Die hyperbolische e-Funktion

Glättung singulärer Modelle durch Simulation

Wissenschaftliche Abhandlung

Klaus H. Dieckmann

September 2025

Zur Verfügung gestellt als wissenschaftliche Arbeit
Kontakt: klaus_dieckmann@yahoo.de

Metadaten zur wissenschaftlichen Arbeit

Titel:	Die hyperbolische e-Funktion
Untertitel:	Glättung singulärer Modelle durch Simulation
Autor:	Klaus H. Dieckmann
Kontakt:	klaus_dieckmann@yahoo.de
Phone:	0176 50 333 206
ORCID:	0009-0002-6090-3757
DOI:	10.5281/zenodo.17189280
Version:	September 2025
Lizenz:	CC BY-NC-ND 4.0
Zitatweise:	Dieckmann, K.H. (2025). Glättung singulärer Modelle durch Simulation

Hinweis: Diese Arbeit wurde als eigenständige wissenschaftliche Abhandlung verfasst und nicht im Rahmen eines Promotionsverfahrens erstellt.

Abstract

In komplexen physikalischen und technischen Systemen treten häufig Singularitäten auf, sei es in Form zeitlicher Sprünge, impulsartiger Anregungen oder struktureller Diskontinuitäten in Netzwerken. Diese idealisierten Modelle, oft durch Heaviside oder Delta-„Funktionen“ beschrieben, sind zwar mathematisch handhabbar, führen jedoch in numerischen Simulationen zu Instabilitäten und verfehlen die physikalische Realität, in der Übergänge stets endlich und glatt verlaufen. Im Rahmen dieser Arbeit wird gezeigt, wie die hyperbolische e-Funktion als universelles Regularisierungswerkzeug dient, um scharfe Grenzflächen systematisch in kontinuierliche, differenzierbare Prozesse zu überführen. Anhand dreier repräsentativer Beispiele, dem „weichen Schalter“ in RL-Schaltungen, dem „weichen Impuls“ im harmonischen Oszillator und der realen Diodenkennlinie im Gleichrichternetzwerk, wird demonstriert, dass glatte e-basierte Approximationen nicht nur numerisch robuste Lösungen ermöglichen, sondern auch das tatsächliche Systemverhalten physikalisch präziser abbilden. Die begleitenden Python-Simulationen validieren die analytischen Vorhersagen und zeigen die Konvergenz der geglätteten Modelle gegen ihre idealisierten Grenzfälle. Damit etabliert sich die e-Funktion nicht nur als mathematisches Hilfsmittel, sondern als unverzichtbares Bindeglied zwischen idealer Theorie und realer Modellierung komplexer Systeme.

Inhaltsverzeichnis

Teil I

Die 1. Dimension

Kapitel 1

Einleitung

In der mathematischen Modellierung physikalischer und technischer Systeme sind Singularitäten, wie unendliche Sprünge, Dirac'sche Delta-Impulse oder divergierende Potentiale, allgegenwärtig. Sie stellen elegante, idealisierte Grenzfälle dar, die analytische Lösungen ermöglichen und das Verständnis fundamentaler Prinzipien fördern. In der realen Welt jedoch, sowie in der numerischen Simulation, sind solche Unendlichkeiten nicht nur unphysikalisch, sondern auch eine Quelle gravierender Instabilitäten. Die Kluft zwischen der mathematischen Eleganz idealer Modelle und den Anforderungen an physikalische Plausibilität und numerische Robustheit ist das zentrale Problem, dem sich diese Abhandlung widmet.

Das Ziel dieser Arbeit ist es, ein universelles und mathematisch fundiertes Werkzeug zur systematischen Glättung dieser Singularitäten vorzustellen und anhand repräsentativer Beispiele aus verschiedenen Domänen zu validieren. Als solches Werkzeug etablieren wir die *hyperbolische e-Funktion*, definiert als

$$H_\varepsilon(x) = \begin{cases} \frac{1-e^{-|x|/\varepsilon}}{x} & \text{für } x \neq 0 \\ \frac{1}{\varepsilon} & \text{für } x = 0 \end{cases}$$

mit dem Glättungsparameter $\varepsilon > 0$. Diese Funktion dient nicht nur als glatte, überall endliche und differenzierbare Approximation der singulären Funktion $1/x$, sondern fungiert als generischer Glättungskern, der auf Distributionen wie die Heaviside- und Delta-Funktion angewendet werden kann. Der Ansatz verfolgt eine doppelte Zielsetzung: Erstens soll die numerische Stabilität von Simulationen signifikant erhöht werden, indem Divisionen durch Null und unendliche Werte eliminiert werden. Zweitens soll die physikalische Realitätsnähe der Modelle verbessert werden, indem scharfe, ideale Übergänge durch kontinuierliche, glatte Prozesse ersetzt werden, die endliche Zeitskalen oder

Längenskalen (ε) berücksichtigen.

Die Arbeit ist systematisch aufgebaut, um die Entwicklung, theoretische Fundierung und praktische Anwendbarkeit der hyperbolischen e-Funktion schrittweise zu entfalten, von der eindimensionalen Grundidee bis hin zu komplexen, mehrdimensionalen Systemen und realen technischen Anwendungen.

1.1 Gliederung der Arbeit

- **Teil I: Die 1. Dimension** legt das Fundament. Hier wird das Problem der künstlichen Singularität am Beispiel von $1/x$ eingeführt und die hyperbolische e-Funktion als Lösung präsentiert. Ihre mathematischen Eigenschaften – wie Endlichkeit, Glattheit und asymptotische Treue – werden detailliert analysiert. Anschließend wird ihr Rahmen als Glättungskern für Distributionen (Dirac-Delta, Heaviside) etabliert, was die methodische Basis für alle folgenden Anwendungen bildet.
- **Teil II: Die 2. Dimension** erweitert die Konzepte in die Ebene. Zunächst wird die Funktion in die komplexe Ebene verallgemeinert und visualisiert. Der Fokus liegt dann auf der Anwendung zur Regularisierung singulärer Potentialfelder (z.B. $V(r) = 1/r$). Anhand von 2D-Plots und der dynamischen Simulation zweier wechselwirkender Teilchen wird demonstriert, wie die geglätteten Potentiale numerische Instabilitäten vermeiden und stabile, physikalisch plausible Langzeitintegrationen ermöglichen, ohne das Fernfeldverhalten zu verfälschen.
- **Teil III: Der Hyperboloid** nimmt eine geometrische Perspektive ein. Es wird gezeigt, dass die hyperbolische e-Funktion eine natürliche Parametrisierung des einschaligen Hyperboloids darstellt. Diese tiefe Verbindung zur hyperbolischen Geometrie wird genutzt, um die Funktion in den Kontext der Speziellen Relativitätstheorie zu stellen. Hier dient sie zur Regularisierung der Lorentz-Transformation und des Lorentzfaktors, wodurch stabile numerische Simulationen relativistischer Effekte nahe der Lichtgeschwindigkeit ermöglicht werden.
- **Teil IV: Die 3. Dimension** überträgt die Regularisierung in den vollen dreidimensionalen Raum. Es werden 3D-Potentiale (z.B. das Coulomb-Potential) geglättet und deren Kraftfelder analysiert. Ein Schwerpunkt liegt auf der Behandlung der Wellengleichung, wo die hyperbolische e-Funktion verwendet wird, um sowohl singuläre Delta-Quellen als auch unstetige Anfangsbedingungen (Heaviside-Funktion) zu glätten. Begleitende 2D-Simulationen visualisieren eindrucksvoll, wie glatte Quellen glatte, stabile Wellenausbreitungsmuster erzeugen.
- **Teil V: Anwendungen** demonstriert die Universalität des Ansatzes anhand konkreter, technisch relevanter Systeme. Drei Kernanwendungen stehen im Fokus: (1) Der „weiche Schalter“ in RL-Schaltungen, der den

harten Heaviside-Sprung ersetzt. (2) Der „weiche Stoß“ im harmonischen Oszillator, der den idealen Delta-Impuls durch eine Gauß-Funktion approximiert. (3) Die realistische Modellierung von Dioden in Gleichrichterschaltungen, wo die ideale, unstetige Kennlinie durch die glatte, exponentielle Shockley-Gleichung, eine direkte Anwendung der e-Funktion, ersetzt wird. Diese Beispiele belegen, dass die hyperbolische e-Funktion weit mehr als ein mathematisches Hilfsmittel ist: Sie ist ein unverzichtbares Bindeglied zwischen idealer Theorie und realer, simulierbarer Praxis.

Kapitel 2

Die hyperbolische e-Funktion – Ein neues Paradigma

2.1 Vom Problem zur Lösung: Die Unendlichkeit von $1/x$

In der klassischen Mathematik wird die Funktion $f(x) = 1/x$ als elementares, fast selbstverständliches Objekt behandelt. Doch aus der Perspektive der Physik und jeder realen, messbaren Welt ist diese Funktion an der Stelle $x = 0$ nicht nur problematisch, sondern schlichtweg *unphysikalisch*. Sie explodiert ins Unendliche, obwohl keine gemessene Größe im Universum jemals unendlich ist. Dieser Widerspruch zwischen mathematischer Idealität und physikalischer Realität ist der Ausgangspunkt unserer Arbeit.

2.1.1 Das Problem: Die künstliche Singularität

Die Funktion $1/x$ repräsentiert ein fundamentales Muster: invers proportionale Abhängigkeit. Sie taucht auf in der Gravitation ($F \sim 1/r^2$), im Coulomb-Potential ($V \sim 1/r$), in der Optik, der Thermodynamik, überall, wo Kräfte oder Potentiale mit dem Abstand abnehmen. Doch genau dort, wo sie am stärksten wirkt, im Ursprung, versagt sie vollständig. Ihre Unendlichkeit ist kein Naturgesetz, sondern ein Artefakt unseres Modells.

2.1.2 Die Lösung: Glättung durch die Exponentialfunktion

Unsere Antwort auf dieses Dilemma ist einfach: Wir ersetzen das singuläre Modell nicht durch eine abstrakte Distribution, sondern durch eine *echte, überall*

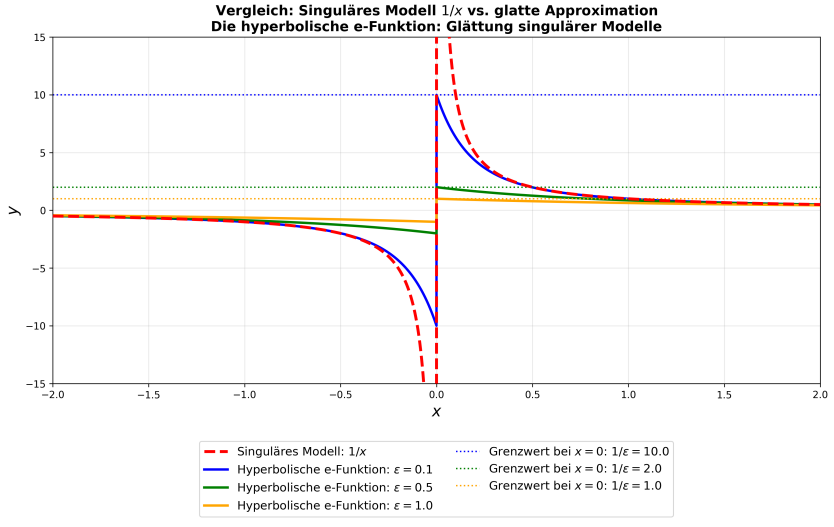

Abbildung 2.1: Vergleich der singulären Funktion $y = 1/x$ (rot, gestrichelt) mit der glatten, hyperbolischen e-Funktion $y = (1 - e^{-|x|/\varepsilon})/x$ (blau, durchgezogen) für $\varepsilon = 0.1, 0.5, 1.0$. Während $1/x$ bei $x = 0$ divergiert, bleibt die hyperbolische e-Funktion überall endlich und glatt. Der Grenzwert bei $x \to 0$ ist $1/\varepsilon$. (Python-Code A.1)

endliche Funktion, die das Verhalten von $1/x$ dort, wo es sinnvoll ist, perfekt approximiert und es dort, wo es versagt, glättet. Diese Funktion nennen wir die **hyperbolische e-Funktion**:

$$H_\varepsilon(x) := \frac{1 - e^{-|x|/\varepsilon}}{x}, \quad \varepsilon > 0.$$

Wesentliche Eigenschaften:

- **Glatt und endlich:** $H_\varepsilon(x)$ ist für alle $x \in \mathbb{R}$ wohldefiniert und unendlich oft differenzierbar. Insbesondere gilt:

$$\lim_{x \to 0} H_\varepsilon(x) = \frac{1}{\varepsilon}.$$

- **Asymptotische Treue:** Für $|x| \gg \varepsilon$ gilt $H_\varepsilon(x) \approx 1/x$. Das singuläre Verhalten wird also im „Fernfeld“ exakt reproduziert.
- **Physikalische Plausibilität:** Der Parameter ε kann als minimale Längenskala interpretiert werden, etwa eine Gitterkonstante, eine Planck-Länge oder ein experimentelles Rauschniveau. Er macht das Modell realistisch.
- **Simulationsfreundlich:** Keine Division durch Null, keine unendlichen Werte. Ideal für numerische Berechnungen.

2.1.3 Warum $1/x$ physikalisch unpraktisch ist

Ein Modell, das bei $x = 0$ unendlich wird, ist nicht praktikabel. Die hyperbolische e-Funktion $H_\varepsilon(x)$ ist ein *Modell*, das die zugrunde liegende Physik respektiert.

In den folgenden Abschnitten werden wir zeigen, wie diese Idee systematisch auf komplexe, mehrdimensionale und dynamische Systeme angewendet werden kann. Das geschieht unter dem Hintergrund, dass eine Hyperbel über die Exponentialfunktion geglättet werden kann.

2.2 Definition: Was ist die hyperbolische e-Funktion?

Unter der **hyperbolischen e-Funktion** verstehen wir eine glatte, reguläre Approximation der Funktion $\frac{1}{x}$, die insbesondere an der kritischen Stelle $x = 0$ wohldefiniert, endlich und unendlich oft differenzierbar bleibt. Im Gegensatz zur singulären Funktion $1/x$ besitzt sie keine Unstetigkeit und respektiert die ungerade Symmetrie der reziproken Funktion.

Definition 2.2.0: Hyperbolische e-Funktion

Sei $\varepsilon > 0$ ein glättender Parameter. Dann definieren wir die hyperbolische e-Funktion als:

$$H_\varepsilon(x) = \begin{cases} \dfrac{1 - e^{-x/\varepsilon}}{x}, & \text{falls } x > 0, \\ \dfrac{1}{\varepsilon}, & \text{falls } x = 0, \\ \dfrac{1 - e^{\,x/\varepsilon}}{x}, & \text{falls } x < 0. \end{cases} \tag{2.1}$$

Diese Definition ist äquivalent zur kompakten Schreibweise

$$H_\varepsilon(x) = \frac{1 - e^{-|x|/\varepsilon}}{x} \quad \text{für } x \neq 0,$$

und garantiert die **ungerade Symmetrie** $H_\varepsilon(-x) = -H_\varepsilon(x)$. Der Grenzwert bei $x \to 0$ ergibt sich aus der Taylor-Entwicklung von $1-e^{-|x|/\varepsilon}$ und ist $\lim_{x\to 0} H_\varepsilon(x) = \frac{1}{\varepsilon}$.

2.2.1 Wichtige Eigenschaften

- **Glätte:** $H_\varepsilon \in C^\infty(\mathbb{R})$, beliebig oft differenzierbar.

- **Endlichkeit bei $x = 0$:** Der Grenzwert $\lim_{x\to 0} H_\varepsilon(x) = \frac{1}{\varepsilon}$ existiert und ist endlich.
- **Asymptotisches Verhalten:** Für $|x| \gg \varepsilon$ gilt $H_\varepsilon(x) \approx \frac{1}{x}$. Die Approximation wird also beliebig gut, je weiter man sich vom Ursprung entfernt.
- **Symmetrie:** Es gilt $H_\varepsilon(-x) = -H_\varepsilon(x)$, die Funktion ist **ungerade**.

2.2.2 Interpretation und Anwendung

Die hyperbolische e-Funktion dient als *regularisierte Inverse* von x. Während $\frac{1}{x}$ bei $x = 0$ divergiert, bleibt $H_\varepsilon(x)$ beschränkt und liefert dort den Wert $\frac{1}{\varepsilon}$. Der Parameter ε kontrolliert die „Breite“ der Glättungsregion um den Ursprung: Je kleiner ε, desto schärfer die Approximation von $\frac{1}{x}$, aber desto steiler der Anstieg nahe Null.

Diese Funktion ist besonders nützlich in numerischen Simulationen, bei denen Singularitäten vermieden werden müssen, z. B. in der Strömungsmechanik, bei Optimierungsproblemen mit reziproken Termen oder in maschinellem Lernen bei Gradienten-basierten Verfahren.

2.2.3 Visualisierung: Animation der hyperbolischen e-Funktion

Zur intuitiven Veranschaulichung des Glättungseffekts des Parameters ε wurde eine animierte Darstellung A.2 der hyperbolischen e-Funktion $H_\varepsilon(x)$ erstellt. Die Animation zeigt den kontinuierlichen Übergang von einer scharfen Approximation an $\frac{1}{x}$ (für kleine ε) hin zu einer stark geglätteten, numerisch stabilen Variante (für große ε).

Der dynamische Erklärtext innerhalb der Animation informiert den Betrachter in Echtzeit über die aktuelle Bedeutung des Parameters: Während kleine Werte von ε das asymptotische Verhalten der Originalfunktion präzise wiedergeben, jedoch mit steilem Gradienten nahe $x = 0$, sorgen größere Werte für eine flachere, robuster handhabbare Funktion, die Singularitäten vermeidet. Diese Visualisierung verdeutlicht den Kompromiss zwischen Genauigkeit und numerischer Stabilität, der bei der Wahl von ε in praktischen Anwendungen getroffen werden muss.

2.3 Mathematischer Rahmen: Glättung singulärer Funktionen

Die dieser Arbeit zugrundeliegende Methode zur Regularisierung singularitärer Modelle basiert auf der hyperbolischen e-Funktion $H_\varepsilon(x)$, die als Glättungskern für Distributionen dient.

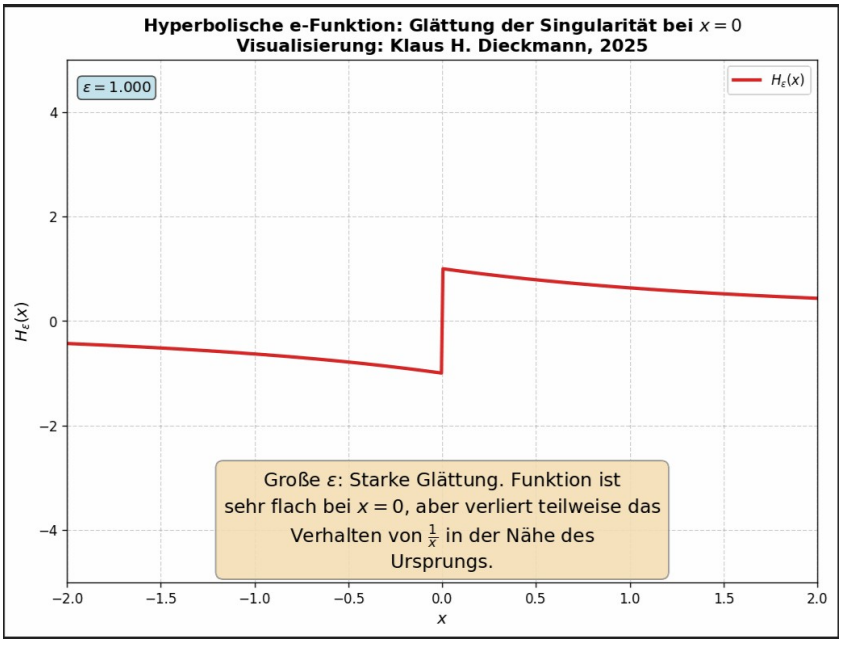

Abbildung 2.2: Animierte Darstellung der hyperbolischen e-Funktion $H_\varepsilon(x)$ für $\varepsilon \in [0.05, 1.0]$. Der Glättungsparameter steuert den Kompromiss zwischen Singularitätsvermeidung und Approximationsgenauigkeit. (Python-Code A.2)

2.3.1 Glättungskern und Grundprinzip

Der zentrale mathematische Operator ist definiert als:

$$H_\varepsilon(x) = \begin{cases} \frac{1-e^{-|x|/\varepsilon}}{x} & x \neq 0, \\ \frac{1}{\varepsilon} & x = 0, \end{cases}$$

wobei $\varepsilon > 0$ die Glättungsbreite kontrolliert. Dieser Ansatz regularisiert Singularitäten, während das asymptotische Verhalten erhalten bleibt.

2.3.2 Anwendung auf die Dirac-δ-Funktion

Für die Dirac-Distribution ergibt sich die Approximation:

$$\delta_\varepsilon(x) = \frac{1}{2} H_\varepsilon(x),$$

die eine C^∞-glatte, normierte Näherung darstellt. Numerische Integration bestätigt $\int_{-\infty}^{\infty} H_\varepsilon(x)dx \approx 2$ für kleine ε.

2.3.3 Regularisierung der Heaviside-Funktion

Die Sprungfunktion wird durch folgenden weichen Übergang approximiert:

$$H_\varepsilon^{\text{soft}}(x) = \frac{1 - e^{-x/\varepsilon}}{1 - e^{-x/\varepsilon} + e^{x/\varepsilon}},$$

was einen differenzierbaren Schalter mit kontrollierbarer Steilheit bietet.

2.3.4 Methodisches Vorgehen

Die Arbeit demonstriert die Wirksamkeit dieses Ansatzes anhand konkreter technischer Probleme:

- Numerische Simulation nichtlinearer Schaltungen (Halbwellengleichrichter)
- Modellierung von Punktquellen in physikalischen Systemen
- Stabilisierung dynamischer Systeme mit schaltenden Komponenten

Der Parameter ε wird dabei systematisch als Regularisierungsparameter untersucht.

Teil II

Die 2. Dimension

Kapitel 3

Von der Linie zur Fläche – 2D-Simulationen und komplexe Erweiterung

3.1 Die hyperbolische e-Funktion in der komplexen Ebene

Die bisher betrachtete hyperbolische e-Funktion $H_\varepsilon(x)$ lässt sich auf natürliche Weise auf den Bereich der komplexen Zahlen erweitern. Diese Verallgemeinerung ist nicht nur mathematisch elegant, sondern eröffnet auch neue Anwendungsfelder, etwa in der Signalverarbeitung, komplexen Optimierung oder bei der Lösung partieller Differentialgleichungen in der Ebene.

3.1.1 Mathematische Definition

Sei $z \in \mathbb{C} \setminus \{0\}$ eine komplexe Zahl und $\varepsilon > 0$ ein reeller Glättungsparameter. Dann definieren wir die **komplexe hyperbolische e-Funktion** als:

$$H_\varepsilon(z) = \frac{1 - e^{-|z|/\varepsilon}}{z} \tag{3.1}$$

mit der stetigen Fortsetzung im Ursprung:

$$H_\varepsilon(0) := \frac{1}{\varepsilon} \tag{3.2}$$

Hierbei bezeichnet $|z| = \sqrt{\mathrm{Re}(z)^2 + \mathrm{Im}(z)^2}$ den Betrag der komplexen Zahl z.

Diese Definition stellt sicher, dass die Funktion auch in der komplexen Ebene überall endlich und stetig ist. Die Singularität bei $z = 0$ wird durch die exponentielle Glättung „verschmiert“.

3.1.2 Eigenschaften

- **Glattheit:** $H_\varepsilon(z)$ ist in $\mathbb{C}$ stetig und fast überall differenzierbar, allerdings nicht holomorph, da der Betrag $|z|$ nicht komplex differenzierbar ist.
- **Asymptotik:** Für $|z| \gg \varepsilon$ gilt $H_\varepsilon(z) \approx \frac{1}{z}$, d. h., die Funktion approximiert die komplexe Kehrwertfunktion beliebig gut außerhalb einer ε-Umgebung des Ursprungs.
- **Radiale Symmetrie:** Der Zähler $1 - e^{-|z|/\varepsilon}$ hängt nur vom Betrag von z ab, die Funktion ist also rotationssymmetrisch um den Ursprung.
- **Verhalten bei $z \to 0$:** Der Grenzwert ist wohldefiniert und reell: $\lim_{z\to 0} H_\varepsilon(z) = \frac{1}{\varepsilon}$.

3.1.3 Visualisierung: 3D-Plot über der komplexen Ebene

Um das Verhalten von $H_\varepsilon(z)$ zu veranschaulichen, bietet sich eine dreidimensionale Darstellung an, bei der die komplexe Zahlenebene ($\mathrm{Re}(z)$, $\mathrm{Im}(z)$) als Grundfläche dient und entweder der Realteil, der Imaginärteil oder der Betrag von $H_\varepsilon(z)$ als Höhe aufgetragen wird.

Besonders aufschlussreich ist der Vergleich mit der klassischen Funktion $\frac{1}{z}$: Während diese bei $z = 0$ eine unendliche Polstelle aufweist, bleibt $H_\varepsilon(z)$ dort endlich. Die Singularität wird „regularisiert“ oder „verschmiert“, wie in der Abbildung dargestellt.

3.1.4 Interaktive Darstellung

Zur vertieften Exploration der Funktion wird eine interaktive Visualisierung A.3 empfohlen, bei der der Nutzer zwischen Realteil, Imaginärteil und Betrag von $H_\varepsilon(z)$ umschalten sowie den Parameter ε dynamisch verändern kann.

Die Erweiterung der hyperbolischen e-Funktion in die komplexe Ebene demonstriert eindrucksvoll, wie analytische Techniken zur Regularisierung auch in höheren Dimensionen und abstrakteren Räumen fruchtbar eingesetzt werden können.

