Wissenschaftliches Rechnen (i788)

Jens Flemming (HTW Dresden)

24.06.2026

Inhaltsverzeichnis

0 Vorwort

Dieses Vorlesungsskript ist die Arbeitsgrundlage für die Veranstaltung “Wissenschaftliches Rechnen” (Modul i788) an der HTW Dresden ab Sommersemester 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 Worum geht es?

Das wissenschaftliche Rechnen (auch: Scientific Computing) als Fachgebiet umfasst alle Aspekte des Einsatzes von Computern zur Klärung von Fragestellungen aus Naturwissenschaften, Technik und Wirtschaft. Das Spektrum reicht von der Modellierung realer Vorgänge über die Implementierung am Computer bis zur Visualisierung der erzielten Resultate.

Der erfolgreiche Einsatz des wissenschaftlichen Rechnens erfordern die Zusammenarbeit von Anwendern aus dem jeweiligen Wissenschaftsbereich, Mathematikern und Informatikern. Alle Beteiligten sollten neben Detailwissen in ihrem Gebiet auch einen Überblick über den Gesamtprozess haben. Nur so sind Missverständnisse in der Kommunikation und Fehlinterpretationen der jeweiligen Anforderungen zu vermeiden.

1.1 Zu lösende Teilprobleme

Wesentliche Schritte beim wissenschaftlichen Rechnen sind:

- Modellierung (Anwender + Mathematiker)
  - Welche mathematischen Modelle für den betrachteten Vorgang gibt es?
  - Welches davon nutzen? Einfaches oder komplexes Modell?
  - Eigenes/neues Modell entwickeln?
  - Wie gut ist das Modell (**Modellierungsfehler**)?
- Diskretisierung (Mathematiker + Informatiker)
  - Wie kann das Modell in computertaugliche Form gebracht werden?
  - Ggf. ist die Auswahl von Algorithmen im nächsten Schritt schon hier einzubeziehen!
  - Welcher Fehler entsteht dabei (**Diskretisierungsfehler**)?
- Implementierung (Informatiker + Mathematiker)
  - Welche Algorithmen wählen?
  - Gibt es geeignete Bibliotheken?
  - Unnötige Ungenauigkeiten in Berechnungen vermeiden (**Approximationsfehler**, **Rundungsfehler**, vgl. Veranstaltung zur numerischen Mathematik)!
  - In welcher Form sollen die Ergebnisse geliefert werden (Tabellen, Visualisierungen,...)?
- Einsatz der Software (Anwender)
  - Interpretation der erzielten Ergebnisse. Sind die plausibel?

Am Ende sollte ein Software-Produkt vorliegen, welches Ergebnisse mit bekannter Genauigkeit liefert. Ein Schwerpunkt des wissenschaftlichen Rechnens wird also die Fehleranalyse sein. **Ohne solide Fehleranalyse sind die erzielten Ergebnisse nicht vertrauenswürdig und damit im Wesentlichen unbrauchbar!** Manchmal kann man die Ergebnisse durch reale Messungen validieren, oft aber nicht oder nicht in hinreichendem Umfang.

Das wissenschaftliche Rechnen ist also eng verbunden mit der numerischen Mathematik. Eine genaue Abgrenzung ist nicht möglich. Üblicherweise zählt man das Entwickeln und Untersuchen von Algorithmen für klar abgegrenzbare, unabhängig vom konkreten Anwendungsfall immer wiederkehrende Teilaufgaben zur numerischen Mathematik. Komplexere Algorithmen und die Koordinierung der Einzelschritte zu einem Gesamtprozess werden hingegen dem wissenschaftlichen Rechnen zugeordnet.

1.2 Beispiele

Die folgenden Beispiel dienen zur Verdeutlichung der im wissenschaftlichen Rechnen zu lösenden Probleme und Teilprobleme. Später werden wir einige der Beispiele detaillierter untersuchen.

Brachistochrone

Anwendungsproblem: 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.

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 $h(x_2) = y_2$. Man kann zeigen, dass die benötigte Zeit $T(h)$ für das Durchlaufen der Bahn h mittels

$$T(h) = \frac{1}{\sqrt{2\,g}} \int_{x_1}^{x_2} \sqrt{\frac{1 + h'(x)^2}{-h(x)}}\,\mathrm{d}x \tag{1.1}$$

berechnet werden kann. Dabei ist g die wirkende Fallbeschleunigung. Die optimale Bahn ist somit als Lösung des Minimierungsproblems

$$T(h) \to \min_{h \in H} \tag{1.2}$$

gegeben, wobei H die Menge aller stetig differenzierbaren Funktionen mit $h(x_1) = y_1$ und $h(x_2) = y_2$ sein soll.

An diesem recht einfachen Problem und seiner Modellierung treten schon verschiedene Aspekte des wissenschaftlichen Rechnens 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**!

Diskretisierung: Die Menge aller stetig differenzierbaren Funktionen ist nicht mit dem Computer darstellbar. Somit müssen wir uns auf eine hinreichend repräsentative, aber endlich dimensionale 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 (z.B. tiefsten Punkt der Bahn analytisch berechnen)? Dann sollte auch die Näherungslösung differenzierbar sein.

Implementierung: Welchen Algorithmus zur Lösung des diskretisierten Optimierungsproblems wählen? Gibt es anwendbare Standardalgorithmen aus Bibliotheken oder muss

ein auf die konkrete Aufgabe maßgeschneiderter Algorithmus entwickelt werden? In welcher Form soll das Ergebnis geliefert werden? Nur die optimale Bahn h oder beispielsweise eine Animation der Bewegung der Kugel entlang der Bahn mit korrekter Geschwindigkeit bzw. Beschleunigung? Beachte: Die gewählte Diskretisierung hat Einfluss auf die Art des entstehenden endlichdimensionalen Optimierungsproblems und damit auf die Auswahl der Algorithmen!

Nebenbemerkung

Das Brachistochrone-Problem kann man analytisch lösen, sodass eine numerische Lösung eigentlich nicht nötig ist. Dennoch sieht man an diesem einfachen Beispiel gut, wie vielfältig die zu klärenden Fragen beim wissenschaftlichen Rechnen selbst für einfache Problem sind.

Laplace-DLTS

Anwendungsproblem: 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.

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

$$C(t) = \int_0^\infty g(s)\, \mathrm{e}^{-s\,t}\, \mathrm{d}s. \tag{1.3}$$

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

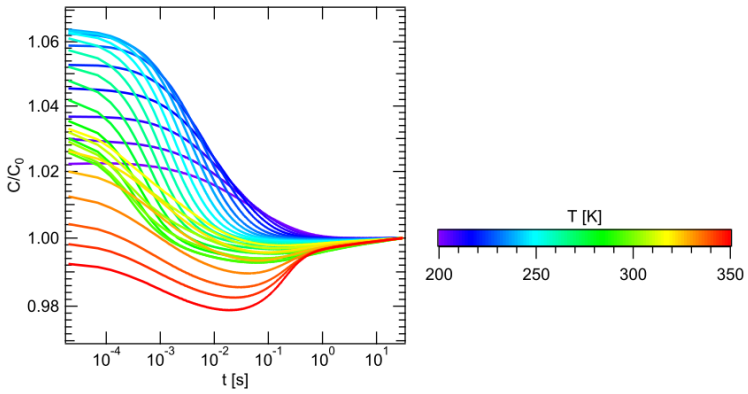


Abbildung 1.1: *
Gesamttransienten $t \mapsto C(t)$ zu verschiedenen Temperaturen T (gemessene Eingangsdaten für das Modell). Die Transienten wurden mit dem Faktor $\frac{1}{C_0}$ skaliert.

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.

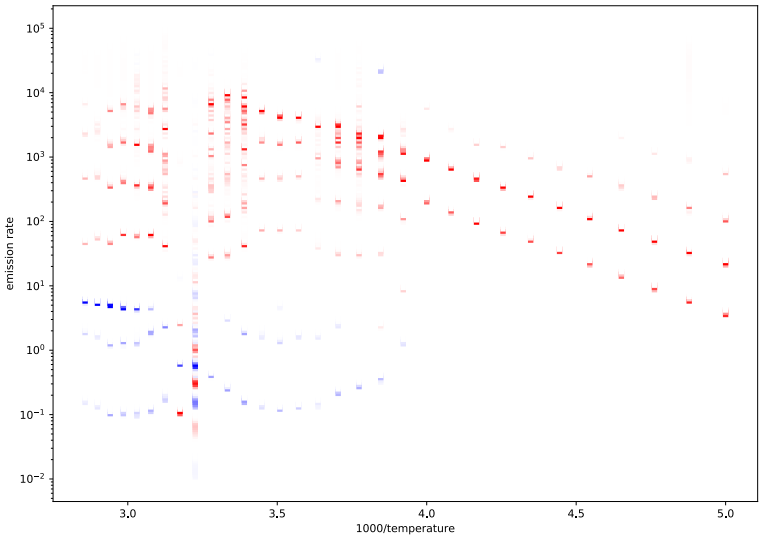


Abbildung 1.2: *
Arrhenius-Plot zu zwei verschiedenen Messreihen (blau und rot). Die Anordnung der Punkte entlang von Geraden ist gut erkennbar Die Farbintensität der Punkte gibt die Höhe der entsprechenden Peaks in der berechneten Funktion g an.

Modellierung: Grundfrage ist hier, ob mehrere Einzelmodelle verwendet werden sollen, also

- Invertieren der Laplace-Transformation zum Erhalt der Abklingratenfunktion g,
- Ermitteln der Peak-Positionen in g bei verschiedenen Temperaturen,
- Finden von Geraden im Arrhenius-Plot,

oder ob das Problem "Ende-zu-Ende" formuliert werden soll: Finde Lage und Anstieg der Geraden im Arrhenius-Plot aus den C-Messungen bei verschiedenen Temperaturen. Die erste Variante ist einfacher zu modellieren, da jedes Teilmodell eine klar abgegrenzte Aufgabe löst. Die zweite Variante vermeidet eventuelle Ungenauigkeiten in den Einzelschritten (Peak-Position bei breiten Peaks,...), ist aber in der mathematischen Formulierung sehr anspruchsvoll. Die zweite Variante ist ein typischer Kandidat für mittels maschinellem Lernen erstellte Modelle, sofern genügend Trainingsdaten beschafft werden können (siehe später). Verfolgen hier nur die erste Variante weiter.

Diskretisierung: Klar ist, dass sowohl g als auch C durch Vektoren zu ersetzen sind, z.B. als Werte von stückweise linearen Funktionen an den Stützstellen. Da C direkt aus Messungen entsteht, sollte sich die Diskretisierung hier an den messtechnischen Gegebenheiten orientieren. Im konkreten Anwendungsfall besteht eine C-Messung aus ca. 300000 Messwerten an äquidistanten Stützstellen. Wird diese hohe Auflösung beibeihalten, entsteht ein extrem hoher Rechenaufwand! Gleichmäßiges Ausdünnen der Messpunkte führt allerdings zu Datenverlust. Ob dieser hinnehmbar ist, muss mit den Anwendern geklärt werden. Es stellt sich heraus, dass diese hohe Auflösung am Anfang des Messzeitraums notwendig ist (sehr schnell abfallende Einzeltransienten), im weiteren Zeitverlauf aber zunehmend ausgedünnt werden kann (langsam abfallende Einzeltransienten). Für C ist also eine geeignete nichtäquidistante Diskretisierung zu wählen. Für die Abklingratenfunktion g sieht es ähnlich aus. Werte Nahe der Null müssen präzise darstellbar sein; größere Werte können mit geringerer Genauigkeit wiedergegeben werden. Alternative: Da aufgrund physikalischer Gegebenheit g nur aus wenigen Peaks besteht, kann man g auch als Summe von Einzelpeaks darstellen. Der Zusammenhang zwischen den entsprechenden Parametern von g (Peak-Positionen, Peak-Höhen) und der Funktion C ist dann allerdings nichtlinear, was das algorithmische Invertieren der Laplace-Transformation deutlich erschwert.

Implementierung: Das numerische Invertieren der Laplace-Transformation ist sehr schwierig, obwohl es sich (nach Diskretisierung) um ein lineares Gleichungssystem handelt. Problem ist die extrem hohe Kondition dieses Gleichungssystems, welche um so höher ist, je feiner (!) diskretisiert wird. Einzige Rettung ist das Einbringen von Zusatzinformationen über die Lösung (z.B. dass diese nur wenige scharfe Peaks hat), was algorithmisch sehr anspruchsvoll werden kann. Siehe Improved evaluation of deep-level transient spectroscopy on perovskite solar cells reveals ionic defect distribution für einen konkreten Algorithmus.

Sehen auch hier typische Aspekte des wissenschaftlichen Rechnens:

- Bei der Modellierung gibt es viele Freiheiten, aber auch Unschärfen. Bereits bei der Modellierung sollte man die weiteren Schritte zur Lösung im Blick haben.

- Diskretisierung unterliegt vielen praktischen Zwängen wie Art und Anzahl verfügbarer Messungen oder verfügbarer Computerressourcen.
- Manchmal gibt es keine Standardalgorithmen, obwohl das gewählte Modell seit Jahrzehnten vielfach im Einsatz ist bei ähnlichen Fragestellungen.

SD-SPIDER

Die kürzesten je von Menschen erzeugten Ereignisse sind mit Femtosekundenlasern erzeugte ultrakurze Laserpulse. Jeder Puls dauert nur wenige Femtosekunden (die Lichtgeschwindigkeit liegt bei 0.3 Mikrometer pro Femtosekunde). Zur Untersuchung solcher Laserpulse können keine einfachen Verfahren eingesetzt werden, da diese praktisch immer auf Abtastung des Signals basieren. Das Signal ist hier aber kürzer als der Abstand zwischen zwei Abtaststellen selbst bei optischen Verfahren.

Eine Möglichkeit zur Charakterisierung von ultrakurzen Laserpulsen ist SD-SPIDER (self-diffraction spectral phase interferometry for direct electric field reconstruction). Kurzfassung: Der zu untersuchende Laserpuls wird auf verschiedene Weisen transformiert und dann mittels eines nichtlinearen optischen Materials (Bariumfluorid) wieder zusammengeführt. Das Ergebnis kann dann durch einen Spektrograf aufgenommen werden. Die ursprüngliche Pulsform kann daraus jedoch nicht direkt abgelesen werden, sondern erfordert relativ komplexe Berechnungen.

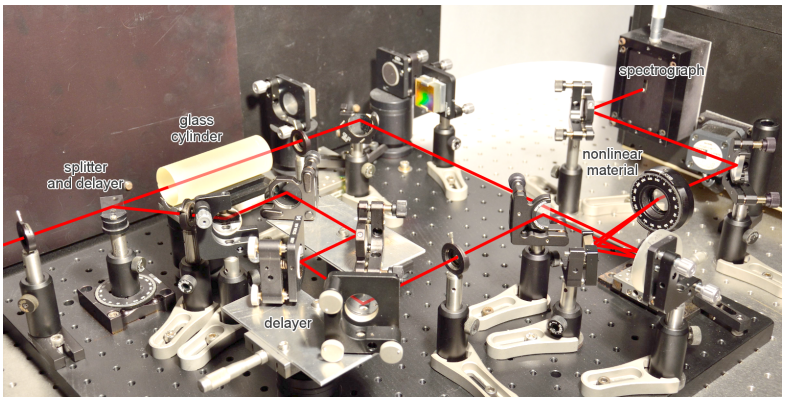


Abbildung 1.3: *
Versuchsaufbau für SD-SPIDER mit Strahlverlauf (Foto aufgenommen von S. Birkholz, Max-Born-Institut für Nichtlineare Optik und Kurzzeitspektroskopie, Berlin; Strahlverlauf und Beschriftung vom Autor eingefügt).

Modellierung: Aus physikalischen Zusammenhängen kann man ableiten, dass der Zusammenhang zwischen gemessener Funktion $y : \mathbb{R} \to \mathbb{C}$ (aus Spektrograf) und gesuchter Funktion $x : [0, 1] \to \mathbb{C}$ (auf Einheitsintervall transformierte Pulsform) durch die nicht-

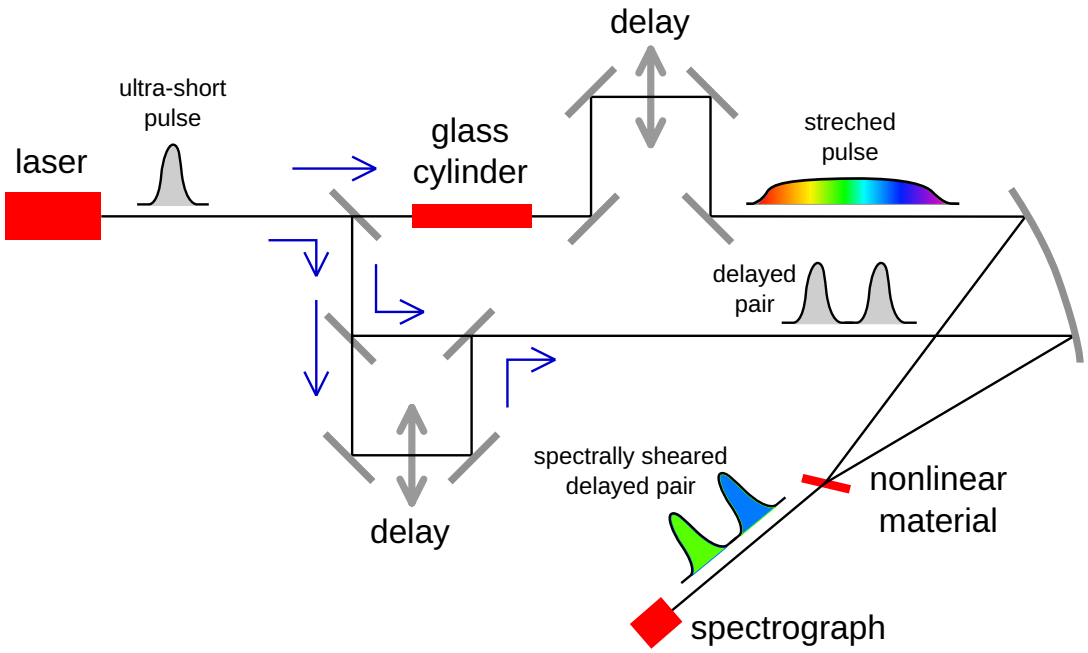


Abbildung 1.4: *
Schematischer Versuchsaufbau für SD-SPIDER.

lineare Integralgleichung

$$y(s) = \int_{\max\{0,s-1\}}^{\min\{s,1\}} k(s,t)\, x(t)\, x(s-t)\, \mathrm{d}t, \quad s \in [0,2] \tag{1.4}$$

gegeben ist. Dabei ist k eine bekannte Funktion, die sogenannte Kernfunktion. Diese enthält Informationen über das eingesetzte nichtlineare Material und weitere Parameter des Experiments. Gleichungen dieser Form heißen Selbstfaltungsgleichungen.

Diskretisierung: Für die Diskretisierung können Standardverfahren eingesetzt werden (z.B. stückweise konstante Funktionen). Allerdings ist zu beachten, dass die Funktionen x und y komplexwertig sind, Funktionswerte also als Paar von reellen Zahlen anzusehen sind mit entsprechenden Anpassungen bei den Rechenoperationen. Ergebnis ist ein nichtlineares Gleichungssystem mit sehr schlechter Kondition (je feiner diskretisiert wird, desto schlechter die Kondition).

Implementierung: Für das zu lösende nichtlineare Gleichungsystem ist kein Algorithmus bekannt, der garantiert die korrekte Lösung oder wenigstens eine hinreichend gute Näherungslösung liefert. Standardverfahren (z.B. Newton-Verfahren) sind im Prinzip anwendbar, aber keine der dafür bekannten Konvergenzaussagen ist im konkreten Fall zutreffend, da das Gleichungssystem die in diesen Aussagen formulierten Voraussetzungen nicht erfüllt. Auf der anderen Seite hat das Gleichungssystem viel Struktur, die sich ggf. zur Entwicklung von Algorithmen nutzen lässt. In der Tat stellt sich heraus, dass man die hier vorhandene "quadratische" Struktur nutzen kann für eine Zerlegung in zwei Teilprobleme. Eins davon ist ein (großes, aber leicht parallelisierbares) lineares Gleichungssystem. Das andere ist ein Eigenwertproblem, welches mit Standardverfahren gelöst werden kann. Für Details siehe [BF14]

Wir sehen:

- Schlecht konditionierte Probleme treten häufiger auf im wissenschaftlichen Rechnen als man erwarten würde. Insbesondere in Physik und Technik sind sie sehr präsent. Entsprechend vorsichtig muss bei der Auswahl von Lösungsverfahren und konkreten Algorithmen vorgegangen werden .
- Für aktuelle Aufgabenstellungen aus der Forschung gibt es meist noch keine Standardverfahren. Bestenfalls kann man bekannte Verfahren neu kombinieren oder anpassen.
- Die Aufgabenverteilung zwischen Mathematik (nichtlineares Problem in einfachere Teilprobleme zerlegen) und Informatik (effiziente Lösbarkeit der Teilprobleme und damit praktische Durchführbarkeit) ist fließend.

Lorenz-System

Schon in der Anfangszeit der Computer wollte man Wetterphänomene simulieren und zukünftige Entwicklungen in der Atmosphre vorhersagen. Aufgrund technischer Beschrän-

kungen mussten die Fragestellungen mit sehr einfachen Modellen formuliert und beantwortet werden. Eins davon ist das Lorenz-System, welches Aussagen zum vertikalen Wärme- und Feuchtigkeitstransport in der Atmosphäre liefert.

Modellierung: Sind x, y, z von der Zeit t abhängige Atmosphärenparameter (vertikaler Feuchtetransport pro Zeiteinheit, Temperaturdifferenz zwischen auf- und absteigenden Luftmassen, Abweichung des vertikalen Temperaturprofils von einer linearen Funktion), so ist deren Zusammenhang durch ein System nichtlinearer gewöhnlicher Differentialgleichungen gegeben:

$$\begin{aligned} x'(t) &= a\,(y(t) - x(t)), \\ y'(t) &= x(t)\,(b - z(t)) - y(t), \\ z'(t) &= x(t)\,y(t) - c\,z(t). \end{aligned} \tag{1.5}$$

Dabei sind a und b Materialparameter der strömenden Luft und c gibt gewisse geometrische Eigenschaften des betrachteten Luftvolumens an.

Diskretisierung: Für das Lösen von gewöhnlichen Differentialgleichungen existieren zahlreiche Verfahren. Die meisten nutzen zur Diskretisierung stückweise lineare Funktionen.

Implementierung: Es können Standardalgorithmen für gewöhnliche Differentialgleichungen genutzt werden. Allerdings ist bei der Auswahl zu beachten, dass diese mit der Tatsache umgehen können müssen, dass das Differentialgleichungssystem schlecht konditioniert ist, also kleine Änderungen in a, b, c zu extrem großen Änderungen in der Lösung führen können (schlechte Kondition). Art und Umfang dieser Abweichungen sehen sehr unsystematisch aus, weshalb man bei den Lösungen des Lorenz-Systems auch von chaotischem Verhalten spricht. Im Kontext des Lorenz-Systems wurde auch erstmals der Begriff Schmetterlingseffekt genutzt.

Bemerkenswerter Aspekt:

- Auch einfache Modelle können sehr komplexes Lösungsverhalten zeigen. Man unterschätze nie die Schwierigkeiten, die auch scheinbar einfache Aufgabenstellungen mit sich bringen können!

Wärmeleitung

Ein während des Produktionsprozesses durch und durch erhitztes Stahlteil muss vor dem nächsten Verarbeitungsschritt unter eine zulässig Höchsttemperatur abkühlen. Wie lange dauert es bis auch im Inneren des Bauteils in jedem Punkt diese Temperatur unterschritten wird?

Modellierung: Bezeichnen mit $B \subseteq \mathbb{R}^3$ das Bauteil als Teilmenge des Raumes und mit ∂B dessen Oberfläche. Sei $u : B \times [0, \infty) \to \mathbb{R}$ die zeitabhängige Temperaturverteilung im Bauteil; $u(x, t)$ gibt also die Temperatur zur Zeit t im Punkt $x \in B$ an. Der Abkühlprozess beginne bei $t = 0$. Aus physikalischen Überlegungen und Gesetzmäßigkeiten

erhält man den Zusammenhang

$$\left(\frac{\partial}{\partial x_1}\right)^2 u(x,t) + \left(\frac{\partial}{\partial x_2}\right)^2 u(x,t) + \left(\frac{\partial}{\partial x_3}\right)^2 u(x,t) = a\,\frac{\partial}{\partial t} u(x,t) \tag{1.6}$$

für alle $x \in B$ und alle $t \geq 0$, wobei $a > 0$ ein Materialparameter ist, der im Wesentlichen die Wärmeleitfähigkeit des Materials wiedergibt. Sind T_U und T_0 die Umgebungstemperatur und die Temperatur des Bauteils zur Zeit $t = 0$, so kommen noch die Anfangsbedingung

$$u(x,0) = T_0 \quad \text{für alle } x \in B \tag{1.7}$$

und die Randbedingung

$$u(x,t) = T_\mathrm{U} \quad \text{für alle } x \in \partial B \text{ und alle } t > 0 \tag{1.8}$$

zu dieser partiellen Differentialgleichung hinzu.

Diskretisierung: Die Diskretisierung kann mittels Standardverfahren erfolgen. Diese führen auf ein lineares Gleichungssystem. Zu beachten ist, dass die Geometrie des Körpers wesentlichen Einfluss auf die Diskretisierung hat. Das Diskretisieren von partiellen Differentialgleichungen ist wesentlicher Bestandteil dieser Veranstaltung.

Implementierung: Grundsätzlich stehen vielfältige Algorithmen in Software-Bibliotheken zur Verfügung. Je nach Geometrie des Bauteils können aber Anpassungen oder Erweiterungen nötig sein.

Relevante Aspekte:

- Der Großteil der Gesetze der (klassischen) Physik ist in Form von partiellen Differentialgleichungen formuliert. Entsprechend häufig tauchen partielle Differentialgleichungen als Modelle in der Anwendung auf.
- Für das Formulieren von partiellen Differentialgleichungen sind solide Fähigkeiten in der mathematischen Sprache sehr hilfreich. Das analytische Untersuchen (und Lösen, sofern möglich) von partiellen Differentialgleichung erfordert sehr tiefe Mathematikkenntnisse.

Elektrische Impedanz-Tomografie

Möchte man das Innere eines Testobjekts zerstörungsfrei untersuchen, kommen verschiedene Tomografieverfahren in Betracht. Eins davon, welches insbesondere in der medizinischen Bildgebung im Einsatz ist, aber mit leichten Modifikationen auch zur Untersuchung von Bodenverhältnissen und Rohstofflagerstätten dient, ist die elektrische Impedanz-Tomografie. Hier werden an der Oberfläche des Testobjekts Elektroden angebracht und in verschiedenen Kombinationen Spannungen angelegt. Die daraus an den anderen Elektroden resultierenden Ströme können gemessen werden und geben Aufschluss über die (ortsveränderliche) Leitfähigkeit und damit über Materialstrukturen im Inneren des Testobjekts.

Modellierung: Sei $B \subseteq \mathbb{R}^2$ das Testobjekt und seien die Elektroden gleichmäßig über dessen Rand ∂B verteilt (ringförmige Anordnung der Elektroden an Testkörper, somit reicht 2D-Betrachtung). Die Leitfähigkeit im Testkörper sei durch eine unbekannte Funktion $a : B \to \mathbb{R}$ gegeben. Bezeichnen wir mit $u : B \to \mathbb{R}$ das elektrische Potential im Testobjekt, so folgt aus physikalischen Überlegungen und Gesetzmäßigkeiten, dass die partielle Differentialgleichung

$$\frac{\partial}{\partial x_1}\left(a(x)\,\frac{\partial}{\partial x_1}u(x)\right) + \frac{\partial}{\partial x_2}\left(a(x)\,\frac{\partial}{\partial x_2}u(x)\right) = 0 \tag{1.9}$$

erfüllt sein muss. Das Anlegen von Spannungen an verschiedenen Elektrodenkombinationen entspricht der Vorgabe verschiedener Randwerte f für Lösungen u dieser Gleichung:

$$u(x) = f(x) \quad \text{für alle } x \in \partial B. \tag{1.10}$$

Die daraus resultierenden Strommessungen entlang des Randes von B liefern jeweils ein Funktion g mit

$$a(x)\,\frac{\partial}{\partial n}u(x) = g(x) \quad \text{für alle } x \in \partial B, \tag{1.11}$$

wobei $\frac{\partial}{\partial n}u(x)$ die Richtungsableitung von u am Randpunkt x in Richtung des Normalenvektors des Randes an diesem Punkt bezeichnet. Eingangsdaten in das Modell der elektrischen Impedanz-Tomografie sind nun hinreichend viele Paare (f, g) aus angelegten Spannungen und daraus resultierenden Strömen. Die Ausgabe des Modells ist die ortsabhängige Leitfähigkeit a. Aus der Kenntnis von Eigenschaften der Lösungen (Randwerte, Fluss über den Rand) soll also auf einen Parameter der zugrunde liegenden partiellen Differentialgleichung geschlossen werden. Diese Art von Aufgabenstellungen nennt man deshalb auch Parameteridentifikationsprobleme.

Diskretisierung: Die Diskretisierung kann mit Standardverfahren erfolgen. Allerdings muss unbedingt die Geometrie der Elektroden mit eingebracht werden um brauchbare Resultate erhalten zu können.

Implementierung: Algorithmen für das Lösen des Impedanz-Tomografie-Problems sind in gewissem Maße noch Gegenstand der Forschung. Letztlich kann man Standardverfahren kombinieren, erhält aber nur sehr schwache Aussagen über die Vertrauenswürdigkeit der Lösungen. Zu allen angelegten Spannungen kann man bei gegebener Leitfähigkeitsverteilung durch Lösen der Differentialgleichung die zu erwartenden Ströme berechnen. Diese vergleicht man mit den gemessenen Strömen und passt die Leitfähigkeitsverteilung ausgehend von einer initialen Verteilung so lange an, bis die Abweichung zwischen berechneten und gemessenen Strömen klein genug ist. Dieser algorithmische Ansatz löst also ein nichtlineares Minimierungsproblem iterativ, wobei jede einzelne Auswertung der Zielfunktion das Lösen mehrerer partieller Differentialgleichungen erfordert (eine Gleichung pro Spannungs-Strom-Paar).

1.3 Fallstudien

Wissenschaftliches Rechnen ist ein sehr umfassendes und fachübergreifendes Thema. In dieser Veranstaltung werden wir keine systematische Abhandlung vornehmen, sondern uns anhand mehrerer umfangreicher Fallstudien mit exemplarischen Anwendungsszenarien und deren Umsetzung von der Modellierung bis zur Implementierung befassen.

Nur an einigen Punken, insbesondere mit Blick auf partielle Differentialgeleichungen, werden wir tiefer in die theoretische Fundierung und Motivation von Algorithmen einsteigen. Partielle Differentialgleichungen sind das Hauptwerkzeug für die Modellierung. Es wird sich allerdings zeigen, dass man große Klassen partieller Differentialgleichungen durch Transformation in Integralgleichungen lösen kann (und dies auch tut), sodass die Diskussion von Integralgleichungen nicht ausbleiben kann.

