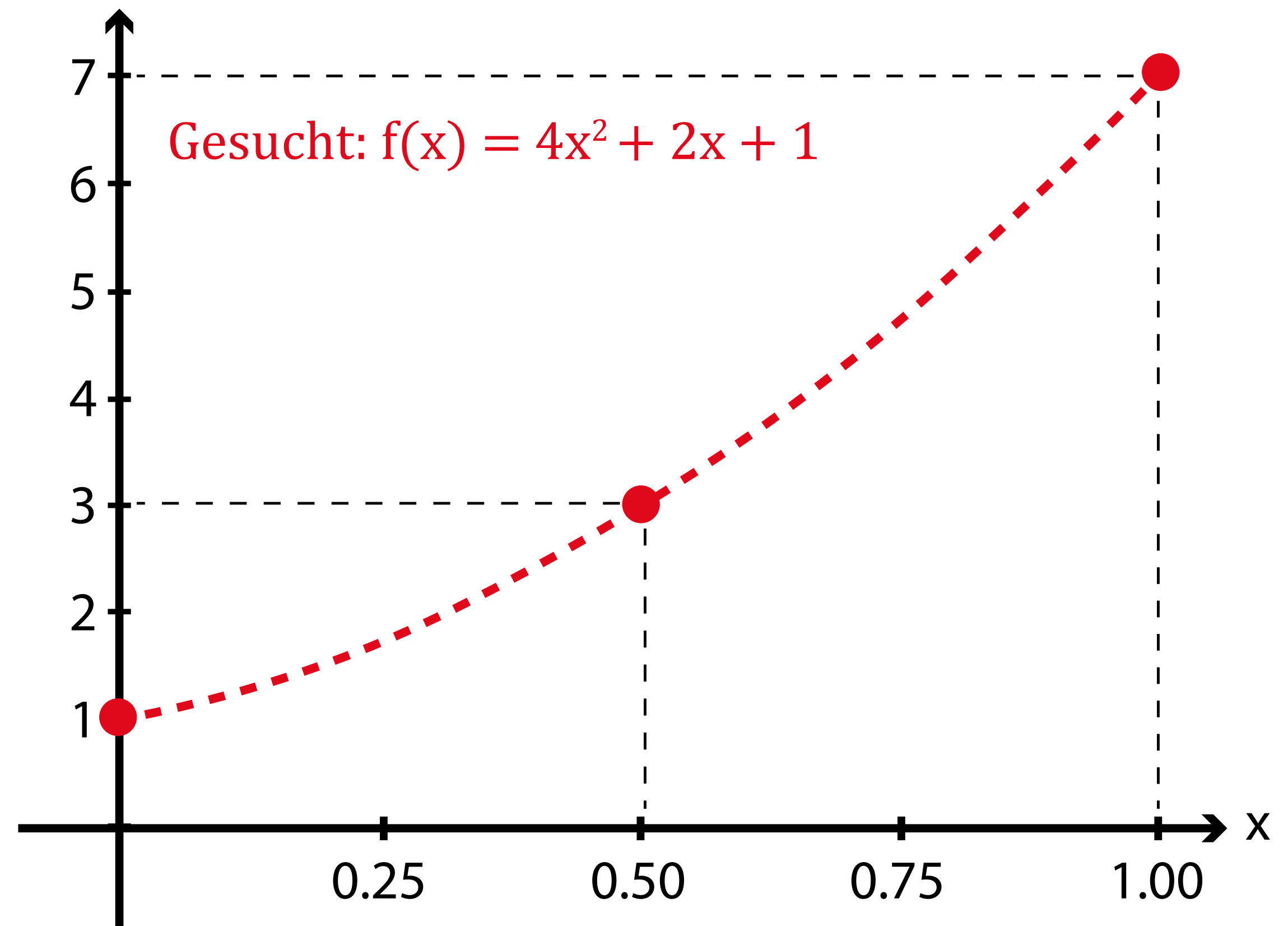
Inter- und Extrapolation

Das Polynom finden wir mit der Formel:

$$P_n(x) = \sum_{i=0}^n f(x_i) L_i(x)$$

Darin verwenden wir die Lagrange Basispolynome:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$$



$$L_0(x) = \prod_{\substack{j=0 \\ j \neq 0}}^2 \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_0 - x_1} \cdot \frac{x - x_2}{x_0 - x_2}$$

$$= \frac{x - 0.5}{0 - 0.5} \cdot \frac{x - 1}{0 - 1} = 2x^2 - 3x + 1$$

$$L_1(x) = \prod_{\substack{j=0 \\ j \neq 1}}^2 \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_1 - x_0} \cdot \frac{x - x_2}{x_1 - x_2}$$

$$= \frac{x - 0}{0.5 - 0} \cdot \frac{x - 1}{0.5 - 1} = -4x^2 + 4x$$

$$L_2(x) = \prod_{\substack{j=0 \\ j \neq 2}}^2 \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_2 - x_0} \cdot \frac{x - x_1}{x_2 - x_1}$$

$$= \frac{x - 0}{1 - 0} \cdot \frac{x - 0.5}{1 - 0.5} = 2x^2 - x$$

$$P_n(x) = \sum_{i=0}^n f(x_i) L_i(x)$$

$$= f(x_0) L_0(x) + f(x_1) L_1(x) + f(x_2) L_2(x)$$

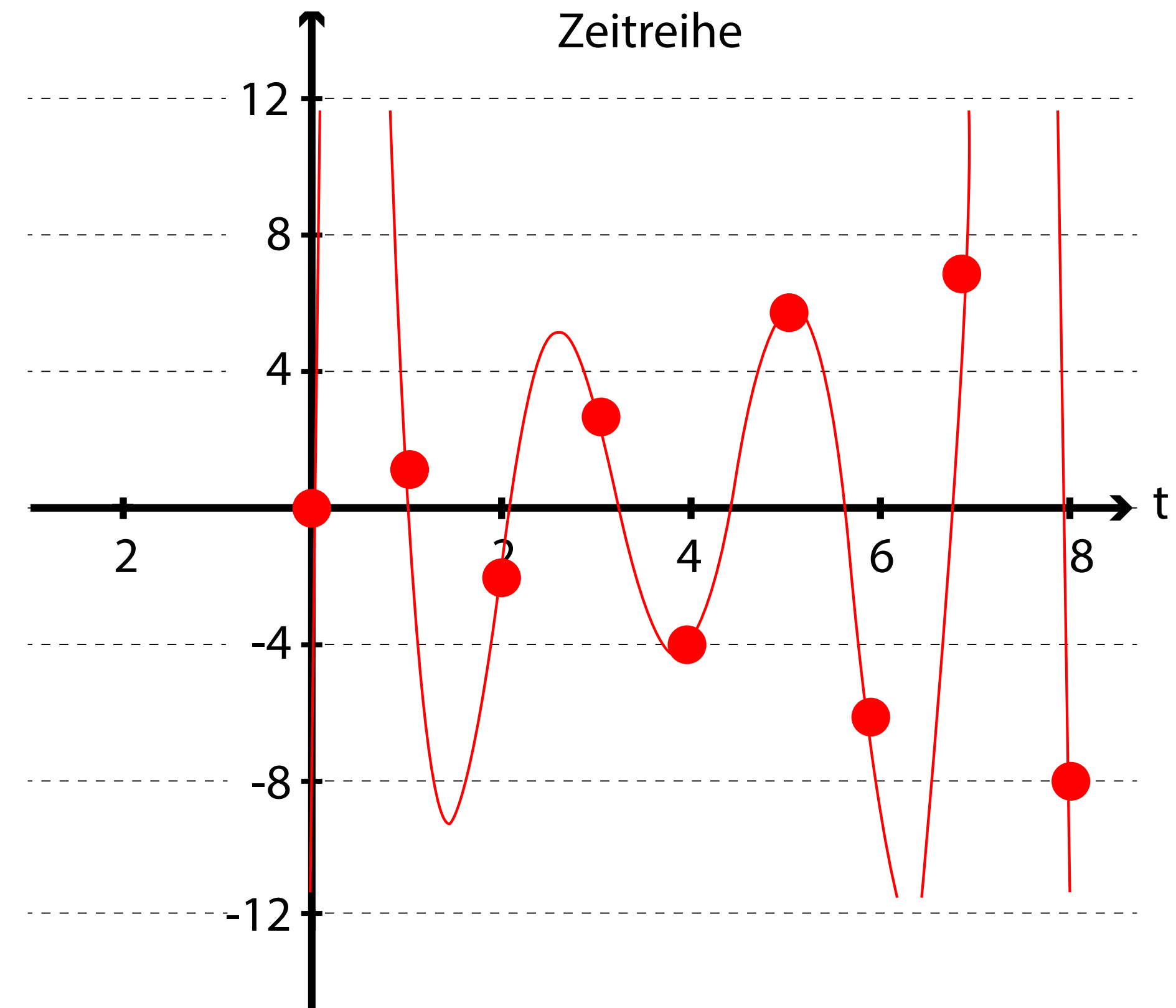
$$= 1 \cdot (2x^2 - 3x + 1) + 3 \cdot (-4x^2 + 4x) + 7 \cdot (2x^2 - x)$$

$$= 4x^2 - 2x + 1$$

Inter- und Extrapolation

Ab der fünften oder sechsten Ordnung werden die gefundenen Polynome instabil.

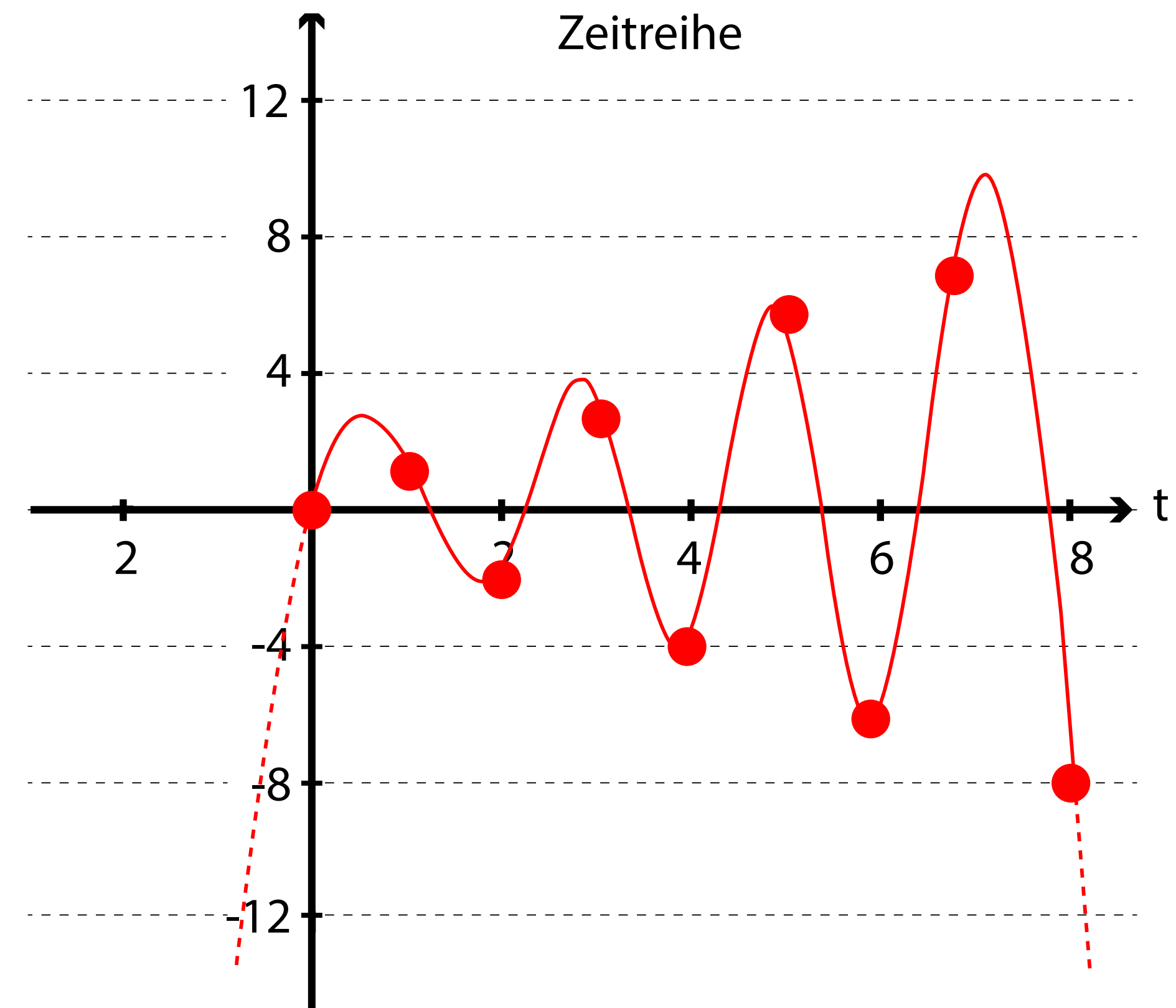
Sie weisen teilweise extreme Ausschläge, auf die nicht zu den gegebenen Daten passen.