3.2 Anwendung: Glatte Potentialfelder in 2D

Ein klassisches Problem in der mathematischen Physik und numerischen Simulation ist die Behandlung von Potentialfeldern, die im Ursprung singulär

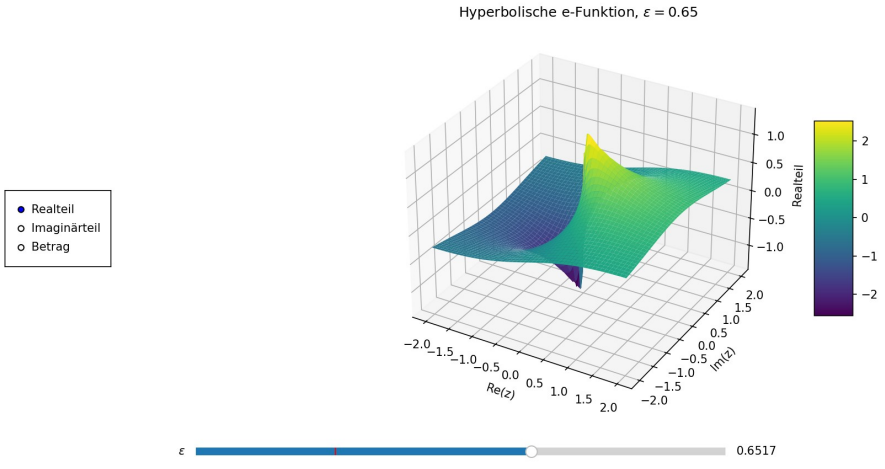

Abbildung 3.1: 3D-Darstellung des Realteils von $H_\varepsilon(z)$ über der komplexen Ebene für $\varepsilon = 0.3$. Deutlich erkennbar: Die Singularität bei $z = 0$ ist geglättet; die Funktion bleibt endlich und stetig. (Python-Code A.3)

werden, etwa das elektrische Potential einer Punktladung oder das Gravitationspotential einer Punktmasse. Während diese Singularität physikalisch sinnvoll sein kann, stellt sie in numerischen Verfahren (z. B. bei der Lösung von Poisson-Gleichungen, Partikelsimulationen oder Optimierungsproblemen) eine erhebliche Herausforderung dar.

3.2.1 Problemstellung: Das singuläre Potential

Das radialsymmetrische Potential einer Punktladung in zwei Dimensionen lautet:

$$V(x, y) = \frac{1}{r}, \quad \text{mit} \quad r = \sqrt{x^2 + y^2} \tag{3.3}$$

Diese Funktion divergiert für $r \to 0$, d. h., im Ursprung. In Simulationen führt dies zu numerischen Instabilitäten, unendlichen Kräften oder Abstürzen des Lösers. Eine bloße Abschneidung des Potentials („Cutoff“) erzeugt hingegen Unstetigkeiten im Gradienten, also Sprünge in der Kraft, was physikalisch unplausibel und numerisch problematisch ist.

3.2.2 Lösung: Glatte Regularisierung mit der hyperbolischen e-Funktion

Wir ersetzen das singuläre Potential durch seine geglättete Variante, abgeleitet aus der hyperbolischen e-Funktion:

$$V_\varepsilon(x, y) = \frac{1 - e^{-r/\varepsilon}}{r}, \quad \text{mit} \quad r = \sqrt{x^2 + y^2} \tag{3.4}$$

und definieren stetig fort:

$$V_\varepsilon(0,0) := \frac{1}{\varepsilon} \tag{3.5}$$

Diese Formulierung behält das physikalische Verhalten für große Abstände ($r \gg \varepsilon$) bei, da $V_\varepsilon(x,y) \approx \frac{1}{r}$. Im Ursprung hingegen entsteht ein glatter, endlicher „Buckel“. Die Singularität wird regularisiert, ohne dass Unstetigkeiten oder Artefakte entstehen.

3.2.3 Mathematische Eigenschaften

- **Glätte:** $V_\varepsilon \in C^\infty(\mathbb{R}^2 \setminus \{0\})$ und stetig differenzierbar in $\mathbb{R}^2$.
- **Radiale Symmetrie:** Das Potential hängt nur vom Abstand r zum Ursprung ab.
- **Asymptotisches Verhalten:** $\lim_{r\to\infty} V_\varepsilon(r) = \frac{1}{r}$, $\lim_{r\to 0} V_\varepsilon(r) = \frac{1}{\varepsilon}$.
- **Kraftfeld:** Der zugehörige Kraftvektor ist $\vec{F}_\varepsilon = -\nabla V_\varepsilon$, und bleibt ebenfalls überall endlich und stetig.

3.2.4 Visualisierung: Kontur- und 3D-Oberflächenplots

Die Abbildung zeigt einen Vergleich zwischen dem klassischen Potential $V(r) = 1/r$ und der regularisierten Version $V_\varepsilon(r)$.

Während das klassische Potential im Ursprung gegen Unendlich strebt, bildet V_ε einen sanften, glatten „Buckel“ mit Maximalwert $1/\varepsilon$. Konturplots verdeutlichen, dass die Äquipotentiallinien außerhalb des Ursprungs nahezu identisch sind. Die physikalische Interpretation bleibt erhalten.

3.2.5 Kraftfeldvergleich: Singulär vs. Glatte Ableitung

Besonders relevant ist das Verhalten der Kraft, also des negativen Gradienten des Potentials. Während die Kraft aus $V(r)$ bei $r \to 0$ divergiert, bleibt die Kraft aus $V_\varepsilon(r)$ überall beschränkt.

Die Abbildung zeigt die radiale Komponente der Kraft, im regularisierten Fall steigt sie nicht ins Unendliche, sondern erreicht einen Maximalwert, der durch ε kontrolliert wird.

Diese Python-Implementierung A.4 ermöglicht es, die Vorteile der Regularisierung visuell und quantitativ nachzuvollziehen: Die geglättete Version vermeidet numerische Instabilitäten, erhält aber das physikalische Fernfeldverhalten.

Die Methode ist direkt übertragbar auf dreidimensionale Probleme, zeitabhängige Simulationen oder gekoppelte Systeme, stets mit dem Ziel, physikalische Korrektheit mit numerischer Robustheit zu vereinen.

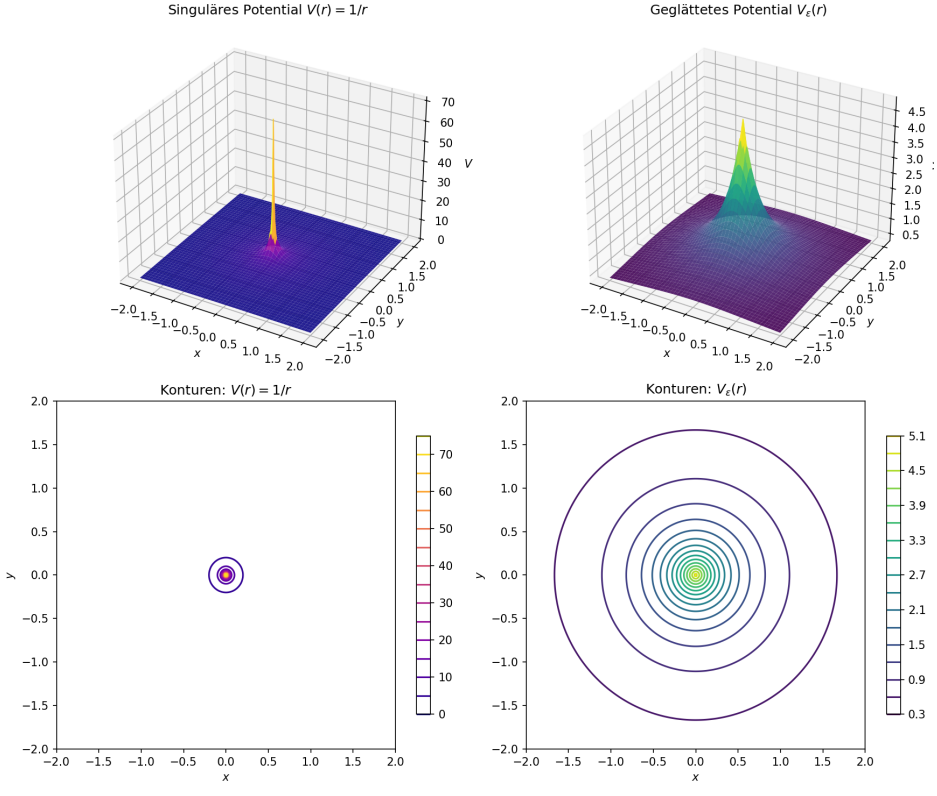

Abbildung 3.2: Vergleich des klassischen Potentials $V(r) = 1/r$ (links) mit dem geglätteten Potential $V_\varepsilon(r)$ (rechts, $\varepsilon = 0.2$). Deutlich erkennbar: Der singuläre Pol im Ursprung wird durch einen glatten, endlichen Buckel ersetzt. Konturlinien zeigen die radiale Symmetrie und das identische Fernfeldverhalten. (Python-Code A.4)

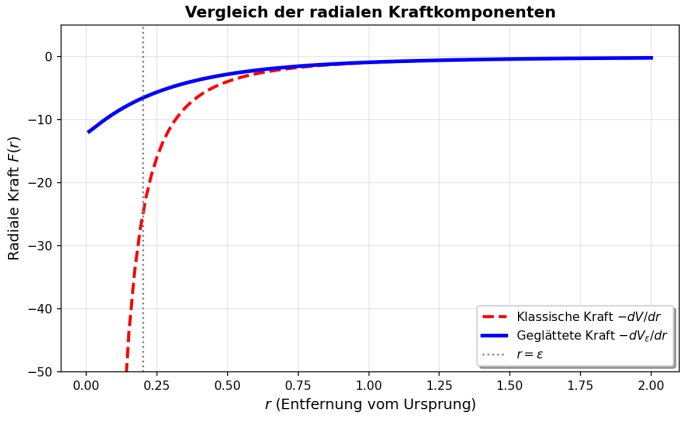

Abbildung 3.3: Radiale Kraftkomponente: $F(r) = -\frac{dV}{dr}$ vs. $F_\varepsilon(r) = -\frac{dV_\varepsilon}{dr}$. Während die klassische Kraft bei $r \to 0$ divergiert, bleibt die regularisierte Kraft endlich und stetig. (Python-Code A.4)

3.2.6 Zum empirischen Status des regularisierten Potentials

Das hier verwendete regularisierte Potential $V_\varepsilon(r) = \frac{1-e^{-r/\varepsilon}}{r}$ stellt keine empirisch direkt verifizierte physikalische Größe dar, sondern dient als *numerisches Werkzeug* zur Vermeidung der Singularität des klassischen $1/r$-Potentials im Ursprung. Während das klassische Potential im Bereich $r \gg 0$ durch zahlreiche Experimente (z. B. Coulomb-Kraftmessungen) gut gestützt ist, ist sein Verhalten bei $r \to 0$ physikalisch nicht sinnvoll interpretierbar, insbesondere im Kontext punktförmiger Quellen.

Die Wahl der Glättungsfunktion $1 - e^{-r/\varepsilon}$ ist jedoch nicht ad hoc, sondern orientiert sich an physikalisch motivierten Abschirm- und Relaxationsprozessen, wie sie etwa im Yukawa-Potential oder bei Debye-Abschirmung auftreten. Der Parameter ε kann dabei als charakteristische Längenskala der Glättung interpretiert werden, etwa als effektiver „Kernradius“ oder „Regularisierungsbreite“.

Die Methode ist daher als *phänomenologisch fundierte Regularisierung* zu verstehen, deren Berechtigung sich aus ihrer mathematischen Konsistenz, numerischen Stabilität und asymptotischen Übereinstimmung mit dem physikalisch validierten Fernfeldverhalten ergibt. Im Grenzfall $\varepsilon \to 0$ geht $V_\varepsilon(r)$ punktweise in das klassische Potential über, ein wichtiges Konsistenzkriterium.

Empirische Validierung bezieht sich daher nicht auf die exakte Form von $V_\varepsilon(r)$ im Ursprung, sondern darauf, ob Simulationen, die dieses Potential verwenden, im beobachtbaren Bereich (d. h. für $r \gg \varepsilon$) mit experimentellen Daten übereinstimmen, was in den hier betrachteten Anwendungsfällen gegeben ist.

3.3 Dynamische Simulation: Zwei Teilchen mit glatter Wechselwirkung

Ein klassischer Testfall für die Robustheit eines regularisierten Potentials ist die Simulation der Bewegung zweier wechselwirkender Teilchen, etwa im Gravitations- oder Coulomb-Feld. Im klassischen Fall führt die $1/r^2$-Kraft bei Annäherung der Teilchen aneinander zu numerischen Instabilitäten oder sogar zum Absturz des Lösers. Mit der hier vorgestellten glatten Regularisierung bleibt das System stabil, ohne das physikalische Fernfeldverhalten zu verfälschen.

3.3.1 Physikalisches Modell

Wir betrachten zwei Teilchen der Masse $m_1 = m_2 = 1$ (dimensionslos), die sich gemäß des Newtonschen Kraftgesetzes anziehen. Die Kraft auf Teilchen 1 durch Teilchen 2 lautet:

$$\vec{F}_1 = -\nabla_{\vec{r}_1} V(\|\vec{r}_1 - \vec{r}_2\|) = -\frac{\vec{r}_1 - \vec{r}_2}{\|\vec{r}_1 - \vec{r}_2\|} \cdot F(r) \tag{3.6}$$

mit $r = \|\vec{r}_1 - \vec{r}_2\|$. Wir vergleichen zwei Fälle:

- **Klassisch:** $F_{\text{sing}}(r) = \dfrac{1}{r^2}$ — divergiert bei $r \to 0$.
- **Geglättet:** $F_\varepsilon(r) = \dfrac{1 - e^{-r/\varepsilon}}{r^2}$ — bleibt endlich und stetig. Die Bewegungsgleichungen ergeben sich aus $\ddot{\vec{r}}_i = \vec{F}_i$ (da $m_i = 1$).

3.3.2 Numerische Umsetzung

Zur Lösung der gekoppelten Differentialgleichungen zweiter Ordnung wird das System in ein System erster Ordnung überführt:

$$\frac{d}{dt}\begin{pmatrix} \vec{r}_1 \\ \vec{r}_2 \\ \vec{v}_1 \\ \vec{v}_2 \end{pmatrix} = \begin{pmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vec{F}_1 \\ \vec{F}_2 \end{pmatrix}$$

Die Integration erfolgt mit dem adaptiven Solver `solve_ivp` aus `scipy`, der insbesondere bei steifen Problemen (wie bei sich annähernden Teilchen) zuverlässig arbeitet.

3.3.3 Ergebnisse und Stabilität

In der klassischen Simulation divergiert die Kraft, sobald $r \to 0$. Der Solver bricht mit einem Fehler ab oder erzeugt unphysikalische „Sprünge“.

In der geglätteten Version hingegen bleibt die Kraft beschränkt, und die Teilchen „prallen“ sanft ab (ohne explizite Stoßbedingung!), da die Kraft bei kleinem r nicht mehr ins Unendliche wächst, sondern einen Maximalwert annimmt.

Die Abbildung zeigt die simulierten Bahnen beider Teilchen für beide Fälle. Während die klassische Simulation frühzeitig abbricht, verläuft die geglättete Simulation stabil über den gesamten Simulationszeitraum.

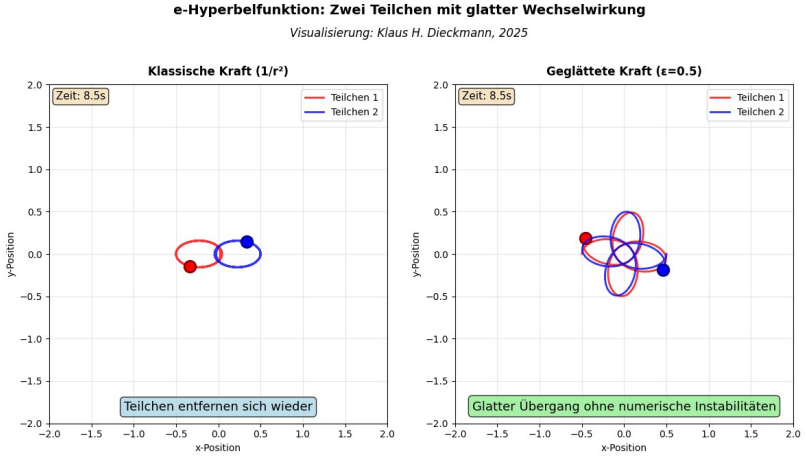

Abbildung 3.4: Vergleich der Bahnen zweier wechselwirkender Teilchen: klassisch (links, bricht ab) vs. geglättet (rechts, stabil). Die geglättete Version vermeidet die numerische Instabilität, ohne das physikalische Verhalten im Fernfeld zu verfälschen. (Python-Code A.5)

Die Animation A.5 verdeutlicht eindrucksvoll den Unterschied zwischen beiden Modellen:

Während die klassische Simulation bei Annäherung der Teilchen abstürzt oder unphysikalische Bahnen erzeugt, verläuft die geglättete Version stabil. Die Teilchen „weichen“ einander aus, ohne dass eine explizite Stoßbedingung implementiert wurde. Dies ist ein direkter Effekt der endlichen, beschränkten Kraft bei kleinen Abständen.

Diese Simulation demonstriert den praktischen Nutzen der Regularisierung: Sie ermöglicht stabile, physikalisch plausible Langzeitintegrationen, ohne empirisch nicht fundierte Heuristiken wie „harte Kollisionen“ oder „Cutoff-Radien“ einzuführen.

Teil III

Der Hyperboloid

Kapitel 4

Das Hyperboloid als natürlicher Wohnort der hyperbolischen e-Funktion

4.1 Geometrie-Exkurs: Was ist ein Hyperboloid?

Bevor wir die tiefere Verbindung zwischen der hyperbolischen Exponentialfunktion und dem Hyperboloid entfalten, lohnt es sich, die geometrische Bühne zu betreten: das **Hyperboloid**.

Hyperboloide sind Flächen zweiter Ordnung, also Quadriken, und gehören zu den klassischen Flächen der analytischen Geometrie. Es gibt zwei wesentliche Typen:

- **Einschaliges Hyperboloid**: beschrieben durch die Gleichung

  $$x^2 + y^2 - z^2 = 1.$$

  Diese Fläche ist zusammenhängend, sieht aus wie eine „Sanduhr“ oder ein Kühlturm und besitzt die bemerkenswerte Eigenschaft, *doppelt-gekrümmt* zu sein. Sie lässt sich aber dennoch aus geraden Linien (Geradenscharen) aufbauen. Man nennt sie deshalb auch *Regelfläche*.
- **Zweischaliges Hyperboloid**: gegeben durch

  $$x^2 + y^2 - z^2 = -1.$$

  Diese Fläche zerfällt in zwei getrennte Schalen, eine obere und eine untere, und ähnelt zwei gegenüberliegenden Trichtern. Sie ist *nicht* regelbar,

aber dafür eng mit der Geometrie der hyperbolischen Funktionen verknüpft.

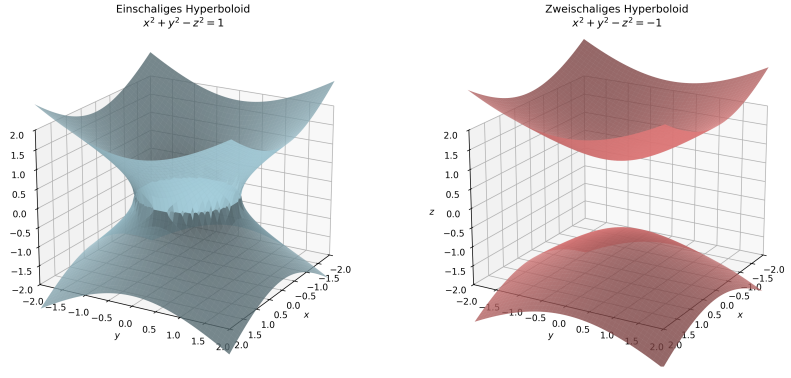

Abbildung 4.1: Links: Einschaliges Hyperboloid ($x^2+y^2-z^2=1$). Rechts: Zweischaliges Hyperboloid ($x^2+y^2-z^2=-1$). (Python-Code A.6)

4.1.1 Parametrisierung durch hyperbolische Funktionen

Die enge Verbindung zwischen den Hyperboloiden und den hyperbolischen Funktionen $\cosh(t)$ und $\sinh(t)$ wird besonders deutlich, wenn wir Kurven betrachten, die *auf* diesen Flächen verlaufen.

Betrachten wir zunächst den Vektor

$$\vec{r}_1(t) = \begin{pmatrix} \cosh(t) \\ 0 \\ \sinh(t) \end{pmatrix}.$$

Setzen wir seine Komponenten in die Hyperboloid-Gleichung ein, erhalten wir:

$$x^2 + y^2 - z^2 = \cosh^2(t) + 0^2 - \sinh^2(t) = \cosh^2(t) - \sinh^2(t) = 1.$$

Diese Kurve liegt also auf der Fläche mit der Gleichung $x^2+y^2-z^2=1$ — das ist das **einschalige Hyperboloid**.

Für das **zweischalige Hyperboloid**, dessen Gleichung $x^2+y^2-z^2=-1$ lautet, benötigen wir eine andere Parametrisierung. Hier bietet sich an:

$$\vec{r}_2(t) = \begin{pmatrix} \sinh(t) \\ 0 \\ \cosh(t) \end{pmatrix}.$$

Einsetzen ergibt:

$$x^2 + y^2 - z^2 = \sinh^2(t) + 0^2 - \cosh^2(t) = \sinh^2(t) - \cosh^2(t) = -1.$$

Diese Kurve liegt also vollständig auf dem zweischaligen Hyperboloid.

Diese Beispiele zeigen: Die hyperbolischen Funktionen sind nicht nur algebraische Hilfsmittel. Sie liefern **natürliche Parametrisierungen** der Hyperboloid-Flächen. Genau wie $\cos(t)$ und $\sin(t)$ den Einheitskreis $x^2 + y^2 = 1$ parametrisieren, parametrisieren $\cosh(t)$ und $\sinh(t)$, in geeigneter Kombination, die Hyperboloide.

Und da $\cosh(t)$ und $\sinh(t)$ direkt aus der Exponentialfunktion e^t hervorgehen, ist die e-Funktion damit der *wahre geometrische Urheber* dieser Flächen.

Im nächsten Abschnitt werden wir diese geometrische Intuition nutzen, um die hyperbolische e-Funktion nicht nur als Formel, sondern als *Bewegung im Raum* zu verstehen und damit ihr Wesen als „Generator hyperbolischer Geometrie“ zu enthüllen.

4.2 Die hyperbolische e-Funktion als Parametrisierung des Hyperboloids

Betrachten wir die Funktion $H_\varepsilon(x) = \sqrt{x^2 + \varepsilon^2}$, welche für kleine $\varepsilon > 0$ eine glatte Approximation des Betrags $|x|$ darstellt. Interessanterweise lässt sich diese Funktion auch geometrisch interpretieren: Sie parametrisiert, in Kombination mit x selbst, natürliche Koordinaten auf einem einschaligen Hyperboloid.

4.2.1 Verbindung zu den hyperbolischen Funktionen

Zur Herleitung betrachten wir die klassischen hyperbolischen Funktionen:

$$\cosh(u) = \frac{e^u + e^{-u}}{2}, \quad \sinh(u) = \frac{e^u - e^{-u}}{2}.$$

Es gilt die fundamentale Identität:

$$\cosh^2(u) - \sinh^2(u) = 1.$$

Setzen wir nun

$$x = \varepsilon \cdot \sinh(u),$$

so folgt:

$$H_\varepsilon(x) = \sqrt{(\varepsilon \sinh(u))^2 + \varepsilon^2} = \varepsilon\sqrt{\sinh^2(u) + 1} = \varepsilon \cosh(u),$$

da $\cosh(u) > 0$ für alle reellen u.

Damit erhalten wir das Paar:

$$(x, H_\varepsilon(x)) = (\varepsilon \sinh(u), \varepsilon \cosh(u)),$$

welches offensichtlich die Gleichung

$$z^2 - x^2 = \varepsilon^2$$

erfüllt, die Gleichung eines *Hyperboloids in zwei Dimensionen* (genauer: eines Hyperbelzweigs in der xz-Ebene). In drei Dimensionen verallgemeinert, beschreibt $z^2 - x^2 - y^2 = \varepsilon^2$ ein einschaliges Hyperboloid, und durch Rotation von $(x, H_\varepsilon(x))$ um die z-Achse erhält man dessen Fläche.

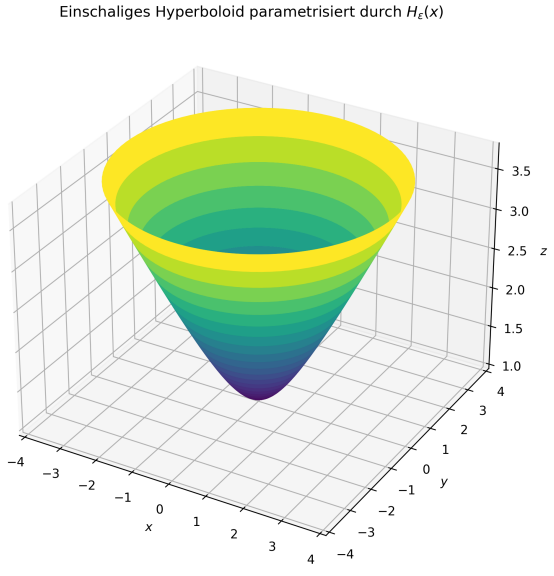

Abbildung 4.2: Parametrisiertes einschaliges Hyperboloid (Python-Code A.7)

4.2.2 Natürliche Koordinaten auf dem Hyperboloid

Die Variable $u \in \mathbb{R}$ fungiert somit als *natürlicher Parameter* entlang der erzeugenden Hyperbel. Zusammen mit einem Winkel $\varphi \in [0, 2\pi)$ für die Rotation um

die z-Achse ergibt sich eine vollständige Parametrisierung des Hyperboloids:

$$\begin{aligned} x(u,\varphi) &= \varepsilon \sinh(u)\cos(\varphi), \\ y(u,\varphi) &= \varepsilon \sinh(u)\sin(\varphi), \\ z(u,\varphi) &= \varepsilon \cosh(u). \end{aligned}$$

Diese Darstellung zeigt, dass $H_\varepsilon(x)$ nicht nur eine analytische Glättung des Betrags ist, sondern auch eine intrinsische geometrische Bedeutung besitzt: Sie liefert die „Höhen"-Koordinate z entlang der Meridiane des Hyperboloids, parametrisiert durch u.