Hauptziele der Veranstaltung sind:

- einen Überblick über die verschiedenen Schritte von der Problemformulierung bis zur Lösung am Computer zu bekommen,
- Begriffe einordnen zu können, die bei der Beschreibung von Algorithmen aus dem Umfeld des wissenschaftlichen Rechnens auftauchen können,
- das Auge für Feinheiten zu schärfen, die durch ungeeignete Implementierung von Algorithmen zu (teils groben) Fehlern in der Lösung führen können.

1.4 Sleipner A zur Mahnung

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.

2 Modellierung

Der erste Schritt beim wissenschaftlichen Rechnen vom Problem zur Lösung ist das Modellieren des zu lösenden Problems, also das Formulieren des Problems in der Sprache der Mathematik. Hier gibt es viel Spielraum. Sehr einfache Modelle lassen sich leichter algorithmisch lösen, geben die Realität unter Umständen aber zu ungenau wieder. Komplexe Modelle hingegen liefern deutlich besser verwertbare Ergebnisse, so man sie algorithmisch überhaupt umgesetzt bekommt. Schon beim Modellieren müssen die algorithmischen Möglichkeiten mitgedacht werden!

2.1 Anforderungen an ein gutes Modell

Als grundlegende Anforderungen an ein Modell können die sogenannten Hadamard-Bedingungen gelten:

- **Existenz:** Das Modell besitzt eine Lösung.
- **Eindeutigkeit:** Das Modell besitzt genau eine Lösung.
- **Stetigkeit:** Die Lösung hängt stetig von den Eingaben ab. Kleine Änderungen in den Eingaben resultieren also auch nur in kleinen Änderungen bei der Lösung.

Ein Modell, das keine Lösung besitzt, ist offensichtlich wertlos. Die Forderung nach Eindeutigkeit ist weniger wesentlich. Manchmal lässt der modellierte Sachverhalt keine eindeutige Lösung zu, z.B. weil zu wenig Messdaten vorhanden sind. In solch einem Fall sollte bei der Modellierung allerdings geprüft werden, ob Eindeutigkeit der Lösung durch sachlich begründbare Zusatzannahmen erreicht werden kann. Nicht-Eindeutigkeit führt oft zu Problemen bei der algorithmischen Umsetzung.

Beispiel (Schlierentomografie)

Bei der Schlierentomografie werden Schlierenfotos eines Flüssigkeitsvolumens (meist ein mit Wasser gefüllter Glaszylinder) aus verschiedenen Richtungen aufgenommen. Aus diesen Fotos kann dann die Druckverteilung in der Flüssigkeit rekonstruiert werden. Die Schlierentomografie ist beispielsweise ein Standardwerkzeug zur Untersuchung des Schalldruckbildes von Ultraschallsonden.

Die Schlierenfotos enthalten keine Vorzeicheninformationen, also keine Information darüber, ab in einer Region Unter- oder Überdruck herrscht. Entsprechend hat das Tomografieproblem (also die 3D-Rekonstruktion der Druckverteilung) stets zwei Lösungen, die sich gerade im Vorzeichen unterscheiden. Um Eindeutigkeit zu erzwingen, müsste man das Vorzeichen in einer Über- bzw. Unterdruckzone vorgeben. Allerdings weiß man im Vorfeld meist nicht, wo Abweichungen vom Normaldruck liegen.

Die Forderung nach Stetigkeit entspricht dem Begriff der Kondition in der numerischen

Mathematik. Hängt die Lösung nicht stetig von den Eingaben ab, so ist das Modell unbrauchbar. Schon kleine, unvermeidbare Rundungsfehler führen zu völlig anderen Ergebnissen als die exakten Werte. Praktisch werden die Eingaben oft Messwerte sein und damit ohnehin einen großen Fehler aufweisen. Verletzt ein ansonsten plausibles Modell die Stetigkeitsforderung, so können allgemein anwendbare Techniken eingesetzt werden, um daraus ein stetiges Modell abzuleiten. Als Beispiel für diese Problematik diskutieren wir später die Computertomografie im Detail.

Die Forderungen von Hadamard stammen aus den 1930er Jahren, wurden also noch vor dem Beginn des Computerzeitalters formuliert. Heute müssen wir noch eine vierte Forderung formulieren:

- **algorithmische Lösbarkeit:** Das Modell muss mit dem Computer lösbar sein, es muss also ein Algorithmus existieren, der die Lösung wenigstens näherungsweise liefert.

Negativbeispiele sind hier vorallem nichtglatte nichtlineare Optimierungsprobleme, die oft als naheliegendes, einfach zu formulierendes Modell auftreten, aber mangels geeigneter Algorithmen praktisch nicht lösbar sind.

2.2 Klassische Modelle

Als klassische Modelle verstehen wir alle Modelle, die folgenden Kriterien genügen:

- Die Eingaben sind Zahlen oder Funktionen.
- Die Ausgaben sind Zahlen oder Funktionen.
- Der Zusammenhang lässt sich mit einfachen mathematischen Werkzeugen in wenigen Formeln darstellen.

Meist wird der Zusammenhang zwischen Ein- und Ausgaben über (partielle) Ableitungen und/oder Integrale von Funktionen dargestellt. Diese werden dann in Gleichungssystemen oder Optimierungsproblemen zu einander in Beziehung gesetzt.

Typische Beispiele sind Modelle aus der klassischen Mechanik, die aus dem Newton'schen Gesetz "$F = m\,a$" gewonnen wurden, und Modelle, die aus Erhaltungssätzen (Erhaltung von Masse, Energie, Trägheitsmoment,...) entstanden sind.

Gegenbeispiel sind quantentheoretische Modelle, die für die Formulierung meist lineare Abbildungen zwischen unendlichdimensionalen Vektorräumen benötigen, sowie statistische und datenbasierte Modelle (siehe unten).

2.3 Beispiele

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:

- Eingaben: Weg-Zeit-Zusammenhang $t \mapsto s(t)$ und Zeitpunkt t^*.
- Ausgabe: Momentangeschwindigkeit v^* zur Zeit t^*.
- Zusammenhang: $v^* = s'(t^*)$.

Für den praktischen Einsatz des Modells werden wir dieses diskretisieren müssen. Die Eingabefunktion s werden wir durch eine auf den Messwerten $s_1, \ldots, s_n$ basierenden Näherung ersetzen müssen. Die Ausgabe besteht nur aus einer Zahl, sodass hier keine Diskretisierung nötig ist. In Abhängigkeit von der Diskretisierung ist dann noch zu klären, wie s' ausgewertet werden kann.

Nebenbemerkung

Dass das Modell kontinuierlich ist, also auf Funktionen basiert, mag übertrieben aufwendig erscheinen, da ja klar ist, dass nur endliche 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 sehr 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 endlichdimensionaler Arbeitsweise an kontinuierlichen (und damit unendlichdimensionalen) Modellen interessiert ist.

Ein anderes, in gewisser Hinsicht leichter handhabbares Modell ergibt sich aus dem Hauptsatz der Differential- und Integralrechnung:

$$s(t) = s(0) + \int_0^t s'(\tau)\,\mathrm{d}\tau. \tag{2.3}$$

Nehmen wir $s(0) = 0$ an und bezeichnen wir für beliebige Funktionen $g : [0, T]$ mit $F(g)$ die Funktion

$$\big(F(g)\big)(t) := \int_0^t g(\tau)\,\mathrm{d}\tau, \quad t \in [0, T], \tag{2.4}$$

so ist die Momentangeschwindigkeit v (zu jedem Zeitpunkt) die Lösung der Integralgleichung

$$F(v) = s. \tag{2.5}$$

Entsprechend folgt dann $v^* = v(t^*)$. Diskretisieren der Gleichung wird auf ein lineares Gleichungssystem führen.

Wir haben also zwei recht unterschiedliche Modelle (Ableitung vs. Integralgleichung) für ein und das selbe Problem gefunden. Später werden wir uns genauer mit deren Diskretisierung, sowie Vor- und Nachteilen befassen.

Brachistochrone

Hatten schon kurz ein Section 1.2 erwähnt. Leiten dieses nun aus elementaren physikalischen Gegebenheiten her.

Vereinfachen durch geeignete Wahl des Koordinatenursprungs die Notation etwas: Die Kugel starte im Punkt $(0, 0)$ zur Zeit $t = 0$ und komme am Endpunkt $(\bar{x}, \bar{y})$ zur Zeit $T > 0$ an. Die Bahn werde durch eine Funktion $h : [0, \bar{x}] \to \mathbb{R}$ beschrieben. Alternativ ist auch die Schreibweise $(x(t), y(t))$ für die Position der Kugel zum Zeitpunkt t nützlich. Zu berechnen ist die benötigte Zeit T in Abhängigkeit von der Bahn h. Benötigte physikalische Größen und Gesetze:

- $E_{\mathrm{pot}}(t) = m\,g\,h(x(t))$ (potentielle Energie der Kugel zur Zeit t relativ zum Startpunkt; m ist Masse der Kugel; g ist die Fallbeschleunigung),

- $E_{\text{kin}}(t) = \frac{1}{2}\, m\, v(t)^2$ (kinetische Energie der Kugel zur Zeit t; $v(t)$ ist der Betrag der Geschwindigkeit zur Zeit t),
- $E_{\text{pot}} + E_{\text{kin}} = 0$ zu jeder Zeit (bei $t = 0$ klar; anschließend gilt Energieerhaltung).

Daraus erhält man die Formel

$$T(h) = \frac{1}{\sqrt{2\,g}} \int_0^{\bar{x}} \sqrt{\frac{1 + h'(x)^2}{-h(x)}}\, \mathrm{d}x \tag{2.6}$$

(IDVID 210). Die schnellste Bahn ist somit die Lösung des Minimierungsproblems

$$T(h) \to \min_{h \in H}, \tag{2.7}$$

wobei H die Menge aller auf $[0, \bar{x}]$ stetig differenzierbaren Funktionen ist.

Die Polstelle im Integrand bei $x = 0$ (also $h(x) = 0$) ist unkritisch, da die minimierende Funktion h endliches Integral besitzen wird, sofern mindestens ein h in H diese Eigenschaft hat (z.B. $h(x) = -\sqrt{x}$).

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:

$$\int_{x_1}^{x_2} F\big(h(x), h'(x), x\big)\, \mathrm{d}x \to \min_{h \in H} \tag{2.8}$$

mit einer das konkrete Problem definierenden Funktion $F : \mathbb{R}^3 \to \mathbb{R}$, wobei $h(x_1)$ und $h(x_2)$ gegeben sind.

Pendel

Die Bewegung eines 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

$$F(t, x(t), y(t)) = m \begin{bmatrix} x''(t) \\ y''(t) \end{bmatrix}, \quad t > 0, \tag{2.9}$$

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

$$\varphi''(t) + \frac{g}{l} \sin \varphi(t) = 0 \tag{2.10}$$

zur Beschreibung der Pendelbewegung (IDVID 230). Diese ist eine nichtlineare gewöhnliche Differentialgleichung zweiter Ordnung, die sich nur sehr mühsam analytisch lösen lässt.

Für kleine Auslenkungen kann man die Näherung

$$\sin \varphi(t) \approx \varphi(t) \tag{2.11}$$

verwenden um eine einfacher zu lösende lineare gewöhnliche Differentialgleichung zu erhalten:

$$\varphi''(t) + \frac{g}{l}\,\varphi(t) = 0. \tag{2.12}$$

Der dabei 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.

Zerfalls- und Wachstumsprozesse

Zerfallsprozesse und Wachstumsprozesse können durch gewöhnliche Differentialgleichungen beschrieben werden. Betrachten hier beispielhaft einen konkreten Zerfallsprozess: 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. Die Funktion $m : [0, \infty) \to \mathbb{R}$ soll durch eine gewöhnliche Differentialgleichung mit Anfangsbedingung beschrieben werden.

Dazu zunächst zwei Beobachtungen:

- Da Zerfallsprozesse je nach Material und Umgebung unterschiedlich schnell verlaufen können, muss die gesuchte Differentialgleichung 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.

Daraus erhält man die Beziehung

$$m' = c\,m \tag{2.13}$$

(IDVID 250). Dies ist eine homogene lineare Differentialgleichung erster Ordnung mit konstanten Koeffizienten. Sie drückt aus, dass die Änderung m' der Materialmenge stets proportional zur vorhandenen Materialmenge m ist.

Die Anfangsbedingung ist

$$m(0) = m_0. \tag{2.14}$$

Für $c > 0$ sind die Lösungen offensichtlich monoton wachsend, sonst monoton fallend. Die allgemeine Lösung ist

$$m(t) = a\,\mathrm{e}^{ct} \tag{2.15}$$

mit $a \in \mathbb{R}$. Einsetzen des Anfangswertes liefert

$$m(t) = m_0\,\mathrm{e}^{ct}. \tag{2.16}$$

2.4 Statistische Modelle

Sind Eingabe und/oder Ausgaben statistische Größen (Zufallsgrößen, Verteilungsfunktionen, Dichtefunktionen,...) so spricht man von statistischen Modellen. Diese unterscheiden sich sowohl in der Herleitung als auch in der algorithmischen Behandlung wesentlich von den klassischen Modellen.

Statistische Modelle treten in allen Wissenschaftsgebieten auf, in denen Systeme als Zusammenspiel sehr vieler gleichartiger Einzelsysteme untersucht werden. Die Spannbreite reicht von den Sozialwissenschaften über die Wirtschaftswissenschaften bis zur Physik.

Beispiel (Wiener-Prozess)

Konkretes Beispiel für ein statistisches Modell aus der Physik ist der Wiener-Prozess zur Beschreibung der Brown'schen Bewegung. Ausgabe des Modell ist eine zufällige Funktion $f : [0, \infty) \to \mathbb{R}$, welche jedem Zeitpunkt die Position eines sich zufällig entlang einer Achse vor oder zurück bewegenden Teilchens zuordnet. Dabei folgen die Differenzen der Positionen zu zwei verschiedenen Zeitpunkten einer Normalverteilung, deren Mittelwert von der Zeitdifferenz abhängt (je mehr Zeit vergangen ist, desto größer ist der Abstand im Mittel).

Der Wiener-Prozess liefert bei gegebenen Parametern (Eingaben) also keine konkrete Funktion als Ergebnis, sondern stets eine andere. Statt an einer einzelnen Funktion ist man an der Wahrscheinlichkeitsverteilung über alle möglichen Funktionen interessiert, sodass diese als Ausgabe des Modells zu betrachten ist.

In der Physik gibt es das Teilgebiet der statistischen Physik, welche statistische Modelle nutzt um insbesondere eine Vielzahl thermodynamischer Zusammenhänge zu formulieren. Statt eine extrem große Anzahl von interagierenden Teilchen durch teilchenbasierte Einzelmodelle zu beschreiben (was die verfügbare Rechenleistung von Computern gar nicht erlaubt), werden nur die statistischen Eigenschaften des Gesamtsystems untersucht.

2.5 Datenbasierte Modelle

Möchte man einen komplexen Vorgang modellieren, zu dessen Abbildungsverhalten schon sehr viele Messwerte vorliegen (welche Eingabe liefert welche Ausgabe?), so kann man

automatisiert aus den Messwerten ein Modell generieren, welches dieses Abbildungsverhalten auf die Menge aller möglichen Eingaben verallgemeinert. Diese Form des Modellierens wird als datenbasiertes Modellieren oder als **maschinelles Lernen** bezeichnet.

Seien $(x_1, y_1), \ldots, (x_n, y_n)$ Paare von Eingaben (Vektoren in $\mathbb{R}^m$) und zugehörigen Ausgaben (reelle Zahlen), die sogenannten **Trainingsdaten**, und sei $f : \mathbb{R}^m \to \mathbb{R}$ eine von reellen Parametern $w_1, \ldots, w_p$ abhängige Funktion. Dann ist das Minimierungsproblem

$$\sum_{l=1}^{n} \big(f(x_l) - y_l\big)^2 \to \min_{w_1,\ldots,w_p} \tag{2.17}$$

zu lösen. Die daraus erhaltenen Parameter liefern ein konkretes f als Modell für den Zusammenhang zwischen beobachteten Ein- und Ausgaben, welches auch bisher nicht beobachtete Eingaben verarbeiten kann.

Typische Wahlen für f sind Linearkombinationen einfacher Ansatzfunktionen (z.B. Monome, Hütchenfunktionen) oder speziell strukturierte, rekursiv definierte Verschachtelungen sehr einfacher Grundfunktionen (z.B. nichtlinear transformiertes Skalarprodukt aus Eingabevektor und Parametervektor). In ersterem Fall spricht man von **linearer Regression**, im zweiten Fall von **künstlichen neuronalen Netzen**.

Vorteile gegenüber klassischen Modellen:

- Physikalische oder anderweitige Zusammenhänge zwischen Ein- und Ausgaben müssen nicht bekannt sein.
- Leichte algorithmische Auswertung der Modelle (zur Eingabe x berechne den Funktionswert $f(x)$).

Nachteile:

- Zuverlässigkeit bei Eingaben, die nicht sehr nah an den Trainingsdaten sind, ist unklar.
- Bereich der Eingaben, die korrekte Ausgaben liefern, ist unklar.
- Lösung des Minimierungsproblems ist im Fall neuronaler Netze sehr aufwendig. Meist ist nur eine sehr grobe Näherung des eigentlichen Minimierers verfügbar.
- Es sind keine strukturellen Erkenntnisse aus dem Modell ableitbar (Warum gerade diese Ausgabe? “Black-Box”).
- Es werden große Mengen qualitativ hochwertiger Trainingsdaten benötigt.

Beispiel (Suszeptibilitätsgewichtete Magnetresonanztomografie)

Die verschiedenen Gewebearten von Tier und Mensch lassen sich in einem externen Magnetfeld unterschiedlich stark magnetisieren, d.h. sie besitzen unterschiedliche magnetische Suszeptibilität. Mit entsprechend konfigurierten Magnetresonanztomografen kann man die Suszeptibilität in einem komplexen Prozess ortsaufgelöst messen

und erhält so eine dreidimensionale Darstellung der Gewebestrukturen. Die suszeptibilitätsgewichtete Bildgebung ist noch in aktiver Entwicklung.

Die physikalischen Vorgänge sind komplex und die Interaktion der Magnetfelder mit dem Gewebe noch nicht vollständig verstanden. Insbesondere muss die Modellierung eine Vielzahl technischer Parameter des verwendeten MR-Tomografen berücksichtigen. Ein möglicher Ausweg ist die Kombination von klassischem Modell für die "grobe" physikalische Modellierung und datenbasiertem Modell für die Feinabstimmung. Das nachgelagerte datenbasierte Teilmodell kann, sofern hinreichend gute Trainingsdaten vorhanden sind, Modellfehler des klassischen Modells verringern. Ein ausschließlich datenbasierter Ansatz würde viel mehr Trainingsdaten benötigen und aufgrund des Black-Box-Verhaltens künstlicher neuronaler Netze nicht die in medizinischen Anwendungen benötigte Zuverlässigkeit liefern.

3 Numerische Differentiation

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

$$f'(x_0) := \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{3.1}$$

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

3.1 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

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \tag{3.2}$$

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

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}. \tag{3.3}$$

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

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2\,h} \tag{3.4}$$

  (IDVID 310).

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

$$f'(x_0) \approx \frac{f(x_0 + h_+) - f(x_0 - h_-)}{h_+ + h_-}. \tag{3.5}$$

Auch in diesem Fall kann die zentrale Differenz als (gewichteter) Mittelwert aus entsprechender Vorwärts- und Rückwärtsdifferenz interpretiert werden (IDVID 315).

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.

3.2 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

$$|f'(x_0) - D(x_0, h)| \leq c\, h^p \quad \text{für alle } h \in (0, h_{\max}] \tag{3.6}$$

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:

$$|f'(x_0) - D(x_0, h)| \leq c\, h \tag{3.7}$$

  mit

$$c = \frac{1}{2} \max_{\xi \in [x_0, x_0 + h_{\max}]} |f''(\xi)|, \tag{3.8}$$

sofern f zweimal stetig differenzierbar ist.

- Rückwärtsdifferenz:

$$|f'(x_0) - D(x_0, h)| \leq c\, h \tag{3.9}$$

mit

$$c = \frac{1}{2} \max_{\xi \in [x_0 - h_{\max}, x_0]} |f''(\xi)|, \tag{3.10}$$

sofern f zweimal stetig differenzierbar ist.

- Zentrale Differenz:

$$|f'(x_0) - D(x_0, h)| \leq c\, h^2 \tag{3.11}$$

mit

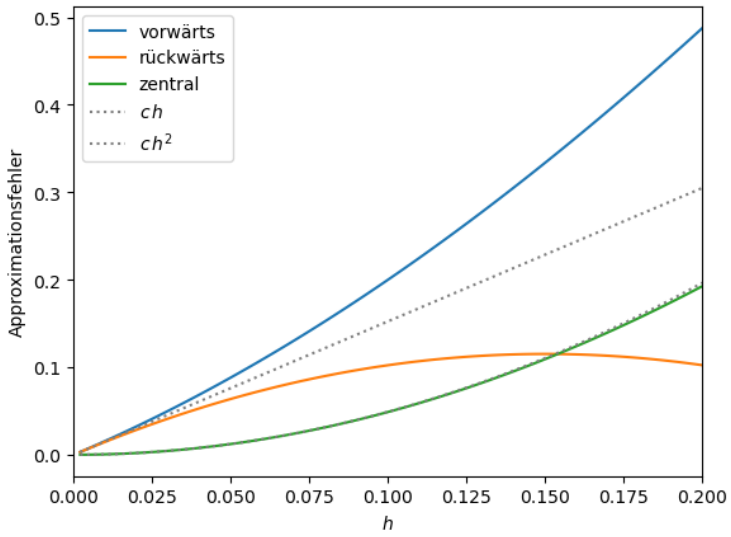
$$c = \frac{1}{6} \max_{\xi \in [x_0 - h_{\max}, x_0 + h_{\max}]} |f'''(\xi)|, \tag{3.12}$$

sofern f dreimal stetig differenzierbar ist.

(IDVID 330). 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:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f''(\xi)|\, h \tag{3.13}$$

  für $h > 0$ und $x \in [a, b - h]$,

- Rückwärtsdifferenz:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f''(\xi)|\, h \tag{3.14}$$

  für $h > 0$ und $x \in [a + h, b]$.

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

- zentrale Differenz:

$$|f'(x) - D(x, h)| \leq \max_{\xi \in [a,b]} |f'''(\xi)|\, h^2 \tag{3.15}$$

  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.

3.3 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

$$D^2(x_0, h) := \frac{f(x_0 + h) - 2\,f(x_0) + f(x_0 - h)}{h^2}. \tag{3.16}$$

Diese erhält auf allen der folgendenen Wege:

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

(IDVID 340). Der Approximationsfehler verhält sich hier quadratisch, falls f wenigstens viermal stetig differenzierbar ist:

$$|f''(x_0) - D^2(x_0, h)| \leq c\,h^2 \quad \text{für alle } h \in (0, h_{\max}] \tag{3.17}$$

mit

$$c = \frac{1}{12} \max_{\xi \in [x_0 - h_{\max}, x_0 + h_{\max}]} |f^{(4)}(\xi)| \tag{3.18}$$

(IDVID 345).

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

$$\frac{1}{h^k} \sum_{i=1}^{l} a_i\, f(\ldots) \tag{3.19}$$

mit

$$\sum_{i=1}^{l} a_i = 0. \tag{3.20}$$

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

4 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 einen anderen Ansatz und zusätzliche theoretische Untermauerung sei auf die Veranstaltung zur numerischen Mathematik verwiesen.

4.1 Idee

Um das Integral

$$I := \int_a^b f(x)\,\mathrm{d}x \tag{4.1}$$

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

$$f(x) \approx \sum_{i=1}^{n} c_i\, b_i(x). \tag{4.2}$$

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

$$I = \sum_{i=1}^{n} c_i \int_a^b b_i(x)\,\mathrm{d}x. \tag{4.3}$$

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.

4.2 Übliche Basisfunktionen

Stückweise konstante Funktionen

Für $a =: x_0 < x_1 < \cdots < x_n := b$ setze

$$b_i(x) := \begin{cases} 1, & \text{falls } x \in [x_{i-1}, x_i), \\ 0, & \text{sonst,} \end{cases} \tag{4.4}$$

für $i = 1, \ldots, n$.

Dann gilt

$$f(x) \approx \sum_{i=1}^{n} f\left(\frac{x_i + x_{i-1}}{2}\right) b_i(x) \tag{4.5}$$

und die Integrale der Basisfunktionen sind

$$\int_a^b b_i(x)\,\mathrm{d}x = x_i - x_{i-1}. \tag{4.6}$$

Stückweise lineare Funktionen

Für $a =: x_1 < x_2 < \cdots < x_n := b$ setze

$$\begin{aligned} b_1(x) &:= \begin{cases} \frac{x_2 - x}{x_2 - x_1}, & \text{falls } x \in [x_1, x_2], \\ 0, & \text{sonst}, \end{cases} \\ b_n(x) &:= \begin{cases} \frac{x - x_{n-1}}{x_n - x_{n-1}}, & \text{falls } x \in [x_{n-1}, x_n], \\ 0, & \text{sonst}, \end{cases} \end{aligned} \tag{4.7}$$

und

$$b_i(x) := \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{falls } x \in [x_{i-1}, x_i], \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{falls } x \in [x_i, x_{i+1}], \\ 0, & \text{sonst}, \end{cases} \tag{4.8}$$

für $i = 2, \ldots, n-1$.

Dann gilt

$$f(x) \approx \sum_{i=1}^{n} f(x_i)\, b_i(x) \tag{4.9}$$

und die Integrale der Basisfunktionen sind

$$\int_a^b b_i(x)\,\mathrm{d}x = \begin{cases} \frac{x_2 - x_1}{2}, & \text{für } i = 1, \\ \frac{x_{i+1} - x_{i-1}}{2}, & \text{für } i = 2, \ldots, n-1, \\ \frac{x_n - x_{n-1}}{2}, & \text{für } i = n. \end{cases} \tag{4.10}$$

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

$$b_i(x) := g(x - x_i) \tag{4.11}$$

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

$$g(x) := \begin{cases} 1 - \frac{n-1}{b-a}\,|x|, & \text{falls } |x| \le \frac{b-a}{n-1}, \\ 0, & \text{sonst} \end{cases} \tag{4.12}$$

  gerade die Basisfunktionen zur Darstellung stückweise linearer Funktionen.

- Mit der Glockenkurve

$$g(x) = \mathrm{e}^{-\frac{x^2}{\sigma^2}} \tag{4.13}$$

  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:

$$f(x_j) = \sum_{i=1}^{n} c_i\, g(x_j - x_i), \quad j = 1, \ldots, n. \tag{4.14}$$

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

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:

$$\begin{aligned} b_0(x) &:= \frac{1}{b-a} \\ b_i^{\sin}(x) &:= \sqrt{\frac{b-a}{2}}\,\sin\left(2\,\pi\,i\,\frac{x-a}{b-a}\right), \\ b_i^{\cos}(x) &:= \sqrt{\frac{b-a}{2}}\,\cos\left(2\,\pi\,i\,\frac{x-a}{b-a}\right). \end{aligned} \tag{4.15}$$

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

$$\begin{aligned} c_0 &:= \int_a^b f(x)\, b_0(x)\, \mathrm{d}x, \\ c_i^{\sin} &:= \int_a^b f(x)\, b_i^{\sin}(x)\, \mathrm{d}x, \\ c_i^{\cos} &:= \int_a^b f(x)\, b_i^{\cos}(x)\, \mathrm{d}x. \end{aligned} \tag{4.16}$$

Bezeichnen wir mit

$$c := (c_0, c_1^{\sin}, c_1^{\cos}, \ldots, c_n^{\sin}, c_n^{\cos}) \tag{4.17}$$

einen Koeffizientenvektor und mit

$$g_c(x) := c_0\, b_0(x) + \sum_{i=1}^{n} c_i^{\sin}\, b_i^{\sin}(x) + c_i^{\cos}\, b_i^{\cos}(x) \tag{4.18}$$

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

$$\int_a^b \big(f(x) - g_c(x)\big)^2\, \mathrm{d}x \to \min_{c \in \mathbb{R}^{2n+1}} \tag{4.19}$$

(ohne Beweis).

Für die Berechnung der Koeffizienten aus endlich vielen Funktionswerten von f existieren sehr effiziente Algorithmen (Schnelle Fourier-Transformation).

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.

4.3 Diskretisierte Stammfunktion

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

$$F : [a, b] \to \mathbb{R}, \quad F(z) = f(a) + \int_a^z f(x)\,\mathrm{d}x \tag{4.20}$$

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

$$F(z_j) = \sum_{i=1}^{n} c_i \int_a^{z_j} b_i(x)\,\mathrm{d}x, \quad j = 1, \ldots, n, \tag{4.21}$$

der als Matrix-Vektor-Produkt

$$Z = A\,c \tag{4.22}$$

mit

$$c := \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}^{\mathrm{T}}, \tag{4.23}$$

$$Z := \begin{bmatrix} F(z_1) & \cdots & F(z_n) \end{bmatrix}^{\mathrm{T}} \tag{4.24}$$

und

$$A := \begin{bmatrix} \int_a^{z_1} b_1(x)\,\mathrm{d}x & \cdots & \int_a^{z_1} b_n(x)\,\mathrm{d}x \\ \vdots & & \vdots \\ \int_a^{z_n} b_1(x)\,\mathrm{d}x & \cdots & \int_a^{z_n} b_n(x)\,\mathrm{d}x \end{bmatrix} \tag{4.25}$$

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.

5 Fallstudie: Computertomografie

Die Computertomografie (CT) ist ein weit verbreitetes medizinisches und industrielles Bildgebungsverfahren. Diskutieren hier alle Teilschritte vom Hardware-Aufbau bis zum Lösungsalgorithmus. Schwerpunkt sind das Erkennen von zu lösenden (Teil-)Problemen, die Auswahl geeigneter Lösungsverfahren und deren Implementierung. Nicht betrachtet wird die Effizienz der Algorithmen, da die derzeit effizientesten Verfahren unsere mathematischen Fähigkeiten übersteigen.

5.1 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).

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).