Inter- und Extrapolation

Eine mögliche Lösung sind Splines: Wir teilen die zu suchende Funktion in Abschnitte auf und suchen für jeden Abschnitt ein passendes Polynom.

Der Vorteil ist ein deutlich stabilerer Verlauf, allerdings nehmen wir auch wieder die Angabe von mehreren Teilfunktionen in Kauf.

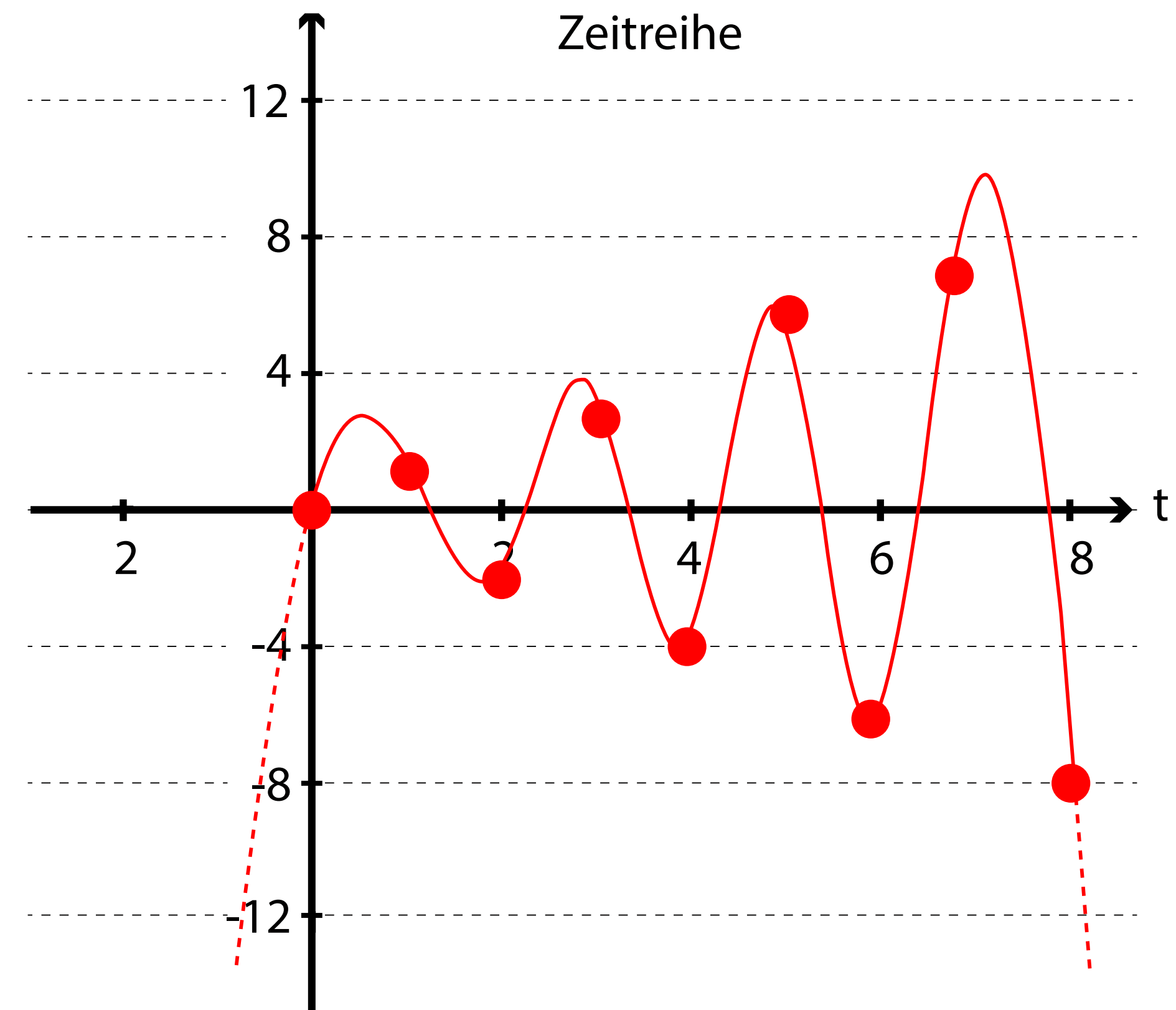


Inter- und Extrapolation

Verwechslungsgefahr! Sowohl bei der Interpolation als auch bei der Kurvenanpassung (beim *fitting*) suchen wir Funktionen, aber es gibt wesentliche Unterschiede:

Ausgangspunkt bei der Anpassung ist nicht zwingend eine Zeitreihe, sondern ganz allgemein eine Reihe an Beobachtungen zweier Merkmale x und y .

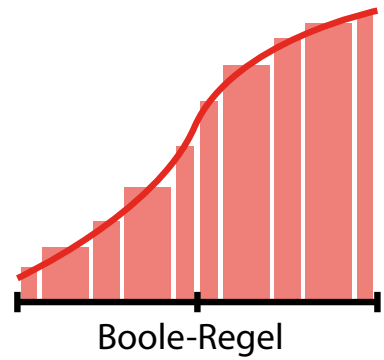
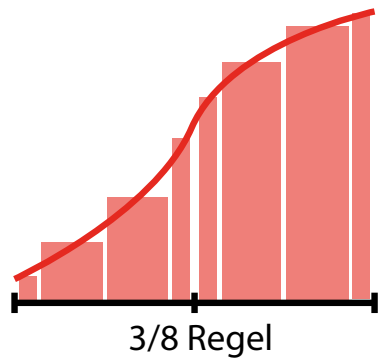
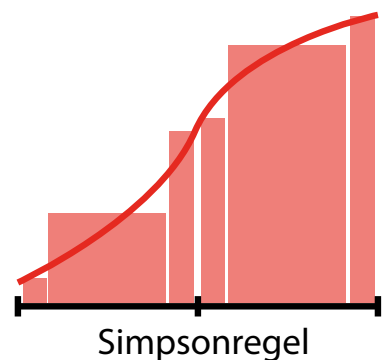
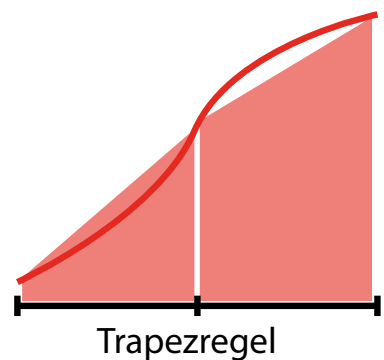
Gesucht ist eine Funktion, die möglichst gut zu diesen Beobachtungen passt. Die gegebenen Punkte müssen nicht exakt getroffen werden.



Newton-Cotes Formeln

Die Gewichtungen der rechts gezeigten Methoden werden aus den Lagrange Basispolynomen hergeleitet.

$$w_i = \int_a^b L_i(x) \, dx \quad L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$$



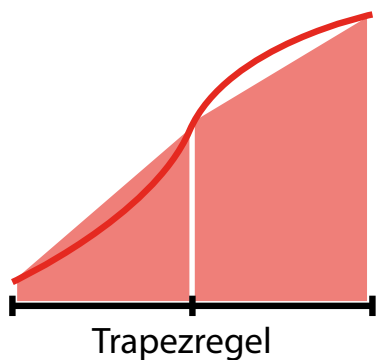
Stützstelle	0	1
Gewichtung	1/2	1/2

Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

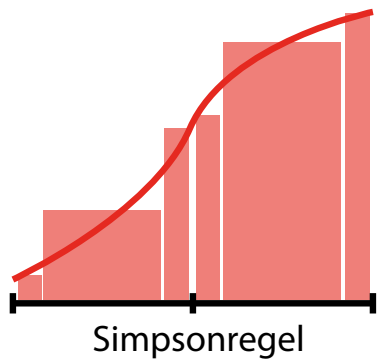
Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

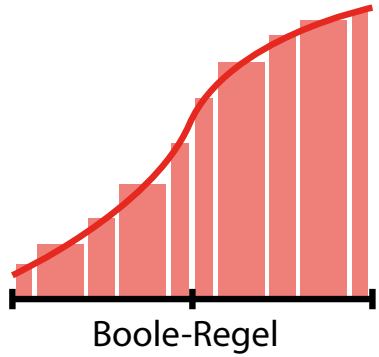
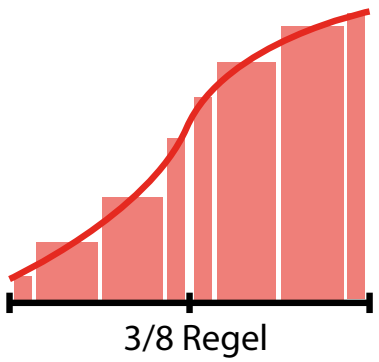
$$w_0 = \int_0^1 L_0(x) \, dx = \int_0^1 2x^2 - 3x + 1 \, dx$$
$$= \left[\frac{2}{3}x^3 - 1.5x^2 + x \right]_0^1 = \frac{1}{6}$$



$$w_1 = \int_0^1 L_1(x) \, dx = \int_0^1 -4x^2 + 4x \, dx$$
$$= \left[-\frac{4}{3}x^3 + 2x^2 \right]_0^1 = \frac{2}{3}$$



$$w_2 = \int_0^1 L_2(x) \, dx = \int_0^1 2x^2 - x \, dx$$
$$= \left[\frac{2}{3}x^3 - 0.5x^2 \right]_0^1 = \frac{1}{6}$$



Stützstelle	0	1
Gewichtung	1/2	1/2

Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

Newton-Cotes Formeln

Schau dir den Beispielcode "Newton-Cotes Basic" an.

- Ergänze die Simpsonregel in der 3/8-Variante und die Boole-Regel.
- Teste alle Regeln mit den gegebenen Integranden
- Überlege theoretisch, welche Integranden Trapez- und Simpsonregel exakt auswerten können sollten.

