Ingenieurmathematik (i939)

Jens Flemming (HTW Dresden)

13.11.2026

Inhaltsverzeichnis

0 Vorwort . . . 1
1 Motivation . . . 2
1.1 Mathematik vs. Rechnen . . . 2
1.2 Mathematik als Sprache . . . 3
1.3 Computer . . . 3
1.4 “KI” . . . 3
1.5 Mittel gegen Komplexität . . . 4
1.6 “Soft-Skills” . . . 5
1.7 Mathematik im Studium . . . 6
1.8 Beispiel: Perovskit-Solarzellen . . . 7
2 Computereinsatz . . . 11
2.1 Überblick . . . 12
2.2 Python . . . 17
2.3 SymPy . . . 25
2.4 NumPy . . . 29
3 Grundlagen der numerischen Mathematik . . . 35
3.1 Beispiele . . . 36
3.2 Zahlen . . . 39
3.3 Fehler . . . 44
3.4 Numerische Differentiation . . . 58
3.5 Numerische Integration . . . 64
3.6 Nichtlineare Gleichungen . . . 69
3.7 Nichtlineare Gleichungssysteme . . . 78
3.8 Lineare Gleichungssysteme . . . 88
4 Modellierung . . . 101
4.1 Wiederholung Differentialrechnung . . . 102
4.2 Wiederholung Integralrechnung . . . 107
4.3 Brachistochrone . . . 111
4.4 Computertomografie . . . 114
4.5 Zerfalls- und Wachstumsprozesse . . . 117
4.6 Pendel . . . 118
4.7 Ölteppich . . . 120
4.8 Schwingende Saite . . . 121
5 Fourier-Analysis . . . 122
5.1 Überblick und Grundlagen . . . 123
5.2 Aperiodische Funktionen . . . 128

5.3 Periodische Funktionen . . . 133
5.4 Diskrete Signale . . . 136
6 Gewöhnliche Differentialgleichungen . . . 145
6.1 Grundlagen . . . 146
6.2 Existenz und Eindeutigkeit von Lösungen . . . 151
6.3 Systeme von Differentialgleichungen . . . 154
6.4 Grafisches Lösen . . . 159
6.5 Euler-Verfahren . . . 165
6.6 Runge-Kutta-Verfahren . . . 172
6.7 Software . . . 176
7 Partielle Differentialgleichungen . . . 179
7.1 Überblick . . . 180
7.2 Finite-Differenzen-Verfahren . . . 188
7.3 Fallstudie: 2D-Kontinuitätsgleichung . . . 193
7.4 Fallstudie: 1D-Wellengleichung . . . 214
7.5 Schwache Formulierung von PDEs . . . 223
7.6 Finite-Elemente-Verfahren . . . 234
8 Übung 1 (Python und Jupyter) . . . 238
8.1 Jupyter . . . 238
8.2 Python . . . 238
9 Übung 2 (NumPy) . . . 240
9.1 Arrays erzeugen . . . 240
9.2 Arrays verändern . . . 240
9.3 Lineare Algebra . . . 241
10 Übung 3 (Zahlen) . . . 242
10.1 Gleitkommazahlen . . . 242
10.2 Patriot-Beispiel . . . 243
11 Übung 4 (Fehler) . . . 245
11.1 Dezimale Gleitkommazahlen . . . 245
11.2 Approximation von π . . . 245
12 Übung 5 (numerische Differentiation) . . . 247
12.1 Daten lesen und prüfen . . . 247
12.2 Preprocessing . . . 248
12.3 Numerische Differentiation . . . 249
13 Übung 6 (nichtlineare Gleichungen) . . . 251
13.1 Kovergenz des Newton-Verfahrens . . . 251
13.2 Nichtlineare Gleichungen mit SciPy . . . 251
14 Übung 7 . . . 253
14.1 Kondition einer Matrix . . . 253
14.2 Lösungsverfahren . . . 253
15 Übung 8 (Differential- und Integralrechnung) . . . 255
15.1 Aufgabe 8.1 . . . 255
15.2 Aufgabe 8.2 . . . 255
15.3 Aufgabe 8.3 . . . 255

15.4 Aufgabe 8.4 . . . 255
15.5 Aufgabe 8.5 . . . 256
15.6 Aufgabe Ü8.6 . . . 256
15.7 Aufgabe Ü8.7 . . . 256
15.8 Aufgabe Ü8.8 . . . 256
15.9 Aufgabe Ü8.9 (optional) . . . 257
16 Übung 9 (Fourier-Transformation) . . . 258
16.1 Glockenkurve . . . 258
16.2 Sägezahnkurve . . . 258
17 Übung 10 (gewöhnliche Differentialgleichungen) . . . 260
17.1 Aufgabe 10.1 . . . 260
17.2 Aufgabe 10.2 . . . 260
17.3 Aufgabe 10.3 . . . 260
17.4 Aufgabe 10.4 . . . 260
18 Übung 11 (Systeme von Differentialgleichungen) . . . 261
18.1 Aufgabe 11.1 (KTA) . . . 261
18.2 Aufgabe 11.2 . . . 261
18.3 Aufgabe 11.3 . . . 261
18.4 Aufgabe 11.4 (optional) . . . 261
19 Übung 12 (PDE, FDM) . . . 262
19.1 Grundlagen . . . 262
19.2 FDM für Poisson-Gleichung . . . 262
20 Übung 13 (PDE, FEM) . . . 265
20.1 Aufgabe 13.1 (KTA) . . . 265
20.2 Aufgabe 13.2 (KTA) . . . 265
20.3 Aufgabe 13.3 . . . 265
21 Download . . . 266
22 Organisatorisches . . . 267
22.1 Ablauf der Veranstaltung . . . 267
22.2 Übung . . . 267
22.3 Prüfung . . . 267
22.4 Hinweise zum Skript . . . 267
22.5 Änderungshistorie . . . 268

0 Vorwort

Dieses Vorlesungsskript ist die Arbeitsgrundlage für die Veranstaltung “Ingenieurmathematik” (Modul i939) an der HTW Dresden ab Wintersemester 2026. Hinweise zum Ablauf der Veranstaltung, insbesondere zum Umgang mit diesem Skript, finden Sie unter Organisatorisches.

Technischer Unterbau ist MyST Markdown mit einem eigenen, auf dem Standard-Book-Theme aufbauenden Theme. Das MyST-Projekt ist noch recht jung und etliche Features sind im Beta-Stadium. Hier und da ist also mit Ecken und Kanten zu rechnen. Fragen und Anregungen zur Technik gern an Jens Flemming.

1 Motivation

Wozu heute noch Mathematik? Das macht doch alles der Computer oder "die KI".

1.1 Mathematik vs. Rechnen

Mathematik wird oft mit Rechnen verwechselt oder gleichgesetzt. Richtig ist: **Mathematik ist die Kunst das Rechnen zu vermeiden!** Mathematik strebt einerseits nach möglichst einfachen Lösungswegen für praktische Aufgabenstellungen und organisiert andererseits die trotzdem oft notwendige Rechenarbeit heute so, dass sie **mit dem Computer** erledigt werden kann.

Manuelles Rechnen ist für Ingenieure heute kaum mehr relevant. Lediglich für den Aufbau eines gewissen Grundverständnisses zu einem zu lösenden Problem kann es sinnvoll sein, einige einfache Spezialfälle per Hand auszurechnen. Alles, was einigermaßen praxisnah ist, ist zu komplex zum manuellen Rechnen und wird dem Computer überlassen.

Beispiel

Theoretisch kann man die Strömungsverhältnisse um ein fliegendes Flugzeug manuell durch Lösen von Differentialgleichungen klären. Schon aus der Tatsache, dass dazu die Form der Flugzeugoberfläche als mathematische Formel gegeben sein müsste, zeigt, dass dieser Gedanke praktisch nicht umsetzbar ist. Vielmehr wird die Flugzeugoberfläche nur als komplexes CAD-Modell bestehend aus Splines, Bézier-Kurven, NURBS usw. im Computer existieren, sodass auch die Strömungssimulation nur einer Computerrechnung zugänglich ist.

Dennoch kann es sinnvoll sein, die Strömungsverhältnisse um einfache Körper wie Kugel oder Kegel durch manuelle Rechnung zu erhellen. Denn die Ergebnisse manueller Rechnung

- sind deutlich vertrauenswürdiger als bei Computerrechnung (mehr dazu im Laufe der Veranstaltung),
- ermöglichen aufgrund ihrer oft einfachen Form ein allgemeineres Verständnis der Lösungseigenschaften des betrachteten Strömungsproblems,
- dienen als wichtige Vergleichsgrundlage bei der Bewertung der Qualität von computerbasierten Lösungen.

Aus dem Beispiel wird klar: Manuelles Rechnen müssen nur noch wenige Ingenieure beherrschen (Grundlagenarbeit, Software-Entwicklung,..). **Solide Fähigkeiten beim Lösen von Aufgabenstellungen mit dem Computer sind hingegen zwingend für jeden Ingenieur.**

1.2 Mathematik als Sprache

Die Mathematik dient ganz wesentlich als Sprache in Technik, Naturwissenschaften und Wirtschaftswissenschaften; und das in zweierlei Hinsicht:

1. Mit den Mitteln der Mathematik werden **Modelle der realen Welt** formuliert. Solche mathematischen Modelle ermöglichen einerseits eine genaue Untersuchung des betrachteten Problems. Andererseits ist die mathematische Formulierung die Grundlage für den Einsatz von ebenfalls mit mathematischen Mitteln formulierten Lösungsverfahren.
2. Soll eine Aufgabenstellung mit dem Computer gelöst werden, so muss die Aufgabe in eine **für Computer verständliche Form** gebracht werden. Computer verstehen nur ganz klare und präzise Anweisungen, sodass man üblicherweise die mathematische Sprache als gemeinsames Ausdrucksmittel von Mensch und Computer wählt (z.B. in Software-Dokumentationen).

1.3 Computer

Landläufig herrscht die Meinung vor, dass Computer immer richtig rechnen. Das stimmt leider nicht, wie wir im Laufe der Veranstaltung immer wieder sehen werden. Wichtige Fähigkeit von Ingenieuren ist heute zu **erkennen, in welchen Situationen der Computer vertrauenswürdige Ergebnisse liefert** und wann nicht! Diese Fähigkeiten zu stärken ist ein wesentliches Ziel dieser Veranstaltung.

Computer sind grundsätzlich als “dumm” zu betrachten. Sie tun exakt das, was man ihnen sagt (nicht, was man meint ihnen gesagt zu haben). Ob der Computer das Richtige tut und ob wir den Ergebnissen trauen können, liegt allein in unserer Hand. Der Mensch muss die Prozesse und Hintergründe verstehen, muss einordnen und bewerten. **Der Computer übernimmt lediglich das Rechnen, nicht das Denken!**

1.4 “KI”

Der Begriff “künstliche Intelligenz” ist irreführend. Tatsächlich handelt es sich bei den heute unter diesem Begriff verstandenen Techniken um **simulierte Intelligenz**. Diese System können **nicht denken, keine zielgerichtete Kreativität** entfalten. Sie interpolieren und (in weit geringerem Maß) extrapolieren lediglich vorhandenes Faktenwissen. Aus dieser Feststellung folgen zwei wichtige Schlüsse für das Dasein als Ingenieur:

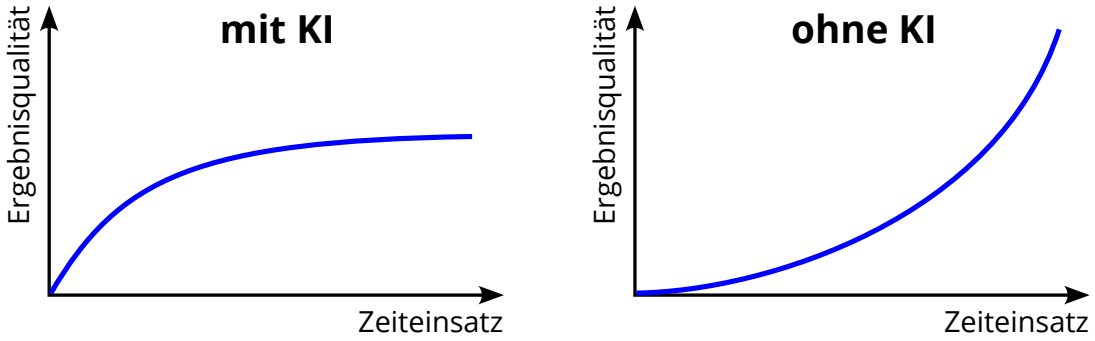
1. Wer nur Standardaufgaben lösen muss/möchte, kommt mit KI oft ans Ziel. Wer wirklich Neues schaffen möchte, muss denken und verstehen.
2. Von KI gelieferte Problemlösungen können beliebig grobe oder auch sehr subtile Fehler enthalten. Gerade im oft sicherheitskritischen Ingenieurbereich müssen aber nachweisbar richtige Lösungen her. Mindestens müssen die Lösungen nach bestem

Wissen und Gewissen korrekt sein; ein Anspruch, den KI-basierte Lösungen nicht erfüllen.

Wichtig: KI arbeitet nicht deterministisch, also nicht Schritt für Schritt nachvollziehbar, sondern statistisch, indem sie eine mit hoher Wahrscheinlichkeit passende Antwort "würfelt". Die Wahrscheinlichkeiten basieren dabei auf der der KI beim Training zugänglichen Wissensbasis. Insbesondere Nischenthemen werden durch Mainstream-Themen überstrahlt. KI-Antworten dazu gleichen einer zufälligen, meist sinnlosen oder ganz falschen Vermischung des eigentlichen Themas mit den Mainstream-Themen im Umfeld. Dabei sind die Antworten sauber und überzeugend formuliert, halten einer inhaltlichen Detailprüfung jedoch nicht stand.

Zur Ideenfindung ist KI durchaus geeignet. Die Abschließende Problemlösung muss jedoch vom Ingenieur vollständig durchdacht und verstanden sein.

Der Einsatz von KI liefert schnelle Erfolge, die langfristig aber hinter den durch Denken und Verstehen erzielbaren Resultaten zurückbleiben.



KI liefert schnell akzeptable Ergebnisse auf mittelmäßigem Niveau. Selbst Denken und Verstehen benötigt mehr Zeit, liefert aber langfristig deutlich hochwertigere Resultate.

1.5 Mittel gegen Komplexität

Neben der direkten Funktion als Sprache dient Mathematik auch dem Strukturieren von Denkprozessen. Die Mathematik schafft vereinfachte Formulierungen der realen Welt und abstrahiert komplexe Zusammenhänge und Vorgänge in klar definierten Begriffen und Strukturen. Durch die Gewöhnung an abstrakte Begriffe (als Ersatz für die viel zu komplexe Realität) werden Denkprozesse vereinfacht oder überhaupt erst möglich.

Beispiel (Reelle Zahlen)

Reelle Zahlen wie $\sqrt{2}$ oder die Kreiszahl π erscheinen uns nach jahrelangem Training in der Schule ganz natürlich und wir arbeiten täglich mit ihnen. In der Rea-

lität haben sie aber im Gegensatz zu rationalen Zahlen keine Entsprechung; im Gegenteil, sie sind ein sehr abstraktes Konzept, welches die Formulierung vieler technischer Zusammenhänge deutlich vereinfacht (IDV 130).

Abstraktion findet auch außerhalb der Mathematik statt, ist aber in der Mathematik zentrales Arbeitsmittel mit Wirkung in andere Wissenschaftsgebiete hinein.

Beispiel (Farben)

Der Begriff “Farbe” ist so alltäglich wie abstrakt. Was bedeutet “die Kirsche ist rot”? Aus mathematischer Sicht ist “rot” eine Äquivalenzklasse bezüglich der Äquivalenzrelation “reflektiert Licht im selben Teil des Spektrums”.



1.6 “Soft-Skills”

Beschäftigt man sich mit Mathematik, erlernt man ganz nebenbei viele Fähigkeiten, die im Privaten wie Beruflichen sehr nützlich sind:

- genaues Analysieren,
- streng logisches Argumentieren,
- exakte Ausdrucksweise,
- scheinbar offensichtliches immer wieder hinterfragen,
- das eigene Denken und Verstehen im Grenzbereich testen.

Das Üben dieser Fähigkeiten in der klaren, strukturierten Welt der Mathematik rüstet für den Weg in diffusere Gebiete (Philosophie, Politik,...).

Beispiel (Logisches Argumentieren)

Hase und Kröte liefern sich ein Wettrennen über 100 Meter. Der Hase sieht die Sache entspannt und lässt die Schildkröte 10 Meter weiter vorn starten. Beide laufen gleichzeitig los. Wer wird gewinnen?

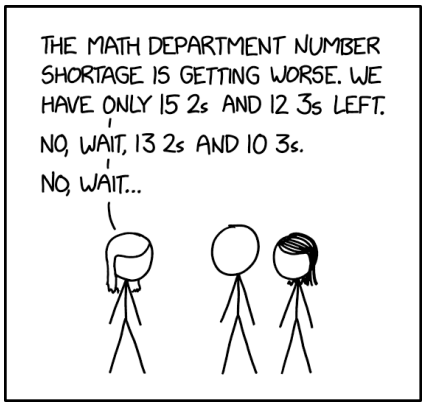
Dazu zwei Argumentationen:

1. Der Hase wird gewinnen, da er viel schneller läuft als die Schildkröte.
2. Die Schildkröte wird gewinnen. Wenn der Hase bei der Startposition der Schildkröte ankommt, ist diese bereits ein Stück gelaufen, liegt also vorn. Kommt der Hase dann wiederum an dieser Position an, ist die Schildkröte wieder ein Stück weiter. Jedesmal, wenn der Hase an der vorherigen Position der Schildkröte ankommt, ist die Schildkröte schon wieder ein Stück weiter. Die Schildkröte liegt immer vorn, wird also gewinnen.

Wer hat recht? Wo ist der Fehler in der anderen Argumentation? (IDV 140)

Beispiel (Scheinbar offensichtliches hinterfragen)

Was heißt eigentlich "gleich viel"? Hat ein gesunder Mensch genauso viele Finger wie Fußzehen? Gibt es gleich viele gerade wie ungerade Zahlen? Gibt es gleich viele Brüche wie es natürliche Zahlen gibt? (IDV 145)

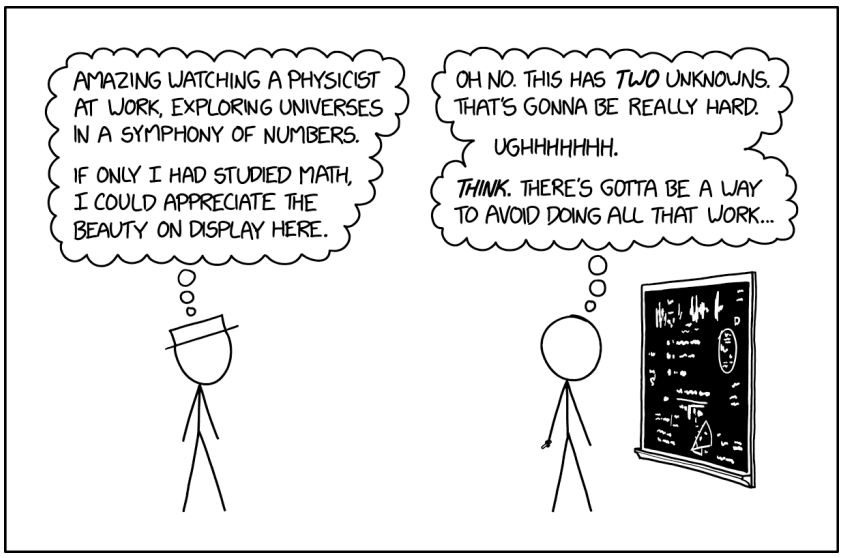


"10 minutes ago we were down to only 2 0s!" "How many do we have now?" "I ... don't know!!" Quelle: Randall Munroe, xkcd.com/3009

1.7 Mathematik im Studium

Im weiteren Studienverlauf wird die Mathematik in vielen Modulen eine tragende Rolle spielen. Einige Module aus dem Wahlpflichtbereich:

- Angewandte Programmierung und KI (Optimierung, Statistik, Programmiergrundlagen).
- Strömungssimulation/Strömungslehre (Differentialgleichungen).
- Konstruktionsbegleitende Simulation (Differentialgleichungen, Fourier-Transformationen).
- Digitale Messtechnik: Bildverarbeitung und Koordinatenmesstechnik (Statistik, Fourier-Transformationen),
- FEM-Simulation Fertigungsverfahren (Differentialgleichungen),
- Mechanismentechnik: Mehrkörpersimulation (Differentialgleichungen).



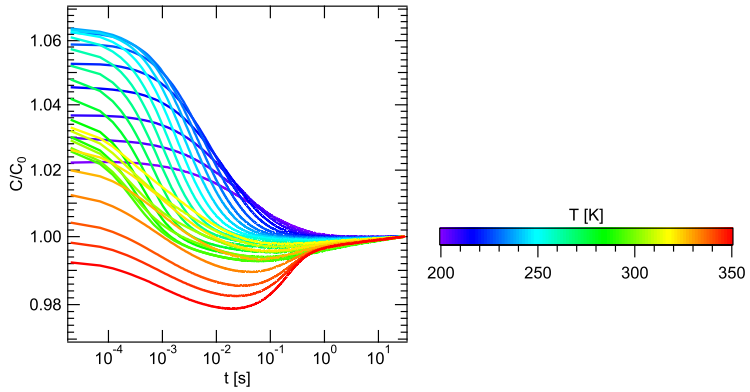
I could type this into a solver, which MIGHT help, but would also mean I have to get a lot of parentheses right... Quelle: Randall Munroe, xkcd.com/2207

1.8 Beispiel: Perovskit-Solarzellen

Neben Silizium werden zunehmend alternative Materialien für Solarzellen untersucht und teils auch schon eingesetzt. Dazu gehört das recht preiswerte Material Perowskit. Durch noch nicht hinreichend präzise kontrollierbare Produktionsprozesse kann es zu Materialfehlern kommen, welche den Wirkungsgrad der Solarzellen verringern. Deshalb ist man an experimentellen Verfahren interessiert, die Einblick in die Art der Materialfehler geben, um anschließend die Produktionsprozesse gezielt zu verbessern. Ein solches Verfahren ist Laplace-DLTS (Laplace deep-level transient spectroscopy).

Die für Laplace-DLTS verwendeten experimentellen Messergebnisse sind ohne weitere algorithmische Auswertung nicht verwertbar. Vereinfacht dargestellt: Sogenannte Elektronenfallen im Material (Materialfehler) werden mit Ladungen gefüllt und anschließend beobachtet man das relativ langsame Abfließen der Ladungen aus den Fallen. Dieser

Vorgang wird bei verschiedenen Temperaturen wiederholt. Es entsteht eine Folge von sogenannten Transienten.



Gemessene Transienten $t \mapsto C(t)$ zu verschiedenen Temperaturen T. Die Transienten wurden mit dem Faktor $\frac{1}{C_0}$ skaliert.

Jede gemessene Transiente entsteht durch Überlagerung der Transienten mehrerer Elektronenfallen im Material. Da das Abklingverhalten einer Einzeltransiente bekannt ist (Exponentialfunktion), kann die Überlagerung rückgängig gemacht werden. Man erhält als Modell für die Entstehung der Transienten die Laplace-Transformation

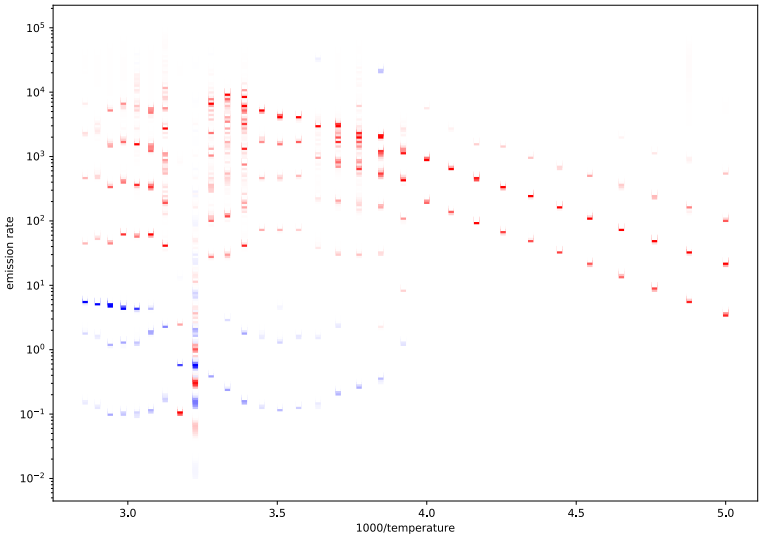
$$C(t) = \int_0^\infty g(s)\,\mathrm{e}^{-s\,t}\,\mathrm{d}s. \tag{1.1}$$

Dies ist eine lineare Integralgleichung. Dabei ist C die in Abhängigkeit der Zeit t gemessene Gesamttransiente zu einer festen Temperatur und g ist die Verteilung der Abklinggeschwindigkeiten in der Menge aller Einzeltransienten. Typischerweise wird g aus wenigen sehr scharfen Peaks bestehen. Aus Position, Höhe und Breite der Peaks kann auf Art und Anzahl der Materialfehler geschlossen werden. Trägt man die Peak-Positionen bei verschiedenen Temperturen in ein Diagramm ein, so erhält man einen sogenannten Arrhenius-Plot. Dieser zeigt typischerweise abfallende Geraden, aus deren Lage und Neigung auf weitere Materialeigenschaften geschlossen werden kann.

Theorie:

1. Messwerte aufnehmen (mit Unterstützung eines Physikers).
2. Laplace-Transformation für jede Transiente C invertieren; liefert jeweils eine Funktion g (Standardaufgabe).
3. Peak-Positionen in g bestimmen und (entsprechend Temperatur) in Arrhenius-Plot eintragen.
4. Position und Anstieg von Geraden im Arrhenius-Plot bestimmen (Standardaufgabe).

Theoretisch ist das alles mit Standardwerkzeugen (Software) machbar.



Arrhenius-Plot zu zwei verschiedenen Messreihen (blau und rot). Die Anordnung der Punkte entlang von Geraden ist gut erkennbar Die Farbintensität der Punkte gibt die Höhe der entsprechenden Peaks in der berechneten Funktion g an.

Praxis:

- Die einzige am Markt verfügbare Software für genau dieses Messproblem ist Jahrzehnte alt und teuer. Wird trotzdem beschafft...
- Die Software kann nur 30000 Messwerte pro Transiente verarbeiten; gemessen werden 300000. Also 90 Prozent der Messwerte wegwerfen...
- Die Software liefert nur "Garbage". Lange Fehlersuche im Messprozess bis klar wird, dass die Software das Problem sein muss.
- Zu lösendes mathematisches Problem verstehen und Software selber bauen. Problem stellt sich als schierig lösbar heraus.
- Hilfe holen (Mathematiker).

Weg zum Ziel:

1. Aufgabenstellung verstehen (gemeinsame Sprache nötig!).
2. Beschaffte Software verstehen (Handbuch lesen; enthält viele "Formeln"):
   - Software geht von zweimal differenzierbaren Funktionen g mit kleinen Werten der zweiten Ableitung aus; wir erwarten aber scharfe Peaks.
   - Software kann nur mit $g \geq 0$ umgehen; bei uns können auch negative Werte auftreten.
3. Beschaffte Software definitv untauglich; eigene Entwicklung nötig.

4. Gemeinsam klären, was eigentlich gesucht wird:
   - g oder
   - Peak-Positionen in g oder
   - Geraden im Arrhenius-Plot?
5. Diskussion der technischen Details:
   - Brauchen wir wirklich 300000 Messwerte? Nein, können aber nicht beliebige Messwerte wegwerfen, sondern nur nach einem gewissen Schema.
   - In welcher Form liegen die Messwerte vor?
   - Was muss die Software können (GUI, Hardware-Anbindung,...)?
6. Mathematiker baut den Berechnungsteil der Software; Physiker baut den Teil für die Verarbeitung der Messwerte. Ingenieur behält den Gesamtprozess im Auge.
7. Wenigstens grob verstehen, was da gerechnet wird.

Was lernen wir daraus?

- Immer ganz genau hinschauen, was der Computer/die Software eigentlich tut.
- Standardverfahren sind nur selten direkt einsetzbar. Meistens muss man Anpassungen vornehmen, gelegentlich Neues entwickeln.
- Problemstellung genau formulieren ("gegeben, gesucht").
- Ingenieur braucht Kenntnisse in angrenzenden Gebieten (Physik, Mathematik, Informatik), damit er die zugelieferten Einzelteile überblickt und zusammenfügen kann.
- "Das macht der Computer" ist schnell gesagt, in der Umsetzung aber alles andere als einfach.

2 Computereinsatz

Grundlagen der Computernutzung in der Ingenieurmathematik.

2.1 Überblick

Der Einsatz des Computers zum Lösen ingenieurmathematischer Aufgabenstellungen kann auf unterschiedliche Weise erfolgen.

2.1.1 Modellierung und Diskretisierung

Allen Herangehensweisen gemeinsam ist, dass das zu lösende Problem zunächst manuell (also ohne Computer) präzise in der Sprache der Mathematik formuliert werden muss, damit es überhaupt der Bearbeitung mit dem Computer zugänglich wird. Am Ende dieses Prozesses steht meist eine Standardaufgabe wie

- Optimierungsproblem,
- Integralgleichung,
- Differentialgleichung,
- statistisches Problem (Parameterschätzung, Hypothesentest,...).

Die meisten aus technischen und physikalischen Betrachtungen abgeleiteten mathematischen Modelle sind kontinuierlich, verwenden also lückenlose Zeit-, Orts- und andere Wertangaben. Mit solchen Modellen kann der Computer nicht umgehen. Man sagt, die **Modelle müssen erst diskretisiert werden**. Beispielsweise müssen zeitliche Verläufe auf die Betrachtung endlich vieler Zeitpunkte reduziert werden.

Beispiel (Momentangeschwindigkeit)

Zu festen Zeitpunkten $t_1, \ldots, t_n$ wird der bis zum jeweiligen Zeitpunkt zurückgelegte Weg $s_1, \ldots, s_n$ gemessen. Gesucht sind die Momentangeschwindigkeiten $v_1, \ldots, v_n$ zu jedem Messzeitpunkt.

Nebenbemerkung

Die **Durchschnittsgeschwindigkeit** bei Durchfahren des Streckenabschnitts zwischen zwei Zeitpunkten lässt sich direkt aus den Messwerten berechnen:

$$\frac{\text{zurückgelegte Strecke}}{\text{benötigte Zeit}}. \tag{2.1}$$

Die **Momentangeschwindigkeit** hingegen ist ein theoretisches Hilfsmittel und kann nicht aus (endlich vielen) Messwerten exakt berechnet werden. Gibt $s : [0, T] \to \mathbb{R}$ die Position des Fahrzeugs zu jedem Zeitpunkt im Zeitraum $[0, T]$ an, so versteht man unter der Momentangeschwindigkeit zur Zeit t den Wert

$$s'(t), \tag{2.2}$$

also den Anstieg des Graphen von s im Punkt $(t, s(t))$.

Ein mögliches Modell zur Berechnung der Momentangeschwindigkeit ergibt sich direkt aus der Definition:

- Eingabe: Weg-Zeit-Zusammenhang $t \mapsto s(t)$ für ein Zeitintervall $[a, b]$.
- Ausgabe: Momentangeschwindigkeit $v(t)$ zur Zeit t für alle $t \in (a, b)$.
- Zusammenhang: $v(t) = s'(t)$ für alle $t \in (a, b)$.

Für den praktischen Einsatz des Modells werden wir dieses diskretisieren müssen. Die Eingabefunktion s werden wir durch eine auf den Messwerten $s_1, \ldots, s_n$ basierende Näherung ersetzen müssen. Die Ausgabe besteht laut Problemstellung ebenfalls nur aus endlich vielen Werten $v_1, \ldots, v_n$. Noch zu klären ist, wie s' auf Basis endlich vieler Messwerte (wenigstens näherungsweise) ausgewertet werden kann (Diskretisierung des Modells).

Nebenbemerkung

Dass das Modell kontinuierlich ist, also auf Funktionen basiert, mag übertrieben aufwendig erscheinen, da ja klar ist, dass nur endlich viele Messwerte zur Verfügung stehen. Der Begriff der Momentangeschwindigkeit lässt aber kein diskretes Modell zu. Für die Formulierung des Modells müssen wir also auf Funktionen zurückgreifen. Wie gut die (theoretischen) Ergebnisse des Modells sich dann praktisch aus endlich vielen Messungen annähern lassen, muss im Rahmen der Diskretisierung geklärt werden.

Wir werden sehen, dass es für die Diskretisierung viele verschiedene Möglichkeiten gibt. Der direkte Vergleich dieser Möglichkeiten ist oft komplizierter als der Vergleich mit einem kontinuierlichen "Referenzmodell", sodass man auch in Zeiten von computerbedingt diskreter Arbeitsweise an kontinuierlichen Modellen interessiert ist.

2.1.2 Symbolisches vs. numerisches Rechnen

Für das Ausführen von Berechnungen mit dem Computer gibt es zwei grundlegend verschiedene Herangehensweisen:

- Beim **symbolischen Rechnen** werden mathematische Formeln durch den Computer in der gleichen Weise umgeformt, kombiniert und gelöst wie man es beim manuellen Rechnen tun würde. Dadurch erhält man exakte Ergebnisse, die wiederum als Formeln vorliegen. Bei der verwendeten Software spricht man auch von **Computer-Algebra-Systemen (CAS)**.
- Beim **numerischen Rechnen** werden nur Grundrechenarten verwendet. An vielen Stellen des Rechenweges werden gerundete oder nur näherungsweise ermittelte Zwischenergebnisse verwendet. Als Endergebnis stehen nur Zahlen zur Verfügung, bei denen nicht mehr erkennbar ist, wie diese entstanden sind (z.B. 1.41421356 statt $\sqrt{2}$).

Im technischen und wissenschaftlichen Bereich wird praktisch nur numerisch gerechnet. Gründe:

- Numerische Berechnungen sind viele schneller, da Computer für diese Art von Berechnungen gebaut sind. Üblich sind heute mehrere Milliarden Grundrechenoperationen pro Sekunde.
- Symbolisches Rechnen liefert bei komplexen Aufgabenstellungen meist keine Lösung, da die Lösungen nicht explizit als Formeln darstellbar sind.

Gelegentlich verwendet man CAS für kleine, klar abgegrenzte Teilschritte, bei denen das Vorliegen einer expliziten Formel in den Folgeschritten (Geschwindigkeits-)Vorteile

bringt.

Beispiel (Automatisches Differenzieren)

Grundlage von "KI" sind künstliche neuronale Netze. Das sind aus Tausenden oder Millionen einfacher Grundfunktionen zusammengesetzte nichtlineare Funktionen, welche Millionen durch ein Optimierungsverfahren zu bestimmende Parameter enthalten. Für die Durchführung der Optimierung werden die partiellen Ableitungen des neuronalen Netzes bezüglich der Parameter benötigt. Diese können gut symbolisch berechnet werden, was Näherungsfehler vermeidet. Außerdem beschleunigt das konkrete Vorliegen einer Formel die Auswertung der Ableitungen während der Optimierung. Verfahrensbedingt müssen diese Ableitungen millionenfach ausgewertet werden.

Das symbolische Berechnen der Ableitungen von Funktionen, die aus einfachen Grundfunktionen zusammengesetzt sind, nennt man auch **automatisches Differenzieren**. Die eigentliche Optimierung erfolgt dann numerisch, da das Problem aufgrund seiner Komplexität nicht symbolisch lösbar ist.

2.1.3 Programmcode vs. grafische Oberfläche

Für praktisch alle Standardaufgaben im Ingenieurwesen gibt es Software mit grafischer Benutzeroberfläche (kurz: GUI für "graphical user interface"). Diese haben jedoch einige Nachteile:

- Möglich ist nur, was die GUI vorsieht. Erweiterungen oder Änderungen sind für Nutzer nur schwer oder gar nicht umsetzbar.
- Abläufe lassen sich schlecht automatisieren, da diese aus Folgen von Mausklicks bestehen.
- Für Aufgabenstellungen, die erstmalig oder nur selten auftreten, existiert keine Software mit GUI (oder ist extrem teuer).

Aus diesen Gründen ist der Einsatz einer geeigneten (!) Programmiersprache auch im Ingenieurbereich für zahlreiche Aufgaben deutlich vorteilhafter. Insbesondere

- sind die Anwendungsmöglichkeiten unbeschränkt,
- lassen sich alle Abläufe ohne große Anpassungen automatisieren,
- kann bei Bedarf im Nachgang eine GUI hinzugefügt werden.

Hinzu kommt, dass sich Programmcode einfacher versionieren lässt und gemeinsames Bearbeiten (auch in Echtzeit) problemlos möglich ist.

Üblich im Ingenieurbereich sind:

- Python/Jupyter (wird in dieser Veranstaltung verwendet),

- Octave oder Matlab (wird zunehmend durch Python verdrängt).

Für die Arbeit mit GUI (z.B. bei FEM-Simulationen) kann beispielsweise FreeCAD verwendet werden.

Nebenbemerkung

Die Arbeit mit Programmcode statt grafischer Oberfläche ist keineswegs komplizierter. Lediglich der Funktionsumfang ist größer. Aufgrund der Vielzahl von Möglichkeiten ist es nicht zweckmäßig für jede Funktion einen Button oder einen Menüeintrag zu bieten. Stattdessen werden alle Aktionen durch Textbefehle ausgelöst.

2.2 Python

In dieser Veranstaltung nutzen wir die Programmiersprache Python um dem Computer seine Aufgaben mitzuteilen. Werkzeuge aus dem Jupyter-Ökosystem ermöglichen die Python-Programmierung in einer sehr komfortablen grafischen Umgebung.

2.2.1 Warum Python?

Python ist eine moderne, kostenfreie und quelloffene ("open source") Programmiersprache. Sie entstand in den 1990er Jahren. Die erste offizielle Version wurde 1994 veröffentlicht. Autor und BDFL ("benevolent dictator for life") ist Guido van Rossum.

Python-Code ist sehr gut lesbar und enthält im Gegensatz zu den meisten anderen Programmiersprachen nur wenige "kryptische" Zeichen und Zeichenketten. Viele technische Aspekte des Programmierens werden von Python erledigt ohne den Nutzer damit zu konfrontieren. Mit Python kann man Software aus allen Bereichen entwickeln; von einfachen Skripten für die tägliche Arbeit bis zu vollwertigen Desktop- oder Webanwendungen. Tausende Erweiterungen ermöglichen zügiges und effizientes Arbeiten.

Als freie Software verfügt Python über eine riesige Online-Community, die mit Rat und Tat zur Seite steht. Kaum eine Frage, die nicht schon irgendwo beantwortet wurde.

Python ist für alle Plattformen verfügbar: Linux, macOS, Windows und viele andere. Python steht beispielsweise hinter großen kommerziellen Projekten wie dem Youtube-Player, läuft eventuell aber auch in der Steuerung Ihrer Waschmaschine.

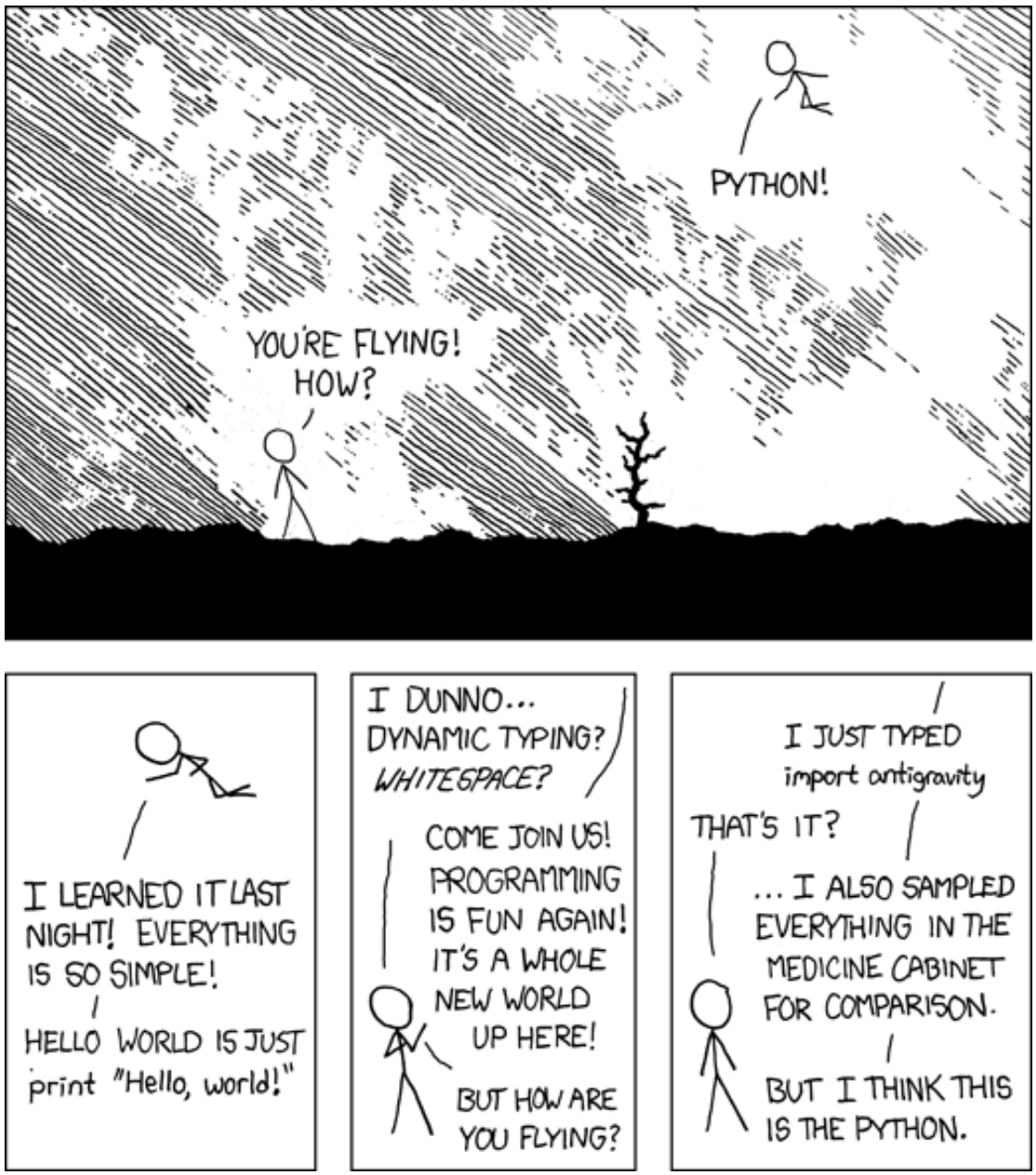
Nebenbemerkung

Es gibt zwei Varianten von Python: Python 2 und Python 3. Entsprechender Quellcode ist nicht kompatibel, d.h. für Python 2 geschriebene Programm funktionieren nicht mit Python 3. Wir verwenden hier ausschließlich Python 3. Python 2 wird von den Entwicklern seit Januar 2020 als veraltet eingestuft und nicht mehr weiterentwickelt.

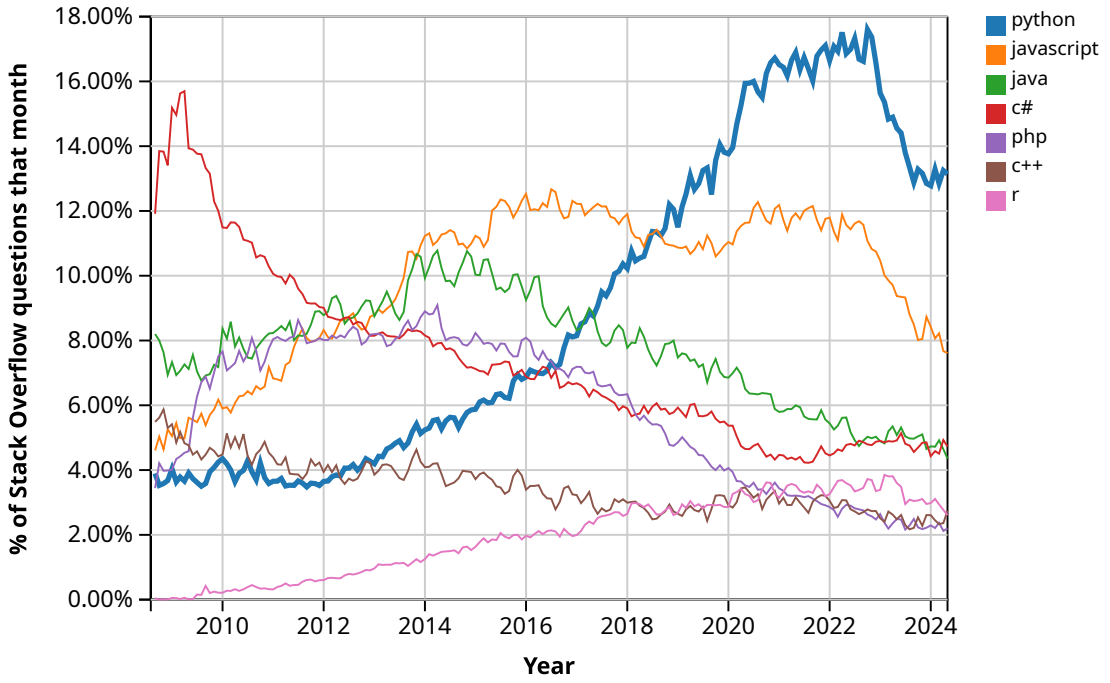
2.2.2 Jupyter

Das Jupyter-Ökosystem ist eine Sammlung von Werkzeugen für die Python-Programmierung mit dem Schwerpunkt auf Nutzerfreundlichkeit und leichte Bedienbarkeit für Anfänger. Jupyter erlaubt das Programmieren mit Python im Webbrowser. Ausgaben, wie beispielsweise komplexe und interaktive Visualisierungen, können direkt zwischen dem zugehörigen Quellcode platziert werden. Alles befindet sich in einem einzigen Dokument: Code, Ausgaben, erklärender Text, Bilder, Formeln,...

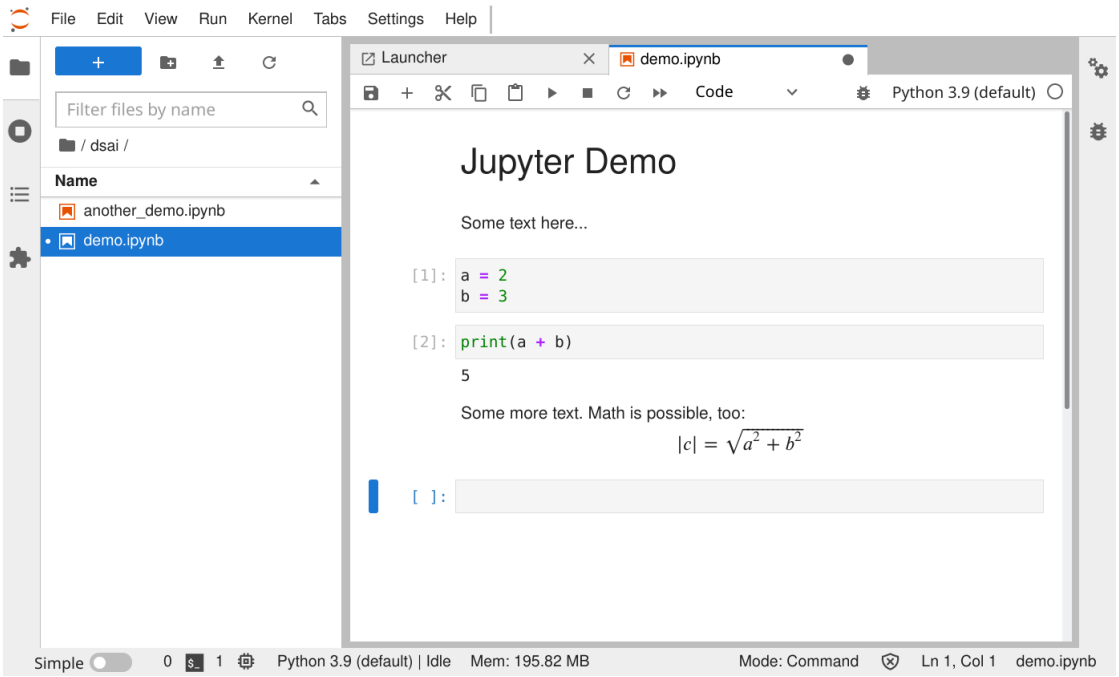
JupyterLab steht in vielen Formen zur Verfügung (vgl. Übung):



I wrote 20 short programs in Python yesterday. It was wonderful. Perl, I'm leaving you. Source: Randall Munroe, xkcd.com/353



Beliebtheit von Programmiersprachen auf Stack Overflow. Quelle: Stack Overflow Trends (vom Autor bearbeitet)



JupyterLab ist das am weitesten verbreitete Mitglied des Jupyter-Ökosystems. Es bringt Python in den Webbrowser.

- als rein browser-basierte Anwendung ohne Installation (Funktionsumfang geringfügig eingeschränkt),
- als lokale Installation auf dem eigenen Computer,
- als Installation auf Servern der Hochschule,
- auf Servern anderer Organisationen.

Welche Variante genutzt wird, ist für die Veranstaltung nebensächlich. Außerdem kann Python auch ganz ohne Jupyter genutzt werden.

2.2.3 Python-Crashkurs

Erwähnen hier als ersten Einstieg in Python beispielhaft einige Grundfunktionen. Weitere Möglichkeiten folgen im Laufe der Veranstaltung nach Bedarf. Für das weiterführende Selbststudium sind das Kapitel Python Crash Course sowie die darauf folgenden Kapitel in Data Science and Artificial Intelligence for Undergraduates geeignet.

2.2.3.1 Taschenrechner

```
# Punkt vor Strich
1 + 2 * 3
```

```
7
```

```
# Potenzen
2 ** 8
```

```
256
```

```
# Wurzeln
2.5 ** 0.5
```

```
1.5811388300841898
```

```
# Division
5 / 2
```

```
2.5
```

```
# ganzzahlige Division
5 // 2
```

```
2
```

```
# Rest bei ganzzahliger Division
5 % 2
```

```
1
```

2.2.3.2 Bessere Ausgaben

```
# einfacher Text
print('irgendein Text')
```

```
irgendein Text
```

```
# Text mit Kommazahl
print(f'Wurzel aus 2 ist {2.0 ** 0.5}.')
```

```
Wurzel aus 2 ist 1.4142135623730951.
```

Mehr Infos zu solchen Ausgaben: f-String und Minisprache für Stringformatierung.

```
# Text mit gerundeter Kommazahl
print(f'Wurzel aus 2 ist {2.0 ** 0.5:.3f}.')
```

```
Wurzel aus 2 ist 1.414.
```

2.2.3.3 Variablen

```
zahl = 2.0
wurzel = zahl ** 0.5
print(f'Wurzel aus {zahl} ist {wurzel:.3f}.')
```

```
Wurzel aus 2.0 ist 1.414.
```

Es gibt verschiedene Variablentypen (Ganzzahl, Kommazahl, Zeichenkette,...).

```
print(type(2))
print(type(2.0))
print(type('Hallo!'))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Umwandlungen sind zwischen geeigneten Typen möglich.

```
zeichenkette1 = '123'
zeichenkette2 = '456'
print(zeichenkette1 + zeichenkette2)

zahl1 = int(zeichenkette1)
zahl2 = int(zeichenkette2)
print(f'und als Zahlen: {zahl1 + zahl2}')
```

```
123456
und als Zahlen: 579
```

2.2.3.4 Listen Listen sind Folgen von Variablen, deren Name einem einheitlichen Schema folgt (Listenname plus Index).

```
liste = ['rot', 'gelb', 'grün', 'blau']
print(liste[0])
print(liste[1])
print(liste[2])
print(liste[3])
print(liste[ -1])
print(liste[1:3])
```

```
rot
gelb
grün
blau
blau
['gelb', 'grün']
```

2.2.3.5 Bedingte Ausführung

```
a = 3
b = 2  # zwei Zahlen (Nutzereingaben oder aus Datei gelesen)

if b != 0:
    print(f'{a} / {b} = {a / b}')
else:
    print('Division durch Null!')
```

```
3 / 2 = 1.5
```

Wichtig: Test auf Gleichheit mit ==. Das einfache = ist der Zuweisungsoperator und kann nicht für Vergleiche verwendet werden.

2.2.3.6 Wiederholte Ausführung

```
liste = ['rot', 'gelb', 'grün', 'blau']

for farbe in liste:
    print(farbe)
```

```
rot
gelb
grün
blau
```

```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

2.2.3.7 Module und Pakete Die Funktionalität von Python kann durch Module und Pakete erweitert werden. Pakete sind Sammlungen von Modulen. Praktisch besteht bei der Verwendung kein Unterschied zwischen Modulen und Paketen.

```
import math

math.cos(math.pi / 4)
```

```
0.7071067811865476
```

```
import math as m

m.cos(m.pi / 4)
```

```
0.7071067811865476
```

```
from math import cos, pi
```

```
cos(pi / 4)
```

```
0.7071067811865476
```

Python verfügt nur über wenige eingebaute Grundfunktionen. Man wird fast immer zusätzliche Module und Pakete benötigen, die ggf. erst installiert werden müssen.

2.3 SymPy

Auch wenn wir in der Veranstaltung kaum symbolisch rechnen werden, sei hier eine kleine Einführung in SymPy gegeben. SymPy ist ein Python-Paket für symbolisches Rechnen (Ableitungen, Stammfunktionen, Formeln vereinfachen, Gleichungen lösen,...). Siehe auch Introductory Tutorial in der SymPy-Dokumentation.

```
import sympy
```

2.3.1 Variablen

SymPy verwendet ganz normale Python-Variablen, die aber einen SymPy-spezifischen Typ haben. Variablen, die später mit SymPy verwendet werden sollen, müssen deshalb zunächst durch SymPy erstellt werden.

```
x = sympy.symbols('x')  # zeige Python -Variable x in Formeln als x an

print(x)
print(type(x))
```

```
x
<class 'sympy.core.symbol.Symbol'>
```

Mehrere Variablen können gemeinsam erstellt werden:

```
x, y, z = sympy.symbols('x y z')

2 * x + y ** 2 + z
```

```
2*x + y**2 + z
```

2.3.2 Formeln vereinfachen

```
x = sympy.symbols('x')

a = (x + 1) ** 2
b = x ** 2 + 2 * x + 1

sympy.simplify(a - b)
```

```
0
```

```
x = sympy.symbols('x')

sympy.simplify(sympy.cos(x) ** 2 + sympy.sin(x) ** 2)
```

```
1
```

2.3.3 Grenzwerte

```
x = sympy.symbols('x')

sympy.limit(sympy.sin(x) / x, x, 0)
```

```
1
```

```
n = sympy.symbols('n')

sympy.limit((1 + 1 / n) ** n, n, sympy.oo)
```

```
E
```

2.3.4 Ableitungen

```
x, c = sympy.symbols('x c')

sympy.diff(x ** 2 * sympy.sin(x + c), x)
```

```
x**2*cos(c + x) + 2*x*sin(c + x)
```

2.3.5 Integrale

```
x = sympy.symbols('x')

sympy.integrate(x ** 2 * sympy.sin(x), x)
```

```
-x**2*cos(x) + 2*x*sin(x) + 2*cos(x)
```

```
x = sympy.symbols('x')

sympy.integrate(x ** 2 * sympy.sin(x), (x, 0, sympy.pi))
```

```
-4 + pi**2
```

2.3.6 Nullstellen von Polynomen

SymPy liefert Polynomnullstellen samt Vielfachheiten:

```
x = sympy.symbols('x')

sympy.roots(x ** 3 - 4 * x ** 2 + x + 6, x)
```

```
{3: 1, 2: 1, -1: 1}
```

```
x = sympy.symbols('x')

sympy.roots(x ** 3 - 3 * x ** 2 + 3 * x - 1, x)
```

```
{1: 3}
```

2.3.7 Gleichungen lösen

```
x = sympy.symbols('x')

sympy.solve(x ** 2 - 2, x)  # löst x ** 2 - 2 = 0
```

```
[ -sqrt(2), sqrt(2)]
```

Das Ergebnis ist eine Liste (kein Intervall!) von Lösungen.

```
x = sympy.symbols('x')

formel = x ** 4 - 4 * x ** 3 + 7 * x ** 2 - 16 * x + 12
sympy.solve(formel, x)
```

```
[1, 3, -2*I, 2*I]
```

```
x = sympy.symbols('x', real=True)

formel = x ** 4 - 4 * x ** 3 + 7 * x ** 2 - 16 * x + 12
sympy.solve(formel, x)
```

```
[1, 3]
```

```
x = sympy.symbols('x')

sympy.solve(sympy.sin(x), x)
```

```
[0, pi]
```

```
x = sympy.symbols('x')

sympy.solveset(sympy.sin(x), x)
```

```
Union(ImageSet(Lambda(_n, 2*_n*pi), Integers), ImageSet(Lambda(_n,
↪  2*_n*pi + pi), Integers))
```

2.3.8 Numerisches Rechnen mit vielen Stellen

```
sympy.N('1 / 6', 50)
```

```
0.16666666666666666666666666666666666666666666666667
```

```
sympy.N('pi', 50)
```

```
3.1415926535897932384626433832795028841971693993751
```

2.4 NumPy

NumPy ist ein Paket, welches Python um Funktionen für das schnelle Rechnen mit großen Zahlenmengen (Matrizen, Vektoren) erweitert. NumPy ist im Python-Ökosystem der De-Facto-Standard für numerische Berechnungen wie beispielsweise das Lösen großer linearer oder nichtlinearer Gleichungssysteme.

```
import numpy as np
```

Grundlage für die Arbeit mit NumPy sind die NumPy-Arrays, welche zur Darstellung von Vektoren und Matrizen dienen.

2.4.1 NumPy-Arrays erstellen

```
np.array([4, 5, 2, 3])  # aus einer Liste
```

```
array([4, 5, 2, 3])
```

```
np.zeros(4)  # eindimensionales Array mit Nullen
```

```
array([0., 0., 0., 0.])
```

```
np.ones((3, 2))  # zweidimensionales Array mit Einsen
```

```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

```
np.eye(5)  # Einheitsmatrix
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

2.4.2 Zugriff auf Array-Elemente

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
print(a[0, 0])  # 1. Zeile, 1. Spalte
print(a[1, 0])  # 2. Zeile, 1. Spalte
print(a[ -1, -1])  # Element unten rechts
```

```
1
4
9
```

```
a[1:3, :2]
```

```
array([[4, 5],
       [7, 8]])
```

```
a[0, :]  # 1. Zeile
```

```
array([1, 2, 3])
```

2.4.3 Eigenschaften von Arrays

```
a.shape  # Größe (Zeilen, Spalten)
```

```
(3, 3)
```

```
a.dtype  # Datentyp der Einträge
```

```
dtype('int64')
```

```
b = np.array([4.1, 6, 3, 5, 4, 3])
print(b.shape)
print(b.dtype)
```

```
(6,)
float64
```

2.4.4 Rechnen mit Arrays

```
a = np.array([4.1, 6, 3])
b = np.array([5, 4, 3])

2 * a + b
```

```
array([13.2, 16. ,  9. ])
```

```
a * b
```

```
array([20.5, 24. ,  9. ])
```

```
A = np.array([[1, 2], [3, 4]])
x = np.array([[5], [6]])  # Spaltenvektor!

print(A)
print()
print(x)
print()
print(A * x)  # elementweise Multiplikation (vorher "Broadcasting")
print()
print(A @ x)  # Matrix -Vektor -Multiplikation
```

```
[[1 2]
 [3 4]]

[[5]
 [6]]

[[ 5 10]
 [18 24]]

[[17]
 [39]]
```

```
print(A * A)  # elementweises Produkt
print()
print(A @ A)  # Matrix -Produkt
```

```
[[ 1  4]
 [ 9 16]]

[[ 7 10]
```

```
 [15 22]]
```

```
A.T  # transponierte Matrix
```

```
array([[1, 3],
       [2, 4]])
```

```
np.sin(A)  # Sinus elementweise
```

```
array([[ 0.84147098,  0.90929743],
       [ 0.14112001, -0.7568025 ]])
```

2.4.5 Arrays vergleichen

```
a = np.array([1, 2, 3])
b = np.array([ -1, 3, 3])

print(a > b)
```

```
[ True False False]
```

```
c = a > b

print(c.any())
print(c.all())
```

```
True
False
```

2.4.6 Weitere Möglichkeiten für den Elementzugriff

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
a[a > 3]
```

```
array([4, 5, 6, 7, 8, 9])
```

```
a[np.logical_and(a > 3, a < 6)]
```

```
array([4, 5])
```

```
a[a > 4] = 20

a
```

```
array([[ 1,  2,  3],
       [ 4, 20, 20],
       [20, 20, 20]])
```

2.4.7 Lineare Algebra

```
a = np.array([1, 2, 3])
b = np.array([1, 0, 2])

print(np.inner(a, b))  # Skalarprodukt
print(np.cross(a, b))  # Kreuzprodukt
```

```
7
[ 4  1 -2]
```

```
A = np.array([[1, 2, 3], [4, 5, 5], [7, 7, 9]])

print(np.linalg.det(A))  # Determinante
print(np.linalg.inv(A))  # Inverse
```

```
-12.999999999999995
[[ -0.76923077 -0.23076923  0.38461538]
 [ 0.07692308  0.92307692 -0.53846154]
 [ 0.53846154 -0.53846154  0.23076923]]
```

```
A = np.array([[2, 0],
              [1, 1]])
b = np.array([2, 3])

print(np.linalg.solve(A, b))  # LGS lösen
```

```
[1. 2.]
```

2.4.8 Zufallszahlen

Für Tests und Simulationen sind Zufallszahlen sehr nützlich. Um Zufallszahlen zu erzeugen, muss zunächst der Zufallszahlengenerator von NumPy mit einem Startwert initialisiert werden. Der Startwert legt die Folge der später generierten Zufallszahlen fest, sodass bei späterer Wiederholung von Rechnungen sehr leicht wieder die gleichen Zufallszahlen verwendet werden können.

```
rng = np.random.default_rng(123)  # wie der Startwert gewählt wird, ist
↪  egal
```

```
rng.integers(23, 42, (4, 10))
```

```
array([[23, 35, 34, 24, 40, 27, 27, 26, 29, 26],
       [29, 38, 31, 40, 31, 28, 37, 38, 39, 39],
       [23, 32, 28, 27, 27, 38, 38, 27, 30, 37],
       [25, 34, 31, 40, 37, 27, 38, 38, 27, 32]])
```

```
rng.uniform(0, 2, (4, 4))  # Gleichverteilung auf [0, 2]
```

```
array([[0.46311125, 0.33180799, 0.99557794, 1.16544928],
       [0.36867597, 0.02978983, 0.94226646, 1.45648666],
       [1.83720098, 1.25106801, 1.83424515, 1.7293805 ],
       [0.43628575, 1.73225486, 1.46150387, 0.55573058]])
```

```
rng.normal(0, 1, (4, 4))  # Standardnormalverteilung
```

```
array([[ 0.85988118,  1.76166124,  0.99332378, -0.29152143],
       [ 0.72812756, -1.26160032,  1.42993853, -0.15647532],
       [ -0.67375915, -0.6390601 , -0.06136133, -0.39278492],
       [ 2.28990995, -0.71818115,  0.03260774,  0.0280499 ]])
```

```

```

3 Grundlagen der numerischen Mathematik

Einführung in die numerische Mathematik.

3.1 Beispiele

Was kann da schon schiefgehen?

3.1.1 Groß plus Klein

Klar:

$$1 + x^{20} - x^{20} = 1,$$

z.B. $x = 10.5$.

Taschenrechner, Python, andere Programmiersprachen: 0.

```
1 + 10.5 ** 20 - 10.5 ** 20
```

```
0.0
```

Alternativ:

```
1 + (10.5 ** 20 - 10.5 ** 20)
```

```
1.0
```

Merke

Numerische Addition mit dem Computer ist nicht assoziativ!

Addiert man eine sehr kleine Zahl zu einer sehr großen Zahl, wird die kleine Zahl unter Umständen ignoriert. Man spricht hier auch von **Absorption**.

3.1.2 Ungenaue Zahlendarstellung

Nicht jede Zahl kann im Computer exakt dargestellt werden, was zu unerwarteten Effekten führen kann:

```
a = 0.1 * 0.1 - 0.01  # sollte theoretisch 0 sein
b = 10 ** 20

a * b
```

```
173.4723475976807
```

3.1.3 Patriot-Raketenabwehr

Siehe GAO/IMTEC-92-26 Patriot Missile Software Problem.

Kurzfassung:

1. Zweiter Golfkrieg ("Operation Desert Storm"). US-Amerikanische Militärbasis in Dharan (Saudi-Arabien) wird durch Patriot-Raketenabwehrsystem vor Scud-Raketen aus dem Irak geschützt. Vorher für dieses Szenario nicht wirklich getestet (Steuersoftware aus den 70er-Jahren).
2. Gelegentlich werden Raketen aus ungeklärten Gründen nicht abgefangen.
3. Jemand schaut mal nach, wo es klemmt.
4. Fehler gefunden: **Rundungsfehler verstärkt sich** im Laufe der Zeit durch ungeschickte Implementierung einer Multiplikation mit 0.1, sodass nach hinreichend langer Up-Time das System nicht mehr präzise genug arbeitet.
5. Anweisung an die Nutzer: System täglich neu starten.
6. Software-Update wird am 16. Februar 1991 verfügbar, impliziert aber 2 Stunden Down-Time. Probleme beim Verteilen usw.
7. Am 25. Februar 1991 sterben 28 US-Soldaten durch eine Scud-Rakete.
8. Jemand spielt das Update ein am 26. Februar 1991.

Details zum Rundungsfehler: IDV 310.