5.2 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

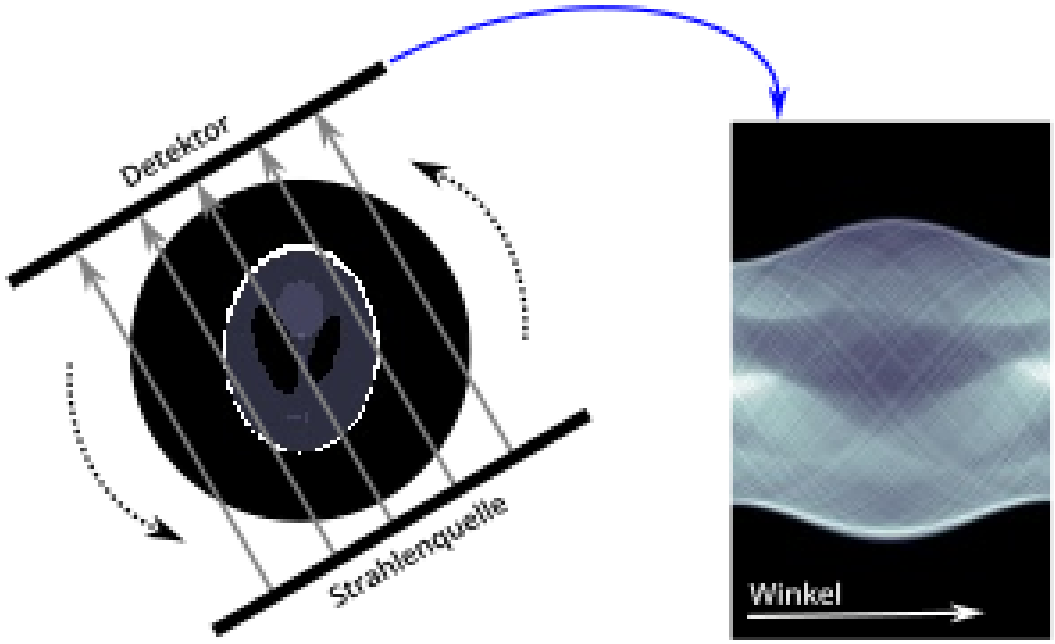


Abbildung 5.1: *
Funktionsprinzip der Computertomografie (links) und ein Sinogramm als Beispiel für die direkt gemessenen Daten, aus denen das Innere des untersuchten Objekts rekonstruiert werden soll (rechts).

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

$$R\,f : [0, 2\,\pi] \times [-1, 1] \to \mathbb{R} \tag{5.1}$$

ab, indem sie f entlang parallel verlaufender Geraden integriert:

$$(R\,f)(\alpha, \sigma) := \int_{-1}^{1} f\left(\sigma \begin{bmatrix} -\sin\alpha \\ \cos\alpha \end{bmatrix} + \tau \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix}\right) \mathrm{d}\tau. \tag{5.2}$$

(IDVID 510, schöne Animation). Bezeichnen wir mit $g(\alpha, \sigma)$ die vom Detekor zum durch (α, σ) beschriebenen Strahlverlauf gemessene Intensität, so gilt

$$g(\alpha, \sigma) = c\,\mathrm{e}^{-(R\,f)(\alpha,\sigma)}, \tag{5.3}$$

wobei $c > 0$ die Intensität der Röntgenquelle angibt (Lambert-beersches Gesetz). Wir erhalten also

$$-\ln\left(\frac{g(\alpha, \sigma)}{c}\right) = (R\,f)(\alpha, \sigma) \tag{5.4}$$

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 werden wir sehen, dass man aus dem kontinuierlichen Modell Lösungsalgorithmen ableiten kann, die man in einem diskreten Modell nicht sehen würde.

Die Radon-Transformation ist eine lineare Abbildung, kann also nach geeigneter Diskretisierung als Matrix-Vektor-Multiplikation umgesetzt werden.

Die Section 2.1 sind allerdings nur teilweise erfüllt:

- Zu gegebenen Daten g findet man stets eine Lösung f (eine mathematisch sauberere Formulierung übersteigt unsere Möglichkeiten).
- Die Lösung ist eindeutig.

- ABER: Die Lösung reagiert sehr empfindlich auf Änderungen in den Daten (Messfehler/Rauschen!).

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. Wie wir mit dieser Situation umgehen, klären wir später.

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.

5.3 Testdaten

Benötigen Daten (Sinogramm) zum Testen verschiedener Lösungsansätze.

Merke!

Anforderungen an die Testdaten:

- möglichst keine Datenfehler (Rauschen,...),
- in verschiedenen Auflösungen verfügbar,
- Lösung möglichst exakt bekannt.

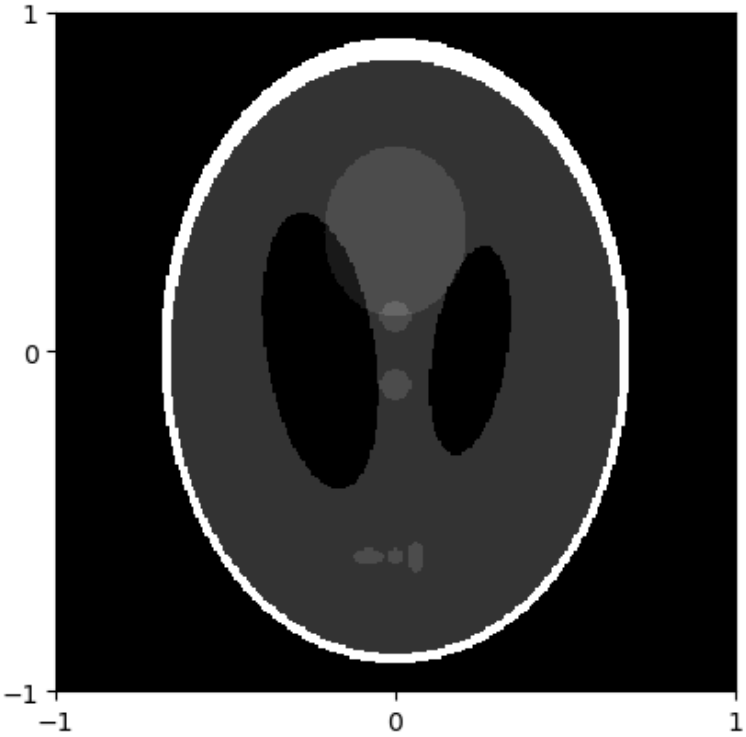
Sind die Testdaten fehlerbehaftet, so ist nicht klar, ob Fehler in der Lösung aus unserem Lösungsansatz (bzw. dessen Implementierung) oder aus den Datenfehlern resultieren. Auch wenn später hochauflösende Daten verwendet werden sollen, sind für die zahlreichen Testläufe niedrig aufgelöste Daten unter Umständen sehr zeitsparend. Ohne Kenntnis der exakten Lösung können wir nicht beurteilen, ob unser Verfahren korrekt arbeitet.

Die obigen Anforderungen sind praktisch nur mit synthetischen Daten erfüllbar. Für die Computertomografie verwendet man gern das so genannte Shepp-Logan-Phantom, welches in [SL74] eingeführt wurde. Dieses besteht aus einer Überlagerung von zehn verschieden großen Ellipsen. Der folgende Code erzeugt ein Rasterbild des Shepp-Logan-Phantoms aus den Ellipsen-Definitionen (IDVID 515). Die in der ursprünglichen Arbeit verwendeten Grauwerte haben zu geringen Kontrast für eine gut erkennbare Darstellung, sodass wir diese ersetzen mit den in The Radon Transform - Theory and Implementation (Toft, 1996) verwendeten Werten.

```
import numpy as np
```

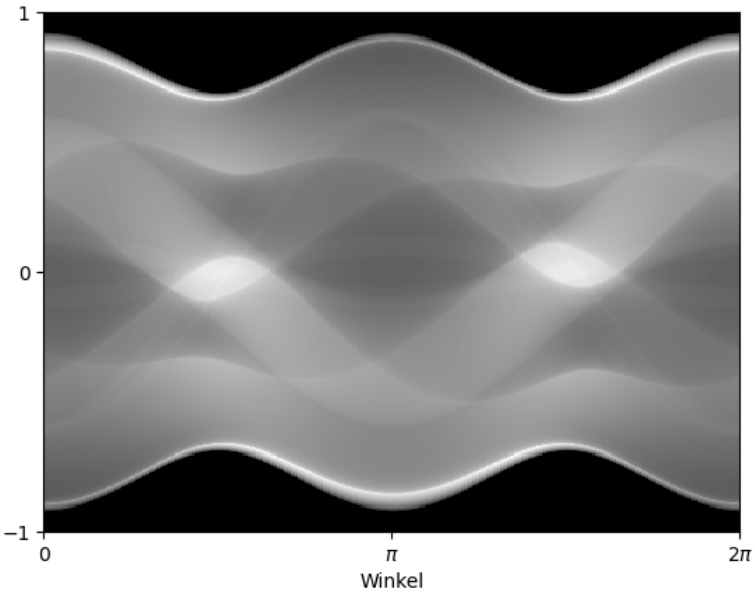
```
import matplotlib.pyplot as plt

def ellipse_dict(x, y, angle, axis1, axis2, value):
    return dict(x=x, y=y, angle=angle, axis1=axis1, axis2=axis2,
    ↪  value=value)

shepp_logan_data = [
    ellipse_dict( 0   ,  0     , 0                , 0.69  , 0.92 ,  1
    ↪  ),
    ellipse_dict( 0   , -0.0184, 0                , 0.6624, 0.874,
    ↪  -0.8),
    ellipse_dict( 0.22,  0     , -18 * np.pi / 360, 0.11  , 0.31 ,
    ↪  -0.2),
    ellipse_dict( -0.22,  0     ,  18 * np.pi / 360, 0.16  , 0.41 ,
    ↪  -0.2),
    ellipse_dict( 0   ,  0.35  , 0                , 0.21  , 0.25 ,
    ↪  0.1),
    ellipse_dict( 0   ,  0.1   , 0                , 0.046 , 0.046,
    ↪  0.1),
    ellipse_dict( 0   , -0.1   , 0                , 0.046 , 0.046,
    ↪  0.1),
    ellipse_dict( -0.08, -0.605 , 0                , 0.046 , 0.023,
    ↪  0.1),
    ellipse_dict( 0   , -0.605 , 0                , 0.023 , 0.023,
    ↪  0.1),
    ellipse_dict( 0.06, -0.605 , 0                , 0.023 , 0.046,
    ↪  0.1)
]

def shepp_logan_image(n):
    img = np.zeros((n, n), dtype=float)
    pixel_centers = np.linspace( -1 + 1/n, 1 - 1/n, n)
    grid_x, grid_y = np.meshgrid(pixel_centers, -pixel_centers)
    for e in shepp_logan_data:
        S_inv = np.array([  # inverse Skalierungsmatrix
            [1 / e['axis1'], 0             ],
            [0,              1 / e['axis2']]
        ])
        R_inv = np.array([  # inverse Rotationsmatrix
            [ np.cos(e['angle']), np.sin(e['angle'])],
            [ -np.sin(e['angle']), np.cos(e['angle'])]
        ])
        A = S_inv @ R_inv
        x = grid_x - e['x']
```

```
        y = grid_y - e['y']
        f = (A[0, 0] * x + A[0, 1] * y) ** 2 + (A[1, 0] * x + A[1, 1] *
        ↪  y) ** 2
        img[f < 1] += e['value']
    return img

fig, ax = plt.subplots()
ax.imshow(
    shepp_logan_image(300),
    cmap='gray',
    interpolation='none',
    extent=( -1, 1, -1, 1)
)
ax.set_xticks([ -1, 0, 1])
ax.set_yticks([ -1, 0, 1])
plt.show()
```



Wesentlicher Grund für die breite Verwendung des Shepp-Logan-Phantoms ist, dass man das Sinogramm in beliebiger Auflösung exakt berechnen kann. Auch für einige andere einfache geometrische Objekte neben Ellipsen ist die Radon-Transformation analytisch auswertbar. Die sich ergebenden Formeln findet man in The Radon Transform - Theory and Implementation (Toft, 1996) ab Seite 199.

```
def shepp_logan_sinogram(angles, n_pixels):
    n_angles = len(angles)
    sino = np.zeros((n_pixels, n_angles), dtype=float)
    alpha, sigma = np.meshgrid(
        angles + 0.5 * np.pi,  # Winkel \theta in Toft 1996 (Seite 24)
        np.linspace(1 - 1/n_pixels, -1 + 1/n_pixels, n_pixels)
    )
    for e in shepp_logan_data:
        # Berechnungen aus Toft 1996 (Seite 200) entnommen
        tmp0 = e['axis1'] ** 2 * np.cos(alpha - e['angle']) ** 2 \
               + e['axis2'] ** 2 * np.sin(alpha - e['angle']) ** 2
        tmp = (sigma - e['x'] * np.cos(alpha) - e['y'] * np.sin(alpha))
        ↪  ** 2
        sino += 2 * e['value'] * e['axis1'] * e['axis2'] \
                * 1/tmp0 * np.sqrt(np.maximum(0, tmp0 - tmp))
    return sino

fig, ax = plt.subplots()
ax.imshow(
    shepp_logan_sinogram(np.linspace(0, 2 * np.pi, 300,
    ↪  endpoint=False), 360),
    cmap='gray',
    interpolation='none',
    extent=(0, 2 * np.pi, -1, 1),
    aspect='auto'
)
ax.set_xlabel('Winkel')
ax.set_xticks([0, np.pi, 2 * np.pi])
ax.set_xticklabels(['0', '$\\pi$', '$2\\pi$'])
ax.set_yticks([ -1, 0, 1])
plt.show()
```



Aufgrund der Symmetrie kann das Objekt bereits aus den Daten für den Winkelbereich $[0, \pi)$ rekonstruiert werden. Genauer gilt:

$$(R\,f)(\alpha, \sigma) = (R\,f)(\alpha + \pi, -\sigma). \tag{5.5}$$

Vorerst arbeiten wir mit (bis auf Rundungsfehler) exakten Daten. Später, wenn das prinzipielle Vorgehen zur Rekonstruktion geklärt ist, werden wir die Testdaten realitätsnäher gestalten.

5.4 Lösungsansätze

Betrachten hier kurz drei grundlegend verschiedene Lösungsansätze. Für die ersten beiden belassen wir es bei einem Proof-of-Concept. Den dritten werden wir im Anschluss detaillierter betrachten.

Merke!

Prinzipielles Unterscheidungsmerkmal bei allen Lösungsverfahren im wissenschaftlichen Rechnen ist der Zeitpunkt des Diskretisierens:

- **erst invertieren, dann diskretisieren** (man löst das Problem im Wesentlichen analytisch und diskretisiert das gelöste Problem) oder
- **erst diskretisieren, dann invertieren** (man diskretisiert das Modell und löst anschließend das diskretisierte Problem).

Beide Ansätze haben Vor- und Nachteile. Für spätes Diskretisieren spricht:

- Analytisches Lösen liefert meist effizientere Verfahren, mehr Einsichten in das Problem und bessere (oder überhaupt erst) Fehlerabschätzungen.
- Das Diskretisieren beschränkt sich auf einfache Teilprobleme, die mit Standardansätzen diskretisiert werden können.

Für direktes Diskretisieren des Modells spricht:

- Analytisches Lösen ist nicht immer möglich.
- Weniger Arbeitsaufwand und weniger mathematische Fähigkeiten nötig.

Die ersten beiden im Folgenden vorgestellten Lösungsansätze diskretisieren spät. Der dritte Ansatz diskretisiert hingegen das Modell direkt und löst dann das diskrete Problem.

Direktes Fourier-Verfahren

Die Radon-Transformation ist eng mit der kontinuierlichen Fourier-Transformation verbunden.

Nebenbemerkung

Fourier-Transformationen sind üblicherweise nicht Thema der Mathematikgrundausbildung (außer in Physik und Elektrotechnik). Grundidee: Jede Funktion lässt sich als Überlagerung von Schwingungen unterschiedlicher Frequenz darstellen. Die Fourier-Transformierte einer Funktion ist wieder eine Funktion, die aber jeder Frequenz den Anteil der entsprechenden Schwingung in der ursprünglichen Funktion zuordnet. Die Umkehrung der Fourier-Transformation heißt inverse Fourier-Transformation. Die Fourier-Transformation kann als Integral über einen komplexwertigen Integrand dargstellt werden.

Es gibt mehrere verschiedene Arten von Fourier-Transformationen, die in der Literatur leider nicht immer klar unterschieden werden (IDVID 520).

Der Zentralschnitt-Satz besagt, dass man aus den Fourier-Transformationen der einzelnen Projektionen die (zweidimensionale) Fourier-Transformation des Objekts erhält (IDVID 525).

Für das Invertieren der Radon-Transformation ergibt sich daraus folgender Ablauf:

1. Fourier-Transformation für jede Projektion berechnen.
2. Fourier-Transformierte des Objekts aus den transformierten Projektionen zusammensetzen.

3. Inverse (zweidimensionale) Fourier-Transformation anwenden.

Theoretisch erhält man so recht einfach die Lösung. Praktisch gibt es aber zwei wesentliche Probleme:

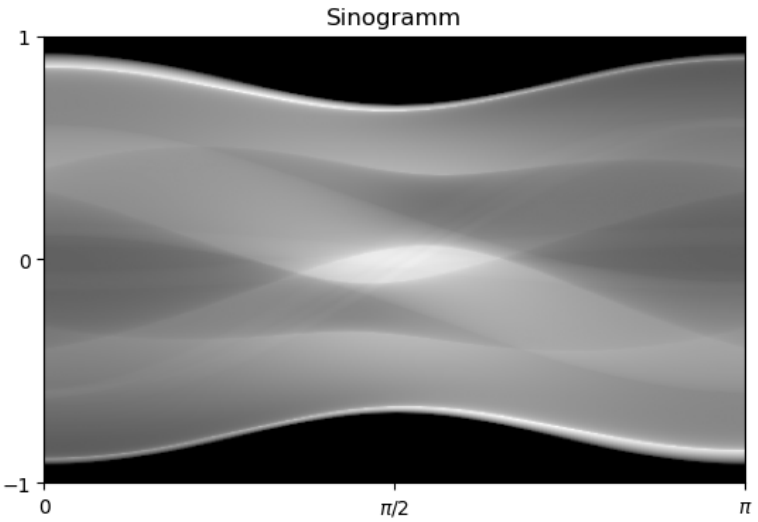
- Software-Bibliotheken für das numerische Rechnen (z.B. NumPy, SciPy) bieten meist nur Implementierungen der Fourier-Transformation für periodische Funktionen, welche mittels schneller Fourier-Transformation sehr effizient berechnet werden kann. Die numerische Berechnung der kontinuierlichen Fourier-Transformation ist mit Blick auf Implementierungs- und Rechenaufwand deutlich anspruchsvoller.
- Die diskretisierte Version von Schritt 2 im obigen Ablauf liefert die Fourier-Transformierte des Objekts auf einem polaren Gitter. Für die inverse Fourier-Transformation in Schritt 3 werden aber Daten auf einem Rechteckgitter benötigt (IDVID 530). Also muss interpoliert werden, was zu verhältnismäßig großen Fehlern im Ergebnis führt. Ein möglicher Ausweg ist das Verwenden der noch recht neuen Schnellen Fourier-Transformation auf nichtäquidistanten Stützstellen.

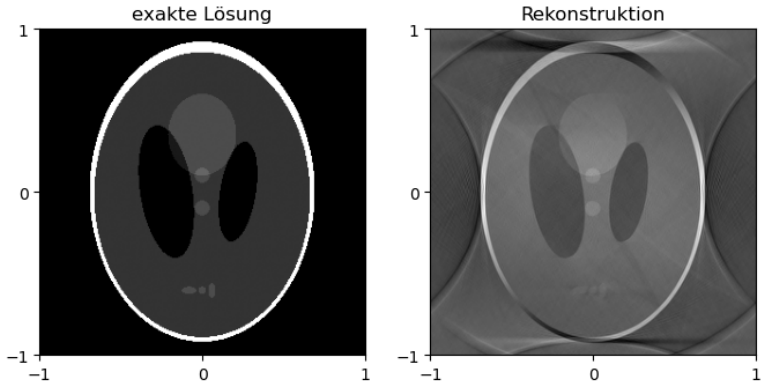
Die folgende Implementierung dient nur als "Proof-of-Concept". Sie löst die beiden genannten Probleme sehr unsauber durch Verwenden stückweise linearer Interpolation und der schnellen Fourier-Transformation auf etwas umstrukturierten Daten. Für Produktiv-Code sollte glattere Interpolation und eine saubere Diskretisierung der kontinuierlichen Fourier-Transformation implementiert werden, was allerdings unsere mathematischen Fähigkeiten übersteigt.

```
import scipy.interpolate

n_angles = 180
n_sino_pixels = 300
n_img_pixels = 300  # muss gleich n_sino_pixels sein, sonst
↪  Rekonstruktion
                    # gezoomt (wegen schlechter Implementierung der F
                    ↪  -Trafos)

# Sinogramm
angles = np.linspace(0, np.pi, n_angles, endpoint=False)
sino = shepp_logan_sinogram(angles, n_sino_pixels)

# Fourier -Transformation der einzelnen Projektionen
sino_fourier = np.fft.fftshift(
    np.fft.fft(
        np.fft.ifftshift(
            sino - sino.mean(axis=0),
            axes=(0,)
        ),
        axis=0
```

```
    ),
    axes=(0,)
)

# polares Gitter (an diesen Stellen kennen wir die 2D -F
↪  -Transformierte)
polar_grid_phi, polar_grid_r = np.meshgrid(
    angles + 0.5 * np.pi,  # Winkel der Projektions"fläche" bzgl. x
    ↪  -Achse
    np.linspace(1 - 1/n_sino_pixels, -1 + 1/n_sino_pixels,
    ↪  n_sino_pixels)
)

# Rechteckgitter (an diesen Stellen benötigen wir die 2D -F
↪  -Transformierte)
rect_grid_x, rect_grid_y = np.meshgrid(
    np.linspace( -1 + 1/n_img_pixels, 1 - 1/n_img_pixels,
    ↪  n_img_pixels),
    np.linspace(1 - 1/n_img_pixels, -1 + 1/n_img_pixels, n_img_pixels)
)

# Interpolation
interpolator = scipy.interpolate.LinearNDInterpolator(
    np.hstack((
        (polar_grid_r * np.cos(polar_grid_phi)).reshape( -1, 1),
        (polar_grid_r * np.sin(polar_grid_phi)).reshape( -1, 1)
    )),
    sino_fourier.reshape( -1),
    fill_value=0
)
sin_fourier_rect = interpolator(rect_grid_x, rect_grid_y)

# inverse 2D -Fourier -Transformation
img = np.fft.fftshift(
    np.fft.ifft2(
        np.fft.ifftshift(
            sin_fourier_rect
        )
    )
).real  # sollte theoretisch reell sein, ist es aber aufgrund von
        # Fehlern (Diskretisierung,...) nicht

# exakte Lösung zum Vergleich
img_exact = shepp_logan_image(n_img_pixels)
```

```
fig, ax = plt.subplots()
ax.imshow(sino, cmap='gray', extent=(0, np.pi, -1, 1))
ax.set_title('Sinogramm')
ax.set_xticks([0, 0.5 * np.pi, np.pi])
ax.set_xticklabels(['0', '$\\pi/2$', '$\\pi$'])
ax.set_yticks([ -1, 0, 1])
plt.show()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(img_exact, cmap='gray', extent=( -1, 1, -1, 1))
ax1.set_title('exakte Lösung')
ax1.set_xticks([ -1, 0, 1])
ax1.set_yticks([ -1, 0, 1])
ax2.imshow(img, cmap='gray', extent=( -1, 1, -1, 1))
ax2.set_title('Rekonstruktion')
ax2.set_xticks([ -1, 0, 1])
ax2.set_yticks([ -1, 0, 1])
plt.show()
```





Die in der Rekonstruktion zusätzlich vorhandenen Strukturen werden als **Artefakte** bezeichnet. Sie entstehen aufgrund der Diskretisierung und sind auch bei besserer Implementierung der Fourier-Transformationen nicht ganz vermeidbar. Aufgrund der recht starken Artefakte beim direkten Fourier-Verfahren hat dieses in der Praxis kaum Relevanz.

Gefilterte Rückprojektion

Die gefilterte Rückprojektion verfolgt die gleiche Idee wie das direkte Fourier-Verfahren (Zentralschnitt-Satz), verzichtet aber auf das numerische Invertieren der zweidimensionalen Fourier-Transformation. Stattdessen wird ein Teil der Berechnung analytisch ausgeführt, sodass das Gitterproblem (polar vs. rechteckig) vermieden wird. Als Ergebnis erhält man den folgenden Ablauf:

1. Fourier-Transformation für jede Projektion berechnen.
2. Jede Fourier-transformierte Projektion mit der Funktion $w \mapsto |w|$ multiplizieren ("ramp filter").
3. Inverser Fourier-Transformation auf jede gefilterte und transformierte Projektion anwenden.
4. Gefilterte Projektionen invertieren (liefert "Streifenbilder").
5. Streifenbilder aller Projektionen überlagern/addieren.

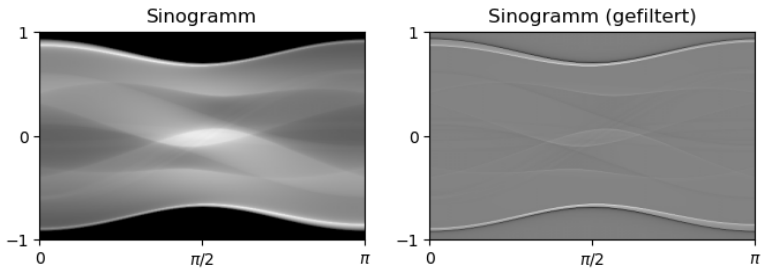
Dieses Vorgehen liefert (theoretisch) die exakte Lösung.

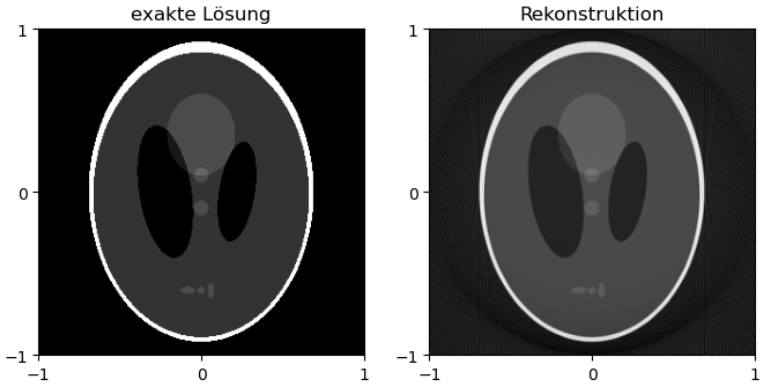
"Proof-of-Concept" mit unsauber diskretisierter Fourier-Transformation:

```
n_angles = 180
n_sino_pixels = 300
n_img_pixels = 300

# Sinogramm
angles = np.linspace(0, np.pi, n_angles, endpoint=False)
```

```
sino = shepp_logan_sinogram(angles, n_sino_pixels)

# Fourier -Transformation der einzelnen Projektionen
sino_fourier = np.fft.fftshift(
    np.fft.fft(
        np.fft.ifftshift(
            sino,
            axes=(0,)
        ),
        axis=0
    ),
    axes=(0,)
)

# Filtern der transformierten Projektionen
sino_pixel_centers = np.linspace(
    -1 + 1/n_sino_pixels, 1 - 1/n_sino_pixels, n_sino_pixels
)
ramp_filter = np.abs(sino_pixel_centers)
sino_fourier_filtered = ramp_filter.reshape( -1, 1) * sino_fourier

# inverse Fourier -Transformation der einzelnen Projektionen
sino_filtered = np.fft.ifftshift(
    np.fft.ifft(
        np.fft.fftshift(
            sino_fourier_filtered,
            axes=(0,)
        ),
        axis=0
    ),
    axes=(0,)
).real  # sollte theoretisch reell sein, ist es aber aufgrund von
        # Fehlern (Diskretisierung,...) nicht

# Rückprojektion
grid_x, grid_y = np.meshgrid(
    np.linspace( -1 + 1/n_img_pixels, 1 - 1/n_img_pixels,
    ↪  n_img_pixels),
    np.linspace(1 - 1/n_img_pixels, -1 + 1/n_img_pixels, n_img_pixels)
)
img = np.zeros((n_img_pixels, n_img_pixels), dtype=float)
for i, angle in enumerate(angles):
    sigma = np.cos(angle + 0.5 * np.pi) * grid_x \
            + np.sin(angle + 0.5 * np.pi) * grid_y
```

```
    img += np.interp(  # Sinogrammspalte an beliebigen Stellen
    ↪  auswerten
        sigma,
        sino_pixel_centers, sino_filtered[:: -1, i],
        left=0, right=0
    )

# exakte Lösung zum Vergleich
img_exact = shepp_logan_image(n_img_pixels)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(sino, cmap='gray', extent=(0, np.pi, -1, 1))
ax1.set_title('Sinogramm')
ax1.set_xticks([0, 0.5 * np.pi, np.pi])
ax1.set_xticklabels(['0', '$\\pi/2$', '$\\pi$'])
ax1.set_yticks([ -1, 0, 1])
ax2.imshow(sino_filtered, cmap='gray', extent=(0, np.pi, -1, 1))
ax2.set_title('Sinogramm (gefiltert)')
ax2.set_xticks([0, 0.5 * np.pi, np.pi])
ax2.set_xticklabels(['0', '$\\pi/2$', '$\\pi$'])
ax2.set_yticks([ -1, 0, 1])
plt.show()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.imshow(img_exact, cmap='gray', extent=( -1, 1, -1, 1))
ax1.set_title('exakte Lösung')
ax1.set_xticks([ -1, 0, 1])
ax1.set_yticks([ -1, 0, 1])
ax2.imshow(img, cmap='gray', extent=( -1, 1, -1, 1))
ax2.set_title('Rekonstruktion')
ax2.set_xticks([ -1, 0, 1])
ax2.set_yticks([ -1, 0, 1])
plt.show()
```





Der Unterschied zwischen gefilterter Rückprojektion und dem direkten Fourier-Verfahren besteht nur in der Frage, ab ein Teil der Integrale in der Formel für die inverse zweidimensionale Fourier-Transformation analytisch oder numerisch berechnet wird! Für das auftretende Doppelintegral wird eine Koordinatentransformation durchgeführt und anschließend das dann innere Integral analytisch berechnet.

Merke!

Teilschritte, die analytisch gelöst werden können, sollten auch analytisch gelöst werden!

Diskretisieren der Radon-Transformation

Ein einfacher Lösungsansatz, der komplexere mathematische Werkzeuge vermeidet und auch bei zahlreichen anderen Aufgaben des wissenschaftlichen Rechnens anwendbar ist, ist das direkte Diskretisieren des "Vorwärtsoperators", hier also der Radon-Transformation. Die Radon-Transformation ist eine lineare Abbildung. Also sollte man erwarten können, dass nach geeigneter Diskretisierung aller beteiligten Größen ein lineares Gleichungssystem entsteht. Dieses kann dann mit Standardverfahren gelöst werden.

Diesen Ansatz werden wir in Kürze detailliert umsetzen. Hier zunächst nur der grobe Ablauf:

1. Wähle eine Basis im Lösungsraum, also eine (endliche) Menge von Funktionen $b_l : \mathbb{R}^2 \to \mathbb{R}$, $l = 1, \ldots, n$, sodass jede Lösung $f : \mathbb{R}^2 \to \mathbb{R}$ näherungsweise als Linearkombination

$$f(x, y) \approx \sum_{l=1}^{n} u_l \, b_l(x, y), \quad (x, y) \in \mathbb{R}^2 \tag{5.6}$$

   dargestellt werden kann.

2. Stelle die Matrix $A \in \mathbb{R}^{m \times n}$ auf, die den Vektor $u = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix}^{\mathrm{T}}$ auf die gegebenen diskreten Sinogrammdaten $v \in \mathbb{R}^m$ abbildet. Dabei ist v der Vektor, der untereinander alle Spalten des Sinogramms enthält.
3. Löse das lineare Gleichungssystem

$$A\,u = v. \tag{5.7}$$

und berechne die zum Lösungsvektor gehörende Linearkombination der Basisfunktionen.