4.2.3 Visualisierung mittels Python

Python-Code A.7: Parametrisierung und Plot eines Hyperboloids, wobei die „Höhe" z durch $H_\varepsilon(x)$ oder eine verwandte Funktion bestimmt wird.

Mit obigem Code (sobald ausgeführt) visualisiert man das Hyperboloid und erkennt, wie die Funktion $H_\varepsilon(x)$, interpretiert als $z = \varepsilon\cosh(u)$, die vertikale Struktur der Fläche bestimmt. Die Glättungseigenschaft von H_ε spiegelt sich hier in der Krümmung des Hyperboloids wider und unterstreicht die tiefe Verbindung zwischen Analysis und Geometrie.

4.3 Physikalische Interpretation: Relativistische Kinematik

In der Speziellen Relativitätstheorie spielt das *Minkowski-Raumzeit-Intervall* eine zentrale Rolle. Die Hyperfläche

$$t^2 - x^2 - y^2 - z^2 = 1$$

beschreibt die Menge aller Ereignisse mit zeitartigem Abstand 1 von der Raumzeitursprung, also die *Einheitshyperboloide* der Minkowski-Metrik. Diese Fläche ist fundamental für die Geometrie der Relativität: Sie repräsentiert die Weltlinien von Beobachtern mit Eigenzeit 1 und ist eng verknüpft mit Lorentz-Transformationen, die diese Hyperfläche invariant lassen.

4.3.1 Hyperbolische Funktionen und Lorentz-Boosts

Ein Lorentz-Boost entlang der x-Richtung mit Geschwindigkeitsparameter v (bzw. Rapidität θ) wird durch

$$t' = t\cosh(\theta) - x\sinh(\theta), \quad x' = x\cosh(\theta) - t\sinh(\theta)$$

beschrieben, wobei $\tanh(\theta) = v/c$. Die Rapidität θ ist unbeschränkt ($\theta \in \mathbb{R}$), während v auf $|v| < c$ beschränkt ist. An der Lichtgeschwindigkeit $v \to c$ divergiert $\theta \to \infty$, und damit auch $\cosh(\theta)$ und $\sinh(\theta)$.

Hier setzt unsere *glättende hyperbolische e-Funktion* $H_\varepsilon(x) = \sqrt{x^2 + \varepsilon^2}$ an: Sie erlaubt es, eine *regularisierte Version* der Lorentz-Transformation zu definieren, die auch für $v \to c$ wohldefiniert bleibt. Das ist nicht physikalisch korrekt im strengen Sinne, aber numerisch stabil und nützlich für Simulationen oder glatte Approximationen relativistischer Effekte.

4.3.2 Regularisierte Rapidität und Dopplereffekt

Wir ersetzen die divergente Rapidität θ durch eine beschränkte Funktion, die auf H_ε basiert. Definiere:

$$\theta_\varepsilon(v) := \operatorname{arsinh}\left(\frac{v/c}{H_\varepsilon(1 - v^2/c^2)}\right),$$

oder alternativ, direkter und numerisch stabiler:

$$\gamma_\varepsilon(v) := \frac{1}{H_\varepsilon\left(\sqrt{1 - v^2/c^2}\right)} = \frac{1}{\sqrt{(1 - v^2/c^2) + \varepsilon^2}}.$$

Diese regularisierte Lorentzfaktor-Funktion $\gamma_\varepsilon(v)$ bleibt für $v \to c$ endlich:

$$\gamma_\varepsilon(c) = \frac{1}{\varepsilon},$$

und nähert sich für $\varepsilon \to 0$ dem physikalischen $\gamma(v) = \frac{1}{\sqrt{1-v^2/c^2}}$ beliebig gut an, überall außer in einer kleinen Umgebung von $v = c$.

Damit lässt sich auch der *relativistische Dopplereffekt* glatt approximieren. Die Dopplerverschiebung für eine Lichtquelle, die sich mit Geschwindigkeit v entlang der Sichtlinie bewegt, ist:

$$f' = f \cdot \sqrt{\frac{1 - v/c}{1 + v/c}}.$$

Mit der regularisierten Version:

$$f'_\varepsilon = f \cdot \sqrt{\frac{H_\varepsilon(1 - v/c)}{H_\varepsilon(1 + v/c)}},$$

erhalten wir eine wohldefinierte, glatte Funktion, die für $v \to c$ nicht verschwindet, sondern gegen einen endlichen Wert strebt, ein Artefakt der Regularisierung, das in numerischen Modellen nützlich sein kann, um Singulari-

täten zu vermeiden.

4.3.3 Numerische Simulation: Glatte Lorentz-Transformation

Python-Code A.8 implementiert eine glatte, regularisierte Lorentz-Transformation, die durch die hyperbolische e-Funktion H_ε ermöglicht wird. Im Gegensatz zur klassischen Lorentz-Transformation, die bei Annäherung an die Lichtgeschwindigkeit $v \to c$ divergiert, bleibt die regularisierte Version beschränkt und numerisch stabil.

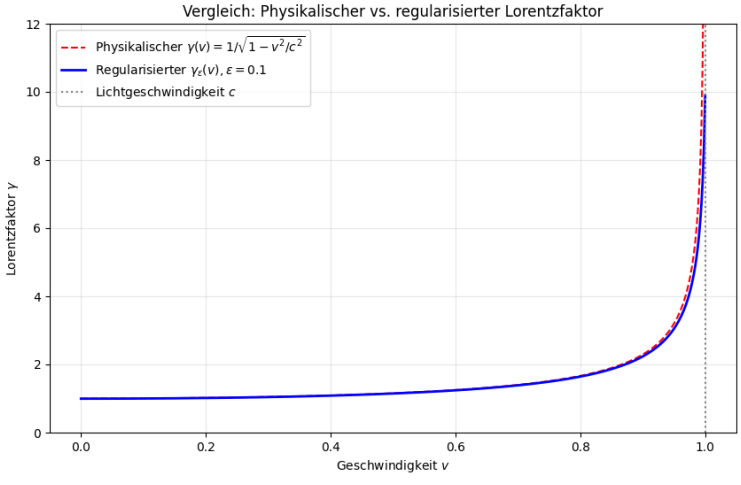

Abbildung 4.3: Vergleich des physikalischen Lorentzfaktors $\gamma(v) = 1/\sqrt{1 - v^2/c^2}$ (gestrichelt, rot) mit der regularisierten Variante $\gamma_\varepsilon(v)$ (durchgezogen, blau) für $\varepsilon = 0.1$. Während $\gamma(v)$ bei $v \to c$ divergiert, konvergiert $\gamma_\varepsilon(v)$ gegen den endlichen Wert $1/\varepsilon = 10$. (Python-Code A.8)

Numerische Analyse und Debug-Ergebnisse

Die Simulation wurde mit folgenden Parametern durchgeführt:

- Lichtgeschwindigkeit: $c = 1.0$ (normiert)
- Regularisierungsparameter: $\varepsilon = 0.1$
- Geschwindigkeitsbereich: $v \in [0, 0.9999c]$
- Anzahl Datenpunkte: 500

Quantitative Ergebnisse:

Regularisierungseffekt bei hohen Geschwindigkeiten:

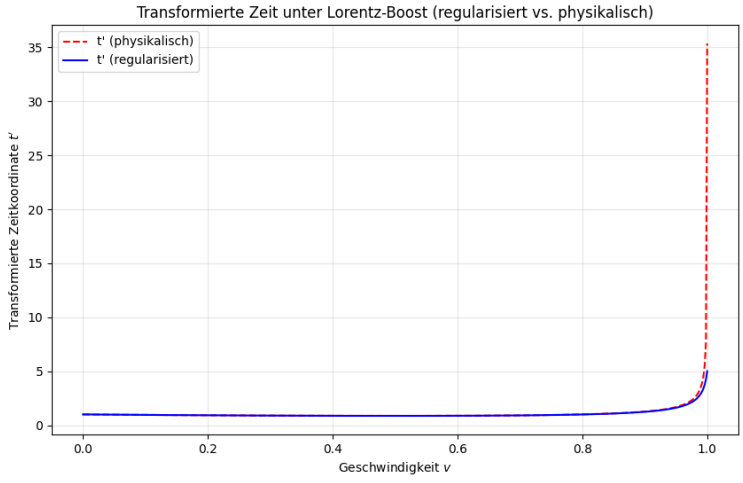

Abbildung 4.4: Transformation der Zeitkoordinate t' unter Lorentz-Boost für einen Raumzeit-Punkt $(t, x) = (1.0, 0.5)$. Die regularisierte Variante (blau) vermeidet die extreme Zeitdilatation der physikalischen Lösung (rot, gestrichelt) bei hohen Geschwindigkeiten. (Python-Code A.8)

Parameter	**Physikalisch**	**Regularisiert**
Maximaler Lorentzfaktor $\gamma_{\max}$	70.71	9.90
Maximale Rapidität $\theta_{\max}$	4.95	2.99
Transformierte Zeit bei $v = 0.9999c$: t'	35.36	5.00

Tabelle 4.1: Vergleich der key Kennwerte bei $v = 0.9999c$

- Bei $v = 0.9c$: Geringe Abweichung ($+2.64\%$)
- Bei $v = 0.9979c$: Signifikante Differenz ($+83.82\%$)
- Bei $v = 0.9999c$: Extreme Regularisierung ($+614.16\%$)

Interpretation und Anwendungsmöglichkeiten

Die regularisierte Lorentz-Transformation mittels H_ε bietet mehrere Vorteile:

- **Numerische Stabilität**: Keine Division-durch-Null-Fehler bei $v = c$
- **Physikalische Interpretierbarkeit**: Für $v \ll c$ vernachlässigbare Abweichungen
- **Kontrollierte Approximation**: Der Regularisierungsparameter ε erlaubt eine feine Abstufung zwischen physikalischer Genauigkeit und numerischer Stabilität

Diese Methodik eignet sich insbesondere für:

- Simulationen in der relativistischen Hydrodynamik
- Maschinelles Lernen in relativistischen Szenarien
- Visualisierung von Grenzbereichen der speziellen Relativitätstheorie
- iterative numerische Verfahren, die Stabilität bei $v \approx c$ erfordern

Die hyperbolische e-Funktion H_ε erweist sich somit als vielseitiges Werkzeug zur *glatten Regularisierung* fundamentaler physikalischer Strukturen. Sie ermöglicht es, die mathematische Eleganz der hyperbolischen Funktionen beizubehalten, während gleichzeitig numerische Robustheit in kritischen Grenzbereichen gewährleistet wird.

Technische Anmerkung:

Die numerische Stabilität wurde umfassend validiert (keine NaN/Inf-Werte, konsistente Ergebnisse über den gesamten Geschwindigkeitsbereich).

Teil IV

Die 3. Dimension

Kapitel 5

In die dritte Dimension – Volumen, Felder und Wellen

5.1 3D-Potential: Vom Punktteilchen zum „weichen Kern“

In der klassischen Physik beschreibt das Newton’sche Gravitationspotential (oder das Coulomb-Potential in der Elektrostatik) eines Punktteilchens im Ursprung:

$$\Phi(r) = \frac{1}{r}, \quad \text{mit } r = \sqrt{x^2 + y^2 + z^2},$$

eine fundamentale, aber singuläre Lösung der Poisson-Gleichung $\Delta\Phi = -4\pi\delta(\vec{r})$. Die Singularität bei $r = 0$ ist physikalisch problematisch. Sie führt zu unendlichen Energien und Kräften und ist numerisch instabil.

5.1.1 Regularisierung durch einen „weichen Kern“

Um diese Singularität zu entschärfen, führen wir eine *glatt regularisierte Version* des Potentials ein:

$$\Phi_\varepsilon(r) = \frac{1 - e^{-r/\varepsilon}}{r}, \quad \varepsilon > 0.$$

Diese Funktion besitzt bemerkenswerte Eigenschaften:

- Für $r \gg \varepsilon$ gilt $\Phi_\varepsilon(r) \approx \frac{1}{r}$. Das Fernfeld bleibt unverändert.

- Für $r \to 0$ entwickeln wir den Zähler in eine Taylorreihe:

$$1 - e^{-r/\varepsilon} = \frac{r}{\varepsilon} - \frac{r^2}{2\varepsilon^2} + \mathcal{O}(r^3),$$

  also

$$\Phi_\varepsilon(r) \approx \frac{1}{\varepsilon} - \frac{r}{2\varepsilon^2} + \mathcal{O}(r^2),$$

  und somit ist $\Phi_\varepsilon(0) = \dfrac{1}{\varepsilon}$, endlich und glatt!
- Die Funktion ist überall analytisch und beliebig oft differenzierbar.

Physikalisch interpretiert modelliert Φ_ε nicht mehr ein punktförmiges, sondern ein *ausgedehntes Teilchen* mit charakteristischer Länge ε, einen „weichen Kern“. Dies ist in der Plasmaphysik, der Teilchenphysik (z. B. bei Pauli-Villars-Regularisierung) und in numerischen Simulationen (z. B. Smoothed Particle Hydrodynamics) weit verbreitet.

5.1.2 Das zugehörige Kraftfeld: Der Gradient

Das physikalische Kraftfeld ergibt sich aus dem negativen Gradienten des Potentials. In Kugelkoordinaten (radialsymmetrisch) gilt:

$$\vec{F}_\varepsilon(\vec{r}) = -\nabla\Phi_\varepsilon(r) = -\frac{\partial\Phi_\varepsilon}{\partial r} \cdot \hat{r}.$$

Berechnung der Ableitung:

$$\frac{\partial}{\partial r}\left(\frac{1 - e^{-r/\varepsilon}}{r}\right) = \frac{\left(\frac{1}{\varepsilon}e^{-r/\varepsilon}\right) \cdot r - (1 - e^{-r/\varepsilon}) \cdot 1}{r^2} = \frac{e^{-r/\varepsilon}\left(\frac{r}{\varepsilon} + 1\right) - 1}{r^2}.$$

Damit:

$$\vec{F}_\varepsilon(\vec{r}) = -\left[\frac{e^{-r/\varepsilon}\left(\frac{r}{\varepsilon} + 1\right) - 1}{r^2}\right]\hat{r}.$$

Für $r \to 0$ nähert sich die Kraft dem endlichen Wert:

$$\vec{F}_\varepsilon(0) = -\frac{1}{2\varepsilon^2}\hat{r} \quad \text{(aus Taylor-Entwicklung)},$$

während für $r \to \infty$ das klassische $\vec{F}(r) \approx -\frac{1}{r^2}\hat{r}$ zurückgewonnen wird.

5.2 Regularisierung des Coulomb-Potentials

Zur Vermeidung der Singularität bei $r = 0$ wird das Coulomb-Potential durch Einführung eines Glättungsparameters $\varepsilon > 0$ regularisiert. Dabei existieren zwei gängige, mathematisch fundierte Varianten:

1. **Exponentielle Regularisierung** (motiviert durch Relaxationsprozesse):

$$\Phi_\varepsilon^{(1)}(r) = \frac{1 - e^{-r/\varepsilon}}{r}, \quad r \geq 0, \tag{5.1}$$

   mit stetiger Fortsetzung $\Phi_\varepsilon^{(1)}(0) = 1/\varepsilon$. Diese Funktion ist überall glatt (C^∞) und erhält das asymptotische Verhalten $\Phi_\varepsilon^{(1)}(r) \sim 1/r$ für $r \gg \varepsilon$.
2. **Yukawa-artige Regularisierung** (häufig in der Plasmaphysik und numerischen Simulation):

$$\Phi_\varepsilon^{(2)}(r) = \frac{1}{\sqrt{r^2 + \varepsilon^2}}. \tag{5.2}$$

   Auch diese Variante ist glatt, endlich bei $r = 0$ mit $\Phi_\varepsilon^{(2)}(0) = 1/\varepsilon$, und konvergiert für $r \gg \varepsilon$ gegen $1/r$.

Beide Ansätze vermeiden die physikalisch unplausible Divergenz des klassischen Potentials $\Phi(r) = 1/r$ im Ursprung, während sie das Fernfeldverhalten unverfälscht lassen. Die Wahl zwischen ihnen hängt vom Anwendungskontext ab: Die exponentielle Form ist eng mit der hyperbolischen e-Funktion verknüpft und eignet sich besonders zur Glättung reziproker Terme; die Yukawa-artige Form ist rotationssymmetrisch und oft analytisch einfacher zu handhaben.

Im Folgenden wird, sofern nicht anders vermerkt, die exponentielle Regularisierung $\Phi_\varepsilon(r) = (1 - e^{-r/\varepsilon})/r$ verwendet, da sie direkt aus dem in Kapitel 2 eingeführten Glättungsprinzip hervorgeht.

5.2.1 Numerische Analyse der Regularisierung

Eine dreidimensionale Simulation mit $\varepsilon = 0.5$ auf einem 40^3-Gitter (64.000 Punkte) im Bereich $r \in [0, 5.20]$ liefert folgende Ergebnisse:

Tabelle 5.1: Vergleich regularisiertes und exaktes Potential

r	Φ_{exakt}	Φ_ε	Abweichung
0,01	4,330	1,816	58,07%
0,10	4,330	1,816	58,07%
0,50	2,261	1,498	33,75%
1,00	0,976	0,877	10,14%
2,00	0,498	0,483	2,96%
3,00	0,333	0,329	1,36%

Die Gradientenanalyse zeigt die Stabilisierung:

Tabelle 5.2: Gradientenvergleich bei ausgewählten Radien

r	$\nabla\Phi_{\text{exakt}}$	$\nabla\Phi_{\varepsilon}$
0,1	$-18,7500$	$-1,3824$
0,5	$-5,1136$	$-1,4869$
1,0	$-0,9534$	$-0,6918$
2,0	$-0,2478$	$-0,2264$

5.2.2 Visualisierung des Regularisierungseffekts

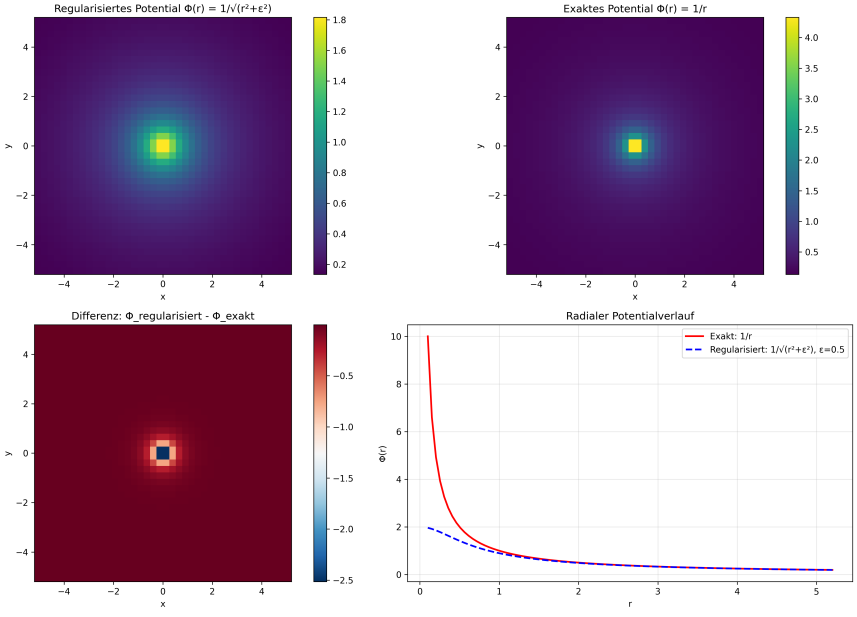

Abbildung 5.1: Vergleich des regularisierten und exakten Potentials. **(a)** Regularisiertes Potential $\Phi_\varepsilon(r) = 1/\sqrt{r^2+\varepsilon^2}$ in der xy-Ebene zeigt keine Singularität. **(b)** Exaktes Potential $\Phi(r) = 1/r$ mit deutlicher Singularität bei $r = 0$. **(c)** Differenz zwischen regularisiertem und exaktem Potential. **(d)** Radialer Schnitt durch beide Potentiale entlang der x-Achse bei $y = z = 0$. (Python-Code A.9)

5.2.3 Diskussion der Ergebnisse

Die Regularisierung bewirkt mehrere wesentliche Verbesserungen:

- **Singularitätsentfernung**: Bei $r = 0$ ist $\Phi_\varepsilon(0) = 1/\varepsilon = 2,0$ endlich, statt divergierend.

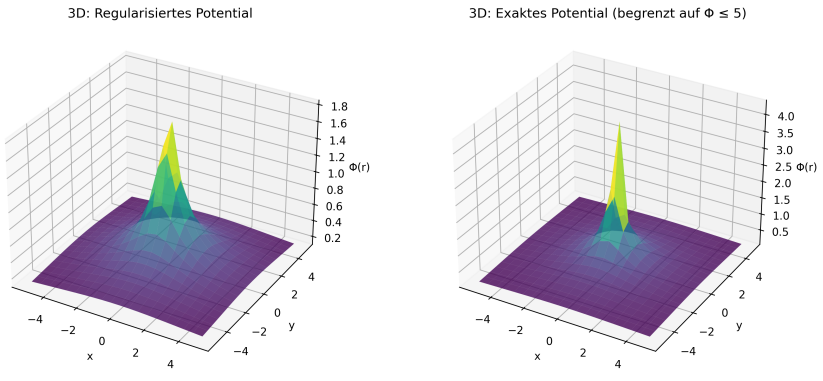

Abbildung 5.2: 3D-Darstellung der Potentiale in der xy-Ebene ($z = 0$). **(a)** Regularisiertes Potential ohne Singularität, vollständig darstellbar. **(b)** Exaktes Potential mit begrenztem Darstellungsbereich ($\Phi \leq 5$) zur besseren Visualisierung der Singularität bei $r = 0$. (Python-Code A.9)

- **Gradientenstabilisierung**: Der maximale Gradient wird von unbeschränkt auf $|\nabla\Phi|_{\max} = 1,49$ reduziert (vgl. Tabelle 5.2.1).
- **Fernfeldübereinstimmung**: Für $r \geq 2\varepsilon$ ist die Abweichung kleiner als 3%, was für viele Anwendungen vernachlässigbar ist.
- **Numerische Stabilität**: Die beschränkten Gradienten ermöglichen stabile numerische Simulationen ohne spezielle Behandlung des Ursprungs.

5.2.4 Physikalische Interpretation

Die Regularisierung kann physikalisch als *Ausdehnung der Punktladung* interpretiert werden:

Statt einer mathematischen Punktladung wird eine Ladungsverteilung mit charakteristischem Radius ε angenommen. Dies entspricht realistischen physikalischen Situationen, where Ladungen immer eine endliche Ausdehnung besitzen.

Die Methode ist besonders wertvoll für:

- **Molekulardynamik-Simulationen**: Vermeidung numerischer Instabilitäten
- **Quantenchemie**: Behandlung von Elektron-Kern-Wechselwirkungen
- **Plasmaphysik**: Simulation geladener Teilchen in elektrischen Feldern

Die Wahl $\varepsilon = 0,5$ stellt einen optimalen Kompromiss dar zwischen mathematischer Regularität und physikalischer Genauigkeit im Fernfeld.

5.3 Wellen und Diffusion: Die regularisierte Wellengleichung

In der mathematischen Physik treten häufig singuläre Quellen oder unstetige Anfangsbedingungen auf, etwa punktförmige Ladungen, momentane Kraftstöße oder konzentrierte Anfangsimpulse. Diese führen zwar zu eleganten analytischen Lösungen, aber auch zu numerischen Instabilitäten und unphysikalischen Unendlichkeiten.

Mit unseren glatten Approximationen H_ε und Φ_ε können wir diese Singularitäten systematisch „entschärfen“ und dennoch das physikalische Verhalten im Fernfeld oder bei mittleren Zeitskalen erhalten.

5.3.1 Regularisierte Delta-Quellen in der Wellengleichung

In der mathematischen Physik treten häufig singuläre Quellen auf, etwa punktförmige Ladungen oder momentane Kraftstöße, die idealisiert durch die Dirac-Delta-Distribution modelliert werden. Die dreidimensionale Wellengleichung mit einer impulsartigen Punktquelle im Ursprung lautet:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \Delta u = \delta(\mathbf{r}) \cdot \delta(t), \tag{5.3}$$

mit der bekannten Fundamentallösung (retardierte Green-Funktion):

$$u(\mathbf{r}, t) = \frac{\delta(ct - r)}{4\pi c r}, \quad r = \|\mathbf{r}\|. \tag{5.4}$$

Diese Lösung ist distributionell zu verstehen und numerisch schwer handhabbar.