Integral	Sollwert	Trapez	Simpson	Boole
$\int_0^1 e^x dx$	1.71828182	0.02083%		
$\int_0^1 \frac{dx}{1+x^2}$	0.78539816	0.01326%		
$\int_0^1 \sqrt{\ln(x+1)} dx$	0.59259042	0.38105%		
$\int_0^1 1 + \sin(2\pi x) dx$	1.00000000	Exakt		

Newton-Cotes Formeln

Die Simpson-Regel und die Boole-Regel sind bei den ersten drei Integranden bei gleicher Anzahl an Funktionsauswertungen genauer.

Trapezregel ist nur bei konstanten und linearen Funktionen exakt.

Die Simpsonregel bei Polynomen erster, zweiter und erstaunlicherweise auch dritter Ordnung.

Integral	Sollwert	Trapez	Simpson	Boole
$\int_0^1 e^x dx$	1.71828182	0.02083%	$< 10^{-5} \%$	$< 10^{-11} \%$
$\int_0^1 \frac{dx}{1+x^2}$	0.78539816	0.01326%	$< 10^{-7} \%$	$< 10^{-9} \%$
$\int_0^1 \sqrt{\ln(x+1)} dx$	0.59259042	0.38105%	0.15351%	0.04759%
$\int_0^1 1 + \sin(2\pi x) dx$	0 .00000000 1	Exakt	$< 10^{-13} \%$	Exakt

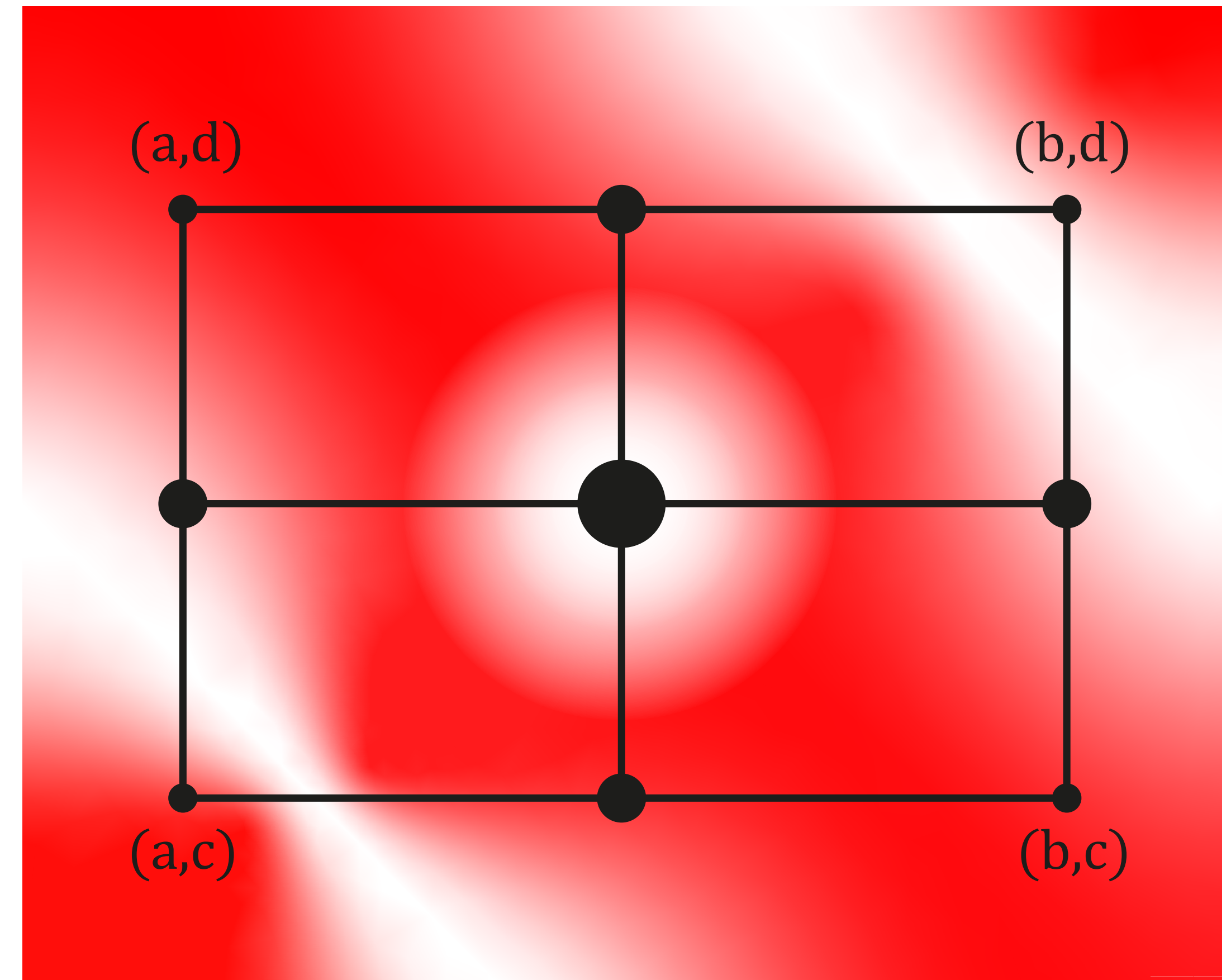
Newton-Cotes Formeln

Die Newton-Cotes Formeln können für Skalarfelder $\mathbb{R}^n \rightarrow \mathbb{R}$ verallgemeinert werden. Die Simpsonregel in $n = 2$ Dimensionen wäre:

$$I = \int_a^b \int_c^d f(x,y) \, dx \, dy$$

$$I \approx \frac{(b-a)(d-c)}{9} \sum_{i=0}^2 \sum_{j=0}^2 w_i w_j f(x_i, y_j)$$

$$w = (1, 4, 1)$$

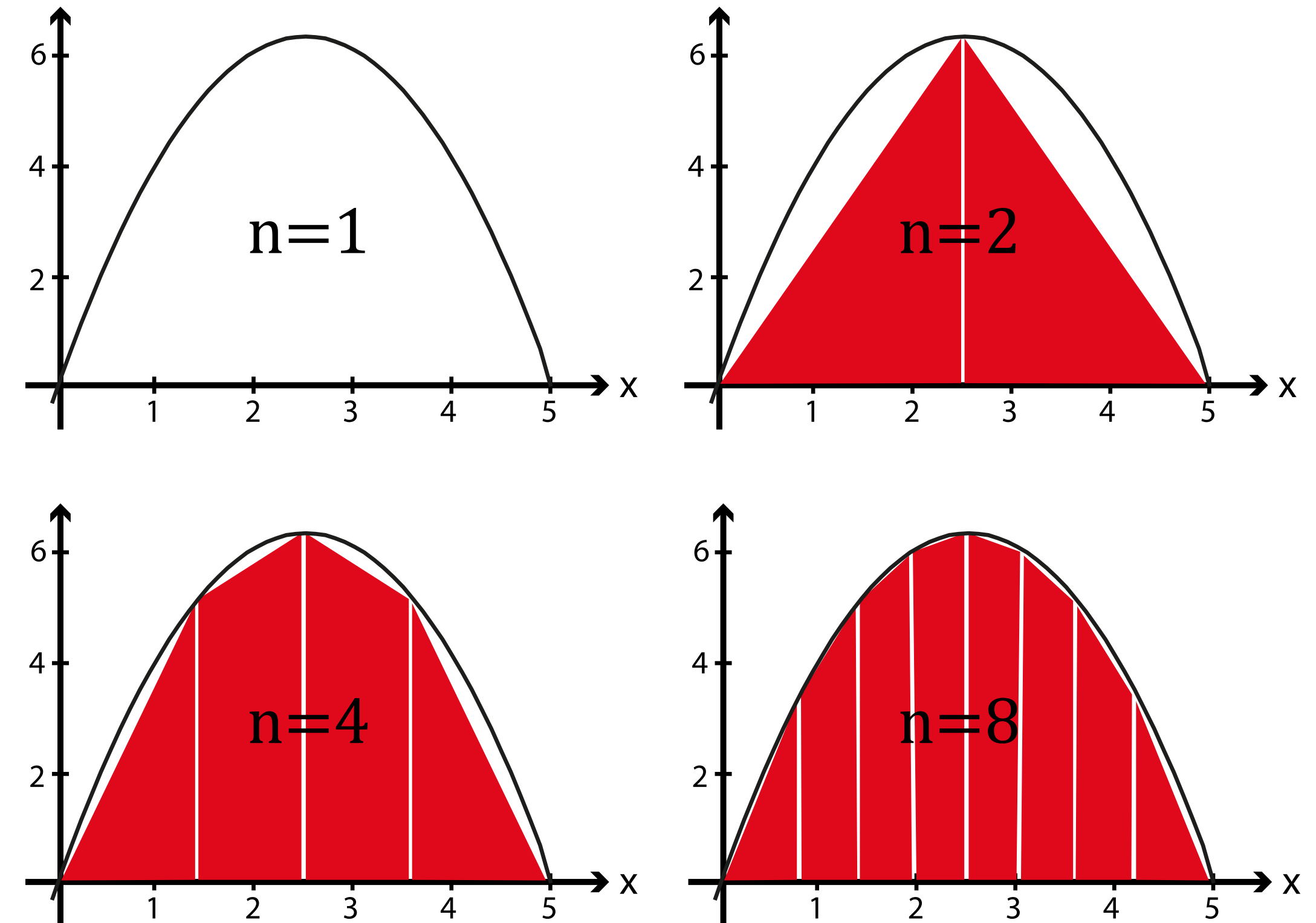


Romberg Methode

Die Rombergmethode erweitert die Trapezmethode.

Statt die Methode, einmal mit einer festen Zahl an Trapezen anzuwenden, wird sie mehrfach mit unterschiedlichem n angewendet, um unterschiedlich genaue Resultate zu bekommen.

Mithilfe der Richardson Extrapolation werden diese anschließend verrechnet, um insgesamt eine sehr genaue Näherung zu bekommen.

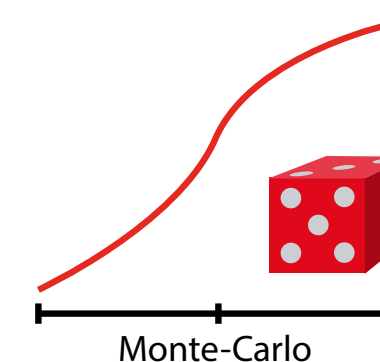
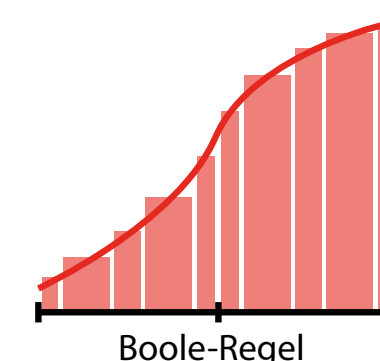
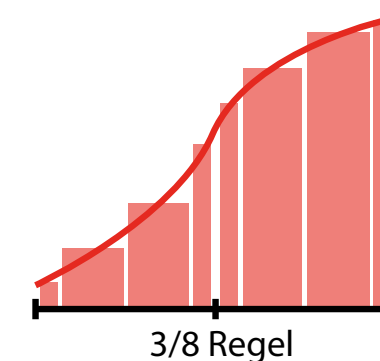
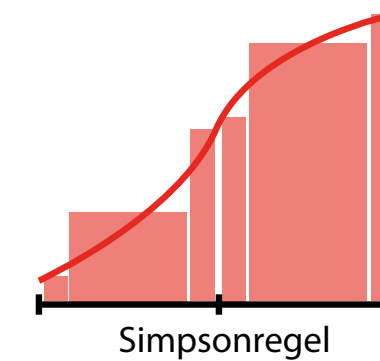


Monte Carlo Integration

Neben den Newton-Cotes Formeln gibt es noch eine ganz andere Herangehensweise für die numerische Integration.