Drei Schwierigkeiten sind zu überwinden:

- **Wie Basis wählen?** Das ist eine Abwägung zwischen Einfachheit und "algorithmischer Raffinesse". Die einfachste Wahl ist die sogenannte Pixelbasis. Jedes b_l repräsentiert genau ein Pixel, hat also den Wert 1 für alle (x, y), die in der Pixelfläche liegen und sonst den Wert Null. Alternativ könnte man z.B. eine Wavelet-Basis wählen, die zusätzliche Möglichkeiten bei der Unterdrückung von (unter Umständen) extremer Fehlerverstärkung im Lösungsprozess bietet (siehe unten) und insbesondere bei hohen Bildauflösungen eine speichereffizientere Darstellung der Bilddaten erlaubt.
- **Wie Matrix aufstellen?** Dies ist eine reine Rechenübung! Wählt man im ersten Schritt die Pixelbasis, so entspricht jede Spalte der Matrix A gerade dem Sinogramm des entsprechenden Pixels (IDVID 540). Müssen also die Radon-Transformierte für jedes einzelne Pixel berechnen.
- **Passt die Matrix in den Speicher?** Schon bei moderater Auflösung von 200 mal 200 Pixel für die Lösung, 200 Pixel pro Sinogrammspalte und 120 Projektionswinkeln hat die Matrix A fast eine Milliarde Einträge (genau: 960000000). Bei Verwendung von 64-Bit-Gleitkommazahlen entspricht dies 8 GB Arbeitsspeicherbedarf! Wird die Auflösung verdoppelt, sind es schon 32 GB. ABER: Die meisten Einträge von A sind Null, da die Spalten gerade die Sinogramme einzelner Pixel sind. Verwendet man spezielle Speicherformate für dünn besetzte Matrizen, so kann die Matrix A problemlos im Speicher abgelegt werden (vgl. auch Veranstaltung zur numerischen Mathematik).

5.5 Umsetzung der direkten Diskretisierung

Setzen den dritten Lösungsansatz nun um.

Radon-Matrix

Die Matrixdarstellung A der Radon-Transformation R entsteht spaltenweise aus den Sinogrammen einzelner Pixel (vgl. oben).

Relevante Pixel Der Aufnahmebereich von Computertomografen ist üblicherweise rund, z.B. die Kreisfläche mit Radius 1 um den Ursprung. Bilddaten im Computer werden hingegen zweckmäßig als rechteckige Anordnung von quadratischen Pixeln dargestellt, z.B. für das um Ursprung zentrierte, achsenparallele Quadrat mit Kantenlänge 2. Entsprechend bleiben einige Pixel außerhalb der Kreisscheibe ungenutzt.

Bei der Implementierung sollten die ungenutzen Pixel keine Auswirkung auf das Sinogramm haben, also die entsprechenden Spalten in der Matrix A auf Null gesetzt werden. Nur die Pixel, die vollständig von der Kreisscheibe mit Radius 1 überdeckt werden, spielen eine Rolle. Ist (x_0, y_0) der Mittelpunkt eines Pixels mit Kantenlänge $2\,a$, so muss also gelten:

$$(|x_0| + a)^2 + (|y_0| + a)^2 \leq 1. \tag{5.8}$$

Radon-Transformation eines Pixels Hat man das Sinogramm eines am Koordinatenursprung zentrierten Pixels zur Hand (siehe unten), so kann man die Sinogramme aller anderen Pixel leicht daraus berechnen, denn für das aus einem Objekt f durch Verschiebung $f_{(x_0,y_0)}(x, y) := f(x - x_0, y - y_0)$ entstandene Objekt $f_{(x_0,y_0)}$ gilt:

$$(R\,f_{(x_0,y_0)})(\alpha, \sigma) = (R\,f)(\alpha, \sigma + x_0\,\sin\alpha - y_0\,\cos\alpha) \tag{5.9}$$

(IDVID 545).

Die Radon-Transformation eines im Koordinatenursprung zentrierten Pixels lässt sich leicht berechnen. Im einfachsten Fall kann man das Pixel als Kreisscheibe betrachten. Dann ist die Radon-Transformierte für alle Projektionswinkel α gleich. Die heute übliche Darstellung von Pixeln als Quadrat erfordert etwas mehr Rechenaufwand, ist aber noch gut machbar. Für ein am Koordinatenursprung zentriertes quadratisches Pixel genügt die Betrachtung des Winkelbereichs $\alpha \in [0, \frac{\pi}{4}]$ und der positiven Hälfte der Detektorpositionen, also $\sigma \in [0, 1]$. Alle anderen Sinogrammwerte ergeben sich aus Symmetrieeigenschaften des Quadrats (IDVID 550):

- Symmetrie des Quadrats an der x-Achse:

$$(R\,f)(2\,\pi - \alpha, \sigma) = (R\,f)(\alpha, \sigma), \tag{5.10}$$

- Rotationssymmetrie des des Quadrats bzgl. Drehungen um Vielfache von $\frac{\pi}{2}$:

$$(R\,f)(\alpha + k\,\frac{\pi}{2}, \sigma) = (R\,f)(\alpha, \sigma), \quad k \in \mathbb{Z}, \tag{5.11}$$

- Symmetrie von Sinogrammen:

$$(R\,f)(\alpha + \pi, -\sigma) = (R\,f)(\alpha, \sigma), \tag{5.12}$$

- Anwenden der beider vorherigen Eigenschaften liefert:

$$(R\,f)(\alpha,-\sigma) = (R\,f)(\alpha,\sigma). \tag{5.13}$$

Für verschobene Objekte $f_{(x_0,y_0)}$ erhält man daraus:

- $(R\,f_{(x_0,y_0)})(2\,\pi-\alpha,\sigma) = (R\,f_{(-x_0,y_0)})(\alpha,\sigma)$,
- $(R\,f_{(x_0,y_0)})(\alpha+\frac{\pi}{2},\sigma) = (R\,f_{(y_0,-x_0)})(\alpha,\sigma)$,
- $(R\,f_{(x_0,y_0)})(\alpha+\pi,\sigma) = (R\,f_{(-x_0,-y_0)})(\alpha,\sigma)$,
- $(R\,f_{(x_0,y_0)})(\alpha+\frac{3\,\pi}{2},\sigma) = (R\,f_{(-y_0,x_0)})(\alpha,\sigma)$,
- $(R\,f_{(x_0,y_0)})(\alpha,-\sigma) = (R\,f_{(-x_0,-y_0)})(\alpha,\sigma)$

(IDVID 553).

Beträgt die Kantenlänge des Pixels $2\,a$, so erhält man für $\alpha \in (0,\frac{\pi}{4}]$ und $\sigma \in [0,1]$

$$(R\,f)(\alpha,\sigma) = \begin{cases} 0, & \text{für } \sigma \geq a\,(\cos\alpha+\sin\alpha), \\ \frac{2\,a}{\cos\alpha}, & \text{für } \sigma \leq a\,(\cos\alpha-\sin\alpha), \\ \frac{a\,(\cos\alpha+\sin\alpha)-\sigma}{(\cos\alpha)\,\sin\alpha}, & \text{sonst} \end{cases} \tag{5.14}$$

sowie für $\alpha = 0$

$$(R\,f)(0,\sigma) = \begin{cases} 2\,a, & \text{für } -a < \sigma \leq a \\ 0, & \text{sonst} \end{cases} \tag{5.15}$$

(IDVID 555).

Nebenbemerkung

Für Vielfache von $\frac{\pi}{2}$ muss man ein Detail beachten: Sieht man die Pixel als Quadrate inklusive Randlinie an, so überlappen die Ränder benachbarter Pixel. Dadurch erscheinen die beiden Projektionspunkte der in Projektionsrichtung verlaufenden Quadratränder jeweils doppelt: einmal im Sinogramm des einen Pixels und einmal im Sinogramm des anderen Pixels. Je nach Anzahl der Stützstellen im Sinogramm bzw. der Pixel im Originalbild können dadurch Diskretisierungsartefakte im Sinogramm entstehen (helle Pixel mit gleichmäßigem Abstand). Deshalb sollte je eine der zwei parallelen Randlinien von jedem Pixel entfernt werden. Bei Winkeln, die keine Vielfachen von $\frac{\pi}{2}$ sind, tritt dieses Problem nicht auf, da nicht entlang der Pixelränder integriert wird. Die Integrationsbereiche schneiden die Pixelränder stets nur in einem Punkt, was keinerlei Einfluss auf die Integrale hat.

Diskretisierung der Projektionswinkel Die Diskretisierung der Projektionswinkel folgt den praktischen Gegebenheiten: Die Aufnahmen werden für eine endliche Anzahl von Winkeln $\alpha_0,\alpha_1,\ldots,\alpha_{n_\alpha-1}$ durchgeführt. Üblicherweise sind diese gleichmäßig über das Intervall $[0,2\,\pi)$ verteilt.

Nebenbemerkung

Da die Implementierung in Python erfolgt, bietet es sich auch in der mathematischen Notation an, die Indizes bei Null starten zu lassen. So können die erhaltenen Formeln leichter in Quellcode übertragen werden.

Möchte man den Fall $\alpha = 0$ vermeiden, kann man die Winkel wie folgt wählen:

$$\alpha_k := \frac{\pi}{n_\alpha} + k\,\frac{2\,\pi}{n_\alpha}, \quad k = 0, 1, \ldots, n_\alpha - 1. \tag{5.16}$$

Diskretisierung der Sinogrammspalten Für gegebenen festen Winkel α müssen wir die Funktion

$$s : [-1, 1] \to \mathbb{R}, \quad s(\sigma) := (R\,g)(\alpha, \sigma) \tag{5.17}$$

durch endlich viele Werte darstellen. Hier ist der Bezug zum Messvorgang nicht mehr so klar wie bei den Winkeln. Besitzt der Computertomograf n_σ gleichmäßig verteilte Detektoren und gehen wir (sehr idealisiert) von linienförmigem Strahlverlauf direkt durch die Mitten der Detektoren aus, so entstehen die Detektormesswerte $s_0, s_1, \ldots, s_{n_\sigma - 1}$ als Funktionsauswertungen

$$s_k = s(\sigma_k) \tag{5.18}$$

mit

$$\sigma_k := 1 - \frac{1}{n_\sigma} - k\,\frac{2}{n_\sigma}, \quad k = 0, 1, \ldots, n_\sigma - 1 \tag{5.19}$$

(IDVID 558).

Nicht-Null-Einträge Die meisten Einträge der Matrix A sind Null und sollen aus Platzgründen nicht im Arbeitsspeicher liegen. Müssen also für jedes Bildpixel und für jeden Winkel α die Indizes k bestimmen, für die $s_k \neq 0$ gilt. Nur diese s_k werden im Arbeitsspeicher abgelegt (vgl. nächsten Abschnitt).

Für $\alpha = 0$ erhalten wir

$$s_k = \begin{cases} 2\,a, & \text{für } k_\mathrm{u} \leq k \leq k_\mathrm{o}, \\ 0, & \text{sonst}, \end{cases} \tag{5.20}$$

wobei

$$k_\mathrm{u} := \left\lceil \frac{n_\sigma}{2}\,\left(1 - \frac{1}{n_\sigma} - y_0 - a\right)\right\rceil \tag{5.21}$$

und

$$k_\mathrm{o} := \left\lceil \frac{n_\sigma}{2}\,\left(1 - \frac{1}{n_\sigma} - y_0 + a\right) - 1\right\rceil. \tag{5.22}$$

Für $\alpha \in (0, \frac{\pi}{4})$ erhalten wir

$$s_k = \begin{cases} \frac{-(x_0-a)\,\sin\alpha+(y_0+a)\,\cos\alpha-1+\frac{1+2k}{n_\sigma}}{(\cos\alpha)\,\sin\alpha}, & \text{für } k_{\mathrm{uu}} \le k \le k_{\mathrm{u}} - 1, \\ \frac{2\,a}{\cos\alpha}, & \text{für } k_{\mathrm{u}} \le k \le k_{\mathrm{o}}, \\ \frac{(x_0+a)\,\sin\alpha-(y_0-a)\,\cos\alpha+1-\frac{1+2k}{n_\sigma}}{(\cos\alpha)\,\sin\alpha}, & \text{für } k_{\mathrm{o}} + 1 \le k \le k_{\mathrm{oo}}, \\ 0, & \text{sonst} \end{cases} \tag{5.23}$$

wobei

$$k_{\mathrm{u}} := \left\lceil \frac{n_\sigma}{2} \left(1 - \frac{1}{n_\sigma} + (x_0 + a)\,\sin\alpha - (y_0 + a)\,\cos\alpha \right) \right\rceil, \tag{5.24}$$

$$k_{\mathrm{o}} := \left\lfloor \frac{n_\sigma}{2} \left(1 - \frac{1}{n_\sigma} + (x_0 - a)\,\sin\alpha - (y_0 - a)\,\cos\alpha \right) \right\rfloor, \tag{5.25}$$

$$k_{\mathrm{uu}} := \left\lfloor \frac{n_\sigma}{2} \left(1 - \frac{1}{n_\sigma} + (x_0 - a)\,\sin\alpha - (y_0 + a)\,\cos\alpha \right) + 1 \right\rfloor, \tag{5.26}$$

$$k_{\mathrm{oo}} := \left\lceil \frac{n_\sigma}{2} \left(1 - \frac{1}{n_\sigma} + (x_0 + a)\,\sin\alpha - (y_0 - a)\,\cos\alpha \right) - 1 \right\rceil \tag{5.27}$$

(IDVID 560).

Nebenbemerkung

Es kann vorkommen, dass $k_{\mathrm{u}} > k_{\mathrm{o}}$ gilt, z.B. wenn die Bildpixel sehr klein und die Sinogrammpixel sehr groß sind. Dann liegt zwischen den ungerundeten Werten für k_{u} und k_{o} keine Ganzzahl, sodass Aufrunden von k_{u} und Abrunden von k_{o} zu dieser unerwarteten Reihenfolge der beiden Werte führt. Dies ist bei der Implementierung zu beachten!

Implementierung Dünn besetzte Matrizen werden von NumPy nicht unterstützt, aber von SciPy. Es gibt verschiedene Typen, die sich in Implementierungsdetails unterscheiden und für manche Zwecke besser als für andere geeignet sind. Wir wollen im Wesenlichen Matrix-Vektor-Produkte berechnen, was mit dem Typ `csr_array` effizient möglich ist. Die Nicht-Null-Einträge der Matrix sind dem Konstruktor als Listen der Einträge, der Zeilenindizes und der Spaltenindizes zu übergeben.

Der folgende Code kann mit Blick auf die Rechenzeit noch optimiert werden, verzichet zum Zwecke der leichteren Übertragbarkeit in andere Sprachen aber auf übermäßige Verwendung von Python- und NumPy-Features wie Broadcasting und boolsches Indizieren.

```
import scipy.sparse as ss

def radon_matrix(n_img_pixels, n_sino_pixels, n_angles):
```

```
    # Pixelanzahlen günstig wählen für einfachere Implementierung
    if n_img_pixels % 2 != 0:
        print('Bildbreite bzw. -höhe muss durch 2 teilbar sein!')
        return None
    if n_angles % 8 != 0:
        print('Winkelanzahl muss durch 8 teilbar sein!')
        return None
    if n_sino_pixels % 2 != 0:
        print('Pixelanzahl in den Sinogrammspalten muss durch 2 teilbar
        ↪  sein!')
        return None

    # Listen für Nicht -Null -Einträge der Matrix A
    Adata = []
    Acols = []
    Arows = []

    # Funktion zum Schreiben einer Sinogrammspalte in die Matrix A
    def set_sino_col(kmin, kmax, s_nonzero, flip_s,
                     sino_col, col, flip_col, row, flip_row):
        # Vorzeichen vor x -Koordinate des Bildpixels drehen?
        if flip_col:
            col = n_img_pixels - 1 - col
        # Vorzeichen vor y -Koordinate des Bildpixels drehen?
        if flip_row:
            row = n_img_pixels - 1 - row
        # Sinogrammspalte am Ursprung spiegeln?
        if flip_s:
            kmin, kmax = n_sino_pixels - 1 - kmax, n_sino_pixels - kmin
            ↪  - 1
            s_nonzero = s_nonzero[:: -1]
        # Position der Sinogrammspalte in A berechnen
        Acol = col * n_img_pixels + row
        Arow = sino_col * n_sino_pixels
        # Daten in A schreiben
        Adata.extend(list(s_nonzero))
        Acols.extend((kmax - kmin + 1) * [Acol])
        Arows.extend(list(range(Arow + kmin, Arow + kmax + 1)))

    # Sinogramme erstellen
    # (Pixel (x, y) und ( -x, -y) haben gerade gespiegelte
    ↪  Sinogrammspalten,
    # sodass nur die Hälfte der Pixel durchlaufen werden muss)
    a = 1 / n_img_pixels  # halbe Pixelbreite
```

```
n_angles4 = n_angles // 4  # entspricht Winkel pi/2
all_x0 = np.linspace( -1 + a, 1 - a, n_img_pixels)
all_y0 = np.linspace(1 - a, 0 + a, n_img_pixels // 2)
angles = np.linspace(
    np.pi / n_angles, 0.25 * np.pi - np.pi / n_angles, n_angles //
    ↪ 8
)
s = np.empty(n_sino_pixels, dtype=float)  # temporäre
↪ Sinogrammspalte
for (img_col, x0) in enumerate(all_x0):
    for (img_row, y0) in enumerate(all_y0):
        # nur Pixel innerhalb der Kreisscheibe!
        if (np.abs(x0) + a) ** 2 + (np.abs(y0) + a) ** 2 > 1:
            continue

        # Winkel zwisch 0 und pi/4 (Rest mittels Spiegeln und
        ↪ Verschieben)
        for (i_alpha, alpha) in enumerate(angles):
            cos = np.cos(alpha)
            sin = np.sin(alpha)

            # Index -Range der Nicht -Null -Einträge (zunächst
            ↪ ungültige Werte)
            kmin, kmax = n_sino_pixels, -1

            # Index -Ranges der verschiedenen Abschnitte der
            ↪ Sinogrammspalte
            c = 1 - 1 / n_sino_pixels  # zur Vereinfachung der 4
            ↪ Formeln
            kuu = int(np.floor(
                n_sino_pixels / 2 * (c + (x0 - a) * sin - (y0 + a)
                ↪ * cos) + 1
            ))
            ku = int(np.ceil(
                n_sino_pixels / 2 * (c + (x0 + a) * sin - (y0 + a)
                ↪ * cos)
            ))
            ko = int(np.floor(
                n_sino_pixels / 2 * (c + (x0 - a) * sin - (y0 - a)
                ↪ * cos)
            ))
            koo = int(np.ceil(
                n_sino_pixels / 2 * (c + (x0 + a) * sin - (y0 - a)
                ↪ * cos) - 1
```

```
        ))

        # Sinogrammwerte für jeden Abschnitt setzen
        if kuu < ku:
            c = 1 - (1 + 2 * np.arange(kuu, ku)) /
            ↪  n_sino_pixels
            s[kuu:ku] = ( -(x0 - a) * sin + (y0 + a) * cos - c)
            ↪  / (cos * sin)
            kmin, kmax = kuu, ku - 1
        if ku <= ko:
            s[ku:(ko + 1)] = 2 * a / cos
            kmin, kmax = np.minimum(kmin, ku), np.maximum(kmax,
            ↪  ko)
        if ko < koo:
            c = 1 - (1 + 2 * np.arange(ko + 1, koo + 1)) /
            ↪  n_sino_pixels
            s[(ko + 1):(koo + 1)] \
                = ((x0 + a) * sin - (y0 - a) * cos + c) / (cos
                ↪  * sin)
            kmin, kmax = np.minimum(kmin, ko + 1),
            ↪  np.maximum(kmax, koo)
        s_nonzero = s[kmin:(kmax + 1)]

        # 0...0.25pi
        set_sino_col(
            kmin, kmax, s_nonzero, False, i_alpha,
            img_col, False, img_row, False
        )
        set_sino_col(
            kmin, kmax, s_nonzero, True, i_alpha,
            img_col, True, img_row, True
        )

        # 0.25pi...0.5pi
        set_sino_col(
            kmin, kmax, s_nonzero, False, n_angles4 - 1 -
            ↪  i_alpha,
            img_row, False, img_col, False
        )
        set_sino_col(
            kmin, kmax, s_nonzero, True, n_angles4 - 1 -
            ↪  i_alpha,
            img_row, True, img_col, True
        )
```

```
# 0.5pi...0.75pi
set_sino_col(
    kmin, kmax, s_nonzero, False, n_angles4 + i_alpha,
    img_row, False, img_col, True
)
set_sino_col(
    kmin, kmax, s_nonzero, True, n_angles4 + i_alpha,
    img_row, True, img_col, False
)

# 0.75pi...pi
set_sino_col(
    kmin, kmax, s_nonzero, False, 2 * n_angles4 - 1 -
    ↪ i_alpha,
    img_col, False, img_row, True
)
set_sino_col(
    kmin, kmax, s_nonzero, True, 2 * n_angles4 - 1 -
    ↪ i_alpha,
    img_col, True, img_row, False
)

# pi...1.25pi
set_sino_col(
    kmin, kmax, s_nonzero, False, 2 * n_angles4 +
    ↪ i_alpha,
    img_col, True, img_row, True
)
set_sino_col(
    kmin, kmax, s_nonzero, True, 2 * n_angles4 +
    ↪ i_alpha,
    img_col, False, img_row, False
)

# 1.25pi...1.5pi
set_sino_col(
    kmin, kmax, s_nonzero, False, 3 * n_angles4 - 1 -
    ↪ i_alpha,
    img_row, True, img_col, True
)
set_sino_col(
    kmin, kmax, s_nonzero, True, 3 * n_angles4 - 1 -
    ↪ i_alpha,
```

```
                    img_row, False, img_col, False
                )

                # 1.5pi...1.75pi
                set_sino_col(
                    kmin, kmax, s_nonzero, False, 3 * n_angles4 +
                    ↪  i_alpha,
                    img_row, True, img_col, False
                )
                set_sino_col(
                    kmin, kmax, s_nonzero, True, 3 * n_angles4 +
                    ↪  i_alpha,
                    img_row, False, img_col, True
                )

                # 1.75pi...2pi
                set_sino_col(
                    kmin, kmax, s_nonzero, False, n_angles - 1 -
                    ↪  i_alpha,
                    img_col, True, img_row, False
                )
                set_sino_col(
                    kmin, kmax, s_nonzero, True, n_angles - 1 -
                    ↪  i_alpha,
                    img_col, False, img_row, True
                )

    # Matrix aus Listen erstellen
    A = ss.csr_array(
        (Adata, (Arows, Acols)),
        shape=(n_sino_pixels * n_angles, n_img_pixels ** 2)
    )

    return A

# Matrix erstellen
n_sino_pixels = 100
n_angles = 200
n_img_pixels = int(np.ceil(np.sqrt(2 * n_sino_pixels * n_angles /
↪  np.pi) / 2) * 2)
print(f'Bildpixel: {n_img_pixels} x {n_img_pixels}')
A = radon_matrix(n_img_pixels, n_sino_pixels, n_angles)

# Anzahl Nicht -Null -Einträge
```

```
size = A.shape[0] * A.shape[1]
nonzeros = len(A.nonzero()[0])
print(f'Einträge in A gesamt: {size:10}')
print(f'Nicht -Null -Einträge:  {nonzeros:10}')
print(f'Besetzungsdichte:     {nonzeros / size * 100:9.2f}%')
```

```
Bildpixel: 114 x 114
```

```
Einträge in A gesamt:  259920000
Nicht -Null -Einträge:     2225296
Besetzungsdichte:          0.86%
```

Nebenbemerkung

Als grobe Faustregel für die Anzahl n_{Bild} der Bildpixel kann man

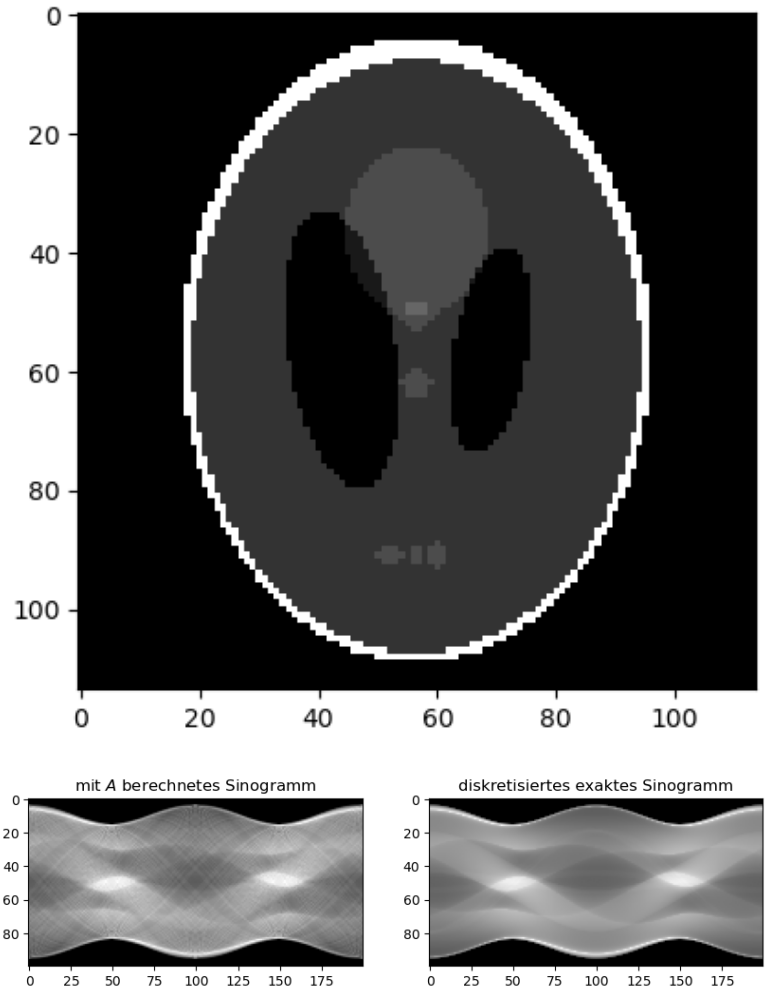
$$n_{\text{Bild}} \approx \sqrt{\frac{2\, n_\alpha\, n_\sigma}{\pi}} \tag{5.28}$$

nutzen. Diese entsteht aus der Annahme, dass die Anzahl der Bildpixel innerhalb der Kreisscheibe etwa so groß sein soll wie die Hälfte der Anzahl der Sinogrammpixel, da das halbe Sinogramm bereits die vollständige Information über das Objekt enthält (IDVID 563).

Zum Test wenden wir die Matrix A auf das diskretisierte Shepp-Logan-Phantom an.

```
img_exact = shepp_logan_image(n_img_pixels)

angles = np.linspace(np.pi / n_angles, 2 * np.pi - np.pi / n_angles,
↪  n_angles)
exact_sino = shepp_logan_sinogram(angles, n_sino_pixels)

u_exact = img_exact.T.reshape( -1)
v = A @ u_exact
sino = v.reshape(n_angles, n_sino_pixels).T

fig, ax = plt.subplots()
ax.imshow(img_exact, cmap='gray')
plt.show()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
ax1.imshow(sino, cmap='gray')
ax2.imshow(exact_sino, cmap='gray')
```

```
ax1.set_title('mit $A$ berechnetes Sinogramm')
ax2.set_title('diskretisiertes exaktes Sinogramm')
plt.show()
```



Die Abweichungen zwischen den beiden Sinogrammen entstehen durch die unterschiedlichen Ansätze:

- links: erst diskretisieren, dann Sinogramm berechnen,
- rechts: erst Sinogramm berechnen, dann diskretisieren.

Nebenbemerkung

Der Rechenaufwand für das Erstellen der Matrix A ist relativ hoch. Praktisch spielt dieser aber keine Rolle, da die Matrix A nur einmalig (pro Computertomograf) erstellt werden muss und dann in einer Datei abgelegt werden kann.

Lösen des Gleichungssystems

Die Matrix A ist im Allgemeinen nicht quadratisch, also nicht invertierbar. Wir können das Gleichungssystem also nicht direkt lösen. Einziger Ausweg ist die Suche nach einem Vektor u, sodass $A\,u$ möglichst nah am gegebenen Datenvektor v liegt. Zu lösen ist also das Minimierungsproblem

$$\|A\,u - v\|^2 \to \min_{u \in \mathbb{R}^n} . \tag{5.29}$$

Dabei ist

$$\|w\| := \sqrt{\sum_{k=1}^{n} w_k^2} \tag{5.30}$$

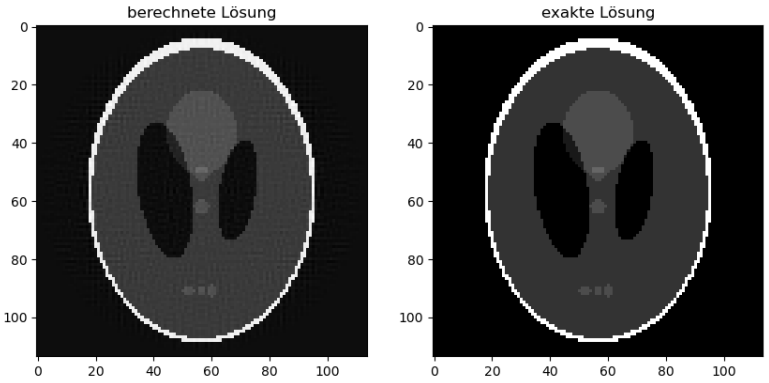
die übliche euklidische Norm eines Vektors w. Dieser Ansatz ist eng verwandt mit der Methode der kleinsten Quadrate (vgl. Veranstaltung zur Numerik). Insbesondere gibt es stets eine eindeutige Lösung und diese kann aus dem linearen Gleichungssystem

$$A^{\mathrm{T}}\,A\,u = A^{\mathrm{T}}\,v \tag{5.31}$$

bestimmt werden, welches eine invertierbare quadratische Systemmatrix besitzt, also eindeutig lösbar ist.

Zum Lösen des Gleichungssystems sollte ein Algorithmus verwendet werden, der speziell für dünn besetzte Matrizen entwickelt wurde (vgl. Veranstaltung zur Numerik). Geeignet ist z.B. `scipy.sparse.linalg.lsqr`.

```
u = scipy.sparse.linalg.lsqr(A, v, iter_lim=1000)[0]

img = u.reshape(n_img_pixels, n_img_pixels).T

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
ax1.imshow(img, cmap='gray')
ax2.imshow(img_exact, cmap='gray')
ax1.set_title('berechnete Lösung')
ax2.set_title('exakte Lösung')
plt.show()
```



Realitätsnahe Daten

Um die Arbeit mit synthetischen Testdaten realitätsnäher zu gestalten, sollten zwei wichtige Regeln beachtet werden:

- Zum Erzeugen der Daten nicht das diskretisierte Modell (hier die Matrix A) verwenden, sondern die Daten auf vom diskretisierten Lösungsprozess unabhängigem Weg erzeugen. Falls es keinen anderen Weg gibt, dann muss wenigstens eine viel feinere Diskretisierung für die Datenerzeugung verwendet werden. Die eigentlichen Daten entstehen dann durch (vom Lösungsprozess unabhängiges) Down-Sampling.
- Die erzeugten Daten sollten mit Rauschen überlagert werden, da dieses bei Realdaten immer auftritt. Nur so können die Auswirkung von eventuellen Instabilitäten des gewählten Lösungsverfahrens (extreme Verstärkung von Datenfehlern) auch an synthetischen Daten beobachtet werden.