3.1.4 Simulation von Schwingungen

Bei der Simulation einer schwingenden Saite kann die Auslenkung der Saite zu diskreten Zeitpunkten berechnet werden. Bei gegebenem Zeitintervall $[0, T]$ hängt die Brauchbarkeit der Simulation wesentlich von der Anzahl der Zeitschritte ab. Sind die Schritte zu groß, so ist die Simulation nicht mehr stabil (später mehr zu diesem Begriff).

Hier die Simulationen für 200 und 198 Zeitschritte:

```
200 Zeitschritte
```


0

[ ] Once [x] Loop [ ] Reflect

```
198 Zeitschritte
```


0

[ ] Once [x] Loop [ ] Reflect

Merke

Wesentliche Aufgabe bei der Simulation technischer Prozess ist die **Klärung der Zuverlässigkeit** der mittels Simulation erhaltenen Resultate!

Ursachen für unbrauchbare Simulationsergebnisse können sein:

- Das gewählte Modell ist ungeeignet.
- Das gewählte Lösungsverfahren ist ungeeignet.
- Die Umsetzung des Lösungsverfahrens am Computer ist fehlerhaft.
- Die Umsetzung am Computer ist theoretisch korrekt, aber es treten numerische Effekte auf, die nicht ausreichend bedacht wurden.

3.2 Zahlen

Numerische Berechnungen verwenden praktisch immer gebrochene Zahlen ("Kommazahlen"). Zwei Varianten für die Darstellung von gebrochenen Zahlen durch Bitfolgen:

- **Festkommazahlen**
  - begrenzte Anzahl Ziffern vor dem Komma,
  - begrenzte Anzahl Ziffern hinter dem Komma,
- **Gleitkommazahlen**
  - begrenzte Gesamtanzahl Ziffern,
  - flexible Position des Kommas (auch weit vor oder hinter der Ziffernfolge).

Festkommazahlen spielen heute nur in Spezialanwendungen eine Rolle, siehe z.B. GNU-Cash und STM32G4 CORDIC co-processor. Auch werden Festkommazahlen von gängigen Programmiersprachen nur in Ausnahmefällen unterstützt (ein Beispiel ist GNU C). Beschäftigen uns hier deshalb nur mit Gleitkommazahlen genauer.

3.2.1 Darstellung von Gleitkommazahlen

Merke

Gleitkommazahlen (engl.: *floating-point numbers*, kurz: *floats*) können durch zwei Ganzzahlen beschrieben werden:

- **Mantisse** $m \in \mathbb{Z}$,
- **Exponent** $e \in \mathbb{Z}$.

Zusammen mit der gewählten Basis $b \in \{2, 10\}$ (Binär- oder Dezimalsystem) ergeben diese die Zahl

$$m\, b^e, \tag{3.1}$$

wobei e die Position des Kommas angibt. Bei $e = 0$ steht das Komma direkt hinter der Ziffernfolge. Positive e ergeben größere Zahlen (Komma weiter rechts), negative e kleinere (Komma weiter links).

Ist die Höchstanzahl der Mantissenziffern (ohne Vorzeichen) festgelegt auf $t \in \mathbb{N}$, so gibt es mehrere Paare (m, e), die die selbe Zahl darstellen. Beispiel für $b = 10$, $t = 4$:

$$123 \cdot 10^1 = 1230 \cdot 10^0. \tag{3.2}$$

Man wählt üblicherweise die Darstellung mit der längsten möglichen Mantisse, d.h. die Mantisse hat stets genau t Ziffern und die erste Ziffer ist nicht Null. Eine so dargestellte Gleitkommazahl heißt **normalisierte Gleitkommazahl**. Diese Darstellung ist

eindeutig, für Null aber nicht möglich. Wenn nicht anders angegeben, meinen wir mit "Gleitkommazahl" im Folgenden stets eine normalisierte Gleitkommazahl.

Die Ziffernanzahl im Exponent spielt in der numerischen Mathematik nur selten eine Rolle. Die Mantissenlänge ist hingegen ausschlaggebend für die Genauigkeit von Berechnungen. Wenn wir im Folgenden Aussagen der Form "für alle reellen Zahlen gilt..." formulieren, meinen wir damit stets nur solche reellen Zahlen, die betragsmäßig kleiner als die größte darstellbare Gleitkommazahl sind, also keinen **Exponentenüberlauf** verursachen.

3.2.2 Datenformat für binäre Gleitkommazahlen

Für die Umwandlung von Mantisse und Exponent einer binären Gleitkommazahl zu einer Bitfolge spielt heute hauptsächlich der Standard IEEE 754 eine Rolle. Siehe Floating-point arithmetic für Ausnahmen.

Benötigte Bitfolgen für eine normalisierte Gleitkommazahl:

- Vorzeichen S der Mantisse:
  - 1 Bit,
  - 0 oder 1 für positive bzw. negative Mantisse.
- Betrag M der Mantisse ohne erste Ziffer:
  - $t-1$ Bit,
  - die erste Ziffer ist immer 1, muss also nicht gespeichert werden ("hidden bit").
- transformierter Exponent E:
  - p Bit,
  - $E = e + t - 1 + 2^{p-1} - 1$,
  - $E = 0$ und $E = 2^p - 1$ markieren spezielle Gleitkommazahlen (siehe unten).

Der Zusammenhang zwischen E und e entsteht dabei aus der Forderung, dass links und rechts der zweiten Mantissenziffer gleich viele Kommapositionen darstellbar sein sollen:

- Der Summand $t-1$ verschiebt das Komma vom Ende der Mantisse zwischen die ersten beiden Ziffern.
- Der Summand $2^{p-1} - 1$ sorgt für die gleiche Anzahl Kommapositionen links und rechts.

Die durch Vorzeichenbit S, Mantissenbetrag M (ohne erste Ziffer) und transformierten Exponent E beschriebene Zahl ist also

$$(-1)^S \left(2^{t-1} + M\right) \cdot 2^{E-(2^{p-1}-1)-(t-1)} \tag{3.3}$$

Übliche Werte für die Bitanzahl von Mantisse und Exponent bei binären Gleitkommazahlen

Bezeichnung	$t-1$	p	Wertebereich von $e+t-1$	dezimale Ziffern von m
16-Bit-Gleitkommazahl	10	5	$-14 \ldots 15$	$3 \ldots 4$
32-Bit-Gleitkommazahl	23	8	$-126 \ldots 127$	$7 \ldots 8$
64-Bit-Gleitkommazahl	52	11	$-1022 \ldots 1023$	$15 \ldots 16$

oder auch

$$(-1)^S \cdot 1.M \cdot 2^{E-(2^{p-1}-1)}, \tag{3.4}$$

wobei $1.M$ als $1 + 2^{1-t} M$ zu verstehen ist.

Spezielle Werte:

- $E = 0$ markiert die Zahl 0, wenn $M = 0$; sonst sogenannte denormalisierte Zahlen (behandeln wir hier nicht).
- $E = 2^p - 1$ markiert ∞, wenn $M = 0$; sonst "NaN", also ungültige Zahl.

Ungültige Zahlen entstehen z.B. als Ergebnis der Division 0/0. Unendliche Werte entstehen z.B. bei 1/0 (dann $+\infty$) oder -1/0 (dann $-\infty$). Die Zahl 0 hat dabei ebenfalls ein Vorzeichen. Betragskleine negative Zahlen werden auf -0 gerundet, positive auf +0. Somit kann 1/0 auch $-\infty$ ergeben, wenn die 0 durch Runden einer negativen Zahl entstanden ist.

Nebenbemerkung

In der Literatur wird der Begriff Gleitkommazahl nicht einheitlich verwendet. Gelegentlich sind damit alle in der Form $m\,b^e$ darstellbaren Zahlen gemeint und die mit dem Computer darstellbaren Zahlen (endlicher Bereich für m und e!) werden Maschinenzahlen genannt. Wir verwenden den Begriff Gleitkommazahl hier ausschließlich für mit dem Computer darstellbare Zahlen bei vorgegebener Ziffernanzahl und vorgegebenem Bereich von Kommapositionen.

3.2.3 Runden

Das übliche Rundungsverfahren für Gleitkommazahlen entspricht nicht exakt dem mathematischen Runden.

Merke

Regeln für das Runden von Gleitkommazahlen:

- Runde zur nächst gelegenen darstellbaren Zahl.
- Wenn die zu rundende Zahl genau mittig zwischen zwei darstellbaren Zahlen liegt, runde so, dass die letzte Ziffer der gerundeten Zahl gerade ist.

Beim mathematischen Runden wird in der zweiten Regel immer die größere der beiden möglichen Zahlen gewählt. Dies führt jedoch zur **statistischen Drift**.

Beispiel: Rechnen mit 2 Kommastellen. Mittelwert der 10 Werte 0.105, 0.115, 0.125, 0.135, 0.145, 0.155, 0.165, 0.175, 0.185, 0.195 ist 0.15 bzw. 0.16 je nach verwendeter Rundungsregel (IDV 320).

Merke

Sei $x \in \mathbb{R}$ eine reelle Zahl, sei $m\,b^e$ die betragsmäßig nächst kleinere (normalisierte) Gleitkommazahl und bezeichne $\mathrm{round}(x)$ die durch Runden aus x erhaltene (normalisierte) Gleitkommazahl. Dann gilt für den **absoluten Rundungsfehler**

$$|x - \mathrm{round}(x)| \leq \tfrac{1}{2}\,|(m+1)\,b^e - m\,b^e| = \tfrac{1}{2}\,b^e \tag{3.5}$$

und für den **relativen Rundungsfehler**

$$\frac{|x - \mathrm{round}(x)|}{|x|} \leq \frac{b^e}{2\,|x|} \leq \frac{b^e}{2\,|m|\,b^e} = \frac{1}{2\,|m|} \leq \frac{1}{2}\,b^{1-t}, \tag{3.6}$$

wobei t die Mantissenlänge ist und die letzte Ungleichheit auf der Tatsache beruht, dass m genau t Ziffern hat und die erste Ziffer nicht Null ist.

Die Schranke $\varepsilon := \frac{1}{2}\,b^{1-t}$ für den relativen Rundungsfehler heißt **Maschinenepsilon**. Aus der Dreiecksungleichung folgen

$$(1-\varepsilon)\,|x| \leq |\mathrm{round}(x)| \leq (1+\varepsilon)\,|x| \tag{3.7}$$

und

$$\frac{1}{1+\varepsilon}\,|\mathrm{round}(x)| \leq |x| \leq \frac{1}{1-\varepsilon}\,|\mathrm{round}(x)| \tag{3.8}$$

(IDV 325). Diese Ungleichungen spielen eine wichtige Rolle bei der Fehleranalyse von Algorithmen. Insbesondere existiert zu jedem reellen x ein ϱ mit

$$\mathrm{round}(x) = (1+\varrho)\,x, \qquad |\varrho| \leq \varepsilon. \tag{3.9}$$

3.2.4 Grundoperationen

Einige grundlegende Operationen, die nach IEEE 754 mit korrektem Runden implementiert sein müssen (meist direkt in der Hardware):

- Addition, Subtraktion,
- Multiplikation, Division,
- Quadratwurzel,
- Umwandlung in Ganzzahl,
- Umwandlung zwischen den verschiedenen Bitanzahlen.

Praktisch alle Prozessoren bieten viele weitere Operationen an. Siehe z.B. Intel®64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture (ab Abschnitt 8.3.7). Mathematische Bibliotheken liegen üblicherweise in an die verschiedenen Prozessoren angepassten Versionen vor, sodass möglichst viele Operationen direkt auf der Hardware ausgeführt werden können.

Als Maß für die Rechengeschwindigkeit eines Prozessors werden gern **FLOPS** (floating-point operations per second, auch: flop/s) angegeben. Eine "floating-point operation" ist dabei meist als kombinierte Addition und Multiplikation zu verstehen. Es gibt jedoch keine allgemein anerkannte feste Definition dieser Einheit! Die Umrechnungen von FLOPS in Prozessortakte ist prozessorabhängig; siehe Floating point operations per second für die Umrechnungsfaktoren bei üblichen Prozessoren.

3.3 Fehler

3.3.1 Fehlerarten

Bei numerischen Berechnungen spielen verschiedene Fehlerarten eine Rolle:

- **Rundungsfehler** (Zwischenergebnisse werden auf die darstellbare Genauigkeit gerundet).
- **Datenfehler** (die Eingabewerte entstehen aus (ungenauen) Messungen),
- **Verfahrensfehler** (die eingesetzten Algorithmen liefern auch bei exakter Rechnung ohne Rundungsfehler nur Näherungslösungen).
- **Modellfehler** (vereinfachte/idealisierte Darstellung der tatsächlichen physikalischen oder anderweitigen Gegebenheiten),

Dabei enthält der Datenfehler praktisch immer auch Rundungsfehler, da die Daten als Gleitkommazahlen an den Algorithmus übergeben werden müssen.

Obwohl die numerische Mathematik sich auf die (Vermeidung der) Auswirkungen von Rundungsfehlern konzentriert, spielen auch die anderen Fehlerarten und -quellen eine Rolle. Insbesondere **zielt man gar nicht auf maximale Genauigkeit** (was mit hohem Rechenaufwand verbunden ist), sondern nur darauf, dass die Auswirkungen von Verfahrens- und Rundungsfehlern nicht größer sind als zwangsläufige Abweichungen aus Daten- und Modellfehlern.

3.3.2 Beispiel: Volumen der Erde

Wollen das Volumen der Erde aus gegebenem Radius r (in Meter) mittels Kugelvolumen

$$V = \frac{4}{3}\,\pi\, r^3 \tag{3.10}$$

berechnen.

Algorithmus

1. Teile 4 durch 3.
2. Multipliziere mit π.
3. Multipliziere dreimal mit r.

Auftretende Fehler:

- Modellfehler: Die Erde ist keine exakte Kugel, sondern ein Ellipsoid (auch nicht exakt).
- Verfahrensfehler: Hier nicht vorhanden, sofern (bei exakten Daten) mit exaktem π gerechnet wird.

- Datenfehler: Exakte Bestimmung von r scheitert schon an der Struktur der Erdoberfläche (Berge, Täler). Selbst mit sehr präziser Messtechnik wird man nur ein Intervall $r \in [a, b]$ angeben können. Wählt man $r = \frac{a+b}{2}$, beträgt der Datenfehler bis zu $\frac{b-a}{2}$.
- Rundungsfehler: r und π müssen für die Darstellung mittels Gleitkommazahlen gerundet werden. Die Zahlen 4 und 3 sind exakt darstellbar. Die Auswirkungen der Multiplikationen und der Division auf die Genauigkeit des Ergebnisses werden weiter unten untersucht.

Aufgrund der unvermeidlichen großen Modell- und Datenfehler gibt es keinen Grund die Berechnungen mit sehr großer Genauigkeit auszuführen. Solange die Rundungsfehler und deren Auswirkungen auf die einzelnen Rechenoperationen kleiner als die Modell- und Datenfehler bleiben, ist die Genauigkeit groß genug.

3.3.3 Bewertung von Algorithmen

Gegeben ist ein Algorithmus, der "ein Problem löst". Genauer: Es gibt eine Menge $X \subseteq \mathbb{R}^d$ von möglichen Eingaben, eine Menge $Y \subseteq \mathbb{R}$ von möglichen Ausgaben und eine (exakte) Lösungsabbildung $f : X \to Y$.

Zentrale Frage: **Wie hängt der Ausgabefehler vom Eingabefehler ab?**

Dabei nehmen wir an, dass es eine unbekannte exakte Eingabe $x \in X$ gibt und eine bekannte mess- und rundungsfehlerbehaftete Eingabe $\tilde{x}$. Der **relative Eingabefehler** ist dann

$$\left|\frac{x_1 - \tilde{x}_1}{x_1}\right| + \cdots + \left|\frac{x_d - \tilde{x}_d}{x_d}\right|. \tag{3.11}$$

Die numerische Auswertung von $f(x)$ erfolgt durch einen (potentiell fehlerhaften) Algorithmus $\tilde{f} : X \to Y$, der auf die fehlerbehafteten Daten $\tilde{x}$ angewendet wird. Für den **relativen Ausgabefehler** erhalten wir somit

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|}. \tag{3.12}$$

Nebenbemerkung (Warum relative Fehler und nicht absolute Fehler?)

Eine Aussage der Form "Der Algorithmus verursacht einen absoluten Fehler von 0.001" hat für sich allein keinen relevanten Informationsgehalt. Verarbeitet der Algorithmus sehr große Zahlen, so spricht diese Aussage für den Algorithmus. Verarbeitet er jedoch beispielsweise Zahlen unter 0.000001, so ist der Algorithmus völlig unbrauchbar.

Relative Fehler liefern uns Informationen über die Anzahl der korrekt berechneten Ziffern unabhängig davon, in welchem Zahlenbereich der Algorithmus zum Einsatz

kommt.

Nachteil relativer Fehler: Wir müssen $x = 0$ und $f(x) = 0$ von den Bertrachtungen ausschließen, was wir stets ohne explizite Erwähnung dieser Tatsache tun werden. Wenn dies sachlich nicht gerechtfertigt ist, bleiben nur absolute Fehler als Ausweg.

Ziel einer **Fehleranalyse** sind Abschätzungen der Form

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|} \leq g(\delta), \tag{3.13}$$

wobei

$$\sum_{i=1}^{d} \left| \frac{x_i - \tilde{x}_i}{x_i} \right| \leq \delta \tag{3.14}$$

gelten soll. Die Zahl δ ist also eine obere Schranke an den zu erwartenden Datenfehler. Praktisch wird δ mindestens so groß sein wie das Maschinenepsilon.

Die Funktion g ist meistens ein Monom $g(\delta) = c\,\delta^p$ mit $p > 0$ und einer (hoffentlich kleinen) Konstante $c > 0$. Je größer p, desto unempfindlicher reagiert der untersuchte Algorithmus auf Datenfehler. In jedem Fall muss $\lim_{\delta \to 0} g(\delta) = 0$ gelten, damit die Fehlerabschätzung eine sinnvolle Aussage liefert ("Wenn wir den Datenfehler kleiner machen, wird auch der Lösungsfehler kleiner").

Zentrale Werkzeuge für die Herleitung solcher Abschätzungen sind die **Kondition** der Abbildung f und die **Stabilität** des Algorithmus $\tilde{f}$, welche im Folgenden eingeführt werden.

3.3.4 Kondition

Merke

Die **Kondition** einer Abbildung $f : X \to Y$ beschreibt den Zusammenhang zwischen Eingabefehler und Ausgabefehler bei exakter Auswertung von f. Sie ist eine Eigenschaft des zu lösenden Problems und hängt in keiner Weise vom gewählten Auswertealgorithmus $\tilde{f}$ ab.

Verstärkt f die Eingabefehler, so ist f **schlecht konditioniert** oder hat schlechte Kondition. Haben Eingabe- und Ausgabefehler die gleiche Größenordnung oder sind die Ausgabefehler sogar kleiner als die Eingabefehler, so ist f **gut konditioniert** oder hat gute Kondition.

Einfaches Beispiel: Berechnung des Schnittpunktes zweier Geraden (IDV 330).

- Verlaufen die Geraden nahezu senkrecht zueinander, dann gute Kondition.
- Verlaufen die Geraden nahezu parallel, dann schlechte Kondition.

Die Kondition hängt nicht nur von f selbst, sondern auch von der konkreten Eingabe x ab!

Die Kondition eines Problems bzw. der entsprechenden Abbildung f kann man in Zahlen fassen. Für die konkrete Definition dieser **Konditionszahl** gibt es verschiedene Ansätze. Wir wählen zunächst den folgenden:

Merke

Sei $f : X \to Y$ zweimal stetig differenzierbar an der exakten Eingabe $x \in X$ und sei $\tilde{x} \in X$ eine entsprechende fehlerbehaftete Eingabe. Die Zahl

$$\kappa(x) := \max_{i \in \{1,\ldots,d\}} \left| \frac{\partial f}{\partial x_i}(x) \right| \left| \frac{x_i}{f(x)} \right| \tag{3.15}$$

heißt **Konditionszahl** von f an der Stelle x. Mit ihr gilt

$$\left| \frac{f(\tilde{x}) - f(x)}{f(x)} \right| \lessapprox \kappa(x) \sum_{i=1}^{d} \left| \frac{\tilde{x}_i - x_i}{x_i} \right| \tag{3.16}$$

(siehe IDV 335). Das Symbol $\lessapprox$ ist als “im Wesentlichen kleiner als” zu lesen.

Der hier vorgestellte Konditionsbegriff führt immer zu linearen Fehlerzusammenhängen g in (3.13). Dieser Ansatz ist in der Literatur auch als *differentielle Fehleranalyse* oder *Störungstheorie 1. Ordnung* zu finden.

Schlecht konditionierte Probleme sind in der Praxis (leider) häufig anzutreffen, z.B. das bei der Computertomografie zu lösende mathematische Problem.

3.3.5 Beispiele zur Kondition

3.3.5.1 Multiplikation

Betrachten

$$f : \mathbb{R}^2 \to \mathbb{R}, \qquad f(x_1, x_2) := x_1 \, x_2. \tag{3.17}$$

Für $x_1 \neq 0$ und $x_2 \neq 0$ erhalten wir $\kappa(x) = 1$. Der Ausgabefehler ist also höchstens so groß wie der Eingabefehler.

Für $x_1 = 0$ oder $x_2 = 0$ können wir nur absolute Fehler betrachten:

$$|f(x) - f(\tilde{x})| = |\tilde{x}_1| \, |\tilde{x}_2|. \tag{3.18}$$

Ist z.B. $x_1 = 0$ und x_2 ist sehr groß, also auch $\tilde{x}_2$ sehr groß, so kann das Produkt der gestörten Eingaben erheblich vom tatsächlichen Produkt 0 abweichen. Wähle z.B. $\tilde{x}_1 = 10^{-5}$ und $x_2 = \tilde{x}_2 = 10^6$. Dann ist das berechnete Produkt 10 statt 0. Praktisch ist dieser Fall aber selten von Interesse, da die Null exakt als Gleitkommazahl darstellbar ist. Dennoch ist an dieser Stelle Vorsicht geboten!

```
a = 0.1 * 0.1 - 0.01
b = 10 ** 20
print(a * b)
```

```
173.4723475976807
```

3.3.5.2 Division

Betrachten

$$f : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \to \mathbb{R}, \qquad f(x_1, x_2) := \frac{x_1}{x_2}. \tag{3.19}$$

Für $x_1 \neq 0$ erhalten wir $\kappa(x) = 1$. Der Ausgabefehler ist als höchstens so groß wie der Eingabefehler.

Für $x_1 = 0$ können wir nur absolute Fehler betrachten:

$$|f(x) - f(\tilde{x})| = \frac{|\tilde{x}_1|}{|\tilde{x}_2|}. \tag{3.20}$$

Die daraus zu ziehenden Schlüsse entsprechen denen bei der Multiplikation.

```
a = 0.1 * 0.1 - 0.01
b = 10 ** ( -20)
print(a / b)
```

```
173.4723475976807
```

3.3.5.3 Quadratwurzel

Betrachten

$$f : [0, \infty) \to \mathbb{R}, \qquad f(x) := \sqrt{x}. \tag{3.21}$$

Für $x \neq 0$ erhalten wir $\kappa(x) = \frac{1}{2}$. Der Ausgabefehler ist also kleiner als der Eingabefehler.

Für $x = 0$ erhalten wir den absoluten Fehler

$$|f(x) - f(\tilde{x})| = \sqrt{\tilde{x}}, \tag{3.22}$$

der bei $\tilde{x} < 1$ stets größer ist als der Eingabefehler $\tilde{x}$. Beachte hier, dass Eingabefehler $\tilde{x} < 0$ zu undefiniertem Ergebnis bzw. komplexen Zahlen führen!

```
a = 0.01 - 0.1 * 0.1
print(a ** 0.5)
```

```
(8.0648442379710 58e -26+1.3170890159654386e -09j)
```

```
from numpy import sqrt

a = 0.01 - 0.1 * 0.1
print(sqrt(a))
```

```
nan
```

```
/tmp/ipykernel_1059403/1240944823.py:4: RuntimeWarning: invalid value
↪  encountered in sqrt
  print(sqrt(a))
```

3.3.5.4 Addition und Subtraktion

Betrachten

$$f : \mathbb{R}^2 \to \mathbb{R}, \qquad f(x_1, x_2) := x_1 + x_2. \tag{3.23}$$

Für $x_1 \neq 0$ und $x_2 \neq 0$ erhalten wir

$$\kappa(x) = \frac{\max\{|x_1|, |x_2|\}}{|x_1 + x_2|}. \tag{3.24}$$

Haben x_1 und x_2 gleiches Vorzeichen, dann ist $\kappa(x) \leq 1$. Haben x_1 und x_2 jedoch unterschiedliches Vorzeichen, so kann $|x_1 + x_2|$ klein sein, obwohl weder $|x_1|$ noch $|x_2|$ klein ist. In diesem Fall ist κ sehr groß, die Addition also schlecht konditioniert! Diese Situation wird als **Auslöschung** bezeichnet.

Beispiel: Rechnen mit 5 Dezimalstellen, $x_1 = 0.120587$, $x_2 = -0.120942$. Durch Runden der Eingabewerte entsteht ein Eingabefehler der Größenordnung 10^{-4}. Der Fehler im Ergebnis liegt jedoch bei 10^{-2}, ist also deutlich größer (Details: IDV 340).

Weiteres Beispiel: Berechnen die Differenz $a - b$, wobei statt a und b die auf 64-Bit-Gleitkommazahlen gerundeten Werte verwendet werden.

```
a = 1 + 11e -15
b = 1 + 10e -15
print(f'a = {a:.20f}')
print(f'b = {b:.20f}')
```

```
a = 1.00000000000001110223
b = 1.00000000000000999201
```

Mit exakten Werten erhalten wir $a - b = 10^{-15}$. Der relative Eingabefehler ist kleiner als 10^{-15}, während der reltative Ausgabefehler größer als 0.11 ist (IDV 345). Somit weicht die tatsächliche Differenz um **mehr als 11 Prozent vom berechneten Wert** ab. Beachte hierbei: Wir haben exakt gerechnet, das Ergebnis also nicht gerundet. Ursache

für den großen Fehler ist somit nicht ein konkreter Algorithmus (z.B. addiere exakt, runde dann auf 64-Bit-Gleitkommazahl), sondern das Problem selbst in Kombination mit leicht gestörten Eingabewerten. Berechnen wir die Differenz algorithmisch (also mit zusätzlichem Runden des Ergebnisses), so erhalten wir

```
c = a - b
print(f'a - b = {c}')
print( 'exakt = 1.0000000000000000e -15')
```

```
a - b = 1.1102230246251565e -15
exakt = 1.0000000000000000e -15
```

Nur die Ziffer vor dem Komma ist korrekt!

Nebenbemerkung (Ist Auslöschung in der Praxis ein relevantes Problem?)

Ob Auslöschung wirklich ein Problem ist, hängt vom Kontext ab. Im obigen Beispiel wurden zwei Zahlen subtrahiert, die beide ungefähr 1 waren. Der absolute Fehler im Ergebnis liegt bei ca. 10^{-16}, was im Vergleich zu 1 verschwindend gering ist. Die Rechnung an sich ist also hinreichend genau. Problematisch ist nur der Fakt, das von den ca. 16 Dezimalziffern des Ergebnisses nur die erste etwas mit dem tatsächlichen Wert zu tun hat. Alle weiteren sind falsch und unbrauchbar. Wird der berechnete Wert als Eingabe für weitere Rechnungen verwendet, ist auch bei gut konditionierten Operationen ein Ausgabefehler von der Größenordnung 10 Prozent zu erwarten!

```
print(1 + c)          # Fehler in c nicht relevant
print(c * 10 ** 15)   # Fehler in c relevant
```

```
1.000000000000001
1.1102230246251565
```

3.3.5.5 Sinus

Betrachten

$$f : \mathbb{R} \to [-1, 1], \qquad f(x) := \sin x. \tag{3.25}$$

Für $x \neq k\,\pi$, $k \in \mathbb{Z}$, erhalten wir

$$\kappa(x) = \frac{|x|\,|\cos x|}{|\sin x|}. \tag{3.26}$$

Für x in der Nähe von Vielfachen von π wird die Kondition also beliebig groß, während f z.B. in der Nähe von $\frac{\pi}{2}$ sehr gut konditioniert ist.

```
from math import sin

def print_sin(a, b):

    sina = sin(a)
    sinb = sin(b)

    print(sina)
    print(sinb)

    print(f'Eingabefehler: {abs((a - b) / a)}')
    print(f'Ausgabefehler: {abs((sina - sinb) / sina)}')

print_sin(
    3.141593,
    3.141593123456789
)
```

```
-3.464102066193935e -07
 -4.6986699585547026e -07
Eingabefehler: 3.929751219718377e -08
Ausgabefehler: 0.3563890060887287
```

Alle Mantissenziffern des für 3.141593 aus dem fehlerbehafteten Wert 3.141593123456789 berechneten Ergebnisses sind falsch, nur der Exponent stimmt. Der relative Fehler liegt bei 34 Prozent. Die Konditionszahl ist $\kappa(3.141593) \approx 9068997$. Beachte, dass die absoluten Fehler bei Eingabe und Ausgabe dennoch etwa gleich groß sind, das Ergebnis je nach Kontext also durchaus brauchbar sein kann.

```
print_sin(
    1.570796,
    1.570796123456789
)
```

```
0.9999999999999466
0.9999999999999793
Eingabefehler: 7.8595049270588e -08
Ausgabefehler: 3.2751579226443866e -14
```

Hier stimmen fast alle Mantissenziffern trotz gleichem absoluten Fehler in der Eingabe wie zuvor (der relative Eingabefehler ist hier sogar doppelt so groß). Der relative Ausgabefehler liegt im Bereich 10^{-13}, ist also um Größenordnungen kleiner als der relative Eingabefehler.

3.3.5.6 Quadratische Gleichung Betrachten das Lösen quadratischer Gleichungen $x^2 + p\,x + q = 0$, wobei wir nur die kleinere der beiden möglichen Lösungen berechnen möchten. Haben mit $D(f) := \{(p,q) \in \mathbb{R}^2 : p^2 \geq 4\,q\}$ also

$$f : D(f) \to \mathbb{R}, \qquad f(p,q) := -\frac{p}{2} - \sqrt{\frac{p^2}{4} - q}. \tag{3.27}$$

Für $p \neq 0$, $q \neq 0$ und $f(p,q) \neq 0$ (also $p > 0$ oder $q \neq 0$) kann man

$$\kappa(p,q) = \max\left\{ \frac{\frac{|p|}{2}}{\sqrt{\frac{p^2}{4} - q}}, \frac{\left|-\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}\right|}{2\sqrt{\frac{p^2}{4} - q}} \right\} \tag{3.28}$$

zeigen. Für $q \leq 0$ gilt somit $\kappa(p,q) \leq 1$, also gute Kondition. Für $q > 0$ wächst die Kondition mit q und wird für $q \to \frac{p^2}{4}$ beliebig groß. Das Problem ist für große q also sehr schlecht konditioniert (grafische Interpretation: IDV 350).

3.3.6 Stabilität

Merke

Ein Algorithmus $\tilde{f}$ zur Auswertung einer Abbildung f heißt **stabil**, wenn $\tilde{f}(x) \approx f(x)$ gilt für alle möglichen Eingaben x.

Ein Algorithmus heißt also stabil, wenn er das tut, was man von ihm erwartet. Für die Formulierung präziserer Aussagen über den erwarteten Zusammenhang zwischen $\tilde{f}(x)$ und $f(x)$ gibt es verschiedene Ansätze. Wir gehen hier nur auf die beiden am weitesten verbreitetenen ein: **Vorwärtsstabilität** und **Rückwärtsstabilität**.

Merke

Sei $\delta > 0$ eine obere Schranke an den erwarteten relativen Eingabefehler für das betrachtete Problem f. Ein Algorithmus $\tilde{f}$ zur Auswertung von f heißt **vorwärtsstabil**, wenn

$$\frac{|f(x) - \tilde{f}(x)|}{|f(x)|} \leq c\,\kappa(x)\,\delta \tag{3.29}$$

für alle möglichen Eingaben x und ein nicht zu großes $c \geq 0$ gilt.

Vorwärtsstabilität sichert also, dass der durch den verwendeten Algorithmus verursachte Fehler in der Größenordnung des auch bei exakter Rechnung nicht zu vermeidenden Fehlers liegt.

Für konkrete Algorithmen lassen sich Aussagen zur Vorwärtsstabilität oft nur schwer herleiten. Deshalb verwendet man fast immer den folgenden alternativen Stabilitätsbegriff.

Merke

Sei $\delta > 0$ eine obere Schranke an den erwarteten relativen Eingabefehler für das betrachtete Problem f. Ein Algorithmus $\tilde{f}$ zur Auswertung von f heißt **rückwärtsstabil**, wenn es zu jeder möglichen Eingabe x eine Eingabe $\hat{x}$ gibt, sodass

$$\tilde{f}(x) = f(\hat{x}) \quad \text{und} \quad \sum_{i=1}^{d} \left| \frac{x_i - \hat{x}_i}{x_i} \right| \leq c\,\delta, \tag{3.30}$$

wobei $c \geq 0$ unabhängig von x und nicht zu groß sein soll.

Rückwärtsstabilität sichert also, dass die Ausgabe des Algorithmus der exakten Rechnung mit einer Eingabe entspricht, die im Rahmen der Größenordnung des ohnehin vorhandenen Eingabefehlers ebenfalls in Frage kommt.

Um Rückwärtsstabilität zu zeigen müssen zwei Fragen beantwortet werden:

1. Gibt es zu jeder Eingabe x ein passendes $\hat{x}$?
2. Wie groß ist der Abstand zwischen x und zugehörigem $\hat{x}$ höchstens?

Merke

Man kann zeigen: Aus der Rückwärtsstabilität folgt stets die Vorwärtsstabilität.

Veranschaulichung beider Stabilitätsbegriffe: IDV 355.

3.3.7 Beispiele für Stabilitätsanalysen

3.3.7.1 Addition/Subtraktion Nach IEEE 754 erfolgt die Addition zweier Gleitkommazahlen nach folgendem Algorithmus:

Algorithmus

1. Berechne das exakte Ergebnis.
2. Runde entsprechend Rundungsregel auf eine Gleitkommazahl.

Sind x_1, x_2 die beiden Summanden und ist ε das Maschinenepsilon, so liefert der Algorithmus das Ergebnis

$$(1 + \varrho)\,(x_1 + x_2) \tag{3.31}$$

für ein ϱ mit $|\varrho| \leq \varepsilon$.

3.3.7.1.1 Vorwärtsanalyse Der einfache Algorithmus erlaubt eine Vorwärtsanalyse:

$$\left|\frac{f(x)-\tilde{f}(x)}{f(x)}\right| = \left|\frac{x_1+x_2-(1+\varrho)\,(x_1+x_2)}{x_1+x_2}\right| = |\rho| \leq \varepsilon. \tag{3.32}$$

Die Kondition der Addition erfüllt $\kappa(x) \geq \frac{1}{2}$ (IDV 360). Wir erhalten somit

$$\left|\frac{f(x)-\tilde{f}(x)}{f(x)}\right| \leq 2\,\kappa(x)\,\varepsilon. \tag{3.33}$$

Da der zu erwartende relative Eingabefehler praktisch immer größer als $2\,\varepsilon$ sein wird (Rundung jedes Summanden auf Gleitkommazahl), ist der untersuchte Algorithmus zur Addition als vorwärtsstabil anzusehen.

3.3.7.1.2 Rückwärtsanalyse Zu Demonstrationszwecken untersuchen wir auch die Rückwärtsstabilität. Haben

$$\begin{aligned}\tilde{f}(x_1,x_2) &= (1+\varrho)\,(x_1+x_2)\\ &= (1+\varrho)\,x_1+(1+\varrho)\,x_2\\ &= f\big((1+\varrho)\,x_1,(1+\varrho)\,x_2\big),\end{aligned} \tag{3.34}$$

mit $\hat{x}_1 := (1+\varrho)\,x_1$, $\hat{x}_2 := (1+\varrho)\,x_2$ also $\tilde{f}(x) = f(\hat{x})$. Der relative Fehler beim Ersetzen von x durch $\hat{x}$ ist

$$\begin{aligned}\left|\frac{x_1-\hat{x}_1}{x_1}\right| + \left|\frac{x_2-\hat{x}_2}{x_2}\right| &= \left|\frac{x_1-(1+\varrho)\,x_1}{x_1}\right| + \left|\frac{x_2-(1+\varrho)\,x_2}{x_2}\right|\\ &= 2\,|\varrho| \leq 2\,\varepsilon.\end{aligned} \tag{3.35}$$

Da die zu erwartenden Eingabefehler größer als $2\,\varepsilon$ sein werden, ist der Algorithmus als rückwärtsstabil anzusehen.

3.3.7.2 Lange Summen Die Summe von n Zahlen soll berechnet werden:

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad f(x) := \sum_{i=1}^{n} x_i. \tag{3.36}$$

Betrachten dazu zunächst den einfachsten Algorithmus:

Algorithmus

1. Addiere x_1 und x_2 exakt.
2. Runde auf Gleitkommazahl.

3. Für $i = 3, \ldots, n$:
   a) Addiere x_i exakt.
   b) Runde auf Gleitkommazahl.

Für die Konstante in der Definition der Rückwärtsstabilität kann man $c \sim n$ zeigen.

Betrachten nun folgenden alternativen Algorithmus für Summen mit $n = 2^m$ Summanden:

Algorithmus

1. Für $k = 0, \ldots, m-1$:
   a) Berechne aus den 2^{m-k} vorhandenen Werten die 2^{m-k-1} paarweisen Summen.
   b) Runde jede dieser Summen auf Gleitkommazahlen.

Hier kann man für die Konstante in der Definition der Rückwärtsstabilität $c \sim m$ zeigen. Der Algorithmus kann also auch für große n noch als rückwärtsstabil (und somit auch vorwärtsstabil) angesehen werden, da $m = \log_2 n$ nur sehr langsam mit n wächst.

3.3.7.3 Quadratische Gleichung

Möchte man die Lösungen einer quadratischen Gleichung berechnen, so führt eine Stabilitätsanalyse des Auswertens der üblichen Lösungsformel zu der Erkenntnis, dass folgender Algorithmus die beste Stabilität aufweist:

Algorithmus

1. Multipliziere p mit p, dann runden.
2. Teile durch 4, dann runden.
3. Subtrahiere q, dann runden.
4. Ziehe die Wurzel, dann runden.
5. Teile $-p$ durch 2, dann runden.
6. Wenn $p > 0$, dann subtrahiere die Wurzel aus Schritt 5 von $-\frac{p}{2}$, sonst addiere sie. Dies liefert eine der beiden Lösungen.
7. Dividiere q durch das Ergebnis von Schritt 6. Dies liefert die andere Lösung.

In Schritt 6 wird die Lösung berechnet, die weniger anfällig für Auslöschung ist. In Schritt 7 kommt dann der Satz von Vieta zum Einsatz, der die Beziehung $x_1 \, x_2 = q$ liefert.

Dieser Algorithmus ist stabil, wenn $q \ll \frac{p^2}{4}$.

3.3.8 Gesamtfehler

Faustregel: Gute Kondition des Problems f und Stabilität des Algorithmus $\tilde{f}$ garantieren einen hinnehmbar kleinen Gesamtfehler. Schlechte Kondition oder fehlende Stabilität lassen keine brauchbaren Ergebnisse erwarten.

Konkret kann man folgendes Resultat zeigen:

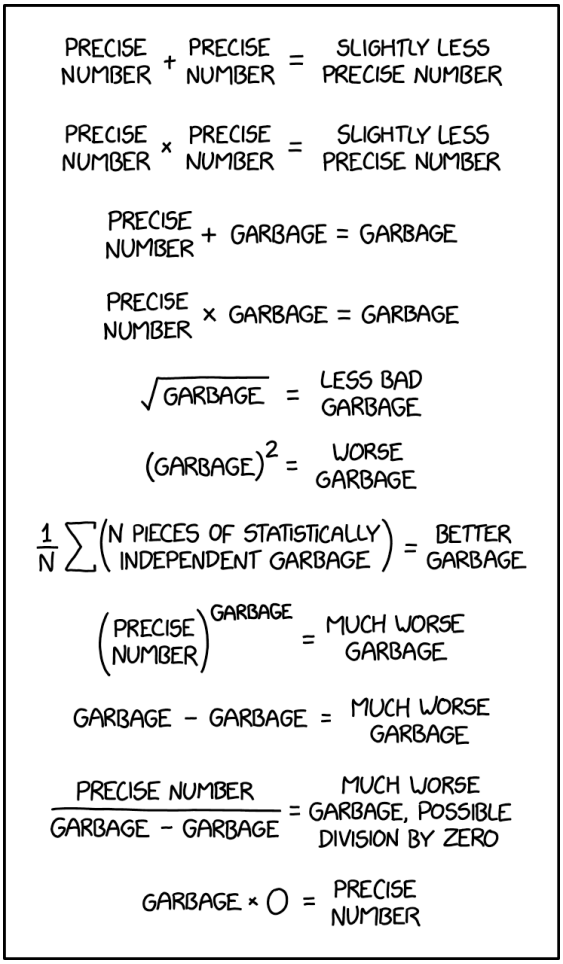
Merke

Sei x die exakte Eingabe zum Problem f mit Kondition κ. Sei $\tilde{x}$ eine gestörte Eingabe, wobei der relative Eingabefehler durch $\delta > 0$ beschränkt sei. Ist der Algorithmus $\tilde{f}$ zur Auswertung von f vorwärts- oder rückwärtsstabil, so gilt für den relativen **Gesamtfehler**

$$\frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|} \leq \big(\kappa(x) + c\,(1 + \kappa(x)\,\delta)\,\kappa(\tilde{x})\big)\,\delta. \tag{3.37}$$

Dabei ist c die Konstante aus der Definition der Vorwärts- bzw. Rückwärtsstabilität.

Die Abschätzung zeigt, dass der Ausgabefehler um so kleiner wird, je kleiner der Eingabefehler ist (wenn $\kappa(\tilde{x})$ bei $\tilde{x} \to x$ beschränkt bleibt).



"Garbage In, Garbage Out" should not be taken to imply any sort of conservation law limiting the amount of garbage produced. Quelle: Randall Munroe, xkcd.com/2295

3.4 Numerische Differentiation

Für eine gegebene differenzierbare Funktion $f : \mathbb{R} \to \mathbb{R}$ soll die Ableitung

$$f'(x_0) := \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{3.38}$$

an einer Stelle $x_0 \in \mathbb{R}$ numerisch berechnet werden.

3.4.1 Idee

Aus der Definition der Ableitung erhält man sofort zwei einfache Möglichkeiten zur näherungsweisen Berechnung von $f'(x)$:

- **Vorwärtsdifferenzenquotient** ($x > x_0$): Zu vorgegebener Schrittweite $h > 0$ berechne

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \tag{3.39}$$

- **Rückwärtsdifferenzenquotient** ($x < x_0$): Zu vorgegebener Schrittweite $h > 0$ berechne

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}. \tag{3.40}$$

Die Vorwärtsdifferenz liefert eine Näherung für die rechtsseitige Ableitung, die Rückwärtsdifferenz für die linksseitige. Für differenzierbares f stimmen links- und rechtsseitige Ableitung überein. Um sich nicht zwischen beiden Varianten entscheiden zu müssen, kann man auch den Mittelwert beider Varianten nutzen:

- **Zentraler Differenzenquotient**: Zu vorgegebener Schrittweite $h > 0$ berechne

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2\,h} \tag{3.41}$$

  (IDV 365).

Kann f nur an vorgegebenen Stellen ausgewertet werden, z.B. wenn f nur an endlich vielen Stellen durch Messungen gegeben ist, so müssen x_0 und h bei Vorwärts- und Rückwärtsdifferenz so gewählt werden, dass f an x_0 und $x_0 + h$ bzw. $x_0 - h$ ausgewertet werden kann. Bei der zentralen Differenz ist x_0 hingegen frei wählbar, aber $x_0 + h$ und $x_0 - h$ müssen die Auswertung von f erlauben. Zu diesem Zweck können auch zwei verschiedene h gewählt werden:

- Zu vorgegebenen Schrittweiten $h_+ > 0$ und $h_- > 0$ berechne

$$f'(x_0) \approx \frac{f(x_0 + h_+) - f(x_0 - h_-)}{h_+ + h_-}. \tag{3.42}$$

Merke

Aus der Definition der Ableitung folgt sofort: Je kleiner die Schrittweite h bei der numerischen Differentiation, desto näher liegt die numerische Lösung am tatsächlichen Wert.

ABER: Diese Feststellung gilt nur, wenn die Funktionswerte von f exakt bekannt sind und die Berechnung des Quotienten ohne Rundungsfehler durchgeführt wird (was praktisch nicht möglich ist). Details dazu später.

3.4.2 Approximationsfehler

Untersuchen den Fehler, der beim Ersetzen der Ableitung $f'(x_0)$ durch einen Differenzenquotient entsteht.

Merke

Gibt es zu gegebenem f, gegebenem x_0 und gegebenem Differenzenquotient $h \mapsto D(x_0, h)$ ein $h_{\max} > 0$, ein $c > 0$ und ein $p \geq 1$, sodass

$$|f'(x_0) - D(x_0, h)| \leq c\, h^p \quad \text{für alle } h \in (0, h_{\max}] \tag{3.43}$$

gilt, so sagt man, dass der **Approximationsfehler bei x_0 von der Ordnung p** ist. Dabei darf c von f, x_0 und $h_{\max}$ abhängen, aber nicht von h.

Da die Aussage nur an einer konkreten Stelle x_0 gilt, spricht man auch vom **lokalen Approximationsfehler**.

Für die drei verschiedenen Differenzenquotienten erhält man:

- Vorwärtsdifferenz:

  $$|f'(x_0) - D(x_0, h)| \leq c\, h \tag{3.44}$$

  mit

  $$c = \frac{1}{2} \max_{\xi \in [x_0, x_0 + h_{\max}]} |f''(\xi)|, \tag{3.45}$$

  sofern f zweimal stetig differenzierbar ist.

- Rückwärtsdifferenz:

$$|f'(x_0) - D(x_0, h)| \leq c\,h \tag{3.46}$$

mit

$$c = \frac{1}{2} \max_{\xi \in [x_0 - h_{\max}, x_0]} |f''(\xi)|, \tag{3.47}$$

sofern f zweimal stetig differenzierbar ist.

- Zentrale Differenz:

$$|f'(x_0) - D(x_0, h)| \leq c\,h^2 \tag{3.48}$$

mit

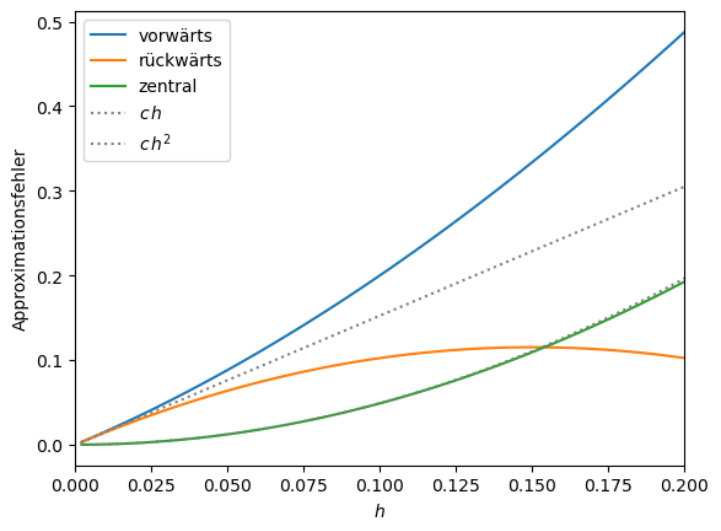
$$c = \frac{1}{6} \max_{\xi \in [x_0 - h_{\max}, x_0 + h_{\max}]} |f'''(\xi)|, \tag{3.49}$$

sofern f dreimal stetig differenzierbar ist.

(IDV 367). Bei den einseitigen Differenzen halbiert sich der Approximationsfehler also, wenn die Schrittweite halbiert wird. Bei der zentralen Differenz sinkt der Fehler auf ein Viertel bei Halbieren der Schrittweite, sofern f glatt genug ist.

Beispiel

Fehler für $h \in (0, 0.2]$ bei der näherungsweisen Berechnung von $f'(0.1)$ für $f(x) = \sin(\pi\,x)$:



Man kann die lokalen Aussagen für den Approximationsfehler zu globalen Aussagen erweitern:

Merke

Für zweimal stetig differenzierbares $f : [a, b] \to \mathbb{R}$ gilt:

- Vorwärtsdifferenz:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f''(\xi)|\, h \tag{3.50}$$

  für $h > 0$ und $x \in [a, b - h]$,

- Rückwärtsdifferenz:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f''(\xi)|\, h \tag{3.51}$$

  für $h > 0$ und $x \in [a + h, b]$.

Für dreimal stetig differenzierbares $f : [a, b] \to \mathbb{R}$ gilt:

- zentrale Differenz:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f'''(\xi)|\, h^2 \tag{3.52}$$

  für $h > 0$ und $x \in [a + h, b - h]$.

Daraus sieht man sofort, dass

- die einseitigen Differenzen exakte Ableitungswerte für lineare Funktionen liefern,
- die zentrale Differenz exakte Werte für lineare und quadratische Funktionen liefert.

3.4.3 Ableitungen höherer Ordnung

Näherungswerte für höhere Ableitungen kann man leicht durch mehrfaches Anwenden der Differenzenquotienten für die erste Ableitung erhalten. Eine übliche Approximation für die zweite Ableitung ist

$$D^2(x_0, h) := \frac{f(x_0 + h) - 2\,f(x_0) + f(x_0 - h)}{h^2}. \tag{3.53}$$

Diese erhält auf allen der folgendenen Wege:

- erst Vorwärtsdifferenz, dann Rückwärtsdifferenz,
- erst Rückwärtsdifferenz, dann Vorwärtsdifferenz,
- zweimal zentrale Differenz mit halber Schrittweite

(IDV 369).

Man kann zeigen, dass sich der Approximationsfehler hier quadratisch verhält, falls f wenigstens viermal stetig differenzierbar ist:

$$|f''(x_0) - D^2(x_0, h)| \leq c\,h^2 \quad \text{für alle } h \in (0, h_{\max}] \tag{3.54}$$

mit

$$c = \frac{1}{12} \max_{\xi \in [x_0 - h_{\max}, x_0 + h_{\max}]} |f^{(4)}(\xi)|. \tag{3.55}$$

Aus der Fehlerabschätzung sieht man sofort, dass $D^2(x_0, h)$ für Polynome dritten Grades exakte Werte liefert.

Falls f nur dreimal stetig differenzierbar ist, erhält man nur einen Approximationsfehler der Ordnung 1. Ist f mehr als viermal stetig differenzierbar, folgt trotzdem nur quadratische Ordnung für den Approximationsfehler (Analoges gilt für die Approximation der ersten Ableitung).

Nebenbemerkung

Näherungsformeln für Ableitungen (egal welcher Ordnung) haben stets die Struktur

$$\frac{1}{h^k} \sum_{i=1}^{l} a_i\, f(\ldots) \tag{3.56}$$

mit

$$\sum_{i=1}^{l} a_i = 0. \tag{3.57}$$

Dies sichert, dass die Formeln wenigstens für konstante Funktionen f stets exakte Werte (also Null) liefern.

3.5 Numerische Integration

Führen ein konkretes Verfahren zur numerischen Integration ein, welches uns gelegentlich gute Dienste leisten wird, insbesondere auch für das Lösen partieller Differentialgleichungen. Für eine umfassende Behandlung des Themas ist hier kein Platz; siehe Standardliteratur zur Numerik für weitere Verfahren.

3.5.1 Idee

Um das Integral

$$I := \int_a^b f(x)\,\mathrm{d}x \tag{3.58}$$

einer stetigen Funktion $f : [a, b] \to \mathbb{R}$ näherungsweise zu berechnen, stellen wir f als Linearkombination einfach integrierbarer Basisfunktionen dar:

$$f(x) \approx \sum_{i=1}^{n} c_i\, b_i(x). \tag{3.59}$$

Dabei sind $c_1, \ldots, c_n \in \mathbb{R}$ die von f abhängenden Koeffizienten und $b_1, \ldots, b_n : [a, b] \to \mathbb{R}$ von f unabhängige Funktionen. Aus den Eigenschaften des Integrals (Linearität!) folgt sofort

$$I = \sum_{i=1}^{n} c_i \int_a^b b_i(x)\,\mathrm{d}x. \tag{3.60}$$

Die Idee, allgemeine Funktionen als Linearkombinationen von zweckmäßig gewählten Basisfunktionen darzustellen, ist im wissenschaftlichen Rechnen weit verbreitet, sodass im Folgenden auch über die Bedürfnisse der numerischen Integration hinausgehende Beispiele für Basisfunktionen genannt werden.

3.5.2 Übliche Basisfunktionen

3.5.2.1 Stückweise konstante Funktionen Für $a =: x_0 < x_1 < \cdots < x_n := b$ setze

$$b_i(x) := \begin{cases} 1, & \text{falls } x \in [x_{i-1}, x_i), \\ 0, & \text{sonst,} \end{cases} \tag{3.61}$$

für $i = 1, \ldots, n$.

Dann gilt

$$f(x) \approx \sum_{i=1}^{n} f\left(\frac{x_i + x_{i-1}}{2}\right) b_i(x) \tag{3.62}$$

und die Integrale der Basisfunktionen sind

$$\int_a^b b_i(x)\,\mathrm{d}x = x_i - x_{i-1}. \tag{3.63}$$

3.5.2.2 Stückweise lineare Funktionen Für $a =: x_1 < x_2 < \cdots < x_n := b$ setze

$$\begin{aligned} b_1(x) &:= \begin{cases} \frac{x_2 - x}{x_2 - x_1}, & \text{falls } x \in [x_1, x_2], \\ 0, & \text{sonst}, \end{cases} \\ b_n(x) &:= \begin{cases} \frac{x - x_{n-1}}{x_n - x_{n-1}}, & \text{falls } x \in [x_{n-1}, x_n], \\ 0, & \text{sonst}, \end{cases} \end{aligned} \tag{3.64}$$

und

$$b_i(x) := \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{falls } x \in [x_{i-1}, x_i], \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{falls } x \in [x_i, x_{i+1}], \\ 0, & \text{sonst}, \end{cases} \tag{3.65}$$

für $i = 2, \ldots, n-1$.

Dann gilt

$$f(x) \approx \sum_{i=1}^{n} f(x_i)\, b_i(x) \tag{3.66}$$

und die Integrale der Basisfunktionen sind

$$\int_a^b b_i(x)\,\mathrm{d}x = \begin{cases} \frac{x_2 - x_1}{2}, & \text{für } i = 1, \\ \frac{x_{i+1} - x_{i-1}}{2}, & \text{für } i = 2, \ldots, n-1, \\ \frac{x_n - x_{n-1}}{2}, & \text{für } i = n. \end{cases} \tag{3.67}$$

3.5.2.3 Radiale Basisfunktionen Seien $x_1, \ldots, x_n \in [a, b]$ paarweise verschieden und sei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion, die nur von $|x|$ und nicht direkt von x abhängt. Dann heißen die Funktionen

$$b_i(x) := g(x - x_i) \tag{3.68}$$

radiale Basisfunktionen. Diese entstehen durch Verschiebung der Funktion g und sind symmetrisch. Üblicherweise hat g Hütchen- oder Glockenform.

Beispiele:

- Für $x_i := (i-1)\,\frac{b-a}{n-1}$ liefert

$$g(x) := \begin{cases} 1 - \frac{n-1}{b-a}\,|x|, & \text{falls } |x| \le \frac{b-a}{n-1}, \\ 0, & \text{sonst} \end{cases} \tag{3.69}$$

  gerade die Basisfunktionen zur Darstellung stückweise linearer Funktionen.

- Mit der Glockenkurve

$$g(x) = \mathrm{e}^{-\frac{x^2}{\sigma^2}} \tag{3.70}$$

erhält man sehr glatte Approximationen von f, wobei $\sigma > 0$ ein Parameter zur Steuerung der Glockenbreite ist. Da sich die Träger (Intervalle mit von Null verschiedenen Funktionswerten) der Basisfunktionen überlappen, können die Koeffizienten c_i zur Darstellung von f nicht direkt aus f abgelesen werden. Stattdessen ist ein lineres Gleichungssystem zu lösen:

$$f(x_j) = \sum_{i=1}^{n} c_i \, g(x_j - x_i), \quad j = 1, \ldots, n. \tag{3.71}$$

Die Systemmatrix hat Einsen auf der Diagonale und (überlicherweise) kleine Werte außerhalb der Diagonale.

3.5.2.4 Fourier-Basis Für glatte Funktionen $f : [a, b] \to \mathbb{R}$, die einen Ausschnitt einer periodischen Funktion darstellen, also $f(a) = f(b)$ erfüllen, bietet sich die sogenannte Fourier-Basis an. Zur Vereinfachung der Formeln wird diese gern anders indiziert:

$$\begin{aligned} b_0(x) &:= \frac{1}{b-a} \\ b_i^{\sin}(x) &:= \sqrt{\frac{b-a}{2}} \, \sin\left(2\,\pi\, i\, \frac{x-a}{b-a}\right), \\ b_i^{\cos}(x) &:= \sqrt{\frac{b-a}{2}} \, \cos\left(2\,\pi\, i\, \frac{x-a}{b-a}\right). \end{aligned} \tag{3.72}$$

Man erhält also die Folge $b_0, b_1^{\sin}, b_1^{\cos}, b_2^{\sin}, b_2^{\cos}, \ldots$ von Basisfunktionen. Die Koeffizienten zur Darstellung von f sind dann

$$\begin{aligned} c_0 &:= \int_a^b f(x)\, b_0(x)\, \mathrm{d}x, \\ c_i^{\sin} &:= \int_a^b f(x)\, b_i^{\sin}(x)\, \mathrm{d}x, \\ c_i^{\cos} &:= \int_a^b f(x)\, b_i^{\cos}(x)\, \mathrm{d}x. \end{aligned} \tag{3.73}$$

Bezeichnen wir mit

$$c := (c_0, c_1^{\sin}, c_1^{\cos}, \ldots, c_n^{\sin}, c_n^{\cos}) \tag{3.74}$$

einen Koeffizientenvektor und mit

$$g_c(x) := c_0\, b_0(x) + \sum_{i=1}^{n} c_i^{\sin}\, b_i^{\sin}(x) + c_i^{\cos}\, b_i^{\cos}(x) \tag{3.75}$$

die zugehörige Linearkombination der Basisfunktionen, so erhält man die zu f passenden Koeffizienten als Lösung des Minimierungsproblems

$$\int_a^b \big(f(x) - g_c(x)\big)^2 \, \mathrm{d}x \to \min_{c \in \mathbb{R}^{2n+1}} \tag{3.76}$$

(mehr dazu später).

3.5.2.5 Wavelet-Basen Wavelets sind Funktionen, aus denen man durch systematisches Skalieren und Verschieben Funktionensysteme mit ähnlichen Eigeschaften wie der Fourier-Basis gewinnen kann:

- beste Approximation im Sinne des integrierten quadratischen Fehlers,
- Berechnung der Koeffizienten mittels sehr effizienter Algorithmen,
- Kodierung grober Strukturen in den ersten Koeffizienten, Details in den späteren.

Jedoch bieten Wavelet-Basen viel mehr Freiheiten, da sie nicht zwingend im Frequenzbereich arbeiten (und damit stets überall im Ortsbereich wirken würden), sondern auch lokal im Ortsbereich Manipulationen erlauben.

Das einfachste Beispiel für Wavelets sind die Haar-Wavelets, welche ein Spezialfall der häufig verwendeten Daubechies-Wavelets sind.

3.5.3 Diskretisierte Stammfunktion

Ist zu $f : [a, b] \to \mathbb{R}$ die Stammfunktion

$$F : [a, b] \to \mathbb{R}, \quad F(z) = f(a) + \int_a^z f(x)\,\mathrm{d}x \tag{3.77}$$

gesucht, so muss (theoretisch) für jedes z eine numerische Integration von f über dem Intervall $[a, z]$ durchgeführt werden. Praktisch sind wir nur an $F(z)$ für endlich viele $z = z_i$ mit $a =: z_0 < z_1 < \cdots < z_n := b$ interessiert. Wegen $F(z_0) = 0$ sollen also n Werte bestimmt werden.

Wurde f auf $[a, b]$ mittels n Basisfunktionen diskretisiert, so besteht zwischen den Koeffizienten $c_1, \ldots, c_n$ und den gesuchten Funktionswerten $F(z_1), \ldots, F(z_n)$ ein linearer Zusammenhang

$$F(z_j) = \sum_{i=1}^{n} c_i \int_a^{z_j} b_i(x)\,\mathrm{d}x, \quad j = 1, \ldots, n, \tag{3.78}$$

der als Matrix-Vektor-Produkt

$$Z = A\,c \tag{3.79}$$

mit

$$c := \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}^{\mathrm{T}}, \tag{3.80}$$

$$Z := \begin{bmatrix} F(z_1) & \cdots & F(z_n) \end{bmatrix}^{\mathrm{T}} \tag{3.81}$$

und

$$A := \begin{bmatrix} \int_a^{z_1} b_1(x)\,\mathrm{d}x & \cdots & \int_a^{z_1} b_n(x)\,\mathrm{d}x \\ \vdots & & \vdots \\ \int_a^{z_n} b_1(x)\,\mathrm{d}x & \cdots & \int_a^{z_n} b_n(x)\,\mathrm{d}x \end{bmatrix} \tag{3.82}$$

geschrieben werden kann. Üblicherweise wählt man die Basisfunktionen so, dass diese zu den Integrationsgrenzen $z_1, \ldots, z_n$ passen, also z.B. stückweise lineare Hütchenfunktionen zu den Punkten $z_1, \ldots, z_n$ oder auf den Intervallen $[z_i, z_{i+1})$ konstante Basisfunktionen.

3.6 Nichtlineare Gleichungen

Beschäftigen uns mit dem numerischen Lösen nichtlinearer Gleichungen

$$f(x) = 0, \tag{3.83}$$

wobei der Definitionsbereich $[a, b]$ von $f : [a, b] \to \mathbb{R}$ so eingeschränkt sei, dass es genau eine Lösung gibt. Nichtlineare Gleichungen können immer auf diese Form gebracht werden, sodass das Lösen nichtlinearer Gleichungen äquivalent zum Finden von Nullstellen ist.

3.6.1 Kondition

Formale Untersuchungen zur Kondition des Problems sind mit unseren Mitteln nicht möglich. Die Eingabe ist hier eine Funktion. Die Ausgabe ist eine Nullstelle der Funktion. Wir müssten uns also mit Abständen in Funktionenräume beschäftigen um überhaupt über Eingabefehler reden zu können. Grundsätzlich gilt aber, dass die Kondition des Problems vor allem von der konkreten Gestalt der Funktion f festgelegt wird.

Für quadratische Gleichungen haben wir die Kondition des Nullstellenfindens genauer untersucht. Allerdings haben wir dort nicht die gesamte quadratische Funktion als Eingabe betrachtet, sondern nur die in der Funktionsdefinition auftretenden Parameter p und q. Somit konnten wir Eingabefehler mit unseren Mittel ausdrücken.

Eine andere Frage ist die nach der Kondition beim Auswerten der Funktion f an einer Stelle x. Diese Frage können wir mit unseren Mitteln beantworten, sofern eine konkrete Funktion betrachtet wird. Für eine allgemeine Funktion f sind keine Aussagen möglich. Da alle numerische Lösungsverfahren die Funktion f an mehreren Stellen auswerten müssen, ist die Frage nach der Kondition dieser Auswertungen sehr relevant für die Stabilität von Lösungsalgorithmen.

3.6.2 Iterative Verfahren

3.6.2.1 Idee Alle Lösungsverfahren für nichtlineare Gleichungen arbeiten **iterativ**, d.h. sie arbeiten nach folgendem Muster:

1. Wähle einen Startwert $x_0 \in [a, b]$.
2. Für $k = 1, 2 \ldots$ wiederhole:
   a) Berechne eine neue Näherungslösung x_k aus x_{k-1}.

Im Prinzip kann x_k nicht nur von x_{k-1} sondern auch von mehreren Vorgängern abhängen, ist aber hier nicht Thema.

Die Folge $x_0, x_1, x_2, \ldots$ liefert immer bessere Näherungen an die tatsächliche Lösung, sofern das Verfahren sinnvoll konstruiert ist.

Merke

Ein iteratives Verfahren heißt **konvergent** (auch: "es konvergiert"), wenn für die exakte Lösung x^* gilt:

$$x^* = \lim_{k \to \infty} x_k. \tag{3.84}$$

Um aus der obigen Verfahrensidee konkrete Algorithmen zu erhalten, müssen wir folgende Fragen beantworten:

- Wie wählen wir x_0?
- Wie berechnen wir x_k aus x_{k-1}?
- Wann brechen wir die Iteration ab?

3.6.2.2 Abbruchkriterium Die Frage nach dem Abbruch der Iteration ist relevant, weil ein Computer nur endlich viele Rechenschritte durchführen kann. Die Antwort ist eine Abwägung zwischen verfügbarer Rechenzeit und akzeptiertem Lösungsfehler. Folgende Varianten sind üblich:

- Falls vor allem die Rechenzeit relevant ist, bricht man nach einer vorgegebenen Anzahl an Iterationen ab.
- Falls vor allem die Genauigkeit der Lösung relevant ist, bricht man erst ab, wenn die gewünschte Genauigkeit erreicht ist. Da man die exakte Lösung nicht kennt, ist die Machbarkeit dieses Ansatzes vom konkreten Verfahren abhängig.
- Falls vor allem die Genauigkeit der Lösung relevant ist, aber diese bei dem gewählten Verfahren nicht zuverlässig bestimmt werden kann, bricht man ab, wenn sich die Näherungslösungen im Rahmen der gewünschten Genauigkeit nicht mehr ändern.
- Praktisch wird man abbrechen, wenn die Genauigkeit hoch genug ist, und zusätzlich eine (hohe) maximale Iterationsanzahl vorgeben um auch in ungünstigen Fällen (langsame Konvergenz) die Rechenzeit in akzeptablem Rahmen zu halten.