Wir wählen die Stellen, an denen wir die Funktion auswerten, zufällig aus!

Warum sollte man das tun?



Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

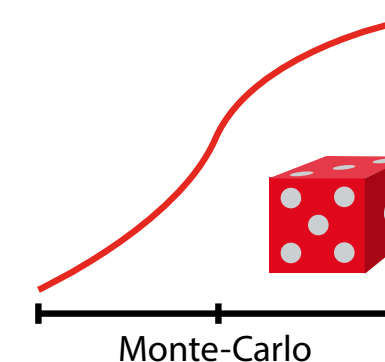
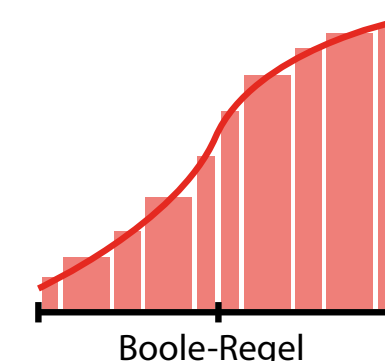
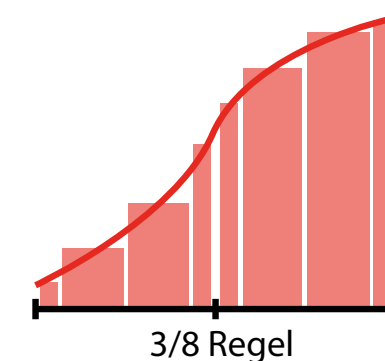
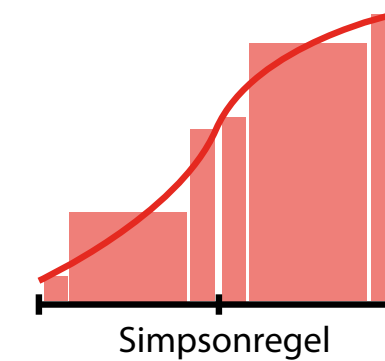
Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

Stützstelle					
Gewichtung	1/n	1/n	1/n	1/n	1/n

Monte Carlo Integration

Die Vorgehensweise stützt sich nicht auf die Approximation durch eine bestimmte Art von Funktion.

Mit einer ausreichend großen Zahl an Auswertungen sollte eine hinreichend gute Genauigkeit erzielt werden können.



Stützstelle	0	1/2	1
Gewichtung	1/6	4/6	1/6

Stützstelle	0	1/3	2/3	1
Gewichtung	1/8	3/8	3/8	1/8

Stützstelle	0	1/4	1/2	3/4	1
Gewichtung	7/90	32/90	12/90	32/90	7/90

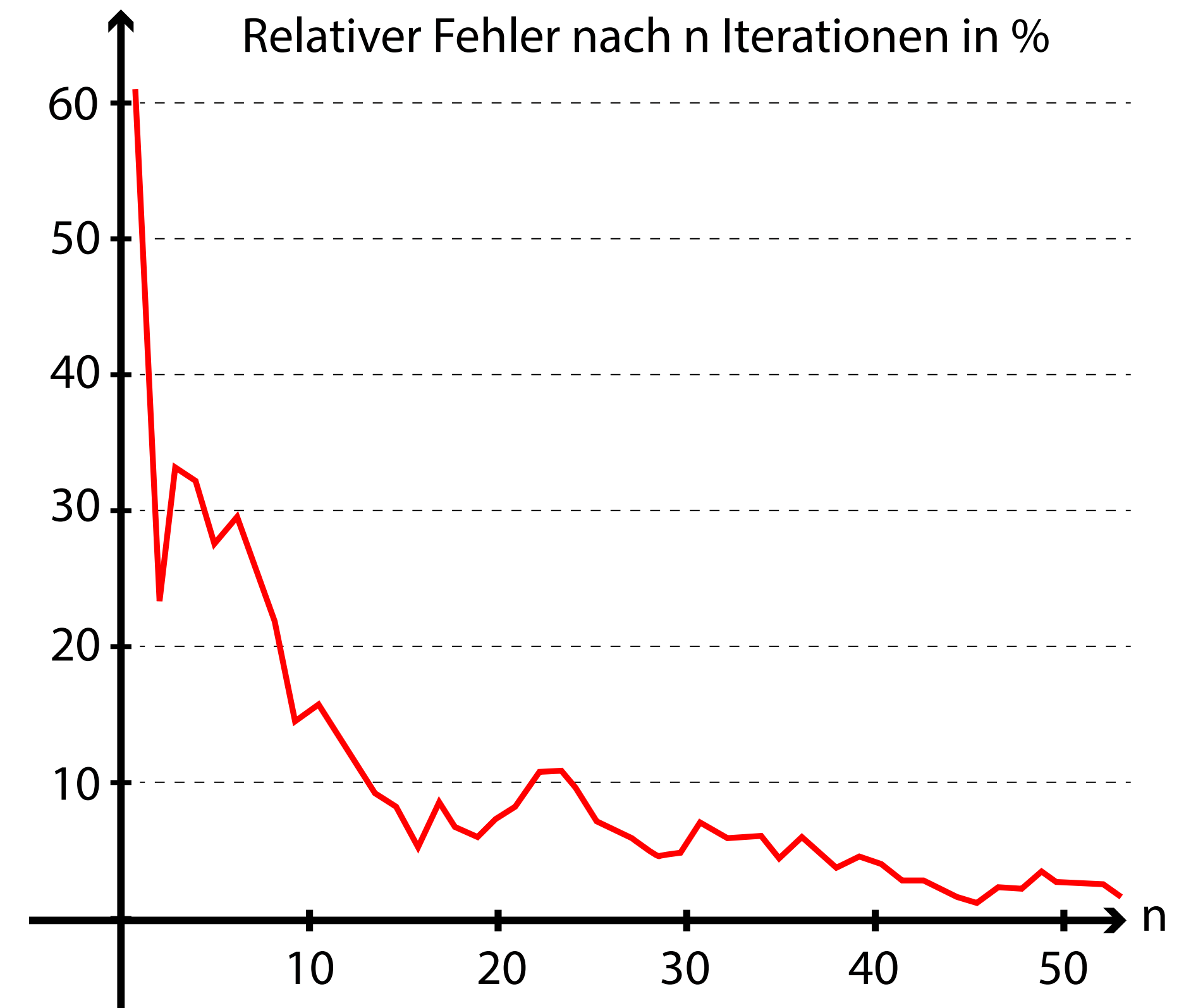
Stützstelle					
Gewichtung	1/n	1/n	1/n	1/n	1/n

Monte Carlo Integration

Trotzdem gibt es einen entscheidenden Nachteil: Das Verfahren ist nicht deterministisch!

Sowohl das Ergebnis als auch der absolute und relative Fehler sind Zufallsvariablen.

Die Genauigkeit in einem einzelnen Durchlauf reicht nicht aus, um das Verfahren zu bewerten!

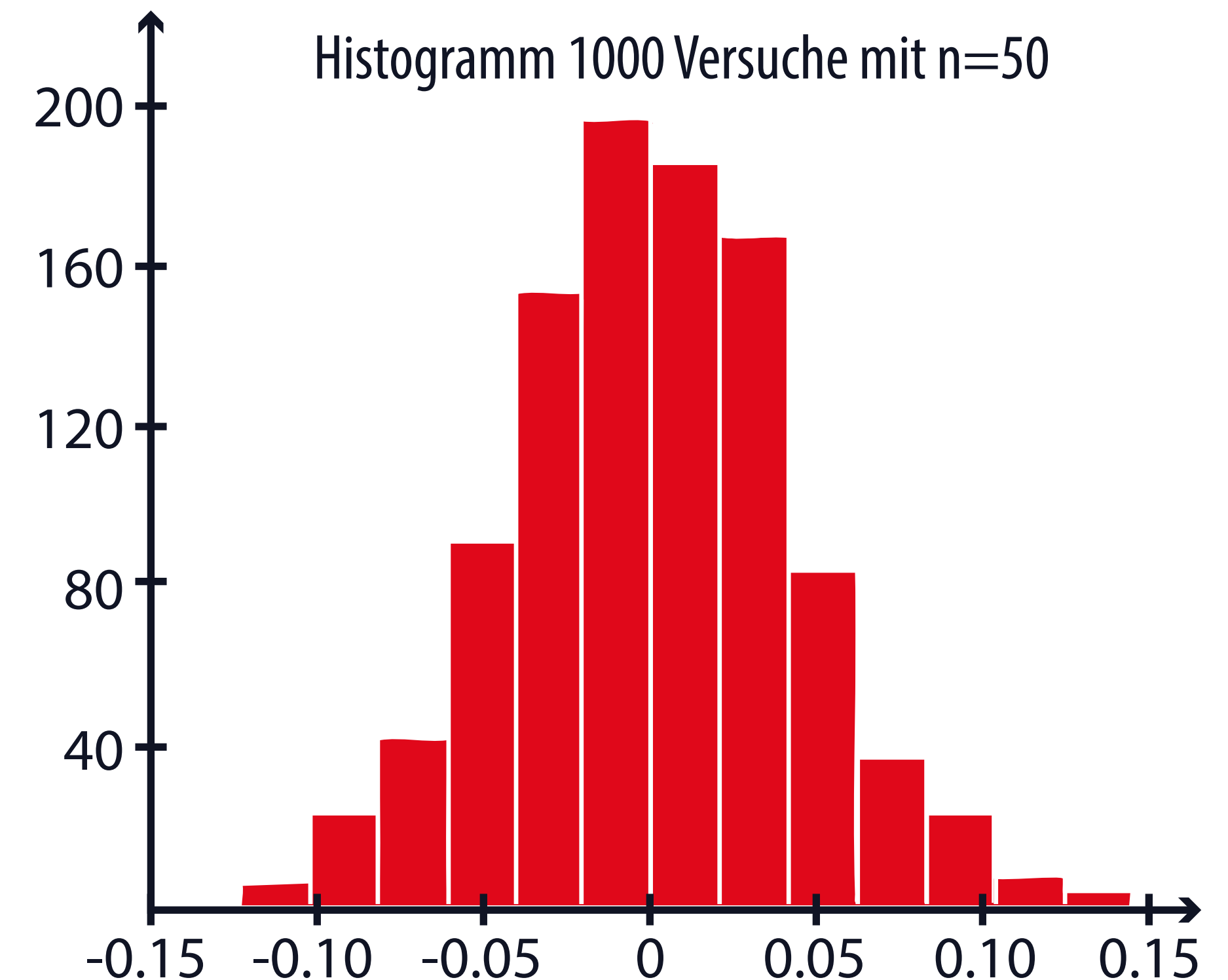


Monte Carlo Integration

Betrachten wir stattdessen die Verteilung des Fehlers über mehrere Durchläufe, stellen wir fest:

Der Fehler ist näherungsweise normalverteilt mit Erwartungswert $\pm 0\%$. Das Verfahren hat keinen Bias, sondern konvergiert gegen den wahren Wert.

Der Verteilung hat eine hohe Varianz. Einzelne Durchläufe zeigen bei $n=50$ Fehler von über $\pm 10\%$ auf.