Für das Shepp-Logan-Phantom können wir die Daten unabhängig von A erzeugen. Um verrauschte Daten mit einem relativen Datenfehler $\delta_{\text{rel}} > 0$ zu erhalten, muss ein zufälliger Vektor $\Delta \in \mathbb{R}^n$ erzeugt werden. Setzt man damit

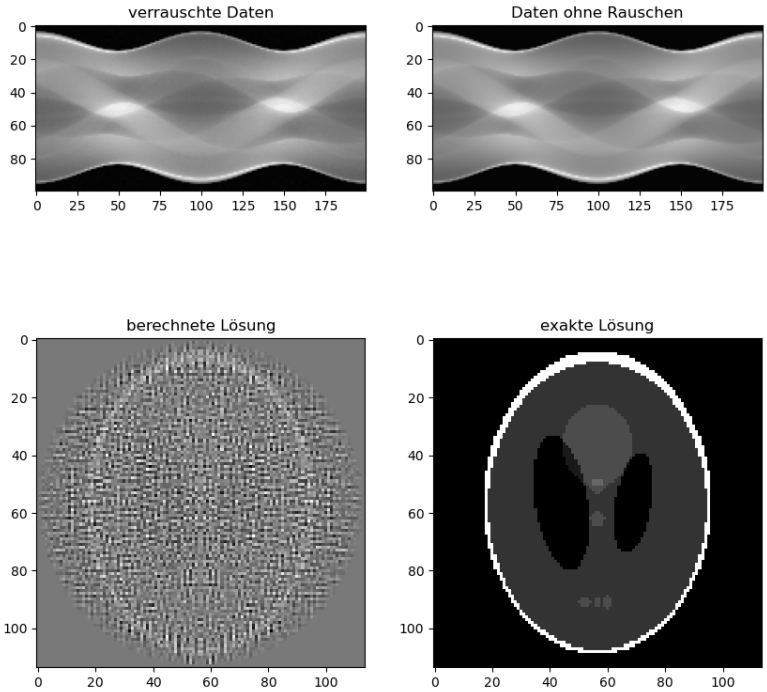
$$\tilde{v} := v + \delta_{\text{rel}} \frac{\|v\|}{\|\Delta\|} \Delta, \tag{5.32}$$

so gilt

$$\frac{\|v - \tilde{v}\|}{\|v\|} = \delta_{\text{rel}}. \tag{5.33}$$

```
# Daten nicht mit A erzeugen!
v = shepp_logan_sinogram(angles, n_sino_pixels).T.reshape( -1)

# 1 Prozent normalverteiltes Rauschen
noise_level = 0.01  # relatives Rauschniveau
rng = np.random.default_rng(0)  # Zufallszahlengenerator
noise = rng.normal(size=v.shape)
v_noisy = v + noise_level * np.linalg.norm(v) / np.linalg.norm(noise) *
↪  noise

u_noisy = scipy.sparse.linalg.lsqr(A, v_noisy, iter_lim=1000)[0]
img_noisy = u_noisy.reshape(n_img_pixels, n_img_pixels).T

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10, 10))

vmin = np.minimum(v_noisy.min(), v.min())
vmax = np.maximum(v_noisy.max(), v.max())
ax1.imshow(v_noisy.reshape(n_angles, n_sino_pixels).T, cmap='gray',
↪  vmin=vmin, vmax=vmax)
```

```
ax2.imshow(v.reshape(n_angles, n_sino_pixels).T, cmap='gray',
↪  vmin=vmin, vmax=vmax)
ax1.set_title('verrauschte Daten')
ax2.set_title('Daten ohne Rauschen')

ax3.imshow(img_noisy, cmap='gray')
ax4.imshow(img_exact, cmap='gray')
ax3.set_title('berechnete Lösung')
ax4.set_title('exakte Lösung')

plt.show()
```



Das Invertieren der Radon-Transformation reagiert extrem empfindlich auf Änderungen in den Daten! Selbst sehr kleine Änderungen können schon zu extremen Abweichungen in der Lösung führen. Die dritte Hadamard-Bedingung ist also nicht erfüllt.

Regularisierung

Um trotz der beobachteten Instabilität akzeptable Lösungen aus verrauschten Daten zu bekommen kann man so genannte Regularisierungsverfahren einsetzen. Diese ändern das eigentlich zu lösende Problem leicht ab, sodass es stabil wird, aber die Lösungen trotzdem in der Nähe der Lösungen des Ausgangsproblems liegen.

Ein einfaches Regularisierungsverfahren ist die Tikhonov-Regularisierung. Das ursprüng-

liche Minimierungsproblem wird durch einen Strafterm ergänzt:

$$\|A\,u - v\|^2 + c\,\|u\|^2 \to \min_{u \in \mathbb{R}^n}. \tag{5.34}$$

Der Regularisierungsparameter $c > 0$ ist dabei geeignet zu wählen. Je kleiner er ist, desto näher liegen die Minimierer an den Lösungen des Ausgangproblems, aber desto instabiler ist der Lösungsprozess. Sehr großes c liefert Lösungen, die kaum auf Datenfehler reagieren, aber weit weg von den Lösungen des Ausgangsproblems liegen.

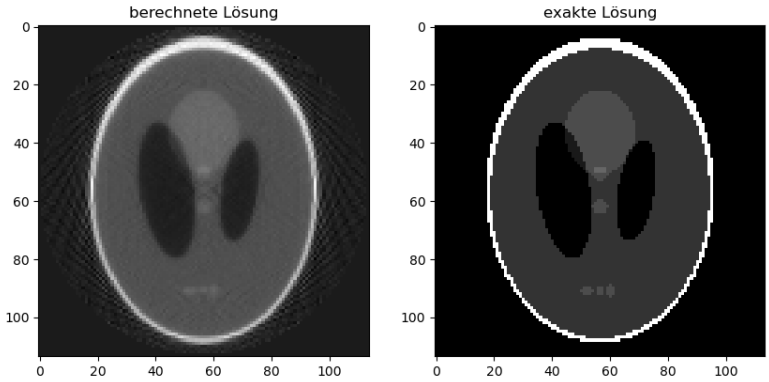
Praktisch versucht man unterschiedliche Werte für c und wählt den Wert, der die besten Ergebnisse auf synthetischen Daten liefert. Es existieren auch verschiedene automatische Parameterwahlstrategien, die aber keine Garantie für eine gute Wahl liefern.

Das Minimierungsproblem kann wieder in ein lineares Gleichungssystem übersetzt werden:

$$A^{\mathrm{T}}\,(A + c\,I)\,u = A^{\mathrm{T}}\,v, \tag{5.35}$$

wobei I die Einheitsmatrix ist.

```
u_noisy = scipy.sparse.linalg.lsqr(A, v_noisy, damp=0.3,
↪  iter_lim=1000)[0]
img_noisy = u_noisy.reshape(n_img_pixels, n_img_pixels).T

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

ax1.imshow(img_noisy, cmap='gray')
ax2.imshow(img_exact, cmap='gray')
ax1.set_title('berechnete Lösung')
ax2.set_title('exakte Lösung')

plt.show()
```

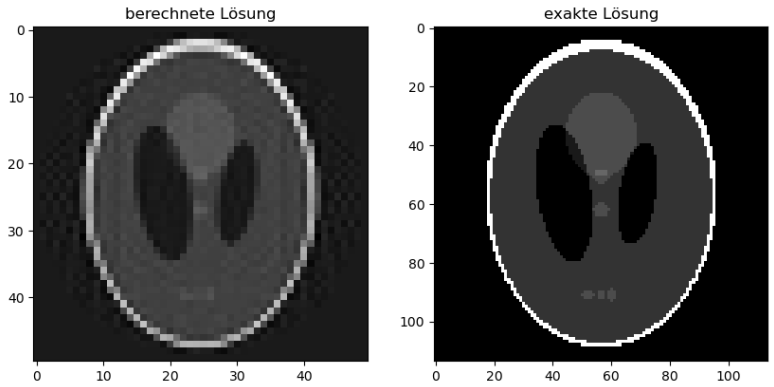


Tikhonov-Regularisierung führt meist zu geglätteten Lösungen. Möchte man dies vermeiden, sollte statt der Pixelbasis eine Wavelet-Basis gewählt und der Strafterm durch $\sum_{k=1}^{n} |u_k|$ ersetzt werden. Dann kann der Minimierer allerdings nicht mehr als Lösung

eines lineare Gleichungssystems berechnet werden. Stattdessen kommen speziell für diese Situation entwickelte Algorithmen zum Einsatz.

Ein andere Regularisierungstechnik ist die Verwendung einer sehr niedrigen Auflösung, was in der Praxis aber nur selten akzeptabel ist.

```
n_img_pixels = 50
A = radon_matrix(n_img_pixels, n_sino_pixels, n_angles)

u_noisy = scipy.sparse.linalg.lsqr(A, v_noisy, iter_lim=1000)[0]
img_noisy = u_noisy.reshape(n_img_pixels, n_img_pixels).T

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

ax1.imshow(img_noisy, cmap='gray')
ax2.imshow(img_exact, cmap='gray')
ax1.set_title('berechnete Lösung')
ax2.set_title('exakte Lösung')

plt.show()
```



Zusammenfassung

Einige Erkenntnisse seien nochmal zusammengefasst:

- Kontinuierliche Modelle sind zwar praxisfern, erlauben aber die Konzentration auf die physikalischen/technischen Zusammenhänge. Man kann das Problem "sehen" ohne sich in den Details der (endlichdimensionalen) Implementierung zu verlieren.
- Es gibt stets eine Vielzahl von Lösungsmöglichkeiten. Im Zweifelsfall gilt: "Keep it simple". Und: Literaturrecherche ist immer eine gute Idee!
- Das Diskretisieren von Modellen ist ein aufwendiger und fehleranfälliger, aber unvermeidbarer Vorgang, der (fast immer) von Hand und sehr gewissenhaft durch-

geführt werden muss.

- Zunächst mit "sauberen" Daten das prinzipielle Vorgehen testen. Anschließend unbedingt auch mit realitätsnahen Daten testen.
- Rechenaufwand kann relevant sein (Lösen des Gleichungssystems) oder völlig egal (Aufstellen der Matrix). Immer den Praxiseinsatz im Blick behalten!

5.6 Realdaten

Wollen unser Vorgehen mit Realdaten testen. Rohe, also unverarbeitete Sinogrammdaten von kommerziellen CT-Geräten sind kaum öffentlich verfügbar. Allerdings gibt es Datensätze von für akademische Zwecke gebauten Geräten. Ein gute Quelle ist die Finnish Inverse Problems Society, welche zum Beispiel Sinogrammdaten einer Walnuss anbietet.

Datensatz

Der Walnuss-Datensatz ist in Tomographic X-ray data of a walnut genauer beschrieben. Einige Eckdaten:

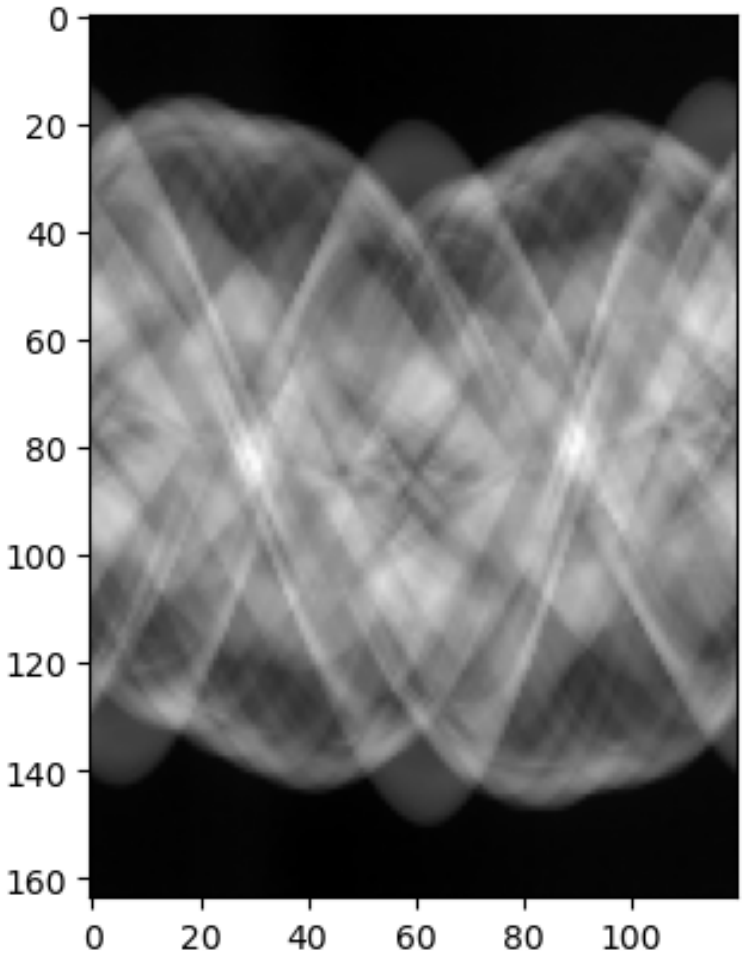
- selbst gebautes CT-Geräte,
- punktförmige Quelle, also keine parallelen Strahlen, sondern fächerförmiger Strahlverlauf,
- flacher Detektor (was unüblich ist bei Fächerstrahl-CT-Geräten),
- Sinogrammdaten für 120 und 1200 Aufnahmewinkel mit 2296 Sinogrammpixeln,
- weitere durch Downsampling gewonnene Sinogramme mit 82, 164, 328 Sinogrammpixeln,

Mit enthalten im Datensatz sind die Radon-Matrizen bzgl. der Pixelbasis. Wir müssen also nicht nochmal selbst die Radon-Transformation diskretisieren.

```
from scipy.io import loadmat

data_dict = loadmat('walnut_data164.mat')
print('keys:')
for key in data_dict:
    print('  ', key)
A = data_dict['A']
print('A:', A.shape, type(A))
sino = data_dict['m']
print('sino:', sino.shape, type(sino))

fig, ax = plt.subplots()
```

```
ax.imshow(sino, cmap='gray')
plt.show()
```

```
keys:
   __header__
   __version__
   __globals__
   A
   m
A: (19680, 26896) <class 'scipy.sparse._csc.csc_matrix'>
sino: (164, 120) <class 'numpy.ndarray'>
```



Lösungsansätze für gefächerte Strahlen

Bisher haben wir nur Computertomografie mit parallel verlaufenden Strahlen betrachtet. Praktisch alle modernen CT-Geräte arbeiten aber mit fächerförmigem oder noch komplexerem (3D) Strahlverlauf. Zur Diskretisierung gefächerter Verläufe gibt es zwei grundlegende Ansätze:

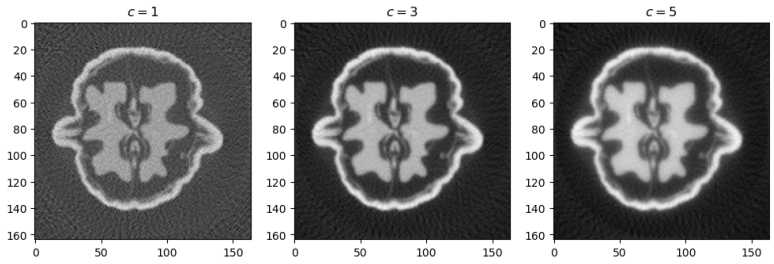
- Analog zu unserem Vorgehen bei parallelen Strahlen wird für jedes Pixel (bzw. jede Basisfunktion) und jeden Strahlwinkel die zugehörige Sinogrammspalte berechnet.
- Man sortiert die Spalten der Sinogramme zwischen den verschiedenen Aufnahmewinkeln um und erhält so Sinogrammdaten für parallele Strahlen (IDVID 570). So kann die oben für parallele Strahlen berechnete Matrix A auch für gefächerte Strahlen verwendet werden. Dieser Ansatz wird als **Rebinning** bezeichnet. Problematisch ist dabei, dass die so erhaltenen (parallelen) Aufnahmewinkel und Pixelpositionen im Sinogramm nicht mehr äquidistant sind, sodass Qualitätsverluste durch notwendiges Resampling der Daten entstehen können.

Rekonstruktion

Schwierigster Punkt bei der Rekonstruktion ist die Wahl des Regularisierungsparameters. Für $c = 0$ erhalten wir keine sinnvolle Rekonstruktion. Für größere Werte werden die Rekonstruktionen besser. Die Wahl eines guten Parameters c hängt von vielen Faktoren ab, insbesondere vom Rauschniveau in den Daten. Über das Rauschniveau liegen uns allerdings keinerlei Informationen vor, sodass wir nur verschiedene Werte für c ausprobieren und die zugehörigen Rekonstruktionen durch manuelle Inspektion bewerten können. Im Prinzip könnte man das zu erwartende Rauschniveau aus entsprechenden Angaben zur verwendeten Hardware ermitteln. Dies ist im vorliegenden Fall jedoch nicht geschehen oder nicht mit veröffentlicht worden.

```
v = sino.T.reshape( -1)

u1 = scipy.sparse.linalg.lsqr(A, v, damp=1, iter_lim=1000)[0]
img1 = u1.reshape(164, 164).T
u2 = scipy.sparse.linalg.lsqr(A, v, damp=3, iter_lim=1000)[0]
img2 = u2.reshape(164, 164).T
u3 = scipy.sparse.linalg.lsqr(A, v, damp=5, iter_lim=1000)[0]
img3 = u3.reshape(164, 164).T

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 5))
ax1.imshow(img1, cmap='gray')
ax2.imshow(img2, cmap='gray')
ax3.imshow(img3, cmap='gray')
ax1.set_title('$c=1$')
ax2.set_title('$c=3$')
```

```
ax3.set_title('$c=5$')
plt.show()
```



6 Partielle Differentialgleichungen

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.

Ist nur eine Funktion einer Veränderlichen gesucht, so spricht man von **gewöhnlichen Differentialgleichungen** (kurz: ODE für "ordinary differential equation"). Diese sind ebenfalls ein weit verbreitetes Werkzeug zur Modellierung von naturwissenschaftlichen, technischen und ökonomischen Zusammenhängen, sind aber deutlich einfacher zu handhaben und deswegen hier nicht explizit Thema. ODEs lassen sich meist mit relativ einfachen Standardverfahren lösen, welche fertig implementiert in Software-Bibliotheken verfügbar sind.

Nebenbemerkung

Für PDEs existiert im Gegensatz zu ODEs (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.

6.1 PDEs erster Ordnung

Struktur

Merke!

Lineare PDE erster Ordnung haben die Form

$$\sum_{i=1}^{n} a_i(\underline{x}) \frac{\partial}{\partial x_i} y(\underline{x}) + b(\underline{x})\, y(\underline{x}) + c(\underline{x}) = 0, \quad \underline{x} \in B. \tag{6.1}$$

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.

Beispiel: Kontinuitätsgleichung

Wollen die Ausbreitung eines Ölteppichs auf einer Wasseroberfläche unter Strömungseinfluss modellieren.

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

$$\varrho_0(x_1, x_2) := \varrho(x_1, x_2, 0) \quad \text{für alle } (x_1, x_2) \in \mathbb{R}^2 \tag{6.2}$$

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.

Die Lösung für allgemeines $\underline{v}$ muss die PDE

$$\frac{\partial}{\partial t}\varrho(x_1, x_2, t) + \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \end{bmatrix} \circ \big(\varrho(x_1, x_2, t)\, \underline{v}(x_1, x_2, t)\big) = 0 \tag{6.3}$$

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 (IDVID 610).

Merke!

PDEs erster Ordnung mit dieser Struktur heißen **Kontinuitätsgleichungen**. Kurzschreibweise:

$$\frac{\partial}{\partial t}\varrho + \operatorname{div}_{\underline{x}}(\varrho\, \underline{v}) = 0, \tag{6.4}$$

wobei $\operatorname{div}_{\underline{x}}$ die Divergenz bzgl. der Ortsvariablen bezeichnet.

Lösungsansätze

PDEs erster Ordnung können auf (nichtlineare) Systeme gewöhnlicher Differentialgleichungen zurückgeführt werden.

Alternativ ist die später zu behandelnde Methode der finiten Differenzen einsetzbar.

6.2 PDEs zweiter Ordnung

Struktur

Merke!

Lineare PDE zweiter Ordnung haben die Form

$$\begin{aligned} &\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(\underline{x})\, \frac{\partial^2}{\partial x_i \partial x_j} y(\underline{x}) \\ &\quad + \sum_{i=1}^{n} b_i(\underline{x})\, \frac{\partial}{\partial x_i} y(\underline{x}) + c(\underline{x})\, y(\underline{x}) + d(\underline{x}) = 0, \quad \underline{x} \in B. \end{aligned} \tag{6.5}$$

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

$$A(\underline{x}) = \begin{bmatrix} a_{11}(\underline{x}) & \cdots & a_{1n}(\underline{x}) \\ \vdots & & \vdots \\ a_{n1}(\underline{x}) & \cdots & a_{nn}(\underline{x}) \end{bmatrix} \tag{6.6}$$

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.

Beispiel: Potentialgleichung

Merke!

PDE der Form

$$\Delta y(x_1, \ldots, x_n) = f(x_1, \ldots, x_n), \quad \underline{x} \in B \tag{6.7}$$

mit dem **Laplace-Operator**

$$\Delta := \frac{\partial^2}{\partial^2 x_1} + \cdots + \frac{\partial^2}{\partial^2 x_n} \tag{6.8}$$

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".

Für die Potentialgleichung auf $B = \mathbb{R}^2 \setminus \{0\}$ ohne Randbedingungen ist bekannt, dass stets radialsymmetrische Lösungen existieren, also Lösungen, die nicht direkt von $\underline{x}$, sondern nur von $r := \sqrt{x_1^2 + \cdots + x_n^2}$ abhängen. Setzen wir $u(r) := y(\underline{x})$ so hat u die Form

$$u(r) = \begin{cases} \ln r + c, & \text{für } n = 2, \\ r^{-(n-2)} + c, & \text{für } n \geq 3. \end{cases} \tag{6.9}$$

Alle Vielfachen davon sind ebenfalls Lösungen (IDVID 620).

Auch zu einem anderen "Mittelpunkt" verschobene Varianten dieser Funktionen sind wieder Lösungen und Summen von Lösungen sind wieder Lösungen. Bei gegebenen Randbedingungen sehen die Lösungen zwangsläufig anders aus.

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.

Beispiel (Faraday'scher Käfig)

Bringt man eine leitfähige hohle Kugel (z.B. Auto) in ein elektrisches Feld (z.B. Gewitter/Blitz), so ordnen sich die Ladungen auf der Kugeloberfläche so an, dass kein Strom fließt. Das Potential auf der Oberfläche ist also konstant. Im Inneren der Kugel befinden sich keine Ladungen:

$$\Delta y(\underline{x}) = 0 \quad \text{für } |\underline{x}| < 1, \qquad y(\underline{x}) = c \quad \text{für } |\underline{x}| = 1. \tag{6.10}$$

Aus dem Maximumprinzip folgt, dass das Potential im Inneren konstant gleich dem Potential an der Oberfläche ist, also nirgends ein Strom fließen kann.

Beispiel: Diffusionsgleichung

Wollen die Wärmeverteilung im Laufe der Zeit in einem Körper $B \subseteq \mathbb{R}^n$ bei gegebenen Anfangs- und Randbedingungen modellieren. Analog zur Massenbilanz beim Ölteppichbeispiel führt hier eine Energiebilanz zur PDE

$$\Delta_{\underline{x}} u(\underline{x}, t) - \frac{\varrho\, c}{\lambda} \frac{\partial}{\partial t} u(\underline{x}, t) = 0. \tag{6.11}$$

(IDVID 630). 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

$$\Delta_{\underline{x}} u - a \frac{\partial}{\partial t} u = 0 \tag{6.12}$$

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:

$$u(\underline{x}, 0) = f(\underline{x}) \quad \text{für alle } \underline{x} \in B \cup \partial B \tag{6.13}$$

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**

  $$u(\underline{x}, t) = g(\underline{x}) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{6.14}$$

  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**

  $$\frac{\partial}{\partial n} u(\underline{x}, t) = h(\underline{x}) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{6.15}$$

  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**

  $$\frac{\partial}{\partial n} u(\underline{x}, t) = \alpha(\underline{x}) \big(u(\underline{x}, t) - U\big) \quad \text{für alle } \underline{x} \in \partial B \quad \text{und alle } t \geq 0 \tag{6.16}$$

  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

$$u_{xx}(x,t) - u_t(x,t) = 0 \quad \text{für alle } x \in (0,1) \quad \text{und alle } t > 0 \tag{6.17}$$

lautet. Die anfängliche Temperaturverteilung sei

$$u(x,0) = \cos(2\,\pi\,x) \quad \text{für alle } x \in [0,1] \tag{6.18}$$

und der Stab sei an den beiden enden perfekt isolierte, also

$$u_x(0,t) = u_x(1,t) = 0 \quad \text{für alle } t \geq 0. \tag{6.19}$$

Für diesen speziellen Fall kann man die Lösungen analytisch bestimmen (tun wir hier nicht) und erhält

$$u(x,t) = \cos(2\,\pi\,x)\,\mathrm{e}^{-4\,\pi^2\,t} \tag{6.20}$$

(IDVID 640).

Beispiel: Wellengleichung

Für die Auslenkung $u(x,t)$ einer Saite $B = (0,1)$ gilt

$$\frac{\sigma}{\varrho}\,u_{xx}(x,t) - u_{tt}(x,t) = 0 \quad \text{für alle } x \in (0,1) \quad \text{und alle } t > 0 \tag{6.21}$$

(IDVID 650). 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 PDE den zeitlichen Verlauf der Auslenkung.

Allgemeiner gilt:

Merke!

PDE mit der Struktur

$$c^2\,\Delta_{\underline{x}} u - \frac{\partial^2}{\partial^2 t} u = 0 \tag{6.22}$$

mit einer Konstante $c > 0$ heißen **Wellengleichungen** und sind stets hyperbolisch.

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,...).

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

$$u(0,t) = u(1,t) = 0 \quad \text{für alle } t \geq 0. \tag{6.23}$$

Für eine gegebene Funktion $f : [0,1] \to \mathbb{R}$ kann man zu den Anfangsbedingungen

$$u(x,0) = f(x) \quad \text{für alle } x \in [0,1] \tag{6.24}$$

und

$$u_t(x,0) = 0 \quad \text{für alle } x \in [0,1] \tag{6.25}$$

(Saite in vorgegebener Lage, aber noch nicht in Bewegung) die Lösung der Wellengleichung

$$c^2\, u_{xx}(x,t) - u_{tt}(x,t) = 0 \quad \text{für alle } x \in [0,1] \quad \text{und alle } t \geq 0 \tag{6.26}$$

analytisch bestimmen (tun wir hier nicht). Man erhält

$$u(x,t) = \sum_{k=1}^{\infty} a_k \, \sin(k\,\pi\,x) \, \cos(c\,k\,\pi\,t) \tag{6.27}$$

wobei

$$a_k := 2 \int_0^1 f(x) \, \sin(k\,\pi\,x)\,\mathrm{d}x \tag{6.28}$$

gerade die Sinus-Anteile der Fourier-Koeffizienten der Anfangsauslenkung f sind.

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 2. Ordnung ausschließlich numerisch.

7 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.

7.1 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 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).

In der Veranstaltung zur numerischen Mathematik wird noch der Begriff der Stabilität eines Algorithmus 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

$$\text{Algorithmus} = \text{Diskretisieren} + \text{Standardverfahren}, \tag{7.1}$$

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.

7.3 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.

7.4 Beispiel: 2D-Poisson-Gleichung

Wollen

$$\Delta u(x,y) = f(x,y), \quad (x,y) \in B \tag{7.2}$$

für das Rechteck

$$B = (0,a) \times (0,b) \tag{7.3}$$

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:

$$u(x,y) = 0 \quad \text{für } (x,y) \in \partial B. \tag{7.4}$$

Gitterpunkte

Wählen in x- und y-Richtung $n_x + 1$ bzw. $n_y + 1$ gleichmäßig verteilte Stützstellen mit Abständen

$$\Delta x := \frac{a}{n_x}, \quad \Delta y := \frac{b}{n_y}, \tag{7.5}$$

sodass die Gitterpunkte

$$(x_i, y_j), \quad x_i = i\,\Delta x, \quad y_j = j\,\Delta y, \quad i = 0,\ldots,n_x, \quad j = 0,\ldots,n_y \tag{7.6}$$

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:

$$u_{i,j} := u(x_i, y_j), \qquad f_{i,j} := f(x_i, y_j). \tag{7.7}$$

Diskretisierung im Inneren

Die zu lösende Gleichung

$$u_{xx}(x,y) + u_{yy}(x,y) = f(x,y) \tag{7.8}$$

betrachten wir nur auf den inneren Gitterpunkten und ersetzen dort die Ableitungen durch die Differenzenquotienten

$$D_x^2(x_i, y_j) := \frac{u_{i+1,j} - 2\,u_{i,j} + u_{i-1,j}}{(\Delta x)^2} \tag{7.9}$$

bzw.

$$D_y^2(x_i, y_i) := \frac{u_{i,j+1} - 2\,u_{i,j} + u_{i,j-1}}{(\Delta y)^2}. \tag{7.10}$$

Erhalten somit

$$D_x^2(x_i, y_j) + D_y^2(x_i, y_j) = f_{i,j}, \quad i = 1,\ldots,n_x - 1, \quad j = 1,\ldots,n_y - 1. \tag{7.11}$$

Diskretisierung der Randbedingung

Die Randbedingung $u(x, y) = 0$ liefert auf den Randgitterpunkten

$$u_{0,j} = u_{n_x,j} = u_{i,0} = u_{i,n_y} = 0 \tag{7.12}$$

für $i = 0, \ldots, n_x$ und $j = 0, \ldots, n_y$.

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.

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:

$$\max_G |u - \tilde{u}| \leq \max_{\partial B \cap G} |u - \tilde{u}| + \frac{r^2}{2} \max_{B \cap G} |f - A\,\tilde{u}|. \tag{7.13}$$

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.

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

$$\max_{(B \cup \partial B) \cap G} |u - \tilde{u}| \leq c \max\{\Delta x, \Delta y\} \tag{7.14}$$

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.

7.5 Nicht triviale 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 (IDVID 750).

8 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

$$c^2 \, u_{xx}(x, t) - u_{tt}(x, t) = 0 \tag{8.1}$$

erfüllt. Eine Anfangsbedingung sei durch

$$u(x, 0) = f(x), \qquad u_t(x, 0) = 0 \tag{8.2}$$

mit $f : [0, a] \to \mathbb{R}$ gegeben.