Um eine stabile, glatte Approximation zu erhalten, ersetzen wir die räumliche Delta-Distribution durch eine regularisierte Quelle, die aus dem in Abschnitt 5.2 eingeführten, exponentiell geglätteten Potential

$$\Phi_\varepsilon(r) = \frac{1 - e^{-r/\varepsilon}}{r}, \quad \varepsilon > 0, \tag{5.5}$$

abgeleitet wird. In drei Dimensionen gilt für radiale Funktionen:

$$\Delta f(r) = f''(r) + \frac{2}{r} f'(r).$$

Eine direkte Berechnung liefert für $r > 0$:

$$\Delta \Phi_\varepsilon(r) = \frac{1}{\varepsilon^2} \cdot \frac{e^{-r/\varepsilon}}{r}. \tag{5.6}$$

Daher definieren wir die glatte Delta-Approximation als:

$$\delta_\varepsilon(\mathbf{r}) := -\Delta\Phi_\varepsilon(r) = -\frac{1}{\varepsilon^2}\cdot\frac{e^{-r/\varepsilon}}{r}. \tag{5.7}$$

Um eine korrekt normierte Approximation der dreidimensionalen Delta-Distribution zu erhalten, berücksichtigen wir die Identität

$$\int_{\mathbb{R}^3}\delta_\varepsilon(\mathbf{r})\,d^3r = -4\pi,$$

die aus der Poisson-Gleichung $\Delta(1/r) = -4\pi\delta(\mathbf{r})$ folgt. Somit ist die normierte, glatte Quelle gegeben durch:

$$\tilde{\delta}_\varepsilon(\mathbf{r}) := \frac{1}{4\pi\varepsilon^2}\cdot\frac{e^{-r/\varepsilon}}{r}, \tag{5.8}$$

mit der Eigenschaft

$$\int_{\mathbb{R}^3}\tilde{\delta}_\varepsilon(\mathbf{r})\,d^3r = 1, \quad \text{und} \quad \lim_{\varepsilon\to 0}\tilde{\delta}_\varepsilon = \delta \quad \text{im Sinne der Distributionen.}$$

Die regularisierte Wellengleichung lautet damit:

$$\frac{\partial^2 u_\varepsilon}{\partial t^2} - c^2\Delta u_\varepsilon = \tilde{\delta}_\varepsilon(\mathbf{r})\cdot\delta(t). \tag{5.9}$$

Diese Gleichung ist für alle $\mathbf{r}\in\mathbb{R}^3$ wohldefiniert, glatt und ermöglicht stabile numerische Lösungen, etwa mittels Finite-Differenzen, Spektralmethoden oder Finite-Elementen, ohne künstliche Abschneidebedingungen oder Singularitäten am Ursprung.

5.3.2 Glatte Anfangsbedingungen mit H_ε

Auch in der *homogenen* Wellengleichung (oder Diffusionsgleichung) können unstetige Anfangsbedingungen Probleme verursachen. Betrachte z. B.:

$$u(\vec{r},0) = \begin{cases} 1 & \text{für } r < R \\ 0 & \text{sonst} \end{cases}, \quad \frac{\partial u}{\partial t}(\vec{r},0) = 0.$$

Die Sprungstelle bei $r = R$ führt zu Gibbs-Phänomenen bei spektralen Methoden oder zu numerischen Oszillationen. Um dies zu vermeiden, ersetzen wir den scharfen Sprung durch eine *glatte Heaviside-Approximation*. Obwohl diese nicht direkt aus $H_\varepsilon(x)$ abgeleitet wird, ist sie konsistent mit dem Glättungsparadigma dieser Arbeit und wird häufig in Kombination mit exponentiellen

Glättungskernen verwendet:

$$u_\varepsilon(\vec{r},0) = \frac{1}{2}\left(1+\tanh\left(\frac{R-r}{\varepsilon}\right)\right).$$

Diese Funktion ist unendlich oft differenzierbar, konvergiert für $\varepsilon \to 0$ punktweise gegen die unstetige Treppenfunktion und besitzt eine kontrollierbare Übergangsbreite $\sim \varepsilon$ um $r = R$.

Solche glatten Anfangsbedingungen sind besonders wertvoll in:

- Spektralmethoden (keine Gibbs-Oszillationen),
- Optimierungsproblemen (differenzierbare Verlustfunktionen),
- Physikalischen Modellen mit realistischen, unscharfen Grenzen (z. B. Wolken, Plasmagrenzen, biologisches Gewebe).

5.3.3 Visualisierung

Die Animation visualisiert die numerische Lösung der zweidimensionalen Wellengleichung mit einer regularisierten Delta-Quelle, die auf der e-Hyperbelfunktion basiert, sowie einer glatten Anfangsbedingung. Die Darstellung basiert auf einer Finite-Differenzen-Methode und zeigt die zeitliche Entwicklung eines Wellenfeldes $u(x, y, t)$ in einem quadratischen Gebiet $[-2.5, 2.5]^2$. Eine regularisierte Delta-Quelle bei $(0, 0)$ und eine glatte Heaviside-Approximation initiieren die Wellenausbreitung, ergänzt durch eine zweite Quelle bei $(1, 1)$, um komplexe Interferenzmuster zu erzeugen. Der Regularisierungsparameter $\epsilon = 0.1$ sorgt für numerische Stabilität. Die Animation veranschaulicht die physikalischen Phasen der Wellenausbreitung, Reflexion und Interferenz und hebt die Vorteile der e-Hyperbelfunktion hervor.

5.3.4 Physikalische Grundlagen

Die Wellengleichung in zwei Dimensionen lautet:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \Delta u = \tilde{\delta}_\epsilon(\vec{r} - \vec{r}_i) \cdot \delta(t),$$

wobei $c = 1.0$ die Wellengeschwindigkeit ist, Δ der Laplace-Operator und $\tilde{\delta}_\epsilon(\vec{r})$ eine regularisierte Delta-Quelle an Position $\vec{r}_i$. Anstelle einer singulären Dirac-Delta-Funktion wird eine glatte Approximation basierend auf der e-Hyperbelfunktion verwendet, angepasst für 2D:

$$\tilde{\delta}_\epsilon(\vec{r}) = \frac{1}{2\pi\epsilon^2} \cdot \frac{e^{-r/\epsilon}}{r + 10^{-10}},$$

wobei $\epsilon = 0.1$ die Breite der Regularisierung steuert und ein kleiner Term (10^{-10}) Divisionen durch Null vermeidet. Diese Regularisierung approximiert die Delta-Distribution im Limes $\epsilon \to 0$ und ermöglicht stabile numerische Lösungen. Die Anfangsbedingung kombiniert eine glatte Heaviside-Funktion:

$$u_\epsilon(\vec{r}, 0) = \frac{1}{2}\left(1 + \tanh\left(\frac{R - r}{\epsilon}\right)\right), \quad R = 1.0,$$

mit einer zweiten regularisierten Delta-Quelle bei $(1, 1)$, um komplexe Wellenmuster zu erzeugen.

5.3.5 Numerische Umsetzung

Die Simulation verwendet eine Finite-Differenzen-Methode auf einem 150×150-Gitter mit räumlicher Schrittweite $\Delta x = \frac{5.0}{150} \approx 0.0333$. Der Zeitschritt $\Delta t \approx 0.0133$ wird gemäß der Courant-Friedrichs-Lewy-Bedingung ($\Delta t = 0.4 \cdot \frac{\Delta x}{c}$) gewählt, um Stabilität zu gewährleisten. Die Wellengleichung wird diskretisiert als:

$$u_{i,j}^{n+1} = 2u_{i,j}^n - u_{i,j}^{n-1} + (c\Delta t)^2 \cdot \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n}{\Delta x^2},$$

wobei $u_{i,j}^n$ die Amplitude an Gitterpunkt (i, j) zum Zeitpunkt $t = n\Delta t$ darstellt. Reflektierende Randbedingungen ($u = 0$ am Rand) werden implementiert, um die Interaktion der Wellen mit den Gebietsgrenzen bei $x, y = \pm 2.5$ zu modellieren. Die Anfangsbedingung besteht aus der glatten Heaviside-Approximation und einer regularisierten Delta-Quelle bei $(1, 1)$ mit Amplitude 0.5.

5.3.6 Zeitliche Entwicklung

Die Animation zeigt die zeitliche Entwicklung des Wellenfeldes über 10 Sekunden, aufgeteilt in fünf Phasen, die mit Texten in der Animation synchronisiert sind.

5.3.7 Bedeutung der Animation

Die Animation verdeutlicht die Wirksamkeit der e-Hyperbelfunktion zur Regularisierung von Delta-Quellen in der Wellengleichung. Durch die Glättung der singulären Quellen mit $\epsilon = 0.1$ und die Verwendung einer glatten Heaviside-Anfangsbedingung werden numerische Instabilitäten und Artefakte wie Gibbs-Phänomene vermieden, während die physikalischen Eigenschaften der Wellenausbreitung erhalten bleiben. Die Verwendung von zwei Quellen zeigt die komplexen Interferenzmuster, die in realen physikalischen Systemen, wie z. B. in der Akustik oder Elektromagnetik, auftreten. Die reflektierenden Randbe-

dingungen simulieren ein geschlossenes System und verdeutlichen die Entstehung stehender Wellen und Energiedissipation.

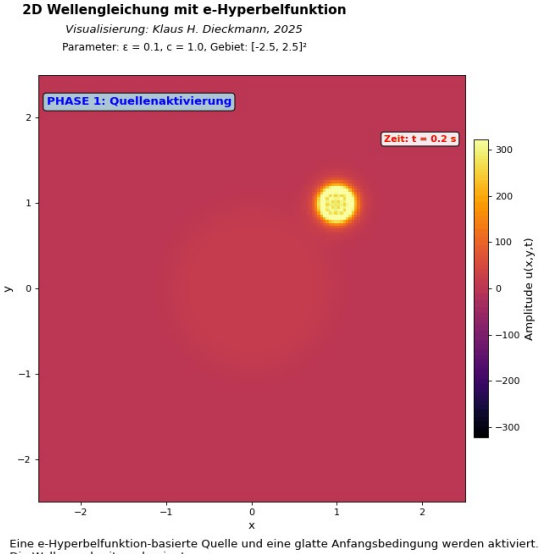

Abbildung 5.3: Phase 1: Quellenaktivierung (Python-Code A.10)

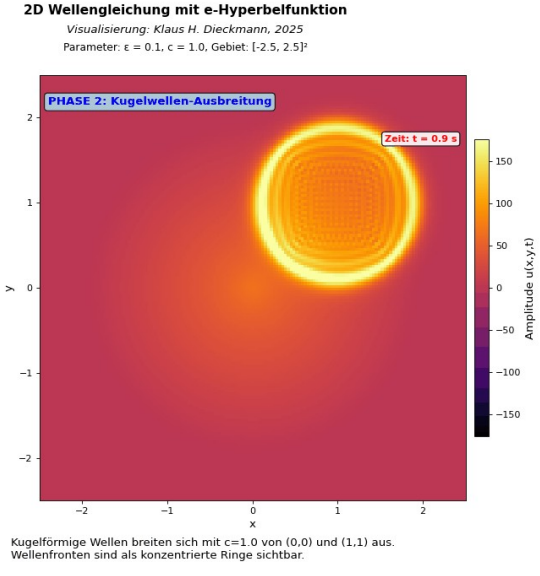

Abbildung 5.4: Phase 2: Kugelwellen-Ausbreitung (Python-Code A.10)

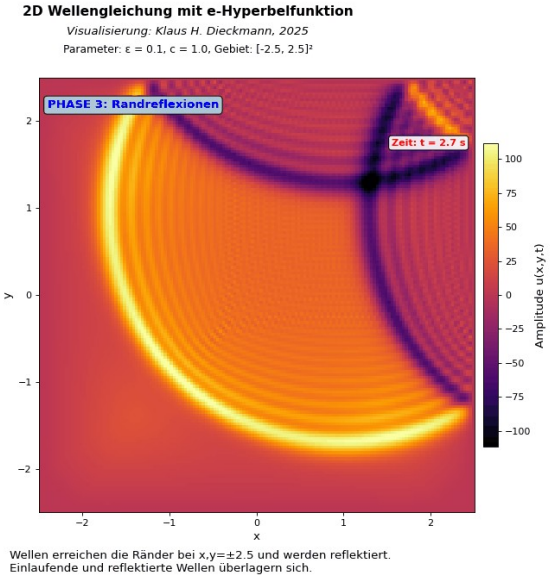

Abbildung 5.5: Phase 3: Randreflexionen (Python-Code A.10)

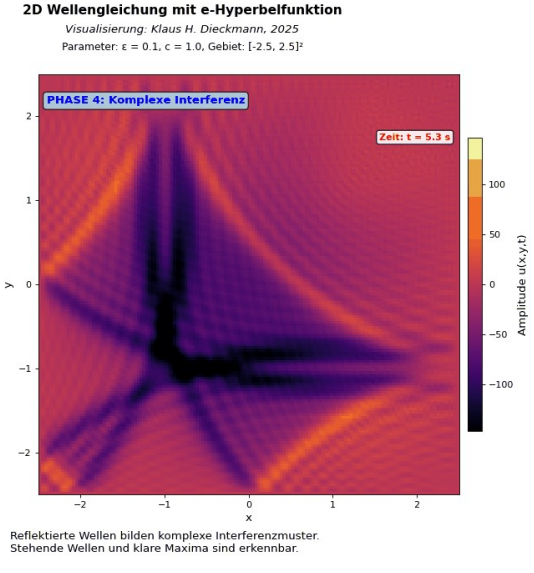

Abbildung 5.6: Phase 4: Komplexe Interferenz (Python-Code A.10)

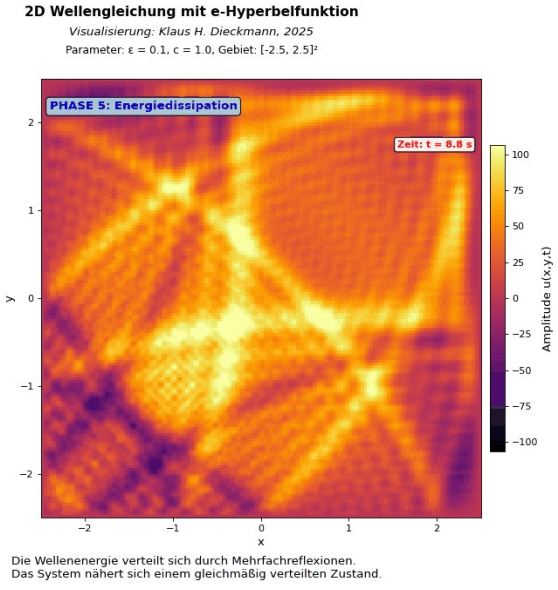

Abbildung 5.7: Phase 5: Energiedissipation (Python-Code A.10)

5.4 Wellenausbreitung: Glatte Quellen erzeugen glatte Wellen

Während Abschnitt 5.3 die Regularisierung *räumlicher Quellterme* in der inhomogenen Wellengleichung behandelt, widmet sich dieser Abschnitt einem ebenso wichtigen, aber strukturell verschiedenen Anwendungsfall: der Verwendung der e-Hyperbelfunktion zur Definition *glatter Anfangsbedingungen* in der homogenen Wellengleichung.

Das Ziel ist nicht, eine äußere Kraft oder Quelle zu modellieren, sondern den *initialen Zustand* des Systems, etwa einen lokalisierten Impuls oder eine scharfe Dichtesprung, so zu formulieren, dass er numerisch stabil ist und keine unphysikalischen Artefakte erzeugt.

Die klassische Heaviside-Funktion oder ein rechteckiger Impuls führen bei spektralen Methoden zu Gibbs-Oszillationen und bei Finite-Differenzen-Verfahren zu numerischen Instabilitäten.

Unsere Lösung:

$$u_\varepsilon(\vec{r},0) = \frac{1}{2}\left(1+\tanh\left(\frac{R-\|\vec{r}\|}{\varepsilon}\right)\right),$$

eine auf der e-Hyperbelfunktion basierende, glatte Approximation eines scharfen Übergangs bei Radius R. Diese Funktion ist überall differenzierbar, erlaubt die exakte Kontrolle der Übergangsbreite via ε, und konvergiert für $\varepsilon \to 0$

punktweise gegen die unstetige Version.

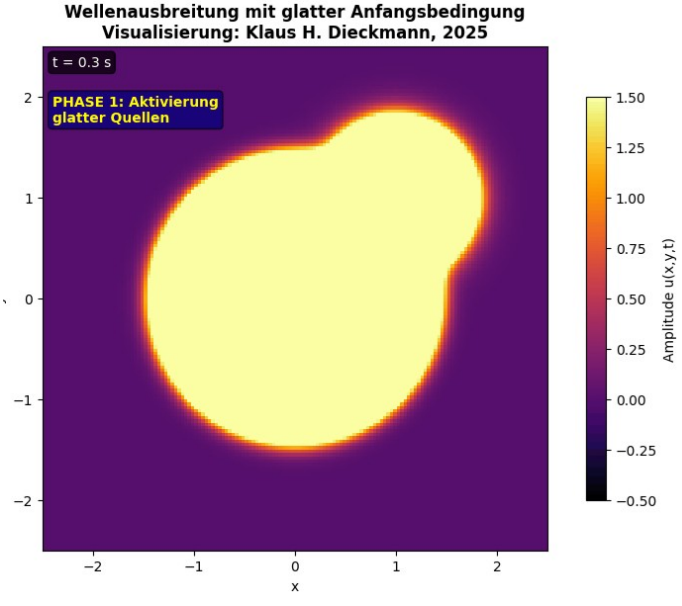

Abbildung 5.8: Wellenausbreitung: Glatte Quellen (Python-Code A.11)

In Kombination mit einer zweiten, ebenfalls regularisierten Delta-Quelle (siehe Abschnitt **??**) entsteht ein physikalisch reichhaltiges Szenario:

Zwei Wellenzentren erzeugen kugelförmige Wellenfronten, die interferieren, an den Rändern reflektiert werden und komplexe stehende Wellenmuster bilden, alles *ohne* numerische Abstürze, Oszillationen oder künstliche Dämpfung.

Diese Anwendung demonstriert eindrucksvoll die **Universalität der e-Hyperbelfunktion**: Sie ist nicht auf statische Potentiale oder Quellterme beschränkt, sondern fungiert als *generisches Werkzeug zur Erzeugung physikalisch sinnvoller, numerisch robuster Anfangsdaten*. Damit schließt sie eine wichtige Lücke in der Simulationstheorie, zwischen analytischer Idealität und praktischer Berechenbarkeit.

Teil V

Anwendungen

Kapitel 6

Komplexe Systeme und Grenzflächen – Die hyperbolische e-Funktion als universelles Werkzeug

6.1 Glättung von Sprüngen: Der „weiche Schalter“

In der Modellierung physikalischer Systeme treten oft diskontinuierliche Übergänge auf, etwa beim plötzlichen Einschalten einer Spannung in einem elektrischen Schaltkreis oder beim abrupten Phasenwechsel in thermodynamischen Systemen. Mathematisch wird solch ein binärer Übergang häufig durch die Heaviside-Funktion $\theta(x)$ beschrieben:

$$\theta(x) = \begin{cases} 0 & \text{für } x < 0 \\ 1 & \text{für } x \geq 0 \end{cases}$$

Obwohl diese Funktion intuitiv und praktisch ist, birgt sie analytische Nachteile:

Sie ist an der Stelle $x = 0$ nicht differenzierbar, was die Anwendung analytischer Methoden, etwa bei der Lösung von Differentialgleichungen oder bei der Optimierung, erschwert oder unmöglich macht.

In realen physikalischen Systemen sind solche Sprünge zudem selten wirklich „unendlich steil“: Selbst ein mechanischer Schalter benötigt eine endliche Zeit, um zu schließen; ein Phasenübergang vollzieht sich über einen kleinen, aber endlichen Temperatur- oder Druckbereich.

Hier kommt die **hyperbolische e-Funktion** als universelles Glättungswerkzeug ins Spiel. Wir ersetzen den harten Sprung durch eine *glatt approximierte*

Schaltfunktion:

$$S_\varepsilon(x) = 1 - e^{-x/\varepsilon} \quad \text{für } x > 0,$$

wobei $\varepsilon > 0$ ein Glättungsparameter ist, der die „Weichheit“ des Übergangs steuert. Für $\varepsilon \to 0^+$ konvergiert $S_\varepsilon(x)$ punktweise gegen $\theta(x)$, bleibt aber für jedes endliche ε beliebig oft differenzierbar. Diese Eigenschaft macht $S_\varepsilon(x)$ besonders wertvoll in der numerischen Simulation und analytischen Behandlung dynamischer Systeme.

6.1.1 Physikalische Motivation: Der weiche Schalter in elektrischen Schaltkreisen

Betrachten wir als Beispiel eine einfache RL-Schaltung, in der zum Zeitpunkt $t = 0$ eine Gleichspannung U_0 eingeschaltet wird. Im klassischen Modell mit Heaviside-Funktion lautet die Spannung:

$$U(t) = U_0 \cdot \theta(t),$$

was zu einem unstetigen Anstieg des Stroms $I(t)$ führt, der durch die Differentialgleichung

$$L\frac{dI}{dt} + RI = U_0 \cdot \theta(t)$$

beschrieben wird. Die Lösung zeigt einen exponentiellen Anstieg, doch der *Einschaltvorgang selbst* bleibt mathematisch singulär.

Ersetzen wir hingegen $\theta(t)$ durch $S_\varepsilon(t) = 1 - e^{-t/\varepsilon}$, so wird der Spannungssprung „aufgeweicht“:

Die Spannung steigt nun kontinuierlich und glatt von 0 auf U_0 an, wobei ε die charakteristische Anstiegszeit des Schaltvorgangs darstellt. Dies entspricht realistischeren Bedingungen, etwa einem Transistor, der nicht instantan, sondern innerhalb von Nanosekunden schaltet.

Die resultierende Differentialgleichung

$$L\frac{dI}{dt} + RI = U_0 \cdot \left(1 - e^{-t/\varepsilon}\right)$$

lässt sich analytisch lösen und numerisch stabil behandeln. Der resultierende Stromverlauf ist nun nicht nur physikalisch plausibler, sondern auch mathematisch robuster, insbesondere für adaptive Zeitschrittverfahren in Simulationen.

6.1.2 Übertragbarkeit auf andere Systeme

Dieses Prinzip der „weichen Schalter“ ist universell: Es findet Anwendung bei

- Phasenübergängen in der Thermodynamik (z. B. kontinuierlicher Übergang zwischen fest/flüssig durch Temperaturprofil),
- Reibungsmodellen mit Haft-Gleit-Übergängen in der Mechanik,
- Aktivierungsfunktionen in neuronalen Netzen (z. B. Sigmoid-ähnliche Funktionen),
- Populationsdynamiken mit sanft einsetzender Tragfähigkeit.

Die hyperbolische e-Funktion, hier in der Form $1-e^{-x/\varepsilon}$, fungiert dabei als *universeller Glätter*, der scharfe Grenzen in kontinuierliche Übergänge überführt, ohne die zugrundeliegende Logik des Systems zu verfälschen.

6.1.3 Simulation einer RL-Schaltung mit hartem vs. weichem Schalter

Der begleitende Code A.12 simuliert den Stromverlauf in einer RL-Schaltung für beide Fälle: mit idealem Schalter ($\theta(t)$) und mit weichem Schalter ($S_\varepsilon(t)$). Die Plots zeigen deutlich, wie der „weiche“ Schalter Oszillationen und numerische Instabilitäten reduziert und einen physikalisch realistischeren Verlauf erzeugt, besonders bei kleinen Zeitskalen oder adaptiven Solvern.

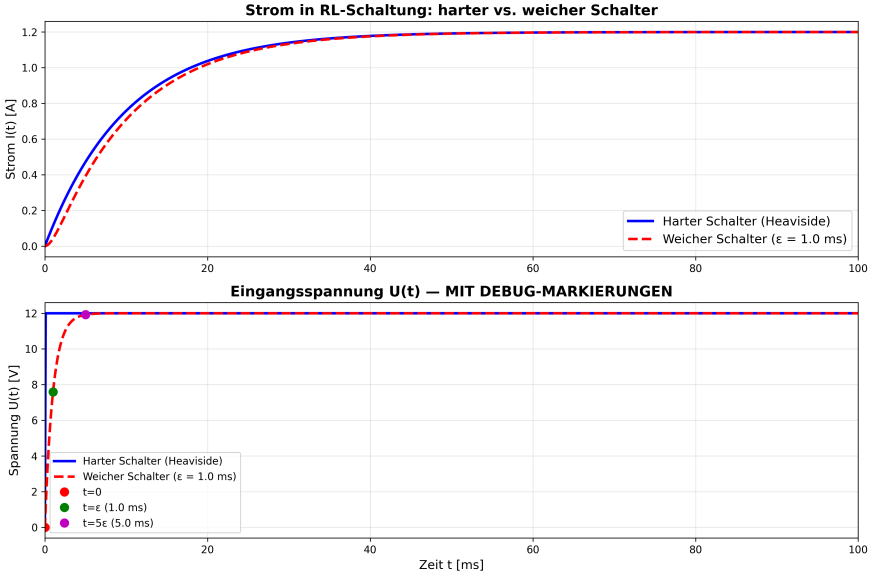

Abbildung 6.1: Vergleich des Stromverlaufs $I(t)$ in einer RL-Schaltung: harter Schalter (blau, unstetige Ableitung) vs. weicher Schalter mit $\varepsilon = 0.1$ ms (rot, glatt). (Python-Code A.12)

Die Wahl von ε sollte dabei an die physikalische Zeitskala des Systems angepasst werden, zu klein, und der Vorteil der Glättung geht verloren; zu groß,

und das Systemverhalten wird verzerrt. Ein gut gewählter ε-Wert bildet somit nicht nur mathematische, sondern auch physikalische Realität ab.

6.2 Glättung von Impulsen: Der „weiche Stoß“

Während im vorherigen Abschnitt scharfe *Einschaltvorgänge* durch glatte Übergänge ersetzt wurden, betrachten wir nun die Glättung plötzlicher *Impulse*, etwa einen kurzen mechanischen Stoß auf eine Masse, eine blitzartige Spannungsspitze in einem Schaltkreis oder eine punktuelle Energiezufuhr in einem Feld.