3.6.2.3 Konvergenzordnung Aus der Beziehung $x^* = \lim_{k \to \infty} x_k$ lassen sich keine Erkenntnisse zur Konvergenzgeschwindigkeit gewinnen. Deshalb ist man an konkreten Abschätzungen des Lösungsfehlers interessiert:

Merke

Gilt

$$|x_k - x^*| \leq c\,\gamma^k, \qquad k = 1, 2, \ldots, \tag{3.85}$$

für ein $\gamma \in (0,1)$ und ein $c > 0$, so heißt das iterative Verfahren **linear konvergent** oder **konvergent mit der Konvergenzordnung 1**.

Gilt sogar

$$|x_k - x^*| \leq c\,\gamma^{(p^k)}, \qquad k = 1, 2, \ldots, \tag{3.86}$$

für ein $p > 1$, ein $\gamma \in (0,1)$ and ein $c > 0$, so heißt das iterative Verfahren **konvergent mit Konvergenzordnung** p.

Aus beiden Abschätzungen folgt insbesondere $x^* = \lim_{k\to\infty} x_k$, also Konvergenz.

Je höher die Konvergenzordnung, desto schneller die Konvergenz. Entsprechend sind bei hoher Konvergenzordnung weniger Iterationen nötig um einen vorgegebenen maximalen Lösungsfehler zu erreichen als bei niedriger Konvergenzordnung.

Abschätzungen der obigen Form sind jedoch selten direkt zu bekommen. Meist fügt man einen Zwischenschritt folgender Form ein:

$$|x_k - x^*| \leq C\,|x_{k-1} - x^*|^p, \qquad k = 1, 2, \ldots, \tag{3.87}$$

wobei $p \geq 1$ und $C > 0$. Daraus folgt dann Konvergenz mit Konvergenzordnung p, sofern gewisse Bedingungen an die Konstante C und den Startfehler $|x_0 - x^*|$ erfüllt sind.

Beispiel

Gilt für ein festes k, dass $|x_{k-1} - x^*| \approx 0.1$, so sinkt der Fehler in den nächsten Schritten bei linearer Konvergenz ($p = 1$) stets nur um den Faktor $C < 1$. Bei quadratischer Konvergenz ($p = 2$) hingegen beträgt der Fehler in den nächsten Schritte etwa 0.01, 0.0001, 0.00000001.

3.6.3 Bisektionsverfahren

3.6.3.1 Verfahren Das Bisektionsverfahren (auch: Intervallschachtelung) ist ein einfaches Verfahren zur Nullstellenbestimmung.

Algorithmus

1. Wähle a_0 and b_0 so, dass $f(a_0)\,f(b_0) \leq 0$ gilt.
2. Setze $x_0 := \frac{a_0+b_0}{2}$.
3. Für $k = 1, 2, \ldots$ wiederhole:

   a) Setze

$$a_k := \begin{cases} a_{k-1}, & \text{falls } f(a_{k-1})\, f(x_{k-1}) \leq 0, \\ x_{k-1}, & \text{sonst} \end{cases} \tag{3.88}$$

und

$$b_k := \begin{cases} x_{k-1}, & \text{falls } f(a_{k-1})\, f(x_{k-1}) \leq 0, \\ b_{k-1}, & \text{sonst.} \end{cases} \tag{3.89}$$

b) Setze $x_k := \frac{a_k + b_k}{2}$.

Die Bedingung der Form $f(a)\, f(b) \leq 0$ sichert, dass entweder im Intervall (a, b) eine Nullstelle liegt (bei strenger Ungleichheit) oder ein Intervallende eine Nullstelle ist (bei Gleichheit). Liegen mehrere Nullstellen im Startintervall $[a_0, b_0]$, so liefert das Verfahren nur eine dieser Nullstellen, wobei keine allgemeine Aussagen darüber möglich sind, welche Nullstelle geliefert wird.

3.6.3.2 Konvergenz und Abbruchkriterium

Man kann leicht zeigen:

Merke

Für stetiges f konvergiert das Bisektionsverfahren linear mit $\gamma = \frac{1}{2}$ und $c = \frac{b_0 - a_0}{2}$.

Ist der gewünschte maximale (absolute) Lösungsfehler $\Delta > 0$ vorgegeben, so kann wegen

$$|x_k - x^*| \leq \frac{b_k - a_k}{2} \tag{3.90}$$

die Iteration gestoppt werden, sobald $b_k - a_k \leq 2\,\Delta$. Wegen

$$b_k - a_k = \frac{b_0 - a_0}{2^k} \tag{3.91}$$

ist die Iterationsanzahl dann

$$n = \left\lceil \log_2 \frac{b_0 - a_0}{2\,\Delta} \right\rceil ; \tag{3.92}$$

insbesondere ist sie im Vorfeld exakt bekannt. Eine zusätzliche Begrenzung der Iterationsanzahl als weiteres Abbruchkriterium ist nicht nötig.

3.6.3.3 Algorithmus

Ein konkreter Algorithmus zur Umsetzung des Bisektionsverfahrens kann wie folgt aussehen:

Algorithmus

1. Zu gegebenen a_0, b_0 mit $a_0 < b_0$ und $f(a_0)\,f(b_0) \leq 0$ setze $a := a_0$, $b := b_0$.
2. Setze $A := f(a)$ und $B := f(b)$.
3. Wiederhole solange $b - a > 2\,\Delta$:
   a) Berechne $x := \frac{a+b}{2}$.
   b) Setze $X := f(x)$.
   c) Falls $A\,X \leq 0$,
   setze $b := x$ und $B := X$,
   sonst
   setze $a := x$ und $A := X$.
4. Berechne das Ergebnis $x := \frac{a+b}{2}$.

Eine vollständige Stabilitätsanalyse ist trotz des relativ einfachen Algorithmus recht mühsam. Das Verfahren gilt als sehr stabil. Wenn die Funktion f also fehlerfrei auswertbar ist (fehlerfreie Eingabe), dann liegt der Ausgabefehler in der Größenordnung des durch übliche fehlerbehaftete Eingaben auch bei exakter Rechnung nicht zu vermeidenden Ausgabefehlers (Kondition!), sofern Δ im Abbruchkriterium entsprechend klein gewählt wird.

3.6.4 Newton-Verfahren

3.6.4.1 Verfahren Das Newton-Verfahren nutzt zusätzlich zu den Funktionswerten von f auch die erste Ableitung f' um Informationen über die Lage einer Nullstelle zu erhalten.

Algorithmus

1. Wähle einen Startpunkt x_0.
2. Für $k = 1, 2, \ldots$ wiederhole:
   a) Setze

$$x_k := x_{k-1} + \frac{f(x_{k-1})}{f'(x_{k-1})}. \tag{3.93}$$

Dieses Verfahren wählt also x_k als Nullstelle der Tangente an f an der Stelle x_{k-1} (IDV 375).

3.6.4.2 Konvergenz und Abbruchkriterium

Merke

Seien a und b so gewählt, dass $x^* \in (a, b)$, und sei f zweimal stetig differenzierbar in (a, b) mit

$$0 < c_1 \leq |f'(x)| \quad \text{und} \quad |f''(x)| \leq c_2 \quad \text{für alle } x \in (a, b) \tag{3.94}$$

für Konstanten c_1 und c_2. Gilt $x_{k-1} \in (a, b)$, so kann man

$$|x_k - x^*| \leq \frac{c_2}{2\,c_1} |x_{k-1} - x^*|^2 \tag{3.95}$$

zeigen.

Sofern alle Iterierten im Intervall (a, b) liegen, folgt aus dieser Abschätzung quadratische Konvergenz $|x_k - x^*| \leq c\,\gamma^{(2^k)}$ mit $c = 1$ und

$$\gamma = \begin{cases} \sqrt{\frac{c_2}{2\,c_1}}\,|x_0 - x^*|, & \text{falls } c_2 < 2\,c_1, \\ \frac{c_2}{2\,c_1}\,|x_0 - x^*|, & \text{sonst,} \end{cases} \tag{3.96}$$

wobei

$$|x_0 - x^*| < \begin{cases} \sqrt{\frac{2\,c_1}{c_2}}, & \text{falls } c_2 < 2\,c_1, \\ \frac{2\,c_1}{c_2}, & \text{sonst} \end{cases} \tag{3.97}$$

gelten muss. Damit das Newton-Verfahren konvergiert müssen also insbesondere zwei Bedingungen erfüllt sein:

- Der Startwert x_0 liegt nah genug bei der (unbekannten) Lösung x^*.
- Die Iterierten x_k müssen in der Nähe von x^* bleiben.

Sind diese Bedingungen erfüllt, so konvergiert das Newton-Verfahren quadratisch. Man spricht auch von **lokaler quadratischer Konvergenz**. Im Gegensatz dazu ist das Bisektionsverfahren global konvergent, allerdings nur linear.

Da die Konstanten c_1 und c_2 praktisch nicht zugänglich sind, kann aus der Abschätzung für die Konvergenzordnung kein in der Praxis einsetzbares Abbruchkriterium gewonnen werden. Man behilft sich mit der Tatsache, dass bei Konvergenz der Abstand $|x_k - x_{k-1}|$ zwischen zwei Iterierten gegen Null geht. Abgebrochen wird also, wenn dieser Abstand unter eine vorgegebene Schranke fällt.

3.6.4.3 Algorithmus Eine konkrete Umsetzung des Newton-Verfahrens kann wie folgt aussehen (ohne explizites Erwähnen des Rundens nach jeder Rechenoperation):

Algorithmus

1. Zu gegebenem Startwert x_0 setzte $x_{\text{alt}} := x_0$.
2. Berechne $x := x_0 - \frac{f(x_0)}{f'(x_0)}$.
3. Wiederhole solange $|x - x_{\text{alt}}| > \Delta$:
   a) Setze $x_{\text{alt}} := x$.
   b) Berechne $x := x - \frac{f(x)}{f'(x)}$.

Dieser Algorithmus ist stabil. Insbesondere werden Rundungsfehler im Laufe der Iterationen nicht verstärkt. Jede neue Iteration kann als Neustart des Algorithmus mit anderem Startwert betrachtet werden.

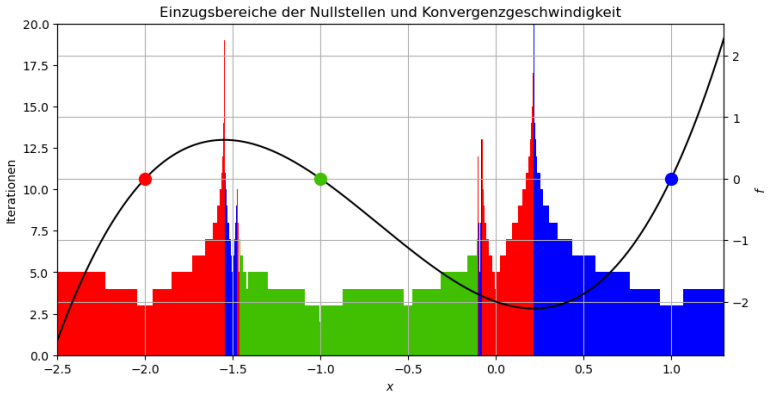
3.6.4.4 Beispiele Um das Problem der nicht globalen Konvergenz zu verdeutlichen führen wir das Newton-Verfahren für viele verschiedene Startwerte x_0 aus und stellen die Konvergenz bzw. Nichtkonvergenz grafisch dar. Dabei sehen wir eine Näherungsfolge $x_0, x_1, x_2, \ldots$ als konvergent an, wenn innerhalb von 1000 Iterationen der Abstand $|x_k - x_{k-1}|$ unter 10^{-10} sinkt. Zusätzlich stellen wir dar, gegen welche von ggf. mehreren Nullstellen die Folge konvergiert.

In den folgenden Diagrammen sind neben der betrachteten Funktion f die Iterationsanzahlen bis zum erstmaligen Unterschreiten der Fehlerschranke und (farblich) die angenäherten Nullstellen ausgehend von verschiedenen Startwerten x_0 dargestellt.

3.6.4.4.1 Polynom 3. Grades Konvergenzverhalten für das Polynom

$$f(x) = x^3 + 2\,x^2 - x - 2, \quad x \in [-2.5, 1.3]. \tag{3.98}$$

```
visualize_newton(
    lambda x: x ** 3 + 2 * x ** 2 - x - 2,
    lambda x: 3 * x ** 2 + 4 * x - 1,
    [ -2, -1, 1],
    -2.5, 1.3
)
```



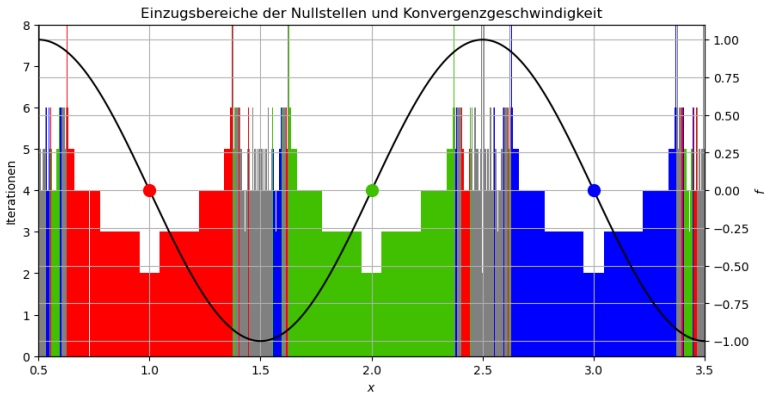
An den Extremstellen gilt $f'(x) = 0$, sodass das Newton-Verfahren dort nicht anwendbar ist (alle Berechnungen liefern `nan`, sodass die Abbruchbedingung nie erfüllt ist). Für x_0, für die weitere Iterierte auf eine Extremstelle fallen, ist das Newton-Verfahren ebenfalls nicht anwendbar. Ist x_0 in der Nähe einer Extremstelle, so führen schon sehr kleine Änderungen an x_0 zum Wechsel der angestrebten Nullstelle.

3.6.4.4.2 Sinus

Konvergenzverhalten für die Sinusfunktion

$$f(x) = \sin(\pi\, x), \quad x \in [0.5, 3.5]. \tag{3.99}$$

```
visualize_newton(
    lambda x: np.sin(np.pi * x),
    lambda x: np.pi * np.cos(np.pi * x),
    [1, 2, 3],
    0.5, 3.5
)
```



Liegt x_0 in der Nähe einer Extremstelle, so verläuft die Tangente sehr flach. Entsprechend weit entfernt ist x_1, sodass auch Startwerte innerhalb des dargestellten Intervalls zur Annäherung an weit entfernt liegende Nullstellen führen können. Liegt die angestrebte

Nullstelle nicht im dargestellten Intervall, so wird der entsprechende Startwert grau eingefärbt (gleiche Farbe wie bei Nicht-Konvergenz).

3.7 Nichtlineare Gleichungssysteme

Wir werfen einen kurzen Blick auf das numerische Lösen nichtlinearer Gleichungssysteme

$$F(x) = 0, \quad F = (f_1, \ldots, f_d) : D \to \mathbb{R}^d, \tag{3.100}$$

wobei $D \subseteq \mathbb{R}^d$ der Definitionsbereich der vektorwertigen Funktion F ist. Beschränken uns auf den Fall, dass die Anzahl der Unbekannten mit der Anzahl der Gleichungen übereinstimmt.

Gelegentlich treten nichtlineare Gleichungssysteme mit nur kleinem d auf. Meist entstehen die Systeme jedoch als endliche Näherung kontinuierlicher Probleme (Strömungsmechanik usw.) und weisen dann sehr viele Unbekannte auf.

Ziel dieses Kapitels ist nicht die detaillierte Behandlung konkreter Verfahren, sondern das Aufzeigen von zusätzlichen Fragen und Problemen, die im eindimensionalen Fall nicht auftraten.

3.7.1 Bisektion vs. Dimension

Prinzipiell lässt sich das Bisektionsverfahren auf höhere Dimensionen übertragen, allerdings nicht so einfach wie es zunächst erscheinen mag; siehe z.B. A rapid Generalized Method of Bisection for solving Systems of Non-linear Equations. Hauptproblem ist die mit der Dimension d exponentiell wachsende Anzahl zu untersuchender Punkte. Während im eindimensionalen Fall der Suchbereich durch zwei Punkte begrenzt ist, werden in d Dimensionen 2^d Punkte benötigt. Dies entspricht gerade der Anzahl der Ecken eines d-dimensionalen Quaders.

Beispiel: Für ein nichtlineares Gleichungssystem mit nur $d = 30$ Unbekannten und Gleichungen wäre das Suchgebiet durch ca. 1 Milliarde Punkte begrenzt!

3.7.2 Newton-Verfahren

Das Newton-Verfahren lässt sich leicht auf Gleichungssysteme übertragen. Das Newton-Verfahren und daraus abgeleitete Verfahren (siehe unten) sind die wichtigsten Verfahren für das Lösen auch großer nichtlinearer Gleichungssysteme.

3.7.2.1 Verfahren Wie bei $d = 1$ lösen wir eine linearisierte Version des nichtlinearen Gleichungsystems, wobei wir die Linearisierungspunkte sukzessive ausgehend von einem Startpunkt x_0 als Lösung des vorhergehenden linearisierten Systems berechnen.

Für gegebenen Punkt x_{k-1} liefert der Satz von Taylor bei zweimal stetig differenzierbarem F die Näherung

$$F(x) \approx F(x_{k-1}) + J_F(x_{k-1})\,(x - x_{k-1}), \tag{3.101}$$

wobei

$$J_F(x_{k-1}) := \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x_{k-1}) & \cdots & \frac{\partial f_1}{\partial x_d}(x_{k-1}) \\ \vdots & & \vdots \\ \frac{\partial f_d}{\partial x_1}(x_{k-1}) & \cdots & \frac{\partial f_d}{\partial x_d}(x_{k-1}) \end{bmatrix} \tag{3.102}$$

die Matrix der ersten partiellen Ableitungen der Komponenten $f_1, \ldots, f_d$ von F ist ("Jacobi-Matrix"). Das nichtlineare Gleichungssystem $F(x) = 0$ wird also in einer (kleinen) Umgebung von x_{k-1} näherungsweise durch das lineare Gleichungssystem

$$J_F(x_{k-1})\, x = J_F(x_{k-1})\, x_{k-1} - F(x_{k-1}) \tag{3.103}$$

beschrieben. Ist $J_F(x_{k-1})$ invertierbar, so besitzt dieses die eindeutige Lösung

$$x_k := x_{k-1} - J_F(x_{k-1})^{-1}\, F(x_{k-1}). \tag{3.104}$$

Dies ist die Iterationsvorschrift für das mehrdimensionale Newton-Verfahren.

3.7.2.2 Probleme Man kann zeigen, dass das mehrdimensionale Newton-Verfahren quadratisch konvergiert, also

$$|x_k - x^*| \leq C\, |x_{k-1} - x^*|^2 \tag{3.105}$$

gilt, sofern die Funktion F gewisse, recht strenge Bedingungen erfüllt und x_0 nah genug an der Lösung x^* liegt. Beachte, dass x^* und alle x_k hier Vektoren aus $\mathbb{R}^d$ sind, in der Ungleichung nun also Längen von Vektoren verglichen werden.

Wie man x_0 in der Praxis wählen muss um quadratische Konvergenz oder überhaupt Konvergenz zu erhalten, ist nicht so klar. Der Satz von Kantorovich liefert hier praktisch verwertbare Anhaltspunkte. Letztlich wird es aber meist auf "Versuch und Irrtum" hinauslaufen.

Neben dem auch im eindimensionalen Fall vorhandenen Konvergenzproblem macht zusätzlich die Jacobi-Matrix $J_F(x_{k-1})$ Probleme. Selbst wenn diese als auswertbare Formel vorliegt, müssen in jedem Interationsschritt die d^2 Einträge neu berechnet werden. Oft ist F so beschaffen, dass man J_F gar nicht explizit in Formeln ausdrücken kann. Dann muss $J_F(x_{k-1})$ numerisch bestimmt werden. Die Probleme um J_F haben vielfältige Lösungen hervorgeracht, die zu unterschiedlichen, so genannten Newton-ähnlichen Verfahren geführt haben.

3.7.3 Newton-ähnliche Verfahren

Bezüglich der Jacobi-Matrix J_F im mehrdimensionalen Newton-Verfahren sind zwei Probleme zu lösen:

- Ersetze J_F durch eine Matrix mit möglichst vielen Nullen (weniger Rechenaufwand) aber annähernd gleicher Wirkung.

- Bestimme J_F numerisch stabil (!) mit möglichst geringem Aufwand.

Der zweite Punkt ist besonders schwierig, weil Differentiation ein schlecht konditioniertes Problem ist.

Lösungsansätze:

- Beim **vereinfachten Newton-Verfahren** wird $J_F(x_{k-1})$ nicht bei jedem Schritt neu berechnet, sondern über einige Schritte konstant gehalten. Hintergrund ist, dass J_F sich von Schritt zu Schritt ohnehin meist nur wenig ändert.
- **Quasi-Newton-Verfahren** bestimmen eine Nährung für J_F aus den zurückliegenden Iterationsschritten. Sie "sammeln" also im Laufe der Zeit Informationen über die Ableitungen von F. Dies ist deutlich effizienter (aber technisch schwieriger) als numerische Differentiation ohne Nutzung der ohnehin bereits bekannten Funktionswerte.
- Bei manchen Anwendungen, z.B. beim Lösen von partiellen Differentialgleichungen, hat J_F eine gut verstandene Struktur, die den Einsatz **schneller Löser** für das lineare Gleichungssystem in jedem Iterationsschritt zulässt. Dies ist z.B. möglich, wenn man weiß, dass J_F nur in der Nähe der Diagonale von Null verschieden ist.

3.7.4 Ausflug: Newton-Fraktal

Um die Konvergenz bzw. Nichtkonvergenz des Newton-Verfahrens in Abhängigkeit vom Startpunkt zu visualisieren betrachten wir verschiedene Beispiele mit $d = 2$.

Wir generieren ein regelmäßiges Rechteckgitter aus Startpunkten und führen das Newton-Verfahren mit jedem dieser Punkte als Startpunkt aus. Jeder Lösung der Gleichung $F(x, y) = (0, 0)$ wird eine Farbe zugeordnet. Die Startpunkte werden in der Farbe der Lösung gefärbt, zu der die Iteration führt. Je langsamer die Konvergenz (hohe Iterationsanzahl), desto dunkler die Farbe.

Startpunkte, die nicht zur Konvergenz führen, werden schwarz erscheinen. Zur leichteren Wahrnehmung dieser Punkte erstellen wir eine zweite Grafik, die diese Startpunkte in weiß auf schwarzem Grund darstellt. Grautöne entsprechen langsamer Konvergenz.

Schauen uns zunächst die Ergebnisse für die Funktion

$$F(x, y) := \begin{bmatrix} x^2 - y^2 - 1 \\ 2\,x\,y \end{bmatrix} \tag{3.106}$$

an. Diese Funktion hat zwei Nullstellen bei $(-1, 0)$ und $(1, 0)$ (weiße Punkte).

```
x, y = sympy.symbols('x y', real=True)

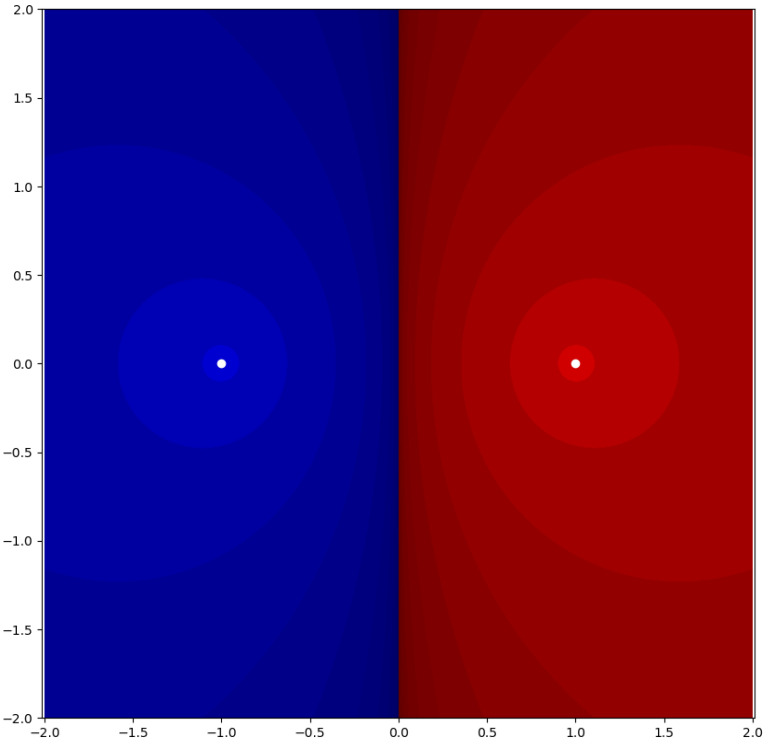
#z = x + y * sympy.I
#f = z ** 2 - 1
```

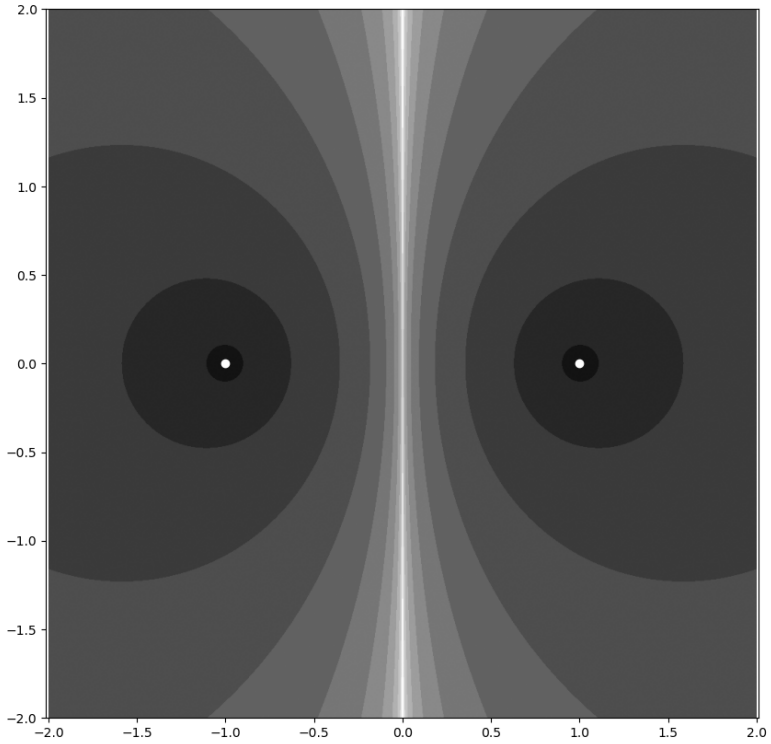
```
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 2 - y ** 2 - 1
f2 = 2 * x * y

roots = [[1, 0], [ -1, 0]]

newton_fractal(f1, f2, x, y, roots)
```





Das Newton-Verfahren konvergiert für alle Startpunkte, die nicht auf der vertikalen Koordinatenachse liegen. Je näher der Startpunkt an einer Nullstelle liegt, desto schneller die Konvergenz.

Erhöhen wir den Polynomgrad auf 3, z.B.

$$F(x,y) := \begin{bmatrix} x^3 - 3\,x\,y^2 - 1 \\ 3\,x^2\,y - y^3 \end{bmatrix}, \tag{3.107}$$

so wird das Konvergenzverhalten deutlich komplexer. Die Funktion hat 3 Nullstellen (weiße Punkte).

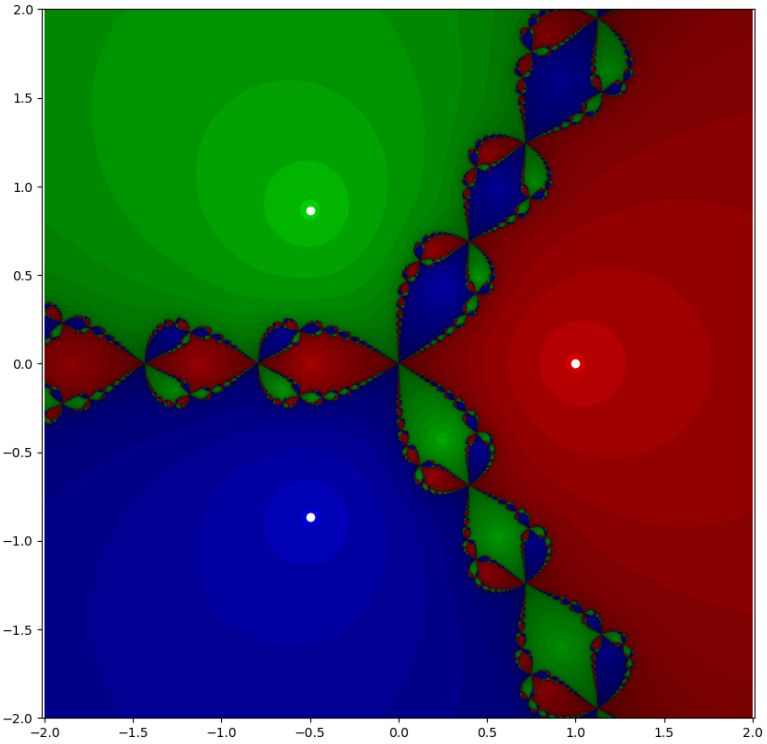
```
x, y = sympy.symbols('x y', real=True)

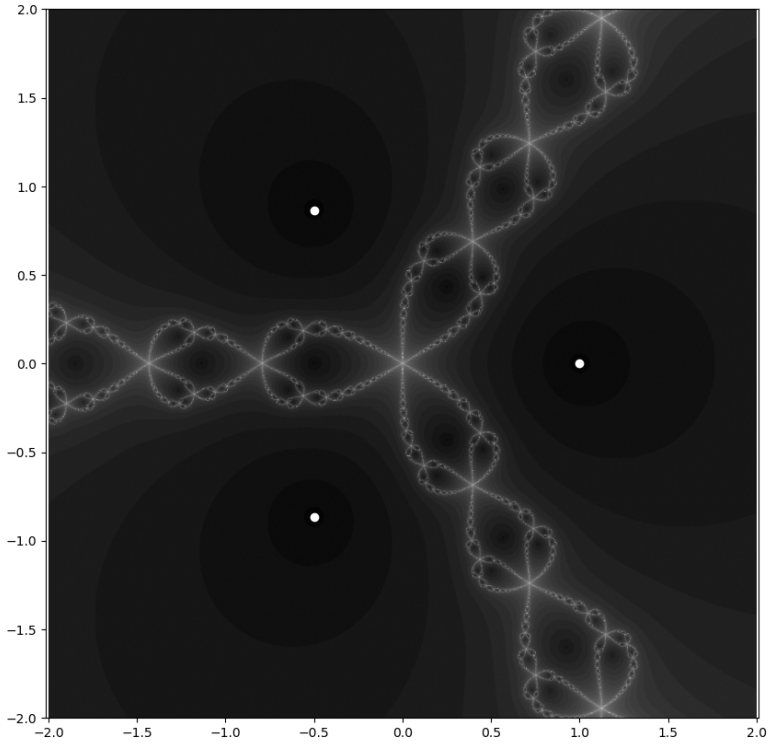
#z = x + y * sympy.I
#f = z ** 3 - 1
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 3 - 3 * x * y ** 2 - 1
f2 = 3 * x ** 2 * y - y ** 3

roots = [[1, 0], [ -0.5, 0.5 * np.sqrt(3)], [ -0.5, - 0.5 *
↪  np.sqrt(3)]]
```

```
newton_fractal(f1, f2, x, y, roots)
```





Bei Start in der Nähe einer Nullstelle konvergiert das Newton-Verfahren zu dieser Nullstelle. Liegt der Startpunkt jedoch etwa gleich weit entfernt von zwei Nullstellen, so führen schon sehr kleine Änderungen am Startwert zu einem Wechsel der angenäherten Nullstelle.

Die Menge der Startpunkte von denen aus das Newton-Verfahren nicht konvergiert heißt **Newton-Fraktal**. Diese Menge hat eine äußerst komplexe sich sowohl in der Ausdehnung als auch im Detail immer wieder wiederholende Struktur. Das Newton-Fraktal ist weder Linie, noch Fläche, sondern eine Menge mit gebrochener Dimension zwischen 1 und 2 bei ca. 1.42. Siehe z.B. Fractal Basins of Attraction Associated with a Damped Newton's Method für Details.

Zur besseren Visualisierung des Newton-Fraktals (Zoom usw.) existieren verschiedene Computerprogramme, z.B. XaoS (open-source für alle gängigen Plattformen).

Polynom 6. Grades:

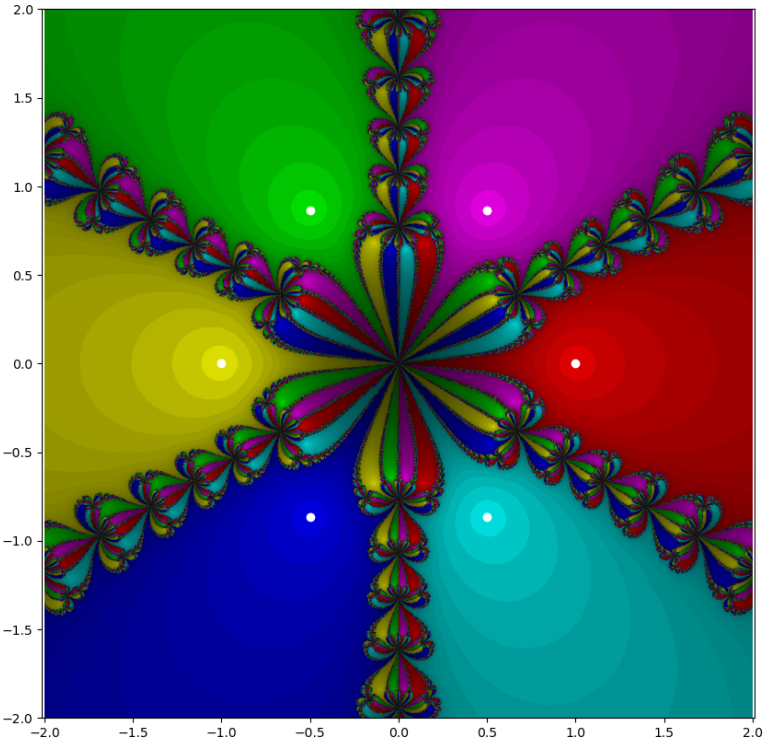
```
x, y = sympy.symbols('x y', real=True)

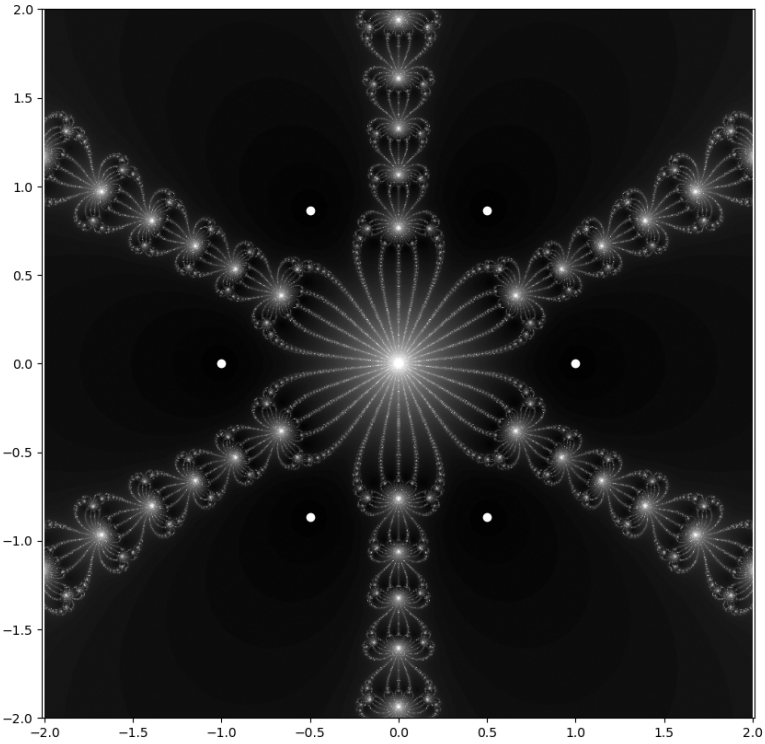
#z = x + y * sympy.I
#f = z ** 6 - 1
#f1 = sympy.re(f)
#f2 = sympy.im(f)
```

```
f1 = x ** 6 - 15 * x ** 4 * y ** 2 + 15 * x ** 2 * y ** 4 - y ** 6 - 1
f2 = 6 * x ** 5 * y - 20 * x ** 3 * y ** 3 + 6 * x * y ** 5

roots = [
    [1, 0], [ -0.5, 0.5 * np.sqrt(3)], [ -0.5, - 0.5 * np.sqrt(3)],
    [ -1, 0], [0.5, 0.5 * np.sqrt(3)], [0.5, - 0.5 * np.sqrt(3)]
]

newton_fractal(f1, f2, x, y, roots)
```





Für manche Funktionen gibt es ganze Flächen an Startpunkten, bei denen das Newton-Verfahren nicht konvergiert. Beispiel:

$$F(x,y) := \begin{bmatrix} x^3 - 3\,x\,y^2 - 2\,x + 2 \\ 3\,x^2\,y - y^3 - 2\,y \end{bmatrix}. \tag{3.108}$$