8.1 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

$$u(0, t) = u(a, t) = 0 \tag{8.3}$$

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

$$u_x(0, t) = u_x(a, t) = 0. \tag{8.4}$$

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

$$u_x(0, t) = 0, \quad u_t(a, t) + c \, u_x(a, t) = 0. \tag{8.5}$$

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 (IDVID 810). 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!

8.2 Diskretisierung

Gitterpunkte

Setzen

$$\Delta x := \frac{a}{n_x}, \qquad \Delta t := \frac{T}{n_t} \tag{8.6}$$

und erhalten so Gitterpunkte (x_i, t_j) mit

$$x_i := i\,\Delta x, \qquad t_j := j\,\Delta t \tag{8.7}$$

für $i = 0, 1, \ldots, n_x$ und $j = 0, 1, \ldots, n_t$.

Gesucht sind die Werte

$$u_{i,j} := u(x_i, t_j) \tag{8.8}$$

von u an den Gitterpunkten.

Die Gitterwerte der Funktion f in der Anfangsbedingung bezeichnen wir entsprechend mit

$$f_i := f(x_i). \tag{8.9}$$

Innere Punkte

Für Punkte im Inneren des betrachteten Gebietes soll die Wellengleichung gelten, wobei wir die zweiten Ableitungen durch entsprechende Differenzenquotienten ersetzen:

$$c^2\,\frac{u_{i-1,j} - 2\,u_{i,j} + u_{i+1,j}}{(\Delta x)^2} - \frac{u_{i,j-1} - 2\,u_{i,j} + u_{i,j+1}}{(\Delta t)^2} = 0 \tag{8.10}$$

für $i = 1, \ldots, n_x - 1$ und $j = 1, \ldots, n_t - 1$.

Umstellen nach $u_{i,j+1}$ liefert

$$u_{i,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,j} - 2\,u_{i,j} + u_{i+1,j}) - (u_{i,j-1} - 2\,u_{i,j}). \tag{8.11}$$

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.

Anfangsbedingung

Wir haben

$$u_{i,0} = f_i \tag{8.12}$$

für $i = 0, \ldots, n_x$.

Die Ableitungen $u_t(x, 0)$ ersetzen wir durch die zentrale Differenz

$$u_t(x_i, 0) \approx \frac{u_{i,1} - u(x_i, -\Delta t)}{2\,\Delta t}. \tag{8.13}$$

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

$$u_{i,1} = u(x_i, -\Delta t), \quad i = 0, \ldots, n_x. \tag{8.14}$$

In Formel (8.11) für $j = 0$ eingesetzt erhalten wir für den ersten Zeitschritt die Formel

$$u_{i,1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,0} - 2\,u_{i,0} + u_{i+1,0}) - (u_{i,1} - 2\,u_{i,0}). \tag{8.15}$$

Auflösen nach $u_{i,1}$ liefert nun

$$u_{i,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{i-1,0} - 2\,u_{i,0} + u_{i+1,0}) + u_{i,0}. \tag{8.16}$$

sodass die unbekannten Werte $u(x_i, -\Delta t)$ gar keine Rolle spielen.

Randbedingungen

Dirichlet Die diskretisierten Dirichlet-Randbedingungen sind

$$u_{0,j} = u_{n_x,j} = 0, \qquad j = 0, \ldots, n_t. \tag{8.17}$$

Neumann links Für die Neumann-Randbedingung links verwenden wir analog zur Zeitableitung in der Anfangsbedingung die zentrale Differenz

$$u_x(0,t_j) \approx \frac{u_{1,j} - u(-\Delta x, t_j)}{2\,\Delta x}, \tag{8.18}$$

welche zur Bedingung

$$u_{1,j} = u(-\Delta x, t_j), \qquad j = 0,\ldots,n_t \tag{8.19}$$

führt. Betrachten wir nun (8.11) für $i = 0$ und setzen dort diese Beziehung ein, so erhalten wir mit

$$u_{0,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{1,j} - 2\,u_{0,j} + u_{1,j}) - (u_{0,j-1} - 2\,u_{0,j}). \tag{8.20}$$

eine Formel zur Berechnung des Randwertes $u_{0,j+1}$, welche für $j \geq 1$ funktioniert. Für $j = 0$ kann mittels Einsetzen von (8.14) für $i = 0$ und Auflösen nach $u_{0,1}$ daraus widerum

$$u_{0,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{1,0} - 2\,u_{0,0} + u_{1,0}) + u_{0,0} \tag{8.21}$$

gewonnen werden.

Neumann rechts Analog zum linken Rand erhält man für den rechten Rand die Berechnungsvorschriften

$$u_{n_x,j+1} = \left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{n_x-1,j} - 2\,u_{n_x,j} + u_{n_x-1,j}) - (u_{n_x,j-1} - 2\,u_{n_x,j}). \tag{8.22}$$

für $j \geq 1$ und

$$u_{n_x,1} = \frac{1}{2}\left(\frac{c\,\Delta t}{\Delta x}\right)^2 (u_{n_x-1,0} - 2\,u_{n_x,0} + u_{n_x-1,0}) + u_{n_x,0}. \tag{8.23}$$

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):

$$u_t(a,t_j) + c\,u_x(a,t_j) \approx \frac{u_{n_x,j+1} - u_{n_x,j}}{\Delta t} + c\,\frac{u_{n_x,j} - u_{n_x-1,j}}{\Delta x}. \tag{8.24}$$

Nullsetzen und Umstellen nach $u_{n_x,j+1}$ liefert

$$u_{n_x,j+1} = u_{n_x,j} - \frac{c\,\Delta t}{\Delta x}\,(u_{n_x,j} - u_{n_x-1,j}) \tag{8.25}$$

für $j \geq 0$.

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.

8.3 Kondition

Man kann zeigen, dass die Berechnung der Funktion u auf dem Gitter gut konditioniert ist, wenn

$$\frac{c\,\Delta t}{\Delta x} \leq 1 \tag{8.26}$$

gilt. Der Wert auf der linken Seite heißt auch **Courant-Zahl**.

8.4 Konvergenz

Man kann zeigen, dass für

$$\frac{c\,\Delta t}{\Delta x} = 1 \tag{8.27}$$

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.

8.5 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
```

8.6 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
```

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

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

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

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

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.

9 Fallstudie: 2D-Kontinuitätsgleichung

Betrachten verschiedene (auch ungeeignete) Verfahren zum Lösen der Kontinuitätsgleichung in zwei Raumdimensionen.

9.1 Problemstellung

Gleichung

Die Kontinuitätsgleichung

$$u_t(x,y,t) + (v\,u)_x(x,y,t) + (w\,u)_y(x,y,t) = 0, \quad (x,y) \in B, \quad ,t \geq 0 \tag{9.1}$$

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.

Anfangsbedingung

Die Anfangskonzentration des durch die Strömung transportierten Stoffs kann durch eine vom Ort abhängige Funktion f ausgedrückt werden:

$$u(x,y,t) = f(x,y) \quad \text{für alle } (x,y) \in B \text{ und } t \geq 0. \tag{9.2}$$

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

$$B := [0, a] \times [0, b]. \tag{9.3}$$

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:

$$v_x(x, y, t) + w_y(x, y, t) = 0 \quad (x, y) \in B, \quad t \geq 0. \tag{9.4}$$

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

$$u_t(x, y, t) + v(x, y, t)\, u_x(x, y, t) + w(x, y, t)\, u_y(x, y, t) = 0 \tag{9.5}$$

vereinfacht werden (IDVID 910).

9.2 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

- $u_{i,j,k} := u(x_i, y_j, t_k)$,
- $v_{i,j} := v(x_i, y_j)$,
- $w_{i,j} := w(x_i, y_j)$,
- $f_{i,j} := f(x_i, y_j)$.

9.3 Einfaches Upwind-Verfahren

Idee

Ersetzt man u_t durch eine Vorwärtsdifferenz und u_x und u_y durch Rückwärtsdifferenzen, so erhält man

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j}\,\frac{\Delta t}{\Delta x}\,(u_{i,j,k} - u_{i-1,j,k}) - w_{i,j}\,\frac{\Delta t}{\Delta y}\,(u_{i,j,k} - u_{i,j-1,k}) \tag{9.6}$$

(IDVID 920). Die Zahlen

$$C_x := \max_{i,j} |v_{i,j}|\,\frac{\Delta t}{\Delta x} \quad \text{und} \quad C_y := \max_{i,j} |w_{i,j}|\,\frac{\Delta t}{\Delta y} \tag{9.7}$$

heißen **Courant-Zahlen**.

Zur Interpretation betrachten wir eine Strömung parallel zur x-Achse, also $w \equiv 0$:

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j}\,\frac{\Delta t}{\Delta x}\,(u_{i,j,k} - u_{i-1,j,k}). \tag{9.8}$$

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

$$|v_{i,j}| \leq \frac{\Delta x}{\Delta t} \quad \text{für alle } i, j. \tag{9.9}$$

- 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.

Umsetzung

Setzen wir

$$u_{i,j,k}^{\text{in},x} := \begin{cases} u_{i-1,j,k}, & \text{falls } v_{i,j} \geq 0, \\ u_{i+1,j,k}, & \text{falls } v_{i,j} < 0 \end{cases} \tag{9.10}$$

und

$$u_{i,j,k}^{\text{in},y} := \begin{cases} u_{i,j-1,k}, & \text{falls } w_{i,j} \geq 0, \\ u_{i,j+1,k}, & \text{falls } w_{i,j} < 0, \end{cases} \tag{9.11}$$

so erhalten wir die diskretisierte Gleichung

$$u_{i,j,k+1} = u_{i,j,k} - |v_{i,j}| \frac{\Delta t}{\Delta x} \left(u_{i,j,k} - u_{i,j,k}^{\text{in},x}\right) - |w_{i,j}| \frac{\Delta t}{\Delta y} \left(u_{i,j,k} - u_{i,j,k}^{\text{in},y}\right) \tag{9.12}$$

(IDVID 625). 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.

Kondition und Konvergenz

Man kann zeigen, dass die Berechnungen im Upwind-Verfahren gut konditioniert sind, wenn

$$C_x + C_y \leq 1 \tag{9.13}$$

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.

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.

9.4 Verbessertes Upwind-Verfahren

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.

Umsetzung

Um die Anzahl der Zeitschritte nicht zu verdoppeln, verwendet man bei der Darstellung dieses Vorgehens in Formeln gern halbe Zeitschritte:

$$u^{\text{in},x}_{i,j,k} := \begin{cases} u_{i-1,j,k}, & \text{falls } v_{i,j} \geq 0, \\ u_{i+1,j,k}, & \text{falls } v_{i,j} < 0, \end{cases} \tag{9.14}$$

$$u_{i,j,k+\frac{1}{2}} := u_{i,j,k} - |v_{i,j}| \frac{\Delta t}{\Delta x} (u_{i,j,k} - u^{\text{in},x}_{i,j,k}), \tag{9.15}$$

$$u^{\text{in},y}_{i,j,k+\frac{1}{2}} := \begin{cases} u_{i,j-1,k+\frac{1}{2}}, & \text{falls } w_{i,j} \geq 0, \\ u_{i,j+1,k+\frac{1}{2}}, & \text{falls } w_{i,j} < 0, \end{cases} \tag{9.16}$$

$$u_{i,j,k+1} := u_{i,j,k+\frac{1}{2}} - |w_{i,j}| \frac{\Delta t}{\Delta y} (u_{i,j,k+\frac{1}{2}} - u^{\text{in},y}_{i,j,k+\frac{1}{2}}). \tag{9.17}$$

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

9.5 Zentrale Differenzen

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.

Umsetzung

Einsetzen der Differenzenquotienten in die PDE und umstellen nach $u_{i,j,k+1}$ liefert

$$u_{i,j,k+1} = u_{i,j,k} - v_{i,j} \frac{\Delta t}{2\,\Delta x} (u_{i+1,j,k} - u_{i-1,j,k}) - w_{i,j} \frac{\Delta t}{2\,\Delta y} (u_{i,j+1,k} - u_{i,j-1,k}) \quad (9.18)$$

für $i = 1, \ldots, n_x - 1$ und $j = 1, \ldots, n_y - 1$ (IDVID 930).

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.

> **Nebenbemerkung**
>
> Die einseitige Differenz am Rand kann auch als zentrale Differenz interpretiert werden, wobei der Funktionswert außerhalb des Gebietes durch lineare Extrapolation ermittelt wird (IDVID 935).

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.

9.6 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

$$u_{i,j,k+1} = \overline{u}_{i,j,k} - v_{i,j} \frac{\Delta t}{2\,\Delta x} \left(u_{i+1,j,k} - u_{i-1,j,k}\right) - w_{i,j} \frac{\Delta t}{2\,\Delta y} \left(u_{i,j+1,k} - u_{i,j-1,k}\right) \tag{9.19}$$

mit

$$\overline{u}_{i,j,k} := \frac{1}{4} \left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k}\right). \tag{9.20}$$

Gilt

$$C_x \leq \frac{1}{2} \quad \text{und} \quad C_y \leq \frac{1}{2} \tag{9.21}$$

mit C_x und C_y aus (9.7), 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.

Der Funktionswert der außerhalb des Gebietes liegenden Nachbarn von Randpunkten muss je nach Strömungsrichtung gewählt werden. Bei Strömung in das Gebiet hinein ist der Funktionswert Null. Bei Strömung aus dem Gebiet heraus ist er unbekannt. Eine Näherung können wir durch lineare Extrapolation ermitteln, vgl. Nebenbemerkung beim FTCS-Verfahren.

```
def cont2d_lax_friedrichs(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'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_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, :, :] = 0
    u[k + 1, 1:, :] += 0.25 * u[k, : -1, :]  # linker Nachbar
    u[k + 1, : -1, :] += 0.25 * u[k, 1:, :]  # rechter Nachbar
    u[k + 1, :, 1:] += 0.25 * u[k, :, : -1]  # unterer Nachbar
    u[k + 1, :, : -1] += 0.25 * u[k, :, 1:]  # oberer Nachbar
    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, :] += 0.25 * u_ext
    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, :] += 0.25 * u_ext
    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] += 0.25 * u_ext
    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] += 0.25 * u_ext
```

```
        u[k + 1, :, -1] -= 0.5 * w[:, -1] * delta_t / delta_y * (u_ext
        ↪  - u[k, :, -2])

    return u
```

```
a = 3
b = 2
T = 4

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_lax_friedrichs(a, b, T, v, w, f, nx, ny, nt)
show_anim(u, a, b, T, v, w, quiver_step=2)
```

```
Courant -Zahlen: 0.49382716049382713, 0.5
```

0

[ ] Once [x] Loop [ ] Reflect

Wie beim Upwind-Verfahren läuft die Anfangsverteilung auch beim Lax-Friedrichs-Verfahren quer zur Strömungsrichtung stärker breit als in Strömungsrichtung.s

9.7 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.

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.

Umsetzung

Berechnen den Herkunftsort des in (x_i, y_j) ankommenden Partikels näherungsweise als

$$\begin{bmatrix} \tilde{x}_i \\ \tilde{y}_j \end{bmatrix} := \begin{bmatrix} x_i \\ y_j \end{bmatrix} - \Delta t \begin{bmatrix} v_{i,j} \\ w_{i,j} \end{bmatrix} \tag{9.22}$$

und setzen

$$u_{i,j,k+1} := u(\tilde{x}_i, \tilde{y}_j, t_k). \tag{9.23}$$

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.

Implementierung

Die Interpolation überlassen wir `scipy.interpolate.RegularGridInterpolator`. 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

9.8 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.

9.9 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:

$$\begin{bmatrix} v(x,y) \\ w(x,y) \end{bmatrix} = \nabla p(x,y) \tag{9.24}$$

mit einer Funktion $p : B \to \mathbb{R}$. Das Potential ist Lösung der Potentialgleichung

$$\Delta p(x,y) = 0 \quad \text{für alle } (x,y) \in B. \tag{9.25}$$

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):

$$\begin{bmatrix} v(x,y) \\ w(x,y) \end{bmatrix} = \left(1 - \frac{R^2}{r^2}\right) \cos\varphi \begin{bmatrix} \cos\varphi \\ \sin\varphi \end{bmatrix} - \left(1 + \frac{R^2}{r^2}\right) \sin\varphi \begin{bmatrix} -\sin\varphi \\ \cos\varphi \end{bmatrix} \tag{9.26}$$

für $\varphi \in [0, 2\,\pi)$ und $r \geq R$, wobei

$$x = x_\mathrm{M} + r\,\cos\varphi, \qquad y = y_\mathrm{M} + r\,\sin\varphi. \tag{9.27}$$

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

10 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.

10.1 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

$$\|f\|_2 := \sqrt{\int_B f^2 \, \mathrm{d}b} \tag{10.1}$$

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

$$[f] := \{g : \ f \text{ und } g \text{ sind fast überall gleich}\} \tag{10.2}$$

eine solche Menge untereinander fast überall gleicher Funktionen, so gilt

$$\|g\|_2 = \|f\|_2 \quad \text{für alle } g \in [f]. \tag{10.3}$$

Wir können diesen Wert also als Eigenschaft der gesamten Äquivalenzklasse ansehen:

$$\|[f]\|_2 := \|f\|_2. \tag{10.4}$$

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

$$f(x) := \begin{cases} -1, & x \leq 0 \\ 1, & x > 0, \end{cases} \tag{10.5}$$

$$g(x) := \begin{cases} -1, & x < 0 \\ 1, & x \geq 0, \end{cases} \tag{10.6}$$

$$h(x) := \begin{cases} -1, & x < 0 \\ 0, & x = 0, \\ 1, & x > 0 \end{cases} \tag{10.7}$$

(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.

10.2 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

$$\int_B u \frac{\partial \varphi}{\partial x_i} \, \mathrm{d}b = -\int_B w \, \varphi \, \mathrm{d}b \quad \text{für alle } \varphi \in C_0^1(B) \tag{10.8}$$

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 (IDVID 1010).

Analoges gilt für partielle Ableitungen.

Beispiel

Die verallgemeinerte Ableitung von

$$f(x) := |x| \tag{10.9}$$

auf $B = (-1, 1)$ ist

$$f'(x) := \begin{cases} -1, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{10.10}$$

(IDVID 1015).

Beispiel

Die verallgemeinerte Ableitung von

$$f(x) := \begin{cases} -1, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{10.11}$$

auf $B = (-1, 1)$ existiert nicht, ist also insbesondere nicht (!) die Nullfunktion (IDVID 1020).

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

$$\|f\|_{2,1} := \sqrt{\|f\|_2^2 + \|f'\|_2^2}. \tag{10.12}$$

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:

$$f \in H^1(B) \quad \Leftrightarrow \quad \|f\|_{2,1} < \infty. \tag{10.13}$$

Insbesondere ist die $H^1(B)$-Norm "stärker" als die $L^2(B)$-Norm im Sinne von

$$\|f\|_2 \leq \|f\|_{2,1}. \tag{10.14}$$

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

$$m < k - \frac{d}{2} \tag{10.15}$$

gilt (Sobolev'scher Einbettungssatz). Für zweidimensionale Gebiete ($d = 2$) folgt Stetigkeit ($m = 0$) also beispielsweise aus zweimaliger verallgemeinerter Differenzierbarkeit ($k = 2$).

10.3 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" zu-

sammengefasst. 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.

10.4 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

$$a(u, v) = b(v) \quad \text{für alle } v \in V \tag{10.16}$$

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

$$|a(v_1, v_2)| \leq c \, \|v_1\| \, \|v_2\| \quad \text{für alle } v_1, v_2 \in V. \tag{10.17}$$

- b ist beschränkt, d.h. es existiert eine Konstante $c > 0$ mit

$$|b(v)| \leq c \, \|v\| \quad \text{für alle } v \in V. \tag{10.18}$$

- a ist elliptisch, d.h. es existiert eine Konstante $c > 0$ mit

$$|a(v, v)| \geq c \, \|v\|^2 \quad \text{für alle } v \in V. \tag{10.19}$$

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

$$\int_B u \, v \, \mathrm{d}b \tag{10.20}$$

gegeben.

Finden wir für eine gegebene PDE eine schwache Formulierung in der Form (10.16), die die drei Bedingungen erfüllt, so sind Existenz und Eindeutigkeit einer Lösung garantiert.

10.5 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

$$v = c_1\, v_1 + \cdots + c_m\, v_m \tag{10.21}$$

mit gewissen Zahlen $c_1, \ldots, c_m$ geschrieben werden. Die Gleichung (10.16) gilt nun genau dann für alle $v \in V_m$, wenn die m Gleichungen

$$a(u, v_k) = b(v_k) \quad \text{für } k = 1, \ldots, m \tag{10.22}$$

erfüllt sind (IDVID 1030).

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

$$u = c_1\, u_1 + \cdots + c_n\, u_n \tag{10.23}$$

geschrieben, sodass nur noch die Zahlen $c_1, \ldots, c_n$ zu ermitteln sind. Einsetzen in (10.22) liefert das lineare Gleichungssystem

$$\sum_{l=1}^{n} c_l\, a(u_l, v_k) = b(v_k) \quad \text{für } k = 1, \ldots, m \tag{10.24}$$

zur Berechnung der c_l und damit zur Berechung einer Näherungslösung u von (10.16) (IDVID 1035).

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.

10.6 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

$$u'(x) = f(x) \quad \text{für alle } x \in B := (0,1) \tag{10.25}$$

gilt. Offensichtlich ist u eine Stammfunktion von f. Um Eindeutigkeit zu sichern fordern wir noch

$$u(0) = 0. \tag{10.26}$$

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:

$$\int_0^1 u'(x)\, v(x)\, \mathrm{d}x = \int_0^1 f(x)\, v(x)\, \mathrm{d}x. \tag{10.27}$$

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 (10.16), wenn wir

$$a(u,v) := \int_0^1 u'(x)\, v(x)\, \mathrm{d}x \tag{10.28}$$

und

$$b(v) := \int_0^1 f(x)\, v(x)\, \mathrm{d}x \tag{10.29}$$

wählen. Somit kann das entsprechende Diskretisierungsverfahren eingesetzt werden, welches zu einem linearen Gleichungssystem führt.

10.7 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

$$\Delta u(x) = f(x) \quad \text{für } x \in B \tag{10.30}$$

fordern wir die Randbedingungen

$$u(x) = g_1(x) \quad \text{für } x \in \Gamma_1 \tag{10.31}$$

und

$$\frac{\partial}{\partial n} u(x) = g_2(x) \quad \text{für x} \in \Gamma_2, \tag{10.32}$$

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)

  $$\int_B \Delta u \, v \, \mathrm{d}b = \int_{\partial B} \frac{\partial}{\partial n} u \, v \, \mathrm{d}a - \int_B \nabla u \circ \nabla v \, \mathrm{d}b \tag{10.33}$$

  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

$$\int_B \nabla u \circ \nabla v \, \mathrm{d}b = \int_{\Gamma_2} g_2 \, v \, \mathrm{d}a - \int_B f \, v \, \mathrm{d}b \tag{10.34}$$

für eine hinreichend große Menge von Testfunktionen v gilt (IDVID 1050).

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

$$V := \{v \in H^1(B) : \ v(x) = 0 \text{ für } x \in \Gamma_1\}. \tag{10.35}$$

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

$$u = u_0 + u_1 \quad \text{mit } u_0 \in V \tag{10.36}$$

schreiben. Einsetzen in die schwache Formulierung liefert die nur formal, nicht inhaltlich geänderte schwache Formulierung:

Merke!

Finde $u_0 \in V$, sodass

$$\int_B \nabla u_0 \circ \nabla v \,\mathrm{d}b = \int_{\Gamma_2} g_2 \, v \,\mathrm{d}a - \int_B f \, v \,\mathrm{d}b - \int_B \nabla u_1 \circ \nabla v \,\mathrm{d}b \tag{10.37}$$

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

$$a(u, v) := \int_B \nabla u \circ \nabla v \,\mathrm{d}b \tag{10.38}$$

und

$$b(v) := \int_{\Gamma_2} g_2 \, v \,\mathrm{d}a - \int_B f \, v \,\mathrm{d}b - \int_B \nabla u_1 \circ \nabla v \,\mathrm{d}b \tag{10.39}$$

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.

11 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

$$\operatorname{supp} v_k := \{x \in B : \ v_k(x) \neq 0\} \tag{11.1}$$

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.

11.1 Ausgangspunkt

Gegeben sei die schwache Formulierung einer PDE in der Form

$$a(u, v) = b(v) \quad \text{für alle } v \in V, \tag{11.2}$$

wobei a, b, V gegeben sind und u gesucht ist (vgl. Section 10.4).

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

$$A\,\underline{u} = \underline{b} \tag{11.3}$$

mit der rechten Seite

$$\underline{b} := \begin{bmatrix} b(v_1) \\ \vdots \\ b(v_n) \end{bmatrix} \tag{11.4}$$

und der Systemmatrix (hier auch **Steifigkeitsmatrix** genannt)

$$A := \begin{bmatrix} a(v_1, v_1) & \cdots & a(v_n, v_1) \\ \vdots & & \vdots \\ a(v_1, v_n) & \cdots & a(v_n, v_n) \end{bmatrix} \tag{11.5}$$

(vgl. Section 10.5). Aus dem Vektor $\underline{u} \in \mathbb{R}^n$ erhalten wir mittels

$$u := \underline{u}_1\, v_1 + \cdots + \underline{u}_n\, v_n \tag{11.6}$$

die gesuchte Lösung der (schwachen Formulierung der) PDE.

Einziger noch zu klärender Punkt ist die konkrete Wahl der Basisfunktionen v_k.

11.2 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 (IDVID 1120).

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.

11.3 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

$$\frac{R_l}{r_l} \leq c \quad \text{für } l = 1, \ldots, N \tag{11.7}$$

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.

11.4 Aufstellen der Steifigkeitsmatrix

Ausgehend von einer gegebenen Zerlegung des Gebiets B in Teilgebiete $T_1, \ldots, T_N$ soll die Steifigkeitsmatrix A berechnet werden. Wir demonstrieren das Vorgehen anhand einer Dreiecksvernetzung im $\mathbb{R}^2$ und stückweise linearen Basisfunktionen. Für andere Vernetzungen und Basisfunktionen ist das Vorgehen völlig analog.

Globale und lokale Knotennummerierung

Die Eckpunkte der Dreiecke seien $P_1, \ldots, P_m$ (Knoten). Punkte, an denen zwei oder mehr Dreiecke sich berühren, werden nur einfach gezählt. Die Nummerierung der Knoten mit $1, \ldots, m$ wird als **globale Nummerierung** bezeichnet.

Zu jedem Dreieck T_l werden die drei Eckpunkte zusätzlich noch mit P_1^l, P_2^l, P_3^l bezeichnet. Dies ist die **lokale Knotennummerierung**.

Im Speicher werden einerseits die Koordinaten der Knoten $P_1, \ldots, P_m$ zusammen mit der jeweiligen globalen Knotenummer abgelegt; andererseits die **Elementzusammenhangstabelle**, welche für jedes Teilgebiet T_l (Zeilen) folgende Werte (Spalten) enthält:

- Teilgebietnummer l,
- globale Nummer des Knotens P_1^l,
- globale Nummer des Knotens P_2^l,
- globale Nummer des Knotens P_3^l,
- Parameter in der PDE für dieses Teilgebiet (Material, Leitfähigkeit,...),

Durch Knotenkoordinaten und Elementzusammenhangstabelle ist die Vernetzung vollständig beschrieben (IDVID 1130).

Referenzelemente

Für die Berechnung von $a(v_k, v_l)$ müssen Integrale über den Teilgebieten T_l berechnet werden. Um dies effizient durchführen zu können, werden die auftretenden Integrale so transformiert, dass der Integrationsbereich immer gleich aussieht (Referenzdreieck oder allgemeiner: **Referenzelement**).

Als Referenzdreieck wird üblicherweise das Dreieck mit den Eckpunkten $(0,0)$, $(1,0)$, $(0,1)$ verwendet. Liegt dieses in einem ξ_1-ξ_2-Koordinatensystem, ist die Transformation auf das Teilgebiet T_l gegeben durch

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \Phi_l(\xi_1, \xi_2) := P_1^l + \xi_1 \, (P_2^l - P_1^l) + \xi_2 \, (P_3^l - P_1^l). \tag{11.8}$$

Formfunktionen über dem Referenzelement

Jede Basisfunktion v_k soll an genau einem Knoten den Wert 1 annehmen, an allen anderen den Wert 0. Auf einem einzelnen Teilgebiet T_l betrachtet sollen die Basisfunktionen linear verlaufen. Für dreieckige Teilgebiete gibt es somit nur vier mögliche Funktionsverläufe:

- überall 0,
- an einem der drei Eckpunkte 1, sonst 0.

Auf dem Referenzdreieck setzen wir:

- $\varphi_1(\xi_1, \xi_2) := 1 - \xi_1 - \xi_2$ (Wert 1 bei $(0,0)$),
- $\varphi_2(\xi_1, \xi_2) := \xi_1$ (Wert 1 bei $(1,0)$),
- $\varphi_3(\xi_1, \xi_2) := \xi_2$ (Wert 1 bei $(0,1)$).

Basisfunktionen

Für jeden Knoten P_k sei I_k die Indexmenge, die gerade die am Knoten anliegenden Teilgebiete beschreibt. Die Basisfunktion v_k ist dann gegeben durch

$$v_k(x) = \begin{cases} \varphi_{i(k,l)}\big(\Phi_l^{-1}(x)\big), & \text{für } x \in T_l,\ l \in I_k, \\ 0, & \text{sonst}, \end{cases} \tag{11.9}$$

wobei $i(k,l) \in \{1,2,3\}$ gerade so gewählt wird, dass $P^l_{i(k,l)} = P_k$ gilt.

Sind die Funktionen in V auf einem Teil des Randes Null (z.B. als Folge von Dirichlet-Randbedingungen in der PDE), so werden für die entsprechenden Randknoten keine Basisfunktionen benötigt!

Bereichsintegrale

Zu einem Knoten P_k einer Dreieckszerlegung enthalte J_k alle Indizes von benachbarten Knoten. Ist die Bilinearform a durch ein Integral über das Gebiet B gegeben, so gilt $a(v_k, v_l) = 0$, falls $l \notin J_k$. Jede Zeile und jede Spalte der Steifigkeitsmatrix hat also höchstens so viele Nicht-Null-Einträge wie die beiden relevanten Knoten Nachbarn haben (Zeile l hat $|J_k|$ Einträge, Spalte k hat $|J_l|$ Einträge. Insbesondere ist die **Steifigkeitsmatrix dünn besetzt**, was FEM für dreidimensionale Gebiete oder sehr feine Netze überhaupt erst möglich macht.

Die Berechnung der Bereichsintegrale über B erfolgt praktisch immer numerisch, auch wenn manuelles Rechnen möglich wäre. Dazu werden die Integrale über den Teilgebieten T_l zunächst auf das Referenzdreieck transformiert. Anschließend erfolgt die numerische Integration mittels Standardverfahren.

Sei Φ_l wie oben die Abbildung des Referenzdreiecks auf T_l. Dann gilt

$$\int_{T_l} h \,\mathrm{d}b = \int_0^1 \int_0^{1-\xi_1} h\big(\Phi_l(\xi_1,\xi_2)\big)\,|\det J_{\Phi_l}(\xi_1,\xi_2)|\,\mathrm{d}\xi_2\,\mathrm{d}\xi_1 \tag{11.10}$$

mit der zu integrierenden Funktion h, wobei

$$\det J_{\Phi_k}(\xi_1,\xi_2) = [P_2^l - P_1^l]_1\,[P_3^l - P_1^l]_2 - [P_2^l - P_1^l]_2\,[P_3^l - P_1^l]_1 \tag{11.11}$$

(IDVID 1140).