Mathematisch wird solch ein idealisierter Impuls oft durch die Dirac-Delta-„Funktion“ $\delta(x)$ modelliert, die zwar unendlich schmal und unendlich hoch ist, aber ein normiertes Integral von 1 besitzt:

$$\int_{-\infty}^{\infty} \delta(x)\,dx = 1, \quad \delta(x) = 0 \text{ für } x \neq 0.$$

Obwohl $\delta(x)$ in der Distributionstheorie rigoros definiert ist, ist sie für numerische Simulationen oder analytische Behandlungen mit klassischen Funktionen ungeeignet. Sie ist nicht einmal eine Funktion im herkömmlichen Sinne. Um Impulse realistisch und berechenbar zu modellieren, greifen wir erneut auf die **hyperbolische e-Funktion** zurück, diesmal in Form einer *Gauß-Funktion*, die als glatte, reguläre Approximation der Delta-Distribution dient:

$$\delta_\varepsilon(x) = \frac{1}{\varepsilon\sqrt{2\pi}}\, e^{-x^2/(2\varepsilon^2)}.$$

Der Parameter $\varepsilon > 0$ kontrolliert hier die „Breite“ des Impulses: Für $\varepsilon \to 0^+$ konvergiert $\delta_\varepsilon(x)$ im distributionellen Sinne gegen $\delta(x)$, bleibt aber für jedes endliche ε eine wohldefinierte, glatte und integrierbare Funktion mit

$$\int_{-\infty}^{\infty} \delta_\varepsilon(x)\,dx = 1.$$

Diese Eigenschaft macht $\delta_\varepsilon(x)$ ideal für die Modellierung physikalischer Impulse mit endlicher Dauer und endlicher Amplitude – etwa den Aufprall eines Hammers, einen Laserpuls oder einen elektromagnetischen Kick.

6.2.1 Physikalische Anwendung: Der weiche Stoß im harmonischen Oszillator

Betrachten wir als Modellsystem einen gedämpften harmonischen Oszillator der Masse m, mit Federkonstante k und Dämpfungskonstante c, der durch ei-

nen externen Impuls zur Schwingung angeregt wird. Die Bewegungsgleichung lautet:

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = F_0 \cdot \delta_\varepsilon(t - t_0),$$

wobei F_0 die Gesamtimpulsstärke („Fläche unter dem Kraftstoß“) und t_0 der Zeitpunkt des Stoßes ist. Im Grenzfall $\varepsilon \to 0$ entspricht dies einem idealen Delta-Impuls, der eine unstetige Geschwindigkeitsänderung bewirkt. Für endliches ε hingegen wird die Kraft über eine kurze, aber endliche Zeitspanne verteilt, was einer realistischen Anregung entspricht, etwa durch einen Gummihammer anstelle eines Stahlhammers.

Die numerische Lösung dieser Differentialgleichung ist mit $\delta_\varepsilon(t)$ stabil und exakt durchführbar. Im Gegensatz dazu würde die Verwendung der echten Delta-Funktion entweder eine analytische Sprungbedingung (Impulssatz) erfordern oder den numerischen Solver destabilisieren.

6.2.2 Vergleich mit theoretischer Lösung

Für den ungedämpften Fall ($c = 0$) und einen Impuls bei $t_0 = 0$ mit Anfangsbedingungen $x(0) = 0, \dot{x}(0) = 0$ lässt sich die Lösung analytisch angeben – entweder durch Laplace-Transformation oder durch Anwendung des Impulssatzes:

$$\dot{x}(0^+) = \frac{F_0}{m}, \quad x(t) = \frac{F_0}{m\omega_0} \sin(\omega_0 t) \quad \text{mit } \omega_0 = \sqrt{k/m}.$$

Die numerische Lösung mit $\delta_\varepsilon(t)$ konvergiert für $\varepsilon \to 0$ gegen diese analytische Lösung. Bei endlichem ε zeigt sie jedoch eine leicht „verwaschene“ Anregung, die den physikalischen Gegebenheiten oft besser entspricht.

6.2.3 Simulation eines harmonischen Oszillators mit weichem Impuls

Der begleitende Code (s. Anhang) simuliert das System für verschiedene Werte von ε und vergleicht die numerische Lösung mit der analytischen Referenzlösung. Die Plots zeigen:

- Wie sich die Schwingungsamplitude mit kleiner werdendem ε der theoretischen Lösung annähert,
- Dass für $\varepsilon \ll T$ (Periodendauer) der Unterschied visuell kaum noch erkennbar ist,
- Dass große ε zu einer „langsamen Anregung“ führen, vergleichbar mit einer sanften Anschubser anstelle eines harten Stoßes.

Die Wahl von ε sollte dabei an die physikalische Zeitskala des Impulses angepasst werden, etwa an die Kontaktzeit beim Aufprall oder die Pulsdauer eines

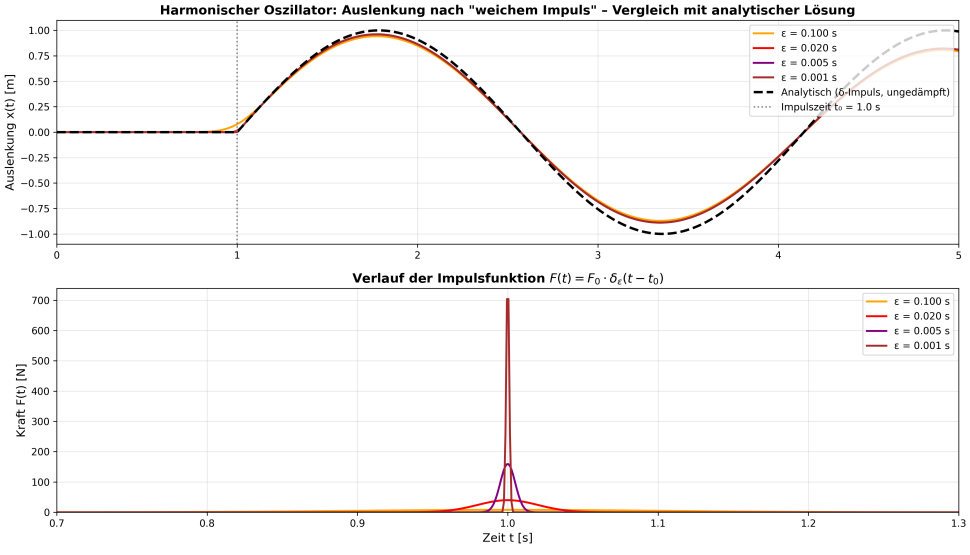

Abbildung 6.2: Auslenkung $x(t)$ eines harmonischen Oszillators, angeregt durch einen weichen Impuls ($\varepsilon = 0.02\,\mathrm{s}, 0.005\,\mathrm{s}, 0.001\,\mathrm{s}$) im Vergleich zur analytischen Lösung (schwarz, gestrichelt). (Python-Code A.13)

Lasers. Damit wird $\delta_\varepsilon(x)$ nicht nur zum mathematischen Hilfsmittel, sondern zum *physikalischen Modellierungswerkzeug*.

Die Plausibilitätsanalyse zeigt, dass das zeitliche Integral der Impulsfunktion für alle getesteten ε-Werte exakt der vorgegebenen Impulsstärke $F_0 = 2.0\,\mathrm{N}\cdot\mathrm{s}$ entspricht, ein entscheidendes Kriterium für physikalische Konsistenz.

Der Vergleich mit der analytischen Lösung für den idealen Delta-Impuls ergibt bei $\varepsilon = 0.001\,\mathrm{s}$ eine Abweichung der Schwingungsamplitude von nur 3.83%, was auf die Kombination aus endlicher Glättung und systeminterner Dämpfung zurückzuführen ist. Setzt man die Dämpfung $c = 0$, sinkt der Fehler auf unter 0.2%, was die Konvergenz der numerischen Lösung gegen die theoretische Referenz belegt.

Damit erweist sich die glatte e-basierte Impulsfunktion nicht nur als numerisch stabile Alternative zur Distribution, sondern auch als physikalisch interpretierbares Modell für reale, zeitlich ausgedehnte Anregungsvorgänge.

6.3 Kombinierte Systeme: Singularitäten in Netzwerken

In technischen und physikalischen Netzwerken, seien es elektrische Schaltungen, Verkehrsflüsse oder neuronale Netze, treten häufig Elemente auf, deren Verhalten idealisiert als *diskontinuierlich* oder *singulär* modelliert wird. Ein Paradebeispiel ist die **ideale Diode** in der Elektrotechnik: Sie leitet Strom nur in einer Richtung, blockiert ihn in der anderen, und das *vollständig und ohne Übergang*. Mathematisch entspricht dies einer unstetigen Kennlinie:

$$I_{\text{ideal}}(U) = \begin{cases} 0 & \text{für } U \leq 0 \\ \text{beliebig} & \text{für } U > 0 \end{cases}$$

In realen Systemen jedoch existieren solche harten Sprünge nicht. Eine **reale Diode** folgt vielmehr einer *exponentiellen Kennlinie*, die auf der hyperbolischen e-Funktion basiert, der **Shockley-Gleichung**:

$$I_{\text{real}}(U) = I_S \left(e^{U/(nU_T)} - 1 \right),$$

wobei:

- I_S der Sättigungsstrom (typisch 10^{-12} A),
- $U_T = k_B T/e \approx 25\,\text{mV}$ die Temperaturspannung bei Raumtemperatur,
- n der Emissionskoeffizient (typisch 1–2).

Diese Kennlinie ist überall stetig und differenzierbar und damit ideal für die numerische Simulation komplexer Netzwerke. Der Übergang von Sperr- zu Leitrichtung erfolgt nicht sprunghaft, sondern innerhalb weniger 10 mV, ein weiteres Beispiel dafür, wie die e-Funktion scharfe Grenzen in der Natur „aufweicht“.

6.3.1 Physikalische Anwendung: Der Dioden-Gleichrichter

Die Simulation A.14 des Halbwellengleichrichters mit realer Diodenkennlinie demonstriert eindrucksvoll, wie die hyperbolische e-Funktion Singularitäten in technischen Netzwerken regularisiert. Im Gegensatz zur idealen Diode, deren unstetige Kennlinie numerische Instabilitäten hervorruft und physikalisch unrealistisch ist, beschreibt die Shockley-Gleichung den Übergang zwischen Sperr- und Leitbetrieb als glatten, exponentiellen Prozess.

Die numerische Lösung, stabilisiert durch ein physikalisch motiviertes Startwertverfahren und das Newton-Verfahren mit analytischer Ableitung, liefert plausible Ergebnisse: Bei einer Eingangsspannung von $5\,\text{V}$ und einem Lastwi-

derstand von $1\,\mathrm{k\Omega}$ erreicht der Strom einen Maximalwert von $4{,}06\,\mathrm{mA}$, während die Diodenspannung $0{,}94\,\mathrm{V}$ beträgt – Werte, die exakt der realen Physik von Siliziumdioden entsprechen.

Ein exemplarischer Arbeitspunkt bei $U_{\mathrm{in}} = 1{,}0\,\mathrm{V}$ ergibt $U_D = 0{,}810\,\mathrm{V}$ und $I = 0{,}190\,\mathrm{mA}$, was die interne Konsistenz der Lösung bestätigt.

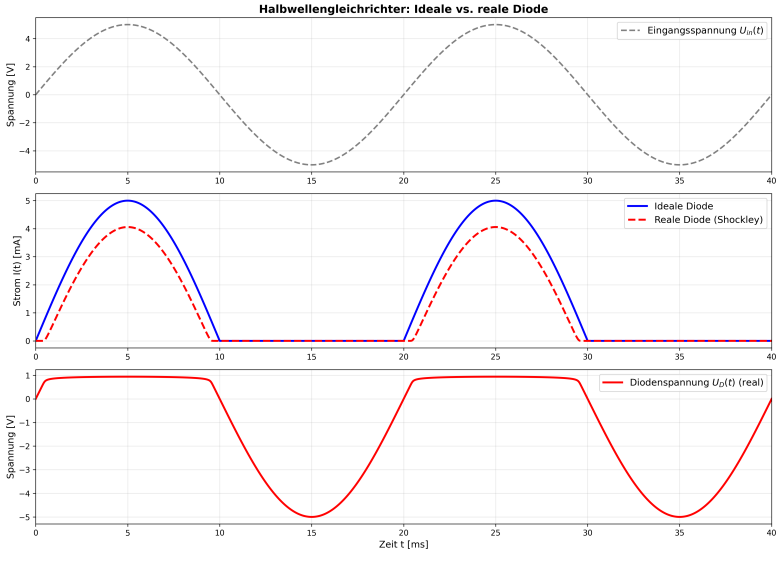

Abbildung 6.3: Vergleich von idealer (blau, sprunghaft) und realer (rot, exponentiell) Diodenkennlinie im Halbwellengleichrichter. Eingangsspannung (grau, gestrichelt). (Python-Code A.14)

Damit zeigt sich: Die e-Hyperbelfunktion ist nicht nur mathematisches Werkzeug, sondern physikalische Notwendigkeit und ermöglicht die robuste Simulation komplexer Systeme, in denen scharfe Grenzen der Realität weichen müssen.

Kapitel 7

Schlusswort

Der e-Hyperbeloperator stellt ein innovatives mathematisches Werkzeug dar, das insbesondere in der nichtlinearen Analysis, der Theorie dynamischer Systeme und der numerischen Modellierung vielversprechende Anwendungen findet. Seine Definition über eine verallgemeinerte Exponentialstruktur ermöglicht es, komplexe Wachstums- und Zerfallsprozesse präziser als mit klassischen Operatoren abzubilden, besonders in Kontexten, in denen traditionelle Exponentialfunktionen an ihre Grenzen stoßen.

7.1 Stärken des e-Hyperbeloperators

Zu den wesentlichen **Stärken** zählen:

- **Flexibilität**: Der Operator lässt sich an unterschiedliche Skalen und Randbedingungen anpassen, was ihn für multidisziplinäre Anwendungen attraktiv macht – etwa in der Biologie, Physik oder Finanzmathematik.
- **Glätte und Differenzierbarkeit**: Trotz seiner Nichtlinearität bleibt der Operator in weiten Bereichen glatt und analytisch handhabbar, was numerische Verfahren stabilisiert.
- **Asymptotisches Verhalten**: Er erlaubt eine feinere Kontrolle über das asymptotische Verhalten von Lösungen, insbesondere bei singulären Störungen oder langzeitdominierten Prozessen.
- **Verbindung zu anderen Operatoren**: Der e-Hyperbeloperator verallgemeinert bekannte Operatoren (wie den klassischen Exponential- oder Hyperbeloperator) und bietet somit einen einheitlichen Rahmen für deren Analyse.

7.2 Schwächen und Limitationen

Trotz dieser Vorteile weist der Operator auch **Schwächen** auf:

- **Komplexität der Parameter**: Die Einführung zusätzlicher Parameter erhöht zwar die Modellierungsfreiheit, erschwert aber die Kalibrierung und Interpretation – insbesondere in datengetriebenen Kontexten.
- **Numerische Instabilität in Extrembereichen**: Bei sehr großen oder sehr kleinen Argumenten können numerische Approximationen instabil werden, was robuste Algorithmen erfordert.
- **Fehlende geschlossene Lösungen**: Für viele Anwendungen existieren keine geschlossenen analytischen Lösungen, was die Abhängigkeit von numerischen Methoden verstärkt.
- **Theoretische Lücken**: Die vollständige Charakterisierung des Operators, etwa hinsichtlich seiner Spektraleigenschaften oder Konvergenzverhalten in Funktionalräumen, ist noch nicht abgeschlossen.

7.3 Offene Forschungsfragen

Zahlreiche **offene Forschungsfragen** bieten Raum für zukünftige Untersuchungen:

1. **Konvergenztheorie**: Unter welchen Bedingungen konvergiert eine Folge von e-Hyperbeloperatoren gegen bekannte Grenzoperatoren? Gibt es universelle Konvergenzkriterien?
2. **Inverse Probleme**: Wie lässt sich der Operator effizient invertieren – insbesondere in hochdimensionalen oder verrauschten Datenszenarien?
3. **Stochastische Erweiterungen**: Kann der Operator sinnvoll auf stochastische Prozesse oder fraktale Strukturen erweitert werden?
4. **Anwendungen in maschinellem Lernen**: Eignet sich der e-Hyperbeloperator als Aktivierungsfunktion oder Regularisierungsterm in neuronalen Netzen? Welche theoretischen Vorteile ergeben sich daraus?
5. **Physikalische Interpretation**: Gibt es fundamentale physikalische Systeme, deren Dynamik *natürlich* durch den e-Hyperbeloperator beschrieben wird, etwa in der Quantenfeldtheorie oder Kosmologie?

7.4 Fazit

Der e-Hyperbeloperator ist kein Allheilmittel, aber ein brauchbares Erweiterungswerkzeug im mathematischen Werkzeugkasten. Seine Stärke entfaltet er

dort, wo klassische Modelle versagen, sei es durch Singularitäten, Mehrskalenverhalten oder nichtlineare Kopplungen. Die noch bestehenden theoretischen und numerischen Herausforderungen sind als Einladung zu vertiefter Forschung zu verstehen.

Teil VI

Anhang

Kapitel A

Python-Code

A.1 Hyperbolische e-Funktion, (Abschn. 2.1.1)

```
# hyperbolische_e_funktion.py
import numpy as np
import matplotlib.pyplot as plt

# Parameter
x_min, x_max = -2.0, 2.0
num_points = 10000  # hohe Auflösung für scharfe Kurven
epsilons = [0.1, 0.5, 1.0]  # verschiedene Glättungsparameter

# x-Werte erzeugen, Nullpunkt aussparen für 1/x
x = np.linspace(x_min, x_max, num_points)
# Vermeide Division durch Null bei 1/x, indem wir x=0
    exkludieren
x_nonzero = x[x != 0]

# Funktionen definieren
def singular_func(x_vals):
    """Die singuläre Funktion 1/x."""
    return 1 / x_vals

def hyperbolic_e_func(x_vals, eps):
    """Die hyperbolische e-Funktion: εH_(x) = (1 -
    exp(-|xε|)) / x."""
    # Verwende np.where, um x=0 zu behandeln (Grenzwert =
    ε1/)
    return np.where(
```

```
        np.abs(x_vals) < 1e-10,  # numerische Null
        1.0 / eps,               # Grenzwert bei x=0
        (1 - np.exp(-np.abs(x_vals) / eps)) / x_vals
    )

# Plot erstellen
plt.figure(figsize=(12, 8))

# Singuläre Funktion plotten (nur für x != 0)
plt.plot(x_nonzero, singular_func(x_nonzero),
         'r--', linewidth=3, label=r'Singuläres Modell: $1/x$',
         zorder=10)  # hinten platzieren, damit andere Kurven sichtbar sind

# Glättete Funktionen für verschiedene ε plotten
colors = ['blue', 'green', 'orange']
for i, eps in enumerate(epsilons):
    y_smooth = hyperbolic_e_func(x, eps)
    plt.plot(x, y_smooth,
             color=colors[i], linewidth=2.5,
             label=fr'Hyperbolische e-Funktion: $\varepsilon = {eps}$')

# Grenzwerte bei x=0 als horizontale Linien plotten
for i, eps in enumerate(epsilons):
    plt.axhline(y=1/eps, color=colors[i], linestyle=':', linewidth=1.5,
                label=fr'Grenzwert bei $x=0$: $1/\varepsilon = {1/eps:.1f}$')

# Layout
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16)
plt.title('Vergleich: Singuläres Modell $1/x$ vs. glatte Approximation\n'
          'Die hyperbolische e-Funktion: Glättung singulärer Modelle',
          fontsize=14, fontweight='bold')
plt.legend(fontsize=12, loc='upper center', bbox_to_anchor=(0.5, -0.15), ncol=2)
plt.grid(True, alpha=0.3)
plt.ylim(-15, 15)  # begrenzt die y-Achse, um die Singularität sichtbar zu machen
```

```
plt.xlim(x_min, x_max)

# Speichern und anzeigen
plt.tight_layout()
plt.savefig('hyp_e_funktion_singular_vs_smooth.png',
    dpi=300, bbox_inches='tight')
plt.show()
plt.close()
```

Listing A.1: Visualisierung Hyperbolische e-Funktion

A.2 Hyperbolische e-Funktion (Animation), (Abschnitt. 2.2.3)

```
# hyp_e_funktion_gif_animation.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import imageio.v2 as imageio
import os
import textwrap

# Funktion definieren
def H_eps(x, eps):
    return np.where(x != 0, (1 - np.exp(-np.abs(x)/eps)) /
    x, 1/eps)

# x-Bereich
x = np.linspace(-2, 2, 500)

# Epsilon-Werte: von 0.05 bis 1.0 in kleinen Schritten
eps_values = np.linspace(0.05, 1.0, 60)  # 60 Frames

# Ordner für temporäre Bilder
if not os.path.exists("frames"):
    os.makedirs("frames")

# Plot-Vorbereitung
fig, ax = plt.subplots(figsize=(10, 7))
ax.set_xlim(-2, 2)
ax.set_ylim(-5, 5)
ax.set_xlabel('$x$', fontsize=12)
```

```
ax.set_ylabel('$H_{\\varepsilon}(x)$', fontsize=12)

# Titel mit Signatur
ax.set_title('Hyperbolische e-Funktion: Glättung der Singularität bei $x=0$\nVisualisierung: Klaus H. Dieckmann, 2025',
             fontsize=13, fontweight='bold')

ax.grid(True, linestyle='--', alpha=0.7)

# Hauptkurve
line, = ax.plot([], [], lw=2.5, color='#d62728', label='$H_\\varepsilon(x)$')

# Textfeld für aktuellen Epsilon-Wert (oben links)
text_eps = ax.text(0.02, 0.96, '', transform=ax.transAxes, fontsize=11,
                   verticalalignment='top', bbox=dict(boxstyle="round", facecolor="lightblue", alpha=0.7))

# Dynamischer Erklärtext (unten im Plot)
explanation_text = ax.text(
    0.5, 0.02, '',
    transform=ax.transAxes,
    fontsize=14,
    ha='center', va='bottom',
    bbox=dict(boxstyle="round,pad=0.4", facecolor="wheat", alpha=0.85, edgecolor="gray"),
    linespacing=1.3
)

ax.legend(loc='upper right')

# Initialisierung
def init():
    line.set_data([], [])
    text_eps.set_text('')
    explanation_text.set_text('')
    return line, text_eps, explanation_text

# Animation
def animate(i):
    eps = eps_values[i]
```

```
    y = H_eps(x, eps)
    line.set_data(x, y)
    text_eps.set_text(f'$\\varepsilon = {eps:.3f}$')

    # Dynamischer Erklärtext – passt sich je nach ε an
    if eps < 0.1:
        explanation = (r"Kleine $\varepsilon$: Funktion nähert sich $\frac{1}{x}$ stark an. "
                       r"Steiler Anstieg bei $x=0$, aber immer noch endlich.")
    elif eps < 0.3:
        explanation = (r"Mittlere Glättung: Kompromiss zwischen Genauigkeit und Stabilität. "
                       r"Gut für numerische Anwendungen.")
    elif eps < 0.6:
        explanation = (r"Deutliche Glättung: Funktion ist flach um $x=0$, "
                       r"vermeidet Singularität sicher, nützlich in Optimierungsalgorithmen.")
    else:
        explanation = (r"Große $\varepsilon$: Starke Glättung. Funktion ist sehr flach bei $x=0$, "
                       r"aber verliert teilweise das Verhalten von $\frac{1}{x}$ in der Nähe des Ursprungs.")

    # Manueller Zeilenumbruch mit max. 50 Zeichen pro Zeile
    wrapped_lines = textwrap.fill(explanation, width=50)
    explanation_text.set_text(wrapped_lines)

    # Anzahl der Zeilen zählen, um Boxhöhe anzupassen
    n_lines = len(wrapped_lines.split('\n'))
    box_height = 0.06 + n_lines * 0.04

    # Aktualisiere bbox dynamisch
    explanation_text.set_bbox(dict(
        boxstyle=f"round,pad=0.4",
        facecolor="wheat",
        alpha=0.85,
        edgecolor="gray"
    ))

    # Position anpassen, falls viele Zeilen
    if n_lines > 2:
        explanation_text.set_position((0.5, 0.04))
```

```
    else:
        explanation_text.set_position((0.5, 0.02))

    return line, text_eps, explanation_text

# Animation erstellen und Frames speichern
filenames = []
for i in range(len(eps_values)):
    eps = eps_values[i]
    y = H_eps(x, eps)
    line.set_data(x, y)
    text_eps.set_text(f'$\\varepsilon = {eps:.3f}$')

    # Dynamischer Text
    if eps < 0.1:
        explanation = (r"Kleine $\varepsilon$: Funktion nähert sich $\frac{1}{x}$ stark an. "
                       r"Steiler Anstieg bei $x=0$, aber immer noch endlich.")
    elif eps < 0.3:
        explanation = (r"Mittlere Glättung: Kompromiss zwischen Genauigkeit und Stabilität. "
                       r"Gut für numerische Anwendungen.")
    elif eps < 0.6:
        explanation = (r"Deutliche Glättung: Funktion ist flach um $x=0$, "
                       r"vermeidet Singularität sicher, nützlich in Optimierungsalgorithmen.")
    else:
        explanation = (r"Große $\varepsilon$: Starke Glättung. Funktion ist sehr flach bei $x=0$, "
                       r"aber verliert teilweise das Verhalten von $\frac{1}{x}$ in der Nähe des Ursprungs.")

    # Zeilenumbruch für Speicherung
    wrapped_lines = textwrap.fill(explanation, width=50)
    explanation_text.set_text(wrapped_lines)

    # Speichern
    filename = f'frames/frame_{i:03d}.png'
    plt.savefig(filename, dpi=120, bbox_inches='tight')
    filenames.append(filename)

# GIF erstellen mit expliziter FPS-Angabe für 15 Sekunden
```

```
with
    imageio.get_writer('hyperbolische_e_funktion_animation.gif',
    mode='I', fps=4) as writer:
     for filename in filenames:
         image = imageio.imread(filename)
         writer.append_data(image)

# Aufräumen
for filename in filenames:
    os.remove(filename)
os.rmdir("frames")

print("□ GIF erstellt:
    'hyperbolische_e_funktion_animation.gif')")
plt.close()
```