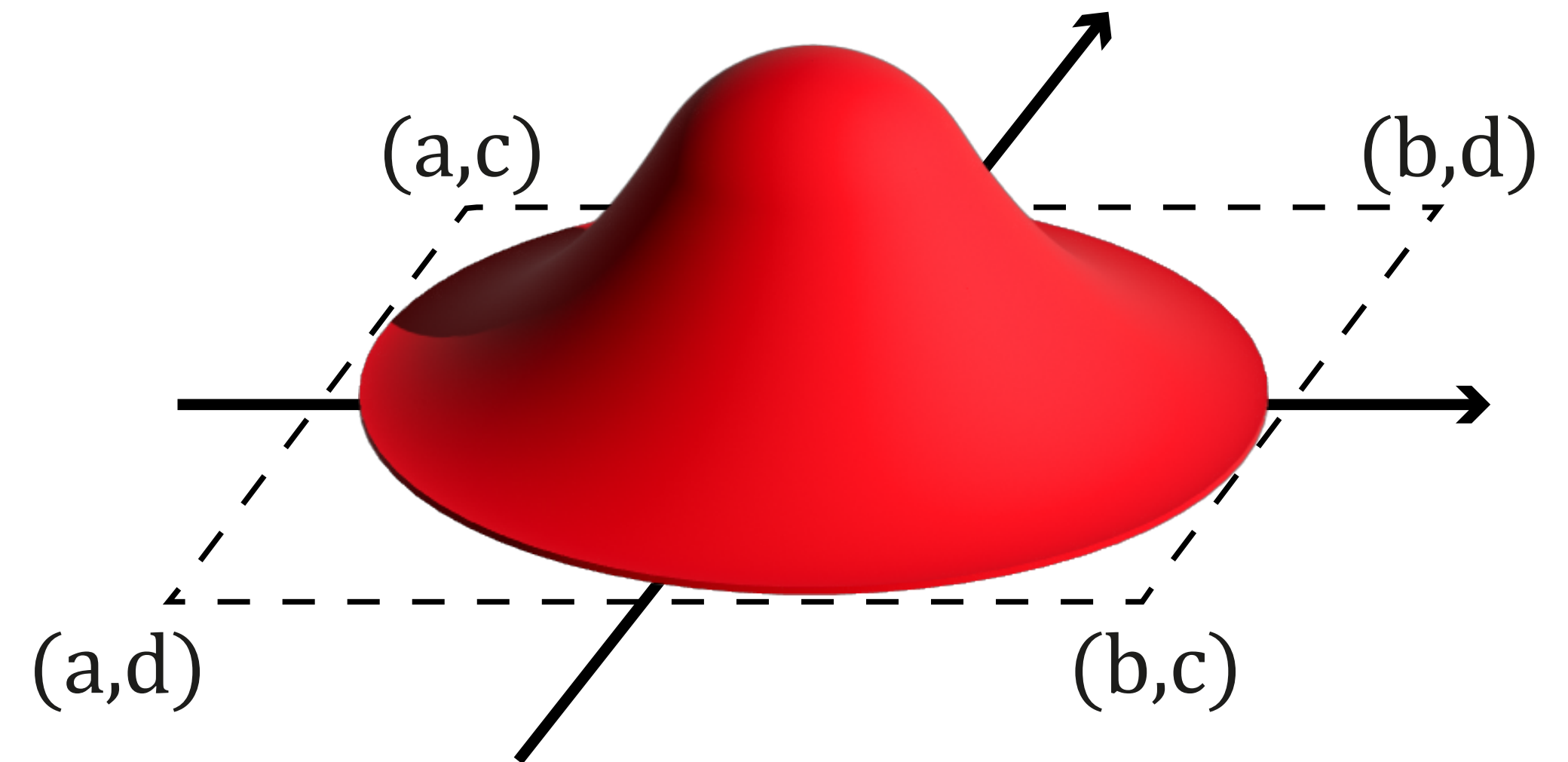


Monte Carlo Integration

Für die hier gezeigten Beispiele ist die Monte-Carlo-Integration tatsächlich nicht die beste Methode.

Interessant ist sie dagegen für die numerische Integration von mehrdimensionalen Integranden.

Je nach Integrand ist es ratsam, die zufälligen Punkte nicht aus der Gleichverteilung zu ziehen!

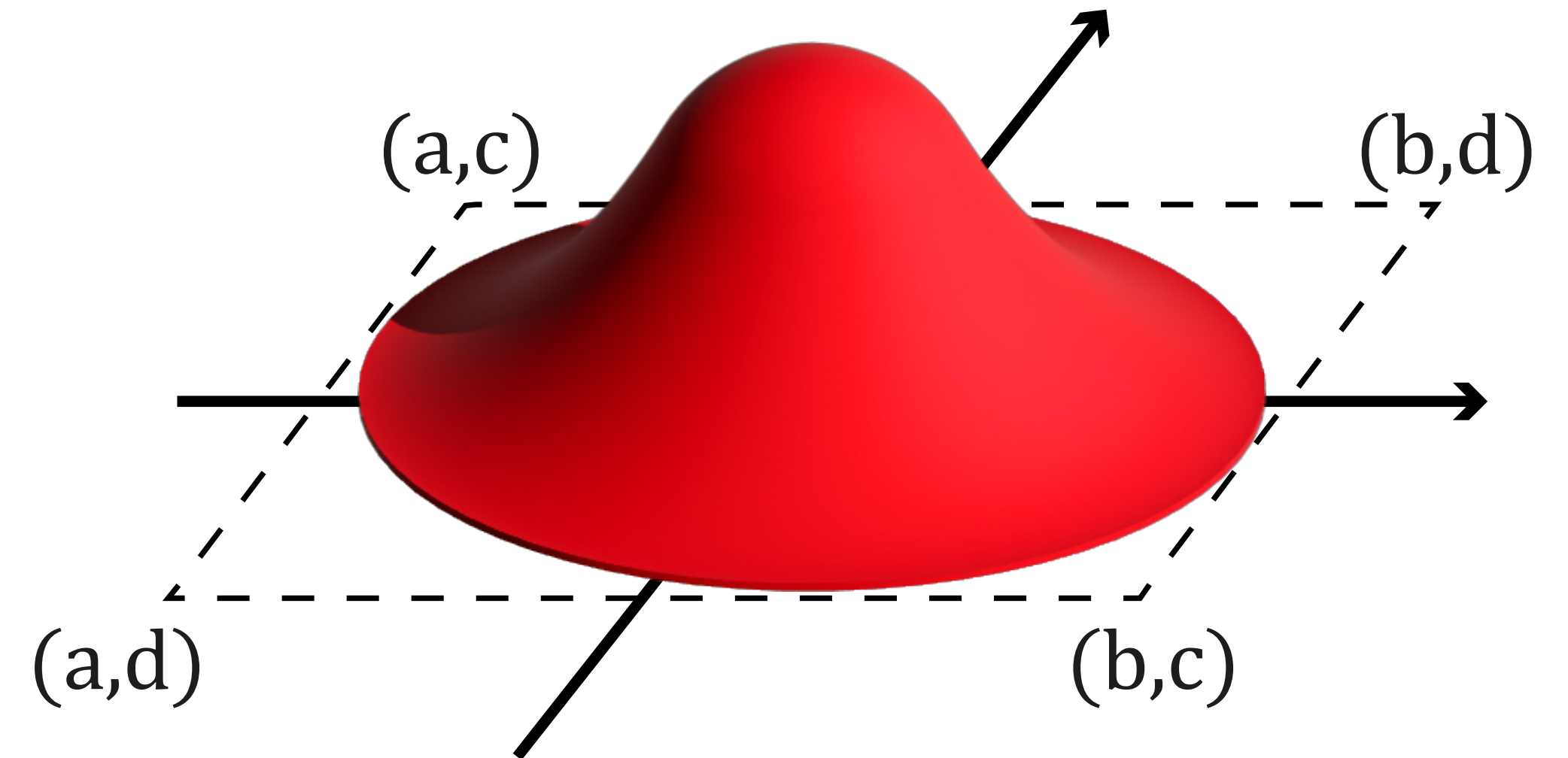


Monte Carlo Integration

Um die Monte Carlo Integration zu verbessern, betreiben wir **importance sampling**. Wir wählen eine Verteilung für die Punkte, die dazu führt dass ...

... weniger Punkte in Bereichen fallen, wo die Funktion einen konstanten Wert hat.

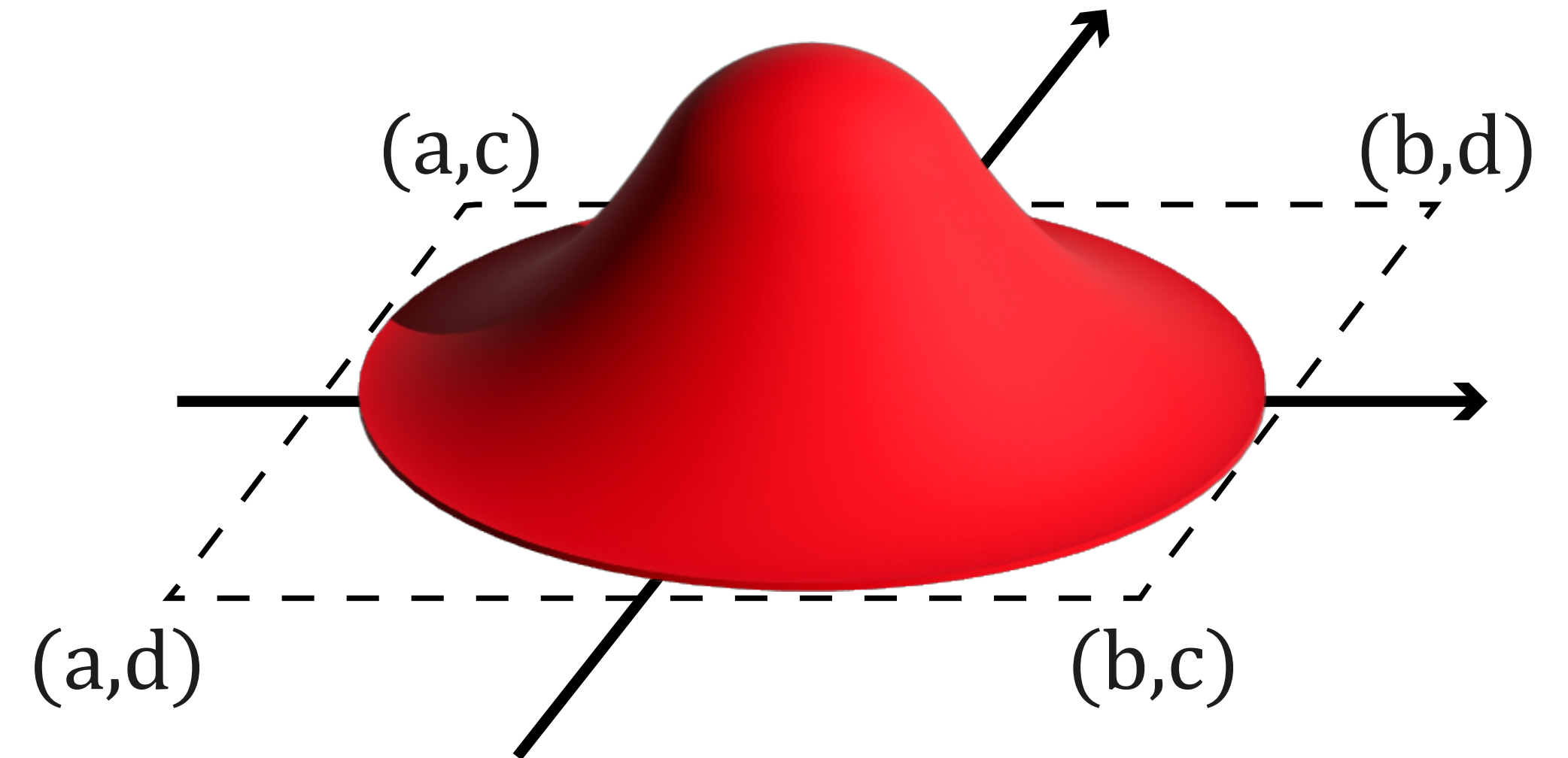
... weniger Punkte in Bereichen fallen, wo die Funktion keinen Beitrag liefert, d. h., 0 ist.