Randbedingungen

Eventuell vorhandene Dirichlet-Randbedingungen sind bereits in den Funktionenraum V bzw. dessen endlichdimensionale Entsprechung V_n integriert und wurden bei der Wahl der Basisfunktionen $v_1, \ldots, v_n$ berücksichtigt (keine Basisfunktionen an Randknoten im Bereich von Dirichlet-Randbedingungen).

Randteile, auf denen Neumann-Randbedingungen gegeben sind, führen üblicherweise zu Kurvenintegralen (2D-Gebiet) oder Oberflächenintegralen (3D-Gebiet) in der schwachen Formulierung der PDE. Diese können völlig analog zu den Bereichsintegralen numerisch berechnet werden. Führen dies hier nur für 2D-Gebiete näher aus.

Sei $\Gamma_2 \subseteq \partial B$ der Teil des Randes, auf dem ein Kurvenintegral

$$\int_{\Gamma_2} h \,\mathrm{d}s \tag{11.12}$$

mit einer Funktion $h : B \to \mathbb{R}$ zu berechnen ist. Das Integral kann in eine Summe von Integralen über die einzelnen Randkanten der Gebietszerlegung geschrieben werden. Ist K eine solche Randkante mit den Endknoten P_l und P_k, so überführt man das Integral

$$\int_K h \,\mathrm{d}s \tag{11.13}$$

in das Integral

$$\int_0^1 h\big(P_l + t\,(P_k - P_l)\big)\,\sqrt{[P_k - P_l]_1^2 + [P_k - P_l]_2^2}\,\mathrm{d}t \tag{11.14}$$

auf der Referenzkante $[0, 1]$ (IDVID 1150).

Ist der Integrand h als Linearkombination der Basisfunktionen $v_1, \ldots, v_n$ gegeben, so haben nur die beiden zu den Endknoten gehörenden Basisfunktionen v_l und v_k Einfluss auf das Integral.

11.5 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 Integrale über jedem Teilgebiet T_l.

12 Fallstudie: Stationäre Wärmeleitung

Lösen ein stationäres Wärmeleitproblem in der Ebene mittels FEM. Dabei bedeutet "stationär", dass wir nicht die zeitliche Entwicklung simulieren, sondern den nach hinreichend langer Wartezeit vom betrachteten physikalischen System eingenommenen Zustand. Sind Wärmezufluss und -abfluss in einem Körper über einen langen Zeitraum konstant, so wird sich eine stabile Temperaturverteilung im Körper einstellen. Dieser stabile Zustand soll simuliert werden.

12.1 Modell

Sei $B \subseteq \mathbb{R}^2$ die Querschnittsfläche eines Kühlkörpers (z.B. einer CPU). Am unteren Rand

$$\Gamma_2 := \{(x, 0) : \ x \in [-1, 1]\} \tag{12.1}$$

erfolgt ein zeitlich konstanter Wärmeeintrag

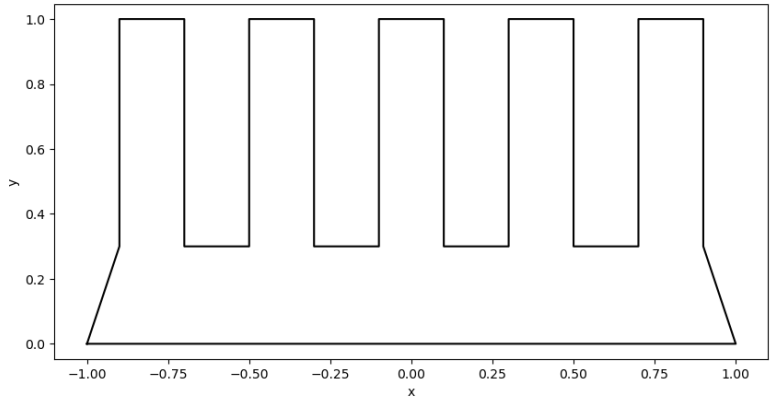
$$g_2(x, 0) := 100\,(1 - x^2). \tag{12.2}$$

Am restlichen Rand Γ_1 liegt die Umgebungstemperatur stets bei

$$g_1(x, y) = 20 \tag{12.3}$$

(Luftstrom).

```python
import matplotlib.pyplot as plt
import numpy as np

# Liste von Eckpunkten zur Beschreibung des polygonalen Gebiets
outline = [[ -1, 0]]
for x in np.linspace( -1 + 0.1, 1 + 0.1, 5, endpoint=False):
    outline.extend([[x, 0.3], [x, 1], [x + 0.2, 1], [x + 0.2, 0.3]])
outline.append([1, 0])

# Randbedingungen
g1 = lambda x, y: 20
g2 = lambda x, y: 100 * (1 - x ** 2)

# Plot der Gebiets
fig, ax = plt.subplots(figsize=(10, 5))
x, y = zip(*(outline + [outline[0]]))
ax.plot(x, y, ' -k')
ax.axis('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



Im Inneren des Kühlkörpers gilt

$$\Delta u = 0 \tag{12.4}$$

für die Temperatur $u : B \to \mathbb{R}$.

12.2 Schwache Formulierung

Die schwache Formulierung erhalten wir als Spezialfall der schwachen Formulierung der Poisson-Gleichung:

Finde $u_0 \in V$, sodass

$$\int_B \nabla u_0 \circ \nabla v \, \mathrm{d}b = \int_{\Gamma_2} g_2 \, v \, \mathrm{d}s - \int_B \nabla u_1 \circ \nabla v \, \mathrm{d}b \tag{12.5}$$

für alle $v \in V$ gilt. Dabei ist

$$V := \{v \in H^1(B) : \ v(x) = 0 \text{ für } x \in \Gamma_1\} \tag{12.6}$$

und $u_1 \in H^1(B)$ ist eine Funktion, die auf Γ_1 gerade g_1 entspricht. Die gesuchte Lösung ist dann

$$u := u_0 + u_1. \tag{12.7}$$

12.3 Gebietszerlegung

Für die Zerlegung des Gebiets B in Dreieckselemente nutzen wir das Python-Modul `pygmsh`.

```
import pygmsh

# Zerlegung in Dreieckselemente
with pygmsh.geo.Geometry() as geom:
    geom.add_polygon(outline, mesh_size=0.05)
```

```
    mesh = geom.generate_mesh()

# Knotenkoordinaten als NumPy -Array
#   axis 0: globale Knotennummer (0, 1,...)
#   axis 1: Koordinate (0 (x), 1 (y))
nodes = mesh.points[:, :2]

# globale Knotennummern der Randkanten als NumPy -Array
#   axis 0: Randkantennummer (0, 1,...)
#   axis 1: lokale Knotennummer (0, 1)
boundary_lines = mesh.cells[0].data

# globale Knotennummern der Dreieckselemente als NumPy -Array
# (Elementzusammenhangstabelle)
#   axis 0: Elementnummer (0, 1,...)
#   axis 1: lokale Knotennummer (0, 1, 2)
triangles = mesh.cells[1].data

del mesh  # nicht mehr benötigt
```

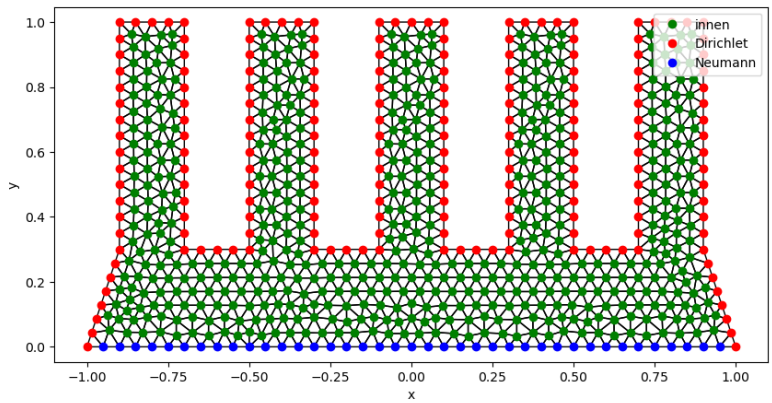
Zur späteren Verwendung trennen wir die von `pygmsh` generierten Knoten nach inneren Knoten, Knoten auf dem Dirichlet-Rand und Knoten auf dem Neumann-Rand.

```
# Knoten nach Lage auswählen (Rand, innen)
b_nodes = set.union(*[{int(n1), int(n2)} for n1, n2 in boundary_lines])
i_nodes = list(set(range(0, nodes.shape[0])) - b_nodes)

# Randknoten in Dirichlet und Neumann trennen
d_nodes = []
n_nodes = []
for n in b_nodes:
    if -1 < nodes[n, 0] < 1 and nodes[n, 1] == 0:
        n_nodes.append(n)
    else:
        d_nodes.append(n)

del b_nodes, boundary_lines  # nicht mehr benötigt
```

Die Gebietszerlegung sollte immer visuell geprüft werden. Insbesondere sollten die Dreiecke möglichst nah an der Form eines gleichseitigen Dreiecks sein.

```
# Plot der Gebietszerlegung
fig, ax = plt.subplots(figsize=(10, 5))

# Dreiecke
```

```
for n1, n2, n3 in triangles:
    n1231 = [n1, n2, n3, n1]
    ax.plot(nodes[n1231, 0], nodes[n1231, 1], ' -k', lw=1)

# Knoten
ax.plot(nodes[i_nodes, 0], nodes[i_nodes, 1], 'og', label='innen')
ax.plot(nodes[d_nodes, 0], nodes[d_nodes, 1], 'or', label='Dirichlet')
ax.plot(nodes[n_nodes, 0], nodes[n_nodes, 1], 'ob', label='Neumann')

# Knotennummern
#for n, (x, y) in enumerate(nodes):
#    ax.text(x + 0.02, y + 0.02, str(n))

ax.axis('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

plt.show()
```



Für den Aufbau der Systemmatrix weiter unten sind die Nachbarschaftsbeziehungen zwischen Knoten sowie zwischen Knoten und Dreieckselementen von Bedeutung.

```
# alle Knoten außer Dirichlet -Rand
ni_nodes = n_nodes + i_nodes

# Dreiecke für jeden Knoten
#   Listenindex ist globale Knotennummer
#   Listeneinträge sind Mengen von Elementnummern
tris_of_nodes = [set() for _ in nodes]
for tri, (n1, n2, n3) in enumerate(triangles):
    tris_of_nodes[n1].add(tri)
```

```
    tris_of_nodes[n2].add(tri)
    tris_of_nodes[n3].add(tri)

# benachbarte Knoten (kleine Knotennummer zuerst)
neighbors = set()
for node, tris_of_node in enumerate(tris_of_nodes):
    for tri in tris_of_node:
        for n in set(triangles[tri, :]) - {node}:
            neighbors.add((node, n) if node < n else (n, node))
```

12.4 Systemmatrix und rechte Seite

Für die numerische Berechnung der in der schwachen Formulierung auftretenden Bereichsintegrale verwenden wir stückweise lineare Funktionen auf der Dreieckszerlegung. Für jeden Knoten wird also eine Basisfunktionen $v_k : B \to \mathbb{R}$ eingeführt, die am entsprechenden Knoten den Wert 1 annimmt und an allen anderen Knoten den Wert 0 hat.

Dirichlet-Randbedingungen

Im kontinuierlichen Modell wird die Lösung u zerlegt in die Summe $u = u_0 + u_1$, wobei u_0 auf dem Dirichlet-Rand Γ_1 Null ist und u_1 auf dem Dirichlet-Rand die Randbedingung erfüllt. Die Funktion u_1 ist abseits von Γ_1 frei wählbar. Sei u_1 also Null im Inneren des Gebiets und auf dem Neumann-Rand.

Im diskretisierten Zustand hat u_0 nur Nicht-Null-Koeffizienten für Basisfunktionen v_k, die zu einem inneren Knoten oder zu einem Knoten auf dem Neumann-Rand gehören. Die Systemmatrix wird also weniger Zeilen bzw. Spalten haben als es Knoten (Basisfunktionen) gibt. Umgekehrt genügen für die diskretisierte Darstellung der Funktion u_1 die Basisfunktionen, die zu einem Knoten auf dem Dirichlet-Rand gehören.

```
# Abbildung der Knotennummern (ohne Dirichlet -Rand) auf Matrixzeilen/
↪  -spalten
node2idx = {n: idx for idx, n in enumerate(ni_nodes)}
```

Die Systemmatrix wird dünn besetzt sein, sodass wir die Einträge zunächst in Listen sammeln und dann später daraus mit SciPy eine dünn besetzte Matrix generieren.

```
# leere (dünn besetzte) Matrix, rechte Seite zunächst überall 0
Adata = []
Acols = []
Arows = []
b = np.zeros(len(ni_nodes), dtype=float)
```

Gradient der Basisfunktionen

Sei P_k der Knoten zur globale Knotennummer k. Der Gradient der zugehörigen Basisfunktion v_k ist Null auf allen Dreieckselemente, die nicht an den Knoten P_k grenzen. Auf an den Knoten P_k angrenzenden Dreiecken ist der Gradient konstant.

Die Berechnung des Gradients von v_k auf einem Dreieck mit Eckknoten P_k, P_l, P_m kann leicht durch Transformation des Dreiecks auf das Referenzdreieck mit den Eckpunkten $(0,0)$, $(1,0)$, $(0,1)$ erfolgen:

Die Transformation der Ecken des Referenzdreiecks auf die Eckknoten P_k, P_l, P_m ist durch

$$\Phi(\xi,\eta) := P_k + \xi_1 (P_l - P_k) + \xi_2 (P_m - P_k) \tag{12.8}$$

gegeben. Die zum betrachteten Dreieck und zur Basisfunktion v_k passende Formfunktion ist φ_1 (vgl. Section 11.4). Mit ihr gilt

$$(v_k \circ \Phi)(\xi,\eta) = \varphi_1(\xi,\eta). \tag{12.9}$$

Andererseits liefert die Kettenregel

$$\nabla(v_k \circ \Phi)(\xi,\eta) = J_\Phi(\xi,\eta)^{\mathrm{T}} \, (\nabla v_k)(\Phi(\xi,\eta)) \tag{12.10}$$

(IDVID 1210), wobei

$$J_\Phi(\xi,\eta) = \begin{bmatrix} [P_l - P_k]_1 & [P_m - P_k]_1 \\ [P_l - P_k]_2 & [P_m - P_k]_2 \end{bmatrix} \tag{12.11}$$

die Jacobi-Matrix zu Φ ist. Somit

$$\begin{aligned} (\nabla v_k)(\Phi(\xi,\eta)) &= J_\Phi(\xi,\eta)^{-\mathrm{T}} \, \nabla(v_k \circ \Phi)(\xi,\eta) \\ &= J_\Phi(\xi,\eta)^{-\mathrm{T}} \, \nabla\varphi_1(\xi,\eta). \end{aligned} \tag{12.12}$$

Analog folgt

$$(\nabla v_l)(\Phi(\xi,\eta)) = J_\Phi(\xi,\eta)^{-\mathrm{T}} \, \nabla\varphi_2(\xi,\eta). \tag{12.13}$$

Bereichsintegrale

Für die Berechnung der Einträge der Systemmatrix und der rechten Seite benötigen wir die Bereichsintegrale

$$\int_B (\nabla v_k)^{\mathrm{T}} \, \nabla v_l \, \mathrm{d}b \tag{12.14}$$

für alle Paare von Basisfunktionen und auch für $k = l$.

Sind die entsprechenden Knoten P_k und P_l nicht benachbart, so ist das Integral Null. Bei benachbarten Knoten ist der Integrand nur auf den beiden Dreiecken von Null verschieden, die sowohl P_k als auch P_l als Eckknoten haben. Bezeichnen wir diese beien Dreiecke mit T und $\tilde{T}$, so gilt

$$\int_B (\nabla v_k)^{\mathrm{T}} \, \nabla v_l \, \mathrm{d}b = \int_T (\nabla v_k)^{\mathrm{T}} \, \nabla v_l \, \mathrm{d}b + \int_{\tilde{T}} (\nabla v_k)^{\mathrm{T}} \, \nabla v_l \, \mathrm{d}b. \tag{12.15}$$

Auf T und $\tilde{T}$ ist der Integrand jeweils konstant.

Sei P_m der dritte Eckknoten im Dreieck T und sei Φ die Transformation des Referenzdreiecks auf das Dreieck T (vgl. oben). Dann gilt

$$\int_T (\nabla v_k)^{\mathrm{T}} \nabla v_l \,\mathrm{d}b = \int_0^1 \int_0^{1-\xi} h(\xi,\eta)\,|\det J_\Phi(\xi,\eta)|\,\mathrm{d}\eta\,\mathrm{d}\xi \tag{12.16}$$

mit

$$\begin{aligned} h(\xi,\eta) &= \big(\nabla v_k(\Phi(\xi,\eta))\big)^{\mathrm{T}} \nabla v_l(\Phi(\xi,\eta)) \\ &= \big(J_\Phi(\xi,\eta)^{-\mathrm{T}} \nabla\varphi_1(\xi,\eta)\big)^{\mathrm{T}} J_\Phi(\xi,\eta)^{-\mathrm{T}} \nabla\varphi_2(\xi,\eta) \\ &= \big(\nabla\varphi_1(\xi,\eta)\big)^{\mathrm{T}} J_\Phi(\xi,\eta)^{-1} J_\Phi(\xi,\eta)^{-\mathrm{T}} \nabla\varphi_2(\xi,\eta) \\ &= \big(\nabla\varphi_1(\xi,\eta)\big)^{\mathrm{T}} \big(J_\Phi(\xi,\eta)^{\mathrm{T}} J_\Phi(\xi,\eta)\big)^{-1} \nabla\varphi_2(\xi,\eta) \end{aligned} \tag{12.17}$$

und

$$\det J_\Phi(\xi,\eta) = \sqrt{\det\big(J_\Phi(\xi,\eta)^{\mathrm{T}} J_\Phi(\xi,\eta)\big)}. \tag{12.18}$$

Die symmetrische und positiv definite Matrix $J_\Phi(\xi,\eta)^{\mathrm{T}} J_\Phi(\xi,\eta)$ nimmt für alle ξ und η den gleichen Wert

$$J_\Phi(\xi,\eta)^{\mathrm{T}} J_\Phi(\xi,\eta) = \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix} \tag{12.19}$$

mit Zahlen $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ an. Auch die Gradienten der Formfunktionen hängen nicht von ξ und η ab:

$$\nabla\varphi_1(\xi,\eta) = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \qquad \nabla\varphi_2(\xi,\eta) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{12.20}$$

Der Integrand ist also konstant und der Integrationsbereich (Referenzdreieck) hat den Flächeninhalt $\frac{1}{2}$. Somit

$$\begin{aligned} \int_T (\nabla v_k)^{\mathrm{T}} \nabla v_l \,\mathrm{d}b &= \frac{1}{2}\,\frac{1}{\alpha\,\gamma - \beta^2} \begin{bmatrix} -1 & -1 \end{bmatrix} \begin{bmatrix} \gamma & -\beta \\ -\beta & \alpha \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \sqrt{\alpha\,\gamma - \beta^2} \\ &= \frac{\beta - \gamma}{2\,\sqrt{\alpha\,\gamma - \beta^2}}. \end{aligned} \tag{12.21}$$

Für $k = l$ erhält man völlig analog (zweimal φ_1 statt φ_1 mit φ_2)

$$\int_T (\nabla v_k)^{\mathrm{T}} \nabla v_k \,\mathrm{d}b = \frac{\alpha + \gamma - 2\,\beta}{2\,\sqrt{\alpha\,\gamma - \beta^2}}. \tag{12.22}$$

Ist P_k oder P_l ein Knoten auf dem Dirichlet-Rand, so spielt der Wert $a(v_k, v_l)$ keine Rolle in der Systemmatrix, taucht aber bei der Berechnung des Bereichsintegrals $a(u_1, v_k)$ bzw. $a(u_1, v_l)$ in der rechten Seite b auf.

```
# Matrix A ohne Diagonale, Dirichlet -Anteil von b
for n1, n2 in neighbors:

    # a(v_n1, v_n2) berechnen
    int_gradgrad = 0
    for tri in tris_of_nodes[n1] & tris_of_nodes[n2]:  # höchstens 2
    ↪  Läufe
        n3 = next(iter(set(triangles[tri, :]) - {n1, n2}))  # dritter
        ↪  Knoten des Dreiecks
        J = np.vstack((nodes[n2, :] - nodes[n1, :], nodes[n3, :] -
        ↪  nodes[n1, :])).T
        JJ = np.matmul(J.T, J)
        detJJ = JJ[0, 0] * JJ[1, 1] - JJ[0, 1] ** 2
        int_gradgrad += (JJ[0, 1] - JJ[1, 1]) / (2 * np.sqrt(detJJ))

    # in Matrix A oder in rechte Seite schreiben?
    n1d = n1 in d_nodes
    n2d = n2 in d_nodes
    if not n1d and not n2d:
        i1 = node2idx[n1]
        i2 = node2idx[n2]
        Acols.append(i1)
        Arows.append(i2)
        Adata.append(int_gradgrad)
        Acols.append(i2)
        Arows.append(i1)
        Adata.append(int_gradgrad)
    elif n1d and not n2d:
        i2 = node2idx[n2]
        b[i2] += -g1(nodes[n1, 0], nodes[n1, 1]) * int_gradgrad
    elif n2d and not n1d:
        i1 = node2idx[n1]
        b[i1] += -g1(nodes[n2, 0], nodes[n2, 1]) * int_gradgrad
    else:  # Dirichlet -Randkante
        pass

# Diagonale von A
for n1 in ni_nodes:
    i1 = node2idx[n1]

    # a(v_n1, v_n1) berechnen
    int_gradgrad = 0
    for tri in tris_of_nodes[n1]:
```

```
        n2, n3 = set(triangles[tri, :]) - {n1}
        J = np.vstack((nodes[n2, :] - nodes[n1, :], nodes[n3, :] -
        ↪  nodes[n1, :])).T
        JJ = np.matmul(J.T, J)
        detJJ = JJ[0, 0] * JJ[1, 1] - JJ[0, 1] ** 2
        int_gradgrad += (JJ[0, 0] + JJ[1, 1] - 2 * JJ[0, 1]) / (2 *
        ↪  np.sqrt(detJJ))

    # in Matrix schreiben
    Acols.append(i1)
    Arows.append(i1)
    Adata.append(int_gradgrad)
```

Neumann-Randbedingungen

In die rechte Seite b müssen noch die Integrale

$$\int_{\Gamma_2} g_2 \, v_k \, \mathrm{d}s \tag{12.23}$$

eingebracht werden, wobei k die globalen Knotennummern der Knoten auf dem Neumann-Rand durchläuft.

Das Integrationsgebiet Γ_2 kann in einzelne Randkanten zerlegt werden, wobei für jede Kante die Endknoten entweder beide auf dem Neumann-Rand liegen oder ein Endknoten auf dem Neumann- und ein Endknoten auf dem Dirichlet-Rand liegt.

In das Integral über eine Randkante gehen nur zwei Basisfunktionen (zwei Neumann-Knoten) oder nur eine Basisfunktion (Dirichlet-Knoten und Neumann-Knoten) ein.

Seien P_k und P_l die Endknoten einer Randkante K. Dann ist durch

$$\Psi(\zeta) := P_k + \zeta \, (P_l - P_k) \tag{12.24}$$

eine Abbildung des Intervalls $[0, 1]$ auf die Randkante gegeben. Damit erhalten wir

$$\int_K g_2 \, v_k \, \mathrm{d}s = \int_0^1 g_2(\Psi(\zeta)) \, (1 - \zeta) \sqrt{[P_k - P_l]_1^2 + [P_k - P_l]_2^2} \, \mathrm{d}\zeta \tag{12.25}$$

und

$$\int_K g_2 \, v_l \, \mathrm{d}s = \int_0^1 g_2(\Psi(\zeta)) \, \zeta \sqrt{[P_k - P_l]_1^2 + [P_k - P_l]_2^2} \, \mathrm{d}\zeta, \tag{12.26}$$

wobei wir

$$v_k(\Psi(\zeta)) = 1 - \zeta \qquad \text{und} \qquad v_l(\Psi(\zeta)) = \zeta \tag{12.27}$$

verwendet haben.

Diese Integrale können leicht mittels numerischer Integration berechnet werden.

```
from scipy.integrate import quad

# Neumann -Anteil von b
for n1, n2 in neighbors:

    # mindestens ein Neumann -Randknoten
    n1n = n1 in n_nodes
    n2n = n2 in n_nodes
    n1d = n1 in d_nodes
    n2d = n2 in d_nodes
    if not((n1n and n2n) or (n1n and n2d) or (n1d and n2n)):
        continue  # keine Neumann -Randkante

    # Integral von g_2 * v_n1 und von g_2 * v_n2
    P1 = nodes[n1, :]
    P2 = nodes[n2, :]
    distP1P2 = np.sqrt((P1[0] - P2[0]) ** 2 + (P1[1] - P2[1]) ** 2)
    if n1n:  # Basisfunktion bei D -Knoten trägt nicht zum Integral bei
        i1 = node2idx[n1]
        func = lambda t: g2(*(P1 - t * (P2 - P1))) * (1 - t)
        b[i1] += distP1P2 * quad(func, 0, 1)[0]
    if n2n:
        i2 = node2idx[n2]
        func = lambda t: g2(*(P1 - t * (P2 - P1))) * t
        b[i2] += distP1P2 * quad(func, 0, 1)[0]
```

12.5 Lösen des Gleichungssystems

Das Gleichungssystem kann mit Standardverfahren für dünn besetzte Matrizen gelöst werden.

```
import scipy.sparse as ss

# Matrix A als SciPy -Objekt
A = ss.csr_array(
    (Adata, (Arows, Acols)),
    shape=(len(ni_nodes), len(ni_nodes))
)

# Gleichungssystem lösen
u0 = ss.linalg.spsolve(A, b)
```

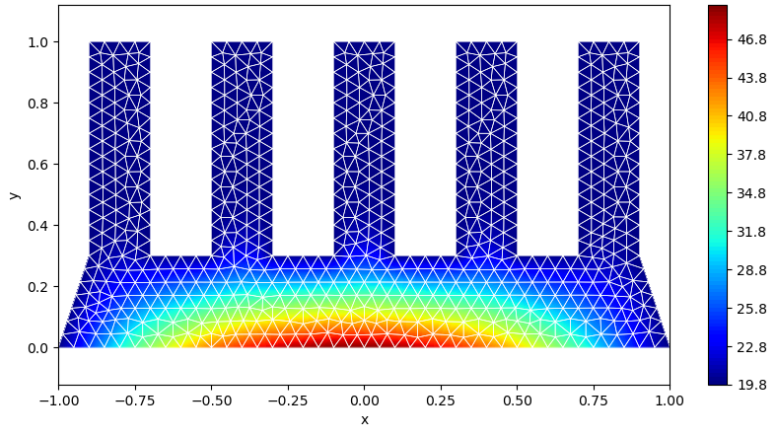
12.6 Zusammensetzen der Lösung

Das Array `u0` enthält nun die Funktionswerte der Lösungsfunktion an den inneren Knoten und an den Randknotes des Neumann-Randes. Auf dem Dirichlet-Rand ist die Lösung bekannt.

```
# Lösung zusammensetzen
u = np.zeros(len(nodes), dtype=float)
u[d_nodes] = g1(nodes[d_nodes, 0], nodes[d_nodes, 1])
u[ni_nodes] = u0
```

Für die Darstellung der Lösung kann der Bereich zwischen den Knoten linear interpoliert werden. Dies entspricht der kontinuierlichen Darstellung der Lösung als Summe stückweise linearer Ansatzfunktionen.

```
# Plot der Lösung
fig, ax = plt.subplots(figsize=(10, 5))

# Lösung
contour_plot = ax.tricontourf(
    nodes[:, 0], nodes[:, 1], triangles, u, cmap='jet', levels=100
)
fig.colorbar(contour_plot, ax=ax, orientation='vertical')

# Dreiecke
for n1, n2, n3 in triangles:
    n1231 = [n1, n2, n3, n1]
    ax.plot(nodes[n1231, 0], nodes[n1231, 1], ' -w', lw=0.5)

ax.axis('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```



12.7 Realitätsnähe

Die Modellierung mit Dirichlet-Randbedingungen ist wenig realitätnah. Um die Oberflächentemperatur konstant bei 20°C zu halten, müsste das Bauteil beispielsweise von einer 20°C warmen Luftströmung umströmt werden, die beliebig viel Wärme aufnehmen kann. Realistischer sind Robin-Randbedingungen, die die Wärmeabgabe proportional zur Differenz zwischen Oberflächen- und Umgebungstemperatur modellieren (vgl. Section 6.2).

Bei der schwachen Formulierung mit Robin-Randbedingungen erhält man sowohl in der Bilinearform a als auch in der rechten seite b zusätzlich ein Kurvenintegral über den Robin-Rand.

13 Praktikum 1

13.1 Effiziente Berechnungen mit NumPy

Aufgabe P1.1

Machen Sie sich mit dem Python-Paket NumPy detaillierter vertraut, insbesondere mit Blick auf das effiziente (Code-Menge, Rechenzeit, Speicherbedarf) Implementieren von Operationen auf großen Arrays. Lesen Sie dazu folgenden Abschnitte von DSAI und probieren Sie die Code-Beispiele aus:

- Advanced Indexing
- Vectorization
- Copies and Views
- Efficiency Considerations

> **Nebenbemerkung**
>
> Das E-Book Data Science and Artificial Intelligence for Undergraduates wurde für Studienanfänger im Fach Data-Science geschrieben, welche über keinerlei Programmierkenntnisse verfügen. Deshalb sind einige Erklärungen etwas ausführlicher gehalten als für unsere Zwecke nötig. Im Folgenden wird diese Quelle mit **DSAI** abgekürzt.

13.2 Keine Schleifen

Lösen Sie die folgenden Aufgabe mit NumPy ohne Verwendung von Python-Schleifen oder ähnlichen Konstrukten (List-Comprehensions,...).

Aufgabe P1.2

Finden Sie den größten Werte aus allen Einträgen in zwei Arrays. Nutzen Sie `np.max` oder `np.maximum` oder beide.

Testen Sie Ihren Code mit

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{und} \quad \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}. \tag{13.1}$$

```
# Ihre Lösung
```

Aufgabe P1.3

Prüfen Sie für ein gegebenes Integer-Array, ob dieses eine 1 enthält.

Testen Sie Ihren Code mit

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{und auch mit} \quad \begin{bmatrix} 0 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}. \tag{13.2}$$

```
# Ihre Lösung
```

Aufgabe P1.4

Setzen Sie in einem Array alle Werte auf Null, die zwischen -1 und 1 liegen.