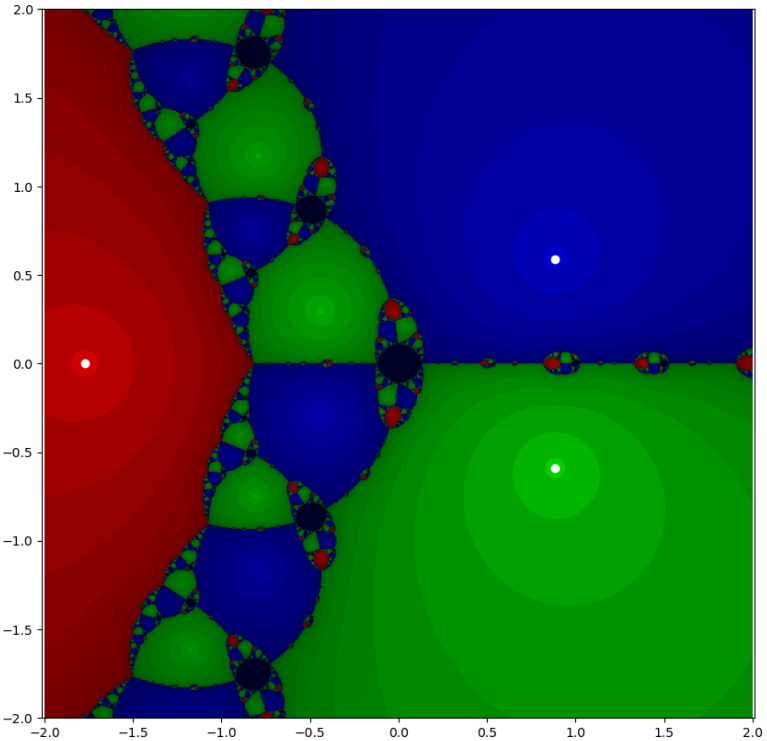
```
x, y = sympy.symbols('x y', real=True)

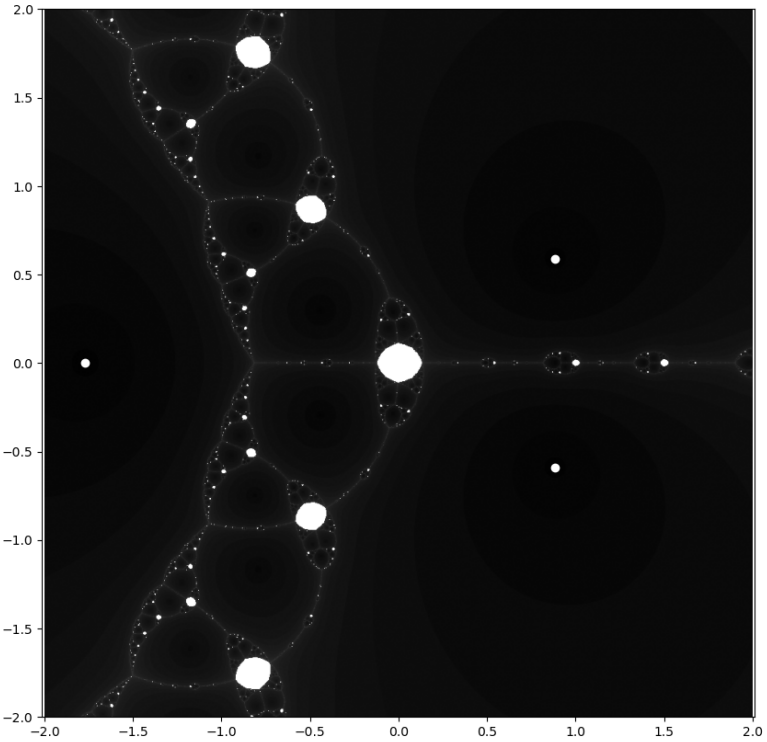
#z = x + y * sympy.I
#f = z ** 3 - 2 * z + 2
#f1 = sympy.re(f)
#f2 = sympy.im(f)

f1 = x ** 3 - 3 * x * y ** 2 - 2 * x + 2
f2 = 3 * x ** 2 * y - y ** 3 - 2 * y

roots = [[ -1.7693, 0], [0.88465, -0.58974], [0.88465, 0.58974]]

newton_fractal(f1, f2, x, y, roots)
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

3.8 Lineare Gleichungssysteme

Lineare Gleichungssysteme

$$\begin{aligned} a_{1,1}\,x_1 + \cdots + a_{1,n}\,x_n &= b_1 \\ &\vdots \\ a_{m,1}\,x_1 + \cdots + a_{m,n}\,x_n &= b_m \end{aligned} \tag{3.109}$$

entstehen in allen Bereichen der Wissenschaft, entweder direkt zur Formulierung eines praktischen Problems oder als Teilschritt innerhalb komplexerer Lösungsstrategien. Dabei sind die Zahlen $a_{i,j}$ und b_i gegeben und die Zahlen $x_1, \ldots, x_n$ gesucht.

Zur Vereinfachung der Darstellung setzt man üblicherweise die **Systemmatrix**

$$A := \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{3.110}$$

und den **Vektor der rechten Seiten**

$$b := \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}. \tag{3.111}$$

Das Gleichungssystem bekommt damit die Form

$$A\,x = b \tag{3.112}$$

mit dem **Lösungsvektor**

$$x := \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{3.113}$$

Es existieren vielfältige numerische Lösungsverfahren mit unterschiedlichen Eigenschaften bzgl. Stabilität, Rechenaufwand, Speicherbedarf und Anwendungsbreite. Schauen uns hier zwei relativ breit anwendbare Verfahren näher an und geben einen kurzen Überblick über weitere Verfahrensideen.

Beschränken uns auf invertierbare quadratische Systemmatrizen, also auf den Fall $m = n$. Insbesondere folgt aus der Forderung der Invertierbarkeit, dass das Gleichungssystem genau eine Lösung besitzen soll. Diese kann dann als

$$x = A^{-1}\,b \tag{3.114}$$

geschrieben werden.

3.8.1 Kondition

Zur Beurteilung der Kondition beim Lösen eines linearen Gleichungssystems muss zunächst geklärt werden, welche Größen als (potentiell fehlerbehaftete) Eingabe zu betrachten sind. Hier gibt es zwei Ansätze:

- Nur b wird als Eingabe betrachtet und es ist die Kondition der Abbildung $b \mapsto A^{-1}\,b$ gesucht. Dieser Ansatz ist leichter zu handhaben und wird immer verwendet, wenn A sehr genau oder sogar exakt bekannt ist.
- A und b werden beide als Eingaben betrachtet und es ist die Kondition der Abbildung $(A, b) \mapsto A^{-1}\,b$ gesucht.

Weiterhin bestehen gewisse Freiheiten bei der Beantwortung der Frage, wie wir Abweichungen zwischen exakten und fehlerbehafteten Ein- bzw. Ausgaben ausdrücken. Bisher haben wir Konditionsbetrachtungen nur für Abbildungen $f : \mathbb{R}^d \to \mathbb{R}$ angestellt. Auf der Eingabeseite hatten wir uns für die Summe der komponentenweisen relativen Fehler entschieden. Auf der Ausgabeseite bestand gar keine Wahl. Im Kontext linearer Gleichungssysteme muss nun auch auf der Ausgabeseite entschieden werden, wie die relativen Fehler in den einzelnen Komponenten zu einem Gesamtausgabefehler zusammengeführt werden.

3.8.1.1 Vektornormen Für einen Vektor $v \in \mathbb{R}^n$, z.B. den Vektor der komponentenweisen relativen Ein- oder Ausgabefehler, können wir verschiedene **Normen** einführen, also Zahlen $\|v\|$, die die Größe der einzelnen Einträge des Vektors sinnvoll zusammenfassen. Unter "sinnvoll" verstehen wir, dass folgende Eigenschaften gelten sollen:

- $\|v\| \geq 0 \quad$ für alle $v \in \mathbb{R}^n$,
- $\|v\| = 0 \;\Rightarrow\; v = 0$,
- $\|\alpha\, v\| = |\alpha|\,\|v\| \quad$ für alle $\alpha \in \mathbb{R}$ und alle $v \in \mathbb{R}^n$,
- $\|v + w\| \leq \|v\| + \|w\| \quad$ für alle $v, w \in \mathbb{R}^n$.

Beispiele:

- **1-Norm**:

$$\|v\|_1 := \sum_{i=1}^{n} |v_i|, \tag{3.115}$$

- **∞-Norm**:

$$\|v\|_\infty := \max_{i \in \{1,\ldots,n\}} |v_i|, \tag{3.116}$$

- **euklidische Norm**:

$$\|v\|_2 := \sqrt{\sum_{i=1}^{n} v_i^2}. \tag{3.117}$$

Alle Normen in $\mathbb{R}^n$ sind **äquivalent**, d.h. für zwei Normen $\|\cdot\|_\alpha$ und $\|\cdot\|_\beta$ gibt es stets Konstanten $c, C > 0$, sodass

$$c\,\|v\|_\alpha \leq \|v\|_\beta \leq C\,\|v\|_\alpha \quad \text{für alle } v \in \mathbb{R}^n \tag{3.118}$$

gilt. Mit Blick auf Fehlerabschätzungen beeinflussen die eingesetzten Normen also höchstens die auftretenden konstanten Faktoren, die ohnehin kaum relevant für die Kernaussagen sind.

3.8.1.2 Matrixnormen Bei Konditionsuntersuchungen werden wir stets auf Vektoren der Form $A\,x$ und $A^{-1}\,b$ stoßen, die wir in Beziehung zu x bzw. b setzen möchten. Zu diesem Zweck haben sich **von einer Vektornorm induzierte Matrixnormen** etabliert.

Merke

Für $A \in \mathbb{R}^{m\times n}$ und eine gegebene Vektornormen $\|\cdot\|_\alpha$ und $\|\cdot\|_\beta$ in $\mathbb{R}^m$ bzw. $\mathbb{R}^n$ ist durch

$$\|A\|_{\alpha,\beta} := \max_{v\in\mathbb{R}^n\setminus\{0\}} \frac{\|A\,v\|_\alpha}{\|v\|_\beta} \tag{3.119}$$

eine **Matrixnorm** gegeben, d.h. es sind die gleichen vier Eigenschaften wie bei Vektornormen erfüllt. Zusätzlich gilt

$$\|A\,v\|_\alpha \leq \|A\|_{\alpha,\beta}\,\|v\|_\beta \quad \text{für alle } v \in \mathbb{R}^m. \tag{3.120}$$

Am häufigsten anzutreffen ist die **Spektralnorm**, die entsteht, wenn für beide Vektornormen die euklidische Norm genutzt wird:

$$\|A\|_2 := \max_{v\in\mathbb{R}^n\setminus\{0\}} \frac{\|A\,v\|_2}{\|v\|_2}. \tag{3.121}$$

Merke

Man kann

$$\|A\|_2 := \sqrt{\lambda_{\max}}, \tag{3.122}$$

zeigen, wobei $\lambda_{\max}$ der größte Eigenwert von $A^{\mathrm{T}}\,A$ ist.

Die Spektralnorm ist von der gelegentlich auch verwendeten **Frobenius-Norm**

$$\|A\|_{\mathrm{F}} := \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{i,j}^2} \tag{3.123}$$

zu unterscheiden, welche nicht durch Vektornormen induziert wird, aber dennoch die vier Normbedingungen erfüllt.

Die von der 1-Norm oder der ∞-Norm induzierten Matrixnormen spielen nur selten eine Rolle, sind aber deutlich einfacher zu berechnen als die Spektralnorm. Für die 1-Norm kann man

$$\|A\|_1 = \max_{j\in\{1,\dots,n\}} \sum_{i=1}^{m} |a_{i,j}| \tag{3.124}$$

zeigen. Für die ∞-Norm erhält man

$$\|A\|_\infty = \max_{i\in\{1,\dots,m\}} \sum_{j=1}^{n} |a_{i,j}|. \tag{3.125}$$

Analog zu den Vektornormen sind auch alle Matrixnormen zu einander **äquivalent**.

3.8.1.3 Fehlerhafte rechte Seite (Konditionszahl) Betrachten wir nur b als Eingabe, so erhalten wir für den relativen Ausgabefehler bei fehlerhafteter Eingabe $\tilde{b}$ die Abschätzung

$$\frac{\|A^{-1}\,b - A^{-1}\,\tilde{b}\|}{\|A^{-1}\,b\|} \le \|A\|\,\|A^{-1}\|\,\frac{\|b-\tilde{b}\|}{\|b\|}, \tag{3.126}$$

wobei die verwendeten Matrix- und Vektornormen kompatibel zu einander sein sollen, ansonsten aber beliebig gewählt werden können (IDV 380).

Beachte, dass beim bisherigen Konditionsbegriff für Abbildungen nach $\mathbb{R}$ (statt $\mathbb{R}^n$) der Eingabefehler gerade die 1-Norm des Vektors der komponentenweisen relativen Fehler war. Im Kontext linearer Gleichungssysteme ist jedoch die Norm des Vektors der absoluten Fehler geteilt durch die Norm des exakten Eingabevektors als Ausdruck des relativen Eingabefehlers besser handhabbar. Auf der Ausgabeseite wird die gleiche Variante zur Beschreibung des relativen Ausgabefehlers verwendet.

Merke

Die Zahl

$$\kappa(A) := \|A\|\,\|A^{-1}\| \tag{3.127}$$

heißt **Konditionszahl** von A. Diese beschreibt die Auswirkungen des relativen Eingabefehlers auf den relativen Ausgabefehler sowohl bei der Multiplikation mit A als auch bei der Multiplikation mit A^{-1} (Gleichungssystem lösen).

Je nach Wahl der Matrixnorm entstehen unterschiedliche Zahlen $\kappa_1(A)$, $\kappa_2(A)$, $\kappa_\infty(A)$. Wenn im Folgenden kein Index angegeben ist, gelten die Resulte für alle Matrixnormen.

Man kann zeigen, dass stets

$$\kappa(A) = \frac{\max\limits_{\|x\|=1} \|A\,x\|}{\min\limits_{\|x\|=1} \|A\,x\|} \geq 1 \tag{3.128}$$

gilt.

Für die euklidische Matrixnorm kann man noch

$$\kappa_2(A) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}} \tag{3.129}$$

zeigen, wobei $\lambda_{\max}$ und $\lambda_{\min}$ der größte und der kleinste Eigenwert von A sind.

3.8.1.4 Fehlerhafte Matrix und rechte Seite Untersuchen noch die Kondition der Abbildung $(A, b) \mapsto A^{-1}\,b$ bei fehlerbehafteter Matrix $\tilde{A}$ und fehlerbehafteter rechter Seite $\tilde{b}$. Hier muss zunächst geklärt werden, unter welchen Bedingungen $\tilde{A}$ überhaupt noch invertierbar ist.

Merke

Ist $A \in \mathbb{R}^{n\times n}$ invertierbar und gilt

$$\|A - \tilde{A}\| < \frac{1}{\|A^{-1}\|}, \tag{3.130}$$

so kann man zeigen, dass auch $\tilde{A}$ invertierbar ist.

Für den Fehlerzusammenhang zwischen Eingabe und Ausgabe kann man dann

$$\frac{\|A^{-1}\,b - \tilde{A}^{-1}\,\tilde{b}\|}{\|A^{-1}\,b\|} \leq \frac{\kappa(A)}{1 - \frac{\|A-\tilde{A}\|}{\|A\|}\,\kappa(A)} \left(\frac{\|A - \tilde{A}\|}{\|A\|} + \frac{\|b - \tilde{b}\|}{\|b\|} \right) \tag{3.131}$$

zeigen, wobei Matrix- und Vektornormen wieder kompatibel zu einander sein sollen. Bei kleinem Fehler in A und nicht zu großer Kondition $\kappa(A)$ beschreibt also auch hier $\kappa(A)$ die Fehlerverstärkung.

3.8.1.5 Beispiele

3.8.1.5.1 Einheitsmatrix Für

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{3.132}$$

gilt $A = A^{-1}$ und $\|A\| = 1 = \|A^{-1}\|$, also $\kappa(A) = 1$.

3.8.1.5.2 Hilbert-Matrix Für

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2\,n-1} \end{bmatrix} \tag{3.133}$$

wächst $\kappa(A)$ exponentiell mit n, siehe The Condition of the Finite Segments of the Hilbert Matrix.

3.8.2 Wie es nicht geht

Für $A \in \mathbb{R}^{n\times n}$ kann die Lösung x des Gleichungssystems mit der Cramer'schen Regel ermittelt werden: Zur Berechnung von x_i sei A_i wie A, aber mit Spalte i ersetzt durch die rechte Seite b. Dann gilt

$$x_i = \frac{\det A_i}{\det A}, \tag{3.134}$$

wobei

$$\det A = \sum_{\sigma} (\operatorname{sgn}\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{3.135}$$

die Determinate von A ist (analog für A_i). Die Summe läuft hier über alle Permutationen $\sigma : \{1,\ldots,n\} \to \{1,\ldots,n\}$, also alle möglichen Anordnungen der Zahlen $1,\ldots,n$. Der Wert $\operatorname{sgn}\sigma$ ist das Vorzeichen der Permutation und ist entweder 1 oder -1.

Der Aufwand für das Berechnen von Determinanten ist enorm! Er liegt bei $n!$ Additionen und $(n-1)\,n!$ Multiplikationen pro Determinante. Für das Lösen eines Gleichungssystems werden $n+1$ Determinanten benötigt. Für $n = 20$ würde ein 1-Peta-FLOPS-Rechner mehr als 11 Tage für das Lösen mittels Cramer'scher Regel benötigen.

Zusätzlich sind auf der Cramer'schen Regel beruhende Algorithmen durch die vielen Differenzen als instabil anzusehen.

Auch das explizite Berechnen der Inverse A^{-1}, um dann $x = A^{-1}\,b$ zu bekommen, ist im Allgemeinen aufwendiger als das Lösen des Gleichungssystems ohne Verwendung der

Inverse. Der Rechenaufwand für den unten behandelten Gauß-Algorithmus entspricht zwar dem des Invertierens (etwa n^3 Grundoperationen), der Gauß-Algorithmus ist aber stabiler. Praktisch auftretende Gleichungssysteme haben meist beim Lösen vorteilhaft nutzbare Zusatzeigenschaften (symmetrisch, dünn besetzt, Bandstruktur,...), die den Einsatz schnellerer und speichersparenderer Algorithmen zulassen. So ist beispielsweise die im Allgemeinen vollbesetzte Inverse von großen Tridiagonalmatrizen, wie sie häufig in der Praxis auftreten, viel zu groß um sie überhaupt im Arbeitsspeicher ablegen zu können.

Grundsätzlich gilt: Inverse vermeiden. Ausnahmen sind sehr kleine, gut konditionierte Gleichungssysteme, die für viele verschiedene rechte Seiten zu lösen sind (z.B. Koordinatentransformationen in der Computergrafik).

3.8.3 Gauß-Algorithmus

3.8.3.1 Idee Folgende Beobachtung lässt sich zum Lösen linearer Gleichungssysteme einsetzen:

Merke

Sind alle Diagonaleinträge von $A \in \mathbb{R}^{n\times n}$ von Null verschieden, so existieren eine untere Dreiecksmatrix L mit Einsen auf der Diagonale und eine obere Dreiecksmatrix R, sodass

$$A = L\,R. \tag{3.136}$$

Die Zerlegung von A in solche Faktoren L und R ist eindeutig.

Die LR-Zerlegung von A kann leicht algorithmisch bestimmt werden (verzichten hier auf die Details). Liegt diese vor, so erfolgt das Lösen des Gleichungssystems durch **Vorwärtssubstitution** und anschließende **Rückwärtssubstitution**:

Algorithmus

1. Löse $L\,y = b$; dazu setze $y_1 := b_1$ und wiederhole für $i = 2, \ldots, n$:

   a) $y_i := b_i - \sum_{j=1}^{i-1} l_{i,j}\,y_j$.

2. Löse $R\,x = y$; dazu setze $x_n := \frac{y_n}{r_{n,n}}$ und wiederhole für $i = n-1, n-2, \ldots, 1$:

   a) $x_i := \frac{1}{r_{i,i}} \left(y_i - \sum_{j=i+1}^{n} r_{i,j}\,x_j \right)$.

3.8.3.2 Stabilität Sei $\tilde{x}$ die mittels Gauß-Algorithmus aus exakten Eingaben A und b numerisch ermittelte und somit (rundungs-)fehlerbehaftete Lösung. Liegt der relative Fehler zwischen b und $A\,\tilde{x}$ im Bereich des erwarteten Eingabefehlers, so ist der Algorithmus stabil. Die Differenz $b - A\tilde{x}$ heißt auch **Residuum**.

Wie groß das Residuum ist, ist allgemein kaum zu bestimmen. Bei schlecht konditioniertem A wird es jedenfalls groß sein, da die Rundungsfehler aus den ersten Rechenschritten analog zu Eingabefehlern verstärkt werden. Der Gauss-Algorithmus ist somit nicht als stabil anzusehen.

Folgende Punkte fassen die Sachlage bzgl. Stabilität zusammen:

- Theoretisch ist der Gauss-Algorithmus instabil.
- Für gewisse Problemklassen (z.B. Tridiagonalmatrizen) ist er garantiert stabil.
- Praktisch tritt Instabilität nur sehr selten auf, sofern der einfache Gauß-Algorithmus durch eine sogenannte Pivotsuche verbessert wird.
- Für nicht zu große Gleichungssysteme ohne spezielle Struktur ist der Gauß-Algorithmus heute das Standardlösungsverfahren.

3.8.3.3 Zeit- und Speicherbedarf Rechenaufwand für die LR-Zerlegung: ca. n^3 Grundoperationen.

Rechenaufwand für Vorwärts- und Rückwärtssubstitution: ca. n^2 Grundoperationen.

Der Rechenaufwand ist für große Gleichungssysteme also relativ hoch. Ist ein System für mehrere rechte Seiten zu lösen, so muss die LR-Zerlegung allerdings nur einmal berechnet werden. Das Lösen der weiteren Systeme benötigt dann nur noch n^2 Operationen.

Bei geschickter Implementierung benötigt der Algorithmus im Wesentlichen nur den Speicher für das Ablegen der Matrix A, also Größenordnung n^2.

Nebenbemerkung

Den Gauß-Algorithmus sollte man (heute) nicht selbst implementieren, da hochwertige Implementierungen in allen gängigen Software-Biliotheken verfügbar sind. Meistens basieren diese auf LAPACK, einer sehr ausgereiften Bibliothek für lineare Algebra, die in für verschiedene Prozessorarchitekturen optimierten Versionen vorliegt.

3.8.4 Cholesky-Verfahren

Für positiv definite symmetrische Matrizen $A \in \mathbb{R}^{n\times n}$ kann die LR-Zerlegung einfacher ausgedrückt und effizienter berechnet werden.

3.8.4.1 Positiv definite Matrizen Eine symmetrische Matrix heißt **positiv definit**, wenn gilt:

$$v^{\mathrm{T}} A\, v > 0 \quad \text{für alle } v \in \mathbb{R}^n \setminus \{0\}. \tag{3.137}$$

Positiv definite symmetrische Matrizen haben viele vorteilhafte Eigenschaften (siehe Grundlagenliteratur zur linearen Algebra):

- A ist stets invertierbar und A^{-1} ist wieder symmetrisch und positiv definit.
- Alle Eigenwerte von A sind reell und positiv.
- $\det A > 0$.
- Alle Diagonaleinträge sind positiv.
- Der betragsgrößte Eintrag von A liegt auf der Diagonale.

Positiv definite symmetrische Matrizen treten in zahlreichen Anwendungen auf. Beispielsweise kann man leicht zeigen, dass Matrizen der Form $A = B^{\mathrm{T}} B$ für invertierbares $B \in \mathbb{R}^{n,n}$ stets symmetrisch und positiv definit sind. Solche Matrizen treten unter anderem beim Berechnen von Skalarprodukten in transformierten Koordinaten auf. B ist dabei die Koordinatentransformation.

3.8.4.2 Cholesky-Zerlegung Man kann folgende spezielle Form der LR-Zerlegung herleiten:

Merke

Für positiv definites symmetrisches $A \in \mathbb{R}^{n\times n}$ existieren stets eine untere Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$ mit Einsen auf der Diagonale und eine Diagonalmatrix $D \in \mathbb{R}^{n\times n}$ mit positiven Einträgen auf der Diagonale, sodass

$$A = L\, D\, L^{\mathrm{T}}. \tag{3.138}$$

L und D sind eindeutig bestimmt.

Diese sogenannte **Cholesky-Zerlegung** erhält man direkt aus der LR-Zerlegung, wenn man die Symmetrie $A = A^{\mathrm{T}}$ ausnutzt. Insbesondere gilt $R = D\, L^{\mathrm{T}}$.

Allerdings geht es auf direktem Wege etwas schneller (verzichten hier auf die Details).

3.8.4.3 Zeit- und Speicherbedarf Der Rechenaufwand liegt wie bei der LR-Zerlegung in der Größenordnung n^3, ist aber trotzdem nur etwa eine Viertel so groß.

Der Speicherbedarf ist etwa doppelt so hoch wie beim Gauß-Algorithmus, da gewisse Zwischenergebnisse zusätzlich zur Matrix A im Speicher abgelegt werden müssen.

3.8.4.4 Gesamtalgorithmus Ist die Cholesky-Zerlegung bekannt, kann das Gleichungssystem $A\,x = b$ analog zur LR-Zerlegung mittels Vorwärts- und anschließender Rückwärtssubstitution gelöst werden:

Algorithmus

1. Löse $L\,y = b$; dazu setze $y_1 := b_1$ und wiederhole für $i = 2, \ldots, n$:

   a) $y_i := b_i - \sum_{j=1}^{i-1} l_{i,j}\, y_j$.

2. Löse $D\,z = y$; dazu wiederhole für $i = 1, \ldots, n$:

   a) $z_i := \frac{y_i}{d_{i,i}}$.

3. Löse $L^{\mathrm{T}}\,x = z$; dazu setze $x_n := z_n$ und wiederhole für $i = n-1, n-2, \ldots, 1$:

   a) $x_i := z_i - \sum_{j=i+1}^{n} l_{j,i}\, x_j$.

3.8.4.5 Stabilität Man kann zeigen, dass das Cholesky-Verfahren rückwärtsstabil ist (also auch vorwärtsstabil). Insbesondere gilt das Cholesky-Verfahren als stabiler als der Gauss-Algorithmus.

3.8.5 Weitere Verfahren

Wir erwähnen kurz weitere Verfahrensideen um bei Bedarf mit den Begrifflichkeiten vertraut zu sein.

3.8.5.1 QR-Zerlegung Die LR-Zerlegung steht nur für quadratische Matrizen zur Verfügung. Für allgemeine rechteckige Matrizen $A \in \mathbb{R}^{m\times n}$, $m \geq n$ (überbestimmtes Gleichungssystem) kann man jedoch stets die sogenannte QR-Zerlegung $A = Q\,R$ finden. Dabei ist $R \in \mathbb{R}^{m\times n}$ wieder eine obere Dreiecksmatrix, wobei die unteren $m - n$ Zeilen mit Nullen aufgefüllt werden, und $Q \in \mathbb{R}^{m\times m}$ eine orthogonale Matrix. Die Spalten von Q sind also orthogonal zu einander und haben Norm 1. Für orthogonale Matrizen gilt stets $Q^{-1} = Q^{\mathrm{T}}$, sodass sich das Lösen des Gleichungssystems $A\,x = b$ auf Rückwärtseinsetzen und Multiplikation mit Q^{T} reduziert.

Für quadratische Matrizen ($m = n$) kann man die QR-Zerlegung mit etwa n^3 Grundoperationen berechnen.

3.8.5.2 Iterative Verfahren Iterative Verfahren liefern im Gegensatz zum Gauß-Algorithmus und zum Cholesky-Verfahren nicht die (bis auf Rundungsfehler) exakte Lösung des Gleichungssystems $A\,x = b$, sondern nur Näherungslösungen. Die Verfahren berechnen zunächst eine sehr grobe Näherung, die dann Schritt für Schritt verfeinert wird. Dieser Ansatz erlaubt deutlich geringere Rechenzeiten bei sehr großen Gleichungssystemen.

3.8.5.2.1 Richardson-Iteration Für die Lösung x des Gleichungssystems und eine beliebige reelle Zahle $\tau > 0$ gilt

$$x = x - \tau\,(A\,x - b) = (I - \tau\,A)\,x + \tau\,b. \tag{3.139}$$

Dies ist eine sogenannte **Fixpunktgleichung**, da das Auswerten der rechten Seite wieder auf x führt. Daraus kann man die Iterationsvorschrift

$$x_k := (I - \tau\,A)\,x_{k-1} + \tau\,b \tag{3.140}$$

ableiten. Man kann zeigen, dass für hinreichend kleines τ die Folge $x_0 := 0, x_1, x_2, \ldots$ gegen die Lösung des Gleichungssystems konvergiert. Je nach gewünschter Genauigkeit kann die Iteration früher oder später abgebrochen werden.

Die Iterationsvorschrift kann zu

$$x_k := M\,x_{k-1} + c \tag{3.141}$$

verallgemeinert werden, wobei $M \in \mathbb{R}^{n\times n}$ und $c \in \mathbb{R}^n$ von A und b abhängen. So erhält man eine Vielzahl verschiedener Lösungsverfahren, die je nach Eigenschaften von A vorteilhafte Eigenschaften wie beispielsweise besonders schnelle Konvergenz haben können. Die Konvergenz der Iterierten zur exakten Lösung ist dabei keinesfalls garantiert, sondern kann nur unter recht engen Voraussetzungen an M und c abgesichert werden.

3.8.5.2.2 CG-Verfahren Für das Verfahren der konjugierten Gradienten (kurz: CG-Verfahren, CG = conjugate gradients) wird das Gleichungssystem durch das Minimierungsproblem

$$J(x) := \frac{1}{2}\,x^{\mathrm{T}}\,A\,x - x^{\mathrm{T}}\,b \to \min_{x\in\mathbb{R}^n} \tag{3.142}$$

ersetzt. Man kann zeigen, dass die Lösung des Minimierungsproblems mit der Lösung des Gleichungssystems übereinstimmt.

Zur Lösung des Minimierungsproblems stehen eine Vielzahl numerischer Optimierungsverfahren zur Verfügung. Das einfachste ist das so genannte Gradienten-Verfahren, welches, beginnend an einem Punkt x_0, den Gradient ∇J der Zielfunktion J berechnet und dann die aktuelle Position in Richtung des negativen Gradienten (Richtung des steilsten Abstiegs!) verändert:

$$x_k := x_{k-1} - s_k\,\nabla J(x_{k-1}). \tag{3.143}$$

Dabei ist s_k die Schrittweite, welche auf verschiedene Arten gewählt werden kann, z.B. konstant. Für den Gradient erhält man

$$\nabla J(x) = A\,x - b. \tag{3.144}$$

Wählt man s_k stets so, dass $J(x_{k-1} - s\,\nabla J(x_{k-1}))$ für $s = s_k$ minimal wird, so erhält man das CG-Verfahren, welches besonders schnell konvergiert. Für konstantes s_k erhält man hingegen das Verfahren des steilsten Abstiegs, welches verhältnismäßig langsam konvergiert.

3.8.5.2.3 Verfahren für dünn besetzte Matrizen In zahlreichen Anwendungen treten dünn besetzte (englisch: sparse) sehr große Matrizen $A \in \mathbb{R}^{n\times n}$ auf, also Matrizen, die nur wenige von Null verschiedene Einträge haben. Die Anzahl der Nicht-Null-Einträge ist üblicherweise ein kleines Vielfaches von n. Diese Matrizen speichert man nicht als Block aus n^2 Zahlen, sondern als Liste der Nicht-Null-Einträge und der zugehörigen Zeilen- und Spalten-Indizes.

Beispielsweise sind Adjazenzmatrizen von Graphen oft dünnbesetzt (und groß). Auch beim numerischen Lösen vieler naturwissenschaftlicher Probleme treten große dünnbesetzte Matrizen auf.

Der Zugriff auf einen durch Zeilen- und Spaltenindex gegebenen Matrixeintrag ist bei dünn besetzten Matrizen mit hohem Aufwand verbunden (Liste durchsuchen!). Matrix-Vektor-Produkte können jedoch schnell berechnet werden (IDV 385). Somit sind iterative Lösungsverfahren gegenüber Gauß-Algorithmus und Cholesky-Verfahren hier klar im Vorteil, da iterative Verfahren meist nur Matrix-Vektor-Produkte auswerten und keine anderweitigen Zugriffe auf die Matrixeinträge tätigen.

Heute übliche Verfahren für große dünn besetzte Gleichungssysteme sind Krylow-Unterraum-Verfahren (Verallgemeinerung des CG-Verfahrens) und Mehrgitterverfahren.

3.8.5.3 Vorkonditionierung Die Konvergenz iterativer Lösungsverfahren kann deutlich beschleunigt werden, wenn das Gleichungssystem im Vorfeld geeignet äquivalent umgeformt wird. Auch für nicht iterative Verfahren kann eine geeignete Umformung zur Verbesserung der Matrixkondition führen. Eine übliche Umformung ist die Multiplikation mit einer geeignet gewählten Matrix $M \in \mathbb{R}^{n\times n}$:

$$M\,A\,x = M\,b. \tag{3.145}$$

Diese Matrix soll leicht zu beschaffen sein und dabei die Inverse A^{-1} möglichst gut annähern. Wie M konkret zu wählen ist, hängt auch vom eingesetzten Lösungsverfahren ab.

Eine einfache Technik zur Vorkonditionierung ist die **Äquilibrierung**: Jede Zeile des Gleichungssystems wird durch die Norm der entsprechenden Zeile der Systemmatrix als

Vektor geteilt. Bei Verwendung der euklidischen Norm also

$$M = \begin{bmatrix} \left(\sum_{j=1}^{n} a_{1,j}^2\right)^{-\frac{1}{2}} & & 0 \\ & \ddots & \\ 0 & & \left(\sum_{j=1}^{n} a_{n,j}^2\right)^{-\frac{1}{2}} \end{bmatrix}. \tag{3.146}$$

4 Modellierung

Beispiel für übliche mathematische Modelle zu Aufgabenstellungen aus Technik und Naturwissenschaften.

4.1 Wiederholung Differentialrechnung

4.1.1 Partielle Ableitungen

Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Funktion von n Veränderlichen $x := (x_1, \ldots, x_n)$. Bezeichnen die zugehörigen ersten **partiellen Ableitungen** mit $f_{x_1}, \ldots, f_{x_n}$ oder (seltener) mit

$$\frac{\partial}{\partial x_1} f, \ldots, \frac{\partial}{\partial x_n} f. \tag{4.1}$$

Der Vektor

$$\nabla f(x) := \begin{bmatrix} f_{x_1}(x) \\ \vdots \\ f_{x_n}(x) \end{bmatrix} \tag{4.2}$$

heißt **Gradient** von f.

Ist $r \in \mathbb{R}^n \setminus 0$ ein Vektor mit $\|r\| = 1$, so ist

$$\frac{\partial}{\partial r} f(x) := r_1\, x_1 + \cdots r_n\, x_n \tag{4.3}$$

die **Richtungsableitung** von f in Richtung r an der Stelle x.

Die Matrix

$$H_f(x) := \begin{bmatrix} f_{x_1x_1} & \cdots & f_{x_1x_n} \\ \vdots & & \vdots \\ f_{x_nx_1} & \cdots & f_{x_nx_n} \end{bmatrix} \in \mathbb{R}^{n \times n} \tag{4.4}$$

der zweiten partiellen Ableitungen heißt **Hesse-Matrix**. Sie ist in allen praktisch relevanten Fällen symmetrisch.

Ist $f : \mathbb{R}^n \to \mathbb{R}^m$ eine vektorwertige Funktion mit Komponentenfunktionen $f_1, \ldots, f_m$, also

$$f = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}, \tag{4.5}$$

so bezeichnen wir mit

$$J_f(x) := \begin{bmatrix} (f_1)_{x_1} & \cdots & (f_1)_{x_n} \\ \vdots & & \vdots \\ (f_m)_{x_1} & \cdots & (f_m)_{x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{4.6}$$

die **Jacobi-Matrix** zu f an der Stelle x. Die Zeilen der Jacobi-Matrix sind also gerade die Gradienten der Komponentengfunktionen.

4.1.2 Differentialoperatoren

Gewisse Kombinationen von partiellen Ableitungen treten bei der mathematischen Modellierung von praktischen Fragestellungen besonders häufig auf und bekommen deshalb eine eigene Kurzschreibweise.

Sei $f : \mathbb{R}^3 \to \mathbb{R}^3$ ein Vektorfeld. Definieren das Skalarfeld

$$(\operatorname{div} f)(x, y, z) := \frac{\partial}{\partial x} f_1(x, y, z) + \frac{\partial}{\partial y} f_2(x, y, z) + \frac{\partial}{\partial z} f_3(x, y, z) \tag{4.7}$$

(**Divergenz** von f) und das Vektorfeld

$$(\operatorname{rot} f)(x, y, z) := \begin{bmatrix} \frac{\partial}{\partial y} f_3(x, y, z) - \frac{\partial}{\partial z} f_2(x, y, z) \\ \frac{\partial}{\partial z} f_1(x, y, z) - \frac{\partial}{\partial x} f_3(x, y, z) \\ \frac{\partial}{\partial x} f_2(x, y, z) - \frac{\partial}{\partial y} f_1(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \times \begin{bmatrix} f_1(x, y, z) \\ f_2(x, y, z) \\ f_3(x, y, z) \end{bmatrix} \tag{4.8}$$

(**Rotation** von f).

Die Divergenz kann als Quellendichte interpretiert werden. In einer Strömung gibt sie beispielsweise an, ob an einem Punkt Material zum durchströmenden Material hinzugefügt wird (positive Divergenz) oder ob Material entnommen wird (negative Divergenz). Siehe Divergenz eines Vektorfeldes, koordinatenfreie Darstellung für eine präzise Formulierung dieser Interpretation.

Die Rotation gibt Drehachse und Drehgeschwindigkeit eines in der Strömung mitschwimmenden Objekts an. Vektorfelder, deren Rotation überall Null ist, heißen wirbelfrei.

Für ein Skalarfeld $f : \mathbb{R}^3 \to \mathbb{R}$ setzen wir noch

$$(\Delta f)(x, y, z) := \frac{\partial^2}{\partial^2 x} f(x, y, z) + \frac{\partial^2}{\partial^2 y} f(x, y, z) + \frac{\partial^2}{\partial^2 z} f(x, y, z) \tag{4.9}$$

(**Laplace-Operator** angewendet auf f). Funktionen, die $\Delta f(x, y, z) = 0$ für alle relevanten x, y, z erfüllen, heißen **harmonisch**. Solche Funktionen treten in Physik und Technik in vielen verschiedenen Kontexten auf (Elektrostatik, Wärmeleitung, Elastizitätslehre,...).

Offensichtlich gilt

$$\Delta f(x, y, z) = \operatorname{div} \nabla f(x, y, z) \tag{4.10}$$

4.1.3 Kurven

Eine Kurve $K \subseteq \mathbb{R}^3$ im Raum ist eine Teilmenge des Raumes, die in der Form

$$K = \big\{ \big(x(t), y(t), z(t)\big) : \ t \in [a, b] \big\} \tag{4.11}$$

mit dem **Parameterintervall** $[a, b]$ und den **Koordinatenfunktionen**

$$x : [a, b] \to \mathbb{R}, \quad y : [a, b] \to \mathbb{R}, \quad z : [a, b] \to \mathbb{R} \tag{4.12}$$

geschrieben werden kann. Meistens entspricht das Parameterintervall einem Zeitintervall, sodass die Kurve jedem Zeitpunkt eine Position im Raum zuordnet.

Beispiel (Schraubenlinie)

Die durch

$$x(t) = \cos(2\,\pi\,t), \qquad y(t) = \sin(2\,\pi\,t), \qquad z(t) = t, \qquad t \in [0, 3] \tag{4.13}$$

gegebene Kurve ist eine Schraubenlinie entlang der z-Achse mit drei Windungen.

Eine Kurve K heißt differenzierbar, wenn die Koordinatenfunktionen differenzierbar sind. Der Vektor

$$\begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \end{bmatrix} \tag{4.14}$$

ist gerade der **Tangentialvektor** an die Kurve K im Punkt $(x(t), y(t), z(t))$. Beschreibt die Kurve die Position eines Teilchens im Laufe der Zeit, so entspricht der Tangentialvektor dem Geschwindigkeitsvektor (Richtung und Betrag). Der Vektor der zweiten Ableitungen kann dann als Beschleunigungsvektor (Richtung und Betrag) interpretiert werden.

Beispiel (Newton'sches Gesetz)

Bewegt sich ein Massepunkt der Masse m entlang einer durch die Koordinatenfunktionen x, y, z gegebenen Kurve durch ein zeitabhängiges Kraftfeld $F : \mathbb{R}^4 \to \mathbb{R}^3$ so gilt zu jedem Zeitpunkt t das Newton'sche Gesetz

$$F\big(x(t), y(t), z(t), t\big) = m \begin{bmatrix} x''(t) \\ y''(t) \\ z''(t) \end{bmatrix} \tag{4.15}$$

("Kraft = Masse $\cdot$ Beschleunigung").

Insbesondere kann man bei gegebenem F, gegebener Anfangsposition $(x(0), y(0), z(0))$ und gegebener Anfangsgeschwindigkeit $(x'(0), y'(0), z'(0))$ aus diesem Zusammenhang die Koordinatenfunktionen und damit den weiteren Bewegungsverlauf berechnen (vgl. Kapitel zu gewöhnlichen Differentialgleichungen).

4.1.4 Flächen

Eine Fläche $S \subseteq \mathbb{R}^3$ im Raum ist eine Teilmenge des Raumes, die in der Form

$$S = \big\{\big(x(u, v), y(u, v), z(u, v)\big) :\ u \in [a, b],\, v \in [v_1(u), v_2(u)]\big\} \tag{4.16}$$

mit den Parametern u, v und den Koordinatenfunktionen x, y, z geschrieben werden kann. Dabei können die Grenzen des Parameters v von u abhängen (IDV 405).

Sind die Koordinatenfunktionen differenzierbar, so kann man in jedem $(x_0, y_0, z_0) = (x(u_0, v_0), y(u_0, v_0), y(u_0, v_0))$ der Fläche eine Ebene mit gleicher Neigung wie die Fläche anlegen, die sogenannte **Tangentialebene**. Sie besteht aus allen Punkten (ξ, η, ζ) des Raumes, die folgende Gleichung erfüllen:

$$0 = \left(\begin{bmatrix} x_u(u_0, v_0) \\ y_u(u_0, v_0) \\ z_u(u_0, v_0) \end{bmatrix} \times \begin{bmatrix} x_v(u_0, v_0) \\ y_v(u_0, v_0) \\ z_v(u_0, v_0) \end{bmatrix} \right)^{\mathrm{T}} \begin{bmatrix} \xi - x_0 \\ \eta - y_0 \\ \zeta - z_0 \end{bmatrix}. \tag{4.17}$$

Der Vektor

$$\begin{bmatrix} x_u(u_0, v_0) \\ y_u(u_0, v_0) \\ z_u(u_0, v_0) \end{bmatrix} \times \begin{bmatrix} x_v(u_0, v_0) \\ y_v(u_0, v_0) \\ z_v(u_0, v_0) \end{bmatrix} \tag{4.18}$$

steht senkrecht auf der Tangentialebene und damit auch senkrecht auf der Fläche. Er wird **Normalenvektor** genannt. Jedes Vielfache dieses Vektors ist ebenfalls ein Normalenvektor.

Um eine gewisse Eindeutigkeit des Normalenvektors zu erreichen, verwendet man üblicherweise Normalenvektoren der Länge 1. Dann gibt es nur noch zwei Möglichkeiten diesen zu wählen. Beide unterscheiden sich nur im Vorzeichen. Welchen der beiden man für gewisse Zwecke (siehe später) wählt, ist egal. Jedoch muss man darauf achten, dass man in jedem Punkt der Fläche die gleiche Richtung für den Normalenvektor wählt, also alle Normalenvektoren auf der gleichen Seite der Fläche liegen.

Allerdings gibt es Flächen, bei denen man die Normalenvektoren nicht in einheitlicher Richtung wählen kann, z.B. das Möbiusband. Solche Flächen heißen *nicht orientierbar*. Wir betrachten stets nur **orientierbare Flächen**. Sind die Koordinatenfunktionen stetig (!) differenzierbar und gilt

$$\begin{bmatrix} x_u(u, v) \\ y_u(u, v) \\ z_u(u, v) \end{bmatrix} \times \begin{bmatrix} x_v(u, v) \\ y_v(u, v) \\ z_v(u, v) \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{für alle } u, v, \tag{4.19}$$

so ist die Fläche orientierbar.

Beispiel (Funktionsgraph)

Für eine stetig differenzierbare Funktion $f : \mathbb{R}^2 \to \mathbb{R}$ bildet der Funktionsgraph eine Fläche. Diese kann durch

$$x(u, v) := u, \quad y(u, v) := v, \quad z(u, v) := f(u, v), \quad u \in \mathbb{R}, \quad v \in \mathbb{R} \tag{4.20}$$

parametrisiert werden.

Als Normalenvektor erhält man

$$\begin{bmatrix} x_u(u,v) \\ y_u(u,v) \\ z_u(u,v) \end{bmatrix} \times \begin{bmatrix} x_v(u,v) \\ y_v(u,v) \\ z_v(u,v) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ f_x(u,v) \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ f_y(u,v) \end{bmatrix} = \begin{bmatrix} -f_x(u,v) \\ -f_y(u,v) \\ 1 \end{bmatrix}. \tag{4.21}$$

Die Tangentialebene in einem Punkt $(x_0, y_0, f(x_0, y_0))$ ist entsprechend durch

$$0 = \begin{bmatrix} -f_x(x_0,y_0) \\ -f_y(x_0,y_0) \\ 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \xi - x_0 \\ \eta - y_0 \\ \zeta - f(x_0,y_0) \end{bmatrix} \tag{4.22}$$

bzw.

$$\zeta = f(x_0,y_0) + f_x(x_0,y_0)\,(\xi - x_0) + f_y(x_0,y_0)\,(\eta - y_0) \tag{4.23}$$

beschrieben.

4.2 Wiederholung Integralrechnung

Neben dem gewöhnlichen Integral

$$\int_a^b f(x)\,\mathrm{d}x \tag{4.24}$$

einer Funktion $f : [a, b] \to \mathbb{R}$ über einem Intervall $[a, b]$ gibt es auch Integralbegriffe für Funktionen mehrerer Veränderlicher zur Berechnung von Kurvenlängen, Inhalten gekrümmter Flächen, Volumen unregelmäßiger Körper und vielem mehr. Diese Integrale sind in ihrer Definition meist recht komplex, können aber stets auf gewöhnliche Integrale von Funktioner einer Veränderlichen zurückgeführt werden. Sammeln hier die entsprechenden Formeln, sodass einer Auswertung der Integrale mittels einfacher numerischer Integration nichts im Wege steht.

4.2.1 Bereichsintegrale für ebene Bereiche

Eine Teilmenge $B \subseteq \mathbb{R}^2$ der Ebene heißt **Normalbereich bzgl. der x-Achse**, wenn sie in der Form

$$B = \{(x, y) \in \mathbb{R}^2 :\ a \le x \le b,\ g(x) \le y \le h(x)\} \tag{4.25}$$

mit einem Intervall $[a, b]$ und Funktionen $g, h : [a, b] \to \mathbb{R}$ geschrieben werden kann.

Hat B die Form

$$B = \{(x, y) \in \mathbb{R}^2 :\ c \le y \le d,\ p(y) \le x \le q(y)\}, \tag{4.26}$$

so handelt es sich um einen **Normalbereich bzgl. der y-Achse**.

Eine gegebene Menge kann gleichzeitig ein Normalbereich bzgl. der x-Achse und bzgl. der y-Achse sein.

Ist B ein Normalbereich bzgl. der x-Achse und ist $f : B \to \mathbb{R}$ eine Funktion, so bezeichnen wir

$$\int_B f\mathrm{db} := \int_a^b \int_{g(x)}^{h(x)} f(x, y)\,\mathrm{d}y\,\mathrm{d}x \tag{4.27}$$

als **Bereichsintegral** von f über dem Bereich B.

Völlig analog sehen Bereichsintegrale für Normalbereiche bzgl. der y-Achse aus.

Bereichsintegrale können als Volumen unter dem Funktionsgraph interpretiert werden. Alternativ liefert das Bereichsintegral die Masse der Fläche B, wenn f die Dichte (Masse pro Flächeneinheit) der Fläche beschreibt.

4.2.2 Bereichsintegrale für räumliche Bereiche

Analog zu ebenen Normalbereichen kann man räumliche Normalbereiche einführen. Jede Reihung der Koordinatenachsen liefert einen anderen Typ von Normalbereich (insgesamt

sechs Stück). Betrachten hier nur Normalbereiche bzgl. x-Achse, dann y-Achse:

$$B = \{(x, y, z) \in \mathbb{R}^3 : \ a \le x \le b, \ g(x) \le y \le h(x), \ p(x, y) \le z \le q(x, y)\} \tag{4.28}$$

mit Zahlen a, b und Funktionen g, h, p, q. Räumliche Normalbereiche entstehen also aus ebenen Normalbereichen, die in z-Richtung zunächst zu einem unendlich langen Prisma erweitert. Dieses Prisma wird dann dann durch die Funktionen p und q "abgeschnitten".

Das zugehörige Bereichsintegral kann als Schachtelung von drei gewöhnlichen eindimensionalen Integralen ausgedrückt werden:

$$\int_B f \mathrm{db} := \int_a^b \int_{g(x)}^{h(x)} \int_{p(x,y)}^{q(x,y)} f(x, y, z) \, \mathrm{d}z \, \mathrm{d}y \, \mathrm{d}x. \tag{4.29}$$

Bereichsintegrale liefern das Volumen des Bereichs B, wenn die Funktion f überall 1 ist. Interpretiert man f als Dichte (Masse pro Volumen), so liefert das Bereichsintegral gerade die Masse des Bereichs.

4.2.3 Kurvenintegrale erster Art

Sei durch

$$x(t), y(t), \quad t \in [a, b] \tag{4.30}$$

eine Kurve K in der Ebene gegeben. Weiter sei $f : \mathbb{R}^2 \to \mathbb{R}$ eine Funktion. Bezeichnen mit

$$\int_K f \, \mathrm{ds} := \int_a^b f(x(t), y(t)) \, \sqrt{x'(t)^2 + y'(t)^2} \, \mathrm{d}t \tag{4.31}$$

das **Kurvenintegral erster Art** von f entlang der Kurve K.

Für Kurven im Raum erfolgt die Definition völlig analog.

Ist f überall 1, so liefert das Kurvenintegral erster Art gerade die Kurvenlänge. Interpretiert man f als Dichte (Masse pro Länge), so liefert das Kurvenintegral erster Art die Masse der Kurve.

4.2.4 Kurvenintegrale zweiter Art

Sei durch

$$x(t), y(t), \quad t \in [a, b] \tag{4.32}$$

eine Kurve K in der Ebene gegeben. Weiter sei $f : \mathbb{R}^2 \to \mathbb{R}^2$ eine Funktion. Bezeichnen mit

$$\int_K f \, \mathrm{dv} := \int_a^b f_1(x(t), y(t)) \, x'(t) + \ f_2(x(t), y(t)) \, y'(t) \ \mathrm{d}t \tag{4.33}$$

das **Kurvenintegral zweiter Art** von f entlang der Kurve K.

Für Kurven im Raum erfolgt die Definition völlig analog.

Ist f ein Kraftfeld, so liefert das Kurvenintegral zweiter Art die Arbeit, die beim Durchlaufen des Kraftfelds entlang der Kurve verrichtet wird.

Offensichtlich gilt

$$\int_K f \,\mathrm{dv} = \int_K f^{\mathrm{T}} \frac{\tau}{\|\tau\|} \,\mathrm{ds} \tag{4.34}$$

mit dem Tangentialvektor

$$\tau(x(t), y(t)) := \begin{bmatrix} x'(t) \\ y'(t)' \end{bmatrix}. \tag{4.35}$$

4.2.5 Oberflächenintegrale erster Art

Sei durch

$$S = \big\{ \big(x(u,v), y(u,v), z(u,v)\big) : \ u \in [a,b],\ v \in [v_1(u), v_2(u)] \big\} \tag{4.36}$$

eine Fläche im Raum gegeben und sei $f : S \to \mathbb{R}$ eine Funktion ein auf dieser Fläche definierte Funktion.

Bezeichnen mit

$$\int_S f \,\mathrm{da} := \int_a^b \int_{v_1(x)}^{v_2(x)} f(x(u,v), y(u,v), z(u,v))\, k(u,v) \,\mathrm{d}v \,\mathrm{d}u \tag{4.37}$$

das **Oberflächenintegral erster Art** von f über der Fläche S. Dabei ist

$$k(u,v) := \left\| \begin{bmatrix} x_u(u,v) \\ y_u(u,v) \\ z_u(u,v) \end{bmatrix} \times \begin{bmatrix} x_v(u,v) \\ y_v(u,v) \\ z_v(u,v) \end{bmatrix} \right\| \tag{4.38}$$

die Länge des durch das Kreuzprodukt gegebenen Normalenvektors.

Ist f überall 1, so liefert das Oberflächenintegral erster Art den Flächeninhalt der Fläche S. Interpretiert man f als Dichte (Masse pro Fläche), so erhält man mit dem Integral die Masse der Fläche.

4.2.6 Oberflächenintegrale zweiter Art

Sei durch

$$S = \big\{ \big(x(u,v), y(u,v), z(u,v)\big) : \ u \in [a,b],\ v \in [v_1(u), v_2(u)] \big\} \tag{4.39}$$

eine Fläche im Raum gegeben und sei $f : \mathbb{R}^3 \to \mathbb{R}^3$ ein Vektorfeld (welches üblicherweise im ganzen Raum gegeben ist, formal aber nur auf S benötigt wird).

Bezeichnen mit

$$\int_S f \,\mathrm{dn} := \int_a^b \int_{v_1(x)}^{v_2(x)} f(x(u,v), \ldots)^{\mathrm{T}}\, n(x(u,v), \ldots) \,\mathrm{d}v \,\mathrm{d}u \tag{4.40}$$

das **Oberflächenintegral zweiter Art** von f über der Fläche S. Dabei ist

$$n(x(u,v),y(u,v),z(u,v)) := \begin{bmatrix} x_u(u,v) \\ y_u(u,v) \\ z_u(u,v) \end{bmatrix} \times \begin{bmatrix} x_v(u,v) \\ y_v(u,v) \\ z_v(u,v) \end{bmatrix} \tag{4.41}$$

ein spezieller Normalenvektor an die Fläche S im Punkt $x(u,v), y(u,v), z(u,v)$.

Beschreibt f einen Materialstrom, so liefert das Oberflächenintegral zweiter Art die Menge des pro Zeiteinheit durch die Fläche S strömenden Materials. Ist S eine geschlossene Fläche (z.B. Kugeloberfläche) und zeigen die Normalenvektoren nach außen, so liefert das Integral die Materialmenge, die aus dem umschlossenen Bereich durch die Strömung heraus transportiert wird. Ein negativer Wert entspricht einem Transport in den Bereich hinein.

Offensichtlich gilt

$$\int_S f \,\mathrm{dn} = \int_S f^{\mathrm{T}} \frac{n}{\|n\|} \,\mathrm{da}. \tag{4.42}$$

4.2.7 Integralsatz von Gauß

Seien $B \subseteq \mathbb{R}^3$ und $f : \mathbb{R}^3 \to \mathbb{R}^3$ hinreichend regulär. Dann gilt

$$\int_B \operatorname{div} f \,\mathrm{db} = \int_{\partial B} f \,\mathrm{dn}, \tag{4.43}$$

wobei ∂B die Oberfläche von B bezeichne. Die Summe aller Quellen und Senken im Gebiet ist also gleich dem Fluss über den Rand.

Für zweidimensionale Bereiche und Vektorfelder gilt analog

$$\int_B \operatorname{div} f \,\mathrm{db} = \int_{\partial B} f^{\mathrm{T}} \frac{n}{\|n\|} \,\mathrm{ds} \tag{4.44}$$

mit dem Normalenvektor

$$n(x(t),y(t)) := \begin{bmatrix} y'(t) \\ -x'(t) \end{bmatrix} \tag{4.45}$$

an die durch die Koordinatenfunktionen x und y parametrisierte Kurve ∂B.

4.3 Brachistochrone

4.3.1 Problemstellung

Eine Kugel soll sich aus "eigener" Kraft (Schwerkraft) entlang einer Bahn von einem höher gelegenen Startpunkt zu einem niedriger gelegenen Zielpunkt bewegen. Wie ist die Bahn zu Formen, damit die Kugel in möglichst kurzer Zeit den Zielpunkt erreicht? Diese optimale Bahn heißt Brachistochrone.

4.3.2 Modell

Sind (x_1, y_1) und (x_2, y_2) Start- und Zielpunkt, so kann die Bahn als Funktion $h : [x_1, x_2] \to \mathbb{R}$ beschrieben werden, insbesondere gilt also $h(x_1) = y_1$ und $h(x_2) = y_2$. Über Betrachtungen zur Energieerhaltung kann man zeigen, dass die benötigte Zeit $T(h)$ für das Durchlaufen der Bahn h mittels

$$T(h) = \frac{1}{\sqrt{2\,g}} \int_{x_1}^{x_2} \sqrt{\frac{1 + h'(x)^2}{-h(x)}}\,\mathrm{d}x \tag{4.46}$$

berechnet werden kann. Dabei ist g die wirkende Fallbeschleunigung. Die optimale Bahn ist somit als Lösung des Minimierungsproblems

$$T(h) \to \min_{h \in H} \tag{4.47}$$

gegeben, wobei H die Menge aller stetig differenzierbaren Funktionen mit $h(x_1) = y_1$ und $h(x_2) = y_2$ sein soll.

4.3.3 Untersuchung des Modells

An diesem recht einfachen Problem und seiner Modellierung treten schon verschiedene allgemein relevante Aspekte zutage:

- Obwohl das Anwendungsproblem im dreidimensionalen Raum formuliert ist, arbeitet das Modell im zweidimensionalen Raum. Dass diese **Vereinfachung** zulässig ist, also keinen (wie hier) oder nur einen vernachlässigbar kleinen Fehler verursacht, muss man bei der Modellierung prüfen!
- Die Formel für $T(h)$ nimmt keine Rücksicht auf die Tatsache, dass die Kugel auf der Bahn rollen und nicht gleiten wird. Man kann sich auch hier überlegen, dass diese Vereinfachung keinen Einfluss auf die Lösung hat.
- Die Vernachlässigung der Reibung zwischen Kugel und Bahn sowie zwischen Kugel und Luft wird hingegen einen Fehler verursachen. Dieser sollte (!) vernachlässigbar sein, wenn die Kugel schwer und sowohl Kugel als auch Bahn sehr glatt sind. Diese Aussage ist keine exakte Fehlerbetrachtung, sondern eine bei der Modellierung leider manchmal nötige **Abschätzung nach "Augenmaß"**.

- Wir haben als Wertebereich für h alle reellen Zahlen zugelassen, obwohl man in Versuchung kommen könnte, nur das Intervall $[y_2, y_1]$ zuzulassen. Es wird sich allerdings herausstellen, dass für die optimale Bahn auch $h(x) < y_2$ für manche x gelten kann. Also: **Vereinfachungen kritisch auf Zulässigkeit prüfen**.
- Im Integrand wird durch $h(x)$ geteilt, der Integrand hat also eine Polstelle. Da das Integral minimiert wird, wird die Polstelle der Lösung aber so beschaffen sein, dass ein endlicher Wert für das Integral entsteht. **Modelle auf mathematische Korrektheit prüfen**!
- Das Modell wird als Optimierungsproblem über einer Menge von Funktionen formuliert. Diese Art mathematischer Probleme taucht in den Grundveranstaltungen zur Mathematik üblicherweise nicht auf. Alle praktisch relevanten und interessanten Probleme erfordern **zusätzliche Mathekenntnisse**!

Das gefundene Minimierungsproblem als Modell für die Berechnung der Brachistochrone lässt sich sogar analytisch lösen, muss also nicht zwingend numerisch gelöst werden. Das Problem gehört aber zu einer allgemeineren Klasse von Problemen, die sich nicht immer analytisch lösen lassen:

$$\int_{x_1}^{x_2} F\big(h(x), h'(x), x\big)\,\mathrm{d}x \to \min_{h \in H} \tag{4.48}$$

mit einer das konkrete Problem definierenden Funktion $F : \mathbb{R}^3 \to \mathbb{R}$, wobei $h(x_1)$ und $h(x_2)$ gegeben sind.

4.3.4 Hinweise zum numerischen Lösen

Werden aus Zeitgründen das Brachistrochronen-Problem nicht numerisch lösen, sondern geben nur einige Hinweise zum Vorgehen.

- Die Menge aller stetig differenzierbaren Funktionen ist nicht mit dem Computer darstellbar. Somit müssen wir uns auf eine hinreichend repräsentative, aber **endlichdimensionale Menge von Funktionen** als Suchraum bei der Optimierung beschränken. Kandidaten sind Polynome oder stückweise Polynome, jedoch auch eine ganze Reihe anderer Ansätze.
- Man kann zeigen, dass die optimale Bahn $h'(x_1) = -\infty$ erfüllt. Mit Polynomen wird man diese Bedingung nie erfüllen können. Verwendet man stückweise Polynome, so sollten die Teilintervalle in der Nähe von x_1 sehr klein sein, damit der starke Abfall von h dort gut angenähert werden kann.
- Obwohl die gesuchte Funktion differenzierbar ist, können auch nicht differenzierbare Ansätze wie stückweise lineare Funktionen zum Ziel führen (**keep it simple**). Dabei ist die Frage zu klären, was eigentlich **das Ziel** ist. Geht es nur um die Visualisierung der optimalen Bahn? Dann reicht eine stückweise lineare Näherungslösung aus. Oder soll die optimale Bahn in Folgeprozessen verwendet werden, die auf Ableitungen beruhen? Dann sollte auch die Näherungslösung differenzierbar

sein.

- Das entstehende Optimierungsproblem kann meist mit Standardverfahren gelöst werden. Welcher Algorithus geeignet ist, hängt von der konkreten Form des Optimierungsproblems ab, also vor allem von der gewählten Diskretisierung.

4.4 Computertomografie

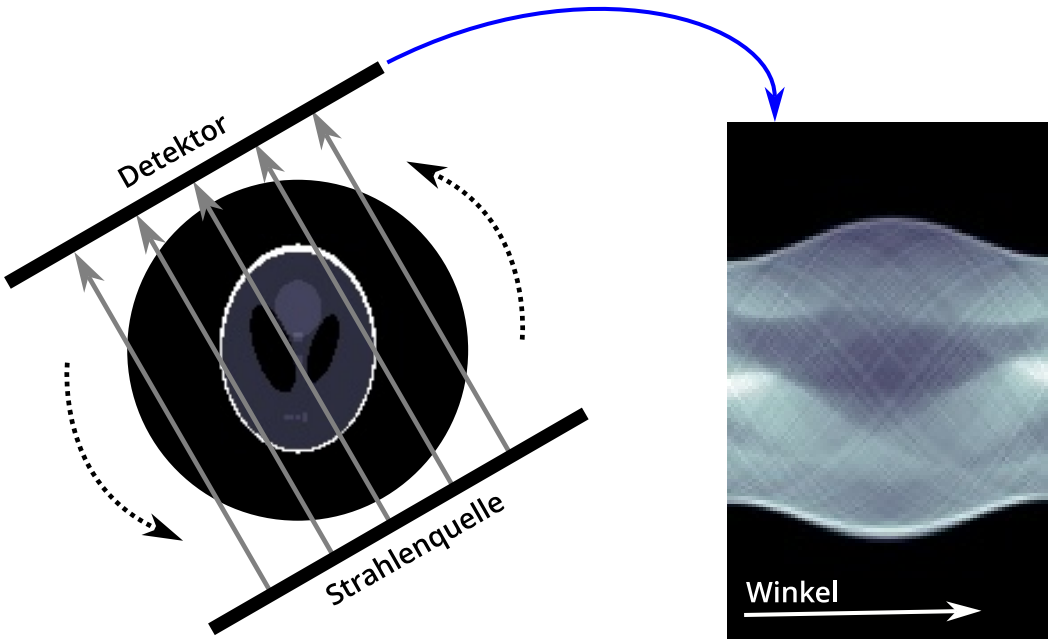
Die Computertomografie (CT) ist ein weit verbreitetes medizinisches und industrielles Bildgebungsverfahren.

4.4.1 Funktionsweise

Grundidee:

1. Röntgenbilder aus vielen verschiedenen Richtungen aufnehmen.
2. Aus diesen Bildern Rückschlüsse auf die Strahlenabsorption im Inneren des Objekts schließen.

Dass man aus den Einzelbildern theoretisch das Innere des Objekts rekonstrukieren kann, ist seit 1917 bekannt (siehe Johann Radon und Radon-Transformation). Die technische Umsetzung benötigt aufgrund des Rechenaufwands einen Computer, sodass erst in den 1970er-Jahren erste CT-Geräte entwickelt wurden. Diese arbeiteten nach dem Parallelstrahlverfahren: Die Einzelbilder wurden schichtweise aufgenommen (also nur eine Rasterzeile jedes Bildes) und die Röntgenstrahlen liefen parallel durch das Objekt (vgl. Abbildung).



Funktionsprinzip der Computertomografie (links) und ein Sinogramm als Beispiel für die direkt gemessenen Daten, aus denen das Innere des untersuchten Objekts rekonstruiert werden soll (rechts).

Heute sind fächerförmige Strahlenverläufe üblich. Teils werden die Schichten auch überlappend aufgenommen oder man löst sich gänzlich vom schichtweisen Vorgehen, indem Röntgenquelle und Detektor spiralförmig um das Objekt bewegt werden (Spiral- oder Helix-CT).

4.4.2 Modellierung

Das Innere des Objekts in einer Aufnahmeschicht sei durch eine stetige Funktion $f : \mathbb{R}^2 \to \mathbb{R}$ beschrieben, die nur im Inneren $\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 < 1\}$ der Einheitskreisscheibe von Null verschieden ist. Der Wert $f(x, y)$ beschreibt die Absorption von Röntgenstrahlen im Punkt (x, y) des Objekts aufgrund des dort vorhandenen Gewebes/Materials.

Nebenbemerkung

Man könnte auch $f : \{(x, y) : x^2 + y^2 < 1\} \to \mathbb{R}$ verwenden. Allerdings wären dann auch (stetige) Funktionen zugelassen, die am Rand der Kreisscheibe ins Unendliche gehen. Um das zu vermeiden wäre noch $f : \{(x, y) : x^2 + y^2 \leq 1\} \to \mathbb{R}$ denkbar mit der Zusatzforderung, dass f auf dem Rand der Kreisscheibe Null sein soll. Dann müssten wir aber später beim Auswerten der Funktion immer darauf achten, dass das Argument (x, y) stets in der Kreisscheibe liegt. Dies würde die Formeln unnötig verkomplizieren.

Der Aufnahmevorgang wird durch die **Radon-Transformation** beschrieben. Diese bildet stetige Funktionen $f : \mathbb{R}^2 \to \mathbb{R}$ (wie oben) auf stetige Funktionen

$$R f : [0, 2\pi] \times [-1, 1] \to \mathbb{R} \tag{4.49}$$

ab, indem sie f entlang parallel verlaufender Geraden integriert:

$$(R f)(\alpha, \sigma) := \int_{-1}^{1} f\left(\sigma \begin{bmatrix} -\sin\alpha \\ \cos\alpha \end{bmatrix} + \tau \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix}\right) \mathrm{d}\tau. \tag{4.50}$$

(IDV 420, schöne Animation). Bezeichnen wir mit $g(\alpha, \sigma)$ die vom Detekor zum durch (α, σ) beschriebenen Strahlverlauf gemessene Intensität, so gilt

$$g(\alpha, \sigma) = c\, \mathrm{e}^{-(R f)(\alpha, \sigma)}, \tag{4.51}$$

wobei $c > 0$ die Intensität der Röntgenquelle angibt (Lambert-beersches Gesetz). Wir erhalten also

$$-\ln\left(\frac{g(\alpha, \sigma)}{c}\right) = (R f)(\alpha, \sigma) \tag{4.52}$$

für den Zusammenhang zwischen Messwerten $g(\alpha, \sigma)$ und der gesuchten Funktion f.

Nebenbemerkung

Wir nutzen hier (zunächst) ein kontinuierliches Modell, obwohl klar ist, dass dieses mit dem Computer nicht lösbar ist. Allerdings können wir so einerseits viele schwierige Fragen (Objektauflösung, Datenauflösung, Strahlbreite, numerische Integration,...) zunächst ausblenden. Andererseits kann man aus dem kontinuierlichen Modell Lösungsverfahren ableiten kann, die man in einem diskreten Modell nicht sehen würde.

4.4.3 Untersuchung des Modells

Die Radon-Transformation ist eine lineare Abbildung, kann also nach geeigneter Diskretisierung als Matrix-Vektor-Multiplikation umgesetzt werden.

Man kann zeigen: Zu gegebenem Sinogramm existiert stets eine Lösung und diese ist eindeutig bestimmt. Allerdings hängen die Lösungen nicht stetig von den Daten ab. Selbst sehr kleine Datenfehler (Rauschen) können zu extrem großen Abweichungen in den rekonstruierten Bildern führen.

Die unstetige Abhängigkeit der Lösung von den Daten ist kein (!) Modellierungsfehler, sondern in der Sache an sich begründet. Manche/viele Vorgänge in der Physik besitzen keine stetige Umkehrung. Diese Problematik muss bei der Auswahl eines Lösungsverfahrens unbedingt berücksichtigt werden.

Das Modell vernachlässigt diverse Details:

- Die Strahlen werden nie exakt entlang paralleler Geraden verlaufen, sondern sich mit Entfernung von der Quelle etwas aufweiten.
- Effekte wie Beugung, Brechung, Strahlaufhärtung werden nicht beachtet.
- Das Objekt wird als starr angenommen, obwohl die Aufnahme ein gewisses Zeitintervall in Anspruch nimmt, in dem Bewegungen stattfinden können.

4.5 Zerfalls- und Wachstumsprozesse

Zerfallsprozesse und Wachstumsprozesse können durch gewöhnliche Differentialgleichungen beschrieben werden. Betrachten hier beispielhaft einen konkreten Zerfallsprozess.

4.5.1 Problemstellung

Eine zum Zeitpunkt $t = 0$ vorhandene Masse m_0 eines Materials unterliege dem radioaktiven Zerfall. Zu jedem Zeitpunkt $t \geq 0$ ist die Masse $m(t)$ des noch nicht zerfallenen Materials gesucht.

Dazu zunächst zwei Beobachtungen:

- Da Zerfallsprozesse je nach Material und Umgebung unterschiedlich schnell verlaufen können, muss das Modell einen Parameter enthalten, in den die Zerfallsgeschwindigkeit eingeht.
- Aus der Beobachtung von Zerfallsprozessen ist bekannt, dass der in einem Zeitintervall zerfallende Massenanteil nicht von der am Intervallanfang vorhandenen Masse abhängt, sondern nur von der Länge des Intervalls.

4.5.2 Modell

Die Funktion $m : [0, \infty) \to \mathbb{R}$ soll durch eine gewöhnliche Differentialgleichung mit Anfangsbedingung beschrieben werden.

Aus obigen Beobachtungen erhält man die Beziehung

$$m' = c\, m \tag{4.53}$$

(IDV 430). Dies ist eine homogene lineare Differentialgleichung erster Ordnung mit konstanten Koeffizienten, welche die gesuchte Funktion $m : [0, \infty) \to \mathbb{R}$ beschreibt. Sie drückt aus, dass die Änderung m' der Materialmenge stets proportional zur vorhandenen Materialmenge m ist.

Zusätzlich ist eine Anfangsbedingung nötig:

$$m(0) = m_0. \tag{4.54}$$

4.5.3 Untersuchung des Modells

Für $c > 0$ sind die Lösungen offensichtlich monoton wachsend, sonst monoton fallend. Die allgemeine Lösung ist

$$m(t) = a\, \mathrm{e}^{ct} \tag{4.55}$$

mit $a \in \mathbb{R}$. Einsetzen des Anfangswertes liefert

$$m(t) = m_0\, \mathrm{e}^{ct}. \tag{4.56}$$

4.6 Pendel

4.6.1 Problemstellung

Die Bewegung einer Masse an einem Fadenpendel soll simuliert werden. Dabei ist nur die Bewegung in einer Ebene von Interesse.

4.6.2 Modell

Die Bewegung des Fadenpendels kann als Funktion $\varphi : [0, \infty) \to [-\frac{\pi}{2}, \frac{\pi}{2}]$ beschrieben werden, die zu jedem Zeitpunkt t die Auslenkung des Pendels als Winkel relativ zur Ruheposition angibt. Sind $\varphi(0)$ (Anfangsauslenkung) und $\varphi'(0)$ (Anfangswinkelgeschwindigkeit) vorgegeben, so ist der Bewegungsablauf durch das Newton'sche Gesetz

$$F(t, x(t), y(t)) = m \begin{bmatrix} x''(t) \\ y''(t) \end{bmatrix}, \quad t > 0, \tag{4.57}$$

eindeutig vorherbestimmt. Dabei geben $x(t)$ und $y(t)$ die Koordinaten der Pendelmasse m zur Zeit t an und F ist die auf die Pendelmasse wirkende Kraft. Diese kann prinzipiell von der Position (und somit indirekt von der Zeit) und auch direkt von der Zeit abhängen (Pendel befindet sich z.B. in einer zeitlich und räumlich veränderlichen Luftströmung). Wirkt nur die Schwerkraft, so erhält man aus dem Newton'schen Gesetz die Gleichung

$$\varphi''(t) + \frac{g}{l} \sin \varphi(t) = 0 \tag{4.58}$$

zur Beschreibung der Pendelbewegung (IDV 440). Diese ist eine nichtlineare gewöhnliche Differentialgleichung zweiter Ordnung, die sich nur sehr mühsam analytisch lösen lässt.

4.6.3 Vereinfachtes Modell

Für kleine Auslenkungen kann man die Näherung

$$\sin \varphi(t) \approx \varphi(t) \tag{4.59}$$

verwenden um eine einfacher zu lösende lineare gewöhnliche Differentialgleichung zu erhalten:

$$\varphi''(t) + \frac{g}{l} \varphi(t) = 0. \tag{4.60}$$

4.6.4 Modellfehler

Der bei der Vereinfachung des Modells entstehende Modellfehler ist um so größer je größer die Auslenkung des Pendels ist. Ob die Situation diese Vereinfachung rechtfertigt, muss im Einzelfall entschieden werden. Zu bedenken ist, dass auch das nichtlineare Modell schon Fehler enthält, z.B.:

- Vernachlässigung der Reibung,
- Idealisierung von Seil und schwingendem Objekt als Massepunkt,
- Vernachlässigung der Dehnung des Seils durch Zugkraft.

4.7 Ölteppich

4.7.1 Problemstellung

Wollen die Ausbreitung eines Ölteppichs auf einer Wasseroberfläche unter Strömungseinfluss simulieren.

Sei $\underline{v}(x_1, x_2, t) \in \mathbb{R}^2$ der Strömungsvektor (Richtung und Geschwindigkeit) einer Wasserobfläche ($\mathbb{R}^2$) am Punkt (x_1, x_2) zur Zeit t. Die Öldichte (Masse pro Fläche) sei durch $\varrho(x_1, x_2, t)$ beschrieben. Ist

$$\varrho_0(x_1, x_2) := \varrho(x_1, x_2, 0) \quad \text{für alle } (x_1, x_2) \in \mathbb{R}^2 \tag{4.61}$$

gegeben, so interessiert uns die Veränderung der Öldichte im Laufe der Zeit.

Offensichtliche Lösungen:

- Falls $\underline{v} \equiv 0$ (keine Strömung), so muss zu jeder Zeit $\varrho \equiv \varrho_0$ gelten.
- Ist $\underline{v}$ unabhängig von Ort und Zeit immer gleich (gleichmäßige Strömung in eine feste Richtung), so entsteht ϱ zur Zeit t durch Verschieben von ϱ_0. Dabei ist die Länge der Verschiebungsstrecke proportional zu t.

4.7.2 Modell

Die Lösung für allgemeines $\underline{v}$ muss die partielle Differentialgleichung

$$\frac{\partial}{\partial t}\varrho(x_1, x_2, t) + \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{bmatrix} \circ \big(\varrho(x_1, x_2, t)\, \underline{v}(x_1, x_2, t)\big) = 0 \tag{4.62}$$

für $(x_1, x_2) \in \mathbb{R}^2$ und $t \in [0, \infty)$ erfüllen. Diese erhält man durch Vergleich der Ölmasse in einem beliebigen Gebiet mit dem Ölzufluss und -abfluss über den Rand des Gebiets und Grenzübergang zu beliebig kleinen Gebieten (IDV 450).

Merke

Partielle Differentiallgleichungen erster Ordnung mit dieser Struktur heißen **Kontinuitätsgleichungen**. Kurzschreibweise:

$$\frac{\partial}{\partial t}\varrho + \operatorname{div}_{\underline{x}}(\varrho\, \underline{v}) = 0. \tag{4.63}$$

4.8 Schwingende Saite

4.8.1 Problemstellung

Die Schwingung einer an beiden Seiten fest eingespannten Saite soll simuliert werden (z.B. Gitarre).

4.8.2 Modell

Modellieren die Saite als Kurvenstück $[0,1]$. Für die Auslenkung $u(x,t)$ der Saite $B = [0,1]$ gilt

$$\frac{\sigma}{\varrho}\, u_{xx}(x,t) - u_{tt}(x,t) = 0 \quad \text{für alle } x \in (0,1) \quad \text{und alle } t > 0 \tag{4.64}$$

(IDV 460). Dabei ist $\varrho > 0$ die Dichte der Saite (Masse pro Länge) und $\sigma > 0$ ist Zugspannung in der Saite im Zustand minimaler Spannung. Bei gegebener Anfangsauslenkung, gegebener Anfangsgeschwindigkeit und geeigneten Randbedingungen beschreibt diese partielle Differentialgleichung den zeitlichen Verlauf der Auslenkung.

Allgemeiner gilt:

Merke

Partielle Differentialgleichungen mit der Struktur

$$c^2\, \Delta_{\underline{x}} u - \frac{\partial^2}{\partial^2 t} u = 0 \tag{4.65}$$

mit einer Konstante $c > 0$ heißen **Wellengleichungen**.

Lösungen der Wellengleichungen beschreiben die Auslenkung $u(\underline{x}, t)$ eines Mediums (Saite, Membran, Luft,...) an einer Stelle $\underline{x} \in \mathbb{R}^n$ zur Zeit $t > 0$. Die Konstante c ist die Ausbreitungsgeschwindigkeit der Wellen (Lichtgeschwindigkeit bei elektromagnetischen Wellen, Schallgeschwindigkeit bei Schallwellen,...).

5 Fourier-Analysis

Vorstellung verschiedener Varianten der Fourier-Transformation.

5.1 Überblick und Grundlagen

5.1.1 Idee

Unter Fourier-Analysis versteht man das Darstellen von Funktionen und diskreten Signalen als **Summe von Kreisfunktionen**. Unter Kreisfunktionen verstehen wir hier Sinus-Funktionen, Cosinus-Funktionen und deren komplexwertige Entsprechungen (sie unten), jeweils mit unterschiedlichen Periodendauern.

Man kann zeigen, dass praktisch alle Funktionen und diskreten Signale ohne Informationsverlust eine solche Darstellung besitzen. Nur die Anteile der verschiedenen Kreisfunktionen bzw. Periodendauern unterscheiden sich von Fall zu Fall. Man spricht hier auch von der Frequenzdarstellung oder vom **Frequenzspektrum** einer Funktion oder eines diskreten Signals.

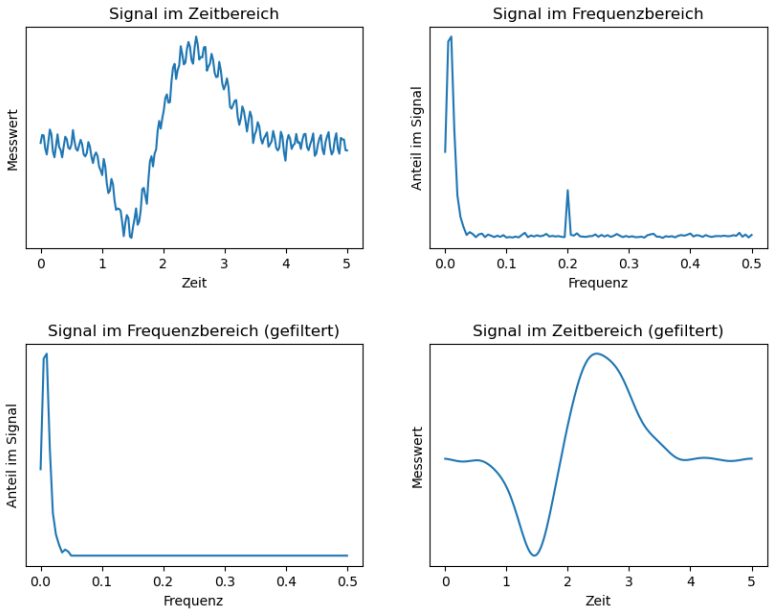
5.1.2 Anwendungen

Die Fourier-Analysis findet in nahezu allen Bereichen von Naturwissenschaft und Technik Anwendung. Insbesondere in der Signalverarbeitung erlaubt sie das einfache **Filtern von Signalen**. Bei der **Untersuchung von Schwingungsprozessen** liefert sie wichtige Einsichten in die Vorgänge (erste Anwendungen fand die Fourier-Analysis bei der Vorhersage der Gezeiten). In der Strömungsmechanik können mittels Fourier-Analysis **Lösungen von Strömungsgleichungen** gefunden werden. In der Physik spielt sie eine zentrale Rolle, z.B. in der **Quantenmechanik** und im Kontext **elektromagnetischer Wellen**.

5.1.3 Beispiel

Durch Messgeräte aufgenommene Signale enthalten praktisch immer einen gewissen Anteil Rauschen. Ist das Grundsignal verhältnismäßig glatt, kann das Rauschen mittels Fourier-Analysis leicht entfernt werden:

1. Transformiere das Signal in den Frequenzbereich.
2. Setze hohe Frequenzanteile auf Null.
3. Transformiere das Signal zurück in den Zeitbereich (oder Ortsbereich, je nach Kontext).



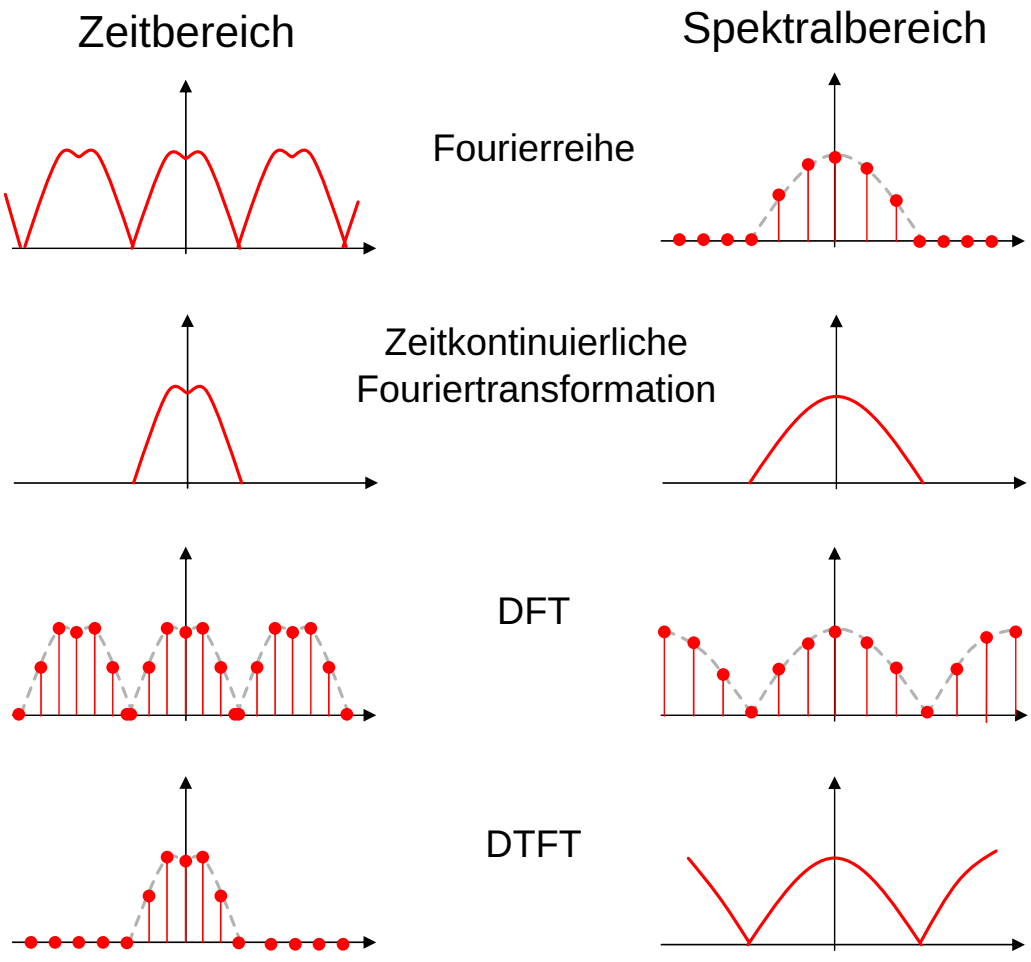
5.1.4 Varianten der Fourier-Transformation

Die Idee, Funktionen und diskrete Signale als Summen periodische Funktionen oder Signale darzustellen, lässt sich in verschiedener Weise umsetzen und auf verschiedene Typen von Funktionen und Signalen anwenden. Es gibt es nicht die eine Fourier-Transformation, sondern mehrere verschiedene. Zusätzlich kann ein und dieselbe Variante durch formal unterschiedliche, aber inhaltlich identische Formeln beschrieben werden. Folgende Varianten von Fourier-Transformationen werden wir behandeln:

- Die **kontinuierliche Fourier-Transformation (CTFT)** transformiert eine auf $\mathbb{R}$ definierte aperiodische Funktion in eine auf $\mathbb{R}$ definierte Funktion.
- Die **Fourier-Transformation für periodische Funktionen** transformiert eine auf $\mathbb{R}$ periodische Funktion (oder eine auch einem Intervall $[a, b]$ definierte Funktion, die periodisch auf $\mathbb{R}$ fortgesetzt wird) in eine Zahlenfolge, also in ein diskretes Signal.
- Die **diskrete Fourier-Transformation (DFT)** transformiert ein endliches diskretes Signal in ein endliches diskretes Signal.

Im Kontext der Fourier-Analysis treten auch folgende Begriffe auf, die wir der Übersicht halber grob einordnen, aber nicht weiter verfolgen:

- Die **schnelle Fourier-Transformation** ist ein spezielles Verfahren zur Berechnung der diskreten Fourier-Transformation. Es handelt sich also nicht um eine zusätzlich Transformationsart, sondern nur um eine konkrete Umsetzung der Transformation am Computer.



Quelle: wikipedia.org, Walter Dvorak, CC BY 3.0 (bearbeitet)

- Als **Fourier-Reihe** wird das Ergebnis der Fourier-Transformation für periodische Funktionen bezeichnet.
- Die **Fourier-Transformation für zeitdiskrete Signale (DTFT)** ist eine weitere, aber weniger bedeutende, Form der Fourier-Transformation. Sie überführt ein unendliches, nicht periodisches diskretes Signale in eine Funktion über $\mathbb{R}$.

5.1.5 Komplexwertige Funktionen

Möchte man eine Funktion als Summe von Kreisfunktionen verschiedener Frequenz darstellen, so werden im Allgemeinen sowohl Sinus-Funktionen als auch Cosinus-Funktionen benötigt. Eine reine Sinus-Summe liefert stets eine ungerade Funktion, während eine reine Cosinus-Summe stets nur gerade Funktionen liefert.

Die parallele Handhabung von zwei Funktionensystemen (Sinus- und Cosinus-Funktionen unterschiedlicher Frequenzen) und entsprechender Anteile in der zu untersuchenden Funktion ist recht unhandlich. Deshalb werden Fourier-Transformationen praktisch immer mit Hilfe komplexer Zahlen dargestellt.

Merke

Sinus- und Cosinus-Funktionen gleicher Frequenz $\omega \in \mathbb{R}$ können als komplexe Exponentialfunktion geschrieben werden:

$$\cos(\omega\, x) + \mathrm{i}\, \sin(\omega\, x) = \mathrm{e}^{\mathrm{i}\,\omega\, x} \qquad \text{für alle } x \in \mathbb{R} \tag{5.1}$$

(vgl. Eulersche Formel).

Wie werden also mit komplexwertigen Funktionen $f : \mathbb{R} \to \mathbb{C}$ arbeiten. Die grafische Darstellung solcher Funktionen kann in einem dreidimensionalen Koordinatensystem erfolgen. Jedem Argument wird ein Punkt in der komplexen Zahlenebene zugeordnet.

Die Berechnung von Fourier-Transformationen macht ganz wesentlich von Integralen komplexwertiger Funktionen Gebrauch. Die Integration getrennt nach Real- und Imaginärteil:

$$\int_a^b f(x)\,\mathrm{d}x := \int_a^b \operatorname{Re} f(x)\,\mathrm{d}x + \mathrm{i} \int_a^b \operatorname{Im} f(x)\,\mathrm{d}x. \tag{5.2}$$

Analog wird mit Ableitungen verfahren:

$$f' = (\operatorname{Re} f)' + \mathrm{i}(\operatorname{Im} f)'. \tag{5.3}$$

Die üblichen Ableitungsregeln gelten auch für komplexwertige Funktionen.

Beispiel

Für

$$f(x) := \mathrm{e}^{\mathrm{i}\,x} \tag{5.4}$$

erkält man beispielsweise

$$f'(x) = \mathrm{i}\,\mathrm{e}^{\mathrm{i}\,x} \tag{5.5}$$

(IDV 505).

5.2 Aperiodische Funktionen

Die Fourier-Transformation für aperiodische (d.h. nicht periodische) Funktionen überführt ein auf $\mathbb{R}$ definierte Funktion wieder in eine auf $\mathbb{R}$ definierte Funktion.

5.2.1 Definition

Die folgende Definition liefert wohldefinierte Ausdrücke, wenn die zu transformierende Funktion $f : \mathbb{R} \to \mathbb{C}$ **absolut integrierbar** ist, d.h. wenn

$$\int_{-\infty}^{\infty} |f(x)| \, \mathrm{d}x < \infty \tag{5.6}$$

gilt. Dabei handelt es sich streng genommen um ein so genanntes Lebesgue-Integral, welches jedoch in allen praktisch relevanten Fällen mit dem üblichen (uneigentlichen) Riemann-Integral übereinstimmt.

Da die Fourier-Transformierte ohnehin komplexwertig sein wird, können wir für die zu transformierende Funktion f ohne Mehraufwand ebenfalls komplexe Funktionswerte zulassen.

Merke

Sei $f : \mathbb{R} \to \mathbb{C}$ absolut integrierbar. Die durch

$$\hat{f}(\omega) := \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{\infty} f(x) \, \mathrm{e}^{-\mathrm{i}\,\omega\,x} \, \mathrm{d}x \tag{5.7}$$

gegebene Funktion $\hat{f} : \mathbb{R} \to \mathbb{C}$ heißt **Fourier-Transformierte** zu f.

Die durch

$$\mathcal{F}\, f := \hat{f} \tag{5.8}$$

definierte Abbildung $\mathcal{F}$ heißt **kontinuierliche Fourier-Transformation** (kurz: CTFT für “continuous time Fourier transform”).

Die Fourier-Transformierte $\hat{f}$ enthält sämtliche Informationen über die ursprüngliche Funktion f. Man kann somit f aus $\hat{f}$ zurückgewinnen. Mit anderen Worten: Die Fourier-Transformation $\mathcal{F}$ ist invertierbar.

Merke

Ist $\hat{f}$ die Fourier-Transformierte zu f und ist $\hat{f}$ absolut integrierbar, so gilt

$$f(x) = (\mathcal{F}^{-1}\hat{f})(x) = \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{\infty} f(x) \, \mathrm{e}^{\mathrm{i}\,\omega\,x} \, \mathrm{d}\omega \tag{5.9}$$

für alle $x \in \mathbb{R}$.

Die Forderung nach absoluter Integrierbarkeit von $\hat{f}$ resultiert aus der Tatsache, das die Menge der absolut integrierbaren Funktionen nicht die natürliche Grundmenge für die Fourier-Transformation ist. Besser geeignet als Grundmenge ist der so genannten Schwartz-Raum. Dieser wird von der Fourier-Transformation $\mathcal{F}$ auf sich selbst abgebildet, sodass auch die Rücktransformation ohne Zusatzforderungen stets machbar ist.

Beispiel (Rechteckfunktion)

Die Fourier-Transformierte der **Rechteckfunktion**

$$r(x) := \begin{cases} 1, & \text{für } |x| < \frac{1}{2}, \\ \frac{1}{2}, & \text{für } |x| = \frac{1}{2}, \\ 0, & \text{für } |x| > \frac{1}{2} \end{cases} \tag{5.10}$$

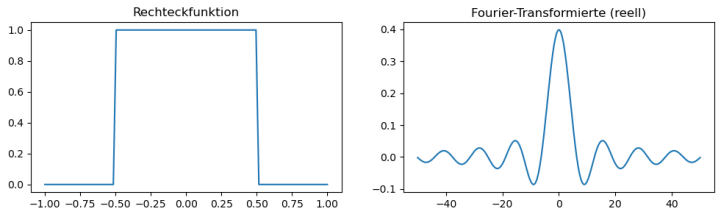
ist gerade

$$\hat{r}(\omega) = \frac{1}{\sqrt{2\,\pi}}\, s\left(\frac{1}{2\,\pi}\,\omega\right), \tag{5.11}$$

wobei

$$s(x) := \frac{\sin(\pi\, x)}{\pi\, x} \tag{5.12}$$

die **Sinc-Funktion** bezeichnet.



5.2.2 Anwendungen

Für in der Praxis oft auftretende Funktionen sind die Fourier-Transformierten in Tabellen erfasst. Neben dem manuellen Berechnen ist auch die numerische Berechnung mit dem Computer prinzipiell machbar. Allerdings birgt diese einige Schwierigkeiten, sodass wir hier nicht näher darauf eingehen. Auch ist die kontinuierliche Fourier-Transformation aus diesem Grund im Gegensatz zur später zu behandelnden diskreten Fourier-Transformation kaum in Standard-Software integriert.

Anwendungen der kontinuierlichen Fourier-Transformation liegen vor allem im theoretischen Bereich. Beispiele:

- Untersuchung des Verhaltens von elektrischen Bauteilen (z.B. Tiefpassfilter),
- Lösen von Differentialgleichungen (siehe später),
- Erklärung/Untersuchung akustischer Phänomene (z.B. Obertöne bei Musikinstru-

menten).

Ein praktische Anwendung der kontinuierlichen Fourier-Transformation findet man bei der Computer-Tomografie. Die dabei auftretende mehrdimensionale Fourier-Transformation wird weiter unten eingeführt.

Beispiel (Zentralschnitt-Satz)

Der Zentralschnitt-Satz (auch bekannt als "Fourier slice theorem") bringt die Fourier-Transformation in enge Verbindung mit der Radon-Transformation. Er besagt, dass man aus den Fourier-Transformierten der einzelnen CT-Projektionen die (zweidimensionale) Fourier-Transformierte des untersuchten Objekts erhält (IDV 510).

Für das Invertieren der Radon-Transformation ergibt sich daraus folgender Ablauf:

1. Fourier-Transformierte für jede Projektion berechnen.
2. Fourier-Transformierte des Objekts aus den transformierten Projektionen zusammensetzen.
3. Inverse (zweidimensionale) Fourier-Transformation anwenden.

5.2.3 Eigenschaften

Fourier-Transformierte werden nur selten über die Definition als Integral berechnet. Üblicherweise entnimmt man die Fourier-Transformierten einiger Grundfunktionen aus Tabellen und verwendet dann diverse Eigenschaften der Fourier-Transformation, um daraus die gesuchte Fourier-Transformierte zu ermitteln. Einige dieser Eigenschaften sind:

- Für $a, b \in \mathbb{C}$ und Funktionen f_1, f_2 gilt

$$\widehat{a\, f_1 + b}\, f_2(\omega) = a\, \hat{f}_1(\omega) + b\, \hat{f}_2(\omega) \tag{5.13}$$

  (Linearität).

- Für $a \in \mathbb{R}$ gilt

$$f(x) = g(x - a) \quad \Leftrightarrow \quad \hat{f}(\omega) = \mathrm{e}^{-\mathrm{i}\, a\, \omega}\, \hat{g}(\omega) \tag{5.14}$$

  (Zeitverschiebung).

- Für $a \in \mathbb{R}$ gilt

$$f(x) = \mathrm{e}^{\mathrm{i}\, a\, \omega}\, g(x) \quad \Leftrightarrow \quad \hat{f}(\omega) = \hat{g}(\omega - a) \tag{5.15}$$

(Frequenzverschiebung).

- Für $a \in \mathbb{R}$ gilt

$$f(x) = g(a\,x) \quad \Leftrightarrow \quad \hat{f}(\omega) = \frac{1}{|a|}\,\hat{g}\left(\frac{\omega}{a}\right). \tag{5.16}$$

- Ableitungen werden in Produkte überführt, d.h.

$$f(x) = g^{(k)}(x) \quad \Leftrightarrow \quad \hat{f}(\omega) = (\mathrm{i}\,\omega)^k\,\hat{g}(\omega). \tag{5.17}$$

- Die Gauß'sche Glockenkurve bleibt unverändert, d.h.

$$f(x) = \mathrm{e}^{-\frac{1}{2}x^2} \quad \Leftrightarrow \quad \hat{f}(\omega) = \mathrm{e}^{-\frac{1}{2}\omega^2}. \tag{5.18}$$

Ein wichtige Rolle spielt auch der sogenannte Faltungssatz, der die Multiplikation zweier Funktionen im Frequenzbereich in der Zeit- bzw. Ortsbereich überträgt.

Merke

Seien $f : \mathbb{R} \to \mathbb{C}$ und $g : \mathbb{R} \to \mathbb{C}$ absolut integrierbar. Die durch

$$(f * g)(x) := \sqrt{2\,\pi}\int_{-\infty}^{\infty} f(y)\,g(x-y)\,\mathrm{d}y \tag{5.19}$$

auf $\mathbb{R}$ definierte Funktion $f * g$ heißt **Faltung** von f und g. Sie ist stets absolut integrierbar und es gilt

$$\widehat{f * g}(\omega) = \hat{f}(\omega)\,\hat{g}(\omega) \quad \text{für alle } \omega \in \mathbb{R}. \tag{5.20}$$

5.2.4 Reellwertige Formulierung

Für reellwertige Funktionen $f : \mathbb{R} \to \mathbb{R}$ kann man alternativ zur Fourier-Transformation auch Cosinus- und Sinus-Transformationen separat betrachten:

$$(\mathcal{C}\,f)(\omega) := \frac{1}{\sqrt{2\,\pi}}\int_{-\infty}^{\infty} f(x)\,\cos(\omega\,x)\,\mathrm{d}x, \tag{5.21}$$

$$(\mathcal{S}\,f)(\omega) := \frac{1}{\sqrt{2\,\pi}}\int_{-\infty}^{\infty} f(x)\,\sin(\omega\,x)\,\mathrm{d}x. \tag{5.22}$$

Die Cosinus-Transformierte ist stets eine gerade Funktion und die Sinus-Transformierte ist stets ungerade:

$$(\mathcal{C}\,f)(-\omega) = (\mathcal{C}\,f)(\omega), \qquad (\mathcal{S}\,f)(-\omega) = -(\mathcal{S}\,f)(\omega). \tag{5.23}$$

Offensichtlich gilt

$$\mathcal{F} f = \mathcal{C} f - \mathrm{i}\,\mathcal{S} f \tag{5.24}$$

und auch

$$\mathcal{F}^{-1} \hat{f} = \mathcal{C} \hat{f} + \mathrm{i}\,\mathcal{S} \hat{f}. \tag{5.25}$$

Sind f_c und f_s die Cosinus- und Sinus-Transformierten einer Funktion f, so kann man daraus durch nochmalige Transformation wieder f erhalten:

$$f = \mathcal{C} f_\mathrm{c} + \mathcal{S} f_\mathrm{s} \tag{5.26}$$

(IDV 515).

5.2.5 Mehrdimensionale Fourier-Transformation

Die Fourier-Transformation kann auch für Funktionen $f : \mathbb{R}^n \to \mathbb{C}$ sinnvoll eingeführt werden:

$$\hat{f}(\omega) := \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{R}^n} f(x)\,\mathrm{e}^{-\mathrm{i}\,\omega^\mathrm{T}\,x}\,\mathrm{db} \tag{5.27}$$

Das Integral ist hier ein Bereichsintegral über den gesamten $\mathbb{R}^n$. Im Exponent steht das Skalarprodukt der beiden Vektoren $x = (x_1, \ldots, x_n)$ und $\omega = (\omega_1, \ldots, \omega_n)$.

Die inverse Fourier-Transformation sieht entsprechend aus:

$$f(x) = \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{R}^n} f(x)\,\mathrm{e}^{\mathrm{i}\,\omega^\mathrm{T}\,x}\,\mathrm{db}. \tag{5.28}$$

Beispiel ((Interpretation der mehrdimensionalen Transformation))

Die Funktionen $x \mapsto \mathrm{e}^{-\mathrm{i}\,\omega^\mathrm{T}\,x}$ (mit festem ω) können für $n > 1$ nicht direkt im dreidimensionalen Raum visualisiert werden. Selbst für $n = 2$ ist Graph der Funktion bereits vierdimensional. Betrachten deshalb Ersatzweise für $n = 2$ nur die ensprechende Cosinus-Funktion

$$(x_1, x_2) \mapsto \cos(\omega_1\,x_1 + \omega_2\,x_2). \tag{5.29}$$

Der Vektor ω gibt Richtung und Frequenz der Schwingung vor. Entlang der Richtung ω entspricht der Graph gerade dem Graph der Cosinus-Funktion, die entsprechend der Länge von ω gestaucht bzw. gestreckt ist. Senkrecht zu ω ist die Funktion konstant.

5.3 Periodische Funktionen

Die Fourier-Transformation für periodische Funktionen überführt eine $\mathbb{R}$ definierte periodische Funktion in eine doppelt unendliche Zahlenfolge $(c_k)_{k\in\mathbb{Z}}$. Im Gegensatz zur Fourier-Transformation bei aperiodischen Funktionen ist die Darstellung periodischer Funktionen im Frequenzbereich also nicht kontinuierlich, sondern es genügen (unendlich viele) diskrete Frequenzen zur vollständigen Beschreibung einer Funktion.

Funktionen die nur auf einem endlichen Intervall $[a, b]$ definiert sind, können stets zu periodischen Funktionen auf ganz $\mathbb{R}$ fortgesetzt werden. Somit kann die Fourier-Transformation auch für auf einem Intervall definierte Funktionen genutzt werden.

5.3.1 Definition

Merke

Sei $f : \mathbb{R} \to \mathbb{C}$ periodisch mit Periode $T > 0$ und sei f auf $[-\frac{T}{2}, \frac{T}{2}]$ absolut integrierbar. Die durch

$$c_k := \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(x)\, \mathrm{e}^{-\mathrm{i}\, \frac{2\,\pi\,k}{T}\, x}\, \mathrm{d}x \tag{5.30}$$

definierte Zahlenfolge $(c_k)_{k\in\mathbb{Z}}$ heißt **Fourier-Transformierte** von f. Die Zahlen c_k heißen **Fourier-Koeffizienten**.

Die durch

$$(\mathcal{F}\, f)_k := c_k \tag{5.31}$$

definierte Abbildung $\mathcal{F}$ heißt **Fourier-Transformation**.

Aus der Fourier-Transformierten kann man die ursprüngliche Funktion als unendliche Summe von Funktionen zurückgewinnnen. Dabei ist zu klären, unter welchen Bedingungen und in welchem Sinn diese Summe gegen die Funktion konvergiert.

Merke

Sei $(c_k)_{k\in\mathbb{Z}}$ die Fourier-Transformierte einer periodischen Funktion f. Die Funktionenreihe

$$x \mapsto \sum_{k=-\infty}^{\infty} c_k\, \mathrm{e}^{\mathrm{i}\, \frac{2\,\pi\,k}{T}\, x} \tag{5.32}$$

wird **Fourier-Reihe** zu f genannt. Sie konvergiert

- gleichmäßig gegen f, dh.

$$\lim_{n\to\infty} \sup_{x\in[-\frac{T}{2},\frac{T}{2}]} \left| \sum_{k=-n}^{n} c_k \, \mathrm{e}^{\mathrm{i}\,\frac{2\,\pi\,k}{T}\,x} - f(x) \right| = 0, \tag{5.33}$$

wenn f stetig differenzierbar ist,

- gleichmäßig gegen f, wenn f stetig ist und

$$\sum_{k=-\infty}^{\infty} |c_k| < \infty \tag{5.34}$$

gilt,

- punktweise gegen f, wenn f bis auf endlich viele Sprungstellen (pro Periode) stetig ist, wobei allerdings an Sprungstellen x_0 nur

$$\lim_{n\to\infty} \sum_{k=-n}^{n} c_k \, \mathrm{e}^{\mathrm{i}\,\frac{2\,\pi\,k}{T}\,x} = \frac{f(x_0-0)+f(x_0+0)}{2} \tag{5.35}$$

gilt.

Die Folge der c_k wird auch als **Frequenzspektrum** von f bezeichnet, während man bei der Folge der Beträge $|c_k|$ vom **Amplitudenspektrum** spricht.

Nebenbemerkung

Entsteht die zu transformierende Funktion f durch Fortsetzung einer nur auf einem endlichen Intervall gegebenen Funktion, so müssen im Kontext obiger Konvergenzaussagen Stetigkeit und Differenzierbarkeit auch an den "Klebestellen" der einzelnen Perioden gegeben sein.

5.3.2 Anwendungen

Die Fourier-Transformation für periodische Funktionen hat innermathematisch wichtige Anwendungen, außerhalb aber kaum. Sie ist jedoch Grundlage und Ausgangspunkt für die in nahezu allen Teilgebieten von Naturwissenschaft und Technik anzutreffende diskrete Fourier-Transformation (siehe nächstes Kapitel).

5.3.3 Reellwertige Formulierung

Für reellwertige periodische Funktionen $f : \mathbb{R} \to \mathbb{R}$ kann man analog zum aperiodischen Fall wieder separat Cosinus- und Sinus-Anteile der Funktion bestimmen. Auf diesem

Weg vermeidet man das Arbeiten mit komplexen Zahlen.

Mit den durch

$$a_k := \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(x)\, \cos\bigl(\tfrac{2\,\pi\,k}{T}\, x\bigr)\, \mathrm{d}x, \tag{5.36}$$

$$b_k := \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(x)\, \sin\bigl(\tfrac{2\,\pi\,k}{T}\, x\bigr)\, \mathrm{d}x \tag{5.37}$$

definierten Folgen $(a_k)_{k\geq 0}$ und $(b_k)_{k\geq 1}$ gilt

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \bigl(a_k\, \cos\bigl(\tfrac{2\,\pi\,k}{T}\, x\bigr) + b_k\, \sin\bigl(\tfrac{2\,\pi\,k}{T}\, x\bigr)\bigr) \tag{5.38}$$

im Sinne obiger Konvergenzaussagen.

Der Zusammenhang zur komplexwertigen Fourier-Transformation ist durch

$$a_k = c_k + c_{-k} \quad (k \geq 0), \qquad b_k = \mathrm{i}\,(c_k - c_{-k}) \quad (k \geq 1) \tag{5.39}$$

bzw.

$$c_k = \frac{a_k - \mathrm{i}\, b_k}{2} \quad (k \geq 1), \qquad c_0 = \frac{a_0}{2}, \qquad c_k = \frac{a_{-k} + \mathrm{i}\, b_{-k}}{2} \quad (k \leq -1) \tag{5.40}$$

gegeben.

5.4 Diskrete Signale

Die diskrete Fourier-Transformation überführt eine periodische doppelt unendliche Zahlenfolge in eine endliche Folge von Zahlen.

Liegt nur ein diskretes Signal enlicher Länge vor, so kann dieses stets zu einer periodischen Folge fortgesetzt werden. Die diskrete Fourier-Transformation ist also auch auf Signale endlicher Länge anwendbar.

5.4.1 Definition

Merke

Sei $(a_k)_{k\in\mathbb{Z}}$ eine periodische Folge komplexer Zahlen mit Periode $N \in \mathbb{N}$. Die durch

$$\hat{a}_k := \frac{1}{N} \sum_{j=0}^{N-1} a_j \, \mathrm{e}^{-\mathrm{i}\,\frac{2\,\pi\,k}{N}\,j} \tag{5.41}$$

definierte endliche Folge $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1}) \in \mathbb{C}^N$ heißt **Fourier-Transformierte** von $(a_k)_{k\in\mathbb{Z}}$. Die Zahlen $\hat{a}_k$ heißen **Fourier-Komponenten**.

Die durch

$$\left(\mathcal{F}\,(a_l)_{l\in\mathbb{Z}}\right)_k := \hat{a}_k \tag{5.42}$$

definierte Abbildung $\mathcal{F}$ heißt **diskrete Fourier-Transformation** (kurz: DFT).

Aus der Fourier-Transformierten kann man stets das ursprüngliche Signal zurückgewinnen.

Merke

Ist $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1}) \in \mathbb{C}^N$ die Fourier-Transformierte einer Folge $(a_k)_{k\in\mathbb{Z}}$, so gilt

$$a_k = \sum_{j=0}^{N-1} \hat{a}_j \, \mathrm{e}^{\mathrm{i}\,\frac{2\,\pi\,k}{N}\,j} \tag{5.43}$$

für $k \in \mathbb{Z}$.

Die diskrete Fourier-Transformation kann als Matrix-Vektor-Produkt formuliert werden: Ist $F \in \mathbb{C}^{N\times N}$ die durch

$$F_{k,j} := \frac{1}{N} \, \mathrm{e}^{-\mathrm{i}\,\frac{2\,\pi\,k}{N}\,j}, \quad k, j = 0, \ldots, N-1 \tag{5.44}$$

definierte Matrix und ist $a := (a_0, a_1, \ldots, a_{N-1})$ eine Periode der Folge $(a_k)_{k\in\mathbb{Z}}$, so gilt

$$\hat{a} = F\,a. \tag{5.45}$$

und umgekehrt

$$a = F^{-1}\,\hat{a}. \tag{5.46}$$

Nebenbemerkung

Die praktische Berechnung der diskreten Fourier-Transformation erfolgt nicht durch Matrix-Vektor-Multiplikation, sondern durch einen speziellen Algorithmus, die sogenannten schnelle Fourier-Transformation (kurz: FFT für "fast Fourier transform"). Dieser Algorithmus benötigt deutlich weniger Rechenzeit als die Matrix-Vektor-Multiplikation.

5.4.2 Zusammenhang mit periodischen Funktionen

Sei $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1}) \in \mathbb{C}^N$ die Fourier-Transformierte von $(a_k)_{k\in\mathbb{Z}}$ und sei $T > 0$. Die durch

$$f(x) := \sum_{j=0}^{N-1} \hat{a}_j\, \mathrm{e}^{\mathrm{i}\,\frac{2\pi j}{T}\,x} \tag{5.47}$$

definierte Funktion $f : \mathbb{R} \to \mathbb{C}$ hat folgende Eigenschaften:

- f ist periodisch mit Periode T.
- Für jedes $k \in \mathbb{Z}$ gilt

  $$f\big(k\,\tfrac{T}{N}\big) = a_k, \tag{5.48}$$

  d.h. f interpoliert die Folge $(a_k)_{k\in\mathbb{Z}}$ auf den Stützstellen $0, \frac{T}{N}, 2\,\frac{T}{N}, \ldots, (N-1)\,\frac{T}{N}$.
- Die Fourier-Transformierte von f ist durch

  $$(\mathcal{F}\,f)_k = \begin{cases} \hat{a}_k, & \text{für } k = 0, \ldots, N-1, \\ 0, & \text{sonst} \end{cases} \tag{5.49}$$

  gegeben.

In diesem Sinne ist die diskrete Fourier-Transformation also eine diskretisierte Version der Fourier-Transformation periodischer Funktionen.

5.4.3 Reellwertige Signale

Analog zur Fourier-Transformation für periodische Funktionen können für reellwertige N-periodische Signale $(a_k)_{k\in\mathbb{Z}}$ komplexe Zahlen in bei der Fourier-Transformation vermieden werden, wenn Cosinus- und Sinus-Anteile separat betrachtet werden.

Für reellwertige Signale gilt außerdem

$$\hat{a}_k = \overline{\hat{a}_{N-k}} \qquad (k = 1, \ldots, N-1), \tag{5.50}$$

d.h. die Information über das ursprüngliche Signal ist vollständig in der Hälfte der Fourier-Komponenten enthalten. Genauer: Die Fourier-Transformierte eines N-periodischen reellwertigen Signals ist durch N reelle Zahlen beschrieben (IDV 520).

5.4.4 Mehrdimensionale Fourier-Transformation

Führen hier nur die Fourier-Transformation für zweidimensionale Signale ein. Für höhere Dimensionen erfolgt die Berechnung völlig analog.

Sei $A \in \mathbb{C}^{m \times n}$ ein zweidimensionales Signal mit einträgen $a_{m,n}$ (welches man sich auch periodisch auf $\mathbb{Z} \times \mathbb{Z}$ fortgesetzt vorstellen kann). Dann ist die **zweidimensionale Fourier-Transformierte** gegeben durch

$$\hat{a}_{k,l} = \frac{1}{M\,N} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{m,n}\, \mathrm{e}^{-\mathrm{i}\cdot 2\,\pi\,(\frac{k}{M}\,m + \frac{l}{N}\,n)} \tag{5.51}$$

für $k = 0, \ldots, M-1$ und $l = 0, \ldots, N-1$.

Die inverse Transformation ist

$$a_{m,n} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} \hat{a}_{k,l}\, \mathrm{e}^{\mathrm{i}\cdot 2\,\pi\,(\frac{k}{M}\,m + \frac{l}{N}\,n)}. \tag{5.52}$$

5.4.5 Anwendungen

Die diskrete Fourier-Transformation ist von grundlegender Bedeutung in vielen Bereichen von Naturwissenschaft und Technik. Die Anwendung reiche von einfachen Aufgaben der Signalverarbeitung (Filter) bis zur Kompression von Audio- und Videosignalen und Schwingungsanalysen aller Art.

5.4.6 Beispiel: Audiosignale

Wollen ein Audiosignal (Klaviermusik) im Frequenzbereich untersuchen.

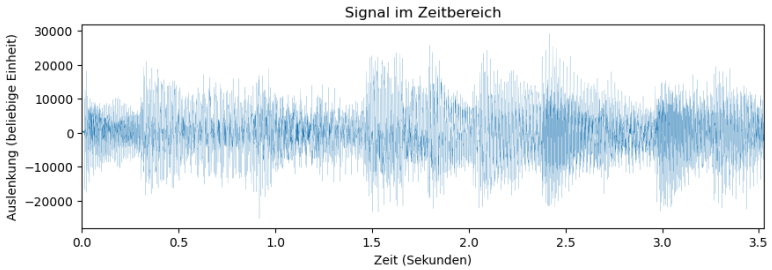
5.4.6.1 Digitale Audiosignale Digitale Audiosignale sind folgen reeller Zahlen, die gewissen äquidistanten Zeitpunkten eine Auslenkung (des Luftdruck bzw. der Lautsprechermembran) zuordnen. Dabei werden üblicherweise einige 10000 Auslenkungswerte pro Sekunde gespeichert. Diese Hohe Anzahl ist nötig um auch hochfrequente Schwingungen hinreichend gut wiedergeben zu können.

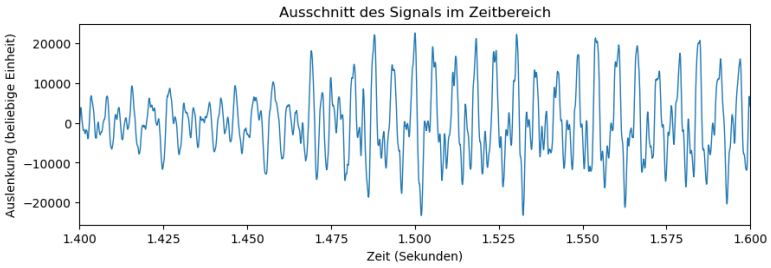
```
from IPython.display import Audio
import matplotlib.pyplot as plt
import numpy as np
import scipy