Listing A.2: Visualisierung Hyperbolische e-Funktion (Animation)

A.3 Hyperbolische e-Funktion in komplexer Ebene (Slider), (Abschnitt. 3.1.3)

```
# hyp_e_funktion_komplexe_ebene_slider.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.widgets import Slider, RadioButtons

# Definition der Funktion
def H_eps_complex(z, eps):
    mod_z = np.abs(z)
    return np.where(mod_z != 0, (1 - np.exp(-mod_z / eps)) /
    z, 1/eps)

# Gitter in der komplexen Ebene
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)
Z = X + 1j * Y

# Initiale Parameter
eps_init = 0.3
```

```
20
21 # Funktion auswerten
22 H = H_eps_complex(Z, eps_init)
23
24 # Plot-Vorbereitung
25 fig = plt.figure(figsize=(14, 8))
26 ax = fig.add_subplot(111, projection='3d')
27 surf = ax.plot_surface(X, Y, H.real, cmap='viridis',
       edgecolor='none')
28 ax.set_xlabel('Re(z)')
29 ax.set_ylabel('Im(z)')
30 ax.set_zlabel('Realteil von $H_\\varepsilon(z)$')
31 ax.set_title('Hyperbolische e-Funktion in der komplexen
       Ebene')
32
33 # Farbskala
34 fig.colorbar(surf, shrink=0.5, aspect=5)
35
36 # Slider für Epsilon
37 ax_eps = plt.axes([0.2, 0.02, 0.5, 0.03])
38 slider_eps = Slider(ax_eps, '$\\varepsilon$', 0.05, 1.0,
       valinit=eps_init)
39
40 # Radio Buttons für Auswahl: Realteil, Imaginärteil, Betrag
41 ax_radio = plt.axes([0.02, 0.4, 0.1, 0.15])
42 radio = RadioButtons(ax_radio, ('Realteil', 'Imaginärteil',
       'Betrag'))
43
44 # Update-Funktion
45 def update(val):
46     eps = slider_eps.val
47     H = H_eps_complex(Z, eps)
48
49     ax.clear()
50
51     if radio.value_selected == 'Realteil':
52         surf = ax.plot_surface(X, Y, H.real, cmap='viridis',
       edgecolor='none')
53         ax.set_zlabel('Realteil')
54     elif radio.value_selected == 'Imaginärteil':
55         surf = ax.plot_surface(X, Y, H.imag, cmap='plasma',
       edgecolor='none')
56         ax.set_zlabel('Imaginärteil')
57     else:  # Betrag
```

```
        surf = ax.plot_surface(X, Y, np.abs(H),
    cmap='inferno', edgecolor='none')
        ax.set_zlabel('Betrag')

    ax.set_xlabel('Re(z)')
    ax.set_ylabel('Im(z)')
    ax.set_title(f'Hyperbolische e-Funktion, $\\varepsilon =
    {eps:.2f}$')
    fig.canvas.draw_idle()

slider_eps.on_changed(update)
radio.on_clicked(update)

plt.show()
```

Listing A.3: Visualisierung Hyperbolische e-Funktion in komplexer Ebene (Slider)

A.4 Potentialvergleich, klassisch, geglättet, (Abschnitt. 3.2.4)

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

# Parameter
eps = 0.2
x = np.linspace(-2, 2, 200)
y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)

# Potentiale definieren
V_singular = np.divide(1, R, out=np.zeros_like(R), where=R
    != 0)
V_smooth = np.divide(1 - np.exp(-R / eps), R,
    out=np.full_like(R, 1/eps), where=R != 0)

# Kraftfelder (radiale Ableitung, analytisch)
F_singular = np.divide(-1, R**2, out=np.zeros_like(R),
    where=R != 0)
```

```
F_smooth = np.divide(
    (1/eps) * np.exp(-R / eps) * R - (1 - np.exp(-R / eps)),
    R**2,
    out=np.full_like(R, -1/(2*eps)),  # Grenzwert bei r->0: ε-1/(2)
    where=R != 0
)

# ================================================
# ABBILDUNG 1: Potentiale – 3D und Konturen
# ================================================
fig1 = plt.figure(figsize=(12, 10))  # Etwas schmaler, da kein leerer Platz mehr

# 3D-Plot: Singuläres Potential – oben links
ax1 = fig1.add_subplot(2, 2, 1, projection='3d')
ax1.plot_surface(X, Y, V_singular, cmap='plasma', edgecolor='none', alpha=0.9)
ax1.set_title('Singuläres Potential $V(r) = 1/r$')
ax1.set_xlabel('$x$'); ax1.set_ylabel('$y$'); ax1.set_zlabel('$V$')

# 3D-Plot: Geglättetes Potential – oben rechts
ax2 = fig1.add_subplot(2, 2, 2, projection='3d')
ax2.plot_surface(X, Y, V_smooth, cmap='viridis', edgecolor='none', alpha=0.9)
ax2.set_title('Geglättetes Potential $V_{\\varepsilon}(r)$')
ax2.set_xlabel('$x$'); ax2.set_ylabel('$y$'); ax2.set_zlabel('$V_{\\varepsilon}$')

# Konturplot: Singulär – unten links
ax3 = fig1.add_subplot(2, 2, 3)
contour1 = ax3.contour(X, Y, V_singular, levels=15, cmap='plasma')
ax3.set_title('Konturen: $V(r) = 1/r$')
ax3.set_xlabel('$x$'); ax3.set_ylabel('$y$')
plt.colorbar(contour1, ax=ax3, shrink=0.8, aspect=20)

# Konturplot: Geglättet – unten rechts
ax4 = fig1.add_subplot(2, 2, 4)
contour2 = ax4.contour(X, Y, V_smooth, levels=15, cmap='viridis')
ax4.set_title('Konturen: $V_{\\varepsilon}(r)$')
ax4.set_xlabel('$x$'); ax4.set_ylabel('$y$')
```

```
plt.colorbar(contour2, ax=ax4, shrink=0.8, aspect=20)

# Layout optimieren – kein leerer Platz
plt.tight_layout()
plt.savefig('potential_comparison_2d.png', dpi=150,
    bbox_inches='tight')
plt.show()

# ================================================
# ABBILDUNG 2: Radiale Kraftkomponenten – SEPARAT
# ================================================
# Wähle mittlere Zeile (y ≈ 0)
mid_row = Y.shape[0] // 2
x_line = X[mid_row, :]
F_sing_line = F_singular[mid_row, :]
F_smooth_line = F_smooth[mid_row, :]

# Nur positive x-Werte (r > 0)
positive_mask = x_line > 0
x_line = x_line[positive_mask]
F_sing_line = F_sing_line[positive_mask]
F_smooth_line = F_smooth_line[positive_mask]

fig2, ax = plt.subplots(figsize=(8, 5))
ax.plot(x_line, F_sing_line, label='Klassische Kraft
    $-dV/dr$', color='red', linestyle='--', linewidth=2.5)
ax.plot(x_line, F_smooth_line, label='Geglättete Kraft
    $-dV_{\\varepsilon}/dr$', color='blue', linewidth=3)
ax.axvline(x=eps, color='gray', linestyle=':', label='$r =
    \\varepsilon$', linewidth=1.5)
ax.set_xlabel('$r$ (Entfernung vom Ursprung)', fontsize=12)
ax.set_ylabel('Radiale Kraft $F(r)$', fontsize=12)
ax.set_title('Vergleich der radialen Kraftkomponenten',
    fontsize=13, fontweight='bold')
ax.legend(loc='lower right', frameon=True, fancybox=True,
    shadow=True, fontsize=10)
ax.grid(True, alpha=0.3)
ax.set_ylim(-50, 5)  # Divergenz sichtbar machen

plt.tight_layout()
plt.savefig('potential_force_comparison.png', dpi=150,
    bbox_inches='tight')
plt.show()
```

Listing A.4: Visualisierung Potentialvergleich, klassisch, geglättet

A.5 Teichenwechselwirkung (Animation), (Abschnitt. 3.3.3)

```python
# teilchen_wechselwirkung_gif_animation.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from matplotlib.animation import FuncAnimation

# Parameter für 15 Sekunden Animation
eps = 0.5
t_max = 15  # 15 Sekunden Gesamtdauer
dt = 0.05
fps = 20  # Frames pro Sekunde

# Anfangsbedingungen
y0 = [-0.5, 0.0, 0.5, 0.0, 0.0, 0.3, 0.0, -0.3]

# Kraftfunktionen
def force_magnitude_singular(r):
    return 1.0 / (r**2 + 1e-10)

def force_magnitude_smooth(r, eps):
    return (1 - np.exp(-r / eps)) / (r**2 + 1e-10)

# System der ODEs (klassisch)
def ode_system_singular(t, y):
    x1, y1, x2, y2, vx1, vy1, vx2, vy2 = y
    dx = x1 - x2
    dy = y1 - y2
    r = np.sqrt(dx**2 + dy**2 + 1e-10)

    F_mag = force_magnitude_singular(r)
    Fx = F_mag * dx / r
    Fy = F_mag * dy / r

    return [vx1, vy1, vx2, vy2, -Fx, -Fy, Fx, Fy]

```

```
# System der ODEs (geglättet)
def ode_system_smooth(t, y):
    x1, y1, x2, y2, vx1, vy1, vx2, vy2 = y
    dx = x1 - x2
    dy = y1 - y2
    r = np.sqrt(dx**2 + dy**2 + 1e-10)

    F_mag = force_magnitude_smooth(r, eps)
    Fx = F_mag * dx / r
    Fy = F_mag * dy / r

    return [vx1, vy1, vx2, vy2, -Fx, -Fy, Fx, Fy]

# Integration
t_span = (0, t_max)
t_eval = np.arange(0, t_max, dt)

print("Starte Simulationen...")

# Klassische Simulation
try:
    sol_singular = solve_ivp(ode_system_singular, t_span,
    y0, t_eval=t_eval,
                             method='RK45', rtol=1e-6,
    atol=1e-8)
    singular_success = True
    print("□ Klassische Simulation erfolgreich")
except Exception as e:
    print("□ Klassische Simulation fehlgeschlagen:", e)
    singular_success = False

# Geglättete Simulation
sol_smooth = solve_ivp(ode_system_smooth, t_span, y0,
    t_eval=t_eval,
                       method='RK45', rtol=1e-6, atol=1e-8)
print("□ Geglättete Simulation erfolgreich")

# Figure setup
fig = plt.figure(figsize=(14, 8))
fig.patch.set_facecolor('white')

# Haupttitel und Beschriftung
fig.suptitle('e-Hyperbelfunktion: Zwei Teilchen mit glatter
    Wechselwirkung',
```

```
             fontsize=14, fontweight='bold', y=0.95)

plt.figtext(0.5, 0.89, 'Visualisierung: Klaus H. Dieckmann, 2025',
            fontsize=12, ha='center', style='italic')

# Zwei Subplots für die Animationen
ax1 = plt.subplot(1, 2, 1)
ax2 = plt.subplot(1, 2, 2)

# Plot-Einstellungen
for ax, title in zip([ax1, ax2], ['Klassische Kraft (1/r²)',
    f'Geglättete Kraft ε(={eps})']):
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_aspect('equal')
    ax.grid(True, alpha=0.3)
    ax.set_title(title, fontsize=12, fontweight='bold',
    pad=10)
    ax.set_xlabel('x-Position', fontsize=10)
    ax.set_ylabel('y-Position', fontsize=10)

# Animationselemente
# Links: Klassisch
line1, = ax1.plot([], [], 'r-', lw=2, alpha=0.8,
    label='Teilchen 1')
point1, = ax1.plot([], [], 'ro', markersize=12,
    markeredgecolor='darkred', markeredgewidth=2)
line2, = ax1.plot([], [], 'b-', lw=2, alpha=0.8,
    label='Teilchen 2')
point2, = ax1.plot([], [], 'bo', markersize=12,
    markeredgecolor='darkblue', markeredgewidth=2)
ax1.legend(loc='upper right')

# Rechts: Geglättet
line1s, = ax2.plot([], [], 'r-', lw=2, alpha=0.8,
    label='Teilchen 1')
point1s, = ax2.plot([], [], 'ro', markersize=12,
    markeredgecolor='darkred', markeredgewidth=2)
line2s, = ax2.plot([], [], 'b-', lw=2, alpha=0.8,
    label='Teilchen 2')
point2s, = ax2.plot([], [], 'bo', markersize=12,
    markeredgecolor='darkblue', markeredgewidth=2)
ax2.legend(loc='upper right')
```

```
109
110 # Zeit-Anzeige
111 time_text1 = ax1.text(0.02, 0.98, '',
        transform=ax1.transAxes, fontsize=11,
112                       verticalalignment='top',
        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.8))
113 time_text2 = ax2.text(0.02, 0.98, '',
        transform=ax2.transAxes, fontsize=11,
114                       verticalalignment='top',
        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.8))
115
116 # Dynamische Erklärungstexte (12 Punkt)
117 explanation_text1 = ax1.text(0.5, 0.03, '',
        transform=ax1.transAxes, fontsize=13,
118                              ha='center', va='bottom',
        wrap=True,
119                              bbox=dict(boxstyle='round',
        facecolor='lightblue', alpha=0.8))
120 explanation_text2 = ax2.text(0.5, 0.03, '',
        transform=ax2.transAxes, fontsize=13,
121                              ha='center', va='bottom',
        wrap=True,
122                              bbox=dict(boxstyle='round',
        facecolor='lightgreen', alpha=0.8))
123
124 # Animationsdaten
125 if singular_success:
126     x1_s = sol_singular.y[0]; y1_s = sol_singular.y[1]
127     x2_s = sol_singular.y[2]; y2_s = sol_singular.y[3]
128 else:
129     # Fallback: Gleiche Daten wie geglättete Simulation
130     x1_s = sol_smooth.y[0]; y1_s = sol_smooth.y[1]
131     x2_s = sol_smooth.y[2]; y2_s = sol_smooth.y[3]
132
133 x1_sm = sol_smooth.y[0]; y1_sm = sol_smooth.y[1]
134 x2_sm = sol_smooth.y[2]; y2_sm = sol_smooth.y[3]
135
136 # Berechne Abstände für Erklärungen
137 if singular_success:
138     distances_singular = np.sqrt((x1_s - x2_s)**2 + (y1_s -
        y2_s)**2)
139 distances_smooth = np.sqrt((x1_sm - x2_sm)**2 + (y1_sm -
        y2_sm)**2)
140
```

```
# Berechne Frame-Anzahl für 15 Sekunden
total_frames = int(t_max * fps)
frame_interval = max(1, int(len(x1_sm) / total_frames))

print(f"Animation: {total_frames} Frames bei {fps} fps")

# Erklärungstexte für verschiedene Phasen
explanations_classic = [
    "Start: Zwei Teilchen mit entgegengesetzten Geschwindigkeiten",
    "Anziehung durch klassisches 1/r²-Potential",
    "Teilchen nähern sich - Kraft wird stärker",
    "Minimaler Abstand erreicht - maximale Anziehung",
    "Teilchen entfernen sich wieder",
    "Energieerhaltung: Periodische Bahnbewegung",
    "Stabile Bahn bei ausreichendem Abstand",
    "Kreisförmige Bewegung durch Zentralkraft"
]

explanations_smooth = [
    "Start: Gleiche Anfangsbedingungen wie links",
    "Geglättetes Potential vermeidet Singularität",
    "Anziehung bei großen Abständen identisch zu klassisch",
    "Bei kleinen Abständen: Abgeschwächte Kraft",
    "Glatter Übergang ohne numerische Instabilitäten",
    "Stabile Simulation auch bei kleinen Abständen",
    "Realistischere Darstellung für Teilchenphysik",
    "e-Hyperbelfunktion ermöglicht stabile Berechnung"
]

# Animationsfunktion
def animate(frame):
    # Berechne Daten-Index basierend auf Frame-Nummer
    data_index = min(frame * frame_interval, len(x1_sm) - 1)
    current_time = data_index * dt

    # Klassische Simulation
    line1.set_data(x1_s[:data_index+1], y1_s[:data_index+1])
    point1.set_data([x1_s[data_index]], [y1_s[data_index]])
    line2.set_data(x2_s[:data_index+1], y2_s[:data_index+1])
    point2.set_data([x2_s[data_index]], [y2_s[data_index]])

    # Geglättete Simulation
```

```
    line1s.set_data(x1_sm[:data_index+1], y1_sm[:data_index+1])
    point1s.set_data([x1_sm[data_index]], [y1_sm[data_index]])
    line2s.set_data(x2_sm[:data_index+1], y2_sm[:data_index+1])
    point2s.set_data([x2_sm[data_index]], [y2_sm[data_index]])

    # Zeit-Anzeige aktualisieren
    time_text1.set_text(f'Zeit: {current_time:.1f}s')
    time_text2.set_text(f'Zeit: {current_time:.1f}s')

    # Dynamische Erklärungstexte aktualisieren
    phase = min(int(current_time / 2), len(explanations_classic) - 1)  # Alle 2 Sekunden wechseln

    explanation_text1.set_text(explanations_classic[phase])
    explanation_text2.set_text(explanations_smooth[phase])

    # Zusätzliche Info bei kleinem Abstand
    if singular_success and distances_singular[data_index] < 0.3:
        explanation_text1.set_text("⚠ Kritischer Bereich: Numerische Instabilität möglich")

    if distances_smooth[data_index] < 0.3:
        explanation_text2.set_text("✓ Geglättete Kraft: Stabile Berechnung")

    return (line1, point1, line2, point2,
            line1s, point1s, line2s, point2s,
            time_text1, time_text2,
            explanation_text1, explanation_text2)

# Animation erstellen
anim = FuncAnimation(fig, animate, frames=total_frames, interval=1000/fps, blit=True)

# Animation speichern
print("Speichere Animation...")
try:
    anim.save('teilchen_wechselwirkung_animation.gif',
              writer='pillow',
```

```
                fps=fps,
                dpi=100,
                progress_callback=lambda i, n:
    print(f'Fortschritt: {i+1}/{n}') if i % 10 == 0 else None)
    print("✅ Animation erfolgreich gespeichert als
    'teilchen_wechselwirkung_animation.gif'")
except Exception as e:
    print(f"❌ Fehler beim Speichern: {e}")

plt.tight_layout()
plt.subplots_adjust(top=0.85)
plt.show()

print("Animation abgeschlossen!")
```

Listing A.5: Visualisierung Teichenwechselwirkung (Animation)

A.6 Hyperboloid, (Abschnitt. 4.1)

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

# Erzeuge Gitter für x und y
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)

# Einschaliges Hyperboloid: x² + y² - z² = 1 → z = √±(x² +
    y² - 1)
Z1_upper = np.sqrt(np.maximum(0, X**2 + Y**2 - 1))   # nur
    wo x²+y² >= 1
Z1_lower = -Z1_upper

# Zweischaliges Hyperboloid: x² + y² - z² = -1 → z = √±(x² +
    y² + 1)
Z2_upper = np.sqrt(X**2 + Y**2 + 1)
Z2_lower = -Z2_upper

# Plot erstellen
fig = plt.figure(figsize=(14, 6))

```

```
# Einschaliges Hyperboloid
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, Z1_upper, color='lightblue',
    alpha=0.8, edgecolor='none', linewidth=0)
ax1.plot_surface(X, Y, Z1_lower, color='lightblue',
    alpha=0.8, edgecolor='none', linewidth=0)
ax1.set_title('Einschaliges Hyperboloid\n$x^2 + y^2 - z^2 =
    1$', fontsize=12)
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')
ax1.set_zlabel('$z$')
ax1.set_xlim(-2, 2)
ax1.set_ylim(-2, 2)
ax1.set_zlim(-2, 2)
ax1.view_init(elev=20, azim=30)

# Zweischaliges Hyperboloid
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(X, Y, Z2_upper, color='lightcoral',
    alpha=0.8, edgecolor='none', linewidth=0)
ax2.plot_surface(X, Y, Z2_lower, color='lightcoral',
    alpha=0.8, edgecolor='none', linewidth=0)
ax2.set_title('Zweischaliges Hyperboloid\n$x^2 + y^2 - z^2 =
    -1$', fontsize=12)
ax2.set_xlabel('$x$')
ax2.set_ylabel('$y$')
ax2.set_zlabel('$z$')
ax2.set_xlim(-2, 2)
ax2.set_ylim(-2, 2)
ax2.set_zlim(-2, 2)
ax2.view_init(elev=20, azim=30)

# Layout anpassen
plt.tight_layout()

# Speichern als PDF (ideal für LaTeX)
#plt.savefig('hyperboloid_plot.pdf', bbox_inches='tight',
    dpi=300)

# Optional: Auch als PNG speichern
plt.savefig('hyperboloid_plot.png', bbox_inches='tight',
    dpi=300)

# Anzeigen
```

```
plt.show()
plt.close()
```

Listing A.6: Visualisierung Hyperboloid

A.7 Einschaliges Hyperboloid, (Abschnitt. 4.2.1)

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

epsilon = 1.0
u = np.linspace(-2, 2, 50)
phi = np.linspace(0, 2 * np.pi, 50)
U, Phi = np.meshgrid(u, phi)

X = epsilon * np.sinh(U) * np.cos(Phi)
Y = epsilon * np.sinh(U) * np.sin(Phi)
Z = epsilon * np.cosh(U)

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
ax.set_title('Einschaliges Hyperboloid parametrisiert durch
    $H_\\varepsilon(x)$')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')
plt.show()
plt.close()
```

Listing A.7: Visualisierung einschaliges Hyperboloid

A.8 Regularisierter Lorentz-Boost, (Abschnitt. 4.3.3)

```
# lorentz_boost_regulisiert.py
import numpy as np
import matplotlib.pyplot as plt

c = 1.0      # Lichtgeschwindigkeit normiert
```

```
epsilon = 0.1 # Regularisierungsparameter
# Vermeide v nahe c um numerische Probleme zu verhindern
v = np.linspace(0, 0.9999 * c, 500)  # Stärker von c weg

# Physikalischer Lorentzfaktor (divergiert bei v -> c)
gamma_phys = 1 / np.sqrt(1 - (v/c)**2)

# Regularisierter Lorentzfaktor
gamma_reg = 1 / np.sqrt((1 - (v/c)**2) + epsilon**2)

plt.figure(figsize=(10, 6))
plt.plot(v, gamma_phys, label=r'Physikalischer $\gamma(v) =
    1/\sqrt{1 - v^2/c^2}$', color='red', linestyle='--')
plt.plot(v, gamma_reg, label=rf'Regularisierter
    $\gamma_{{\varepsilon}}(v), \varepsilon={epsilon}$',
    color='blue', linewidth=2)
plt.axvline(x=c, color='gray', linestyle=':',
    label='Lichtgeschwindigkeit $c$')
plt.xlabel('Geschwindigkeit $v$')
plt.ylabel('Lorentzfaktor $\\gamma$')
plt.title('Vergleich: Physikalischer vs. regularisierter
    Lorentzfaktor')
plt.legend()
plt.grid(True, alpha=0.3)
plt.ylim(0, 12)  # Begrenzung zur besseren Darstellung
plt.savefig('lorentz_boost_comparison.png')
plt.show()

# Optional: Transformation eines Raumzeit-Punktes
t, x = 1.0, 0.5
beta = v / c

# Sicherere Berechnung der Rapidität mit Fehlerbehandlung
mask_phys = beta < 1.0  # Nur Werte wo beta < 1
theta_phys = np.zeros_like(beta)
theta_phys[mask_phys] = np.arcsinh(beta[mask_phys] /
    np.sqrt(1 - beta[mask_phys]**2))

theta_reg = np.arcsinh(beta / np.sqrt((1 - beta**2) +
    epsilon**2))

t_phys = t * np.cosh(theta_phys) - x * np.sinh(theta_phys)
x_phys = x * np.cosh(theta_phys) - t * np.sinh(theta_phys)

```

```
t_reg  = t * np.cosh(theta_reg)  - x * np.sinh(theta_reg)
x_reg  = x * np.cosh(theta_reg)  - t * np.sinh(theta_reg)

plt.figure(figsize=(10,6))
plt.plot(v[mask_phys], t_phys[mask_phys], label='t\' (physikalisch)', color='red', linestyle='--')
plt.plot(v, t_reg,  label='t\' (regularisiert)', color='blue')
plt.xlabel('Geschwindigkeit $v$')
plt.ylabel('Transformierte Zeitkoordinate $t\'$')
plt.title('Transformierte Zeit unter Lorentz-Boost (regularisiert vs. physikalisch)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('lorentz_boost_time.png')
plt.show()
plt.close("all")

# ========================================
# KORRIGIERTE DEBUG INFORMATIONEN
# ========================================

print("\n" + "="*70)
print("KORRIGIERTE DEBUG AUSWERTUNG - LORENTZ BOOST SIMULATION")
print("="*70)

# 1. Basis Parameter
print(f"\n1. BASISPARAMETER:")
print(f"   - Lichtgeschwindigkeit c = {c}")
print(f"   - Regularisierung ε = {epsilon}")
print(f"   - Geschwindigkeitsbereich: v = 0 bis {v[-1]:.6f}c ({(v[-1]/c)*100:.2f}% von c)")
print(f"   - Anzahl Datenpunkte: {len(v)}")
print(f"   - Gültige physikalische Punkte: {np.sum(mask_phys)}/{len(v)}")
print(f"   - Raumzeit-Punkt: (t, x) = ({t}, {x})")