Monte Carlo Integration

Damit keine Verzerrung entsteht und das Ergebnis erwartungstreu bleibt, müssen wir die Funktionsauswertungen gewichten.

Das Gewicht ist der Kehrwert der relativen Wahrscheinlichkeit. Ist ein Punkt doppelt so wahrscheinlich, wird er nur halb so stark gewichtet.

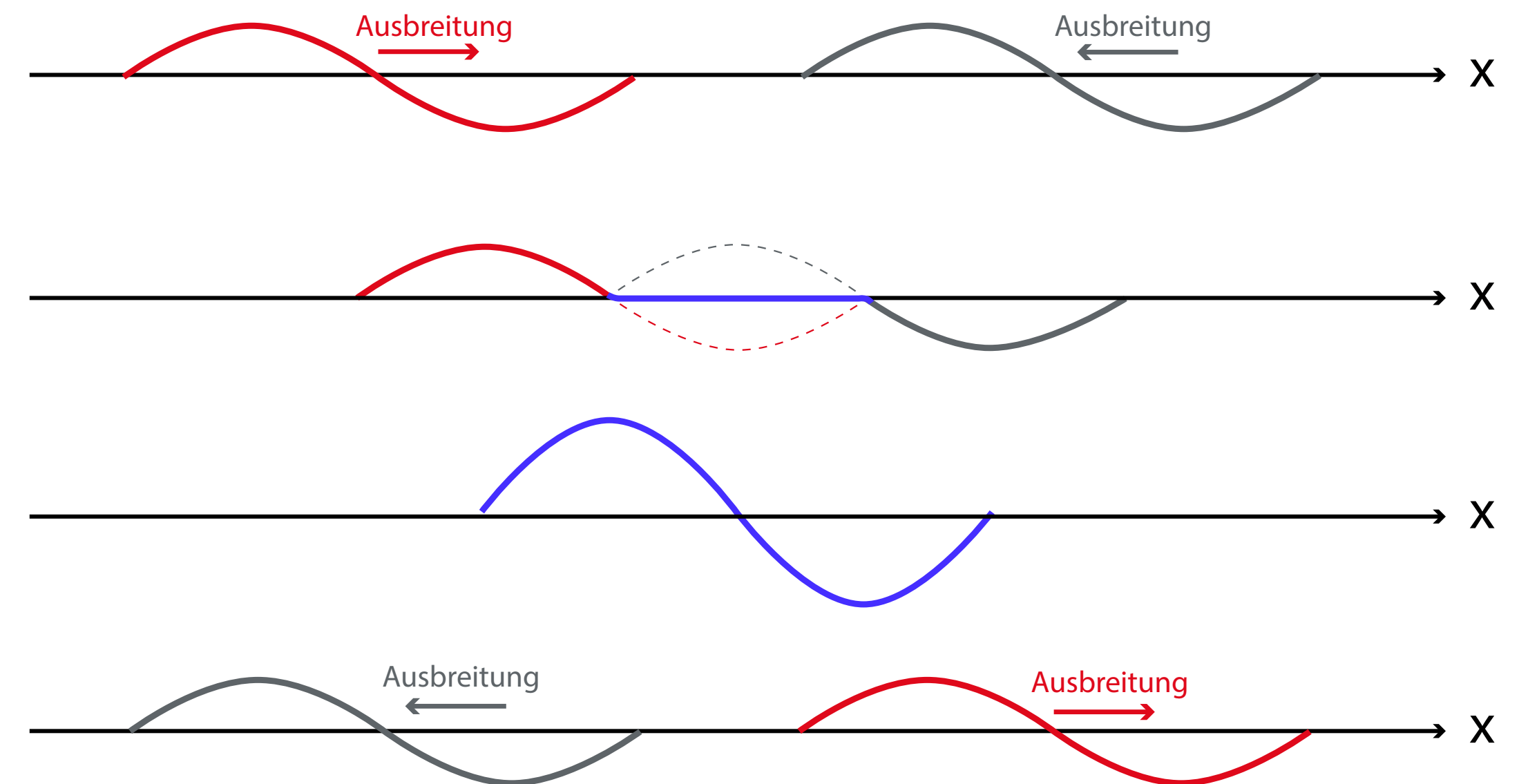


Fouriertransformation

Was passiert, wenn zwei Wellen aufeinandertreffen? Die Antwort hängt natürlich von der Art dieser Wellen ab, wobei wir sehr häufig Linearität annehmen können:

Die Wellen beeinflussen sich nicht.

Die Amplitude an einem bestimmten Ort und Zeitpunkt entspricht der Summe der beiden Teilamplituden.

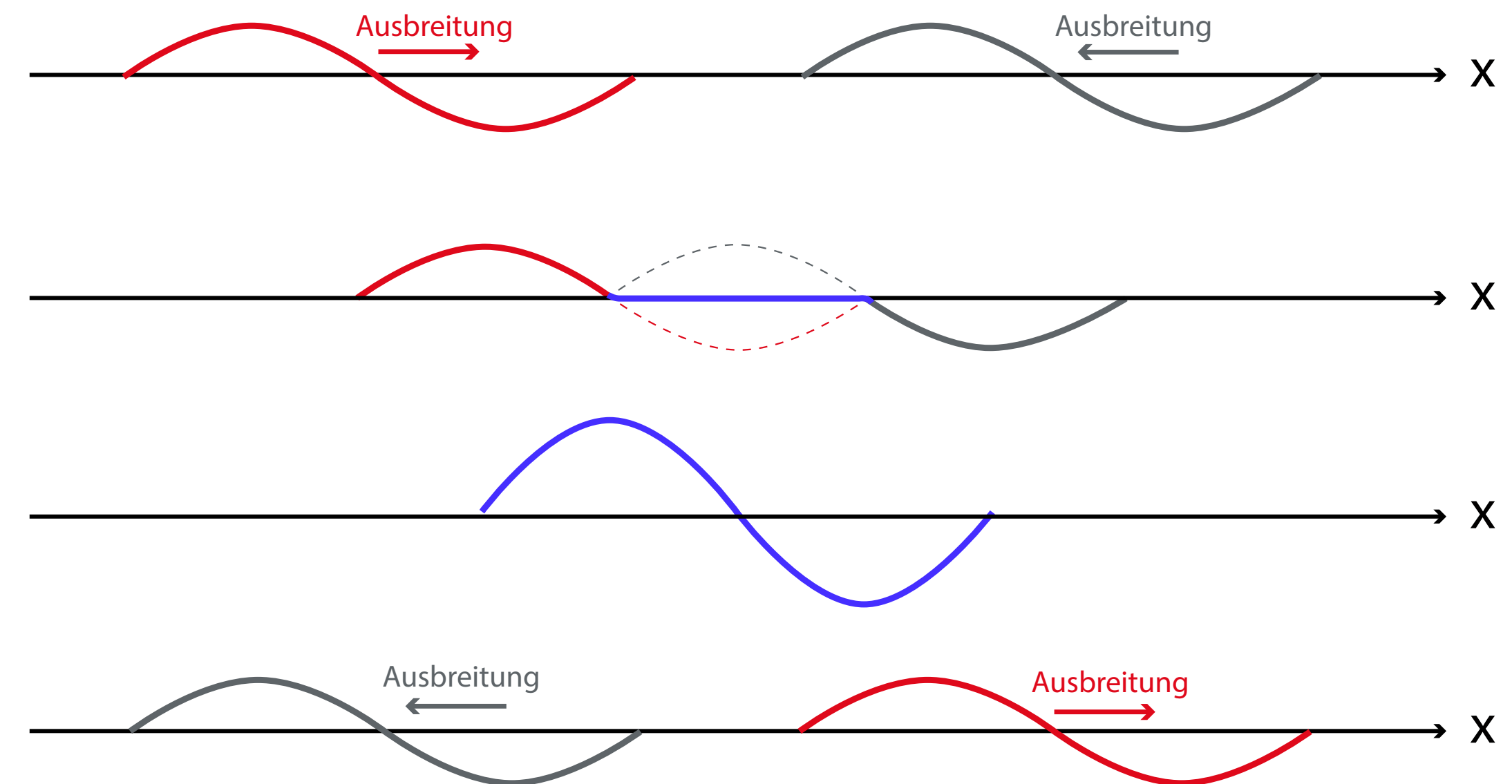


Fouriertransformation

Destruktive Interferenz Sind die Wellen gegenphasig, dann schwächen sich die Amplituden ab.

Konstruktive Interferenz Sind die Wellen gleichphasig, dann verstärken sich die Amplituden.

Die einzelnen Wellen bleiben aber unverändert.

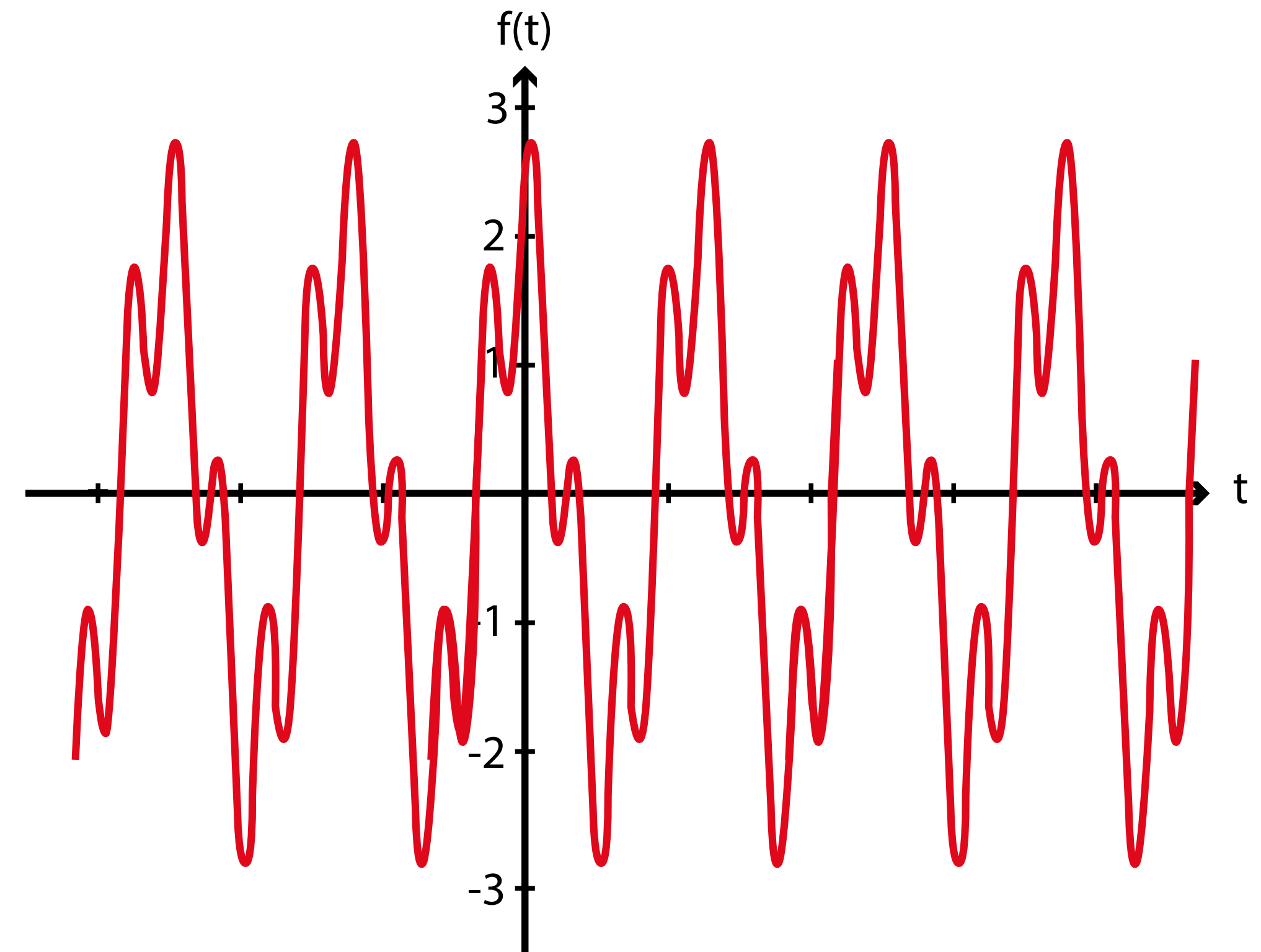


Fouriertransformation

Überlagern sich mehrere Wellen an einem bestimmten Ort, ergibt sich daraus eine Gesamtwellenfunktion $f(t)$.