# WAV -Datei mit einem Track (mono!) laden
samples_per_sec, data = scipy.io.wavfile.read('audio.wav')

# Daten plotten
t = np.arange(0, len(data)) / samples_per_sec  # Zeitpunkte
fig, ax = plt.subplots(figsize=(10, 3))
ax.plot(t, data, lw=0.1)
ax.set_xlim(t.min(), t.max())
ax.set_xlabel('Zeit (Sekunden)')
ax.set_ylabel('Auslenkung (beliebige Einheit)')
ax.set_title('Signal im Zeitbereich')
plt.show()

# Ausschnitt plotten
t_min = 1.4
t_max = 1.6
mask = np.logical_and(t > t_min, t < t_max)
fig, ax = plt.subplots(figsize=(10, 3))
ax.plot(t[mask], data[mask], lw=1)
ax.set_xlim(t_min, t_max)
ax.set_xlabel('Zeit (Sekunden)')
ax.set_ylabel('Auslenkung (beliebige Einheit)')
ax.set_title('Ausschnitt des Signals im Zeitbereich')
plt.show()
```





5.4.6.2 Spektrum Die diskrete Fourier-Transformation kann mit numpy.fft.fft (Komponenten) und numpy.fft.fftfreq (zugehörige Frequenzen) leicht berechnet werden.

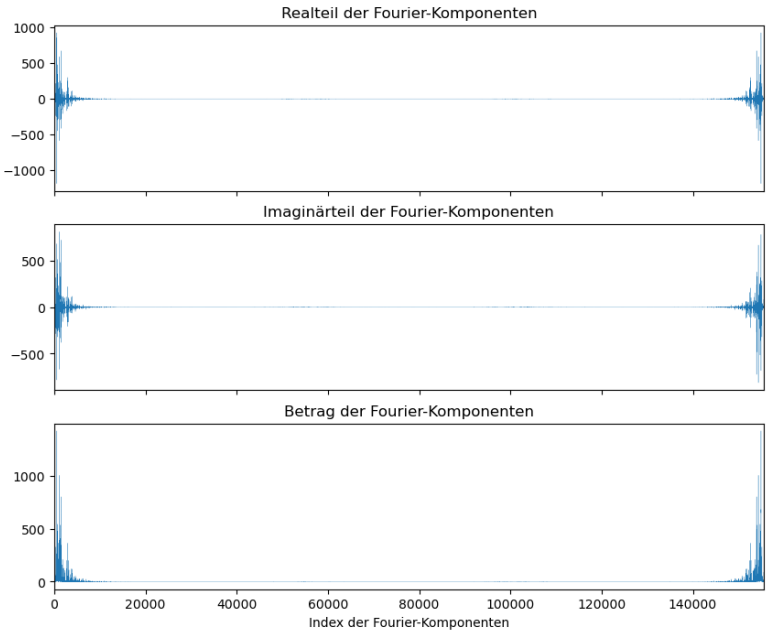
Achtung!

Der Skalierungsfaktor vor dem Integral wird sehr unterschiedlich gehandhabt. Wir verwenden hier $\frac{1}{N}$ für die Vorwärtstransformation und 1 für die inverse Transformation. Nutzt man eine Bibliotheksfunktion für die Berechnung der Fourier-Transformation, so sollte man sich stets im Klaren darüber sein, welchen Skalierungsfaktor die Funktion verwendet. Gegebenenfalls muss manuell korrigiert werden!

```
# diskrete F -Trafo mittels schneller F -Trafo
spec = 1 / len(data) * np.fft.fft(data)

# Frequenzen zu den Fourier -Komponenten
freq = np.fft.fftfreq(len(data), 1 / samples_per_sec)

# Fourier -Komponenten plotten
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 8), sharex=True)
ax1.plot(spec.real, lw=0.2)
ax2.plot(spec.imag, lw=0.2)
ax3.plot(np.abs(spec), lw=0.2)
ax3.set_xlim(0, len(spec))
ax3.set_xlabel('Index der Fourier -Komponenten')
ax1.set_title('Realteil der Fourier -Komponenten')
ax2.set_title('Imaginärteil der Fourier -Komponenten')
ax3.set_title('Betrag der Fourier -Komponenten')
plt.show()
```



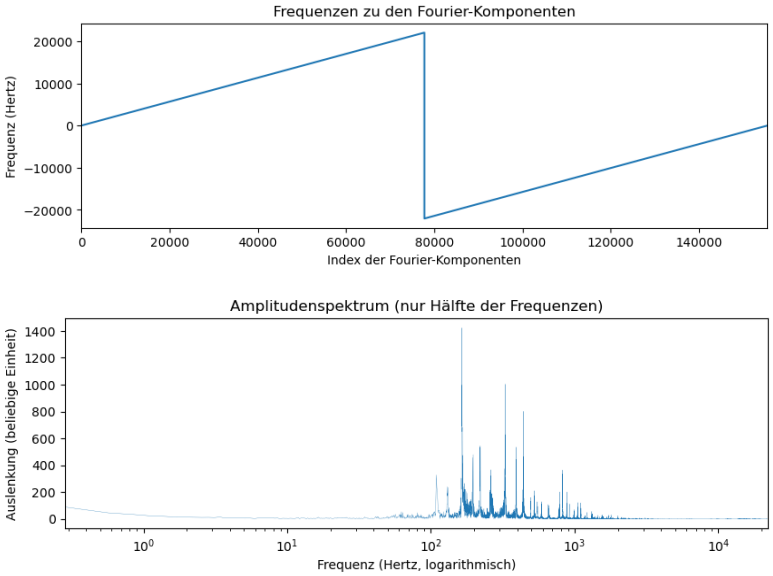
Für die Darstellung als Spektrum müssen die Kompmentenindizes durch die zugehörigen Frequenzen ersetzt werden. Da Audiosignale reelle Zahlenfolgen sind, ist die vollständige Information über das Signal bereits in er Hälfte der Fourier-Komponenten enthalten.

Beachte: Da bei der diskreten Fourier-Transformation die Signale als periodische doppelt unendliche Folgen interpretiert werden, ist es egal, ob die Indizes (und damit die Frequenzen) von 0 bis $N-1$ oder von $-\frac{N-1}{2}$ bis $\frac{N-1}{2}$ laufen. Letztere Variante liefert statt hohen Frequenzen negative Frequenzen, was jedoch aus Sicht der diskreten Fourier-Transformation völlig äquivalent ist.

```
# Frequenzen plotten
fig, ax = plt.subplots(figsize=(10, 3))
ax.plot(freq)
ax.set_xlabel('Index der Fourier -Komponenten')
ax.set_ylabel('Frequenz (Hertz)')
ax.set_xlim(0, len(freq))
ax.set_title('Frequenzen zu den Fourier -Komponenten')
plt.show()

# halbes Amplitudenspektrum plotten (logarithmische Frequenzachse)
freq_pos_mask = freq > 0
fig, ax = plt.subplots(figsize=(10, 3))
ax.semilogx(freq[freq_pos_mask], np.abs(spec[freq_pos_mask]), lw=0.2)
ax.set_xlim(freq[freq_pos_mask].min(), freq[freq_pos_mask].max())
ax.set_xlabel('Frequenz (Hertz, logarithmisch)')
ax.set_ylabel('Auslenkung (beliebige Einheit)')
```

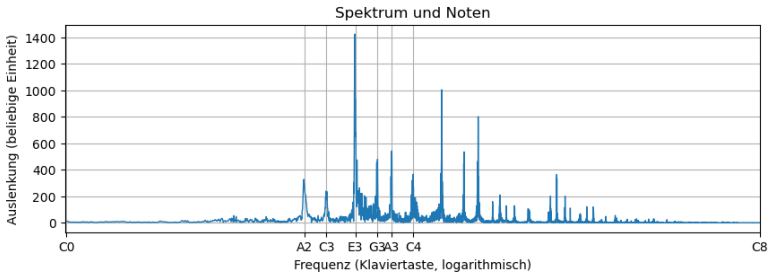
```
ax.set_title('Amplitudenspektrum (nur Hälfte der Frequenzen)')
plt.show()
```



5.4.6.3 Interpretation Im Spektrum kann man sehr deutlich Peaks bei den in der Musik üblichen Frequenzen erkennen (vgl. Frequenzen der gleichstufigen Stimmung).

```
# Frequenzbereich eines Klaviers (ohne Obertöne)
piano_mask = np.logical_and(freq > 16, freq < 4187)

# Spektrum mit Markierung der zu hörenden Noten
fig, ax = plt.subplots(figsize=(10, 3))
ax.semilogx(freq[piano_mask], np.abs(spec[piano_mask]), lw=1)
ax.set_xlim(freq[piano_mask].min(), freq[piano_mask].max())
ax.set_xticks([16.3516, 110, 130.813, 164.814, 195.998, 220, 261.626,
↪  4186.01])
ax.set_xticklabels(['C0', 'A2', 'C3', 'E3', 'G3', 'A3', 'C4', 'C8'])
ax.get_xaxis().set_tick_params(which='minor', size=0)
ax.grid()
ax.set_xlabel('Frequenz (Klaviertaste, logarithmisch)')
ax.set_ylabel('Auslenkung (beliebige Einheit)')
ax.set_title('Spektrum und Noten')
plt.show()
```



Die nicht markierten Peaks entstehen durch die Obertöne, die zu jedem Grundton erklingen.

Der Peak bei E3 ist höher als bei anderen Noten, da diese Note im Audiosignal öfter vorkommt als die anderen.

5.4.6.4 Kurzzeit-Fourier-Transformation

Die Kurzzeit-Fourier-Transformation (kurz: STFT für "short-time Fourier transform"; auch gefensterte Fourier-Transformation genannt) zerlegt das zu transformierende Signal in kleine, überlappende Teilstücke. Für jedes Teilstück wird dann eine diskrete Fourier-Transformation durchgeführt, sodass eine **Folge zeitabhängiger Spektren** entsteht. Das Zerlegen erfolgt dabei nicht durch Abschneiden, sondern durch Multiplikation mit einer glatten Funktion, z.B. einer Gauß'schen Glockenkurve. So werden zu starke Artefakte im Frequenzbereich durch die Sprünge an den Schnittstellen vermieden.

Spektrogramme lassen sich relativ leicht mit scipy.signal.ShortTimeFFT.spectrogram erzeugen.

```
# Fensterbreite und -versatz
win_width = 10000  # Breit in Samples
win_std = 5000  # Standardabweichung in Samples
hop = 200  # Versatz in Samples

# Spektrogramm erzeugen
win = scipy.signal.windows.gaussian(win_width, std=win_std, sym=True)
SFT = scipy.signal.ShortTimeFFT(win, hop=hop, fs=samples_per_sec)#,
↪  scale_to='magnitude')
s = SFT.spectrogram(data)

# Spektrogramm logarithmisch skalieren (für bessere Darstellung)
s_scaled =  np.log10(np.fmax(s, 5e7))

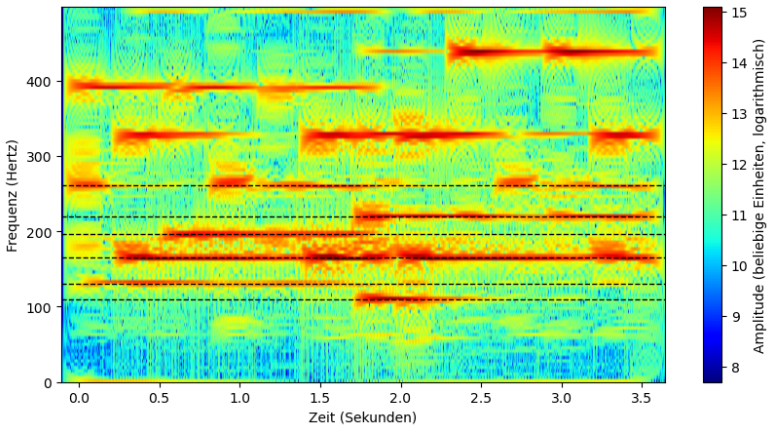
# Indizes in Sekunden und Hertz umrechnen
t_min, t_max, freq_min, freq_max = SFT.extent(len(data))
freq = np.linspace(freq_min, freq_max, s.shape[0])
freq_mask = freq < 500  # nur unteren Teil des Spektrogramms plotten
```

```
# plotten
fig, ax = plt.subplots(figsize=(10, 5))

# Spektrogramm
img = ax.imshow(
    s_scaled[freq_mask, :],
    origin='lower', aspect='auto',
    extent=(t_min, t_max, freq[freq_mask].min(),
    ↪  freq[freq_mask].max()),
    cmap='jet'
)
fig.colorbar(img, label='Amplitude (beliebige Einheiten,
↪  logarithmisch)')

# Linien für Klaviertasten
for f in [110, 130.813, 164.814, 195.998, 220, 261.626]:
    ax.plot([t_min, t_max], [f, f], ' - -k', lw=1)

ax.set_xlabel('Zeit (Sekunden)')
ax.set_ylabel('Frequenz (Hertz)')
plt.show()
```



Die farbkodierte Darstellung der Amplituden in Abhängigkeit von Zeit und Freqenz wird als **Spektrogramm** bezeichnet. Im Spektrogramm erkennt man neben den dominanten Frequenzen auch deren zeitliches Auftreten.

6 Gewöhnliche Differentialgleichungen

Theoretische Grundlagen und numerische Verfahren zum Lösen von (Systemen von) gewöhnlichen Differentialgleichungen.

6.1 Grundlagen

6.1.1 Abgrenzung und Bedeutung

Eine Gleichung, die eine Funktion und ihre Ableitungen zueinander in Beziehung setzt, heißt **Differentialgleichung**.

Betrachten zunächst nur **gewöhnliche Differentialgleichungen**, d.h. die gesuchte Funktion hängt nur von eine Veränderlichen ab.

Hängt die gesuchte Funktion von mehreren Veränderlichen ab, so spricht man von **partiellen Differentialgleichungen**. Diese werden später behandelt.

Aus der Untersuchung technischer, naturwissenschaftlicher oder auch wirtschaftswissenschaftlicher Zusammenhänge ergeben sich sehr häufig Differentialgleichungen, die die gesuchte Größe (eine Funktion) beschreiben. Mit Differentialgleichungen können verschiedenste Vorgänge modelliert und Gesetzmäßigkeiten formuliert werden. Insbesondere sind Differentialgleichungen das wichtigste und mächtigste mathematische Werkzeug der klassischen Mechanik!

Beispiel (Newton'sches Gesetz)

Bewegt sich ein Objekt der Masse m durch ein Kraftfeld $F = F(x, y, z, t)$, so gilt für dessen Flugbahn zu jeder Zeit und an jedem Ort das Newton'sche Gesetz

$$\text{Kraft} = \text{Masse} \cdot \text{Beschleunigung}.$$

Beschreiben wir den Aufenthaltsort des Objekts zur Zeit t durch die Koordinatenfunktionen $x = x(t)$, $y = y(t)$, $z = z(t)$, so kann dieses Gesetz als

$$F_1 = m\,x'', \qquad F_2 = m\,y'', \qquad F_3 = m\,z'' \tag{6.1}$$

geschrieben werden. Dies ist ein **System von Differentialgleichungen**!

Beachte: Bei der Formulierung von Differenzialgleichung verzichtet man zu Gunsten der Übersichtlichkeit meist auf die Angabe der Funktionsargumente. Mit Argumenten hätte die erste Gleichung im Beispiel die Form

$$F_1\big(x(t), y(t), z(t), t\big) = m\,x''(t). \tag{6.2}$$

Auch gibt man den Definitionsbereich der gesuchten Funktion nur an, wenn dieser im Vorfeld bekannt ist. Meist ergibt sich der (maximale) Definitionsbereich jedoch erst im Laufe des Lösungsprozessen.

Innerhalb der Mathematik können Differentialgleichungen zur Beschreibung wichtiger Funktionen dienen.

Beispiel (Definition Exponentialfunktion)

Die Funktion $y = y(x)$, die

$$y' = y \qquad \text{und} \qquad y(0) = 1 \tag{6.3}$$

erfüllt, heißt Exponentialfunktion.

6.1.2 Begriffe

Der Grad der höchsten in einer Differentialgleichung auftretenden Ableitung heißt **Ordnung** der Differentialgleichung.

Steht die höchste auftretende Ableitung allein auf einer Seite der Gleichung und befinden sich auf der anderen Seite nur niedrigere Ableitungen der gesuchten Funktion, so spricht man von einer expliziten Differentialgleichung. Andernfalls von einer impliziten Differentialgleichung.

Explizite Differentialgleichungen können also in der Form

$$y^{(n)} = f\big(x, y, y', \ldots, y^{(n-1)}\big) \tag{6.4}$$

mit einer Funktion $f : \mathbb{R}^{n+1} \to \mathbb{R}$ geschrieben werden.

Implizite Differentialgleichungen haben die Form

$$F\big(x, y, y', \ldots, y^{(n)}\big) = 0 \tag{6.5}$$

mit einer Funktion $F : \mathbb{R}^{n+2} \to \mathbb{R}$.

Merke

Eine Funktion $y : (a, b) \to \mathbb{R}$ heißt **Lösung** einer gegebenen Differentialgleichung, wenn y hinreichend oft differenzierbar ist und y die Differentialgleichung für alle $x \in (a, b)$ erfüllt.

Fast alle Differentialgleichungen besitzen unendlich viele Lösungen. Sucht man alle Lösungen einer Differentialgleichung, so spricht man von der **allgemeinen Lösung**. Dies ist meist eine Formel, in der einige Parameter enthalten sind. Interessiert man sich nur für irgendeine Lösung oder für eine konkrete Lösung, die Zusatzbedingungen erfüllt, so spricht man von einer **speziellen Lösung**.

Differentialgleichungen stehen in der Praxis selten allein, sondern meist gemeinsam mit Zusatzbedingungen. Im Wesentlichen treten zwei Typen von Zusatzbedingungen auf:

- **Anfangsbedingungen**: Die gesuchte Funktion und eventuell auch einige Ableitungen sind an einer festen Stelle x_0 bekannt. Steht x für die Zeit, so ist der

Zustand am Anfang des betrachteten Zeitraums meist bekannt und die weitere Entwicklung soll ermittelt werden.

- **Randbedingungen**: Die gesuchte Funktion ist an einer oder mehreren Stellen vorgegeben. Steht x für eine Ortskoordinate, so ist der Zustand an den Rändern des betrachteten Intervalls meist bekannt und der Zustand im Inneren ist zu ermitteln.

Differentialgleichungen mit Anfangsbedingungen heißen **Anfangswertprobleme (AWP)**. Differentialgleichungen mit Randbedingungen heißen **Randwertprobleme (RWP)**.

Beispiel (einseitig fest eingespannter Balken)

Das eine Ende eines Balkens der Länge l sei waagerecht und unbeweglich eingespannt. Das andere Ende sei frei beweglich. Legen wir den Ursprung eines zweidimensionalen Koordinatensystems an das feste Ende und wählen wir die x-Achse in Balkenrichtung und die y-Achse nach unten, so kann die Form des Balkens unter Lasteinwirkung als positive Funktion $y : [0, l] \to [0, \infty)$ dargestellt werden. Die Biegelinie y kann als Lösung des Anfangswertproblems

$$y'' = \frac{F}{E\,I}\,(l - x), \qquad y(0) = 0, \qquad y'(0) = 0 \tag{6.6}$$

berechnet werden. Dabei ist F eine am Balkenende von oben wirkende Kraft, E der E-Modul und I das Flächenträgheitsmoment des Balkenquerschnitts.

Hier handelt es sich um eine explizite Differentialgleichung zweiter Ordnung. Man kann zeigen, dass die allgemeine Lösung durch

$$y(x) = \frac{F}{E\,I}\left(\frac{l}{2}\,x^2 - \frac{1}{6}\,x^3\right) + c_1\,x + c_2, \quad c_1, c_2 \in \mathbb{R} \tag{6.7}$$

gegeben ist. Einsetzen der Anfangsbedingungen liefert die spezielle Lösung

$$y(x) = \frac{F}{E\,I}\left(\frac{l}{2}\,x^2 - \frac{1}{6}\,x^3\right). \tag{6.8}$$

Die Auslenkung am freien Balkenende ist somit

$$\frac{1}{3}\,\frac{F\,l^3}{E\,I}. \tag{6.9}$$

Beispiel (beidseitig gelenkig gelagerter Balken)

Die Biegelinie eines beidseitig gelenkig gelagerten Balkens der Länge l ergibt sich

als Lösung des Randwertproblems

$$y'' = -\frac{q}{2\,E\,I}\,(l\,x - x^2), \qquad y(0) = 0, \qquad y(l) = 0 \tag{6.10}$$

(Koordinatensystem wie im vorhergehenden Beispiel). Dabei ist $q = q(x)$ die Kraftdichte (Kraft pro Längeneinheit) der auf dem Balken verteilten Lasten. Nehmen hier der Einfachheit halber an, dass q konstant ist, so kann man zeigen, dass

$$y(x) = -\frac{q}{2\,E\,I}\,\left(\frac{l}{6}\,x^3 - \frac{1}{12}\,x^4\right) + c_1\,x + c_2, \quad c_1, c_2 \in \mathbb{R} \tag{6.11}$$

die allgemeine Lösung ist. Einsetzen der Anfangsbedingungen liefert die spezielle Lösung

$$y(x) = -\frac{q}{2\,E\,I}\,\left(\frac{l}{6}\,x^3 - \frac{1}{12}\,x^4 - \frac{l^3}{12}\,x\right). \tag{6.12}$$

Die Durchbiegung in Balkenmitte ist somit

$$\frac{5}{384}\,\frac{q\,l^4}{E\,I}. \tag{6.13}$$

6.1.3 Klassifikation

Es gibt zahlreiche verschiedene Verfahren zum Lösen von Differentialgleichungen. Welches Verfahren zum Ziel führt, kann anhand gewisser Eigenschaften von Differentialgleichungen herausgefunden werden. Solche Eigenschaften sind:

- Ordnung: Man unterscheidet Differentialgleichungen erster Ordnung und höherer Ordnung.
- Linearität: Eine Differentialgleichung heißt **linear**, wenn y und seine Ableitungen nur allein (eventuell mit Faktor davor) und nie als Produkte, Potenzen usw. vorkommen. Eine lineare Differentialgleichung hat also stets die Form

$$a_n(x)\,y^{(n)} + \cdots + a_1(x)\,y' + a_0(x)\,y = b(x). \tag{6.14}$$

Bei linearen Differentialgleichungen unterscheidet man weiter:

- Homogenität: Eine lineare Differentialgleichung heißt **homogen**, wenn das Absolutglied b (auch Störglied genannt) überall Null ist. Andernfalls heißt sie **inhomogen**.
- Konstante Koeffizienten: Man unterscheidet lineare Differentialgleichungen mit konstanten Koeffizienten, d.h. die $a_0, \ldots, a_n$ sind Zahlen, und mit nicht konstanten Koeffizienten, d.h. die $a_0, \ldots, a_n$ sind Funktionen.

Soll eine Differentialgleichung gelöst werden, so ist diese zunächst zu klassifizieren. Anhand der festgestellten Eigenschaften kann anschließend ein Lösungsverfahren ausgewählt werden. Dies gilt vor allem für das manuelle Lösen. Für das numerische Lösen spielt die Klassifikation nur eine untergeordnete Rolle.

6.1.4 Lösungsmethoden

Differentialgleichungen wurden früher stets **analytisch**, d.h. manuell, gelöst. Man suchte also eine Formel, die die Lösungsfunktion exakt beschreibt. Entsprechend vielfältige manuelle Lösungsverfahren existieren und werden auch heute in einfachen Fällen noch eingesetzt. Praktisch werden Differentialgleichungen heute **numerisch** mit dem Computer gelöst. Hier steht ebenfalls eine große Anzahl an Verfahren zur Verfügung, die jedoch stets nur eine Näherung der Lösung liefern, nie die exakte Lösung. Computeralgebrasysteme können nicht zu komplexe Differentialgleichungen auch **symbolisch** lösen.

Manchmal ist man gar nicht an der genauen Lösung interessiert, sondern möchte nur einige Eigenschaften der Lösung in Erfahrung bringen. Eine solche Eigenschaft kann zum Beispiel das asymptotische Verhalten der gesuchten Funktion sein, wenn x für die Zeit steht.

Ist nur der grobe Verlauf der Lösungskurven von Interesse, können auch grafische Lösungsverfahren in Betracht kommen.

Konzentrieren uns hier auf das numerische Lösen und auf das grafische “Lösen”, wobei letzteres ebenfalls mit Hilfe des Computers erfolgt und mit wenig Aufwand einen ersten Eindruck von der zu erwartenden Lösung verschafft.

6.2 Existenz und Eindeutigkeit von Lösungen

6.2.1 Problembeschreibung

Es gibt keine allgemein gültige Aussage zum Umfang der Lösungsmengen von Differentialgleichungen. Manche Differentialgleichungen besitzen gar keine Lösung, andere besitzen genau eine oder nur wenige Lösungen und die meisten besitzen unendlich viele Lösungen.

Oft sehen verschiedene Lösungen einer Differentialgleichung sehr ähnlich aus (Funktionenschar). Manchmal treten aber auch grundverschiedene Funktionen gemeinsam als Lösungen auf.

Beispiel

Einige Differentialgleichungen mit unterschiedlich vielen Lösungen:

- $(y')^2 + 1 = 0$ hat keine Lösung, da $(y')^2 \geq 0$.
- $(y')^2 + y^2 = 0$ hat genau eine Lösung: $y(x) = 0$.
- $(y')^2 - 4\,y = 0$ hat unendlich viele Lösungen: die Nullfunktion, die Parabeln $y(x) = (x - c)^2$ für $c \in \mathbb{R}$ und diverse Kombinationen aus Nullfunktion und Parabelästen.
- $(y')^2 + x\,y' - y = 0$ hat zwei verschiedene Typen von Lösungen: Die Schar $y(x) = c\,x + c^2$ mit $c \in \mathbb{R}$ und die **singuläre Lösung** $y(x) = -\frac{x^2}{4}$.

Bei Differentialgleichungen mit unendlich vielen Lösungen können Zusatzforderungen (Anfangs- oder Randbedingungen) an die gesuchte Funktion gestellt werden. Dann stellt sich jedoch wieder die Frage, ob die Differentialgleichung samt Zusatzbedingungen immer noch lösbar ist und, wenn ja, wie viele Lösungen sie besitzt.

Oft ist auch nicht klar, auf welchem Intervall die Lösungen "leben". Kann man auf ganz $\mathbb{R}$ definierte Lösungen finden oder nur auf einem kleinen Intervall?

6.2.2 Grundlegende Sätze

Es gibt eine Reihe von mathematischen Sätzen, die sich mit der Existenz und der Eindeutigkeit von Lösungen zu Differentialgleichungen beschäftigen. Beschränken uns auf zwei solche Aussagen über das explizite Anfangswertproblem

$$y' = f(x, y), \qquad y(x_0) = y_0 \tag{6.15}$$

erster Ordnung.

Merke (Satz von Peano)

Sei f stetig auf dem Rechteck

$$(x_0 - a, x_0 + a) \times (y_0 - b, y_0 + b). \tag{6.16}$$

Dann gibt es eine Zahl $c > 0$, sodass obiges Anfangswertproblem auf dem Intervall $(x_0 - c, x_0 + c)$ mindestens eine Lösung besitzt.

Die Zahl c im Satz kann man konkret angeben. Sie ist der kleinere der beiden Werte

$$a \qquad \text{und} \qquad \frac{b}{\max f}, \tag{6.17}$$

wobei $\max f$ den größten Wert von f auf dem Rechteck bezeichne.

Merke (Satz von Picard-Lindelöf)

Seien die Voraussetzungen des Satzes von Peano erfüllt und besitze f eine stetige partielle Ableitung nach y. Dann ist die Lösung aus dem Satz von Peano eindeutig bestimmt.

Die beiden Sätze gelten analog auch für Anfangswertprobleme höherer Ordnung. Die Anzahl der Anfangsbedingungen muss gerade der Ordnung der Differentialgleichung entsprechen:

$$y^{(n)} = f\big(x, y, y', \ldots, y^{(n-1)}\big), \tag{6.18}$$

$$y(x_0) = y_{0,0}, \quad y'(x_0) = y_{0,1}, \quad \ldots, \quad y^{(n-1)}(x_0) = y_{0,n-1}. \tag{6.19}$$

Das Rechteck im Satz von Peano ist durch einen $(n+1)$-dimensionalen Hyperquader zu ersetzen.

Beispiel

Für das Anfangswertproblem

$$y'' + 3\,y' - 5\,y = x^2, \qquad y(1) = 2, \qquad y'(1) = 3 \tag{6.20}$$

sind die Voraussetzungen der beiden Sätze erfüllt. Das Anfangswertproblem ist also eindeutig lösbar (IDV 610).

Beispiel

Das Anfangswertproblem

$$y' = 3\,y^{\frac{2}{3}}, \qquad y(0) = 0 \tag{6.21}$$

besitzt unendlich viele Lösungen, denn für jedes $c > 0$ ist

$$y(x) = \begin{cases} 0, & \text{für } x \leq c, \\ (x-c)^3, & \text{für } x > c. \end{cases} \tag{6.22}$$

eine Lösung. Außerdem gibt es auch noch weitere Lösungen.

Da

$$f(x,y) = 3\,y^{\frac{2}{3}} \tag{6.23}$$

in $(0,0)$ bzgl. y nicht partiell differenzierbar ist, ist der Satz von Picard-Lindelöf nicht anwendbar.

6.2.3 Eigenwertprobleme

Enthält eine Differentialgleichung oder eine ihrer Zusatzbedingungen einen Parameter, so kann die Lösbarkeit der Differentialgleichung vom Wert des Parameters abhängen. Die Frage nach Parameterwerten für welche eine Differentialgleichung (nicht triviale) Lösungen besitzt, bezeichnet man als **Eigenwertproblem**.

Beispiel (Euler'sche Knickstab, zweiter Euler'scher Knickfall)

Auf einen an beiden Enden gelenkig gelagerten Stab der Länge l wirkt eine Kraft F entlang der Stabachse (Druckstab). Dadurch kann es zum Ausknicken und eventuell zum Bruch des Stabes kommen. Die Auslenkung $y(x)$ des Balkens an der Stelle $x \in (0,l)$ kann durch die Differentialgleichung

$$y'' = -\frac{F}{E\,I}\,y \tag{6.24}$$

mit den Randwerten

$$y(0) = 0, \qquad y(l) = 0 \tag{6.25}$$

beschrieben werden. Dabei sind E der E-Modul und I das Flächenträgheitsmoment. Man kann zeigen, dass das Randwertproblem nur für

$$F = E\,I\,\left(\frac{k\,\pi}{l}\right)^2, \qquad k = 1, 2, \ldots \tag{6.26}$$

nicht triviale Lösungen besitzt:

$$y(x) = c\,\sin\left(\frac{k\,\pi}{l}\,x\right), \qquad c \in \mathbb{R}. \tag{6.27}$$

Die Kraft $F = E\,I\,(\frac{\pi}{l})^2$ zu $k = 1$ heißt *Euler'sche Knickkraft*.

6.3 Systeme von Differentialgleichungen

Häufig werden technische oder physikalische Prozesse nicht nur durch eine Differentialgleichung bestimmt, sondern durch ein System aus mehreren Differentialgleichungen.

6.3.1 Systeme erster Ordnung

Merke

Sei $f : \mathbb{R}^{(n+1)} \to \mathbb{R}$ eine Funktion. Suchen wir eine vektorwertige Funktion $y : (a, b) \to \mathbb{R}^n$ mit

$$y' = f(x, y) \tag{6.28}$$

so spricht man von einem **Differentialgleichungssystem erster Ordnung**. Ausführliche Schreibweise:

$$\begin{aligned} y_1' &= f_1(x, y_1, \ldots, y_n), \\ &\vdots \\ y_n' &= f_n(x, y_1, \ldots, y_n). \end{aligned} \tag{6.29}$$

Zu diesem werden üblicherweise n Anfangsbedingungen

$$y(x_0) = y_0 \tag{6.30}$$

für ein $y_0 \in \mathbb{R}^n$ vorgegeben.

6.3.2 Systeme höherer Ordnung

Systeme höherer Ordnung spielen nur eine Nebenrolle, da sie in Systeme erster Ordnung umgewandelt werden können.

Beispiel

Hatten in Beispiel (Newton'sches Gesetz) bereits das Newton'sche Gesetz als System von Differentialgleichungen formuliert:

$$F(t, x) = m\, x''(t), \quad t \geq 0, \quad x(0) = x_0, \quad x'(0) = v_0, \tag{6.31}$$

wobei $m > 0$ (Masse), $F : \mathbb{R}^4 \to \mathbb{R}$ (Kraftfeld), $x_0 \in \mathbb{R}^3$ (Anfangsposition) und $v_0 \in \mathbb{R}^3$ (Anfangsgeschwindigkeit) gegeben sind und die Bahn $x : [0, \infty) \to \mathbb{R}^3$ des Massepunkts gesucht ist.

Hierbei handelt es sich um ein System von drei Differentialgleichungen zweiter Ordnung.

Setzen wir

$$\begin{aligned} v_1 &:= x_1', \\ v_2 &:= x_2', \\ v_3 &:= x_3', \end{aligned} \tag{6.32}$$

so kann das ursprüngliche System als

$$\begin{aligned} F_1 &= m\, v_1', \\ F_2 &= m\, v_2', \\ F_3 &= m\, v_3' \end{aligned} \tag{6.33}$$

geschrieben werden. Wir erhalten also ein zum ursprünglichen System äquivalentes System von sechs Differentialgleichungen erster Ordnung. Die Anfangsbedingungen sind

$$x(0) = x_0, \qquad v(0) = v_0. \tag{6.34}$$

Nach dem Lösen dieses Systems können die zusätzlich eingeführten Funktionen v_1, v_2, v_3 wieder verworfen werden.

Die im Beispiel verwendete Idee zur Umwandlung von Systemen höherer Ordnung in Systeme erster Ordnung kann man allgemein anwenden. Insbesondere lassen sich so auch einzelne Gleichungen höherer Ordnung in Systeme erster Ordnung umwandeln.

Merke

Die Differentialgleichung n-ter Ordnung

$$y^{(n)} = f(x, y', y'', \ldots, y^{(n-1)}) \tag{6.35}$$

ist äquivalent zum System erster Ordnung

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= y_3, \\ &\vdots \\ y_{n-1}' &= y_n, \\ y_n &= f(x, y_1, y_2, \ldots, y_{n-1}) \end{aligned} \tag{6.36}$$

mit n Gleichungen (IDV 620).

Die Randbedingungen

$$y(x_0) = a_0, \quad y'(x_0) = a_1, \quad \ldots, \quad y^{(n-1)} = a_{n-1} \tag{6.37}$$

der Gleichung n-ter Ordnung entsprechen gerade den Randbedingungen

$$y_1(x_0) = x_0, \quad y_2(x_0) = a_1, \quad \ldots, \quad y_n(x_0) = a_{n-1} \tag{6.38}$$

für das System erster Ordnung.

Beachte: Jede Differentialgleichung höherer Ordnung kann in ein System erster Ordnung überführt werden. Allerdings gibt es Systeme erster Ordnung, die nicht in eine Differentialgleichung höherer Ordnung überführt werden können. Ob und wie ein System in eine Gleichung höherer Ordnung überführt werden kann, ist im Einzelfall zu klären. Es gibt dafür kein allgemein anwendbares Vorgehen.

Beispiel

Das System

$$\begin{aligned} y_1' &= 5\,y_1 + y_2, \\ y_2' &= -4\,y_1 + y_2 \end{aligned} \tag{6.39}$$

ist äquivalent zu Gleichung

$$y'' - 6\,y' + 9\,y = 0. \tag{6.40}$$

zweiter Ordnung (IDV 625).

6.3.3 Lineare Systeme erster Ordnung

Ein wichtige Rolle in Anwendungen spielen lineare Systeme erster Ordnung. Dieses besitzen die Struktur

$$\begin{aligned} y_1' &= a_{1,1}(x)\,y_1 + \cdots + a_{1,n}\,y_n + b_1(x), \\ &\vdots \\ y_n' &= a_{n,1}(x)\,y_1 + \cdots + a_{n,n}\,y_n + b_1(x) \end{aligned} \tag{6.41}$$

mit gegebenen Funktionen $a_{i,j}$ und b_i.

Setzt man

$$A := \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix} \qquad \text{und} \qquad b := \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}, \tag{6.42}$$

so kann ein lineares System erster Ordnung in der Form

$$y' = A(x)\,y + b \tag{6.43}$$

geschrieben werden.

Beispiel

In obigem Beispiel hatten wird das System

$$\begin{aligned} x_1' &= v_1, \\ x_2' &= v_2, \\ x_3' &= v_3, \\ v_1' &= \frac{F_1}{m}, \\ v_2' &= \frac{F_2}{m}, \\ v_3' &= \frac{F_3}{m} \end{aligned} \tag{6.44}$$

erhalten. Die Systemmatrix ist

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.45}$$

und die rechte Seite ist

$$b = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{F_1}{m} \\ \frac{F_2}{m} \\ \frac{F_3}{m} \end{bmatrix}. \tag{6.46}$$

6.3.4 Autonome Systeme

Ein System von Differentialgleichungen heißt **autonom**, wenn die in der expliziten Darstellung die rechte Seite nicht explizit von der Veränderlichen abhängt, wenn es also von der Form

$$y' = f(y) \tag{6.47}$$

ist mit einer Funktion $F : \mathbb{R}^n \to \mathbb{R}^n$.

Für autonome Systeme existieren spezielle Lösungsverfahren, welche die zusätzliche Strukturinformation vorteilhaft nutzen können.

Beispiel

Bleibt das Kraftfeld im obigen Beispiel über den betrachteten Zeitraum unverän-

dert, so ist das aus dem Newton'schen Gesetz resultierende Differentialgleichungssystem autonom.

6.4 Grafisches Lösen

Um einen ersten Eindruck von der Gestalt der Lösungen einer Differentialgleichung zu bekommen, kann diese grafisch gelöst werden. Dies ist möglich bei

- Gleichungen erster Ordnung (2d-Grafik),
- Systemen erster Ordnung mit zwei Gleichungen (3d-Grafik),
- autonomen Systemen erster Ordnung mit zwei Gleichungen (2d-Grafik),
- autonomen Systemen erster Ordnung mit drei Gleichungen (3d-Grafik).

Dabei ist die Arbeit mit 3d-Grafiken allerdings praktisch meist zu aufwendig und zu unübersichtlich.

6.4.1 Idee

Liegt die zu lösende Differentialgleichung in expliziter Form

$$y' = f(x, y) \tag{6.48}$$

mit gegebenem $f : \mathbb{R}^2 \to \mathbb{R}$ vor, so kennt man an jedem Punkt $(x, y(x))$ einer Lösung y auch deren Anstieg, denn dieser ist gerade $f(x, y(x))$. Für jeden Punkt (x_0, y_0) der Ebene kann man also eine Aussage der Form

> Wenn eine Lösung durch den Punkt (x_0, y_0) verläuft, so hat sie dort den Anstieg $f(x_0, y_0)$.

machen. Visualisiert man diesen Anstieg an hinreichend vielen Punkten, so bekommt man einen guten Eindruck vom Verlauf der Lösungskurven.

Die Abbildung

$$(x, y) \mapsto \begin{bmatrix} 1 \\ f(x, y) \end{bmatrix}, \quad (x, y) \in \mathbb{R}^2 \tag{6.49}$$

heißt **Richtungsfeld** der Differentialgleichung.

6.4.2 Umsetzung für Gleichungen

Geben uns im Rechteck $[a, b]\times[c, d]$ für geeignet gewählte Stützstellenanzahlen $n_x, n_y \in \mathbb{N}$ ein regelmäßiges Gitter von Punkten (x_k, y, l) mit

$$x_k := k\,\frac{b-a}{n_x}, \quad k = 0, \ldots, n_x, \qquad y_l := l\,\frac{b-a}{n_y}, \quad l = 0, \ldots, n_y \tag{6.50}$$

vor. An jedem Gitterpunkt wird der Vektor

$$\begin{bmatrix} 1 \\ f(x_k, y_l) \end{bmatrix} \tag{6.51}$$

gezeichnet.

Die Länge der Vektoren ist von untergeordnetem Interesse, da diese aus der Richtung folgt (je steiler der Vektor, das größer die Länge). Wichtig ist nur die Richtung der Vektoren, sodass man auch mit normierten Vektoren arbeiten kann.

In Python kann das Richtungsfeld leicht mit matplotlib.pyplot.quiver visualisiert werden. Die Länge der Vektoren wird so skaliert, dass die Darstellung übersichtlich bleibt.

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

# Funktion mit zwei Argumenten (Differentialgleichung)
f = lambda x, y: x * y

# Bildausschnitt und Punktdichte im Gitter
a, b = -3, 3
c, d = -2, 2
nx = 15
ny = 15

# x = Matrix der x -Koordinaten aller Gitterpunkte, y analog
x, y = np.meshgrid(
    np.linspace(a, b, nx),
    np.linspace(c, d, ny)
)

# Richtungen (normiert)
length = np.sqrt(1 + f(x, y) ** 2)  # Länge der Vektoren
dir_x = np.ones_like(x) / length
dir_y = f(x, y) / length

# Lösungskurven als Beispiel
solution_x = np.linspace(a, b, 100)#
solution1_y = np.exp(0.5 * solution_x ** 2)
solution2_y = -0.25 * np.exp(0.5 * solution_x ** 2)

# Plot
fig, ax = plt.subplots()

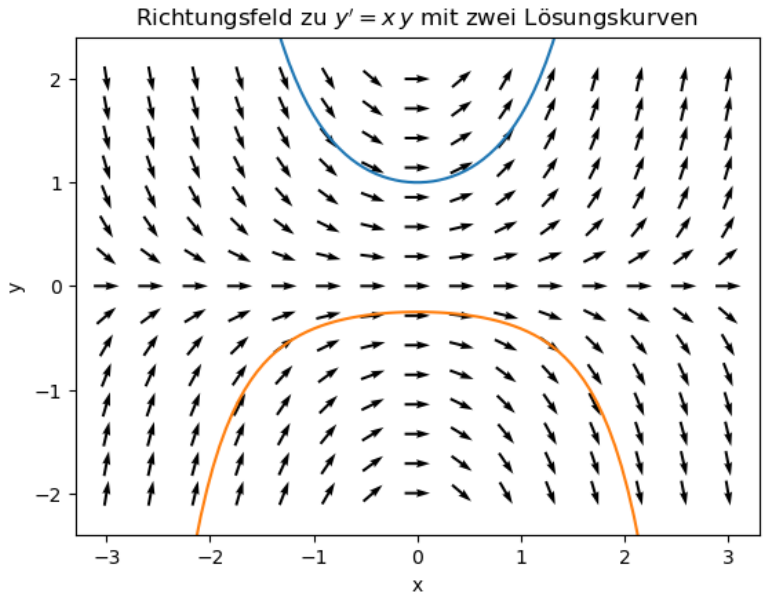
ax.quiver(x, y, dir_x, dir_y, pivot='middle', angles='xy')

ax.plot(solution_x, solution1_y)
ax.plot(solution_x, solution2_y)
```

```
ax.set_aspect('equal', 'box')  # gleiche Einteilung beider Achsen
↪  (keine Verzerrung)
ax.set_ylim(c - 0.1 * (d - c), d + 0.1 * (d - c))

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Richtungsfeld zu $y\'=x\\,y$ mit zwei Lösungskurven')

plt.show()
```



6.4.3 Umsetzung für Systeme

Für ein explizites System aus zwei Differentialgleichungen

$$\begin{aligned} y_1' &= F_1(x, y_1, y_2), \\ y_2' &= F_2(x, y_1, y_2) \end{aligned} \tag{6.52}$$

kann das Richtungsfeld im dreidimensionalen Raum dargestellt werden. Analog zum Vorgehen bei einer einzelnen Differentialgleichung wird hier jedem Punkt $(x, y_1, y_2) \in \mathbb{R}^3$ der Vektor

$$\begin{bmatrix} 1 \\ F_1(x, y_1, y_2) \\ F_2(x, y_1, y_2) \end{bmatrix} \tag{6.53}$$

zugeordnet. Die Lösungskurven folgen dann wieder der durch die Vektoren gegebenen "Strömung".