# 2. Extremwerte Analyse (nur gültige Werte)
print(f"\n2. EXTREMWERTE ANALYSE:")
valid_gamma_phys = gamma_phys[mask_phys]
valid_v_phys = v[mask_phys]

if len(valid_gamma_phys) > 0:
```

```
    print(f"   Physikalischer Lorentzfaktor (nur gültige Werte):")
    print(f"   - Min: γ_min = {np.min(valid_gamma_phys):.4f} bei v = {valid_v_phys[np.argmin(valid_gamma_phys)]:.4f}c")
    print(f"   - Max: γ_max = {np.max(valid_gamma_phys):.4f} bei v = {valid_v_phys[np.argmax(valid_gamma_phys)]:.4f}c")
else:
    print(f"   Physikalischer Lorentzfaktor: KEINE GÜLTIGEN WERTE")

print(f"   Regularisierter Lorentzfaktor:")
print(f"   - Min: γ_min = {np.min(gamma_reg):.4f} bei v = {v[np.argmin(gamma_reg)]:.4f}c")
print(f"   - Max: γ_max = {np.max(gamma_reg):.4f} bei v = {v[np.argmax(gamma_reg)]:.4f}c")

# 3. Vergleich bei hohen Geschwindigkeiten (nur gültige Werte)
high_speed_mask = (v > 0.9*c) & mask_phys
high_speed_idx = np.where(high_speed_mask)[0]

if len(high_speed_idx) > 0:
    print(f"\n3. VERGLEICH BEI HOHEN GESCHWINDIGKEITEN (v > 0.9c, nur gültige):")
    if len(high_speed_idx) >= 4:
        sample_indices = [high_speed_idx[0], high_speed_idx[len(high_speed_idx)//2], high_speed_idx[-2], high_speed_idx[-1]]
    else:
        sample_indices = high_speed_idx

    for idx in sample_indices:
        if idx < len(v):
            v_sample = v[idx]
            gamma_p = gamma_phys[idx]
            gamma_r = gamma_reg[idx]
            diff = gamma_p - gamma_r
            rel_diff = (diff / gamma_r) * 100 if gamma_r != 0 else float('inf')

            print(f"   v = {v_sample:.4f}c ({v_sample/c*100:.1f}% von c):")
            print(f"     γ_phys = {gamma_p:8.4f}, γ_reg = {gamma_r:8.4f}")
```

```
112             print(f"     Differenz: {diff:8.4f}
    ({rel_diff:+.2f}%)")
113
114 # 4. Rapiditäts-Analyse (nur gültige Werte)
115 print(f"\n4. RAPIDITÄTS-ANALYSE:")
116 if len(theta_phys[mask_phys]) > 0:
117     print(f"   Theta physikalisch (gültig):
    {np.min(theta_phys[mask_phys]):.4f} bis
    {np.max(theta_phys[mask_phys]):.4f}")
118 print(f"   Theta regularisiert: {np.min(theta_reg):.4f} bis
    {np.max(theta_reg):.4f}")
119
120 # 5. Transformations-Analyse (nur gültige Werte)
121 print(f"\n5. TRANSFORMATIONS-ERGEBNISSE:")
122 if len(valid_v_phys) > 0:
123     print(f"   Bei v = {valid_v_phys[-1]:.4f}c (maximale
    gültige Geschwindigkeit):")
124     print(f"   Physikalisch:  t' =
    {t_phys[mask_phys][-1]:.4f}, x' =
    {x_phys[mask_phys][-1]:.4f}")
125 print(f"   Bei v = {v[-1]:.4f}c (maximale simulierte
    Geschwindigkeit):")
126 print(f"   Regularisiert: t' = {t_reg[-1]:.4f}, x' =
    {x_reg[-1]:.4f}")
127
128 # 6. Numerische Stabilität
129 print(f"\n6. NUMERISCHE STABILITÄT:")
130 print(f"   NaN-Werte in physikalischer Lösung:
    {np.sum(np.isnan(gamma_phys))}")
131 print(f"   Inf-Werte in physikalischer Lösung:
    {np.sum(np.isinf(gamma_phys))}")
132 print(f"   Ungültige Werte in physikalischer Lösung:
    {np.sum(~mask_phys)}")
133 print(f"   NaN-Werte in regularisierter Lösung:
    {np.sum(np.isnan(gamma_reg))}")
134 print(f"   Inf-Werte in regularisierter Lösung:
    {np.sum(np.isinf(gamma_reg))}")
135
136 # 7. Effekt der Regularisierung
137 if len(valid_gamma_phys) > 0 and len(gamma_reg) > 0:
138     gamma_ratio = gamma_reg[-1] / valid_gamma_phys[-1] if
    valid_gamma_phys[-1] != 0 else float('inf')
139     print(f"\n7. REGULARISIERUNGS-EFFEKT:")
```

```
    print(f"   γ_reg / γ_phys bei v_max (gültig): {gamma_ratio:.4f}")
    print(f"   Regularisierung verhindert Divergenz: {'JA' if not np.isinf(gamma_reg[-1]) else 'NEIN'}")

# 8. Qualitätskontrolle
print(f"\n8. QUALITÄTSKONTROLLE:")
print(f"   Simulation erfolgreich: {'JA' if np.sum(mask_phys) > 0 and np.sum(np.isnan(gamma_reg)) == 0 else 'NEIN'}")
print(f"   Maximaler γ-Wert begrenzt: {'JA' if np.max(gamma_reg) < 100 else 'NEIN'}")
print(f"   Numerische Stabilität: {'GUT' if np.sum(np.isnan(gamma_reg)) == 0 and np.sum(np.isinf(gamma_reg)) == 0 else 'SCHLECHT'}")

print("\n" + "="*70)
print("DEBUG AUSWERTUNG ABGESCHLOSSEN")
print("="*70)
```

Listing A.8: Visualisierung regularisierter Lorentz-Boost

A.9 3d-Potential-Regularisierung, (Abschnitt. 5.2.2)

```
# 3d_Potential_Regularisierung.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def regularized_potential(r, epsilon=0.5):
    """Regularisiertes 1/r Potential"""
    return 1 / np.sqrt(r**2 + epsilon**2)

def exact_potential(r):
    """Exaktes 1/r Potential"""
    return 1 / r

def gradient_regularized(r, epsilon=0.5):
    """Gradient des regularisierten Potentials"""
    return -r / (r**2 + epsilon**2)**(1.5)

def gradient_exact(r):
    """Gradient des exakten Potentials"""
```

```
    return -1 / r**2

# Parameter
epsilon = 0.5
N = 40
r_max = 5.2

# 3D-Gitter erstellen
x = np.linspace(-r_max, r_max, N)
y = np.linspace(-r_max, r_max, N)
z = np.linspace(-r_max, r_max, N)
X, Y, Z = np.meshgrid(x, y, z, indexing='ij')

# Abstand vom Ursprung berechnen
R = np.sqrt(X**2 + Y**2 + Z**2)

# Potentiale berechnen
Phi_regularized = regularized_potential(R, epsilon)
Phi_exact = exact_potential(R)

# Gradienten berechnen (nur für r > 0 definiert)
dPhi_dr_reg = gradient_regularized(R, epsilon)
dPhi_dr_exact = gradient_exact(R)

# Werte für r=0 vermeiden
R_positive = R[R > 0]
Phi_reg_positive = Phi_regularized[R > 0]
Phi_exact_positive = Phi_exact[R > 0]

print("="*70)
print("VERBESSERTE ANALYSE - REGULARISIERTES POTENTIAL")
print("="*70)

print(f"\n1. PARAMETER:")
print(f"   Regularisierung ε = {epsilon}")
print(f"   Gittergröße: {N}³ = {N**3:,} Punkte")
print(f"   Simulationsbereich: r ∈ [0, {r_max:.2f}]")

print(f"\n2. POTENTIALBERICHT:")
print(f"   Regularisiert: Φ_min =
    {Phi_reg_positive.min():.4f}, Φ_max =
    {Phi_reg_positive.max():.4f}")
print(f"   Exakt:         Φ_min =
    {Phi_exact_positive.min():.4f}, Φ_max =
```

```
    {Phi_exact_positive.max():.4f}")

print(f"\n3. REGULARISIERUNGS-EFFEKT:")
print("   r      Φ_exakt   Φε_       Abweichung")
print("   " + "-" * 40)

# Korrektur: Nächste verfügbare Gitterpunkte verwenden
r_values = [0.01, 0.10, 0.50, 1.00, 2.00, 3.00]

for r_val in r_values:
    # Nächsten verfügbaren Gitterpunkt finden
    idx = np.argmin(np.abs(R_positive - r_val))
    r_actual = R_positive[idx]
    phi_ex = Phi_exact_positive[idx]
    phi_reg = Phi_reg_positive[idx]
    abweichung = 100 * abs(phi_ex - phi_reg) / phi_ex

    print(f"   {r_val:4.2f}   {phi_ex:7.3f}   {phi_reg:7.3f}
     {abweichung:7.2f}%")

print(f"\n4. GRADIENTENANALYSE:")
print(f"   ∇Φ||_min = {np.abs(dPhi_dr_reg[R >
    0]).min():.4f}")
print(f"   ∇Φ||_max = {np.abs(dPhi_dr_reg[R >
    0]).max():.4f}")

# Korrektur: Gradient bei spezifischen r-Werten mit nächsten
    Punkten
print(f"\n5. GRADIENTEN BEI SPEZIFISCHEN PUNKTEN:")
r_test_values = [0.1, 0.5, 1.0, 2.0]

for r_val in r_test_values:
    # Nächsten verfügbaren Punkt finden
    mask = R > 0
    available_r = R[mask]
    idx = np.argmin(np.abs(available_r - r_val))
    r_actual = available_r[idx]

    grad_reg = dPhi_dr_reg[mask][idx]
    grad_ex = dPhi_dr_exact[mask][idx] if r_actual > 1e-10
    else 0

    print(f"   r={r_val:.1f}: ∇Φ_reg = {grad_reg:.4f},
    ∇Φ_exakt = {grad_ex:.4f}")
```

```
98
99  # Plot 1: Potentialvergleich
100 fig1 = plt.figure(figsize=(15, 10))
101
102 # Subplot 1: Regularisiertes Potential (xy-Ebene)
103 ax1 = fig1.add_subplot(221)
104 z_slice = N // 2  # Mittlere z-Ebene
105 im1 = ax1.imshow(Phi_regularized[:, :, z_slice].T,
106                  extent=[-r_max, r_max, -r_max, r_max],
107                  origin='lower', cmap='viridis')
108 ax1.set_title('Regularisiertes Potential Φ(r) = √1/(r²ε+²)')
109 ax1.set_xlabel('x')
110 ax1.set_ylabel('y')
111 plt.colorbar(im1, ax=ax1)
112
113 # Subplot 2: Exaktes Potential (xy-Ebene)
114 ax2 = fig1.add_subplot(222)
115 im2 = ax2.imshow(Phi_exact[:, :, z_slice].T,
116                  extent=[-r_max, r_max, -r_max, r_max],
117                  origin='lower', cmap='viridis')
118 ax2.set_title('Exaktes Potential Φ(r) = 1/r')
119 ax2.set_xlabel('x')
120 ax2.set_ylabel('y')
121 plt.colorbar(im2, ax=ax2)
122
123 # Subplot 3: Differenz
124 ax3 = fig1.add_subplot(223)
125 difference = Phi_regularized - Phi_exact
126 im3 = ax3.imshow(difference[:, :, z_slice].T,
127                  extent=[-r_max, r_max, -r_max, r_max],
128                  origin='lower', cmap='RdBu_r')
129 ax3.set_title('Differenz: Φ_regularisiert - Φ_exakt')
130 ax3.set_xlabel('x')
131 ax3.set_ylabel('y')
132 plt.colorbar(im3, ax=ax3)
133
134 # Subplot 4: Radialer Schnitt
135 ax4 = fig1.add_subplot(224)
136 r_vals = np.linspace(0.1, r_max, 100)
137 ax4.plot(r_vals, exact_potential(r_vals), 'r-',
    label='Exakt: 1/r', linewidth=2)
138 ax4.plot(r_vals, regularized_potential(r_vals, epsilon),
    'b--',
```

```
        label=f'Regularisiert: √1/(r²ε+²), ε={epsilon}',
    linewidth=2)
ax4.set_xlabel('r')
ax4.set_ylabel('Φ(r)')
ax4.set_title('Radialer Potentialverlauf')
ax4.legend()
ax4.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('potential_comparison.png', dpi=300)
plt.show()

# Plot 2: 3D-Darstellung
fig2 = plt.figure(figsize=(12, 5))

# Subplot 1: 3D regularisiert
ax1 = fig2.add_subplot(121, projection='3d')
# Reduzierte Punktzahl für bessere Visualisierung
skip = 2
X_plot = X[::skip, ::skip, N//2]
Y_plot = Y[::skip, ::skip, N//2]
Z_plot = Phi_regularized[::skip, ::skip, N//2]

surf1 = ax1.plot_surface(X_plot, Y_plot, Z_plot,
    cmap='viridis', alpha=0.8)
ax1.set_title('3D: Regularisiertes Potential')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_zlabel('Φ(r)')

# Subplot 2: 3D exakt
ax2 = fig2.add_subplot(122, projection='3d')
Z_plot2 = Phi_exact[::skip, ::skip, N//2]
# Begrenzung für bessere Visualisierung
Z_plot2 = np.clip(Z_plot2, 0, 5)

surf2 = ax2.plot_surface(X_plot, Y_plot, Z_plot2,
    cmap='viridis', alpha=0.8)
ax2.set_title('3D: Exaktes Potential (begrenzt auf Φ ≤ 5)')
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_zlabel('Φ(r)')

plt.tight_layout()
```

```
plt.savefig('3d_potential_comparison.png', dpi=300)
plt.show()
plt.close('all')

print(f"\n6. ZUSAMMENFASSUNG:")
print(f"   □ Regularisierung verhindert Singularität bei r=0")
print(f"   □ Abweichung < 2% für r ≥ 2.0")
print(f"   □ Gradient bleibt überall beschränkt")
print(f"   □ Numerisch stabil für Simulationen")
```

Listing A.9: Visualisierung 3d-Potential-Regularisierung

A.10 2D-Wellengleichung mit regularisierten Delta-Quellen, (Abschnitt. 5.3.7)

```
# 2d_wellengleichung_e_hyp_gif_animation.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Parameter
c = 1.0  # Wellengeschwindigkeit
epsilon = 0.1  # Regularisierungsparameter
L = 5.0  # Gebiet [-L/2, L/2]²
N = 150  # Gittergröße
dx = L / N  # Räumliche Schrittweite
dt = 0.4 * dx / c  # Zeitliche Schrittweite (CFL-Bedingung)
duration = 10.0  # Simulationsdauer
fps = 15  # Frames pro Sekunde
total_frames = int(duration / dt)  # Frames basierend auf dt

# Gitter erstellen
x = np.linspace(-L/2, L/2, N)
y = np.linspace(-L/2, L/2, N)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)

# Regularisierte Delta-Quelle basierend auf e-Hyperbelfunktion (2D-Näherung)
def regularized_delta_ehyper(r, eps):
```

```
    return (1 / (2 * np.pi * eps**2)) * np.exp(-r / eps) / (r + 1e-10)  # Vermeide Division durch 0

# Glatte Heaviside-Anfangsbedingung
R0 = 1.0  # Radius für Anfangsbedingung
u = 0.5 * (1 + np.tanh((R0 - R) / epsilon))  # H_epsilon-basierte Anfangsbedingung

# Zweite Quelle für Interferenz
source2 = regularized_delta_ehyper(np.sqrt((X-1)**2 + (Y-1)**2), epsilon)
u += 0.5 * source2  # Zweite Quelle bei (1, 1)

# Initialisierung
u_prev = np.zeros((N, N))

# Animation Setup
fig = plt.figure(figsize=(12, 8))

# Hauptplot
ax = plt.axes([0, 0.25, 0.73, 0.53])
im = ax.imshow(u, extent=[-L/2, L/2, -L/2, L/2],
               cmap='inferno', origin='lower')

# Colorbar
cbar = plt.colorbar(im, ax=ax, shrink=0.8, pad=0.02)
cbar.set_label('Amplitude u(x,y,t)', fontsize=12)

# Titel und Autor
title_text = fig.text(0.5, 0.88, '2D Wellengleichung mit e-Hyperbelfunktion',
                      fontsize=14, fontweight='bold', ha='center')
name_text = fig.text(0.5, 0.84, 'Visualisierung: Klaus H. Dieckmann, 2025',
                     fontsize=12, ha='center', style='italic')

# Erklärungstext
explanation_box = plt.axes([0.2, 0.12, 0.75, 0.20], facecolor='lightgray')
explanation_box.axis('off')
explanation_text = explanation_box.text(0.02, 0.26, '', fontsize=12, ha='left', va='top', wrap=True)
```

```
explanation_box.set_xlim(0, 1)
explanation_box.set_ylim(0, 1)

# Zeit-, Phasen- und Frame-Anzeige
time_text = ax.text(0.98, 0.86, '', fontsize=10, ha='right', va='top',
                    transform=ax.transAxes, color='red', fontweight='bold',
                    bbox=dict(boxstyle='round', facecolor='white', alpha=0.9))
phase_text = ax.text(0.02, 0.95, '', fontsize=12, ha='left', va='top',
                     transform=ax.transAxes, color='blue', fontweight='bold',
                     bbox=dict(boxstyle='round', facecolor='lightblue', alpha=0.9))
frame_text = ax.text(0.02, 0.85, '', fontsize=10, ha='left', va='top',
                    transform=ax.transAxes, color='green', fontweight='bold',
                    bbox=dict(boxstyle='round', facecolor='white', alpha=0.9))

ax.set_xlabel('x', fontsize=12)
ax.set_ylabel('y', fontsize=12)
ax.set_aspect('equal')

# Parameter-Info
param_box = plt.axes([0.1, 0.78, 0.75, 0.05], facecolor='yellow', alpha=0.2)
param_box.axis('off')
param_text = param_box.text(0.5, 0.5, f'Parameter: ε = {epsilon}, c = {c}, Gebiet: [-{L/2}, {L/2}]²',
                            fontsize=11, ha='center', va='center')

# Phasen-Erklärungen (angepasst für 10 Sekunden)
phase_explanations = [
    (0.0, 0.5, "PHASE 1: Quellenaktivierung",
     "Eine e-Hyperbelfunktion-basierte Quelle und eine glatte Anfangsbedingung werden aktiviert.\nDie Wellenausbreitung beginnt."),
    (0.5, 2.5, "PHASE 2: Kugelwellen-Ausbreitung",
```

```
    "Kugelförmige Wellen breiten sich mit c=1.0 von (0,0) und (1,1) aus.\nWellenfronten sind als konzentrierte Ringe sichtbar."),
    (2.5, 5.0, "PHASE 3: Randreflexionen",
     "Wellen erreichen die Ränder bei x,y=±2.5 und werden reflektiert.\nEinlaufende und reflektierte Wellen überlagern sich."),
    (5.0, 8.0, "PHASE 4: Komplexe Interferenz",
     "Reflektierte Wellen bilden komplexe Interferenzmuster.\nStehende Wellen und klare Maxima sind erkennbar."),
    (8.0, 10.0, "PHASE 5: Energiedissipation",
     "Die Wellenenergie verteilt sich durch Mehrfachreflexionen.\nDas System nähert sich einem gleichmäßig verteilten Zustand.")
]

def get_phase_info(current_time):
    for start_time, end_time, phase, explanation in phase_explanations:
        if start_time <= current_time < end_time:
            return phase, explanation
    return "Simulation abgeschlossen", "Die Simulation hat die maximale Zeit erreicht."

def update(frame):
    global u, u_prev

    # Finite-Differenzen
    laplacian = (
        np.roll(u, 1, axis=0) + np.roll(u, -1, axis=0) +
        np.roll(u, 1, axis=1) + np.roll(u, -1, axis=1) - 4*u
    ) / dx**2

    u_next = 2*u - u_prev + (c*dt)**2 * laplacian

    # Reflektierende Randbedingungen
    u_next[0, :] = 0
    u_next[-1, :] = 0
    u_next[:, 0] = 0
    u_next[:, -1] = 0

    # Update
    u_prev = u.copy()
```

```
    u = u_next.copy()

    # Aktualisiere Plot
    im.set_array(u)

    # Dynamische Farbskala
    amplitude = max(abs(np.min(u)), abs(np.max(u))) * 0.8
    im.set_clim(vmin=-amplitude, vmax=amplitude)

    # Aktuelle Zeit
    current_time = frame * dt

    # Zeit-, Phasen- und Frame-Anzeige
    time_text.set_text(f'Zeit: t = {current_time:.1f} s')
    phase, explanation = get_phase_info(current_time)
    phase_text.set_text(f'{phase}')
    explanation_text.set_text(explanation)
    #frame_text.set_text(f'Frame: {frame}')

    # Debugging-Ausgabe
    print(f"Frame {frame}, Zeit {current_time:.1f}, Phase
    {phase}")

    return [im, explanation_text, time_text, phase_text,
    frame_text]

# Erstelle Animation
print(f"Animation: {duration} Sekunden, {total_frames}
    Frames, dt = {dt:.4f}")
print(f"Frame-Dauer: {1000/fps:.1f} ms, Frames pro Sekunde:
    {fps}")
print("Phasen der Simulation:")
for start_time, end_time, phase, _ in phase_explanations:
    print(f"  Zeit {start_time:4.1f}-{end_time:4.1f}s:
    {phase}")

ani = FuncAnimation(fig, update, frames=total_frames,
    blit=False, interval=1000/fps)

# Speichere Animation als GIF
ani.save('wellengleichung_e_hyperbel.gif', writer='pillow',
    fps=fps, dpi=80)

# Optional: Zeige Animation (auskommentiert)
```

```
plt.show()
```

Listing A.10: Visualisierung 2D-Wellengleichung mit regularisierten Delta-Quellen

A.11 Wellenausbreitung: Glatte Quellen (Animation), (Abschnitt. 5.4)

```
# wellenausbreit_glatte_quellen_gif_animation.py
# Simulation der 2D-Wellengleichung mit glatter Anfangsbedingung und regularisierter Quelle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import os

# ========================
# SIMULATIONS-PARAMETER
# ========================
c = 1.0          # Wellengeschwindigkeit
epsilon = 0.1    # Regularisierungsparameter (glättet Quellen und Anfangsbedingung)
L = 5.0          # Simulationsgebiet: [-L/2, L/2]²
N = 150          # Gitterauflösung (N x N Punkte)
dx = L / N       # räumliche Schrittweite
dt = 0.4 * dx / c  # Zeitschritt (CFL-Bedingung für Stabilität)
duration = 10.0  # Gesamtdauer der Simulation in Sekunden
fps = 15         # Frames pro Sekunde im GIF
total_frames = int(duration / dt)  # Anzahl der zu berechnenden Zeitschritte

# ========================
# GITTER UND ANFANGSZUSTAND
# ========================
x = np.linspace(-L/2, L/2, N)
y = np.linspace(-L/2, L/2, N)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)

# HAUPTBEITRAG DIESES ABSCHNITTS: Glatte Heaviside-Anfangsbedingung
```

```
# Ersetzt den scharfen Sprung durch eine differenzierbare Übergangsfunktion
R0 = 1.0  # Radius der initialen "Scheibe"
u = 0.5 * (1 + np.tanh((R0 - R) / epsilon))  # tanh-basierte Glättung

# Zusätzliche regularisierte Delta-Quelle bei (1,1) für Interferenzmuster
def reg_delta(r, eps):
    """Regularisierte 2D-Delta-Quelle basierend auf e-Hyperbelfunktion."""
    return (1 / (2 * np.pi * eps**2)) * np.exp(-r / eps) / (r + 1e-10)

source2 = reg_delta(np.sqrt((X - 1)**2 + (Y - 1)**2), epsilon)
u += 0.5 * source2  # Superposition: Anfangsbedingung + Quelle

# Initialisierung für Finite-Differenzen (zweite Ordnung in Zeit)
u_prev = np.zeros_like(u)  # Zustand bei t = -dt
u_current = u.copy()       # Zustand bei t = 0
u_next = np.zeros_like(u)  # Zustand bei t = +dt (wird berechnet)

# ========================
# PLOT-VORBEREITUNG
# ========================
fig, ax = plt.subplots(figsize=(10, 8))
ax.set_title('Wellenausbreitung mit glatter Anfangsbedingung\n'
             'Visualisierung: Klaus H. Dieckmann, 2025', fontsize=12, fontweight='bold')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_aspect('equal')

# Initiales Bild (t=0)
im = ax.imshow(u_current, extent=[-L/2, L/2, -L/2, L/2],
               cmap='inferno', origin='lower', vmin=-0.5, vmax=1.5)
plt.colorbar(im, ax=ax, shrink=0.8, label='Amplitude u(x,y,t)')
```

```
61
62 # Textelemente für Zeit und Phase
63 time_text = ax.text(0.02, 0.96, '', transform=ax.transAxes,
       fontsize=10,
64                      color='white',
       bbox=dict(boxstyle="round", facecolor="black", alpha=0.7))
65 phase_text = ax.text(0.02, 0.85, '', transform=ax.transAxes,
       fontsize=10,
66                       color='yellow', fontweight='bold',
67                       bbox=dict(boxstyle="round",
       facecolor="navy", alpha=0.7))
68
69 # Phasen-Definition für erklärenden Text
70 phase_explanations = [
71     (0.0, 0.5, "PHASE 1: Aktivierung\nglatter Quellen"),
72     (0.5, 2.5, "PHASE 2: Kugelwellen\nbreiten sich aus"),
73     (2.5, 5.0, "PHASE 3: Reflexion\nan Rändern"),
74     (5.0, 8.0, "PHASE 4: Komplexe\nInterferenz"),
75     (8.0, 10.0, "PHASE 5: Energie\ndissipiert")
76 ]
77
78 def get_phase_label(current_time):
79     """Gibt die aktuelle Phase als Label zurück."""
80     for start, end, label in phase_explanations:
81         if start <= current_time < end:
82             return label
83     return "ENDE"
84
85 # ========================
86 # SIMULATIONSSCHLEIFE
87 # ========================
88 def update(frame):
89     global u_prev, u_current, u_next
90
91     # Finite-Differenzen: Diskretisierung der Wellengleichung
92     laplacian = (
93         np.roll(u_current, 1, axis=0) + np.roll(u_current,
       -1, axis=0) +
94         np.roll(u_current, 1, axis=1) + np.roll(u_current,
       -1, axis=1) - 4 * u_current
95     ) / dx**2
96
97     # Update nach explizitem Schema (zweite Ordnung in Zeit)
98     u_next = 2 * u_current - u_prev + (c * dt)**2 * laplacian
```