Die Gesamtwellenfunktion $f(t)$ enthält alle einzelnen Wellen, aber wie können wir diese einzelnen Wellen voneinander trennen?

Aus welchen Wellen besteht der rechts gezeigte Graph?



Fouriertransformation

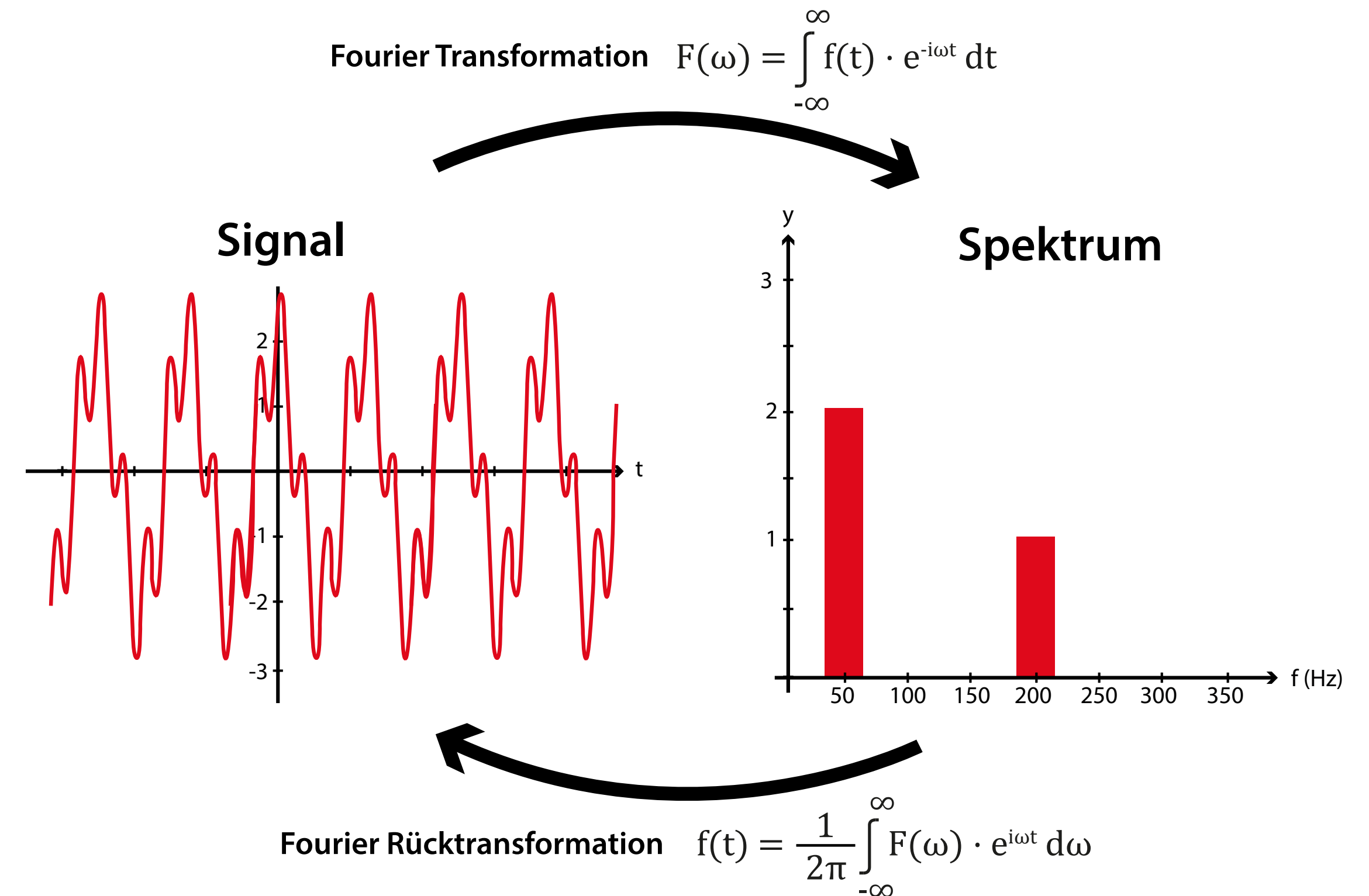
Die Fouriertransformation ist eine Abbildung der zeitlichen Darstellung $f(t)$ in eine spektrale Darstellung $F(\omega)$.

$f(t)$ Amplitude abhängig von der Zeit

$|F(\omega)|$ Anteil der Frequenz am Gesamtsignal

Mathematische Grundlage ist die Darstellung einer Welle als Exponentialfunktion mit komplexem Exponent:

$$A \cdot [\cos(\omega t) + i \cdot \sin(\omega t)] = A e^{i\omega t}$$



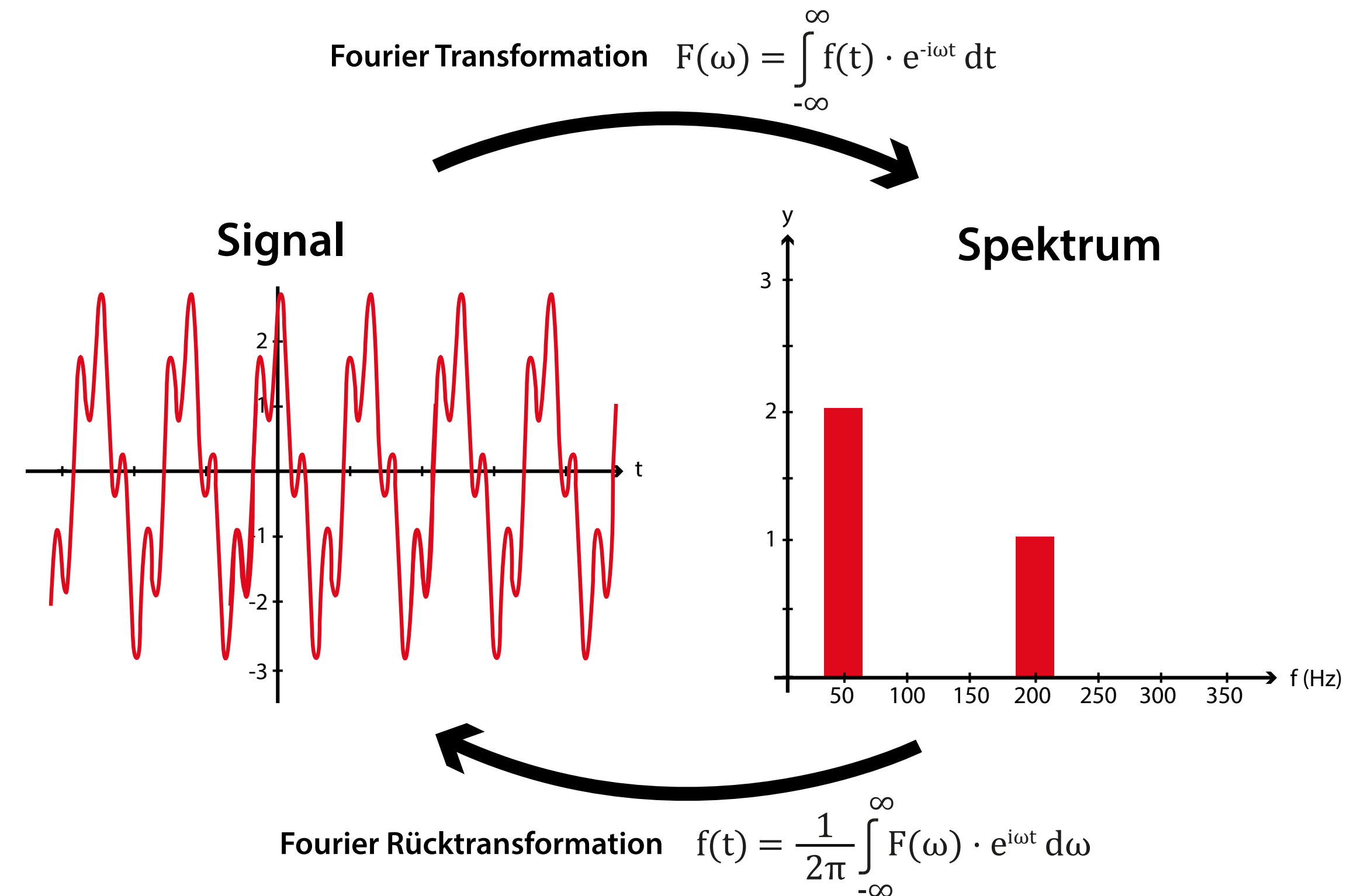
Fouriertransformation

Die Transformation vom Signal zum Spektrum erfolgt durch die Integraltransformation:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{-i\omega t} dt$$

Die Rücktransformation vom Spektrum zum Signal erfolgt analog dazu über:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \cdot e^{i\omega t} d\omega$$