Beispiel (Räuber-Beute-Modell, Lotka 1925, Volterra 1926))

Betrachten ein einfaches abgeschlossenen Ökosystem (z.B. Insel), in dem es Beutetiere gibt, die stets genug Nahrung finden, und Raubtiere, die sich ausschließlich von den Beutetieren ernähren. Zu jeder Zeit $t \geq 0$ bezeichnen wir die Anzahl der Beutetiere mit $B(t)$ und die Anzahl der Raubtiere mit $R(t)$. Folgende Bedingungen könne für die Beschreibung des Zusammenhangs zwischen den Anzahlen von Raub- und Beutetieren formuliert werden:

- Ohne Nahrung sterben die Raubtiere mit der Rate $\alpha_1 \in (0,1)$.
- Die Vemehrungsrate der Raubtiere ist proportional zum Nahrungsangebot mit Proportionalitätsfaktor $\beta_1 > 1$.
- Ohne Angriffe durch Raubtiere vermehren sich die Beutetiere mit der Rate $\alpha_2 > 1$.
- Die Sterberate der Beutetiere durch Raubtierangriffe ist proportional zur Raubtieranzahl mit Proportionalitätsfaktor $\beta_2 \in (0,1)$.

Diese Bedingungen führen für die Bestimmung der Funktionen $B : [0,\infty) \to [0,\infty)$ und $R : [0,\infty) \to [0,\infty)$ auf das Differentialgleichungssystem

$$\begin{aligned} R'(t) &= R(t)\,\big(-\alpha_1 + \beta_1\, B(t)\big), \\ B'(t) &= B(t)\,\big(\alpha_2 - \beta_2\, R(t)\big). \end{aligned} \tag{6.54}$$

Stellen das Richtungsfeld des Lotka-Volterra-Modells grafisch dar, um einen Eindruck vom Verhalten der Lösungskurven zu bekommen. Allerdings wird die dreidimensionale Darstellung meist sehr unübersichtlich.

Beim Lotka-Volterra-Modell handelt es sich um ein autonomes System von Differentialgleichungen. Das Richtungsfeld ist bezüglich der Zeit also konstant, sodass die dritte Dimension (Zeitachse) kein zusätzlichen Informationen enthält. Der Verlauf der Lösungskurven kann deutlich übersichtlicher aus dem zweidimensionalen Richtungsfeld zur Zeit $t = 0$ (oder jeder anderen Zeit) entnommen werden. Dazu wird die Zeitkoordinate der Richtungen einfach ignoriert.

```
# Differentialgleichung
alpha1 = 0.75
beta1 = 1.25
alpha2 = 0.6
beta2 = 1.4
f1 = lambda R, B: R * ( -alpha1 + beta1 * B)
f2 = lambda R, B: B * (alpha2 - beta2 * R)

# Bildausschnitt und Punktdichte im Gitter
cR, dR = 0, 2
```

```
cB, dB = 0, 2
nR = 15
nB = 15

# R = Matrix der R -Koordinaten aller Gitterpunkte, B analog
R, B = np.meshgrid(
    np.linspace(cR, dR, nR),
    np.linspace(cB, dB, nB)
)

# Richtungen (normiert)
length = np.sqrt(1 + f1(R, B) ** 2 + f2(R, B) ** 2)  # Länge der
↪  Vektoren
dir_R = f1(R, B) / length
dir_B = f2(R, B) / length

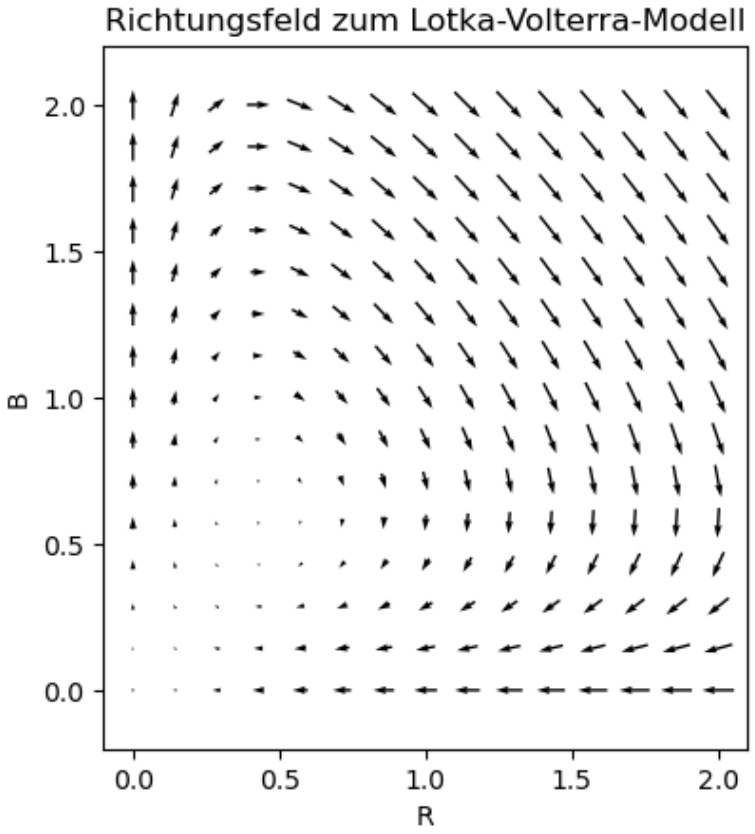
# Plot
fig, ax = plt.subplots()

ax.quiver(R, B, dir_R, dir_B, pivot='middle', angles='xy')

ax.set_aspect('equal', 'box')  # gleiche Einteilung beider Achsen
↪  (keine Verzerrung)
ax.set_ylim(cR - 0.1 * (dR - cR), dR + 0.1 * (dR - cR))
ax.set_ylim(cB - 0.1 * (dB - cB), dB + 0.1 * (dB - cB))

ax.set_xlabel('R')
ax.set_ylabel('B')
ax.set_title('Richtungsfeld zum Lotka -Volterra -Modell')

plt.show()
```



Für autonome Systeme mit drei Gleichungen kann man eine entsprechende dreidimensionale Darstellung erstellen, die aber aufgrund der mangelhaften Übersichtlichkeit praktisch kaum relevant ist.

6.5 Euler-Verfahren

Das explizite und das implizite Euler-Verfahren setzen die naheliegende Idee um, den Ableitung y' durch einen Differenzenquotient auf einen äquidistanten Gitter zu ersetzen (vgl. numerische Differentiation). Damit gehören sie zu den einfachsten Verfahren für das numerische Lösen von gewöhnlichen Differentialgleichungen und Systemen von Differentialgleichungen.

Das explizite Euler-Verfahren ist praktisch kaum relevant, da die Lösungen zu ungenau sind. Das implizite Euler-Verfahren ist hingegen praktisch einsetzbar.

6.5.1 Explizites Euler-Verfahren

Führen das Verfahren zunächst für eine einzelne Gleichung

$$y' = f(x, y), \qquad y(x_0) = y_0 \tag{6.55}$$

einschließlich Anfangswert ein.

6.5.1.1 Formulierung Wählen im Intervall $[x_0, \infty)$ ein äquidistantes Gitter mit Stützstellenabstand $h > 0$:

$$x_k := x_0 + k\,h, \qquad k = 0, 1, 2, \ldots. \tag{6.56}$$

Für $k = 0, 1, 2, \ldots$ können damit $y'(x_k)$ durch den üblichen Vorwärtsdifferenzenquotient ersetzen und erhalten damit für die Differentialgleichung an den Stellen $x_0, x_1, x_2, \ldots$:

$$\frac{y(x_{k+1}) - y(x_k)}{h} = f(x_k, y(x_k)). \tag{6.57}$$

Mit

$$y_k := y(x_k), \qquad k = 1, 2, \ldots, \tag{6.58}$$

liefert Umstellen nach $y(x_{k+1})$ die Iterationsvorschrift für das explizite Euler-Verfahren:

$$y_{k+1} = y_k + h\,f(x_k, y_k), \tag{6.59}$$

wobei y_0 durch die Anfangsbedingung $y(x_0) = y_0$ der Differentialgleichung gegeben ist.

Dieses Verfahren folgt ausgehend vom Punkt (x_0, y_0) schrittweise dem Richtungsfeld (IDV 630).

6.5.1.2 Eigenschaften Das explizite Euler-Verfahren liefert nur für kleine Schrittweiten h und auch nur für wenige Schritte brauchbare Lösungen. Je größer h ist oder je mehr Schritte ausgeführt werden, desto größer wird die Differenz zwischen exakter und numerisch berechneter Lösung. Man kann zeigen, dass stets

$$|y(x_k) - y_k| \le c_1\,h\left(\mathrm{e}^{c_2\,h\,k} - 1\right) \tag{6.60}$$

gilt mit Konstanten $c_1, c_2 > 0$ die nur von f abhängen. Diese Abschätzung liefert zwei Erkenntnisse:

- Für feste Schrittweite h kann die Differenz zwischen berechnetem und exaktem Funktionswert exponentiell bezüglich der Schrittanzahl wachsen.
- Betrachtet man immer kleinere Schrittweiten $h \to 0$, wählt dabei aber k in Abhängigkeit von h so, dass x_k stets die selbe Stelle x beschreibt (also $h\,k$ konstant ist), so geht der Fehler bei $|y(x) - y_k|$ gegen Null. Genauer gilt

$$|y(x) - y_k| \leq c_3\, h \qquad \text{bei } h \to 0. \tag{6.61}$$

  Halbieren der Schrittweite führt also beispielsweise zum Halbieren des Fehlers.

Beispiel

Wollen das Anfangswertproblem

$$y' = \sin x - \frac{y}{t}, \qquad y(1) = 1 \tag{6.62}$$

numerisch mit dem explizten Euler-Verfahren lösen. Die exakte Lösung (zu Vergleichszwecken) ist

$$y(x) = \frac{\sin x - x\,\cos x + 1 - \sin 1 + \cos 1}{x}. \tag{6.63}$$

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

f = lambda x, y: np.sin(x) - y / x
x0 = 1
y0 = 1

x_max = x0 + 8  # Intervall, auf dem die Lösung berechnet werden
↪  soll
h = 0.5  # Schrittweite

# Listen für berechnete Werte
x = [x0]  # zunächst nur Startwert
y = [y0]

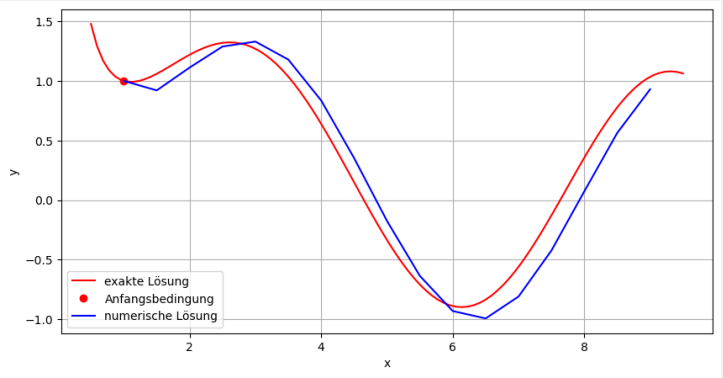
# Berechnung
while x[ -1] < x_max:
    x_new = x[ -1] + h
    y_new = y[ -1] + h * f(x[ -1], y[ -1])
    x.append(x_new)
    y.append(y_new)
```

```
# Plot
fig, ax = plt.subplots(figsize=(10, 5))

sol_x = np.linspace(x0 - 0.5, x_max + 0.5, 100)
sol_y = (np.sin(sol_x) - sol_x * np.cos(sol_x) + 1 - np.sin(1) +
↪  np.cos(1)) / sol_x
ax.plot(sol_x, sol_y, ' -r', label='exakte Lösung')
ax.plot(x0, y0, 'or', label='Anfangsbedingung')

ax.plot(x, y, ' -b', label='numerische Lösung')

ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid()
plt.show()
```



Das Abdriften der numerischen Lösung ist deutlich erkennbar.

Das Abdriften kann extreme Züge annehmen bis hin zur völligen Unbrauchbarkeit der Ergebnisse.

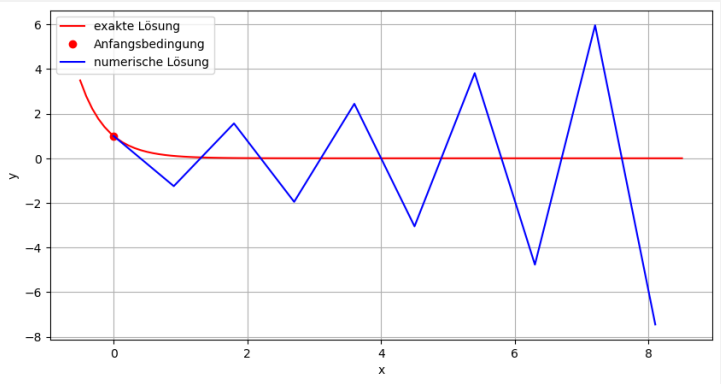
Beispiel

Wollen das Anfangswertproblem

$$y' = -\frac{5}{2}\,y, \qquad y(0) = 1 \tag{6.64}$$

numerisch lösen. Die exakte Lösung (zu Vergleichszwecken) ist

$$y(x) = \mathrm{e}^{-\frac{5}{2}x}. \tag{6.65}$$



Die numerische Lösung oszilliert mit zunehmender Schrittanzahl immer stärker und hat mit der tatsächlichen Lösung nichts mehr zu tun.

6.5.1.3 Systeme Für Systeme

$$\begin{aligned} y_1' &= f_1(x, y_1, \dots, y_n), \\ &\vdots \\ y_n' &= f_n(x, y_1, \dots, y_n). \end{aligned} \tag{6.66}$$

von Differentialgleichungen mit Anfangswerten

$$y_1(x_0) = y_{1,0}, \quad \dots, \quad y_n(x_0) = y_{n,0} \tag{6.67}$$

kann das explizite Euler-Verfahren auf jede Gleichung angewendet werden. Man erhält die Iterationsvorschrift

$$\begin{bmatrix} y_{1,k+1} \\ \vdots \\ y_{n,k+1} \end{bmatrix} = \begin{bmatrix} y_{1,k} \\ \vdots \\ y_{n,k} \end{bmatrix} + h \begin{bmatrix} f_1(x_k, y_{1,k}, \dots, y_{n,k}) \\ \vdots \\ f_n(x_k, y_{1,k}, \dots, y_{n,k}) \end{bmatrix} \tag{6.68}$$

für $k = 1, 2, \dots$.

Gleichungen höherer Ordnung können stets als Systeme erster Ordnung geschrieben werden, sodass das explizite Euler-Verfahren auch für Gleichungen höherer Ordnung einsetzbar ist.

6.5.2 Implizites Euler-Verfahren

6.5.2.1 Formulierung Nutzt man statt Vorwärtsdifferenzen Rückwärtsdifferenzen, so erhält man

$$\frac{y(x_{k+1}) - y(x_k)}{h} = f(x_{k+1}, y(x_{k+1})), \quad k = 0, 1, 2, \dots \tag{6.69}$$

aus der Differentialgleichung; also den Zusammenhang

$$y_{k+1} = y_k + h\, f(x_{k+1}, y_{k+1}). \tag{6.70}$$

Zur Berechnung von y_{k+1} aus y_k ist somit ein (nichtlineares) Gleichungssystem zu lösen, da die gesuchte Größe y_{k+1} nur implizit durch eine Gleichung beschrieben wird.

Das implizite Euler-Verfahren wählt den Wert y_{k+1} gerade so, dass man bei einem Schritt entgegen des Richtungsfeldes der Differentialgleichung gerade den vorhergehenden Punkt (x_k, y_k) erreicht (IDV 635).

6.5.2.2 Eigenschaften Beim Fehlerverhalten des impliziten Euler-Verfahrens gelten grundsätzlich die gleichen Aussagen wie beim expliziten Euler-Verfahren. Allerdings sind die Situationen, in den des exponentielle Fehlerwachstum tatsächlich auftritt deutlich seltener. Diese Situationen kann man recht konkret angeben, was wir hier aber nicht tun.

Beispiel

Lösen das Anfangswertproblem

$$y' = \sin x - \frac{y}{t}, \qquad y(1) = 1 \tag{6.71}$$

aus dem obigen Beispiel numerisch mit dem impliziten Euler-Verfahren.

```
import scipy

f = lambda x, y: np.sin(x) - y / x
x0 = 1
y0 = 1

x_max = x0 + 8  # Intervall, auf dem die Lösung berechnet werden
↪  soll
h = 0.5  # Schrittweite

# Listen für berechnete Werte
x = [x0]  # zunächst nur Startwert
y = [y0]

# Berechnung
while x[ -1] < x_max:
    x_new = x[ -1] + h
    y_new = scipy.optimize.newton(
        lambda yk1: yk1 - y[ -1] - h * f(x_new, yk1),
        y[ -1],  # Startwert
        lambda yk1: 1 - h * ( -1 / x_new)  # Ableitung
```

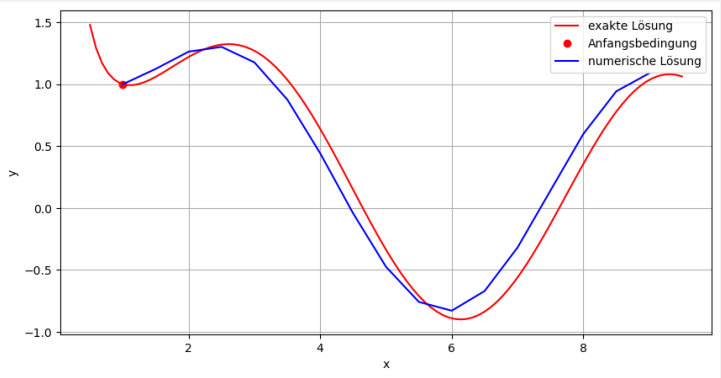
```
    )
    x.append(x_new)
    y.append(y_new)

# Plot
fig, ax = plt.subplots(figsize=(10, 5))

sol_x = np.linspace(x0 - 0.5, x_max + 0.5, 100)
sol_y = (np.sin(sol_x) - sol_x * np.cos(sol_x) + 1 - np.sin(1) +
↪  np.cos(1)) / sol_x
ax.plot(sol_x, sol_y, ' -r', label='exakte Lösung')
ax.plot(x0, y0, 'or', label='Anfangsbedingung')

ax.plot(x, y, ' -b', label='numerische Lösung')

ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid()
plt.show()
```

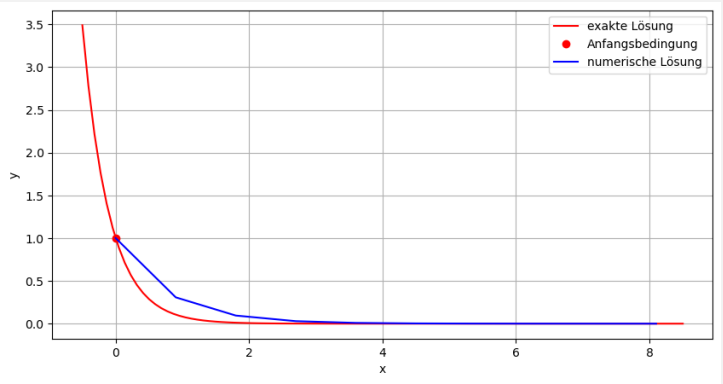


Das Abdriften der numerischen Lösung ist deutlich erkennbar.

Beispiel

Lösen das Anfangswertproblem

$$y' = -\frac{5}{2}\,y, \qquad y(0) = 1. \tag{6.72}$$



Im Gegensatz zum expliziten Euler-Verfahren treten hier keine immer stärker werdenden Oszillationen auf.

Der Preis für das bessere Fehlerverhalten ist die Notwendigkeit, in jedem Schritt eine nichtlineare Gleichung zu Lösen. Da allerdings mit dem vorhergehenden Funktionswert y_k ein guter Startwert zur Verfügung steht, sind meist nur sehr wenige Newton-Iterationen nötig.

6.5.2.3 Systeme Analog zum expliziten Euler-Verfahren kann auch das implizite Euler-Verfahren zum Lösen von Differentialgleichungssystemen (und damit von Gleichungen höherer Ordnung) eingesetzt werden. In jedem Schritt ist dann ein nichtlineares Gleichungssystem zu lösen.

6.6 Runge-Kutta-Verfahren

Man kann systematisch bessere Verfahren zum numerischen Lösen von Differentialgleichungen entwickeln. Diese Systematik stammt von Runge (1895) und Kutta (1901) und wurde von Butcher (1963) noch etwas vereinfacht. Geben hier nur einen groben Überblick und schauen uns an, wie die in diesem Kontext auftretenden *Butcher-Tableaus* zu verstehen sind.

6.6.1 Ziel

Zunächst ist zu klären, wie man einem Verfahren ansieht, dass es "besser" ist als die Euler-Verfahren.

Grundsätzlich ist man für gegebene Schrittanzahl $n \in \mathbb{N}$ am Gesamtfehler

$$\max_{k=1,\dots,n} |y(x_k) - y_k| \tag{6.73}$$

interessiert in Abhängigkeit von der Schrittweite h. Für die beiden Euler-Verfahren gilt

$$\max_{k=1,\dots,n} |y(x_k) - y_k| \leq c\,h \tag{6.74}$$

für kleine $h > 0$ und eine Konstante $c > 0$. Besser ist

$$\max_{k=1,\dots,n} |y(x_k) - y_k| \leq c\,h^p \tag{6.75}$$

mit $p > 1$, da zum Beispiel für $p = 2$ das Halbieren der Schrittweite den Fehler auf ein Viertel reduziert. Man nennt den Exponent p auch **Konvergenzordnung** des Verfahrens.

Je höher die Konvergenzordnung, desto größer kann h gewählt werden und desto geringer also der Rechenaufwandt.

6.6.2 Idee

Zur Erhöhung der Konvergenzordnung führt man Zwischenschritte ein, d.h. das Verfahren wertet f an mehr Stellen aus und erhält so mehr Informationen über die Differentialgleichung. Dabei kann die Wahl der Stellen für die zusätzlichen Auswertungen auch von den Funktionswerten von f abhängen.

Um geeignete Zwischenschritt zu finden, kann man aus der Forderung nach hoher Konvergenzordnung gewisse Gleichungen aufstellen. Das so entstehende nichtlineare Gleichungssystem muss dann symbolisch gelöst werden und liefert die unter behandelten Butcher-Tableaus. Man kann zeigen, dass beispielsweise für Konvergenzordnung $p = 10$ mindestens 10 Zwischenschritte nötig sind, deren Berechnungsvorschriften aus der Lösung eines nichtlinearen Gleichungssystems mit 200 Gleichungen gewonnen werden.

6.6.3 Heun-Verfahren

Als einfaches Verfahren mit einem Zwischenschritt betrachten wir kurz das Heun-Verfahren. Dieses führt zunächst einen Schritt des expliziten Euler-Verfahrens aus, um eine Näherung $\tilde{y}_{k+1}$ der Ableitung $y'(x_{k+1})$ zu erhalten. Anschließend führt es den eigentlichen Schritt aus, wobei der verwendete Anstieg gerade der Mittelwert aus y_k und $\tilde{y}_{k+1}$ ist:

$$\begin{aligned} \tilde{y}_{k+1} &:= y_k + h\, f(x_k, y_k), \\ y_{k+1} &:= y_k + h\, \frac{f(x_k, y_k) + f(x_{k+1}, \tilde{y}_{k+1})}{2}. \end{aligned} \tag{6.76}$$

Das Heun-Verfahren hat Konvergenzordnung $p = 2$ und liefert damit bei gleicher Schrittweite genauere Ergebnisse.

6.6.4 Allgemeine Formulierung

Die ein Runge-Kutta-Verfahren mit s Zwischenschritten beschreibenden Zahlen werden üblicherweise als sogenanntes **Butcher-Tableau** geschrieben.

$$\begin{array}{c|ccccc} c_1 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_2 \end{array} \tag{6.77}$$

Die Iterationsvorschrift für das zugehörige Verfahren ist dann

$$\begin{aligned} z_1 &:= f\big(x_k + c_1\, h, y_k\big), \\ z_2 &:= f\big(x_k + c_2\, h, y_k + h\, a_{21}\, z_1\big), \\ z_3 &:= f\big(x_k + c_3\, h, y_k + h\, (a_{31}\, z_1 + a_{32}\, z_2)\big), \\ &\vdots \\ z_s &:= f\big(x_k + c_s\, h, y_k + h\, (a_{s,1}\, z_1 + \cdots + a_{s,s-1}\, z_{s-1})\big), \\ y_{k+1} &:= y_k + h \sum_{i=1}^{s} b_i\, z_i, \end{aligned} \tag{6.78}$$

wobei die Werte $z_1, \ldots, z_s$ nach jedem Schritt verworfen werden können (und somit keinen Index k tragen).

Ein Runge-Kutta-Verfahren mit s Zwischenschritten wird auch als **s-stufiges** Runge-Kutta-Verfahren bezeichnet.

Beispiel

Das explizite Euler-Verfahren kann als einstufiges Runge-Kutta-Verfahren mit dem Butcher-Tableau

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \tag{6.79}$$

interpretiert werden.

Das Heun-Verfahren ist ein zweistufiges Runge-Kutta-Verfahren:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \tag{6.80}$$

6.6.5 Implizite Runge-Kutta-Verfahren

Bei der hier formulierten Variante handelt es sich um explizite Runge-Kutta-Verfahren, da alle Schritt direkt berechnet werden können. Es gibt jedoch auch implizite Runge-Kutta-Verfahren, bei denen $z_1, \ldots, z_s$ als Lösung eines nichtlinearen Gleichungssystems entstehen. In diesem Fall hat das Butcher-Tableau keine Dreieckstruktur, sondern ist voll besetzt, also rechteckig.

Implizite Runge-Kutta-Verfahren haben bei gleichem s üblicherweise eine höhere Konvergenzordnung als explizite Verfahren. Während bei expliziten Verfahren höchstens Kovergenzordnung s möglich ist, kann man mit impliziten Verfahren die Konvergenzordnung $2\,s$ erreichen.

Beispiel

Das implizite Euler-Verfahren kann als einstufiges implizites Runge-Kutta-Verfahren mit dem Butcher-Tableau

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{6.81}$$

interpretiert werden.

6.6.6 Klassisches Runge-Kutta-Verfahren

Als klassisches Runge-Kutta-Verfahren oder als RK4-Verfahren wird das explizite Runge-Kutta-Verfahren mit dem Butcher-Tableau

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{6.82}$$

bezeichnet.

6.6.7 Systeme

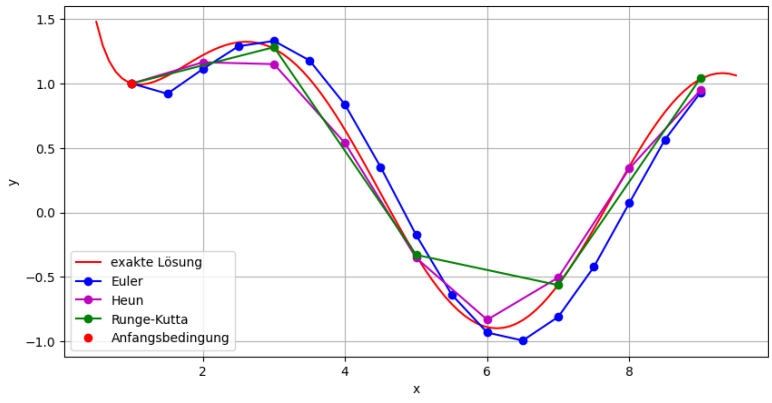
Analog zu den Euler-Verfahren kann das Runge-Kutta-Verfahren ohne Änderungen an den Formeln auf Systeme von Differentialgleichungen angewendet werden. Die Zwischenwerte z_i sind dann Vektoren mit n Komponenten, wobei n die Anzahl der Gleichungen im System ist.

6.6.8 Numerischer Vergleich

Die unterschiedliche Konvergenzordnung von Euler-, Heun- und klassischem Runge-Kutta-Verfahren kann man leicht numerisch nachvollziehen. Lösen dazu das Anfangswertproblem

$$y' = \sin x - \frac{y}{t}, \qquad y(1) = 1. \tag{6.83}$$

Wählen die Schrittweiten so, dass der Rechenaufwandt für die verschiedenen Verfahren in etwa gleich ist. Ist h die Schrittweite für das Euler-Verfahren, so führen wir das Heun-Verfahren mit Schrittweite $2\,h$ und das klassische Runge-Kutta-Verfahren mit Schrittweite $4\,h$ aus.



6.7 Software

Lösungsverfahren für gewöhnliche Differentialgleichungen muss man nur in wenigen Spezialfällen selbst implementieren, da entsprechende Funktionen üblicherweise in Software-Bibliotheken für numerische Berechnungen vorliegen. Oft erfolgt auch die Schrittweitenwahl automatisch (teils sogar adaptiv). In diesem Fall muss lediglich die gewünschte obere Schranke an den Fehler

$$\max_{k=1,\dots,n} |y_k - y(x_k)| \tag{6.84}$$

zwischen numerischer Lösung und exakter Lösung vorgegeben werden. Die Algorithmen wählen dann die Schrittweite so klein, dass diese Schranke mit hoher Wahrscheinlichkeit erfüllt ist (keine Garantie!).

SciPy bietet z.B. `scipy.integrate.solve_ivp` für das Lösen von Systemen erster Ordnung. Diese Funktion verwendet standardmäßig das klassische Runge-Kutta-Verfahren, kann aber auch andere Verfahren nutzen.

Beispiel (Lorenz-System)

Das nach Edward N. Lorenz benannte System

$$\begin{aligned} y_1' &= a\,(y_2 - y_1) \\ y_2' &= y_1\,(b - y_3) - y_2 \\ y_3' &= y_1\,y_2 - c\,y_3 \end{aligned} \tag{6.85}$$

von drei Differentialgleichungen sollte ursprünglich gewisse Prozesse in der Atmosphäre modellieren, spielt aber auch in vielen anderen Bereichen eine Rolle: Anwendungen:

- Strömungen in dünnen Gas- oder Flüssigkeitsschichten (a, b sind Materialparameter, c hängt von den Maßen der Schicht ab, y liefert gewisse Eigenschaften der Strömung, siehe Lorenz system),
- (einfache) Modellierung von Lasern ([Hak75]),
- Stromstärkeverlauf in geschlitzten Faraday-Generatoren ([Kno81]),
- Modellierung bestimmter Betriebszustände von bürstenlosen Gleichstrommotoren ([Hem94]).

```
import numpy as np
import plotly.graph_objects as go
import scipy

# Differentialgleichungssystem
def f(x, y):
    alpha, beta, gamma = 10, 28, 8/3
```

```
    return np.array([
        alpha * (y[1] - y[0]),
        y[0] * (beta - y[2]) - y[1],
        y[0] * y[1] - gamma * y[2]
    ])

# System lösen (automatische Schrittweitenwahl)
result = scipy.integrate.solve_ivp(
    f,  # Differentialgleichung
    (0, 40),  # Intervall, auf dem die Lösung berechnet werden
    ↪  soll
    np.array([10, 2, 3]),  # Startwert
    atol=1e -10,  # maximaler absoluter Lösungsfehler
    max_step=2e -2,  # größte zulässige Schrittweite
)
x = result.t
y = result.y

# Lösung darstellen
fig = go.Figure()
fig.layout.margin = dict(l=0, r=0, b=0, t=0)  # breiten Rand
↪  vermeiden
fig.add_trace(go.Scatter3d(
    x=y[0, :], y=y[1, :], z=y[2, :],
    mode='lines',
    line={
        'width': 2,
        'color': x,    # Liniensegmente nach Position in der
        ↪  Lösung färben
        'autocolorscale': False,    # wir wollen eigene Farbskala
        ↪  festlegen
        'colorscale': [[0, 'rgb(255,0,0)'],    # rot für x = 0
                       [0.5, 'rgb(0,255,0)'],  # grün für x = 0.5
                       [1, 'rgb(0,0,255)']]    # blau ab x = 1
    },
    hoverinfo = 'none'    # keine Infos bei Überfahren mit der
    ↪  Maus
))
fig.update_layout(
    scene_xaxis_title_text='y<sub>1</sub>',
    scene_yaxis_title_text='y<sub>2</sub>',
    scene_zaxis_title_text='y<sub>3</sub>',
    scene_camera_eye=dict(x=1,y= -1,z=1)
```

```
)
fig.show()
```

7 Partielle Differentialgleichungen

Theoretische Grundlagen und numerische Verfahren zum Lösen von partiellen Differentialgleichungen.

7.1 Überblick

7.1.1 Begriffe und Einordnung

Gleichungen, die eine Funktion *mehrerer* Veränderlicher mit ihren partiellen Ableitungen in Verbindung setzen, heißen **partielle Differentialgleichungen** (kurz: PDE für "partial differential equation"). Lösung einer PDE ist also eine Funktion.

Zusammenhänge und Gesetzmäßigkeiten großer Teile der Physik und anderer Naturwissenschaften, aber auch der Wirtschaftswissenschaften können durch partielle PDEs ausgedrückt werden. Sie sind somit eines der wichtigsten Modellierungswerkzeuge.

Die numerische (und analytische) Behandlung von PDEs ist deutlich anspruchsvoller als bei gewöhnlichen Differentialgleichungen. Insbesondere gibt es keine allgemein anwendbaren Lösungsverfahren, sondern jede Gleichung oder Klasse von Gleichungen erfordert andere Ansätze.

Nebenbemerkung

Für PDEs existiert im Gegensatz zu gewöhnlichen Differentialgleichungen (noch) keine abgeschlossene Lösungstheorie. Deshalb werden PDEs meist fallbezogen behandelt. Existierende umfassendere Theorien zu PDEs sind mathematisch extrem anspruchsvoll und können deshalb hier nicht behandelt werden.

Wir beschränken uns hier auf wichtige, oft auftretende PDE-Typen. Insbesondere werden wir nur **lineare PDEs** betrachten, also solche, in denen die gesuchte Funktion und deren partielle Ableitungen nur in Linearkombinationen miteinander verknüpft werden.

Merke

Die Ordnung der höchsten in einer PDE auftretenden partielle Ableitung heißt **Ordnung der PDE**.

Das Lösen von PDEs erfolgt heute ausschließlich numerisch. Analytisches Lösen ist nur in wenigen Spezialfällen möglich. Schon die Frage, ob überhaupt eine Lösung existiert und diese eindeutig ist, ist oft nur mit erheblichem mathematischen Aufwand zu beantworten.

7.1.2 Mahnendes Beispiel (Sleipner A)

Als Negativbeispiel sei der Untergang der norwegischen Ölbohrplattform "Sleipner A" in der Nordsee am 23. August 1991 noch während des Baus erwähnt. Die tragende Betonkonstruktion wurde ausschließlich durch numerische Berechnungen am Computer geplant. Zum Einsatz kam die damals übliche und auch heute noch genutzte Software NASTRAN der NASA zum Lösen partieller Differentialgleichungen. Durch **ungünstige Diskretisierung** der zugrunde liegenden partiellen Differentialgleichung, die die in den

Betonteilen wirkenden Kräfte beschreiben sollte, wurden einige Bereiche mit besonders hohen Lasten zu schwach dimensioniert. Die berechneten Lasten waren nur etwa halb so hoch wie die tatsächlichen. Die Plattform versank im 200 Meter tiefen Meer und löste durch Implosionen ein schwaches Erdbeben aus.

Die numerischen Berechnungen wurden nicht auf Plausibilität geprüft und nicht mit klassischen Berechnungsverfahren für Betontragwerke verglichen. Die Berechnungssoftware trifft hier übrigens keine Schuld. Es handelt sich um einen Fehler der Bediener, die ungeeignete Parameter gewählt haben. Für mehr Informationen zu den Ursachen des Untergangs siehe The sinking of the Sleipner A offshore platform und Die Ursache für den Totalverlust der Betonplattform Sleipner A und Verweise darin.

7.1.3 PDEs erster Ordnung

7.1.3.1 Struktur

Merke

Lineare PDE erster Ordnung haben die Form

$$\sum_{i=1}^{n} a_i(\underline{x}) \frac{\partial}{\partial x_i} y(\underline{x}) + b(\underline{x})\, y(\underline{x}) + c(\underline{x}) = 0, \quad \underline{x} \in B. \tag{7.1}$$

Dabei sind

- $\underline{x} = (x_1, \ldots, x_n)$ die Variablen, von denen die gesuchte Funktion abhängt,
- $y : B \to \mathbb{R}$ die gesuchte Funktion,
- $B \subseteq \mathbb{R}^n$ der Definitionsbereich der gesuchten Funktion (offene Menge),
- $a_1, \ldots, a_n, b, c$ gegebene Funktionen (Koeffizienten genannt).

Zusätzlich können noch Anfangs- oder Randbedingungen gegeben sein. Anfangsbedingungen geben die gesuchte Funktion zu einem Zeitpunkt vor, sofern eine der Variablen der Zeit entspricht. Randbedingungen geben die gesuchte Funktion auf dem Rand ∂B von B vor. Kombinationen (Anfangsrandbedingung) und weitere Arten von Randbedingungen (z.B. Vorgabe der Richtungsableitungen senkrecht zur Randlinie) sind in vielfältiger Weise möglich und üblich.

7.1.3.2 Beispiel: Kontinuitätsgleichung Ein typisches Beispiel ist die bereits behandelte Section 4.7.2.

7.1.3.3 Lösungsansätze PDEs erster Ordnung können auf (nichtlineare) Systeme gewöhnlicher Differentialgleichungen zurückgeführt werden.

Alternativ sind die später zu behandelnden **Finite-Differenzen-Verfahren** für das numerische Lösen einsetzbar.

7.1.4 PDEs zweiter Ordnung

7.1.4.1 Struktur

Merke

Lineare PDE zweiter Ordnung haben die Form

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(\underline{x})\,\frac{\partial^2}{\partial x_i \partial x_j} y(\underline{x}) \\ + \sum_{i=1}^{n} b_i(\underline{x})\,\frac{\partial}{\partial x_i} y(\underline{x}) + c(\underline{x})\,y(\underline{x}) + d(\underline{x}) = 0, \quad \underline{x} \in B.\end{aligned} \tag{7.2}$$

Dabei sind

- $\underline{x} = (x_1, \ldots, x_n)$ die Variablen, von denen die gesuchte Funktion abhängt,
- $y : B \to \mathbb{R}$ die gesuchte Funktion,
- $B \subseteq \mathbb{R}^n$ der Definitionsbereich der gesuchten Funktion (offene Menge),
- a_{ij}, b_i, c, d gegebene Funktionen (Koeffizienten genannt).

Analog zu PDE erster Ordnung werden meist zusätzliche Anfangs- und Randbedingungen formuliert.

Existenz und Eindeutigkeit von Lösungen sind nicht allgemein bekannt, sondern müssen im Einzelfall untersucht werden.

Die Lösungseigenschaften und geeignete Lösungsverfahren hängen wesentlich von der Koeffizientenmatrix

$$A(\underline{x}) = \begin{bmatrix} a_{11}(\underline{x}) & \cdots & a_{1n}(\underline{x}) \\ \vdots & & \vdots \\ a_{n1}(\underline{x}) & \cdots & a_{nn}(\underline{x}) \end{bmatrix} \tag{7.3}$$

ab. Diese ist üblicherweise symmetrisch für alle $\underline{x}$. In diesem Fall unterscheidet man vier Situationen:

Merke

- Ist $A(\underline{x})$ in jedem Punkt positiv definit oder in jedem Punkt negativ definit, so heißt die PDE **elliptisch**.

- Ist $A(\underline{x})$ in jedem Punkt positiv semidefinit (aber nicht positiv definit) oder in jedem Punkt negativ semidefinit (aber nicht negativ definit), so heißt die PDE **parabolisch**.
- Ist $A(\underline{x})$ in jedem Punkt indefinit, so heißt die PDE **hyperbolisch**.
- $A(\underline{x})$ hat unterschiedliche Definitheitseigenschaften in B, ist also beispielsweise in einem Teilgebiet elliptisch und in einem anderen hyperbolisch.

7.1.4.2 Beispiel: Potentialgleichung

Merke

PDE der Form

$$\Delta y(x_1, \ldots, x_n) = f(x_1, \ldots, x_n), \quad \underline{x} \in B \tag{7.4}$$

mit dem **Laplace-Operator**

$$\Delta := \frac{\partial^2}{\partial^2 x_1} + \cdots + \frac{\partial^2}{\partial^2 x_n} \tag{7.5}$$

und gegebener Funktion $f : \mathbb{R}^n \to \mathbb{R}$ heißen **Poisson-Gleichung**. Ist $f \equiv 0$, so spricht man von einer **Laplace-Gleichung** oder auch **Potentialgleichung**.

Poisson-Gleichungen sind elliptische PDE. Sie tauchen in zahlreichen physikalischen Zusammenhängen auf (Elektrostatik, Wärmeleitung, Strömungsmechanik,...). Die gegebene Funktion f modelliert dabei das Vorhandensein von Quellen und Senken, in der Elektrostatik also z.B. die Ladungsverteilung im Gebiet. ∇y ist dann der daraus resultierende Stromfluss und $\Delta y = \operatorname{div} \nabla y$ liefert die Quellen und Senken in diesem Fluss. Die Potentialgleichung $\Delta y = 0$ modelliert ein Erhaltungsgesetz, welches für viele physikalische Prozesse gilt: “Was rein fließt, muss auch wieder raus fließen”.

Merke

Man unterscheidet drei Typen von Randbedingungen:

- **Dirichlet-Randbedingung** (y ist auf dem gesamten Rand gegeben),
- **Neumann-Randbedingung** (die Richtungsableitungen von y in Normalenrichtung sind auf dem gesamten Rand gegeben),
- **gemischte Randbedingungen** (teils Dirichlet, teils Neumann, z.B. Elektrische Impedanztomografie).

Bei Dirichlet-Randbedingungen ist die Lösung eindeutig, falls sie existiert. Bei Neumann-Randbedingungen unterscheiden sich alle Lösungen nur um eine additive Konstante.

Merke

Für die Potentialgleichung gilt das **Maximumprinzip**: Erfüllt eine Funktion y die Potentialgleichung in einem Gebiet B, so nimmt die Funktion ihr Maximum (und auch ihr Minimum) auf dem Rand ∂B des Gebiets an.

Diese Eigenschaft kann man oft nutzen, um wichtige Aussagen über die Lösungen zu erhalten ohne die Lösungen explizit finden zu müssen.

7.1.4.3 Beispiel: Diffusionsgleichung Wollen die Wärmeverteilung im Laufe der Zeit in einem Körper $B \subseteq \mathbb{R}^n$ bei gegebenen Anfangs- und Randbedingungen modellieren. Völlig analog zur Massenbilanz beim Section 4.7.2 führt hier eine Energiebilanz zur PDE

$$\Delta_{\underline{x}} u(\underline{x}, t) - \frac{\varrho\, c}{\lambda} \frac{\partial}{\partial t} u(\underline{x}, t) = 0. \tag{7.6}$$

Dabei sind

- $u(\underline{x}, t)$ die Temperatur des Körpers am Punkt $\underline{x}$ zur Teit t,
- ϱ die Dichte des Materials (überall gleich),
- c die spezifische Wärmekapazität des Materials (überall gleich),
- λ die spezifische Wärmeleitfähigkeit des Materials (überall gleich).

Merke

PDE mit der Struktur

$$\Delta_{\underline{x}} u - a \frac{\partial}{\partial t} u = 0 \tag{7.7}$$

mit einer Konstante $a > 0$ heißen **Diffusionsgleichungen** und sind stets parabolisch.

Um die Temperatur zu einer Zeit t berechnen zu können, muss einerseits die Temperaturverteilung zu einem früheren Zeitpunkt bekannt sein (Anfangsbedingung); andererseits muss der Wärmezufluss oder -abfluss über den Rand des Gebietes im Laufe der Zeit bekannt sein.

Merke

Üblich Anfangsbedingung:

$$u(\underline{x}, 0) = f(\underline{x}) \quad \text{für alle } \underline{x} \in B \cup \partial B \tag{7.8}$$

mit einer gegebenen Funktion f, die nur vom Ort abhängt.

Für die Formulierung von Randbedingungen gibt es verschiedene Varianten, die

auch auf verschiedenen Teilen des Randes unterschiedlich eingesetzt werden können:

- Ist die Umgebungs- bzw. Oberflächentemperatur konstant über die Zeit, so kommt die sogenannte **Dirichlet-Randbedingung**

$$u(\underline{x}, t) = g(\underline{x}) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{7.9}$$

  zum Einsatz. Zu beachten ist, dass diese kompatibel mit der Anfangsbedingung sein muss, also $f(\underline{x}) = g(\underline{x})$ für $\underline{x} \in \partial B$.

- Ist nur der Wärmefluss über den Rand bekannt, z.B. bei (perfekter) Isolation oder Vorhandensein einer Wärmequelle, so kommt die **Neumann-Randbedingung**

$$\frac{\partial}{\partial n} u(\underline{x}, t) = h(\underline{x}) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{7.10}$$

  zum Einsatz, wobei $\frac{\partial}{\partial n} u$ die Normalenableitung von u in einem Randpunkt bezeichnet. Perfekte Isolation entspricht $h \equiv 0$.

- Hängt der Wärmefluss über den Rand von der Temperaturdifferenz zwischen Objekt und Umgebung ab (also insbesondere von der Objekttemperatur) wird die **Robin-Randbedingung**

$$\frac{\partial}{\partial n} u(\underline{x}, t) = \alpha(\underline{x}) \big(u(\underline{x}, t) - U\big) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{7.11}$$

  verwendet. Dabei sind $U \in \mathbb{R}$ die Umgebungstemperatur und α die Wärmeübergangsfunktion, die sich aus den konkreten physikalischen Vorgängen ergibt, die zum Wärmeabfluss führen (z.B. vorbeiströmendes Kühlmittel).

Betrachten ein einfaches 1D-Beispiel um eine Idee vom Verhalten der Lösungen der Wärmeleitungsgleichung zu bekommen.

Beispiel (isolierter Stab)

Sei $n = 1$ und $B = (0, 1)$ modelliere einen dünnen Stab. Alle Materialparameter seien 1, sodass die Wärmeleitungsgleichung

$$u_{xx}(x, t) - u_t(x, t) = 0 \quad \text{für alle } x \in (0, 1) \quad \text{und alle } t > 0 \tag{7.12}$$

lautet. Die anfängliche Temperaturverteilung sei

$$u(x, 0) = \cos(2\,\pi\, x) \quad \text{für alle } x \in [0, 1] \tag{7.13}$$

und der Stab sei an den beiden enden perfekt isolierte, also

$$u_x(0,t) = u_x(1,t) = 0 \quad \text{für alle } t \geq 0. \tag{7.14}$$

Für diesen speziellen Fall kann man die Lösungen analytisch bestimmen (tun wir hier nicht) und erhält

$$u(x,t) = \cos(2\,\pi\,x)\,\mathrm{e}^{-4\,\pi^2\,t} \tag{7.15}$$

(IDV 710).

7.1.4.4 Beispiel: Wellengleichung Hatten die Wellengleichung bereits am Beispiel der schwingenden Saite eingeführt.

Merke

Üblicherweise werden zwei Anfangsbedingungen benötigt:

- Anfangsauslenkung $u(x,0)$,
- Anfangsgeschwindigkeit $\frac{\partial}{\partial t}u(x,0)$.

Als Randbedingungen kommen analog zu den Diffusionsgleichungen Dirichlet-Randbedingungen (Rand fest) und Neumann-Randbedingungen in Frage. Bei Neumann-Randbedingungen ist die physikalische Interpretation schwierig. Im Wesentlichen wird dadurch die Neigung des Randes (z.B. einer Membran) vorgegeben.

Die Struktur der Lösungen hängt stark von der Raumdimension ab.

Beispiel (eingespannte Saite)

Eine an beiden Enden fest eingespannte Saite $B = (0,1)$ erfüllt die Randbedingungen

$$u(0,t) = u(1,t) = 0 \quad \text{für alle } t \geq 0. \tag{7.16}$$

Für eine gegebene Funktion $f : [0,1] \to \mathbb{R}$ kann man zu den Anfangsbedingungen

$$u(x,0) = f(x) \quad \text{für alle } x \in [0,1] \tag{7.17}$$

und

$$u_t(x,0) = 0 \quad \text{für alle } x \in [0,1] \tag{7.18}$$

(Saite in vorgegebener Lage, aber noch nicht in Bewegung) die Lösung der Wellengleichung

$$c^2\,u_{xx}(x,t) - u_{tt}(x,t) = 0 \quad \text{für alle } x \in [0,1] \quad \text{und alle } t \geq 0 \tag{7.19}$$

analytisch bestimmen (tun wir hier nicht). Man erhält

$$u(x,t) = \sum_{k=1}^{\infty} a_k \, \sin(k\,\pi\,x) \, \cos(c\,k\,\pi\,t) \tag{7.20}$$

wobei

$$a_k := 2 \int_0^1 f(x) \, \sin(k\,\pi\,x)\,\mathrm{d}x \tag{7.21}$$

gerade die Sinus-Anteile der Fourier-Koeffizienten der Anfangsauslenkung f sind.

Auch abseits der Wellengleichung ist die Fourier-Transformation ein wichtiges Hilfsmittel beim analytischen Lösen von PDE.

7.1.4.5 Lösungsansätze Je nach Gleichungstyp (elliptisch, parabolisch, hyperbolisch) und Randbedingungen kommen verschiedene analytische Lösungsansätze in Frage. Diese sind allerdings fast immer extrem aufwendig und erfordern tiefere mathematische Kenntnisse (Umgang mit Funktionenreihen,...). Teilweise lassen sich PDEs auf gewöhnliche Differentialgleichungen oder Systeme gewöhnlicher Differentialgleichungen zurückführen, welche dann ebenfalls mit recht hohem Aufwand zu lösen sind.

Für den praktischen Einsatz erfolgt das Lösen von PDE zweiter Ordnung ausschließlich numerisch. Die wichtigsten Verfahrensklassen sind:

- **Finite-Differenzen-Verfahren** (einfache Herleitung, begrenzte Anwendbarkeit bei komplizierten Gebieten und anderen praktisch relevanten Situationen),
- **Finite-Elemente-Verfahren** (erfordern mehr Aufwand in der Herleitung bzw. für das Verständnis, sind aber sehr allgemein einsetzbar).

7.2 Finite-Differenzen-Verfahren

Das Finite-Differenzen-Verfahren (kurz FDM für "finite difference method") ist zwar in mancher Hinsicht den später zu behandelnden Finite-Elemente-Verfahren unterlegen, ist aufgrund seiner Einfachheit aber ein guter Einstieg in das numerische Lösen von PDEs und weit verbreitet im praktischen Einsatz.

7.2.1 Idee

Die Grundidee ist sehr einfach:

Merke

Ersetze alle in der zu lösenden PDE auftretenden Ableitungen durch Differenzenquotienten bzgl. eines (hinreichend feinen) Gitters.

Statt einer Funktion sind also nur noch Funktionswerte auf endlich vielen Gitterpunkten gesucht. Diese Werte stehen durch die Differenzenquotienten zueinander in Beziehung. Für lineare PDE führt dieses Vorgehen praktisch immer zu einem linearen Gleichungssystem, welches mit Standardverfahren gelöst werden kann.

Hauptschwierigkeit beim Finite-Differenzen-Verfahren ist das Einbinden von Randbedingungen für Gebiete mit krumlinigem Rand, da die Gitterpunkte eines regelmäßigen Gitters nur selten auf dem Rand liegen werden.

Nebenbemerkung

Prinzipiell sind die hier diskutierten Ideen und Verfahren auch auf gewöhnliche Differentialgleichung (ODE) anwendbar. Diese werden hier allerdings nicht explizit behandelt, da sie einerseits methodisch mehr Möglichkeiten bieten als PDE und andererseits weniger Detailprobleme bei der Umsetzung der Verfahren verursachen. Setzt man die Idee des Finite-Differenzen-Verfahrens für ODE um, so erhält man einige der einfacheren ODE-Verfahren.

7.2.2 Fehlerabschätzungen

Theoretische Resultate zur Genauigkeit der erhaltenen Lösungen existieren vor allem für elliptische lineare PDE. Das Finite-Differenzen-Verfahren ist zwar prinzipiell auch für andere PDE einsetzbar, jedoch besteht ohne hinreichende theoretische Untermauerung keine Garantie, dass die erhaltenen diskreten Lösungen in hinreichend engem Zusammenhang mit den tatsächlichen Lösungen der PDE stehen.

Merke

Konkret sollten zwei Eigenschaften für jedes konkrete Lösungsverfahren gegeben sein:

- gute **Kondition** (kleine Fehler in den Eingabedaten (Anfangs- und Randbedingungungen) führen auch nur zu kleinen Fehlern in den Lösungen),
- **Konvergenz** (je feiner die Diskretisierung, desto kleiner die Abweichung zwischen diskreter und kontinuierlicher Lösung).

Hatten im Kapitel zu den Grundlagen der numerischen Mathematik noch der Begriff der Section 3.3.6 eingeführt, welcher besagt, dass der Algorithmus hinreichend genau das kontinuierliche Problem löst. Aus guter Kondition des kontinuierlichen Problems und Stabilität des Algorithmus erhält man dann die gute Kondition des Algorithmus. Im Kontext von PDEs gilt

$$\text{Algorithmus} = \text{Diskretisieren} + \text{Standardverfahren}, \tag{7.22}$$

sodass man davon ausgehen kann, dass gute Kondition des diskretisierten Problems auch gute Kondition des Algorithmus liefert. Ensprechend beschränkt man sich auf die Untersuchung der Kondition des diskretisierten Problems.

Die Kondition des diskretisierten Problems ist durch die Konditionszahl der Systemmatrix gegeben. Diese muss meist aufwendig bestimmt oder wenigstens nach oben abgeschätzt werden.

Untersuchungen zur Konvergenz sind ebenfalls mit hohem Aufwand verbunden und deshalb nicht Bestandteil dieser Veranstaltung. Für alle üblichen PDEs findet man entsprechende Herleitungen in der Literatur. Wir beschränken uns hier auf die Angaben der Resultate um einen Eindruck von der zu erwartenden Lösungsqualität zu bekommen.

7.2.3 Allgemeines Vorgehen

Unabhängig von der zu lösenden PDE sind folgende Schritt auszuführen:

1. Gitterpunkte für jede Variable wählen.
2. Ableitungen an Gitterpunkten, die nicht auf dem Rand des betrachteten Gebietes liegen, durch Differenzenquotienten entsprechend den Nachbargitterpunkten ersetzen.
3. Anfangs- und Randbedingungen durch Funktionswerte bzw. deren Ableitungen (Differenzenquotienten) auf Gitterpunkten des Randes ausdrücken.
4. Entstehendes Gleichungssystem lösen.

Die konkrete Umsetzung dieser Schritte hängt im Detail von den Gegebenheiten der vorliegenden PDE, insbesondere von den Anfangs- und Randbedingungen ab.

7.2.4 Beispiel: 2D-Poisson-Gleichung

Wollen

$$\Delta u(x,y) = f(x,y), \quad (x,y) \in B \tag{7.23}$$

für das Rechteck

$$B = (0,a) \times (0,b) \tag{7.24}$$

lösen. Die Lösung gibt beispielsweise die Auslenkung einer der Last f ausgesetzten Membran an, solange die Auslenkung "klein" ist. Am Rand sei die Membran fest eingespannt:

$$u(x,y) = 0 \quad \text{für } (x,y) \in \partial B. \tag{7.25}$$

7.2.4.1 Gitterpunkte Wählen in x- und y-Richtung $n_x + 1$ bzw. $n_y + 1$ gleichmäßig verteilte Stützstellen mit Abständen

$$\Delta x := \frac{a}{n_x}, \quad \Delta y := \frac{b}{n_y}, \tag{7.26}$$

sodass die Gitterpunkte

$$(x_i, y_j), \quad x_i = i\,\Delta x, \quad y_j = j\,\Delta y, \quad i = 0,\ldots,n_x, \quad j = 0,\ldots,n_y \tag{7.27}$$

entstehen.

Gilt $i \in \{0, n_x\}$ oder $j \in \{0, n_y\}$, so ist (x_i, x_j) ein Randpunkt des Gebietes B.

Führen folgende Kurzbezeichnungen ein:

$$u_{i,j} := u(x_i, y_j), \qquad f_{i,j} := f(x_i, y_j). \tag{7.28}$$

7.2.4.2 Diskretisierung im Inneren Die zu lösende Gleichung

$$u_{xx}(x,y) + u_{yy}(x,y) = f(x,y) \tag{7.29}$$

betrachten wir nur auf den inneren Gitterpunkten und ersetzen dort die Ableitungen durch die Differenzenquotienten

$$D_x^2(x_i, y_j) := \frac{u_{i+1,j} - 2\,u_{i,j} + u_{i-1,j}}{(\Delta x)^2} \tag{7.30}$$

bzw.

$$D_y^2(x_i, y_i) := \frac{u_{i,j+1} - 2\,u_{i,j} + u_{i,j-1}}{(\Delta y)^2}. \tag{7.31}$$

Erhalten somit

$$D_x^2(x_i, y_j) + D_y^2(x_i, y_j) = f_{i,j}, \quad i = 1,\ldots,n_x - 1, \quad j = 1,\ldots,n_y - 1. \tag{7.32}$$

7.2.4.3 Diskretisierung der Randbedingung Die Randbedingung $u(x, y) = 0$ liefert auf den Randgitterpunkten

$$u_{0,j} = u_{n_x,j} = u_{i,0} = u_{i,n_y} = 0 \tag{7.33}$$

für $i = 0, \ldots, n_x$ und $j = 0, \ldots, n_y$.

7.2.4.4 Gleichungssystem Die an den inneren Punkten diskretisierte PDE liefert $(n_x - 1)\,(n_y - 1)$ Gleichungen für $(n_x + 1)\,(n_y + 1)$ Unbekannte. Aus der diskretisierten Randbedingung können allerdings die Werte gerade für soviele Unbekannte direkt entnommen werden, dass für die restlichen Unbekannten ein Gleichungssystem mit quadratischer Systemmatrix entsteht. Man kann zeigen (tun wir hier nicht), dass dieses eindeutig lösbar ist.

7.2.4.5 Kondition Man kann zeigen (tun wir hier nicht), dass das diskretisierte Problem gut konditioniert ist:

Merke

Sei u die (nur auf den Gitterpunkten gegebene) exakte Lösung des diskretisierten Problems und sei $\tilde{u}$ die (nur auf den Gitterpunkten) berechnete Lösung. Weiter seien

- $r > 0$ der Radius eines Kreises um den Koordinatenursprung, der das Gebiet B vollständig enthält,
- $G \subseteq \mathbb{R}^2$ die Menge aller Gitterpunkte,
- $A \in \mathbb{R}^{((n_x+1)\,(n_y+1))\times((n_x+1)\,(n_y+1))}$ die Systemmatrix des zu lösenden Gleichungssystems einschließlich der Gleichungen für die Randbedingung.

Dann gilt:

$$\max_G |u - \tilde{u}| \leq \max_{\partial B \cap G} |u - \tilde{u}| + \frac{r^2}{2} \max_{B \cap G} |f - A\,\tilde{u}|. \tag{7.34}$$

Der erste Summand gibt den Fehler in der Randbedingung an. Dieser wird meist Null sein oder nur in der Größenordnung des Gleitkommarundungsfehlers liegen. Der zweite Summand gibt an, wie gut die (in der Praxis immer näherungsweise) Lösung des Gleichungssystems die (diskretisierte) rechte Seite f der PDE annähert. Sind beide Fehler klein, wird also auch der Abstand zwischen berechneter und exakter Lösung klein sein.

7.2.4.6 Konvergenz Man kann zeigen (tun wir hier nicht):

Merke

Sei die exakte Lösung u dreimal stetig differenzierbar, sei $\tilde{u}$ die (nur auf den Gitterpunkten gegebene) Lösung des diskretisierten Problems und bezeichne $G \subseteq \mathbb{R}^2$ die Menge der Gitterpunkte. Dann gilt

$$\max_G |u - \tilde{u}| \leq c \max\{\Delta x, \Delta y\} \tag{7.35}$$

mit einer von Δx und Δy unabhängigen Konstante $c > 0$.

Ist u sogar viermal stetig differenzierbar, so erhält man diese Abschätzung mit $\max\{\Delta x, \Delta y\}^2$.

Je feiner das Gitter, desto kleiner wird also der Fehler in der Lösung sein.

Beachte: In der Konvergenzaussage hier ist u die exakte Lösung der PDE, in der obigen Konditionsaussage ist u hingegen die exakte Lösung des diskretisierten Problems. In beiden Fällen ist $\tilde{u}$ die berechnete Lösung des diskretisierten Problems.

7.2.5 Nicht trivial berandete Gebiete

Liegen auf dem Rand ∂B des Gebiets B kaum oder gar keine Gitterpunkte, müssen am Rand zusätzliche Gitterpunkte eingefügt werden. Dies geschieht üblicherweise durch achsenparallele Verschiebung des jeweils ersten außerhalb des Gebietes liegenden Gitterpunktes. Dadurch sind die Gitterpunkte nicht mehr äquidistant, was zu höherem Aufwand beim Aufstellen des Gleichungssystems führt. Mit diesem Vorgehen hat jeder innere Punkt stets vier Nachbarpunkte, sodass die Differenzenquotienten problemlos aufgestellt werden können (IDV 720).

7.3 Fallstudie: 2D-Kontinuitätsgleichung

Betrachten verschiedene (auch ungeeignete) Verfahren zum Lösen der Kontinuitätsgleichung in zwei Raumdimensionen.

7.3.1 Problemstellung

7.3.1.1 Gleichung Die Kontinuitätsgleichung

$$u_t(x,y,t) + (v\,u)_x(x,y,t) + (w\,u)_y(x,y,t) = 0, \quad (x,y) \in B, \quad ,t \geq 0 \tag{7.36}$$

beschreibt z.B. die Masseerhaltung beim Materialtransport in einer Strömung. Dabei ist $u(x,y,t)$ die Dichte des transportierten Materials (z.B. eine Ölschicht auf einer Wasseroberfläche) im Punkt (x,y) zur Zeit t. Die Werte $v(x,y)$ und $w(x,y)$ beschreiben gemeinsam als Vektor Strömungsrichtung und Strömungsgeschwindigkeit (z.B. einer Wasseroberfläche). Zur durch die Funktionen v und w gegebenen Strömung ist die Funktion u gesucht.

Zu beachten ist, dass die Kontinuitätsgleichung keine Diffussionsprozesse abbildet, sondern lediglich den Transport durch Strömung. Ist z.B. die Wasseroberfläche strömungsfrei, so wird die anfängliche Ölverteilung in der Lösung der Gleichung über die Zeit erhalten bleiben. Praktisch ist allerdings davon auszugehen, dass der Ölteppich "breitfließt", also von sich aus die Dichte verringert und die beanspruchte Fläche vergrößert. Für die kombinierte Betrachtung von Transport- und Diffusionsprozessen kann man auch eine PDE herleiten und lösen. Der Einfachheit halber verzichten wir hier jedoch auf den Diffusionsanteil. Praktisch bedeutet dies, dass unser Ölteppich bereits eine Dichte erreicht hat, bei der die Ölschicht ohne externe Kräfte (Strömung) sich nicht mehr verändert.

Ebenfalls zur Vereinfachung beschränken wir uns auf eine im Zeitverlauf unveränderliche Strömung. Die Funktionen v und w hängen also nur vom Ort, aber nicht von der Zeit ab. Ein Teil der unten betrachteten Lösungsverfahren ist auch für zeitlich veränderliche Strömungen einsetzbar.

7.3.1.2 Anfangsbedingung Die Anfangskonzentration des durch die Strömung transportierten Stoffs kann durch eine vom Ort abhängige Funktion f ausgedrückt werden:

$$u(x,y,t) = f(x,y) \quad \text{für alle } (x,y) \in B \text{ und } t \geq 0. \tag{7.37}$$

7.3.1.3 Randbedingungen Das Formulieren von sinnvollen Randbedingungen ist schwierig. Folgende Bedingungen sollen am Rand erfüllt sein:

- Kein Material strömt vom Rand des betrachteten Gebietes $B \subseteq \mathbb{R}^2$ ein.
- Material kann frei über den Rand mit der Strömung abtransportiert werden.

Je nach Strömungsrichtung in Bezug auf den Rand, ist die Randbedingung also "kein Fluss über den Rand" oder "Fluss über den Rand in unbekannter Höhe". Diese Randbedingungen lassen sich nur mühsam in Formeln ausdrücken. Wir werden Sie deshalb je nach Lösungsverfahren individuell in die Algorithmen integrieren.

Wir interpretieren hier B als Ausschnitt aus einem räumlich größeren Strömungsgeschehen. Durch einen Unfall auf einer Ölbohrplattform könnte Öl freigesetzt worden sein und wir wollen die Ausbreitung des Ölteppichs in einem gewissen Bereich unter gegebenen Strömungsverhältnissen vorhersagen. Der Bereich soll dabei so groß sein, dass der anfängliche Ölteppich komplett darin enthalten ist. Im Laufe der Zeit wird das Öl den Rand erreichen und unseren Vorhersagebereich verlassen.

Wählen

$$B := [0, a] \times [0, b]. \tag{7.38}$$

7.3.1.4 Inkompressible Strömung Zur Vereinfachung beschränken wir uns auf inkompressible Strömungen, also solche, bei denen Zufluss und Abfluss in jedem Punkt identisch sind. In der Sprache der Differentialoperatoren soll die Divergenz des Strömungsfeldes überall Null sein:

$$v_x(x, y, t) + w_y(x, y, t) = 0 \quad (x, y) \in B, \quad t \geq 0. \tag{7.39}$$

Auf eine Herleitung dieses Zusammenhangs verzichten wir hier (siehe z.B. Wikipedia zu "Incompressible flow" für eine Variante).

Bezogen auf das Beispiel eines Ölteppichs auf einer Wasserobfläche bedeutet Inkompressibilität insbesondere, dass keine Strömung in vertikale Richtung stattfindet. Es kann also kein Wasser (und damit Öl) nach unten "verschwinden" oder von unten "auftauchen".

Durch die Annahme einer inkompressiblen Strömung kann die Kontinuitätsgleichung zu

$$u_t(x, y, t) + v(x, y, t)\, u_x(x, y, t) + w(x, y, t)\, u_y(x, y, t) = 0 \tag{7.40}$$

vereinfacht werden (IDV 725).

7.3.2 Diskretisierung

Setzen

- $\Delta x := \frac{a}{n_x}$,
- $\Delta y := \frac{b}{n_y}$,
- $\Delta t := \frac{T}{n_t}$

und wählen in den Variablen x, y, z gleichmäßig verteilte Stützstellen

- $x_i := i\,\Delta x, \quad i = 0, 1, \ldots, n_x,$
- $y_j := i\,\Delta y, \quad j = 0, 1, \ldots, n_y,$
- $t_k := i\,\Delta t, \quad k = 0, 1, \ldots, n_t.$

Bezeichnen die Funktionswerte an den Stützstellen entsprechend mit

- $u_{i,j,k} := u(x_i, y_j, t_k),$
- $v_{i,j} := v(x_i, y_j),$
- $w_{i,j} := w(x_i, y_j),$
- $f_{i,j} := f(x_i, y_j).$

7.3.3 Einfaches Upwind-Verfahren

7.3.3.1 Idee Ersetzt man u_t durch eine Vorwärtsdifferenz und u_x und u_y durch Rückwärtsdifferenzen, so erhält man

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j}\,\frac{\Delta t}{\Delta x}\,(u_{i,j,k} - u_{i-1,j,k}) - w_{i,j}\,\frac{\Delta t}{\Delta y}\,(u_{i,j,k} - u_{i,j-1,k}) \tag{7.41}$$

(IDV 730). Die Zahlen

$$C_x := \max_{i,j} |v_{i,j}|\,\frac{\Delta t}{\Delta x} \quad \text{und} \quad C_y := \max_{i,j} |w_{i,j}|\,\frac{\Delta t}{\Delta y} \tag{7.42}$$

heißen **Courant-Zahlen**.

Zur Interpretation betrachten wir eine Strömung parallel zur x-Achse, also $w \equiv 0$:

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j}\,\frac{\Delta t}{\Delta x}\,(u_{i,j,k} - u_{i-1,j,k}). \tag{7.43}$$

Wie sehen hier:

- Sind alle $v_{i,j}$ positiv, so ändert sich die Materialmenge an der Stelle (x_i, y_j) pro Zeitschritt indem der Anteil $v_{i,j}\,\frac{\Delta t}{\Delta x}\,u_{i,j,k}$ der vorhandenen Menge $u_{i,j,k}$ abfließt und der Anteil $v_{i,j}\,\frac{\Delta t}{\Delta x}\,u_{i-1,j,k}$ aus Strömungsrichtung hinzukommt.
- Es sollte $C_x \leq 1$ gelten, da sonst mehr Material pro Zeitschritt abfließen würde als vorhanden ist. Der Abstand Δt der Zeitstützstellen muss also klein genug sein. Wir können $\frac{\Delta x}{\Delta t}$ als höchste mit unserer Diskretisierung simulierbare Strömungsgeschwindigkeit interpretieren, denn die Bedingung $C_x \leq 1$ besagt

$$|v_{i,j}| \leq \frac{\Delta x}{\Delta t} \quad \text{für alle } i, j. \tag{7.44}$$

- Sind die $v_{i,j}$ negativ, so wird der Materialzuwachs pro Zeitschritt aus der stromabwärts (!) gelegenen Materialmenge $u_{i-1,j,k}$ berechnet. Dies ist zwar theoretisch eine valide Approximation der Ableitung in x-Richtung, praktisch aber wenig sinnvoll. Deshalb sollte bei negativem $v_{i,j}$ eine Vorwärtsdifferenz verwendet werden.

7.3.3.2 Umsetzung Setzen wir

$$u_{i,j,k}^{\text{in},x} := \begin{cases} u_{i-1,j,k}, & \text{falls } v_{i,j} \geq 0, \\ u_{i+1,j,k}, & \text{falls } v_{i,j} < 0 \end{cases} \tag{7.45}$$

und

$$u_{i,j,k}^{\text{in},y} := \begin{cases} u_{i,j-1,k}, & \text{falls } w_{i,j} \geq 0, \\ u_{i,j+1,k}, & \text{falls } w_{i,j} < 0, \end{cases} \tag{7.46}$$

so erhalten wir die diskretisierte Gleichung

$$u_{i,j,k+1} = u_{i,j,k} - |v_{i,j}| \frac{\Delta t}{\Delta x} \left(u_{i,j,k} - u_{i,j,k}^{\text{in},x}\right) - |w_{i,j}| \frac{\Delta t}{\Delta y} \left(u_{i,j,k} - u_{i,j,k}^{\text{in},y}\right) \tag{7.47}$$

(IDV 735). Dies ist das sogenannte Upwind-Verfahren (Upwind = gegen den Wind bzw. die Strömung)

Die Randbedingungen können hier leicht integriert werden. Liegt (x_i, y_j) an einem Rand mit Zustrom in das Gebiet, so setzt man $u_{i,j,k}^{\text{in},x} := 0$ und $u_{i,j,k}^{\text{in},y} := 0$. An Rändern mit Abfluss aus dem Gebiet ist nichts zu tun.

7.3.3.3 Kondition und Konvergenz Man kann zeigen, dass die Berechnungen im Upwind-Verfahren gut konditioniert sind, wenn

$$C_x + C_y \leq 1 \tag{7.48}$$

gilt.

Allerdings liegt **keine Konvergenz** vor, da die Materialverteilung im Laufe der Zeit "breitläuft". Hintergrund ist, dass man die gewählte Diskretisierung der Kontinuitätsgleichung auch als Diskretisierung (mittels zentraler Differenzen) einer kombinierten Kontinuitäts-Diffusions-Gleichung interpretieren kann. Zwei verschiedene PDE führen also zum selben diskretisierten Problem. Konvergenz kann aber höchstens bezüglich einer dieser beiden PDE vorliegen.

Nebenbemerkung

Bei Aussagen zur Konvergenz muss man sehr genau prüfen, in welchem Sinne Konvergenz jeweils verstand wird. Betrachtet man beispielsweise nur ein festes Zeitintervall $[0, T]$, so kann man die Abweichung zwischen diskreter und kontinuierlicher Lösung durchaus beliebig klein bekommen indem man die Diskretrisierung in Raum und Zeit fein genug wählt. In diesem Sinne liegt also doch Konvergenz vor. Möchte man die Zeit hingegen nicht begrenzen, so werden die Lösungen immer breitlaufen, egal wie fein die Diskretisierung ist.

7.3.3.4 Implementierung Schwierigster Punkt bei der Implementierung ist die Fallunterscheidung je nach Strömungsrichtung. Eine effiziente Lösung in Python (also ohne explizite Schleifen) ist mittels Bool'scher Indizierung von NumPy-Arrays möglich.

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

def cont2d_upwind(a, b, T, v, w, f, nx, ny, nt):

    delta_x = a / nx
    delta_y = b / ny
    delta_t = T / nt

    x = np.linspace(0, a, nx + 1)
    y = np.linspace(0, b, ny + 1)
    grid_x, grid_y = np.meshgrid(x, y, indexing='ij')  # x ist erste
    ↪ Dimension

    v = v(grid_x, grid_y)
    w = w(grid_x, grid_y)
    Cx = np.abs(v).max() * delta_t / delta_x
    Cy = np.abs(w).max() * delta_t / delta_y
    print(f'Summe der Courant -Zahlen: {Cx + Cy}')

    u = np.empty((nt + 1, nx + 1, ny + 1), dtype=float)

    # Anfangsbedingungen
    u[0, :, :] = f(grid_x, grid_y)

    # Zeitschritte
    v_pos_mask = v >= 0
    v_pos_mask[0, :] = False  # kein Zufluss von linkem Rand
    v_neg_mask = np.logical_not(v_pos_mask)
    v_neg_mask[ -1, :] = False  # kein Zufluss von rechtem Rand
    w_pos_mask = w >= 0
    w_pos_mask[:, 0] = False  # kein Zufluss vom oberen Rand
    w_neg_mask = np.logical_not(w_pos_mask)
    w_neg_mask[:, -1] = False  # kein Zufluss vom unteren Rand

    u_inx = np.zeros((nx + 1, ny + 1), dtype=float)
    u_iny = np.zeros((nx + 1, ny + 1), dtype=float)
    for k in range(0, nt):

        u_inx[0, :] = 0
        u_inx[ -1, :] = 0
```

```
        u_inx[v_pos_mask] = u[k, : -1, :][v_pos_mask[1:, :]]
        u_inx[v_neg_mask] = u[k, 1:, :][v_neg_mask[: -1, :]]
        u_iny[:, 0] = 0
        u_iny[:, -1] = 0
        u_iny[w_pos_mask] = u[k, :, : -1][w_pos_mask[:, 1:]]
        u_iny[w_neg_mask] = u[k, :, 1:][w_neg_mask[:, : -1]]

        u[k + 1, :, :] = u[k, :, :] \
                         - np.abs(v) * delta_t / delta_x * (u[k, :, :]
                         ↪  - u_inx) \
                         - np.abs(w) * delta_t / delta_y * (u[k, :, :]
                         ↪  - u_iny)

    return u
```