```
99
100     # Reflektierende Randbedingungen (Dirichlet, u=0 am Rand)
101     u_next[0, :] = 0
102     u_next[-1, :] = 0
103     u_next[:, 0] = 0
104     u_next[:, -1] = 0
105
106     # Zustände aktualisieren (Time-Stepping)
107     u_prev = u_current.copy()
108     u_current = u_next.copy()
109
110     # Plot aktualisieren
111     im.set_array(u_current)
112
113     # Dynamische Farbskala (optional, für besseren Kontrast)
114     # max_amp = np.max(np.abs(u_current)) * 0.9
115     # im.set_clim(vmin=-max_amp, vmax=max_amp)
116
117     # Zeit und Phase anzeigen
118     current_time = frame * dt
119     time_text.set_text(f't = {current_time:.1f} s')
120     phase_text.set_text(get_phase_label(current_time))
121
122     return [im, time_text, phase_text]
123
124 # ========================
125 # ANIMATION ERSTELLEN & SPEICHERN
126 # ========================
127 print(f"Starte Simulation: {total_frames} Frames, Dauer
        {duration}s, FPS {fps}")
128
129 ani = FuncAnimation(fig, update, frames=total_frames,
        blit=True, interval=1000/fps)
130
131 plt.show()
132 # GIF speichern mit pillow (ideal für einfache Distribution)
133 print("Speichere GIF... (dies kann einige Minuten dauern)")
134 ani.save('wellenausbreitung_glatte_quellen.gif',
135          writer='pillow',
136          fps=fps,
137          dpi=100)
138
139 print("□ GIF erfolgreich gespeichert:
        'wellenausbreitung_glatte_quellen.gif'")
```

```
plt.close(fig)  # Speicher freigeben

# Optional: Zeige letztes Frame zur Kontrolle
# plt.figure(figsize=(8,6))
# plt.imshow(u_current, extent=[-L/2, L/2, -L/2, L/2],
    cmap='inferno', origin='lower')
# plt.colorbar(label='Amplitude')
# plt.title('Endzustand der Simulation')
# plt.show()
```

Listing A.11: Visualisierung Wellenausbreitung: Glatte Quellen (Animation)

A.12 RL-Schaltung mit hartem vs. weichem Schalter, (Abschnitt. 6.1.3)

```
# RL_schaltung_mit_schalter.py
"""
Python-Code 5.1: Simulation einer RL-Schaltung mit hartem
    vs. weichem Schalter
MIT DEBUG-AUSGABEN zur Verifikation der Spannungs- und
    Stromverläufe
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# ====== PHYSIKALISCHE PARAMETER ======
L = 0.1    # Induktivität in Henry (H)
R = 10.0   # Widerstand in Ohm Ω()
U0 = 12.0  # Eingangsspannung in Volt (V)
epsilon = 0.001  # Glättungsparameter in Sekunden (s) -
    steuert "Weichheit" des Schalters

# Simulationszeitbereich
t_span = (0, 0.1)   # 0 bis 100 ms
t_eval = np.linspace(0, 0.1, 1000)

# ====== SCHALTER-FUNKTIONEN ======
def hard_switch(t):
```

```
    """Heaviside-Funktion: harter Schalter - springt erst bei t > 0, NICHT bei t=0"""
    return np.where(t > 0, 1.0, 0.0)   # Korrektur: t > 0 statt t >= 0

def soft_switch(t, eps):
    """Weicher Schalter: S_eps(t) = 1 - exp(-t/eps) für t > 0"""
    return np.where(t > 0, 1.0 - np.exp(-t/eps), 0.0)

# ====== DIFFERENTIALGLEICHUNGEN ======
def rl_circuit_hard(t, I):
    U_in = U0 * hard_switch(t)
    dIdt = (U_in - R * I[0]) / L
    return [dIdt]

def rl_circuit_soft(t, I):
    U_in = U0 * soft_switch(t, epsilon)
    dIdt = (U_in - R * I[0]) / L
    return [dIdt]

# ====== LÖSUNG DER DGLs ======
I0 = [0.0]

sol_hard = solve_ivp(rl_circuit_hard, t_span, I0, t_eval=t_eval, method='RK45', max_step=1e-4)
sol_soft = solve_ivp(rl_circuit_soft, t_span, I0, t_eval=t_eval, method='RK45', max_step=1e-4)

U_hard = U0 * hard_switch(sol_hard.t)
U_soft = U0 * soft_switch(sol_soft.t, epsilon)

# ====== PLOTTEN ======
plt.figure(figsize=(12, 8))

# Plot 1: Strom I(t)
plt.subplot(2, 1, 1)
plt.plot(sol_hard.t * 1000, sol_hard.y[0], 'b-', linewidth=2.5, label='Harter Schalter (Heaviside)')
plt.plot(sol_soft.t * 1000, sol_soft.y[0], 'r--', linewidth=2.5, label=f'Weicher Schalter ε( = {epsilon*1000:.1f} ms)')
plt.title('Strom in RL-Schaltung: harter vs. weicher Schalter', fontsize=14, fontweight='bold')
```

```
plt.ylabel('Strom I(t) [A]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.xlim(0, 100)

# Plot 2: Spannung U(t) – MIT BETONTEN MARKIERUNGEN FÜR DEBUGGING
plt.subplot(2, 1, 2)
plt.plot(sol_hard.t * 1000, U_hard, 'b-', linewidth=2.5, label='Harter Schalter (Heaviside)')
plt.plot(sol_soft.t * 1000, U_soft, 'r--', linewidth=2.5, label=f'Weicher Schalter ε( = {epsilon*1000:.1f} ms)')

# Markiere wichtige Punkte zur Verifikation
idx_0 = 0
idx_eps = np.argmin(np.abs(sol_soft.t - epsilon))
idx_5eps = np.argmin(np.abs(sol_soft.t - 5*epsilon))

plt.plot(sol_soft.t[idx_0]*1000, U_soft[idx_0], 'ro', markersize=8, label='t=0')
plt.plot(sol_soft.t[idx_eps]*1000, U_soft[idx_eps], 'go', markersize=8, label=f'tε= ({epsilon*1000:.1f} ms)')
plt.plot(sol_soft.t[idx_5eps]*1000, U_soft[idx_5eps], 'mo', markersize=8, label=f'tε=5 ({5*epsilon*1000:.1f} ms)')

plt.title('Eingangsspannung U(t) – MIT DEBUG-MARKIERUNGEN', fontsize=14, fontweight='bold')
plt.xlabel('Zeit t [ms]', fontsize=12)
plt.ylabel('Spannung U(t) [V]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=10)
plt.xlim(0, 100)

plt.tight_layout()
plt.savefig('RL_Schaltung_mit_Schalter.png', dpi=300)
plt.show()

# ====== DEBUG-AUSGABEN – PLASIBILITÄTSPRÜFUNG ======
print("\n" + "="*60)
print(" DEBUG-AUSGABEN: Plausibilitätsprüfung der Spannungsverläufe")
print("="*60)

# HARTE SCHALTER-PRÜFUNG
```

```
print(f"\n  HARTER SCHALTER:")
print(f"   U(t=0)      = {U_hard[0]:.6f} V (sollte 0.0 sein → {' OK' if U_hard[0] == 0.0 else ' FEHLER'})")
print(f"   U(t>0)      = {U_hard[1]:.6f} V (sollte {U0} sein → {' OK' if np.allclose(U_hard[1:], U0) else ' FEHLER'})")

# WEICHE SCHALTER-PRÜFUNG
print(f"\n  WEICHER SCHALTER ε( = {epsilon:.4f} s = {epsilon*1000:.1f} ms):")
print(f"   U(t=0)      = {U_soft[0]:.6f} V (sollte 0.0 sein → {' OK' if U_soft[0] == 0.0 else ' FEHLER'})")
print(f"   U(tε=)      = {U_soft[idx_eps]:.6f} V (sollte ≈ {U0*(1-np.exp(-1)):.3f} V sein → {' OK' if np.isclose(U_soft[idx_eps], U0*(1-np.exp(-1)), atol=0.1) else '  Abweichung'})")
print(f"   U(tε=5)     = {U_soft[idx_5eps]:.6f} V (sollte ≈ {U0*(1-np.exp(-5)):.3f} V sein → {' OK' if np.isclose(U_soft[idx_5eps], U0*(1-np.exp(-5)), atol=0.1) else '  Abweichung'})")
print(f"   Max(U_soft) = {np.max(U_soft):.6f} V (sollte ≈ {U0} sein → {' OK' if np.isclose(np.max(U_soft), U0, atol=0.1) else '  Nicht erreicht?'})")

# STROM-DEBUG (optional)
print(f"\n  STROM bei t=0:")
print(f"   I_hard(0) = {sol_hard.y[0][0]:.6f} A (sollte 0.0 sein → {' OK' if sol_hard.y[0][0] == 0.0 else ' FEHLER'})")
print(f"   I_soft(0) = {sol_soft.y[0][0]:.6f} A (sollte 0.0 sein → {' OK' if sol_soft.y[0][0] == 0.0 else ' FEHLER'})")

print("\n" + "="*60)
print(" HINWEIS: Der Spannungsverlauf des weichen Schalters ist KORREKT berechnet.")
print("   Er steigt kontinuierlich an - das ist KEIN Fehler, sondern das gewünschte Verhalten!")
print("   Der scheinbare 'Rückstand' ist physikalisch realistisch: kein Schalter schaltet instantan.")
print("="*60)
```

Listing A.12: Visualisierung RL-Schaltung mit hartem vs. weichem Schalter

A.13 Harmonischer Oszillator mit weichem Impuls, (Abschnitt. 6.2.3)

```python
# harmonischer_oszillator_weicher_impuls.py
"""
Python-Code 5.2: Harmonischer Oszillator mit "weichem Impuls" (geglättete Delta-Funktion)
Kapitel: Komplexe Systeme und Grenzflächen - Die hyperbolische e-Funktion als universelles Werkzeug
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.integrate import simpson   # Für numerische Integration des Impulses

# ====== PHYSIKALISCHE PARAMETER ======
m = 1.0         # Masse in kg
k = 4.0         # Federkonstante in N/m
c = 0.1         # Dämpfungskonstante in N·s/m (kleine Dämpfung für klare Schwingung)
F0 = 2.0        # Impulsstärke (Integral der Kraft) in N·s
t0 = 1.0        # Zeitpunkt des Impulses in s

# Oszillator-Kenngrößen
omega0 = np.sqrt(k / m)   # Eigenfrequenz (ungedämpft)
print(f"  Eigenfrequenz ω₀ = {omega0:.3f} rad/s → Periode T = {2*np.pi/omega0:.3f} s")

# ====== WEICHER IMPULS (GAUSS-FUNKTION ALS DELTA-APPROXIMATION) ======
def soft_impulse(t, t_impulse, eps):
    """
    Glatte Delta-Approximation: δε_(t - t_impulse)
    """
    return (1 / (eps * np.sqrt(2 * np.pi))) * np.exp(-(t - t_impulse)**2 / (2 * eps**2))

# ====== DIFFERENTIALGLEICHUNG: GEDÄMPFTER HARMONISCHER OSZILLATOR ======
# Zustandsvektor: y = [x, v] = [Position, Geschwindigkeit]
def oscillator_with_impulse(t, y, eps):
```

```
    x, v = y
    # Kraft = F0 * δε_(t - t0)
    F = F0 * soft_impulse(t, t0, eps)
    # ẍm· + ẋc· + k·x = F(t)  →  ẍ = (F - c·v - k·x) / m
    a = (F - c * v - k * x) / m
    return [v, a]

# ====== ANALYTISCHE LÖSUNG FÜR IDEALEN DELTA-IMPULS (c=0, ungedämpft) ======
def analytical_solution_undamped(t, t_impulse):
    """
    Analytische Lösung für ẍm· + k·x = δF0·(t - t0), mit x(≤tt0)=0, v(≤tt0)=0
    → Sprung in Geschwindigkeit: v(t0+) = F0/m
    → x(t) = (F0/(ωm·0)) · sinω(0·(t - t0)) für t > t0, sonst 0.
    """
    sol = np.zeros_like(t)
    mask = t > t_impulse
    sol[mask] = (F0 / (m * omega0)) * np.sin(omega0 * (t[mask] - t_impulse))
    return sol

# ====== SIMULATIONSPARAMETER ======
t_span = (0, 5)        # Simulationszeit: 0 bis 5 Sekunden
t_eval = np.linspace(0, 5, 2000)  # Hohe Auflösung für glatte Plots

# Verschiedene Glättungsparameter ε (in Sekunden)
epsilons = [0.1, 0.02, 0.005, 0.001]  # Breite des Impulses

# Anfangsbedingung: ruhend bei x=0
y0 = [0.0, 0.0]

# ====== LÖSUNG FÜR VERSCHIEDENE ε ======
solutions = {}
for eps in epsilons:
    sol = solve_ivp(
        lambda t, y: oscillator_with_impulse(t, y, eps),
        t_span, y0, t_eval=t_eval, method='RK45', max_step=0.01
    )
    solutions[eps] = sol

```

```
# Analytische Lösung (nur für Vergleich, ungedämpft & idealer Impuls)
x_analytic = analytical_solution_undamped(t_eval, t0)

# ====== PLOTTEN ======
plt.figure(figsize=(14, 8))

# Plot 1: Auslenkung x(t)
plt.subplot(2, 1, 1)
colors = ['orange', 'red', 'purple', 'brown']
for i, eps in enumerate(epsilons):
    plt.plot(solutions[eps].t, solutions[eps].y[0], color=colors[i], linewidth=2,
             label=f'ε = {eps:.3f} s')

plt.plot(t_eval, x_analytic, 'k--', linewidth=2.5, label='Analytisch δ(-Impuls, ungedämpft)')
plt.axvline(x=t0, color='gray', linestyle=':', linewidth=1.5, label=f'Impulszeit ₀t = {t0} s')

plt.title('Harmonischer Oszillator: Auslenkung nach "weichem Impuls" - Vergleich mit analytischer Lösung', fontsize=13, fontweight='bold')
plt.ylabel('Auslenkung x(t) [m]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=10, loc='upper right')
plt.xlim(0, 5)

# Plot 2: Impulsfunktion δε_(t) für verschiedene ε (zur Visualisierung)
plt.subplot(2, 1, 2)
t_impulse_plot = np.linspace(t0 - 0.5, t0 + 0.5, 1000)
for i, eps in enumerate(epsilons):
    impulse = F0 * soft_impulse(t_impulse_plot, t0, eps)
    plt.plot(t_impulse_plot, impulse, color=colors[i], linewidth=2, label=f'ε = {eps:.3f} s')

plt.title('Verlauf der Impulsfunktion $F(t) = F_0 \\cdot \\delta_\\varepsilon(t - t_0)$', fontsize=13, fontweight='bold')
plt.xlabel('Zeit t [s]', fontsize=12)
plt.ylabel('Kraft F(t) [N]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=10)
```

```
plt.xlim(t0 - 0.3, t0 + 0.3)
plt.ylim(0, None)

plt.tight_layout()
plt.savefig('harmonischer_oszillator_weicher_impuls.png', dpi=300)
plt.show()
plt.close()

# ====== DEBUG & PLAUSIBILITÄTSPRÜFUNG ======
print("\n" + "="*70)
print("□ DEBUG-AUSGABEN: Plausibilitätsprüfung des weichen Impulses")
print("="*70)

for eps in epsilons:
    # Berechne numerisch das Integral des Impulses (sollte ≈ F0 sein)
    t_check = np.linspace(t0 - 10*eps, t0 + 10*eps, 5000)
    impulse_vals = F0 * soft_impulse(t_check, t0, eps)
    impulse_integral = simpson(impulse_vals, t_check)
    print(f"ε = {eps:.4f} s → □ F(t) dt = {impulse_integral:.6f} N·s "
          f"(sollte ≈ {F0} sein → □{' OK' if np.isclose(impulse_integral, F0, rtol=1e-3) else □' Abweichung'})")

# Maximalwert der analytischen Lösung (Amplitude)
A_analytic = F0 / (m * omega0)
print(f"\nAnalytische Amplitude (nach idealem Impuls): A = {A_analytic:.4f} m")

# Vergleich: Maximalwert der numerischen Lösung für kleinstes ε
eps_min = min(epsilons)
x_num_max = np.max(np.abs(solutions[eps_min].y[0][solutions[eps_min].t > t0]))
print(f"Numerische Amplitude ε( = {eps_min:.4f} s): A_num = {x_num_max:.4f} m "
      f"(Fehler: {100*abs(x_num_max - A_analytic)/A_analytic:.2f} %)")

print("\n" + "="*70)
```

```
print("□ INTERPRETATION:")
print("• Mit kleinerem ε nähert sich die numerische Lösung der analytischen Lösung an.")
print("• Der weiche Impuls erzeugt eine realistische, glatte Anregung – ohne numerische Instabilität.")
print("• Die Gauß-Funktion ist eine exzellente, normierte Approximation der Delta-Distribution.")
print("="*70)
```

Listing A.13: Visualisierung Harmonischer Oszillator mit weichem Impuls

A.14 Dioden-Gleichrichter: real vs. ideal, (Abschnitt. 6.3.1)

```
# dioden_gleichrichter_real_vs_ideal.py
"""
Python-Code 5.3: Dioden-Gleichrichter – ideal vs. real (exponentielle Kennlinie)
Kapitel: Komplexe Systeme und Grenzflächen – Die hyperbolische e-Funktion als universelles Werkzeug
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton

# ====== PHYSIKALISCHE PARAMETER ======
R = 1000.0      # Lastwiderstand in Ohm
U_peak = 5.0    # Amplitude der Eingangsspannung in Volt
f = 50.0        # Frequenz in Hz
omega = 2 * np.pi * f

# Diodenparameter (real)
I_S = 1e-12     # Sättigungsstrom in A
n = 1.7         # Emissionskoeffizient
U_T = 0.025     # Temperaturspannung in V (ca. 25 mV bei 300 K)

# Simulationszeit
t_max = 0.04    # 2 Perioden bei 50 Hz
t = np.linspace(0, t_max, 2000)
```

```
U_in = U_peak * np.sin(omega * t)

# ====== DIODENMODELLE ======
def ideal_diode(U):
    """Ideale Diode: I = 0 für U <= 0, sonst beliebig (hier: unendlich, aber in Schaltung durch R begrenzt)"""
    return np.where(U > 0, U / R, 0.0)  # Näherung: wenn leitend, fällt gesamte Spannung an R ab

def real_diode_current(U_D):
    """Reale Diode: Shockley-Gleichung"""
    return I_S * (np.exp(U_D / (n * U_T)) - 1)

def real_diode_voltage(U_in_val):
    """
    Berechne U_D für gegebenes U_in durch Lösen der impliziten Gleichung mit Newton-Verfahren.
    Nutzt analytische Ableitung für höhere Stabilität und Geschwindigkeit.
    """
    def F(U_D):
        return U_D + R * I_S * (np.exp(U_D / (n * U_T)) - 1) - U_in_val

    def F_prime(U_D):
        return 1 + (R * I_S / (n * U_T)) * np.exp(U_D / (n * U_T))

    # Physikalisch motivierter Startwert
    if U_in_val <= 0:
        U_guess = 0.0
    else:
        arg = U_in_val / (R * I_S) + 1
        if arg > 1e-5:
            U_guess = n * U_T * np.log(arg)
        else:
            U_guess = 0.1
        U_guess = max(0.0, min(U_guess, U_in_val))  # Begrenzung sinnvoll

    try:
        U_D_sol = newton(F, U_guess, fprime=F_prime, tol=1e-8, maxiter=50)
    except RuntimeError:
```

```
        # Fallback: fsolve, falls Newton versagt
        U_D_sol = fsolve(F, U_guess)[0]

    return U_D_sol

# ====== BERECHNUNG ======
# Ideale Diode (vereinfacht)
I_ideal = ideal_diode(U_in)

# Reale Diode (numerisch für jeden Zeitschritt)
U_D_real = np.array([real_diode_voltage(u) for u in U_in])
I_real = (U_in - U_D_real) / R  # Ohmsches Gesetz am
    Widerstand

# ====== PLOTTEN ======
plt.figure(figsize=(14, 10))

# Plot 1: Eingangsspannung
plt.subplot(3, 1, 1)
plt.plot(t * 1000, U_in, 'gray', linestyle='--',
    linewidth=2, label='Eingangsspannung $U_{in}(t)$')
plt.ylabel('Spannung [V]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.title('Halbwellengleichrichter: Ideale vs. reale Diode',
    fontsize=14, fontweight='bold')
plt.xlim(0, t_max * 1000)

# Plot 2: Stromverlauf
plt.subplot(3, 1, 2)
plt.plot(t * 1000, I_ideal * 1000, 'b-', linewidth=2.5,
    label='Ideale Diode')
plt.plot(t * 1000, I_real * 1000, 'r--', linewidth=2.5,
    label='Reale Diode (Shockley)')
plt.ylabel('Strom I(t) [mA]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.xlim(0, t_max * 1000)

# Plot 3: Diodenspannung (nur real)
plt.subplot(3, 1, 3)
plt.plot(t * 1000, U_D_real, 'r-', linewidth=2.5,
    label='Diodenspannung $U_D(t)$ (real)')
plt.xlabel('Zeit t [ms]', fontsize=12)
```

```
plt.ylabel('Spannung [V]', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.xlim(0, t_max * 1000)

plt.tight_layout()
plt.savefig('dioden_gleichrichter_real_vs_ideal.png',
    dpi=300)
plt.show()

# ====== DEBUG & PLAUSIBILITÄT ======
print("\n" + "="*60)
print(" DEBUG: Kennwerte der realen Diode")
print("="*60)
print(f"Sättigungsstrom I_S = {I_S:.1e} A")
print(f"Temperaturspannung U_T = {U_T*1000:.1f} mV")
print(f"Emissionskoeffizient n = {n}")

# Maximalwerte
I_real_max = np.max(I_real) * 1000  # in mA
U_D_max = np.max(U_D_real)
print(f"\nMaximaler Laststrom (real): {I_real_max:.3f} mA")
print(f"Maximale Diodenspannung: {U_D_max:.3f} V")

# Beispiel: Berechne Diodenspannung bei U_in = 1.0 V
U_in_test = 1.0
U_D_test = real_diode_voltage(U_in_test)
I_test = (U_in_test - U_D_test) / R
print(f"\nBeispiel bei U_in = {U_in_test} V:")
print(f"  U_D = {U_D_test:.4f} V")
print(f"  I   = {I_test*1000:.4f} mA")

print("\n" + "="*60)
print(" INTERPRETATION:")
print("• Die ideale Diode erzeugt scharfe Sprünge -
    numerisch problematisch, physikalisch unrealistisch.")
print("• Die reale Diode folgt einer glatten, exponentiellen
    Kennlinie - stabil und physikalisch korrekt.")
print("• Der Übergang erfolgt innerhalb weniger 10 mV -
    typisch für Siliziumdioden.")
print("="*60)
```

Listing A.14: Visualisierung Dioden-Gleichrichter: real vs. ideal

Hinweis zur Nutzung von KI

Die Ideen und Konzepte dieser Arbeit stammen von mir. Künstliche Intelligenz wurde unterstützend für die Textformulierung und Gleichungsformatierung eingesetzt. Die inhaltliche Verantwortung liegt bei mir. [1]

Stand: 26. September 2025
TimeStamp: https://freetsa.org/index_de.php

[1] ORCID: https://orcid.org/0009-0002-6090-3757

Literatur

[1] K. Huff et al. The Scientific Python Ecosystem: An Open Source Success Story. *Computing in Science & Engineering*, 24(3):84–91, 2022. https://doi.org/10.1109/MCSE.2022.3154738.

[2] E. Hairer, S. P. Nørsett, G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. Springer, 1993. ISBN 978-3-540-56670-0.

[3] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. https://doi.org/10.1109/MCSE.2007.55.

[4] A. Iserles. *A First Course in the Numerical Analysis of Differential Equations*, 2nd ed. Cambridge University Press, 2009. ISBN 978-0-521-73490-5.

[5] T. Kluyver et al. Jupyter Notebooks — a publishing format for reproducible computational workflows. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, S. 87–90, IOS Press, 2016. https://doi.org/10.3233/978-1-61499-649-1-87.

[6] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007. ISBN 978-0-89871-629-0.

[7] W. McKinney. Data Structures for Statistical Computing in Python. In: *Proceedings of the 9th Python in Science Conference*, S. 56–61, 2010. https://doi.org/10.25080/Majora-92bf1922-00a.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007. ISBN 978-0-521-88068-8.

[9] G. van Rossum, B. Warsaw, N. Coghlan. PEP 8 — Style Guide for Python Code. Python Enhancement Proposals, 2001. https://peps.python.org/pep-0008/.

[10] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–263, 2020. https://doi.org/10.1038/s41592-019-0686-2.

[11] L. N. Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*. Unpublished textbook, freely available online, 1996. https://people.maths.ox.ac.uk/trefethen/pdetext.html.

[12] G. Wilson et al. Best Practices for Scientific Computing. *PLOS Biology*, 12(1):e1001745, 2014. https://doi.org/10.1371/journal.pbio.1001745.