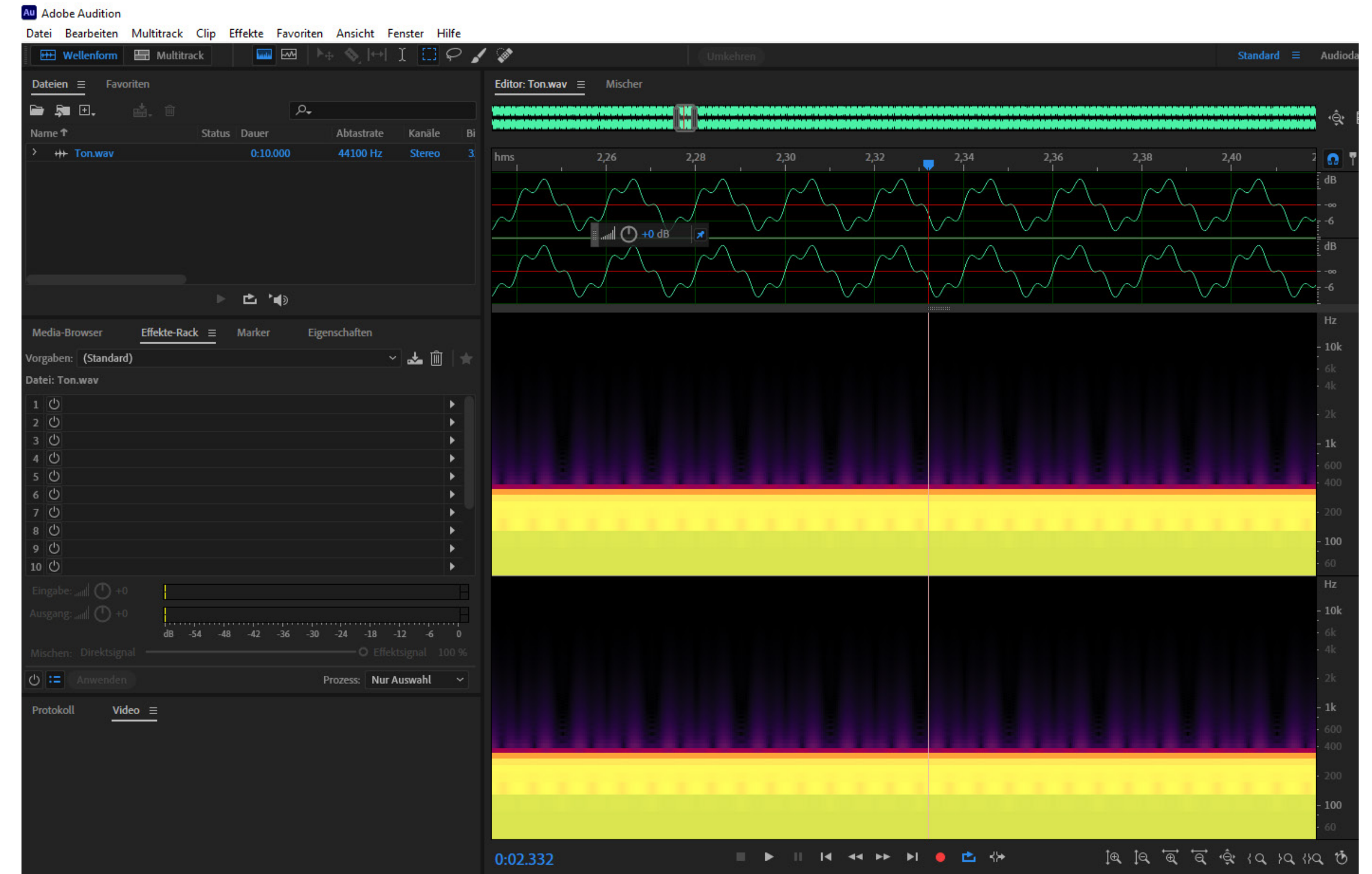


Fouriertransformation

Interessant für Physiker, Ingenieure und Tontechniker, aber was hat Data Science damit zu tun?

In der Data Science analysieren wir unsere Daten unter anderem auch auf Periodizität. Welche zyklischen Effekte finde ich in meinen Daten?

Sobald es mehrere zyklischen Effekte gibt und noch Trends dazukommen, können wir die Periodizität nicht mehr mit einem Liniendiagramm und bloßem Auge identifizieren.

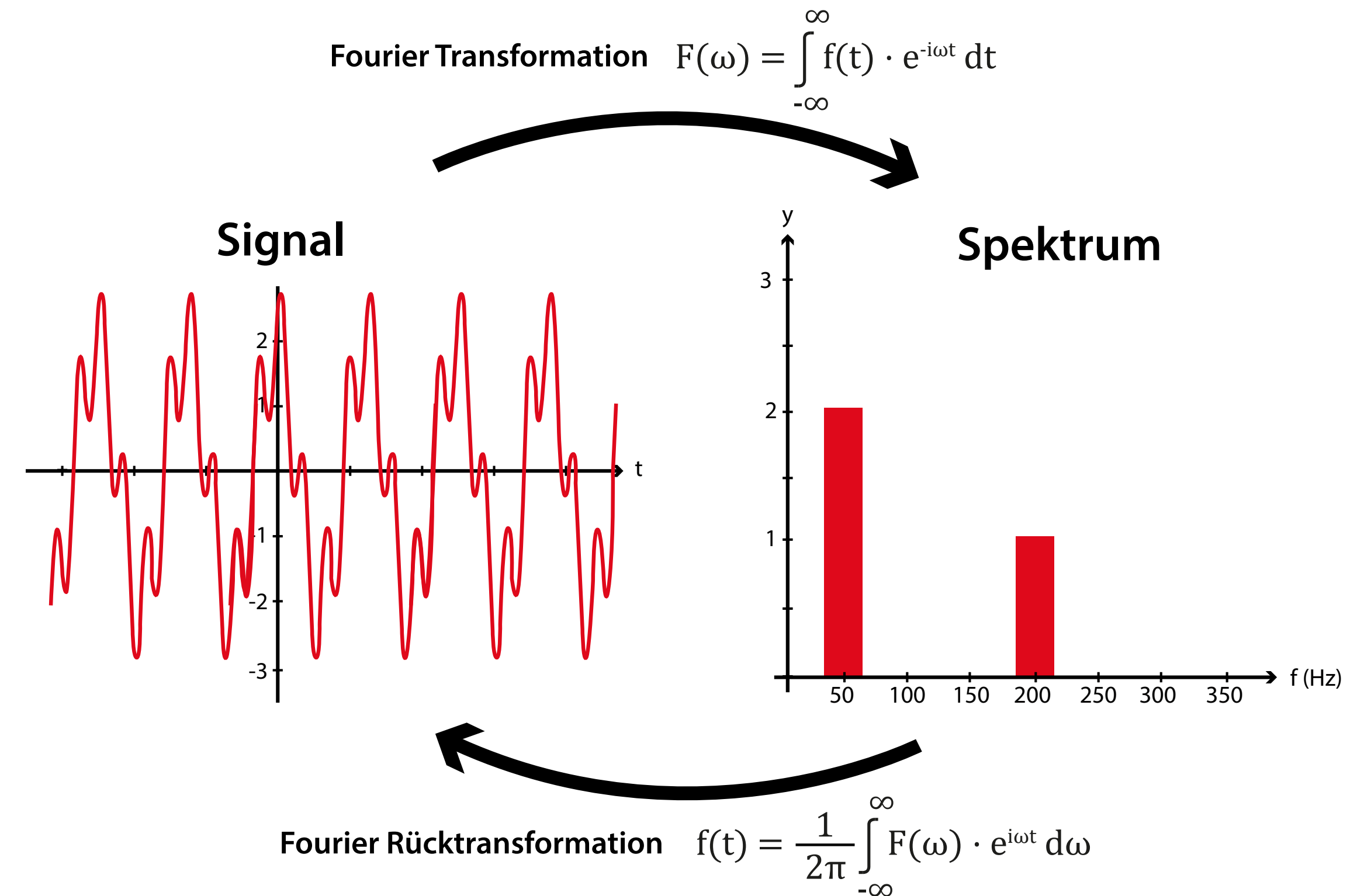


Fouriertransformation

Mit der Fourieranalyse können wir Zeitreihen auf periodische Effekte untersuchen. Aber eine Zeitreihe ist doch keine Funktion?

Es gibt eine diskrete Variante der Fouriertransformation (DFT), die wir für Zeitreihen verwenden können.

Wir erreichen damit aber wieder den Punkt, wo analytische Lösungen von Hand unmöglich oder zumindest unpraktikabel sind.

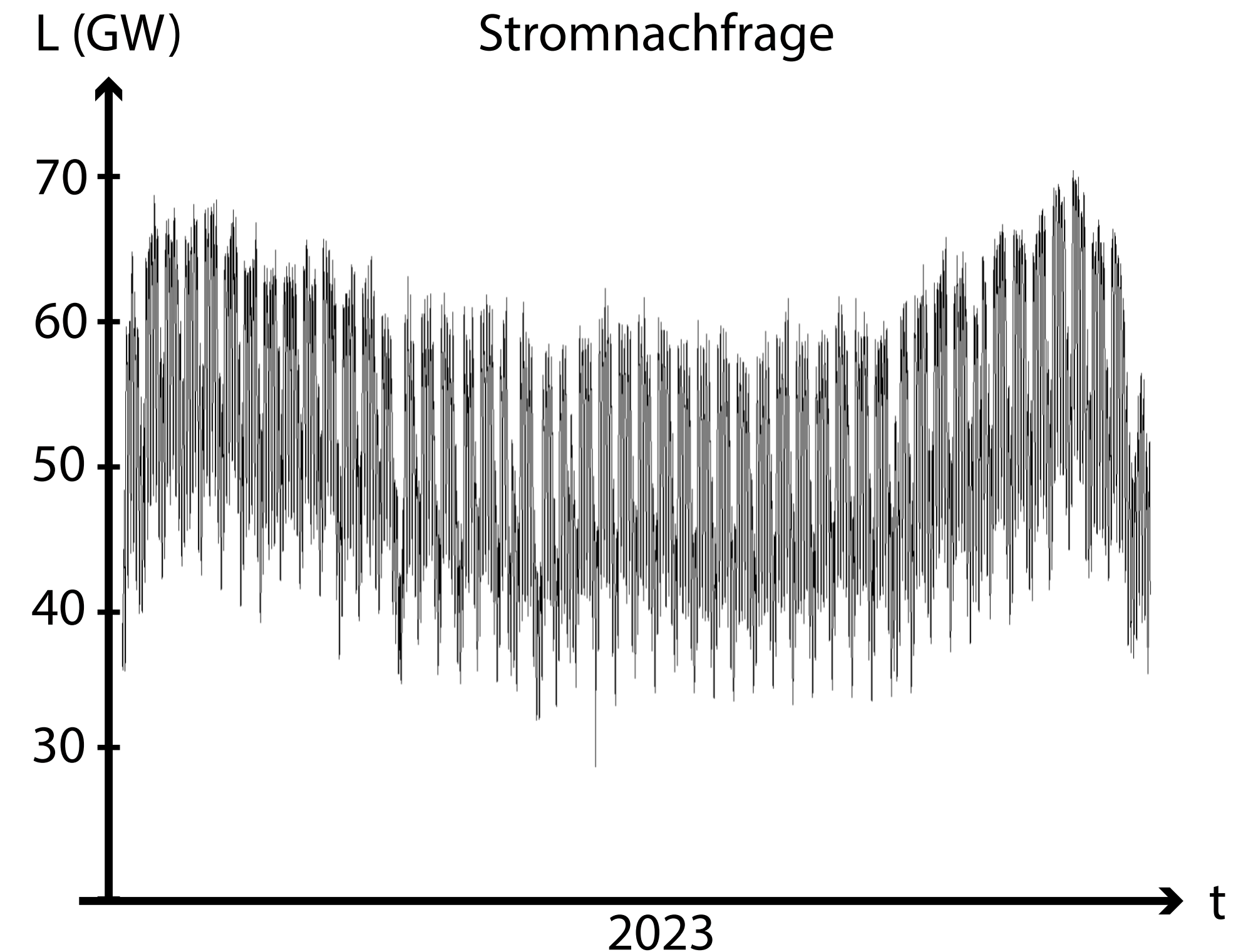


Fouriertransformation

Wir verwenden den numerischen Fast Fourier Transformation (FFT) Algorithmus.

Rechts sehen wir die Netzlast (Stromverbrauch) in Deutschland über das Jahr 2023. Welche Zyklen haben diese Daten?

Das Vorgehen ist dasselbe wie bei unserem Tonbeispiel ...



Fouriertransformation

In unserem Frequenzspektrum sehen wir deutliche Peaks bei:

0.006x pro Stunde Wöchentlich

0.042x pro Stunde Täglich

0.084x pro Stunde Alle 12 Stunden

Als Übungsaufgabe führen wir dieselbe Analyse mit der Stromproduktion durch. Welche Periodizitäten gibt es auf der Angebotsseite? Decken sich diese mit der Nachfrage?