Neben der Lösung u visualisieren wir auch das Strömungsfeld.

```
import matplotlib.animation as anim
from IPython.display import HTML

def show_anim(u, a, b, T, v, w, quiver_step=1):

    nt = u.shape[0] - 1
    nx = u.shape[1] - 1
    ny = u.shape[2] - 1
    x = np.linspace(0, a, nx + 1)
    y = np.linspace(0, b, ny + 1)
    grid_x, grid_y = np.meshgrid(x, y, indexing='ij')  # x ist erste
    ↪  Dimension

    v = v(grid_x, grid_y)
    w = w(grid_x, grid_y)

    # Bildrate auf 25 fps begrenzen
    fps_real = nt / T
    fps_show = 25
    skip_frames = int(np.floor(fps_real / fps_show))
    n_frames = (nt + 1) // (skip_frames + 1)

    # Figure/Axes erstellen
    fig, ax = plt.subplots()
    ax.set_xlim(0, a)
    ax.set_ylim(0, b)
    ax.set_xlabel('x')
```

```
    ax.set_ylabel('y')

    # Update -Funktion
    def update(frame):
        ax.clear()
        ax.contourf(
            grid_x, grid_y, u[(skip_frames + 1) * frame, :, :],
            levels=100, cmap='jet', vmin=0, vmax=1
        )
        ax.quiver(
            grid_x[::quiver_step, ::quiver_step],
            grid_y[::quiver_step, ::quiver_step],
            v[::quiver_step, ::quiver_step],
            w[::quiver_step, ::quiver_step],
            angles='xy', pivot='mid', color='#ffffff'
        )
        ax.axis('equal')
        ax.set_xlabel('x')
        ax.set_ylabel('y')

    # Animation erstellen und anzeigen
    fa = anim.FuncAnimation(fig, update, frames=n_frames,
    ↪  interval=1000/fps_show)
    display(HTML(fa.to_jshtml()))
    plt.close()  # verhindert automatischen Aufruf von plt.show() durch
    ↪  Jupyter
```

Als Anfangsverteilung f verwenden wir einen kreisförmigen Bereich mit zum Rand linear abfallender Konzentration.

Eine einfache Strömung mit konstanter Richtung und konstanter Geschwindigkeit:

```
a = 3
b = 2
T = 2

nx = 40
ny = int(np.ceil(nx / a * b))  # \Delta y = \Delta x
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1),
    indexing='ij'
)

v = lambda x, y: np.ones_like(x)
```

```
w = lambda x, y: np.ones_like(x)

f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.3) ** 2 + (y/b
↪  - 0.3) ** 2))

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
nt = int(np.ceil(T * (v_max * nx / a + w_max * ny / b)))  # Courant <=
↪  1

u = cont2d_upwind(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

```
Summe der Courant -Zahlen: 0.9938271604938271
```

0

[ ] Once [x] Loop [ ] Reflect

Wir sehen, dass der Ölteppich breitläuft wie wir oben schon festgestellt hatten, obwohl das kontinuierliche Problem gar keine Diffusion modelliert. Zusätzlich verändert sich auch die Form: quer zur Strömung läuft der Ölteppich stärker breit als in Strömungsrichtung. Mehr dazu beim nächsten Verfahren.

Betrachten noch eine kreisförmige Strömung. Nach Ablauf des Zeitintervalls sollte bei exakter Rechnung wieder die Anfangssituation vorliegen, was offensichtlich nicht der Falls ist.

```
T = 2 * np.pi

def r_phi(x, y):
    x = x - 0.5 * a
    y = y - 0.5 * b
    r = np.sqrt(x ** 2 + y ** 2)
    phi = np.arctan2(y, x)
    return r, phi
def v(x, y):
    r, phi = r_phi(x, y)
    return -r * np.sin(phi)
def w(x, y):
    r, phi = r_phi(x, y)
    return r * np.cos(phi)

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
```

```
nt = int(np.ceil(T * (v_max * nx / a + w_max * ny / b)))  # Courant <=
↪  1

u = cont2d_upwind(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

```
Summe der Courant -Zahlen: 0.9953316347458231
```

0

[ ] Once [x] Loop [ ] Reflect

Nebenbemerkung

Alternativ zum Test mit einer kreisförmigen Strömung kann man die Simulation auch mit umgekehrter Strömungsrichtung wiederholen. Nutzt man das Ergebnis der ersten Simulation als Anfangswert für die zweite Simulation, so sollte die zweite Simulation bei exakter Rechnung am Ende wieder den Anfangswert der ersten Simulation liefern.

7.3.4 Verbessertes Upwind-Verfahren

7.3.4.1 Idee Hatten beim Upwind-Verfahren beobachtet, dass der Ölteppich quer zur Strömung stärker breitläuft als in Strömungsrichtung. Dies liegt an der gewählten Diskretisierung: Die Strömung wird separat in x- und in y-Richtung simuliert. Ist in einem Punkt (x_i, y_j) die Strömungsrichtung $(1, 1)$ (also 45°), so müsste eigentlich Material vom Punkt (x_{i-1}, y_{j-1}) (links unten) einströmen. In der Simulation kommt aber sämtlicher Zufluss nur von (x_{i-1}, y_j) (links) und (x_i, y_{j-1}) (unten). Entsprechend wirkt das Breitlaufen stärker nach links und unten (also nicht in Strömungsrichtung) als direkt nach links unten (also in Strömungsrichtung).

Um diesen Effekt mit einfachen Mitteln zu verringern, kann man die Berechnungen in x- und y-Richtung zeitlich nacheinander durchführen. Es wird also erst der Zufluss in x-Richtung in jedem Punkt berechnet und anschließend im nächsten Zeitschritt der Zufluss in y-Richtung. Dadurch fließt bei Betrachtung eines Punktes (x_i, y_j) zunächst Material von (x_{i-1}, y_{j-1}) (links unten) nach (x_i, y_{j-1}) (unten) und dann nach oben in den betrachteten Punkt. Der Punkt (x_i, y_j) bekommt nun also Zufluss von links, links unten und unten.

Dieses Vorgehen wird auch als **Corner-Transport-Upwind-Verfahren (CTU)** bezeichnet.

7.3.4.2 Umsetzung Um die Anzahl der Zeitschritte nicht zu verdoppeln, verwendet man bei der Darstellung dieses Vorgehens in Formeln gern halbe Zeitschritte:

$$u_{i,j,k}^{\text{in},x} := \begin{cases} u_{i-1,j,k}, & \text{falls } v_{i,j} \geq 0, \\ u_{i+1,j,k}, & \text{falls } v_{i,j} < 0, \end{cases} \tag{7.49}$$

$$u_{i,j,k+\frac{1}{2}} := u_{i,j,k} - |v_{i,j}| \frac{\Delta t}{\Delta x} (u_{i,j,k} - u_{i,j,k}^{\text{in},x}), \tag{7.50}$$

$$u_{i,j,k+\frac{1}{2}}^{\text{in},y} := \begin{cases} u_{i,j-1,k+\frac{1}{2}}, & \text{falls } w_{i,j} \geq 0, \\ u_{i,j+1,k+\frac{1}{2}}, & \text{falls } w_{i,j} < 0, \end{cases} \tag{7.51}$$

$$u_{i,j,k+1} := u_{i,j,k+\frac{1}{2}} - |w_{i,j}| \frac{\Delta t}{\Delta y} (u_{i,j,k+\frac{1}{2}} - u_{i,j,k+\frac{1}{2}}^{\text{in},y}). \tag{7.52}$$

7.3.4.3 Implementierung

```
def cont2d_corner(a, b, T, v, w, f, nx, ny, nt):

    delta_x = a / nx
    delta_y = b / ny
    delta_t = T / nt

    x = np.linspace(0, a, nx + 1)
    y = np.linspace(0, b, ny + 1)
    grid_x, grid_y = np.meshgrid(x, y, indexing='ij')  # x ist erste
    ↪ Dimension

    v = v(grid_x, grid_y)
    w = w(grid_x, grid_y)
    Cx = np.abs(v).max() * delta_t / delta_x
    Cy = np.abs(w).max() * delta_t / delta_y
    print(f'Summe der Courant -Zahlen: {Cx + Cy}')

    u = np.empty((nt + 1, nx + 1, ny + 1), dtype=float)

    # Anfangsbedingungen
    u[0, :, :] = f(grid_x, grid_y)

    # Zeitschritte
    v_pos_mask = v >= 0
    v_pos_mask[0, :] = False  # kein Zufluss von linkem Rand
    v_neg_mask = np.logical_not(v_pos_mask)
```

```
    v_neg_mask[ -1, :] = False  # kein Zufluss von rechtem Rand
    w_pos_mask = w >= 0
    w_pos_mask[:, 0] = False  # kein Zufluss vom oberen Rand
    w_neg_mask = np.logical_not(w_pos_mask)
    w_neg_mask[:, -1] = False  # kein Zufluss vom unteren Rand

    u_inx = np.zeros((nx + 1, ny + 1), dtype=float)
    u_iny = np.zeros((nx + 1, ny + 1), dtype=float)
    for k in range(0, nt):

        u_inx[0, :] = 0
        u_inx[ -1, :] = 0
        u_inx[v_pos_mask] = u[k, : -1, :][v_pos_mask[1:, :]]
        u_inx[v_neg_mask] = u[k, 1:, :][v_neg_mask[: -1, :]]

        u_tmp = u[k, :, :] - np.abs(v) * delta_t / delta_x * (u[k, :,
        ↪  :] - u_inx)

        u_iny[:, 0] = 0
        u_iny[:, -1] = 0
        u_iny[w_pos_mask] = u_tmp[:, : -1][w_pos_mask[:, 1:]]
        u_iny[w_neg_mask] = u_tmp[:, 1:][w_neg_mask[:, : -1]]

        u[k + 1, :, :] = u_tmp - np.abs(w) * delta_t / delta_y * (u_tmp
        ↪  - u_iny)

    return u
```

Nun bleibt der Ölteppich kreisförmig:

```
a = 3
b = 2
T = 2

nx = 40
ny = int(np.ceil(nx / a * b))  # \delta y = \delta x
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1),
    indexing='ij'
)

v = lambda x, y: np.ones_like(x)
w = lambda x, y: np.ones_like(x)
```

```
f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.3) ** 2 + (y/b
↪  - 0.3) ** 2))

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
nt = int(np.ceil(T * (v_max * nx / a + w_max * ny / b)))  # Courant <=
↪  1

u = cont2d_corner(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

```
Summe der Courant -Zahlen: 0.9938271604938271
```

0

[ ] Once [x] Loop [ ] Reflect

7.3.5 Zentrale Differenzen

7.3.5.1 Idee Um eventuell bessere Ergebnisse zu erzielen, könnte man zentrale Differenzenquotienten verwenden. In der Zeit ist dies ungünstig, da wir dann das entstehende Gleichungssystem nicht mehr einfach durch Umstellen lösen können. In den Raumkoordinaten spricht aber erstmal nichts dagegen. Insbesondere ist damit die Fallunterscheidung je nach Strömungsrichtung nicht mehr nötig. Diesen Ansatz nennt man auch **FTCS-Verfahren** (für "forward time centered space").

Es wird sicher allerdings zeigen, dass die Lösung des diskretisierten Problems schlecht konditioniert ist. Das Verfahren ist also praktisch nicht einsetzbar.

7.3.5.2 Umsetzung Einsetzen der Differenzenquotienten in die PDE und umstellen nach $u_{i,j,k+1}$ liefert

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j} \frac{\Delta t}{2\,\Delta x} (u_{i+1,j,k} - u_{i-1,j,k}) - w_{i,j} \frac{\Delta t}{2\,\Delta y} (u_{i,j+1,k} - u_{i,j-1,k}) \quad (7.53)$$

für $i = 1, \ldots, n_x - 1$ und $j = 1, \ldots, n_y - 1$ (IDV 740).

An den Rändern des Gebiets stehen keine zentralen Differenzen zur Verfügung. Ist die Strömung in das Gebiet hinein gerichtet, so ist der Funktionswert außerhalb des Randes Null. Bei Abfluss aus dem Gebiet, ist der Funktionswert allerdings unklar. Hier können wir nur auf einseitige Differenzen ausweichen.

7.3.5.3 Implementierung

```
def cont2d_ftcs(a, b, T, v, w, f, nx, ny, nt):

    delta_x = a / nx
    delta_y = b / ny
    delta_t = T / nt

    x = np.linspace(0, a, nx + 1)
    y = np.linspace(0, b, ny + 1)
    grid_x, grid_y = np.meshgrid(x, y, indexing='ij')  # x ist erste
    ↪ Dimension

    v = v(grid_x, grid_y)
    w = w(grid_x, grid_y)

    u = np.empty((nt + 1, nx + 1, ny + 1), dtype=float)

    # Anfangsbedingungen
    u[0, :, :] = f(grid_x, grid_y)

    # Zeitschritte
    v_pos_left_mask = v[0, :] >= 0
    v_neg_right_mask = v[ -1, :] < 0
    w_pos_bottom_mask = w[:, 0] >= 0
    w_neg_top_mask = w[:, -1,] < 0

    for k in range(0, nt):

        # innere Punkte
        u[k + 1, :, :] = u[k, :, :]
        u[k + 1, 1: -1, :] -= 0.5 * v[1: -1, :] * delta_t / delta_x *
        ↪ (u[k, 2:, :] - u[k, : -2, :])
        u[k + 1, :, 1: -1] -= 0.5 * w[:, 1: -1] * delta_t / delta_y *
        ↪ (u[k, :, 2:] - u[k, :, : -2])

        # linker Rand
        u_ext = 2 * u[k, 0, :] - u[k, 1, :]
        u_ext[v_pos_left_mask] = 0
        u[k + 1, 0, :] -= 0.5 * v[0, :] * delta_t / delta_x * (u[k, 1,
        ↪ :] - u_ext)

        # rechter Rand
        u_ext = 2 * u[k, -2, :] - u[k, -1, :]
        u_ext[v_neg_right_mask] = 0
```

```
        u[k + 1, -1, :] -= 0.5 * v[ -1, :] * delta_t / delta_x * (u_ext
        ↪  - u[k, -2, :])

        # unterer Rand
        u_ext = 2 * u[k, :, 0] - u[k, :, 1]
        u_ext[w_pos_bottom_mask] = 0
        u[k + 1, :, 0] -= 0.5 * w[:, 0] * delta_t / delta_y * (u[k, :,
        ↪  1] - u_ext)

        # oberer Rand
        u_ext = 2 * u[k, :, -2] - u[k, :, -1]
        u_ext[w_neg_top_mask] = 0
        u[k + 1, :, -1] -= 0.5 * w[:, -1] * delta_t / delta_y * (u_ext
        ↪  - u[k, :, -2])

    return u
```

```
a = 3
b = 2
T = 2

nx = 40
ny = int(np.ceil(nx / a * b))  # \delta y = \delta x
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1),
    indexing='ij'
)

v = lambda x, y: np.ones_like(x)
w = lambda x, y: np.ones_like(x)

f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.3) ** 2 + (y/b
↪  - 0.3) ** 2))

nt = 200  # Wert viel höher als bei vorherigen Simulationen!

u = cont2d_ftcs(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

0

[ ] Once [x] Loop [ ] Reflect

Wir sehen, dass trotz recht feiner Zeitdiskretisierung schon nach wenigen Schritten Oszillationen entstehen, die sich immer weiter verstärken.

7.3.6 Lax-Friedrichs-Verfahren

Mit etwas Aufwand kann man zeigen, dass das FTCS-Verfahren gut konditioniert wird, wenn man den Wert $u_{i,j,k}$ in der Formel zur Berechnung von $u_{i,j,k+1}$ durch den Mittelwert über die vier Nachbarpunkte ersetzt, also

$$u_{i,j,k+1} = \overline{u}_{i,j,k} - v_{i,j} \frac{\Delta t}{2\,\Delta x} \left(u_{i+1,j,k} - u_{i-1,j,k}\right) - w_{i,j} \frac{\Delta t}{2\,\Delta y} \left(u_{i,j+1,k} - u_{i,j-1,k}\right) \tag{7.54}$$

mit

$$\overline{u}_{i,j,k} := \frac{1}{4} \left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k}\right). \tag{7.55}$$

Gilt

$$C_x \leq \frac{1}{2} \quad \text{und} \quad C_y \leq \frac{1}{2} \tag{7.56}$$

mit C_x und C_y aus (7.42), so ist die Berechung gut konditioniert.

Das diskretisierte Problem kann gleichzeitig als Diskretisierung einer PDE mit Diffusionsterm interpretiert werden, sodass die anfängliche Materialdichte im Laufe der Zeit wieder breitlaufen wird. Die Ergebnisse werden also nicht deutlich besser sein als beim Upwind-Verfahren, sodass wir auf die Umsetzung verzichten.

7.3.7 Semi-Lagrange-Verfahren

Gut funktionierende Finite-Differenzen-Ansätze für die Kontinuitätsgleichung sind schwierig zu finden. Das Crank-Nicolson-Verfahren liefert gute Ergebnisse, erfordert aber deutlich mehr Rechenaufwand, da die Lösung für den nächsten Zeitschritt nur implizit gegeben ist, also als Lösung eines linearen Gleichungssystems bestimmt werden muss.

Ein anderer, nicht auf finiten Differenzen beruhender Ansatz, ist das sogenannte Semi-Langrange-Verfahren. Dieses beruht wie das nachfolgend betrachtete Partikelverfahren auf der Idee, die Bewegung einzelner Partikel in der Strömung zu verfolgen.

7.3.7.1 Idee Um die Stoffkonzentration $u_{i,j,k+1}$ an einem Punkt (x_i, y_j) zur Zeit t_{k+1} zu ermitteln, berechnen wir, wo ein Partikel zur Zeit t_k gewesen sein muss, damit es zur Zeit t_{k+1} durch die Strömung zum betrachteten Punkt getragen wird. Der gesuchte Wert $u_{i,j,k+1}$ wird dann etwa so groß sein wie u am Ausgangspunkt des Partikels zur Zeit t_k.

Führen wir diese "Rückwärtsrechnung" mit jedem Gitterpunkt durch, so erhalten wir wieder Zeitschritt für Zeitschritt die Stoffkonzentration im gesamten Gebiet.

7.3.7.2 Umsetzung Berechnen den Herkunftsort des in (x_i, y_j) ankommenden Partikels näherungsweise als

$$\begin{bmatrix} \tilde{x}_i \\ \tilde{y}_j \end{bmatrix} := \begin{bmatrix} x_i \\ y_j \end{bmatrix} - \Delta t \begin{bmatrix} v_{i,j} \\ w_{i,j} \end{bmatrix} \tag{7.57}$$

und setzen

$$u_{i,j,k+1} := u(\tilde{x}_i, \tilde{y}_j, t_k). \tag{7.58}$$

So einfach die Idee klingt, gibt es doch einige Probleme zu lösen:

- Die Bestimmung des Herkunftsortes eines Partikels ist nur einigermaßen genau, wenn das Strömungsfeld entlang der zurückgelegten Strecke nahezu konstant ist. Je nach Strömungsfeld, müssen die Zeitschritte also klein genug sein.
- Das Auswerten von $u(\cdot, \cdot, t_k)$ ist nur an den Gitterpunkten möglich. Die Punkte $(\tilde{x}_i, \tilde{y}_j)$ werden aber nicht mit den Gitterpunkten zusammenfallen. Wir müssen also interpolieren.
- Liegt ein Punkt $(\tilde{x}_i, \tilde{y}_j)$ außerhalb des betrachteten Gebiets, so ist klar, dass die Strömung bei (x_i, y_j) in das Gebiet hinein gerichtet ist. Wir können in diesem Fall also $u_{i,j,k+1} := 0$ setzen entsprechend den Randbedingungen.

Wird stückweise lineare Interpolation genutz, so kann man zeigen, dass das Verfahren im Wesentlichen äquivalent zum Upwind-Verfahren ist. Um bessere Ergebnisse (weniger Diffusion) zu erzielen, kommen also nur glattere Interpolationen in Frage.

7.3.7.3 Implementierung Die Interpolation überlassen wir `scipy.interpolate.RegularGridInte`
Das vereinfacht die Implementierung erheblich.

```
from scipy.interpolate import RegularGridInterpolator

def cont2d_semi_lagrange(a, b, T, v, w, f, nx, ny, nt):

    delta_x = a / nx
    delta_y = b / ny
    delta_t = T / nt

    x = np.linspace(0, a, nx + 1)
    y = np.linspace(0, b, ny + 1)
    grid_x, grid_y = np.meshgrid(x, y, indexing='ij')  # x ist erste
    ↪ Dimension

    v = v(grid_x, grid_y)
    w = w(grid_x, grid_y)

    u = np.empty((nt + 1, nx + 1, ny + 1), dtype=float)
```

```
    # Anfangsbedingungen
    u[0, :, :] = f(grid_x, grid_y)

    # Zeitschritte
    grid_x_tilde = grid_x - delta_t * v
    grid_y_tilde = grid_y - delta_t * w

    for k in range(0, nt):

        interpolator = RegularGridInterpolator(
            (x, y), u[k, :, :],
            method='cubic',
            bounds_error=False, fill_value=0  # nicht extrapolieren,
            ↪  sondern 0
        )
        u[k + 1, :, :] = interpolator((grid_x_tilde, grid_y_tilde))

    return u
```

```
a = 3
b = 2
T = 2

nx = 40
ny = int(np.ceil(nx / a * b))  # \delta y = \delta x
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1),
    indexing='ij'
)

v = lambda x, y: np.ones_like(x)
w = lambda x, y: np.ones_like(x)

f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.3) ** 2 + (y/b
↪  - 0.3) ** 2))

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
nt = int(np.ceil(2 * T * np.maximum(v_max * nx / a, w_max * ny / b)))
↪  # Courant <= 0.5

u = cont2d_semi_lagrange(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

0

[ ] Once [x] Loop [ ] Reflect

Die Diffusion ist nun deutlich geringer, aber nicht ganz verschwunden.

```
T = 2 * np.pi

def r_phi(x, y):
    x = x - 0.5 * a
    y = y - 0.5 * b
    r = np.sqrt(x ** 2 + y ** 2)
    phi = np.arctan2(y, x)
    return r, phi
def v(x, y):
    r, phi = r_phi(x, y)
    return -r * np.sin(phi)
def w(x, y):
    r, phi = r_phi(x, y)
    return r * np.cos(phi)

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
nt = int(np.ceil(T * (v_max * nx / a + w_max * ny / b)))  # Courant <=
↪  1

u = cont2d_semi_lagrange(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

0

[ ] Once [x] Loop [ ] Reflect

7.3.8 Partikelverfahren

Beim Semi-Lagrange-Verfahren wird in jedem Zeitschritt ein neuer "Satz" Partikel betrachtet und über einen Zeitschritt rückwärts verfolgt. Man kann die Idee auch umgekehrt verwenden: Man startet zur Zeit t_0 mit einer zufälligen oder regelmäßigen Partikelverteilung und verfolgt die Bewegungen dieser festen Partikelmenge Zeitschritt für Zeitschritt. Die Werte $u_{i,j,k+1}$ erhält man dann aus Interpolation der Werte $u(\tilde{x}_{i,0}, \tilde{y}_{j,0}, t_0)$, wobei $(\tilde{x}_{i,0}, \tilde{y}_{j,0})$ die anfänglichen Partikelpositionen sind.

Auch hier sind wieder einige Detailprobleme zu lösen:

- Die Interpolation ist aufwendiger als beim Semi-Lagrange-Verfahren, da die Stützstellen kein regelmäßiges Gitter bilden.
- Es können Teilgebiete entstehen, die keinerlei Partikel enthalten. Je nach Situation kann dies bedeuten, dass dort die Stoffkonzentration Null ist oder dass die Partikeldichte sehr gering ist. Ein klare Abgrenzung zwischen "keine Partikel" und "wenige Partikel" ist schwierig.
- In der Nähe eines Gitterpunktes können sich viele Partikel aufhalten, was allerdings keinen nennenswerten Einfluss auf die Interpolation hat. Dadurch gehen Partikel "verloren" und hohe Stoffkonzentrationen werden nicht erkannt.

Die letzten beiden Problempunkte kann man lösen indem man das Gebiet in rechteckige Zellen teilt; eine Zelle pro Gitterpunkt mit dem Gitterpunkt als Mittelpunkt. Die Stoffdichte in jeder Zelle ergibt sich dann als Summe aller von den Partikeln in der Zelle getragenen Stoffkonzentrationen.

7.3.9 Strömung um Hindernisse

Für komplexere Gebiete $B \subseteq \mathbb{R}^2$, die beispielsweise auch Hindernisse enthalten können, muss das Strömungsfeld separat simuliert werden. Für inkompressible Strömungen kann das Geschwindigkeitsfeld stets als Gradient eines sogenannten Potentials ausgedrückt werden:

$$\begin{bmatrix} v(x,y) \\ w(x,y) \end{bmatrix} = \nabla p(x,y) \tag{7.59}$$

mit einer Funktion $p : B \to \mathbb{R}$. Das Potential ist Lösung der Potentialgleichung

$$\Delta p(x,y) = 0 \quad \text{für alle } (x,y) \in B. \tag{7.60}$$

Mittels Neumann-Randbedingungen wird festgelegt über welche Teile des Randes Zu- bzw. Abfluss stattfinden (positive bzw. negative Ableitung in Normalenrichtung) und welche Teile undurchlässig sind (Ableitung in Normalenrichtung ist Null).

Beachtet die Anfangsverteilung die Form des Gebietes B (Dichte ist Null innerhalb von Hindernissen), so kann das Gebiet B für das Lösen der Kontinuitätsgleichung vereinfacht werden, indem Hindernisse nicht mit modelliert werden. Allerdings kann dieser Ansatz zu unerwartetem verhalten der Dichte am Rand von Hindernissen führen. Insbesondere beim Semi-Lagrange-Verfahren kann Masse verloren gehen. Partikelverfahren liefern hier bessere Ergebnisse.

Als Beispiel betrachten wir die Strömung um eine Kreisscheibe mit Radius R und Mittelpunkt $(x_\mathrm{M}, y_\mathrm{M})$. Für diesen Fall ist die Lösung der Potentialgleichung und damit das Strömungsfeld bekannt (siehe Potential flow around a circular cylinder für eine Herleitung):

$$\begin{bmatrix} v(x,y) \\ w(x,y) \end{bmatrix} = \left(1 - \frac{R^2}{r^2}\right) \cos\varphi \begin{bmatrix} \cos\varphi \\ \sin\varphi \end{bmatrix} - \left(1 + \frac{R^2}{r^2}\right) \sin\varphi \begin{bmatrix} -\sin\varphi \\ \cos\varphi \end{bmatrix} \tag{7.61}$$

für $\varphi \in [0, 2\,\pi)$ und $r \geq R$, wobei

$$x = x_\mathrm{M} + r\,\cos\varphi, \qquad y = y_\mathrm{M} + r\,\sin\varphi. \tag{7.62}$$

Innerhalb des Kreises ($r < R$) setzen wir das Strömungsfeld auf Null.

```
a = 3
b = 2
T = 3

nx = 40
ny = int(np.ceil(nx / a * b))  # \delta y = \delta x
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1),
    indexing='ij'
)

def flow_field(x, y, x0, y0):
    R = 0.3
    x = x - x0
    y = y - y0
    r = np.sqrt(x ** 2 + y ** 2)
    phi = np.arctan2(y, x)
    r_mask = r > R
    s = np.zeros_like(x)
    psi = np.zeros_like(x)
    s[r_mask] = (1 - R ** 2 / r[r_mask] ** 2) * np.cos(phi[r_mask])
    psi[r_mask] = -(1 + R ** 2 / r[r_mask] ** 2) * np.sin(phi[r_mask])
    return s * np.cos(phi) - psi * np.sin(phi), s * np.sin(phi) + psi *
    ↪  np.cos(phi)

def v(x, y):
    return flow_field(x, y, 0.5 * a, 0.5 * b)[0]
def w(x, y):
    return flow_field(x, y, 0.5 * a, 0.5 * b)[1]

f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.2) ** 2 + (y/b
↪  - 0.3) ** 2))

v_max = np.abs(v(grid_x, grid_y)).max()
w_max = np.abs(w(grid_x, grid_y)).max()
nt = int(np.ceil(2 * T * np.maximum(v_max * nx / a, w_max * ny / b)))
↪  # Courant <= 0.5
```

```
u = cont2d_semi_lagrange(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

0

[ ] Once [x] Loop [ ] Reflect

Direkt vor und hinter der Kreisscheibe ändert sich die Richtung der Strömung stark, sodass die für das Semi-Langrange-Verfahren notwendige Bedingung einer lokal konstanten Strömung nicht mehr gegeben ist. Entsprechend groß sind die entstehenden Fehler in der Simulation. Dem könnte man mit einer Verkleinerung von Δt entgegenwirken, was aber wiederum die Diffusionseffekte verstärkt.

Am folgenden Beispiel sind die Fehler noch deutlicher. Die Artefakte in der linken Bildhälfte entstehen durch die Interpolation, welche von SciPy nur näherungsweise vorgenommen wird (obwohl alle umgebenden Punkte den Funktionswert 0 haben, sind die interpolierten Werte im Bereich 10^{-7}).

```
f = lambda x, y: np.maximum(0, 1 - 10 * np.sqrt((x/a - 0.2) ** 2 + (y/b
↪  - 0.5) ** 2))

u = cont2d_semi_lagrange(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

0

[ ] Once [x] Loop [ ] Reflect

7.4 Fallstudie: 1D-Wellengleichung

Betrachten das Lösen der Wellengleichung in einer Raumdimension mittels Finite-Differenzen-Verfahren genauer. Sei dazu $B = [0, a]$ das betrachtete Gebiet, sei $[0, T]$ das betrachtete Zeitintervall und sei $c > 0$ die Ausbreitungsgeschwindigkeit der Wellen. Gesucht ist eine Funktion $u : [0, a] \times [0, T] \to \mathbb{R}$, die

$$c^2 \, u_{xx}(x, t) - u_{tt}(x, t) = 0 \tag{7.63}$$

erfüllt. Eine Anfangsbedingung sei durch

$$u(x, 0) = f(x), \qquad u_t(x, 0) = 0 \tag{7.64}$$

mit $f : [0, a] \to \mathbb{R}$ gegeben.

7.4.1 Randbedingungen

Wir betrachten verschiedene Randbedingungen:

- beidseitig feste Einspannung (Dirichlet),
- beidseitig keine Übertragung von Vertikalkräften auf den Rand (Neumann),
- linkes Ende ohne vertikale Kraftübertragung (Neumann), rechtes Ende frei.

Im ersten Fall soll also

$$u(0, t) = u(a, t) = 0 \tag{7.65}$$

gelten.

Im zweiten Fall sollen keine Vertikalkräfte auf den Rand übertragen werden. Stellt man sich die Funktion u als Verlauf einer Gitarrensaite vor, soll die Zugkraft also nur horizontal auf die Befestigung am Rand wirken. In vertikale Richtung ist die Befestigung frei beweglich. Erhaltene aus diesen Überlegungen

$$u_x(0, t) = u_x(a, t) = 0. \tag{7.66}$$

Im dritten zu betrachtenden Fall soll die Welle am rechten Rand frei auslaufen können. Das Verhalten am Rand soll also so sein, als würde der betrachtete Bereich B nach rechts beliebig groß sein, wir aber nur den Ausschnitt $[0, a]$ sehen. Zusammen mit der Neumann-Bedingung für den linken Rand erhält man

$$u_x(0, t) = 0, \quad u_t(a, t) + c \, u_x(a, t) = 0. \tag{7.67}$$

Dass diese Randbedingung für unsere Zweck die richtige ist, sieht man, indem man $u(x, t) = g(x - c\,t)$ für eine beliebige Funktion g einsetzt. Diese Wahl von u löst die Wellengleichung einschließlich Randbedingung am rechten Rand und entspricht einer nach rechts laufenden Welle (IDV 745). Nach rechts laufende Wellen können also ungehindert den betrachteten Bereich verlassen. Verzichtet man auf diese Randbedingung, so ist die Lösung der Wellengleichung nicht mehr eindeutig bestimmt.

Nebenbemerkung

Im dritten Fall haben wir nebenbei gesehen, dass nach rechts laufende Wellen stets die Wellengleichung lösen. Gleiches gilt für nach links laufende Wellen $u(x,t) = g(x + c\,t)$. Da die Wellengleichung linear ist, ist auch die Summe zweier Lösungen wieder Lösung, also z.B. die Überlagerung zweier nach links oder rechts laufender Wellen. Zu beachten ist, dass auch andere Lösungen als nur die nach links oder rechts laufenden Wellen in Frage kommen!

7.4.2 Diskretisierung

7.4.2.1 Gitterpunkte Setzen

$$\Delta x := \frac{a}{n_x}, \qquad \Delta t := \frac{T}{n_t} \tag{7.68}$$

und erhalten so Gitterpunkte (x_i, t_j) mit

$$x_i := i\,\Delta x, \qquad t_j := j\,\Delta t \tag{7.69}$$

für $i = 0, 1, \ldots, n_x$ und $j = 0, 1, \ldots, n_t$.

Gesucht sind die Werte

$$u_{i,j} := u(x_i, t_j) \tag{7.70}$$

von u an den Gitterpunkten.

Die Gitterwerte der Funktion f in der Anfangsbedingung bezeichnen wir entsprechend mit

$$f_i := f(x_i). \tag{7.71}$$

7.4.2.2 Innere Punkte Für Punkte im Inneren des betrachteten Gebietes soll die Wellengleichung gelten, wobei wir die zweiten Ableitungen durch entsprechende Differenzenquotienten ersetzen:

$$c^2\,\frac{u_{i-1,j} - 2\,u_{i,j} + u_{i+1,j}}{(\Delta x)^2} - \frac{u_{i,j-1} - 2\,u_{i,j} + u_{i,j+1}}{(\Delta t)^2} = 0 \tag{7.72}$$

für $i = 1, \ldots, n_x - 1$ und $j = 1, \ldots, n_t - 1$.

Umstellen nach $u_{i,j+1}$ liefert

$$u_{i,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,j} - 2\,u_{i,j} + u_{i+1,j}) - (u_{i,j-1} - 2\,u_{i,j}). \tag{7.73}$$

Sind für ein festes j die $u_{i,j-1}$ und die $u_{i,j}$ für alle $i = 0, \ldots, n_x$ bekannt, so erhält man aus dieser Formel die $u_{i,j+1}$ für $i = 1, \ldots, n_x$. Wir können u also Zeitschritt für Zeitschritt berechnen ohne ein Gleichungssystem lösen zu müssen.

Die Obergrenze T für die Zeit spielt hier keine Rolle. Theoretisch kann die Iteration beliebig lang laufen.

7.4.2.3 Anfangsbedingung

Wir haben

$$u_{i,0} = f_i \tag{7.74}$$

für $i = 0, \ldots, n_x$.

Die Ableitungen $u_t(x, 0)$ ersetzen wir durch die zentrale Differenz

$$u_t(x_i, 0) \approx \frac{u_{i,1} - u(x_i, -\Delta t)}{2\,\Delta t}. \tag{7.75}$$

Im Vergleich zu einem einseitigen Differenzenquotient erscheint dies sinnvoller, da der Zeitbereich $[0, T]$ nur einen Ausschnitt aus einem größeren Zeitbereich darstellt, der beobachtete Prozess also auch schon vor $t = 0$ läuft. Andererseits bringt diese Wahl das (scheinbare) Problem mit sich, dass wir $u(x_i, -\Delta t)$ nicht kennen. Aus der kontinuierlichen Anfangsbedingung $u_t(x, 0) = 0$ wird nun die diskrete Anfangsbedingung

$$u_{i,1} = u(x_i, -\Delta t), \quad i = 0, \ldots, n_x. \tag{7.76}$$

In Formel (7.73) für $j = 0$ eingesetzt erhalten wir für den ersten Zeitschritt die Formel

$$u_{i,1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,0} - 2\,u_{i,0} + u_{i+1,0}) - (u_{i,1} - 2\,u_{i,0}). \tag{7.77}$$

Auflösen nach $u_{i,1}$ liefert nun

$$u_{i,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,0} - 2\,u_{i,0} + u_{i+1,0}) + u_{i,0}. \tag{7.78}$$

sodass die unbekannten Werte $u(x_i, -\Delta t)$ gar keine Rolle spielen.

7.4.2.4 Randbedingungen

7.4.2.4.1 Dirichlet

Die diskretisierten Dirichlet-Randbedingungen sind

$$u_{0,j} = u_{n_x,j} = 0, \qquad j = 0, \ldots, n_t. \tag{7.79}$$

7.4.2.4.2 Neumann links

Für die Neumann-Randbedingung links verwenden wir analog zur Zeitableitung in der Anfangsbedingung die zentrale Differenz

$$u_x(0, t_j) \approx \frac{u_{1,j} - u(-\Delta x, t_j)}{2\,\Delta x}, \tag{7.80}$$

welche zur Bedingung

$$u_{1,j} = u(-\Delta x, t_j), \qquad j = 0, \ldots, n_t \tag{7.81}$$

führt. Betrachten wir nun (7.73) für $i = 0$ und setzen dort diese Beziehung ein, so erhalten wir mit

$$u_{0,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{1,j} - 2\,u_{0,j} + u_{1,j}) - (u_{0,j-1} - 2\,u_{0,j}). \tag{7.82}$$

eine Formel zur Berechnung des Randwertes $u_{0,j+1}$, welche für $j \geq 1$ funktioniert. Für $j = 0$ kann mittels Einsetzen von (7.76) für $i = 0$ und Auflösen nach $u_{0,1}$ daraus widerum

$$u_{0,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{1,0} - 2\,u_{0,0} + u_{1,0}) + u_{0,0} \tag{7.83}$$

gewonnen werden.

7.4.2.4.3 Neumann rechts Analog zum linken Rand erhält man für den rechten Rand die Berechnungsvorschriften

$$u_{n_x,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{n_x-1,j} - 2\,u_{n_x,j} + u_{n_x-1,j}) - (u_{n_x,j-1} - 2\,u_{n_x,j}). \tag{7.84}$$

für $j \geq 1$ und

$$u_{n_x,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{n_x-1,0} - 2\,u_{n_x,0} + u_{n_x-1,0}) + u_{n_x,0}. \tag{7.85}$$

7.4.2.4.4 Freier Rand rechts Im Prinzip können die beiden Ableitungen in der Bedingung $u_t(a,t) + c\,u_x(a,t) = 0$ wie oben durch zentrale Differenzenquotienten ersetzt werden. Die auftretenden Funktionswerte außerhalb der Orts- und Zeitintervalle $[0,a]$ und $[0,T]$ können dann wieder eliminiert werden, was allerdings recht aufwendig ist. Der Einfacheit halber beschränken wir uns hier auf einseitige Differenzenquotienten (vorwärts in der Zeit, rückwärts im Ort):

$$u_t(a,t_j) + c\,u_x(a,t_j) \approx \frac{u_{n_x,j+1} - u_{n_x,j}}{\Delta t} + c\,\frac{u_{n_x,j} - u_{n_x-1,j}}{\Delta x}. \tag{7.86}$$

Nullsetzen und Umstellen nach $u_{n_x,j+1}$ liefert

$$u_{n_x,j+1} = u_{n_x,j} - \frac{c\,\Delta t}{\Delta x}\,(u_{n_x,j} - u_{n_x-1,j}) \tag{7.87}$$

für $j \geq 0$.

7.4.2.4.5 Zusammenfassung Zur Berechnung aller $u_{i,j}$ bietet sich nun folgendes Vorgehen an:

1. Für $j = 0$ die Werte f_i verwenden.

2. Für $j = 1$ und $i = 1, \ldots, n_x - 1$ die aus der Anfangsbedingung abgeleitete Formel für innere Punkte verwenden.
3. Für $j = 1$ und $i \in \{0, n_x\}$ die Formeln aus den Randbedingungen nutzen.
4. Für $j = 2, 3, \ldots$:
   a) Für $i = 1, \ldots, n_x - 1$ die Formel für innere Punkte nutzen.
   b) Für $i \in \{0, n_x\}$ die Formeln aus den Randbedingungen nutzen.

7.4.3 Kondition

Man kann zeigen, dass die Berechnung der Funktion u auf dem Gitter gut konditioniert ist, wenn

$$\frac{c\,\Delta t}{\Delta x} \leq 1 \tag{7.88}$$

gilt. Der Wert auf der linken Seite heißt auch **Courant-Zahl**.

7.4.4 Konvergenz

Man kann zeigen, dass für

$$\frac{c\,\Delta t}{\Delta x} = 1 \tag{7.89}$$

die auf dem Gitter (ohne Rundungsfehler) berechneten Werte für u exakt sind. Werden Δx und Δt gleichmäßig verkleinert, so konvergiert die Lösung des diskretisierten Problems also gegen die exakte Lösung.

Ist die Courant-Zahl größer als 1, so treten im Laufe der Zeit immer größere Abweichungen zwischen diskretisierter und exakter Lösung auf.

7.4.5 Implementierung

Die Implementierung kann entsprechend den Formeln oben erfolgen.

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

def wave1d(a, T, c, f, nx, nt, left='d', right='d'):

    delta_x = a / nx
    delta_t = T / nt
    C = c * delta_t / delta_x
    print(f'Courant -Zahl: {C:.2f}')

    x = np.linspace(0, a, nx + 1)
```

```
    u = np.empty((nt + 1, nx + 1), dtype=float)

    # Anfangsbedingungen
    u[0, :] = f(x)
    u[1, 1: -1] = 0.5 * C ** 2 * (u[0, : -2] - 2 * u[0, 1: -1] + u[0,
    ↪  2:]) + u[0, 1: -1]
    if left == 'd':
        u[1, 0] = 0
    else:
        u[1, 0] = 0.5 * C ** 2 * (2 * u[0, 1] - 2 * u[0, 0]) + u[0, 0]
    if right == 'd':
        u[1, -1] = 0
    elif right == 'n':
        u[1, -1] = 0.5 * C ** 2 * (2 * u[0, -2] - 2 * u[0, -1]) + u[0,
        ↪  -1]
    else:
        u[1, -1] = u[0, -1] - C * (u[0, -1] - u[0, -2])

    # Zeitschritte
    for j in range(1, nt):

        # innere Punkte
        u[j + 1, 1: -1] = C ** 2 * (u[j, : -2] - 2 * u[j, 1: -1] + u[j,
        ↪  2:]) \
                        - (u[j - 1, 1: -1] - 2 * u[j, 1: -1])

        # Randbedingungen
        if left == 'd':
            u[j + 1, 0] = 0
        else:
            u[j + 1, 0] = 0.5 * C ** 2 * (2 * u[j, 1] - 2 * u[j, 0]) \
                          - (u[j - 1, 0] - 2 * u[j, 0])
        if right == 'd':
            u[j + 1, -1] = 0
        elif right == 'n':
            u[j + 1, -1] = 0.5 * C ** 2 * (2 * u[j, -2] - 2 * u[j, -1])
            ↪  \
                           - (u[j - 1, -1] - 2 * u[j, -1])
        else:
            u[j + 1, -1] = u[j, -1] - C * (u[j, -1] - u[j, -2])

    return u
```

Die einzelnen Zeitschritte können als Animation visualisiert werden.

```
import matplotlib.animation as anim
from IPython.display import HTML

def show_anim(u, a, T):

    nt = u.shape[0] - 1
    nx = u.shape[1] - 1
    x = np.linspace(0, a, nx + 1)

    # Bildrate auf 25 fps begrenzen
    fps_real = nt / T
    fps_show = 25
    skip_frames = int(np.floor(fps_real / fps_show))
    n_frames = (nt + 1) // (skip_frames + 1)

    # Figure/Axes erstellen
    fig, ax = plt.subplots()
    ax.set_xlim(0, a)
    ax.set_ylim( -1.1, 1.1)
    ax.set_xlabel('x')
    ax.set_ylabel('u(x,t)')
    ax.grid()

    # Linienobjekt erstellen (wird während Animation verändert)
    line = ax.plot(0, 0, ' -b')[0]

    # Update -Funktion
    def update(frame):
        line.set_data(x, u[(skip_frames + 1) * frame, :])

    # Animation erstellen und anzeigen
    fa = anim.FuncAnimation(fig, update, frames=n_frames,
    ↪  interval=1000/fps_show)
    display(HTML(fa.to_jshtml()))
    plt.close()  # verhindert automatischen Aufruf von plt.show() durch
    ↪  Jupyter
```

7.4.6 Experimente

Verwenden folgende Parameter:

```
a = 2
T = 4
```

```
c = 1
nx = 100
nt = int(np.ceil(c * nx * T / a))  # Courant = 1 oder wenigstens <= 1
print(f'{nt} Zeitschritte')
```

```
200 Zeitschritte
```

7.4.6.1 Dirichlet-Randbedingungen

```
f = lambda x: np.sin(1/a * np.pi * x)
u = wave1d(a, T, c, f, nx, nt, 'd', 'd')
show_anim(u, a, T)
```

```
Courant -Zahl: 1.00
```

0

[ ] Once [x] Loop [ ] Reflect

7.4.6.2 Neumann-Randbedingungen Möchten wir Neumann-Randbedingungen verwenden, so sollte die Anfangsbedingung diese auch erfüllen:

```
f = lambda x: np.cos(2/a * np.pi * x)
u = wave1d(a, T, c, f, nx, nt, 'n', 'n')
show_anim(u, a, T)
```

```
Courant -Zahl: 1.00
```

0

[ ] Once [x] Loop [ ] Reflect

7.4.6.3 Links Neumann-, rechts freie Randbedingung

```
f = lambda x: np.exp( -(x - 0.5 * a) ** 2 / 0.05)
u = wave1d(a, T, c, f, nx, nt, 'n', 'f')
show_anim(u, a, T)
```

```
Courant -Zahl: 1.00
```

0

[ ] Once [x] Loop [ ] Reflect

7.4.6.4 Links Dirichlet-, rechts Neumann-Randbedingung

Verlängern den Beobachtungszeitraum um eine volle Periode des Schwingungsvorgangs zu sehen:

```
T = 8
nt = int(np.ceil(c * nx * T / a))  # Courant = 1 oder wenigstens <= 1
print(f'{nt} Zeitschritte')

u = wave1d(a, T, c, f, nx, nt, 'd', 'n')
show_anim(u, a, T)
```

```
400 Zeitschritte
Courant -Zahl: 1.00
```

0

[ ] Once [x] Loop [ ] Reflect

7.4.6.5 Courant-Zahl zu groß

```
T = 4
nt = int(np.ceil(0.99 * c * nx * T / a))  # Courant > 1
print(f'{nt} Zeitschritte')

f = lambda x: np.sin(1/a * np.pi * x)
u = wave1d(a, T, c, f, nx, nt, 'd', 'd')
show_anim(u, a, T)
```

```
198 Zeitschritte
Courant -Zahl: 1.01
```

0

[ ] Once [x] Loop [ ] Reflect

Obwohl die Courant-Zahl hier nur ganz wenig über der Eins liegt, wird die Berechnung schlecht konditioniert, sodass die unvermeidbaren Rundungsfehler innerhalb weniger Rechenschritte extrem verstärkt werden und die numerische Lösung nichts mehr mit der exakten Lösungen des diskretisierten Problems zu tun hat.

7.5 Schwache Formulierung von PDEs

Ziel ist die Umsetzung einer weiteren, deutlich besseren Klasse von Lösungsverfahren für partielle Differentialgleichung: die so genannten Finite-Elemente-Verfahren (FEM). Diese lösen einige Probleme, die bei FDM große Schwierigkeiten bereiten:

- Umgang mit komplexen Gebietsrändern,
- unterschiedliche Diskretisierungsfeinheit in verschiedenen Teilgebieten,
- Handhabung von Lösungsfunktionen mit nicht glatten Stellen (z.B. “Knicke” oder Sprünge an Materialgrenzen).

Finite-Elemente-Verfahren arbeiten nicht direkt mit der gegebenen PDE, sondern verwenden eine Umformulierung als Integralgleichungen. Diese sogenannte schwache Formulierung benötigt einigen mathematischen Unterbau, der über die üblichen Grundvorlesungen zur Mathematik hinausgeht. Hier führen wir die Dinge nur soweit ein, dass wir FEM anwenden können, und verzichten an manchen Stellen auf hundertprozentige Exaktheit.

7.5.1 Quadratisch integrierbare Funktionen

Sei $B \subseteq \mathbb{R}^n$ beschränkt. Dann können wir jeder Funktion $f : B \to \mathbb{R}$, für die das Integral existiert und endlich ist, den Wert

$$\|f\|_2 := \sqrt{\int_B f^2 \,\mathrm{d}b} \tag{7.90}$$

zuordnen. Funktionen, für die das Integral existiert und endlich ist, heißen **quadratisch integrierbar**.

Nebenbemerkung

Bei dem Integral handelt es sich streng genommen nicht um das übliche Riemann-Bereichsintegral, sondern um ein Lebesgue-Integral. Dieses existiert für mehr Funktionen als das Riemann-Integral, hat aber den selben Wert, wenn beide existieren. In der Mathematik gibt es viele verschiedene Integralbegriffe, die sich nur in Details unterscheiden, welche für Anwender praktisch nie eine Rolle spielen.

Skalare Vielfache und Summen quadratische integrierbarer Funktionen sind wieder quadratisch Integrierbar (man sagt: sie bilden einen linearen Raum oder auch Vektorraum).

Merke

Für zwei quadratisch integrierbare Funktionen f und g kann $\|f - g\|_2 = 0$ gelten, obwohl sie nicht gleich sind. Zum Beispiel könnten sich ihre Funktionswerte nur

an genau einer Stelle $x \in B$ unterscheiden.

Gilt $\|f - g\|_2 = 0$ für zwei quadratisch integrierbare Funktionen, so sagt man, dass f und g **fast überall gleich** sind.

Für die praktische Anwendung genügt es, wenn man sich unter "fast überall gleich" vorstellt, dass zwei Funktionen sich nur an endlich vielen oder abzählbar vielen Stellen unterscheiden. Es gibt aber auch überabzählbare Mengen, die beim Integrieren keine Rolle spielen (z.B. die Cantor-Menge).

Merke

Die Relation "fast überall gleich" zwischen Funktionen auf $B \subseteq \mathbb{R}^n$ ist eine Äquivalenzrelation. Das heißt insbesondere, dass für Funktionen f, g, h aus der Fast-Überall-Gleichheit von f zu g und von g zu h stets die Fast-Überall-Gleichheit von f zu h folgt. Man kann die Menge der quadratisch integrierbaren Funktionen also vollständig in paarweise disjunkte Teilmengen (Äquivalenzklassen) untereinander fast überall gleicher Funktionen zerlegen.

Bezeichnen wir mit

$$[f] := \{g : \ f \text{ und } g \text{ sind fast überall gleich}\} \tag{7.91}$$

eine solche Menge untereinander fast überall gleicher Funktionen, so gilt

$$\|g\|_2 = \|f\|_2 \quad \text{für alle } g \in [f]. \tag{7.92}$$

Wir können diesen Wert also als Eigenschaft der gesamten Äquivalenzklasse ansehen:

$$\|[f]\|_2 := \|f\|_2. \tag{7.93}$$

Die Menge aller Äquivalenzklassen (bzgl. Fast-Überall-Gleichheit) quadratisch integrierbarer Funktionen auf B bezeichnen wir mit $L^2(B)$ und nennen sie den **Raum der quadratisch integrierbaren Funktionen**.

$L^2(B)$ ist ein normierter linearer Raum mit der Norm $\|[f]\|_2$ für $[f] \in L^2(B)$. Die Summe zweier Äquivalenzklassen ist dabei die Äquivalenzklasse, zu der die Summe zweier beliebiger Vertreter der Summanden-Äquivalenzklassen gehört. Analog ist das skalare Vielfache einer Äquivalenzklasse definiert. "Normiert" heißt an dieser Stelle, dass wir den Elementen des Raumes eine "Länge" zuordnen können und damit auch Abstände zwischen Elementen messen können.

Merke

Solange man sich auf die Operationen

- Summe, Produkt, Quotient,
- skalares Vielfaches,
- Integral

beschränkt, spielt es keine Rolle, welche Vertreter aus den beteiligten Äquivlenzklassen gewählt werden. Wir können (und werden) also so tun, als wären die Elemente von $L^2(B)$ selbst Funktionen. Entsprechend werden wir auch die Schreibweise $[f]$ im Folgenden nicht mehr verwenden, sondern direkt f schreiben und damit aber die Äquivalenzklasse meinen.

"Verbotene" Operationen in diesem Kontext sind Punktauswertungen. Einem Ausdruck der Form $[f](x)$ kann kein Sinn beigemessen werden, da $g(x)$ für jedes $g \in [f]$ anders aussehen kann.

In diesem Sinne sehen wir die Elemente von $L^2(B)$ als Funktionen an, die wir zwar im Ganzen betrachten, aber nicht an einzelnen Punkten auswerten können. Man spricht manchmal auch von verallgemeinerten Funktionen.

Beispiel

Sei $B = (-1, 1)$. Die drei Funktionen

$$f(x) := \begin{cases} -1, & x \leq 0 \\ 1, & x > 0, \end{cases} \tag{7.94}$$

$$g(x) := \begin{cases} -1, & x < 0 \\ 1, & x \geq 0, \end{cases} \tag{7.95}$$

$$h(x) := \begin{cases} -1, & x < 0 \\ 0, & x = 0, \\ 1, & x > 0 \end{cases} \tag{7.96}$$

(und noch viele weitere) gehören zur selben Äquivalenzklasse bzgl. Fast-Überall-Gleichheit. Wir sehen diese Funktionen also als identisch an, solange wir keine Punktauswertungen benötigen.

Nebenbemerkung

Da der Rand ∂B eines (praktisch relevanten) Gebiets B eine "Menge vom Maß Null" ist, also beim Integrieren darüber nichts zum Wert des Integrals beiträgt, spielt es für die Definition von $L^2(B)$ keine Rolle, ob B offen oder abgeschlossen oder nichts von beiden ist.

7.5.2 Schwache Ableitungen

Der übliche Ableitungsbegriff beruht auf der Punktauswertung von Funktionen (Differenzenquotient), ist also für verallgemeinerte Funktionen unbrauchbar. Führen deshalb Ableitungen nun anders ein als bisher gewohnt und werden dann feststellen, dass das Ergebnis im Wesentlichen das selbe ist, aber auch verallgemeinerte Funktionen abdeckt.

Merke

Bezeichnen mit $C_0^1(B)$ die Menge aller auf dem beschränkten Gebiet B stetig differenzierbaren Funktionen, die auf dem Rand ∂B Null sind (präziser: sich stetig durch Null auf den Rand fortsetzen lassen).

Eine Funktion $w \in L^2(B)$ heißt **(verallgemeinerte) partielle Ableitung** der Funktion $u \in L^2(B)$ nach x_i, wenn

$$\int_B u \frac{\partial \varphi}{\partial x_i} \,\mathrm{d}b = -\int_B w\,\varphi \,\mathrm{d}b \quad \text{für alle } \varphi \in C_0^1(B) \tag{7.97}$$

gilt.

In diesem Kontext werden die Funktionen φ auch **Testfunktionen** genannt.

Merke

Ist $B = (a, b)$ und ist u stetig differenzierbar, so ist u stets verallgemeinert differenzierbar und die verallgemeinerte Ableitung stimmt mit der üblichen Ableitung überein (IDV 750).

Analoges gilt für partielle Ableitungen.

Beispiel

Die verallgemeinerte Ableitung von

$$f(x) := |x| \tag{7.98}$$

auf $B = (-1, 1)$ ist

$$f'(x) := \begin{cases} -1, & x \le 0 \\ 1, & x > 0 \end{cases} \tag{7.99}$$

(IDV 755).

Beispiel

Die verallgemeinerte Ableitung von

$$f(x) := \begin{cases} -1, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{7.100}$$

auf $B = (-1, 1)$ existiert nicht, ist also insbesondere nicht (!) die Nullfunktion (IDV 760).

Offensichtlich ist es bei der Definition der verallgemeinerten Ableitung völlig egal, welcher konkrete Vertreter der betrachteten Äquivalenzklasse gewählt wird. Verallgemeinerte Differenzierbarkeit ist also wieder eine Eigenschaft der gesamten Äquivalenzklasse.

Merke

Die Menge aller verallgemeinert differenzierbaren quadratisch integrierbaren Funktionen auf einem Gebiet B, deren Ableitungen wieder quadratisch integrierbar sind, bezeichnen wir mit $H^1(B)$ (**Sobolev-Raum 1. Ordnung**). Sie ist ein normierter linearer Raum mit der Norm

$$\|f\|_{2,1} := \sqrt{\|f\|_2^2 + \|f'\|_2^2}. \tag{7.101}$$

Nebenbemerkung

Für $H^1(B)$ ist natürlich auch die $L^2(B)$-Norm eine Norm. Allerdings ist man insbesondere in Fehlerabschätzungen bestrebt Normen zu wählen, die den betrachteten Raum charakterisieren. Man kann zeigen, dass für eine differenzierbare Funktion $f \in L^2(B)$ gilt:

$$f \in H^1(B) \quad \Leftrightarrow \quad \|f\|_{2,1} < \infty. \tag{7.102}$$

Insbesondere ist die $H^1(B)$-Norm "stärker" als die $L^2(B)$-Norm im Sinne von

$$\|f\|_2 \leq \|f\|_{2,1}. \tag{7.103}$$

Abschätzungen für in der $H^1(B)$-Norm ausgedrückte Fehler liefern also automatisch auch Abschätzungen für $L^2(B)$-Fehler. Allerdings kann man aus $H^1(B)$-basierten Abschätzungen zusätzlich noch schließen, dass nicht nur der Fehler in den Funktionswerten, sondern auch der Fehler in den Ableitungen (Anstiegsverhalten) klein ist.

Merke

Für $B = (a, b)$ (offen!) kann man zeigen, dass $H^1(B)$-Funktionen stetig sind in dem Sinn, dass jede Äquivalenzklasse einen stetigen Vertreter enthält. Punktauswertungen sind hier also sinnvoll möglich!

Für zwei- und höherdimensionale Gebiete gilt dies nicht. Jedoch impliziert k-fache verallgemeinerte Differenzierbarkeit in einem offenen d-dimensionalen Gebiet (mit quadratisch integrierbaren Ableitungen) m-malige klassische stetige Differenzierbarkeit, falls

$$m < k - \frac{d}{2} \tag{7.104}$$

gilt (Sobolev'scher Einbettungssatz). Für zweidimensionale Gebiete ($d = 2$) folgt Stetigkeit ($m = 0$) also beispielsweise aus zweimaliger verallgemeinerter Differenzierbarkeit ($k = 2$).

7.5.3 Idee der schwachen Formulierung

Da man gern auch nicht klassisch differenzierbare Funktionen als Lösungen von PDEs zulassen möchte, erweitert man die Lösungssuche auf verallgemeinert differenzierbare Funktionen. Verallgemeinerte Differenzierbarkeit ist jedoch nicht punktweise als Grenzwert von Differenzenquotienten definiert, sondern "global" für die ganze Funktion. Entsprechend müssen die betrachteten PDE durch im Wesentlichen äquivalente Integralgleichungen ersetzt werden, die dann für gewisse Mengen von Testfunktionen erfüllt sein sollen.

Die Umformulierung als Integralgleichung öffnet gleichzeit die Tür für flexiblere Diskretisierungs- und Lösungsverfahren, die bei klassisch formulierten PDEs so nicht einsetzbar sind.

Wie eine solche "schwache" Formulierung aussieht, hängt von der konkreten PDE ab. Für einige Klassen von PDE haben sich allgemein übliche schwache Formulierungen etabliert, sodass man dann auch von der (!) schwachen Formulierung spricht. Ziele beim Suchen einer schwachen Formulierung sind:

- Vermeidung klassischer Ableitungen,
- möglichst niedrige Ordnung auftretender (verallgemeinerter) Ableitungen,
- einfaches algorithmisches Lösen mit Standardverfahren.

Die ersten beiden Punkte werde auch unter "geringe Regularität der Lösungen" zusammengefasst. Hinter dem dritten Punkt verbirgt sich das in Kürze zu behandelnde Finite-Elemente-Verfahren. Als Vorbereitung betrachten wir eine recht allgemeine Problemklasse, die übliche schwache Formulierungen für PDEs abdeckt, aber auch abseits von PDEs von Bedeutung ist.

7.5.4 Abstrakte Form schwacher Formulierungen

Merke

Sei V ein hinreichend "schöner" Funktionenraum, z.B. $L^2(B)$ oder $H^1(B)$. Weiter sei $a : V \times V \to \mathbb{R}$ in beiden Argumenten linear ("Bilinearform") und $b : V \to \mathbb{R}$ linear. Gesucht ist eine Funktion $u \in V$, sodass

$$a(u, v) = b(v) \quad \text{für alle } v \in V \tag{7.105}$$

gilt.

Man kann zeigen, dass für dieses Problem stets eine eindeutig bestimmte Lösung u existiert, sofern a und b folgende Bedingungen erfüllen:

- a ist beschränkt, d.h. es existiert eine Konstante $c > 0$ mit

$$|a(v_1, v_2)| \leq c \, \|v_1\| \, \|v_2\| \quad \text{für alle } v_1, v_2 \in V. \tag{7.106}$$

- b ist beschränkt, d.h. es existiert eine Konstante $c > 0$ mit

$$|b(v)| \leq c \, \|v\| \quad \text{für alle } v \in V. \tag{7.107}$$

- a ist elliptisch, d.h. es existiert eine Konstante $c > 0$ mit

$$|a(v, v)| \geq c \, \|v\|^2 \quad \text{für alle } v \in V. \tag{7.108}$$

Nebenbemerkung

Hinreichend "schön" sind Funktionenräume, die ein Skalarprodukt besitzen, also das Messen von Winkeln erlauben. In der mathematischen Literatur spricht man hier von Hilbert-Räumen. Das Skalarprodukt zwischer Funktionen u und v in $L^2(B)$ ist durch

$$\int_B u \, v \, \mathrm{d}b \tag{7.109}$$

gegeben.

Finden wir für eine gegebene PDE eine schwache Formulierung in der Form (7.105), die die drei Bedingungen erfüllt, so sind Existenz und Eindeutigkeit einer Lösung garantiert.

7.5.5 Diskretisierung der abstrakten Formulierung

Üblicherweise ist V unendlichdimensional. Für das Lösen am Computer müssen wir uns also auf einen (endlichdimensionalen) Teilraum V_m einschränken. Hier bezeichne m die Dimension des Teilraums. In diesem Teilraum können wir eine Basis $v_1, \ldots, v_m \in V$ wählen. Jede Funktion v in V_m kann also in der Form

$$v = c_1\, v_1 + \cdots + c_m\, v_m \tag{7.110}$$

mit gewissen Zahlen $c_1, \ldots, c_m$ geschrieben werden. Die Gleichung (7.105) gilt nun genau dann für alle $v \in V_m$, wenn die m Gleichungen

$$a(u, v_k) = b(v_k) \quad \text{für } k = 1, \ldots, m \tag{7.111}$$

erfüllt sind (IDV 765).

Analog wählt man einen zweiten (oder den gleichen) Teilraum U_n von V und beschränkt die Lösungssuche auf diesen. Die gesuchte Lösung u wird mittels einer Basis $u_1, \ldots, u_n \in U_n$ als

$$u = c_1\, u_1 + \cdots + c_n\, u_n \tag{7.112}$$

geschrieben, sodass nur noch die Zahlen $c_1, \ldots, c_n$ zu ermitteln sind. Einsetzen in (7.111) liefert das lineare Gleichungssystem

$$\sum_{l=1}^{n} c_l\, a(u_l, v_k) = b(v_k) \quad \text{für } k = 1, \ldots, m \tag{7.113}$$

zur Berechnung der c_l und damit zur Berechung einer Näherungslösung u von (7.105) (IDV 770).

Merke

Man kann zeigen, dass unter obigen Voraussetzungen für Existenz und Eindeutigkeit einer Lösung auch das diskrete Problem (lineares Gleichungssystem) stets eine eindeutige Lösung besitzt, sofern $m = n$ gilt. Außerdem kann man zeigen, dass unter diesen Voraussetzungen und für $m = n$ die Lösungen des diskreten Problems gegen die Lösung des kontinuierlichen Problems konvergieren (bzgl. der Norm in V), wenn $n \to \infty$. Die Konvergenzgeschwindigkeit hängt dabei davon ab, wie groß der Abstand zwischen der Lösung des kontinuierlichen Problems und den gewählten Teilräumen U_n ist.

7.5.6 Beispiel: Schwache Formulierung für eine PDE erster Ordnung

Betrachten als einfaches Beispiel eine PDE erster Ordnung auf einem eindimensionalen Gebiet (also eigentlich eine gewöhnliche Differentialgleichung). Zu gegebener Funktion f ist eine Funktion u gesucht, sodass

$$u'(x) = f(x) \quad \text{für alle } x \in B := (0, 1) \tag{7.114}$$

gilt. Offensichtlich ist u eine Stammfunktion von f. Um Eindeutigkeit zu sichern fordern wir noch

$$u(0) = 0. \tag{7.115}$$

Die Funktion f könnte z.B. eine von einem Beschleunigungssensor gemessene Beschleunigung sein und u ist die daraus zu ermittelnde Geschwindigkeit (x wäre dann die Zeit).

Klar ist, dass die Lösung u klassisch differenzierbar sein muss. Hat f aber beispielsweise eine Sprungstelle (sprunghafte Erhöhung der Beschleunigung durch "Auffahrunfall"), so müsste u einen Knick (nicht differenzierbare Stelle) haben. Dieser Fall kann mit der klassischen Formulierung der PDE nicht abgedeckt werden.

Um eine schwache Formulierung zu erhalten multiplizieren wir die Differentialgleichung zunächst mit Testfunktionen $v : [0, 1] \to \mathbb{R}$ und integrieren beide Seiten über das Gebiet:

$$\int_0^1 u'(x)\, v(x)\, \mathrm{d}x = \int_0^1 f(x)\, v(x)\, \mathrm{d}x. \tag{7.116}$$

Ist $f \in L^2(B)$, so kann man zeigen, dass das rechte Integral für alle $v \in L^2(B)$ einen endlichen Wert liefert. Für $u \in H^1(B)$ ist entsprechend das linke Integral wohldefiniert. Können die Gleichheit der Integrale also für alle $v \in V := L^2(B)$ fordern.

Man kann nun zeigen, dass diese schwache Formulierung der PDE für jedes $f \in L^2(B)$ eine eindeutige Lösung besitzt; auch dann, wenn wir u' als schwache Ableitung interpretieren. Es sind nun auch nicht klassisch differenzierbare Lösungen möglich. Die von den Lösungen geforderte Regularität ist also niedriger als bei der ursprünglichen PDE ohne wesentliche Änderungen am Modell.

Die gefundene schwache Formulierung passt in das abstrakte Schema (7.105), wenn wir

$$a(u, v) := \int_0^1 u'(x)\, v(x)\, \mathrm{d}x \tag{7.117}$$

und

$$b(v) := \int_0^1 f(x)\, v(x)\, \mathrm{d}x \tag{7.118}$$

wählen. Somit kann das entsprechende Diskretisierungsverfahren eingesetzt werden, welches zu einem linearen Gleichungssystem führt.

7.5.7 Schwache Formulierung für elliptische PDE

Leiten eine schwache Formulierung für die Poisson-Gleichung mit Dirichlet- und Neumann-Randbedingungen her. Völlig analog kann man schwache Formulierungen für alle elliptischen (!) PDE zweiter Ordnung erhalten. Entsprechend sind bei allen elliptischen PDE Standardlösungsverfahren einsetzbar.

Sei $B \subseteq \mathbb{R}^d$ ein Gebiet und $f : B \to \mathbb{R}$ eine gegebene Funktion. Zur PDE

$$\Delta u(x) = f(x) \quad \text{für } x \in B \tag{7.119}$$

fordern wir die Randbedingungen

$$u(x) = g_1(x) \quad \text{für } x \in \Gamma_1 \tag{7.120}$$

und

$$\frac{\partial}{\partial n} u(x) = g_2(x) \quad \text{für x} \in \Gamma_2, \tag{7.121}$$

wobei $\partial B = \Gamma_1 \cap \Gamma_2$ eine disjunkte Zerlegung des Randes von B ist. Γ_1 ist gerade der Teil des Randes, auf dem die gesuchte Funktion bekannt ist; Γ_2 ist der Teil des Randes, auf dem der Fluss über den Rand gegeben ist.

Offensichtlich muss die gesuchte Lösung u mindestens zweimal (klassisch) differenzierbar sein. Mit den folgenden Schritten erhalten wir eine schwache Formulierung:

- Multipliziere die PDE mit einer Testfunktion v, die auf dem Rand Γ_1 Null ist.
- Integriere beide Seiten der entstandenen Gleichung über ganz B.
- Verwende die Green'sche Formel (partielle Integration für Bereichsintegrale)

  $$\int_B \Delta u \, v \, \mathrm{d}b = \int_{\partial B} \frac{\partial}{\partial n} u \, v \, \mathrm{d}a - \int_B \nabla u \circ \nabla v \, \mathrm{d}b \tag{7.122}$$

  zum Verringern der Ableitungsordnung von u, wobei das Integral über ∂B ein Oberflächenintegral erster Art ist (bzw. ein Kurvenintegral erster Art bei $d = 2$).
- Vereinfache die Gleichung durch Einsetzen der Neumann-Randbedingungen und durch Beschränkung auf Testfunktionen, die auf Γ_1 Null sind.

Suchen nun also eine Funktion u, sodass

$$\int_B \nabla u \circ \nabla v \, \mathrm{d}b = \int_{\Gamma_2} g_2 \, v \, \mathrm{d}a - \int_B f \, v \, \mathrm{d}b \tag{7.123}$$

für eine hinreichend große Menge von Testfunktionen v gilt (IDV 775).

Die gesuchte Lösung muss nun einerseits in $H^1(B)$ liegen (damit ∇u im schwachen Sinn existiert) und andererseits die Dirichlet-Randbedingung erfüllen. Die Testfunktionen müssen ebenfalls in $H^1(B)$ liegen und auf dem Rand Γ_1 Null sein. Wählen also

$$V := \{v \in H^1(B) : \ v(x) = 0 \text{ für } x \in \Gamma_1\}. \tag{7.124}$$

Die Menge $\{u \in H^1(B) : \ u(x) = g_1(x) \text{ für } x \in \Gamma_1\}$, auf die wir die Lösungssuche eingeschränkt haben, ist kein linearer Raum mehr (wegen Randbedingung) und unterscheidet sich von V, was zu Problemen bei der Anwendung der abstrakten Formulierung mittels Bilinearform a führt. Wählen wir eine beliebige, feste Funktion u_1 aus der Suchmenge, so können wir aber jedes andere u aus dieser Menge als

$$u = u_0 + u_1 \quad \text{mit } u_0 \in V \tag{7.125}$$

schreiben. Einsetzen in die schwache Formulierung liefert die nur formal, nicht inhaltlich geänderte schwache Formulierung:

Merke

Finde $u_0 \in V$, sodass

$$\int_B \nabla u_0 \circ \nabla v \, \mathrm{d}b = \int_{\Gamma_2} g_2 \, v \, \mathrm{d}a - \int_B f \, v \, \mathrm{d}b - \int_B \nabla u_1 \circ \nabla v \, \mathrm{d}b \tag{7.126}$$

für alle $v \in V$ gilt. Dabei sind f, u_1, g_1 gegeben.

Nebenbemerkung

Funktionen in $H^1(B)$ sind nur für eindimensionale Gebiete B stetig. Für zwei- und höherdimensionale Gebiete kann Punktauswertungen also kein Sinn beigemessen werden, sodass Vorgaben wie "auf dem Rand ist die Funktion Null" eigentlich nicht möglich sind. Man kann dennoch eine sinnvolle und mathematisch korrekte Formulierung für "auf dem Rand Null" finden. Grob formuliert: Die Funktion lässt sich beliebig gut durch stetige Funktionen annähern, die auf dem Rand Null sind. Die Ausarbeitung der Details übersteigt jedoch unsere mathematischen Kenntnisse, sodass wir es bei der etwas unsauberen Formulierung belassen.

Während in der ursprünglichen PDE die gesuchte Lösung noch zweimal klassisch differenzierbar sein musste, müssen wir bei der schwachen Formulierung nur einmalige schwache Differenzierbarkeit fordern. Außerdem passt die schwache Formulierung mit

$$a(u, v) := \int_B \nabla u \circ \nabla v \, \mathrm{d}b \tag{7.127}$$

und

$$b(v) := \int_{\Gamma_2} g_2 \, v \, \mathrm{d}a - \int_B f \, v \, \mathrm{d}b - \int_B \nabla u_1 \circ \nabla v \, \mathrm{d}b \tag{7.128}$$

in das abstrakte Schema, ist also Standardlösungsverfahren zugänglich.

Merke

Man kann zeigen, dass die Voraussetzungen für Existenz und Eindeutigkeit einer Lösung bei der abstrakten Formulierung durch die schwache Formulierung der Poisson-Gleichung erfüllt sind. Auch kann man unter geeigneten Voraussetzungen an die Form des Gebiets B zeigen, dass die Lösung stetig (in geeignetem Sinn!) von den "Daten" f, g_1, g_2 abhängt.

7.6 Finite-Elemente-Verfahren

Finite-Elemente-Verfahren (kurz: FEM für *finite elemente methods*) basieren auf der schwachen Formulierung von PDE. Hauptmerkmal ist, dass für die Diskretisierung Basisfunktionen $v_k : B \to \mathbb{R}$ mit sehr kleinem Träger

$$\operatorname{supp} v_k := \{x \in B : \ v_k(x) \neq 0\} \tag{7.129}$$

verwendet werden. Dadurch entsteht ein lineares Gleichungssystem mit dünn besetzter Systemmatrix, welches effizient gelöst werden kann. Für die Diskretisierung werden keine regelmäßigen Gitter von Stützstellen benötigt, sodass auch PDE auf komplexen Gebieten B mittels FEM gelöst werden können.

7.6.1 Ausgangspunkt

Gegeben sei die schwache Formulierung einer PDE in der Form

$$a(u, v) = b(v) \quad \text{für alle } v \in V, \tag{7.130}$$

wobei a, b, V gegeben sind und u gesucht ist (vgl. Section 7.5.4).

Weiter sei $V_n \subseteq V$ ein endlichdimensionaler Teilraum von V und $\{v_1, \ldots, v_n\}$ sei eine Basis in V_n, d.h. jede Funktion in V_n kann als Linearkombination der Funktionen $v_1, \ldots, v_n$ dargestellt werden. Durch Einschränkung der schwachen Formulierung auf V_n erhalten wir das diskretisierte Problem

$$A\,\underline{u} = \underline{b} \tag{7.131}$$

mit der rechten Seite

$$\underline{b} := \begin{bmatrix} b(v_1) \\ \vdots \\ b(v_n) \end{bmatrix} \tag{7.132}$$

und der Systemmatrix (hier auch **Steifigkeitsmatrix** genannt)

$$A := \begin{bmatrix} a(v_1, v_1) & \cdots & a(v_n, v_1) \\ \vdots & & \vdots \\ a(v_1, v_n) & \cdots & a(v_n, v_n) \end{bmatrix} \tag{7.133}$$

(vgl. Section 7.5.5). Aus dem Vektor $\underline{u} \in \mathbb{R}^n$ erhalten wir mittels

$$u := \underline{u}_1\, v_1 + \cdots + \underline{u}_n\, v_n \tag{7.134}$$

die gesuchte Lösung der (schwachen Formulierung der) PDE.

Einziger noch zu klärender Punkt ist die konkrete Wahl der Basisfunktionen v_k.

7.6.2 Wahl der Basis

Die Wahl der Basis $\{v_1, \ldots, v_n\}$ erfolgt in zwei Schritten:

1. Zerlegung des Gebiets B in disjunkte gleichartige Teilmengen $T_1, \ldots, T_N \subseteq B$ (die sogenannten **finiten Elemente**). Dabei ist mit "gleichartig" gemeint, dass zum Beispiel alle Teilmengen Dreiecke sind oder alle sind Vierecke oder alle sind Tetraeder (für $B \subseteq \mathbb{R}^3$) usw.
2. Wahl der v_k so, dass die Träger $\operatorname{supp} v_k$ jeweils der Vereinigung von nur wenigen Teilgebieten T_l entsprechen.

Die Zerlegung in Teilgebiete erfolgt stets so, dass sich nur Ecken mit Ecken berühren, nie Ecken mit Kanten oder Flächen. Die Ecken werden auch **Knoten** genannt und im Folgenden mit $P_1, \ldots, P_m$ bezeichnet.

Sind die T_l Dreiecke, so werden die v_k gern als stückweise lineare Funktionen gewählt: Zu jedem Knoten P_k wird genau eine Basisfunktion gewählt. Diese ist 1 an dem Knoten und Null an allen anderen Knoten (IDV 780).

Analog kann bei Tetraedern in $\mathbb{R}^3$ vorgegangen werden. Neben stückweise linearen sind auch stückweise quadratische usw. Basisfunktionen üblich. Da Funktionen höherer Ordnung mehr Freiheitsgrade besitzen, werden je nach Bedarf zusätzliche Knoten auf den Berührungskanten oder -flächen der Teilgebiete T_l eingefügt.

7.6.3 Gebietszerlegung

Die konkrete Wahl der Zerlegung in Teilgebiete unterliegt diversen Einflussfaktoren:

- erforderliche Genauigkeit der Berechnungen (ggf. höhere Knotendichte an "interessanten" Stellen),
- exakte Darstellung von Materialgrenzen (diese sollten immer auf Kanten/Flächen der Zerlegung liegen),
- exakte Darstellung der Grenzen zwischen verschiedenen Typen von Randbedingungen,
- exakte Darstellung von Unstetigkeitsstellen in den Randbedingungen,
- aus theoretischen Ergebnissen bekannte Einflüsse auf den Diskretisierungsfehler (z.B. dichtere Knoten in der Nähe konkaver Gebietsgrenzen),
- höhere Knotendichte an krumlinigen bzw. krumflächigen Gebietsgrenzen (hinreichend genaue Approximation durch Polygon bzw. Polyeder).

Die **Vernetzung** (d.h. die Wahl der Zerlegung) kann manuell oder durch Algorithmen erfolgen. Ein verbreiteter Algorithmus ist die Delaunay-Triangulation.

Gebietszerlegungen können besser oder schlechter für FEM geeignet sein als andere. Ins-

besondere sollte die Form der Teilmengen hinreichend einheitlich sein. Diese Forderung kann wie folgt präzisiert werden:

Merke

Für jedes Teilgebiet T_l seien $r_l > 0$ der Radius eines größten eingeschlossenen Kreises und $R_l > 0$ der Radius des kleinsten umschließenden Kreises. Die Zerlegung $T_1, \ldots, T_N$ heißt **isotrop**, wenn es eine Konstante $c > 0$ gibt, sodass

$$\frac{R_l}{r_l} \leq c \quad \text{für } l = 1, \ldots, N \tag{7.135}$$

gilt.

Neben Algorithmen zur Erzeugung von Vernetzungen existieren auch Algorithmen zur Verbesserung von Netzen im Sinne der Isotropie (**Netzglättung**).

Auch können gegebene Netze weiter verfeinert werden durch weiteres Unterteilen der vorhandenen Teilgebiete (**Netzverfeinerung**). Insbesondere die adaptive Netzverfeinerung ist praktisch von Bedeutung: In Regionen, in denen eine vorherige Netzverfeinerung zu einer deutlichen Änderung der Lösung geführt hat, wird weiter verfeinert. In Regionen ohne nennenswerte Änderung ist keine weitere Verfeinerung nötig. Durch dieses Vorgehen können Rechenzeit und Speicherplatz gespart werden bei gleichzeitig kleinem Diskretisierungsfehler.

7.6.4 Aufstellen der Steifigkeitsmatrix

Das Aufsteller der Steifigkeitsmatrix ist ein sehr aufwendiger Vorgang, der üblicherweise durch spezielle Software erledigt wird. Insbesondere müssen die in der schwachen Formulierung der PDE auftretenden Integrale (Bereichsintegrale, Oberflächenintegrale, Kurvenintegrale) über den Teilgebieten T_l und deren Randkanten numerisch berechnet werden.

Das konkrete Vorgehen hängt stark von der betrachteten Gleichung und den Anfangs- bzw. Randbedingungen ab. Verzichten an dieser Stelle auf die Details und verweisen auf entsprechende Veranstaltungen im weiteren Studienverlauf.

Im Python-Ökosystem ist FEniCS ein mächtiges und weit verbreitetes Paket für die Umsetzung von FEM. CAM-Software bietet meist auch eine Möglichkeit für die Umsetzung einfacher FEM, z.B. FreeCAD mit dem FEM Workbench.

7.6.5 Aufstellen der rechten Seite

Das Aufstellen der rechten Seite $\underline{b}$ im zu lösenden Gleichungssystem erfolgt analog zur Systemmatrix durch numerische Berechnung der in der Linearform b enthaltenen Inte-

grale über jedem Teilgebiet T_l und ggf. Randkanten.

8 Übung 1 (Python und Jupyter)

8.1 Jupyter

8.1.1 Aufgabe 1.1

Starten Sie durch Klick auf den Link ein browser-basiertes JupyterLab

Erstellen Sie ein neues Dokument ("Notebook"), berechnen Sie darin `123 * 456` und speichern Sie das Dokument.

Das gespeicherte Dokument liegt nun im internen Speicherbereich Ihres Browsers. Um es bei Ihren Dateien zu speichern, können Sie den Download-Button verwenden.

8.1.2 Aufgabe 1.2

Starten Sie ein JupyterLab auf mybinder.org. Laden Sie die in der vorhergehenden Aufgabe gespeicherte Datei dort hoch und öffnen Sie sie.

8.1.3 Aufgabe 1.3

Machen Sie sich mit Jupyter-Notebooks vertraut. Fügen Sie neue Zellen ein. Probieren Sie auch den Zelltyp "Markdown" aus für Text, der nicht von Python verarbeitet werden soll.

8.1.4 Aufgabe 1.4 (optional)

Testen Sie weitere Möglichkeiten zur Jupyter-Nutzung:

- JupyterLab der Fakultät Informatik/Mathematik.
- JupyterLab lokal installieren (siehe z.B. Install Jupyter Locally in Data Science and Artificial Intelligence for Undergraduates),

8.2 Python

8.2.1 Aufgabe 1.5

Werten Sie das Polynom $x^3 + 2\,x^2 + 1$ an den Stellen 1, 3, 4, 6, 8 aus. Ergänzen Sie dazu den folgenden Code:

```
stellen = [1, 3, 4, 6, 8]
for x in stellen:
    # hier ergänzen
```

8.2.2 Aufgabe 1.6

Geben Sie alle durch 4 teilbaren Zahlen zwischen 9 und 99 aus. Verwenden Sie dazu eine `for`-Schleife.

```
# Ihre Lösung
```

8.2.3 Aufgabe 1.7

Zählen Sie die geraden Zahlen in einer gegebenen Liste.

```
liste = [2, 5, 4, 3, 3, 2, 0, 6, 7, 5, 6, 7]
# weitere Listen zum Testen:
#liste = [2, 4]
#liste = [1, 3]
#liste = []

# Ihre Lösung
```

9 Übung 2 (NumPy)

9.1 Arrays erzeugen

9.1.1 Aufgabe 2.1

Erzeugen Sie die Matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{9.1}$$

als NumPy-Array.

```
# Ihre Lösung
```

9.1.2 Aufgabe 2.2

Erzeugen Sie für verschiedene $n \geq 3$ die $(n \times n)$-Matrix

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & 0 & \cdots & 0 & 1 \\ 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{9.2}$$

als NumPy-Array.

```
n = 10

# Ihre Lösung
```

9.2 Arrays verändern

9.2.1 Aufgabe 2.3

Setzen Sie alle negativen Einträge in einem gegebenem NumPy-Array auf Null.

```
A = np.array([[ -2, 4, 3], [2, -4, -5], [3, 4, -3]])

# Ihre Lösung
```

9.2.2 Aufgabe 2.4

Vertauschen Sie in einem gegebenem Array mit gerader Zeilen- und Spaltenanzahl das linke obere Viertel mit dem unteren rechten Viertel. Beispiel:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad \text{wird zu} \quad \begin{bmatrix} 11 & 12 & 3 & 4 \\ 15 & 16 & 7 & 8 \\ 9 & 10 & 1 & 2 \\ 13 & 14 & 5 & 6 \end{bmatrix} \tag{9.3}$$

```
A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15,
↪  16]])

# Ihre Lösung
```

9.3 Lineare Algebra

9.3.1 Aufgabe 2.5

Berechnen Sie zu gegebenen Matrizen $A, B \in \mathbb{R}^{3\times3}$ den Ausdruck

$$A^{\mathrm{T}}\,A + B^{\mathrm{T}}\,B. \tag{9.4}$$

```
A = np.array([[1, 2, 3], [5, 6, 7], [9, 10, 11]])
B = np.array([[0, 2, 1], [3, 6, 5], [9, 8, 7]])

# Ihre Lösung
```

9.3.2 Aufgabe 2.6

Berechnen Sie die Inverse zu einer gegebenen Matrix $A \in \mathbb{R}^{3\times3}$ und lösen Sie damit das lineare Gleichungssystem

$$A\,x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}. \tag{9.5}$$

Lösen Sie das Gleichungssystem auch direkt mit `np.linalg.solve` und vergleichen sie die Ergebnisse.

```
A = np.array([[1, 2, 3], [4, 5, 5], [7, 7, 9]])

# Ihre Lösung
```

10 Übung 3 (Zahlen)

10.1 Gleitkommazahlen

Bezeichnen im Folgenden mit "float8" binäre Gleitkommazahlen mit Mantissenlänge 5 Bit (einschließlich Hidden-Bit) und Exponentenlänge 3 Bit. Das erste Bit codiert das Vorzeichen, dann folgt der Exponent, dann die Mantisse. Das Bitfeld SEEEMMMM kodiert also die Zahl

$$(-1)^S \, (2^{t-1} + M) \cdot 2^{E-(2^{p-1}-1)-(t-1)}, \tag{10.1}$$

wobei $p = 3$ und $t = 5$.

10.1.1 Aufgabe 3.1 (KTA)

Welcher Dezimalzahl entsprechen die folgenden binär dargestellten float8-Zahlen?

- **(a)** 00110000
- **(b)** 00111000
- **(c)** 00010000

10.1.2 Aufgabe 3.2 (KTA)

Finden Sie die Dezimaldarstellung für folgende normalisierte float8-Zahlen:

- **(a)** die kleinste positive,
- **(b)** die größte positive.

10.1.3 Aufgabe 3.3

Welche float8-Zahl entspricht am besten den folgenden Dezimalzahlen (im Sinne der nach IEEE 754 üblichen Rundungsregel)?

- **(a)** 2
- **(b)** 2.0625
- **(c)** 2.1
- **(d)** 2.1875

10.2 Patriot-Beispiel

10.2.1 Aufgabe 3.4 (KTA)

Erklären Sie die Unterschiede in den sechs Ausgaben des folgenden Codes (vgl. Patriot-Beispiel).

```
import numpy as np
import sympy

print('float16')
a = np.float16(3600000)
b = np.float16(0.1)
print(f'a = {a:.70f}')
print(f'b = {b:.70f}')
print(f'a * b = {a * b:.70f}')
ab_exakt = sympy.N(f'{a:.70f} * {b:.70f}', 70)
print(f'a * b = {ab_exakt} (exakt)')
print()

print('float32')
a = np.float32(3600000)
b = np.float32(0.1)
print(f'a = {a:.70f}')
print(f'b = {b:.70f}')
print(f'a * b = {a * b:.70f}')
ab_exakt = sympy.N(f'{a:.70f} * {b:.70f}', 70)
print(f'a * b = {ab_exakt} (exakt)')
print()

print('float64')
a = np.float64(3600000)
b = np.float64(0.1)
print(f'a = {a:.70f}')
print(f'b = {b:.70f}')
print(f'a * b = {a * b:.70f}')
ab_exakt = sympy.N(f'{a:.70f} * {b:.70f}', 70)
print(f'a * b = {ab_exakt} (exakt)')
print()
```

```
float16
a = inf
b =
↪  0.0999755859375000000000000000000000000000000000000000000000000000000000000000000000
a * b = inf
```

```
a * b = 0.0999755859375*inf (exakt)

float32
a =
↪  3600000.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
b =
↪  0.100000001490116119384765625000000000000000000000000000000000000000000000000000000000000000000000000000
a * b =
↪  360000.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a * b =
↪  360000.005364418029785156250000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  (exakt)

float64
a =
↪  3600000.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
b =
↪  0.100000000000000005551115123125782702118158340454101562500000000000000000000000000000000000000000000000000
a * b =
↪  360000.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a * b =
↪  360000.000000000019984014443252817727625370025634765625000000000000000000000000000000000000000000000000000
↪  (exakt)
```

```
/tmp/ipykernel_957757/2967486232.py:5: RuntimeWarning: overflow
↪  encountered in cast
  a = np.float16(3600000)
```

10.2.2 Aufgabe 3.5 (KTA)

Wie könnte man beim numerischen Berechnen von $0.1 \cdot 3600000$ mit dem Computer garantiert sämtliche Rundungsfehler vermeiden?

11 Übung 4 (Fehler)

11.1 Dezimale Gleitkommazahlen

Wir rechnen mit dezimalen Gleitkommazahlen $m\,10^e$, wobei m höchstens 3 Stellen hat und $e \geq 0$ gelten soll. Gerundet wird stets mathematisch.

11.1.1 Aufgabe 4.1 (KTA)

Geben Sie folgende dezimale Gleitkommazahlen nach obigem Schema an:

- **(a)** kleinste,
- **(b)** zweitkleinste,
- **(c)** größte vierstellige Zahl.

11.1.2 Aufgabe 4.2 (KTA)

Berechnen Sie die folgenden Ausdrücke indem Sie vor und nach jeder Rechenoperation runden. Vergleichen Sie mit dem exakten Ergebnis und begründen Sie den Unterschied.

- **(a)** $100.5 - 0.4$
- **(b)** $210.51 - 209.49$
- **(c)** $((1000 + 4) + 4) + 4$
- **(d)** $1000 + ((4 + 4) + 4)$

11.2 Approximation von π

Mit jeder der beiden rekursiv definierten Zahlenfolgen

$$a_0 := \frac{1}{\sqrt{3}}, \qquad a_{i+1} := \frac{\sqrt{a_i^2 + 1} - 1}{a_i} \quad \text{für } i = 1, 2, \ldots \tag{11.1}$$

und

$$b_0 := \frac{1}{\sqrt{3}}, \qquad b_{i+1} := \frac{b_i}{\sqrt{b_i^2 + 1} + 1} \quad \text{für } i = 1, 2, \ldots \tag{11.2}$$

kann π aufgrund

$$\pi = 6 \lim_{i \to \infty} 2^i\, a_i \qquad \text{bzw.} \qquad \pi = 6 \lim_{i \to \infty} 2^i\, b_i \tag{11.3}$$

beliebig genau approximiert werden.

11.2.1 Aufgabe 4.3 (KTA)

Überzeugen Sie sich zunächst, dass $a_i = b_i$ für alle i gilt.

11.2.2 Aufgabe 4.4 (KTA)

Untersuchen Sie die Kondition für die Berechnung von a_{i+1} aus a_i. Ist für die Kondition bei der alternativen Berechnung ein anderes Ergebnis zu erwarten?

11.2.3 Aufgabe 4.5 (KTA)

Untersuchen Sie die Stabilität beim Berechnen von a_{i+1} und b_{i+1} aus a_i bzw. b_i mit dem jeweils direkt aus der Formel ableitbaren Algorithmus.

11.2.4 Aufgabe 4.6

Implementieren Sie beide Algorithmen aus der vorhergehenden Aufgabe. Vergleichen Sie die erzielten Approximationen für π mit dem genauen Wert

$$\pi = 3.14159265358979323846264338\ldots. \tag{11.4}$$

Passen diese Ergebnisse zu den Stabilitätsaussagen aus der vorhergehenden Aufgabe?

```
# Ihre Lösung
```

12 Übung 5 (numerische Differentiation)

In dieser Übung werden wir Schritt für Schritt die Momentangeschwindigkeit eines Fahrrads aus einem aufgezeichneten GPS-Track berechnen.

12.1 Daten lesen und prüfen

12.1.1 Aufgabe 5.1

Lesen Sie die CSV-Datei track.csv mit dem unten gegebenen Programmcode ein. Die Datei enthält eine Tabelle mit drei Spalten:

- vergangene Zeit seit Messbeginn in Sekunden,
- zurückgelegte Strecke seit Messbeginn in Meter,
- Höhe des Messpunktes ausgehend vom Startpunkt in Meter.

Jede Zeile entspricht einem Messpunkt entlang der Strecke. Die CSV-Datei ist auf folgende Weise entstanden:

1. GPS-Track mittels OsmAnd aufzeichnen. Einstellung: **alle 2 Sekunden ein Messpunkt**.
2. Messpunkte aus GPX-Datei auf Straßennetz projizieren zur Verbesserung der Genauigkeit (z.B. mit Valhalla).
3. Abstand der Messpunkte bestimmen. Dies liefert die bis zum jeweiligen Messpunkt gefahrene Strecke.
4. Zeiten und Höhen so verschieben, dass die erste Messung bei Sekunde 0 und Höhe 0 erfolgt.

```
import csv
import numpy as np

path = ''  # hier Pfad zur runtergeladenen Datei eintragen

# leere Listen für Aufnahme der Messwerte
times = []
dists = []
alts = []

# Datei öffnen
with open(path) as csvfile:
    reader = csv.reader(csvfile)

    # Zeile für Zeile aus Datei lesen
```

```
    for row in reader:
        if row[0] == 'time':  # Kopfzeile überspringen
            continue

        # Zeichenketten in Zahlen umwandeln und an Listen anhängen
        times.append(float(row[0]))
        dists.append(float(row[1]))
        alts.append(float(row[2]))

# Listen in NumPy -Arrays umwandeln
times = np.array(times)
dists = np.array(dists)
alts = np.array(alts)
```

12.1.2 Aufgabe 5.2

Prüfen Sie, dass die gefahrene Strecke von Punkt zu Punkt anwächst, also nicht rückwärts gefahren wurde.

```
# Ihre Lösung
```

12.1.3 Aufgabe 5.3

Prüfen Sie, ob die Zeitabstände zwischen zwei benachbarten Punkten tatsächlich 2 Sekunden betragen.

```
# Ihre Lösung
```

12.2 Preprocessing

12.2.1 Aufgabe 5.4

Für die Berechnung der Geschwindigkeit erscheint es sinnvoll, die tatsächlich gefahrene Strecke im Raum zu nutzen statt die Strecke in der Ebene. Berechnen Sie die zurückgelegte Strecke im Raum an jedem Punkt.

```
# Ihre Lösung
```

12.2.2 Aufgabe 5.5

Wie groß ist die Abweichung zwischen 2D- und 3D-Strecke? Ist diese Abweichung relevant angesichts eines Messfehlers von 2 Meter und mehr bei GPS-Messungen?

```
# Ihre Lösung
```

12.3 Numerische Differentiation

12.3.1 Aufgabe 5.6

Berechnen Sie die Momentangeschwindigkeit als Ableitung der Zeit-Weg-Funktion, welche nur an endlich vielen Punkten bekannt ist. Verwenden Sie dazu vier verschiedene Ansätze:

- Vorwärtsdifferenzen,
- Rückwärtsdifferenzen,
- zentrale Differenzen an den Messpunkten,
- zentrale Differenzen zwischen den Messpunkten.

Stellen Sie alle vier Geschwindigkeitsverläufe grafisch dar (in km/h), indem Sie folgenden Programmcode anpassen:

```
import matplotlib.pyplot as plt

# Plot erstellen
fig, ax = plt.subplots(figsize=(14, 5))

# Linie einzeichnen (ggf. mehrfach verwenden)
ax.plot(1d_array_x, 1d_array_y, ' -', label='Text für Legende')

# Legende einfügen
ax.legend()

# Achsenbeschriftungen
ax.set_xlabel('Text')
ax.set_ylabel('Text')

# Plot anzeigen
plt.show()
```

```
# Ihre Lösung
```

12.3.2 Aufgabe 5.7

Berechnen Sie den Bereich möglicher Geschwindigkeiten, wenn das Fahrrad für 2 Sekunden konstant 30 km/h fährt und der Messfehler bei der Positionsbestimmung höchstens 2 Meter beträgt.

Passt dieser Bereich zu den Schwankungen in den berechneten Geschwindigkeitsverläufen?

```
# Ihre Lösung
```

13 Übung 6 (nichtlineare Gleichungen)

13.1 Kovergenz des Newton-Verfahrens

13.1.1 Aufgabe 6.1 (KTA)

Führen Sie zwei Schritte des Newton-Verfahrens für die Funktion

$$f(x) = x^3 - 2\,x + 2 \tag{13.1}$$

mit dem Startwert $x_0 = 1$ manuell aus. Ist bei Fortführung der Iteration mit Konvergenz zu rechnen?

13.1.2 Aufgabe 6.2 (KTA)

Führen Sie zwei Schritte des Newton-Verfahrens für die Funktion

$$f(x) = \frac{x}{x^2 + 2} \tag{13.2}$$

mit dem Startwert $x_0 = 2$ manuell aus. Ist bei Fortführung der Iteration mit Konvergenz zu rechnen? Was passiert bei $x_0 \approx \sqrt{2}$?

13.1.3 Aufgabe 6.3 (KTA)

Führen Sie einen Schritt des Newton-Verfahrens für die Funktion

$$f(x) = \sqrt{25 - x^2} - 1 \tag{13.3}$$

mit dem Startwert $x_0 = 3$ manuell aus. Sind weitere Schritte möglich?

13.2 Nichtlineare Gleichungen mit SciPy

13.2.1 Aufgabe 6.4

Lösen Sie die Gleichung

$$x + \sin x = -1 \tag{13.4}$$

mittels `scipy.optimize.bisect` und `scipy.optimize.newton`. Welche Ursachen können zu Abweichungen zwischen den beiden Lösungen führen?

```
# Ihre Lösung
```

13.2.2 Aufgabe 6.5

Lösen Sie das Gleichungssystem

$$x^3 - 3\,x\,y^2 = 1$$
$$x^2\,y - y^3 = 0$$

auf folgende Weisen:

- **(a)** per Hand,
- **(b)** symbolisch mit SymPy,
- **(c)** numerisch mit `scipy.optimize.root(method='broyden1')` (wählen Sie mehrere verschiedene Startpunkte).

Zusatz: Finden Sie heraus, was die Funktion in (c) tut.

```
# Ihre Lösung
```

14 Übung 7

14.1 Kondition einer Matrix

14.1.1 Aufgabe 7.1

Berechnen Sie die Normen $\|A\|_1$, $\|A\|_2$ und $\|A\|_\infty$ sowie die Konditionszahlen $\kappa_1(A)$, $\kappa_2(A)$ und $\kappa_\infty(A)$ mit `numpy.linalg.norm` bzw. `numpy.linalg.cond` für die Matrizen

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \tag{14.1}$$

$$A = \begin{bmatrix} 0.99 & 1 \\ 1 & 1 \end{bmatrix} \tag{14.2}$$

und

$$A = \begin{bmatrix} 1 & 0.99 & 0 \\ 1 & 0.01 & 2 \\ 0.5 & 0 & 1 \end{bmatrix}. \tag{14.3}$$

Ist die Konditionszahl immer groß, wenn die Matrixspalten fast linear abhängig sind?

```
# Ihre Lösung
```

14.1.2 Aufgabe 7.2 (KTA)

Die Konditionzahl der Matrix A sei $\kappa_2(A) = 20$. Wie groß darf der relative Fehler in b beim exakten Lösen von $A\,x = b$ sein, damit der relative Lösungsfehler höchstens 1 Prozent beträgt?

Ist diese Fehlerschranke erfüllt, wenn statt der exakten rechten Seite $b = [2, 1, 2]^{\mathrm{T}}$ die fehlerbehaftete rechte Seite $\tilde{b} = [2.03, 0.96, 2]^{\mathrm{T}}$ verwendet wird?

14.2 Lösungsverfahren

14.2.1 Aufgabe 7.3

Lösen Sie das Gleichungssystem

$$\begin{aligned} 50\,x_1 + 20\,x_3 &= 2 \\ 2\,x_1 + x_2 + x_3 &= 0 \\ x_1 + 2\,x_2 + 3\,x_3 &= 1 \end{aligned} \tag{14.4}$$

mit `numpy.linalg.solve`.

Welches Lösungsverfahren kommt hier zum Einsatz?

```
# Ihre Lösung
```

14.2.2 Aufgabe 7.4 (optional)

Führen Sie für das Gleichungssystem in der vorhergehenden Aufgabe eine Äquilibrierung (Vorkonditionierung) durch. Lösen Sie das System dann nochmals und vergleichen Sie die Lösung mit der zuvor erhaltenen.

Vergleichen Sie auch die Konditionszahlen vor und nach der Äquilibrierung.

```
# Ihre Lösung
```

15 Übung 8 (Differential- und Integralrechnung)

15.1 Aufgabe 8.1

Berechnen Sie die Rotation des durch

$$f(x, y, z) = (x^2 + y\,z + z,\ y + x\,z,\ x\,y\,z). \tag{15.1}$$

gegebenen Vektorfeldes f.

Wie bewegt sich ein (kleines und leichtes) Objekt, welches im Punkt $(1, 0, 0)$ in die Strömung gegeben wird (Bewegungsrichtung, Drehachse)?

15.2 Aufgabe 8.2

Die durch

$$x = 0, \qquad y = 0, \qquad x + y = 4 \tag{15.2}$$

beschriebenen drei Flächen begrenzen ein (unendlich langes) dreiseitiges Prisma im $\mathbb{R}^3$. Aus diesem wird durch Schnitte entlang der beiden durch

$$z = 0, \qquad z = x^2 + y^2 \tag{15.3}$$

beschriebenen Flächen ein endliches Stück herausgeschnitten. Geben Sie eine Formel für die Berechnung des Volumens dieses Stücks an, welche nur gewöhnliche eindimensionale Integrale enthält.

15.3 Aufgabe 8.3

Geben Sie eine Formel für das Berechnen der Länge des zwischen $(0, 0)$ und $(1, 1)$ liegenden Stücks K der Parabel $y = x^2$ an, die nur gewöhnliche eindimensionale Integrale enthält.

15.4 Aufgabe 8.4

Das durch

$$f(x, y) = \begin{bmatrix} x^2 + y^2 \\ x\,y \end{bmatrix} \tag{15.4}$$

gegebene Kraftfeld wird entlang der Strecke von $(0, 0)$ nach $(2, 2)$ durchquert. Gegen Sie eine Formel für die dabei verrichtete Arbeit an, die nur gewöhnliche eindimensionale Integrale enthält.

15.5 Aufgabe 8.5

Beschreibe

$$f(x, y) = \begin{bmatrix} 2\,x + y \\ x + 2\,y \end{bmatrix} \tag{15.5}$$

eine Strömung in der Ebene. Geben Sie eine Formel für die verrichtete Arbeit beim Durchqueren der Strömung entlang des Viertelkreises um den Ursprung von $(2, 0)$ nach $(0, 2)$ an, die nur gewöhnliche eindimensionale Integrale enthält.

15.6 Aufgabe Ü8.6

Ein Stück parabelförmige Rinne besitzt den Querschnitt $z = x^2$, $x \in [-1, 1]$, verläuft entlang der y-Achse und ist 10 Einheiten lang. Im Scheitelpunkt liegt das Flächengewicht des verwendeten Blechs bei 0.2 Masseneinheiten pro Flächeneinheit und fällt dann linear mit der z-Koordinate auf 0.1 an den Rinnenrändern ab. Geben Sie eine Formel für die Berechnung der Masse der Rinne an, die nur gewöhnliche eindimensionale Integrale enthält.

15.7 Aufgabe Ü8.7

Ein gebogenes Drahtgitter habe die gleiche Form wie die Rinne in Aufgabe 8.6. Das Gitter wird in einer Wasserströmung platziert, welche durch das Vektorfeld

$$f(x, y, z) = \begin{bmatrix} x^2 + y4 \\ 3\,x\,y \\ 2\,x\,z + z^2 \end{bmatrix} \tag{15.6}$$

gegeben ist (Richtung und Geschwindigkeit). Geben Sie eine Formel für das pro Zeiteinheit durch das Gitter strömende Wasservolumen an, die nur gewöhnliche eindimensionale Integrale enthält.

15.8 Aufgabe Ü8.8

Überprüfen Sie den Integralsatz von Gauß für das Vektorfeld aus Aufgabe 8.1. Verwenden Sie als Gebiet B das Zwischen den folgenden vier Flächen liegende Volumen:

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

Verwenden Sie für die Berechnung der Integrale SymPy.

15.9 Aufgabe Ü8.9 (optional)

Berechnen Sie die Integrale in Aufgabe 8.8 numerisch mit SciPy. Hinweise und Beispiele dazu finden Sie in der Dokumentation von SciPy.

```
# Ihre Lösung
```

16 Übung 9 (Fourier-Transformation)

16.1 Glockenkurve

16.1.1 Aufgabe 9.1 (KTA)

Bestimmen Sie die Fourier-Transformierte der allgemeinen Glockenkurve

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) = \mathrm{e}^{-\frac{(x-\mu)^2}{2\,\sigma^2}}, \quad \mu \in \mathbb{R}, \quad \sigma > 0, \tag{16.1}$$

unter Anwendung der Eigenschaften der Fourier-Transformation!

16.1.2 Aufgabe 9.2 (KTA)

Ermitteln Sie die Fourier-Transformierte der Ableitung der allgemeinen Glockenkurve aus der vorhergehenden Aufgabe!

16.2 Sägezahnkurve

16.2.1 Aufgabe 9.3

Skizzieren Sie die Funktion

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) = \begin{cases} x, & \text{für } x \in [0, 2\,\pi), \\ 0, & \text{sonst} \end{cases}. \tag{16.2}$$

Geben Sie das Integral zur Berechnung der Fourier-Transformierten an und ermitteln Sie die Fourier-Transformierte durch symbolische Integration mit SymPy. Stellen Sie die Fourier-Transformierte sowei deren Real- und Imaginärteil grafisch dar.

```
# Ihre Lösung
```

16.2.2 Aufgabe 9.4

Skizzieren Sie die $2\,\pi$-periodische Funktion

$$g : \mathbb{R} \to \mathbb{R}, \quad f(x) = x \text{ für } x \in [0, 2\,\pi). \tag{16.3}$$

Geben Sie das Integral zur Berechnung der Fourier-Transformierten an und ermitteln Sie die Fourier-Transformierte durch symbolische Integration mit SymPy. Stellen Sie den Imaginärteil der Fourier-Transformierten grafisch dar.

```
# Ihre Lösung
```

16.2.3 Aufgabe 9.5 (KTA)

Zeigen Sie, dass die abgebrochene Fourier-Reihe

$$x \mapsto \sum_{k=-n}^{n} c_k \, \mathrm{e}^{\mathrm{i}\,k\,x} \tag{16.4}$$

stets reellwertig ist. Berechnen Sie dazu die Summen $c_{-k} + c_k$ für $k \geq 1$.

16.2.4 Aufgabe 9.6

Stellen Sie die abgebrochenen Fourier-Reihen aus der vorhergehenden Aufgabe für $n = 0, 1 \ldots, 5$ und $x \in [-4\,\pi, 4\,\pi]$ grafisch dar.

```
# Ihre Lösung
```

16.2.5 Aufgabe 9.7

Werten Sie die periodische Funktion aus den vorhergenden Aufgaben an $N \in \mathbb{N}$ gleichmäßig im Intervall $[0, 2\,\pi)$ verteilten Stützstellen aus. Führen Sie für die so erhaltene endliche Zahlenfolge eine diskrete Fourier-Transformation durch (nutzen Sie dafür `numpy.fft.fft`). Stellen Sie Real- und Imaginärteil der Fourier-Transformierten grafisch dar.

```
# Ihre Lösung
```

16.2.6 Aufgabe 9.8

Vergleich Sie die Fourier-Komponenten $c_{-n}, \ldots, c_n$ aus Aufgabe 9.4 mit der diskreten Fourier-Transformierten aus Aufgabe 9.8 bei $N = 2\,n + 1$. Achten Sie insbesondere auf die Skalierung und auf die von `numpy.fft.fft` verwendete Reihenfolge der Fourier-Komponenten.

Die Dokumentation des `numpy.fft`-Moduls und die Funktion numpy.fft.fftfreq helfen hier eventuell weiter.

```
# Ihre Lösung
```

17 Übung 10 (gewöhnliche Differentialgleichungen)

Aus einem homogenen Material der Dichte $\varrho > 0$ soll eine Säule der Höhe $H > 0$ mit kreisrundem, sich noch oben verjüngenden Querschnitt hergestellt werden. Zu jeder Höhe $h \in [0, H]$ liegt also ein anderer Radius $r(h) > 0$ vor. Auf der Oberseite der Säule ruht eine gleichmäßig verteilte Masse $M > 0$. Die Radien sollen gerade so gewählt werden, dass der Druck (Kraft pro Flächeneinheit) auf die Querschnittsflächen in jeder Höhe gleich ist und höchsten den Wert $P > 0$ annimmt.

17.1 Aufgabe 10.1

Modellieren Sie den Verlauf der Funktion $r : [0, H] \to [0, \infty)$ als Lösung einer gewöhnlichen Differentialgleichung. Drücken Sie dazu zunächst die Querschnittsfläche der Säule, die auf einer Querschnittsfläche aufliegende Masse und den auf eine Querschnittsfläche wirkenden Druck in Abhängigkeit von der Höhe $h \in [0, H]$ aus. Zur Vereinfachung der Formeln ist es zweckmäßig, das Argument h als Abstand zur Oberseite der Säule zu interpretieren. Der Wert h entspricht dann also der Höhe $H - h$.

Geben Sie auch die passende Anfangsbedingung an.

17.2 Aufgabe 10.2

Verschaffen Sie sich einen ersten Eindruck von der Form der Säule, indem Sie das Richtungsfeld der Differentialgleichung grafisch darstellen. Prüfen Sie die Ergebnisse auf Plausibilität!

```
# Ihre Lösung
```

17.3 Aufgabe 10.3

Lösen Sie die Differentialgleichung numerisch (z.B. mit SciPy).

Stellen Sie den Schnitt entlang der Säulenachse grafisch dar.

```
# Ihre Lösung
```

17.4 Aufgabe 10.4

Betrachten Sie Lösungen für verschiedene Werte der Parameter M, φ, P. Sieht die Lösungen plausibel aus?

18 Übung 11 (Systeme von Differentialgleichungen)

Betrachten die nicht linearisierte Pendelgleichung

$$\varphi''(t) + \frac{g}{l} \sin \varphi(t) = 0, \qquad t > 0, \qquad \varphi(0) = \varphi_0, \qquad \varphi'(0) = 0, \tag{18.1}$$

wobei $\varphi(t)$ die Auslenkung eines Pendels der Länge l zum Zeitpunkt t ist (vgl. Modellierung Pendel).

18.1 Aufgabe 11.1 (KTA)

Formulieren Sie die Pendelgleichung als System von Differentialgleichungen erster Ordnung einschließlich Anfangsbedingung.

18.2 Aufgabe 11.2

Lösen Sie das System aus der vorhergehenden Aufgabe numerisch.

```
# Ihre Lösung
```

18.3 Aufgabe 11.3

Vergleichen Sie die Lösung zur nicht linearisierten Pendelgleichung mit der Lösung

$$\psi(t) := \varphi_0 \cos\left(\sqrt{\frac{g}{l}}\, t\right), \qquad t \geq 0, \tag{18.2}$$

der linearisierten Gleichung für kleine und große Anfangsauslenkungen.

```
# Ihre Lösung
```

18.4 Aufgabe 11.4 (optional)

Versuch Sie, die nicht linearisierte Pendelgleichung mit SymPy symbolisch zu lösen. Hinweise und Beispiele dazu finden Sie in der SymPy-Dokumentation.

```
# Ihre Lösung
```

19 Übung 12 (PDE, FDM)

19.1 Grundlagen

19.1.1 Aufgabe 12.1 (KTA)

Entscheiden Sie, ob es sich bei folgende PDE um eine Kontinuitätsgleichung, einen Diffusionsgleichung, eine Wellengleichung oder eine Poisson-Gleichung handelt.

- **(a)** $u_{xx}(x,y,t) + u_{yy}(x,y,t) + u_{tt}(x,y,t) + 5\,t = 0,$
- **(b)** $\mathrm{div}\nabla u(x,y,z,t) = 5\,u_t(x,y,z,t).$

19.1.2 Aufgabe 12.2 (KTA)

Formulieren Sie den folgenden Sachverhalt mit Hilfe einer PDE: Ein zylindrisches Bauteil wurde durchgängig auf eine Temperatur von 1000°C gebracht und soll nun abgekühlt werden. Dazu wird es in eine Kühlmittelströmung gebracht, die die Oberflächentemperatur des Bauteils konstant bei 50°C hält. Wie lange dauert es bis die Bauteiltemperatiur an jeder Stelle unter 100°C gesunken ist?

19.2 FDM für Poisson-Gleichung

Wir setzen das Lösen der Section 7.2.4 für das Gebiet

$$B := [0,1] \times [0,2] \tag{19.1}$$

am Computer um.

19.2.1 Aufgabe 12.3

Stellen Sie die Systemmatrix des zu lösenden Gleichungssystems für die grobe Diskretisierung mit $n_x = 4$, $n_y = 5$ manuell auf.

19.2.2 Aufgabe 12.4

Implementieren Sie das Lösen der Poisson-Gleichung. Testen Sie Ihren Code zunächst mit der rechten Seite $f \equiv 1$. Stellen Sie die Lösung grafisch dar. Ergänzen Sie dazu folgenden Programmcode:

```
import numpy as np
import plotly.graph_objects as go
```

```
a = 1
b = 2
nx = a * 10
ny = b * 10
dx = a / nx
dy = b / ny

# rechte Seite
f_cont = lambda x, y: np.ones(x.shape, dtype=float)

# Koordinaten der Gitterpunkte
grid_x, grid_y = np.meshgrid(
    np.linspace(0, a, nx + 1),
    np.linspace(0, b, ny + 1)
)

# Systemmatrix (zunächst überall Nullen)
A = np.zeros(((nx - 1) * (ny - 1), (nx - 1) * (ny - 1)))

# 2d -Index (i, j) auf 1d -Index abbilden (i=1,...,nx -1, j=1,...,ny
↪  -1)
k = lambda i, j: (j - 1) * (nx - 1) + i - 1

# Systemmatrix befüllen
# TODO

# diskretisierte rechte Seite
f = f_cont(grid_x[1: -1, 1: -1], grid_y[1: -1]).reshape( -1)

# LGS lösen
u_inner = np.linalg.solve(A, f).reshape(ny - 1, nx - 1)

# Lösung des LGS mit Randbedingung kombinieren
u = np.zeros((ny + 1, nx + 1), dtype=float)
# TODO

# Lösung darstellen
fig = go.Figure()
fig.layout.width = 800
fig.layout.height = 600
fig.layout.scene.aspectmode = 'data'
fig.add_trace(
    go.Surface(x=grid_x, y=grid_y, z=u, colorscale='Jet')
)
```

```
fig.show()
```

19.2.3 Aufgaben 12.5

Stellen Sie eine Vermutung über das Aussehen der Lösung an, wenn die rechte Seite f nur auf einem rechteckigen, achsenparallelen Teilgebiet den Wert 1 hat und ansonsten Null ist. Verwenden Sie als Ausgangspunkt Ihrer Überlegungen die Interpretation der Lösung als Verformung einer Membran unter der Last f.

Lösen Sie die Poisson-Gleichung mit einer solchen rechten Seite. Vergleichen Sie die Lösung mit Ihrer Vermutung. Begründen Sie eventuelle Abweichungen zu Ihrer Vermutung. Beispielcode zum Erzeugen von f:

```
def f_cont(x, y):
    f = np.zeros(x.shape, dtype=float)
    f[(x.shape[0] // 4):(2 * x.shape[0] // 4), (x.shape[1] // 4):(2 *
    ↪  x.shape[1] // 4)] = 5
    return f
```

```
# Ihre Lösung
```

19.2.4 Aufgaben 12.6

Wie könnte man f wählen um die Verformung einer Membran beim Belasten mit einem starren Quader wenigstens grob zu simulieren?

20 Übung 13 (PDE, FEM)

20.1 Aufgabe 13.1 (KTA)

Finden Sie die schwache Ableitung für die durch

$$f(x) = \begin{cases} x^2, & \text{für } x \leq 0, \\ x, & \text{für } x > 0 \end{cases} \tag{20.1}$$

auf dem Intervall $[-1, 1]$ gegebene Funktion f.

20.2 Aufgabe 13.2 (KTA)

Wir betrachten auf dem Gebiet $B = [0, 1]$ die Potential-Gleichung

$$u''(x) = \frac{1}{4} - \left(x - \frac{1}{2}\right)^2 \quad \text{für } x \in B \tag{20.2}$$

mit den Randbedingungen

$$u(0) = 0 \qquad \text{und} \qquad u'(1) = \frac{1}{4}. \tag{20.3}$$

Geben Sie eine schwache Formulierung dieses Problems an, die nur die erste schwache Ableitung von u benötigt!

20.3 Aufgabe 13.3

Zum Lösen des in der vorhergehenden Aufgabe formulierten Problems mittels FEM wählen wir Knoten an den Stellen $\frac{k}{n}$, $k = 0, 1, \ldots, n$ und stückweise lineare Basisfunktionen an jedem Knoten ("Hütchen" mit Breite $\frac{2}{n}$).

Geben Sie Formeln für die Einträge (Systemmatrix, rechte Seite) des zu lösenden FEM-Gleichungssystems an.

Geben Sie die Systemmatrix für $n = 5$ an.

21 Download

Zum Offline-Lesen steht das Vorlesungsskript als PDF-Datei zur Verfügung. Aus naheliegenden Gründen sind Animationen dort nicht enthalten. Generell ist die Konvertierung nach PDF noch etwas experimentell, sodass Darstellungsfehler nicht auszuschließen sind.

Download PDF-Datei

Aufgabenblätter für die Übung können Sie auch einzeln als PDF-Datei auf der entsprechenden Seite runterladen (Button oben rechts).

22 Organisatorisches

Modulbeschreibung in Modulux

22.1 Ablauf der Veranstaltung

- wöchentlich 1.5 Einheiten Vorlesung und eine Einheit Übung
- **Vorbereitung:** Skript vor Vorlesung und Übung mindestens überfliegen!
- **Nachbereitung:** Alles verstanden? Rückstände aus der Übung nachholen!

22.2 Übung

- Übung im Computer-Pool (oder eigener Computer)
- benötigte Software: Webbrowser (je nach Vorlieben eine lokale Python/Jupyter-Installation)

22.3 Prüfung

- schriftliche Klausur, 150 Minuten
- Beispielaufgaben sind bei den Übungsaufgaben gekennzeichnet
- keine Hilfsmittel außer Papier, Stifte, einfacher Taschenrechner (kein CAS, keine Grafik)
- Klausur auch ohne Taschenrechner gut lösbar
- Beispielklausur (dient nur zur groben Orientierung; grundsätzlich können alle behandelten Themen und Aufgabentypen Teil der Klausur sein)

22.4 Hinweise zum Skript

- Download als PDF-Datei möglich
- Aufgabenzettel für die Übung auch einzeln als PDF-Datei
- Aufgabenzettel für die Übung als Jupyter-Notebook (Lösen der Aufgaben direkt im Dokument)

Viele Rechnungen, Beweise, Herleitungen, Zeichnungen fehlen im Skript und werden an der Tafel geliefert. Deshalb:

- mitschreiben oder
- Fotos von der Tafel machen

Die Technik hinter dem Skript ist noch in Entwicklung und teils etwas “beta”. Hinweise zu Problemen und Verbesserungsvorschläge gern an den Vorlesenden!

22.5 Änderungshistorie

Da dieses Skript für das Wintersemester 2026/27 vollständig neu strukturiert und formuliert wurde, werden im Laufe des Semesters sehr wahrscheinlich Korrekturen und Ergänzungen nötig werden. Der besseren Nachvollziehbarkeit halber werden die inhaltlich relevanten Anpassungen nach Semesterbeginn hier dokumentiert:

- 13.11.2026:
  - Korrektur an Formel für Frobenius-Norm in Abschnitt 3.8.1.2.
- 12.11.2026:
  - Fehlende Klammer in Aufgabe 4.2(c) ergänzt.
  - Tippfehler in Abschnitt 3.8 korrigiert.
- 06.11.2026:
  - Fehler in 3.7.2.2 korrigiert: Die Jacobi-Matrix ist im Allgemeinen nicht symmetrisch.
- 05.11.2026:
  - Diverse Tippfehler korrigiert.
  - Fehler im Beispielcode in Übungsaufgabe 5.1 korrigiert.
- 29.10.2026:
  - Diverse Tippfehler korrigiert.
- 23.10.2026:
  - Korrektur in Abschnitt 3.3.5.6 an Formeln für $D(f)$ und f.
- 22.10.2026:
  - Formel für absoluten Fehler in Abschnitt 3.3.5.3 korrigiert.
  - Kennzeichnung “KTA” von Übungsaufgabe 3.3 entfernt.
  - Tippfehler in Unterkapiteln 3.2 und 3.3 korrigiert.
  - Notebook-Download der Übungsaufgaben repariert.
- 15.10.2026:
  - Notebook-Download der Übungsaufgaben repariert.

  - Tippfehler in den Abschnitten 3.2 und 3.3 korrigiert.

- 09.10.2026:
  - Tippfehler in Kapiteln 1 und 2 korrigiert.
- 08.10.2026:
  - Kapitel zur Statistik entfernt (ist nicht mehr Teil des Moduls).
- 07.10.2026:
  - Formatierung in Übung 1 korrigiert.
  - Beispielklausur hinzugefügt.

Literaturverzeichnis

[Hak75] H Haken. Analogy between higher instabilities in fluids and lasers. *Physics Letters A*, 53(1):77–78, 5 1975.

[Hem94] N. Hemati. Strange attractors in brushless DC motors. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 41(1):40–45, 1994.

[Kno81] E. Knobloch. Chaos in the segmented disc dynamo. *Physics Letters A*, 82(9):439–440, 4 1981.