Testen Sie Ihren Code mit

$$\begin{bmatrix} -2 & -0.5 & 0 \\ 0.4 & 5 & 0.9 \\ 1 & 8 & 9. \end{bmatrix} \tag{13.3}$$

```
# Ihre Lösung
```

Aufgabe P1.5

Schreiben Sie ein Funktion, die einen Integer-Wert n übernimmt und ein $n \times n$-Array mit Einsen am Rand und Nullen im Inneren zurückgibt.

Beispiel für $n = 5$:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{13.4}$$

```
# Ihre Lösung
```

Aufgabe P1.6

Addieren Sie die erste Zeile eines Arrays zu allen anderen Zeilen des Arrays.

Testen Sie Ihren Code mit

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}. \tag{13.5}$$

```
# Ihre Lösung
```

Aufgabe P1.7

Aus gegebenes quadratisches Array mit ungerader Zeilen- bzw. Spaltenanzahl soll wie folgt verändert werden:

- Alle Einträge am Rand bleiben erhalten.
- Alle inneren Einträge werden auf Null gesetzt.
- Der Eintrag in der Mitte enthält den Mittelwert aller Randelemente.

Beispiel:

$$\text{Original:} \quad \begin{bmatrix} -1 & 2 & -1 & 2 & -1 \\ 2 & -2 & 3 & -2 & 2 \\ -1 & 3 & -3 & 3 & -1 \\ 2 & -2 & 3 & -2 & 2 \\ -1 & 2 & -1 & 2 & -1 \end{bmatrix}, \tag{13.6}$$

$$\text{Ergebnis:} \quad \begin{bmatrix} -1 & 2 & -1 & 2 & -1 \\ 2 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0.5 & 0 & -1 \\ 2 & 0 & 0 & 0 & 2 \\ -1 & 2 & -1 & 2 & -1 \end{bmatrix}. \tag{13.7}$$

```
# Ihre Lösung
```

14 Praktikum 2

In diesem Praktikum werden wir Schritt für Schritt die Momentangeschwindigkeit eines Fahrrads aus einem aufgezeichneten GPS-Track berechnen.

14.1 Daten lesen und prüfen

Aufgabe P2.1

Lesen Sie die CSV-Datei track.csv ein (z.B. mit dem `csv`-Modul). Diese hat drei Spalten für die Fahrzeit, die bisher zurückgelegte Strecke in der Ebene und die Höhe des Messpunktes. 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.
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.

Alle Strecken in Meter, alle Zeiten in Sekunden.

```
# Ihre Lösung
```

Aufgabe P2.2

Prüfen Sie, dass die gefahrene Strecke von Punkt zu Punkt anwächst, also nicht rückwärts gefahren wurde.

```
# Ihre Lösung
```

Aufgabe P2.3

Prüfen Sie, ob die Zeitabstände zwischen zwei benachbarten Punkten tatsächlich 2 Sekunden betragen.

```
# Ihre Lösung
```

14.2 Preprocessing

Aufgabe P2.4

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
```

Aufgabe P2.5

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
```

14.3 Numerische Differentiation

Aufgabe P2.6

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).

```
# Ihre Lösung
```

Aufgabe P2.7

Lösen Sie die vorhergehende Aufgabe nochmals (nur zentrale Differenzen zwischen den Punkten), aber mit weniger Messpunkten (z.B. nur jeder 3. oder nur jeder 6.). Wie ändert sich der Verlauf der Geschwindigkeit?

```
# Ihre Lösung
```

14.4 Schwankungen wegen GPS-Fehler?

Aufgabe P2.8

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
```

15 Praktikum 3

Wir setzen Praktikum 2 fort und ergänzen dieses um alternative Vorgehensweisen.

15.1 Daten lesen

Aufgabe P3.1

Lesen Sie die Zeit- und Wegdaten wie in Praktikum 2 ein.

```
# Ihre Lösung
```

15.2 Ableitung als Umkehrung der Stammfunktion

Zu einem gegebenem Geschwindigkeitsverlauf kann durch Matrix-Vektor-Multiplikation die zurückgelegte Entfernung als Stammfunktion des Geschwindigkeitsverlaufs berechnet werden. Umgekehrt liefert das Lösen des entsprechenden linearen Gleichungssystems die Ableitung des Weges, also die Geschwindigkeit.

Aufgaben P3.2

Stellen Sie die Matrix zur Berechnung des Weges aus der Geschwindigkeit auf, vgl. Section 4.3. Nutzen Sie folgende Diskretisierung:

- Die Wegfunktion (Stammfunktion) soll an den durch die verfügbaren Daten gegebenen Stützstellen berechnet werden.
- Die Geschwindigkeitsfunktion soll stückweise konstant sein und die Stützstellen sollen mit denen der Wegfunktion übereinstimmen.

Aufgaben P3.3

Berechnen Sie die Geschwindigkeit aus dem Weg, indem Sie das durch die eben aufgestellte Matrix gegebene Gleichungssystem lösen (z.B. mit numpy.linalg.solve).

Stellen Sie das Ergebnis grafisch dar und vergleichen Sie mit den Ergebnissen aus Praktikum 2. Nützlich: matplotlib.pyplot.stairs.

15.3 Problematische Diskretisierung

Aufgaben P3.4

Lösen Sie die beiden vorhergehenden Aufgaben nochmals, nun aber mit einer stückweise linearen Geschwindigkeitsfunktion. Falls nötig, können Sie annehmen, dass die Geschwindigkeit zur Zeit 0 bei 0 km/h lag.

Aufgaben P3.5

Suchen Sie nach einer Erklärung für das beobachtete Lösungsverhalten. Lösen Sie dazu das Gleichungssystem für folgende Situationen sowohl numerisch als auch manuell:

- Zeitmessungen $(0, 2, 4, 6, 8, 10)$, Wegmessungen $(0, 1, 1, 1, 1, 1)$ (losfahren und dann anhalten),
- Zeitmessungen $(0, 2, 4, 6, 8, 10)$, Wegmessungen $(0, 1, 2, 3, 4, 5)$ (konstante Geschwindigkeit).

16 Praktikum 4

Wir setzen das Lösen der Section 7.4 für das Gebiet

$$B := [0, 1] \times [0, 2] \tag{16.1}$$

am Computer um.

16.1 Lösungsalgorithmus

Aufgabe P4.1

Stellen Sie die Systemmatrix des zu lösenden Gleichungssystems für die grobe Diskretisierung mit $n_x = 4$, $n_y = 5$ manuell auf.

Aufgabe P4.2

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, z.B. mit `matplotlib.pyplot.contourf` oder mit `plotly.graph_objects.Surface` (siehe auch Passing x and y data to 3D Surface Plot).

```
# Ihre Lösung
```

16.2 Interpretation der Lösungen

Aufgaben P4.3

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.

Aufgaben P4.4

Wie könnte man f wählen um die Verformung einer Membran beim Belasten mit einem starren Quader wenigstens grob zu simulieren?

17 Praktikum 5

17.1 Aufgabe P5.1

Implementieren Sie das Finite-Differenzen-Verfahren für die 2D-Wellengleichung aus Übung 5.

Achten Sie bei der Wahl der Diskretisierungsparameter auf den Speicherbedarf!

```
# Ihre Lösung
```

17.2 Aufgabe P5.2

Visualisieren Sie die berechneten Lösungen für folgende Anfangsbedingungen:

- $$f(x,y) = \left(0.5 - \left|\frac{x}{a} - 0.5\right|\right)\,\left(0.5 - \left|\frac{y}{b} - 0.5\right|\right), \tag{17.1}$$
- $$f(x,y) = g(x,y)\exp\left(-200\,\left(\left(\frac{x}{a} - 0.5\right)^2 + \left(\frac{y}{b} - 0.5\right)^2\right)\right), \tag{17.2}$$

  wobei

  $$g(x,y) = \left(1 - \left(2\,\frac{x}{a} - 1\right)^8\right)\,\left(1 - \left(2\,\frac{y}{b} - 1\right)^8\right) \tag{17.3}$$

  die Randbedingung sichert.

```
# Ihre Lösung
```

18 Praktikum 6

18.1 Aufgabe P6.1

Lösen Sie die PDE aus Aufgabe Ü6.3 am Computer mit den Knoten und den Basisfunktionen aus Aufgabe Ü6.5 für große n und vergleichen Sie die Lösung mit der exakten Lösung. Die Integrale für die rechte Seite können zum Beispiel mit `scipy.integrate.quad` berechnet werden.

```
# Ihre Lösung
```

19 Übung 1

19.1 Auffrischung Stetigkeit

Aufgabe Ü1.1

Sei $f : \mathbb{R} \to \mathbb{R}$ eine Funktion. Geben Sie eine Definition folgender Begriffe an:

- Stetigkeit an einer Stelle x_0.
- Stetigkeit der Funktion f.

Aufgabe Ü1.2

Sind folgende Funktionen f auf $\mathbb{R}$ stetig?

- $f(x) = \begin{cases} 1, & \text{für } x \geq 0, \\ 0, & \text{für } x < 0, \end{cases}$
- $f(x) = \begin{cases} 1, & \text{für } x \neq 0, \\ 0, & \text{für } x = 0. \end{cases}$

Aufgabe Ü1.3

Geben Sie den größtmöglichen Definitionsbereich der Funktion

$$f(x) = \frac{1}{|x-1|} \tag{19.1}$$

an. Ist diese Funktion stetig?

19.2 Auffrischung 1D-Differenzialrechnung

Aufgabe Ü1.4

Sei $f : \mathbb{R} \to \mathbb{R}$ eine stetige Funktion. Geben Sie eine Definition folgender Begriffe an:

- Differenzenquotient,
- Differentialquotient,
- Differenzierbarkeit von f an einer Stelle x_0,
- Differenzierbarkeit von f,
- Ableitung von f,
- Stetige Differenzierbarkeit von f.

Aufgabe Ü1.5

Zeigen Sie, dass die Funktion

$$f(x) = \begin{cases} \sqrt{x}, & \text{für } x \geq 0, \\ -\sqrt{-x}, & \text{für } x < 0, \end{cases} \tag{19.2}$$

in $x = 0$ stetig, aber nicht differenzierbar ist.

Aufgabe Ü1.6

Zeigen Sie, dass die Funktion

$$f(x) = \begin{cases} x^2 \sin(\frac{1}{x}), & \text{für } x \neq 0, \\ 0, & \text{für } x = 0, \end{cases} \tag{19.3}$$

auf $\mathbb{R}$ differenzierbar, aber die Ableitung nicht stetig ist.

Aufgabe Ü1.7

Bis zu welcher Ordnung besitzt die Funktion

$$f(x) = \begin{cases} x^2, & \text{für } x < 0, \\ x^3, & \text{für } x \geq 0, \end{cases} \tag{19.4}$$

stetige Ableitungen?

19.3 Auffrischung mehrdimensionale Differenzialrechung

Aufgabe Ü1.8

Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Funktion. Geben Sie eine Definition folgender Begriffe an:

- Stetigkeit in einem Punkt $x_0 \in \mathbb{R}^n$,
- partielle Ableitung,
- Gradient,
- Richtungsableitung,
- Hesse-Matrix.

Aufgabe Ü1.9

Ist die Funktion

$$f(x,y) = \begin{cases} \frac{x\,y}{x^2+y^2}, & \text{für } (x,y) \neq (0,0), \\ 0, & \text{für } (x,y) = (0,0), \end{cases} \tag{19.5}$$

stetig in $(0,0)$?

Aufgabe Ü1.10

Geben Sie für die Funktion aus der vorhergehenden Aufgabe die Richtung des steilsten Anstiegs im Punkt $(0,1)$ an.

Aufgabe Ü1.11

Sei $f : \mathbb{R}^{\times} \to \mathbb{R}^m$ eine vektorwertige Funktion. Was versteht man unter der Jacobi-Matrix von f?

20 Übung 2

20.1 Auffrischung Taylor-Formel

Aufgabe Ü2.1

Sei $f : \mathbb{R} \to \mathbb{R}$ eine beliebig oft differenzierbare Funktion und sei $x_0 \in \mathbb{R}$. Geben Sie Formeln für folgende Objekte an:

- Taylor-Polynom vom Grad 1 zu f in x_0,
- Taylor-Polynom vom Grad 2 zu f in x_0,
- Taylor-Polynom vom Grad k zu f in x_0,
- Taylor-Restglied in Lagrange-Form zum Taylor-Polynom vom Grad k.

Aufgabe Ü2.2

Finden Sie das Taylor-Polynom vom Grad 3 bei $x_0 = 0$ zu $f(x) = \sin x$.

Wie groß ist höchstens der Fehler, wenn auf dem Intervall $[-\frac{\pi}{4}, \frac{\pi}{4}]$ dieses Taylor-Polynom statt f ausgewertet wird?

Aufgabe Ü2.3

Sei $f : \mathbb{R}^2 \to \mathbb{R}$ eine beliebig oft differenzierbare Funktion und sei $(x_0, y_0) \in \mathbb{R}^2$. Geben Sie eine Formel für das Taylor-Polynom vom Grad 2 zu f in (x_0, y_0) an.

Aufgabe Ü2.4

Finden Sie das Taylor-Polynom vom Grad 2 im Punkt $(0, 1)$ für die Funktion $f(x, y) = y \ln(y - 3\,x)$.

20.2 Auffrischung 1D-Integralrechnung

Aufgabe Ü2.5

Geben Sie Definitionen für folgende Begriffe bzgl. einer Funktion $f : \mathbb{R} \to \mathbb{R}$ an:

- Stammfunktion von f,
- unbestimmtes Integral von f,
- bestimmtes Integral von f auf dem Intervall $[a, b]$,
- uneigentliches Integral auf dem Intervall $[0, \infty]$.

Formulieren Sie den Hauptsatz der Differential- und Integralrechnung.

Aufgabe Ü2.6

Geben Sie eine Formel für die Berechnung des Integrals

$$\int_a^b f(x)\, g'(x)\, \mathrm{d}x \tag{20.1}$$

aus dem Integral des Produkts $f'(x)g(x)$ an (partielle Integration).

Aufgabe Ü2.7

Berechnen Sie

$$\int_0^1 x\, \mathrm{e}^x\, \mathrm{d}x \tag{20.2}$$

mittels partieller Integration.

Aufgabe Ü2.8

Berechnen Sie

$$\int_0^1 x^2\, (x^3 + 2)^6\, \mathrm{d}x \tag{20.3}$$

mittels geeigneter Substitution eines Teils des Integranden durch eine neue Variable. Geben Sie auch eine allgemeine Formel für Ihr Vorgehen an.

21 Übung 3

21.1 Auffrischung Bereichsintegrale

Aufgabe Ü3.1

Berechnen Sie das Doppelintegral

$$\int_0^1 \int_0^x \sqrt{x\,y}\,\mathrm{d}y\,\mathrm{d}x. \tag{21.1}$$

Aufgabe Ü3.2

Sei $f(x, y) : \mathbb{R}^2 \to \mathbb{R}$ eine Funktion. Wiederholen Sie die Begriffe

- Bereichsintegral bzgl. einer Teilmenge $B \subseteq \mathbb{R}^2$,
- Normalbereich bzgl. x-Achse,
- Normalbereich bzgl. y-Achse.

Was versteht man unter einem Normalbereich im $\mathbb{R}^3$?

Aufgabe Ü3.3

Die durch

$$x = 0, \qquad y = 0, \qquad x + y = 4 \tag{21.2}$$

beschriebenen drei Flächen begrenzen ein (unendlich langes) dreiseitiges Prisma im $\mathbb{R}^3$. Aus diesem wird durch Schnitte entlang der beiden durch

$$z = 0, \qquad z = x^2 + y^2 \tag{21.3}$$

beschriebenen Flächen ein endliches Stück herausgeschnitten. Berechnen Sie das Volumen dieses Stücks.

21.2 Auffrischung Kurvenintegrale

Kurvenintegrale erster Art

Sei durch

$$K : [a, b] \to \mathbb{R}, \quad t \mapsto (x(t), y(t)) \tag{21.4}$$

eine Kurve in der Ebene gegeben. Weiter sei $f : \mathbb{R}^2 \to \mathbb{R}$ eine Funktion. Bezeichnen mit

$$\int_K f\,\mathrm{ds} := \int_a^b f(x(t), y(t))\,\sqrt{x'(t)^2 + y'(t)^2}\,\mathrm{d}t \tag{21.5}$$

das **Kurvenintegral erster Art** von f entlang der Kurve K.

Aufgabe Ü3.4

Berechnen Sie

$$\int_K x \,\mathrm{ds} \tag{21.6}$$

für eine Kreislinie K mit Radius 1 um den Koordinatenursprung.

Aufgabe Ü3.5

Berechnen Sie die Länge des zwischen $(0,0)$ und $(1,1)$ liegenden Stücks K der Parabel $y = x^2$, indem Sie das Integral

$$\int_K 1 \,\mathrm{ds} \tag{21.7}$$

auswerten.

Kurvenintegrale zweiter Art

Sei durch

$$K : [a,b] \to \mathbb{R}, \quad t \mapsto (x(t), y(t)) \tag{21.8}$$

eine Kurve in der Ebene gegeben. Weiter sei $f : \mathbb{R}^2 \to \mathbb{R}^2$ eine Funktion. Bezeichnen mit

$$\int_K f \,\mathrm{dv} := \int_a^b f_1(x(t), y(t))\, x'(t) + f_2(x(t), y(t))\, y'(t) \,\mathrm{d}t \tag{21.9}$$

das **Kurvenintegral zweiter Art** von f entlang der Kurve K.

Aufgabe Ü3.6

Berechnen Sie

$$\int_K f \,\mathrm{dv} \tag{21.10}$$

für

$$f(x,y) = \begin{bmatrix} x^2 + y^2 \\ x\,y \end{bmatrix} \tag{21.11}$$

entlang der Strecke von $(0,0)$ nach $(2,2)$.

Aufgabe Ü3.7

Beschreibe

$$f(x,y) = \begin{bmatrix} 2\,x + y \\ x + 2\,y \end{bmatrix} \tag{21.12}$$

eine Strömung in der Ebene. Berechnen Sie die verrichtete Arbeit beim Durchqueren der Strömung entlang des Viertelkreises K um den Ursprung von $(2,0)$ nach $(0,2)$ als Kurvenintegral

$$\int_K f \,\mathrm{dv}. \tag{21.13}$$

21.3 Auffrischung Oberflächenintegrale

Oberflächenintegrale erster Art

Sei durch

$$S : [a,b] \times [c,d] \to \mathbb{R}^3, \qquad (u,v) \mapsto \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix} \tag{21.14}$$

eine Fläche im Raum gegeben und sei $f : \mathbb{R}^3 \to \mathbb{R}$ eine Funktion. Bezeichnen mit

$$\int_S f \,\mathrm{da} := \int_a^b \int_c^d f(x(u,v), y(u,v), z(u,v))\, k(u,v) \,\mathrm{d}v \,\mathrm{d}u \tag{21.15}$$

das **Overflächenintegral erster Art** von f über der Fläche S. Dabei ist

$$k(u,v) := \left| \begin{bmatrix} x_u(u,v) \\ y_u(u,v) \\ z_u(u,v) \end{bmatrix} \times \begin{bmatrix} x_v(u,v) \\ y_v(u,v) \\ z_v(u,v) \end{bmatrix} \right| \tag{21.16}$$

die Länge des Kreuzprodukts der Vektoren der partiellen Ableitungen von S nach u bzw. v.

Statt rechteckiger Parameterbereiche $[a,b] \times [c,d]$ sind auch Normalbereiche als Parameterbereiche möglich.

Aufgabe Ü3.8

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. Die Masse der Rinne kann als Oberflächenintegral

$$\int_S f \,\mathrm{da} \tag{21.17}$$

bestimmt werden, wobei f das (ortsabhängige) Flächengewicht des Blechs ist. Vereinfachen Sie dieses Integral zu einem gewöhnlichen Integral bzgl. einer Funktion über einem Intervall.

Oberflächenintegrale zweiter Art

Sei durch

$$S : [a, b] \times [c, d] \to \mathbb{R}^3, \qquad (u, v) \mapsto \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix} \tag{21.18}$$

eine Fläche im Raum gegeben und sei $f : \mathbb{R}^3 \to \mathbb{R}^3$ eine Funktion. Bezeichnen mit

$$\int_S f \,\mathrm{dn} := \int_a^b \int_c^d f(x(u, v), \ldots) \circ n(x(u, v), \ldots) \,\mathrm{d}v \,\mathrm{d}u \tag{21.19}$$

das **Oberflächenintegral zweiter Art** von f über der Fläche S. Dabei ist

$$n(x(u, v), y(u, v), z(u, v)) := \begin{bmatrix} x_u(u, v) \\ y_u(u, v) \\ z_u(u, v) \end{bmatrix} \times \begin{bmatrix} x_v(u, v) \\ y_v(u, v) \\ z_v(u, v) \end{bmatrix} \tag{21.20}$$

ein spezieller Normalenvektor an die Fläche S im Punkt $x(u, v), y(u, v), z(u, v)$.

Statt rechteckiger Parameterbereiche $[a, b] \times [c, d]$ sind auch Normalbereiche als Parameterbereiche möglich.

Aufgabe Ü3.9

Ein gebogenes Drahtgitter habe die Form wie in Aufgabe Ü3.8. Dieses wird in einer Wasserströmung platziert, welche durch das Vektorfeld

$$f(x, y, z) = \begin{bmatrix} x^2 + y4 \\ 3\,x\,y \\ 2\,x\,z + z^2 \end{bmatrix} \tag{21.21}$$

gegeben ist (Richtung und Geschwindigkeit). Berechnen Sie das pro Zeiteinheit durch das Gitter strömende Wasservolumen als Oberflächenintegral

$$\left| \int_S f \,\mathrm{dn} \right|. \tag{21.22}$$

22 Übung 4

22.1 Differentialoperatoren

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

$$(\operatorname{div} f)(x, y, z) := \frac{\partial}{\partial x} f_1(x, y, z) + \frac{\partial}{\partial y} f_2(x, y, z) + \frac{\partial}{\partial z} f_3(x, y, z) \tag{22.1}$$

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

$$(\operatorname{rot} f)(x, y, z) := \begin{bmatrix} \frac{\partial}{\partial y} f_3(x, y, z) - \frac{\partial}{\partial z} f_2(x, y, z) \\ \frac{\partial}{\partial z} f_1(x, y, z) - \frac{\partial}{\partial x} f_3(x, y, z) \\ \frac{\partial}{\partial x} f_2(x, y, z) - \frac{\partial}{\partial y} f_1(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \times \begin{bmatrix} f_1(x, y, z) \\ f_2(x, y, z) \\ f_3(x, y, z) \end{bmatrix} \tag{22.2}$$

(**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

$$(\Delta f)(x, y, z) := \frac{\partial^2}{\partial^2 x} f(x, y, z) + \frac{\partial^2}{\partial^2 y} f(x, y, z) + \frac{\partial^2}{\partial^2 z} f(x, y, z) \tag{22.3}$$

(**Laplace-Operator** angewendet auf f).

Aufgabe Ü4.1

Überzeugen Sie sich, dass $\Delta f = \operatorname{div} \nabla f$ gilt.

Aufgabe Ü4.2

Berechnen Sie die Rotation von

$$f(x, y, z) = (x^2 + y\,z + z,\ y + x\,z,\ x\,y\,z). \tag{22.4}$$

Wie bewegt sich ein (kleines und leichtes) Objekt, welches im Punkt $(1, 0, 0)$ in die Strömung gegeben wird (Bewegungsrichtung, Drehachse)?

22.2 Integralsatz von Gauß

Seien $B \subseteq \mathbb{R}^3$ und $f : \mathbb{R}^3 \to \mathbb{R}^3$ hinreichend regulär. Dann gilt

$$\int_B \operatorname{div} f \,\mathrm{db} = \int_{\partial B} f \,\mathrm{dn}, \tag{22.5}$$

wobei ∂B die Oberfläche von B bezeichne. Die Summe aller Quellen und Senken im Gebiet ist also gleich dem Fluss über den Rand.

Aufgabe Ü4.3

Überprüfen Sie den Integralsatz von Gauß für das Vektorfeld aus Aufgabe Ü4.2. Verwenden Sie als Gebiet B das Zwischen den folgenden vier Flächen liegende Volumen:

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

23 Übung 5

23.1 Aufgabe Ü5.1

Leiten Sie die Formeln für ein Finite-Differenzen-Verfahren zum Lösen der Wellengleichung in zwei Raumdimensionen mit Dirichlet-Randbedingungen auf einem Rechteckgebiet her:

$$c^2 \left(u_{xx}(x,y,t) + u_{yy}(x,y,t)\right) - u_{tt}(x,y,t) = 0, \quad (x,y) \in (0,a) \times (0,b),\, t > 0, \tag{23.1}$$

$$u(x,y,0) = f(x,y), \quad (x,y) \in [0,a] \times [0,b], \tag{23.2}$$

$$u_t(x,y,0) = 0, \quad (x,y) \in [0,a] \times [0,b], \tag{23.3}$$

$$u(x,y,t) = 0, \quad (x,y) \in \partial((0,a) \times (0,b)), \quad t \geq 0. \tag{23.4}$$

Orientieren Sie sich dabei am Vorgehen in Fallstudie: 1D-Wellengleichung.

23.2 Aufgabe Ü5.2

Das diskretisierte Problem ist gut konditioniert, wenn die Courant-Zahlen

$$C_x := \frac{c\,\Delta t}{\Delta x} \quad \text{und} \quad C_y := \frac{c\,\Delta t}{\Delta y} \tag{23.5}$$

die Bedingung

$$C_x^2 + C_y^2 \leq 1 \tag{23.6}$$

erfüllen. Die diskretisierten Lösungen konvergieren gegen die exakte Lösung, wenn Gleichheit gilt (völlig analog zum 1D-Fall).

Wie ist die Anzahl n_t der Unterteilungen des Zeitintervalls $[0,T]$ in Abhängigkeit von n_x und n_y (Unterteilungen x- und y-Richtung) zu wählen, damit die Ungleichung erfüllt ist und $C_x^2 + C_y^2$ möglichst nah bei 1 liegt?

23.3 Aufgabe Ü5.3

Wie viel Arbeitsspeicher wird für das Ablegen der Lösung für alle Zeitschritte in Abhängigkeit von n_t, n_x, n_y benötigt?

24 Übung 6

24.1 Verallgemeinerte Funktionen

Aufgabe Ü6.1

Sei $B = [0, 1]$. Für welche Exponenten a gehört die durch

$$f(x) = x^a \tag{24.1}$$

gegebene Funktion f zu $L^2(B)$? Hinweis: Für Potenzfunktionen nimmt das Lebesque-Integral genau dann einen endlichen Wert an, wenn das entsprechende uneigentliche Riemann-Integral existiert.

Aufgabe Ü6.2

Finden Sie die schwache Ableitung für die durch

$$f(x) = \begin{cases} x^2, & \text{für } x \leq 0, \\ x, & \text{für } x > 0 \end{cases} \tag{24.2}$$

auf dem Intervall $[-1, 1]$ gegebenene Funktion f.

24.2 FEM

Wir betrachten auf dem Gebiet $B = [0, 1]$ die Potential-Gleichung

$$u''(x) = \frac{1}{4} - \left(x - \frac{1}{2}\right)^2 \quad \text{für } x \in B \tag{24.3}$$

mit den Randbedingungen

$$u(0) = 0 \qquad \text{und} \qquad u'(1) = \frac{1}{4}. \tag{24.4}$$

Aufgabe Ü6.3

Finden Sie eine Lösung ohne Computer!

Aufgabe Ü6.4

Geben Sie eine schwache Formulierung dieses Problems an, die nur die erste schwache Ableitung von u benötigt!

Aufgabe Ü6.5

Zum Lösen des 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.

25 Download

Zum Offline-Lesen steht das Vorlesungsskript als PDF-Datei zur Verfügung. Aus naheliegenden Gründen sind Animationen dort nicht enthalten. Generall ist die Konvertierung nach PDF noch etwas experimentell, sodass Darstellungsfehler nicht auszuschließen sind.

Download PDF-Datei

Aufgabenblätter für Übung und Praktikum können Sie auch einzeln als PDF-Datei oder als Jupyter-Notebook auf der entsprechenden Seite runterladen (Button oben rechts).

26 Organisatorisches

Modulbeschreibung in Modulux

26.1 Ablauf der Veranstaltung

- ein Einheit Vorlesung pro Woche
- eine Einheit Übung/Praktikum pro Woche
- Vorlesung und Übung/Praktikum wie im Stundenplan (Ausnahmen bestätigen die Regel)
- Übung und Praktikum im Wesentlichen im Wechsel (je nach Stand der Vorlesung)
- **Vorbereitung:** Skript vor Vorlesung und Übung/Praktikum mindestens überfliegen!
- **Nachbereitung:** Alles verstanden? Rückstände aus Übung/Praktikum nachholen!

26.2 Praktika

- Praktika/Übungen im Computer-Pool (oder eigener Computer)
- benötigte Software: Webbrowser (je nach Vorlieben eine lokale Python/Jupyter-Installation)

26.3 Prüfung

- mündlich, 30 Minuten
- Prüfungsschwerpunkte
- Am Beginn der Prüfung werden zufällig drei Schwerpunktthemen ausgewählt. Ein Thema kann durch den Prüfling abgewählt werden. Die anderen beiden Themen werden in der Prüfung besprochen.

26.4 Hinweise zum Skript

- Download als PDF-Datei möglich
- Aufgabenzettel für Übung/Praktikum auch einzeln als PDF-Datei
- Aufgabenzettel für Übung/Praktikum 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 (nicht vom Vorlesenden) machen
- (optional) Fotos zum Verlinken im Skript abliefern (Upload im OPAL-Kurs)
- bitte keine Videos (entweder professionell oder gar nicht; ersteres aktuell nicht leistbar)

Die Technik hinter dem Skript ist noch in Entwicklung und teils etwas “beta”. Hinweise zu Problemen und Verbesserungsvorschläge gern an den Vorlesenden!

27 Prüfungsschwerpunkte

Die folgenden Schwerpunktthemen für die Prüfung dienen zur besseren Orientierung bei der Prüfungsvorbereitung. Grundsätzlich können aber alle Inhalte der Lehrveranstaltung eine Rolle in der Prüfung spielen.

27.1 1. Modellierung

Fehlerarten im wissenschaftlichen Rechnen, Arbeitsschritte vom praktischen Problem zur Lösung am Computer, Hadamard-Bedingungen, Beispiel für ein klassisches Modell, Diskussion der Begriffe und Abläufe an diesem Beispiel.

27.2 2. Numerische Differentiation und Integration

Verfahren für erste und zweite Ableitungen, Begriff der Fehlerordnung, Herleiten von Fehlerabschätzungen, Herleiten von Verfahren für höhere Ableitungen, Möglichkeiten der numerischen Integration (Ansatz und Beispiele für Basisfunktionen)

27.3 3. Computertomografie

Problemstellung, Schritte beim Diskretisieren (direkte Diskretisierung der Radon-Transformation), Probleme beim Lösen des Gleichungssystems und Regularisierung.

27.4 4. PDE

Beispiel einer PDE zweiter Ordnung für ein konkretes Gebiet einschließlich Randbedingungen (als Formeln aufschreiben), Klassifikation von PDE zweiter Ordnung (elliptisch, hyperbolisch, parabolisch).

27.5 5. FDM

Idee von FDM, Vorteile, Nachteile, grober Ablauf an einem Beispiel, Umgang mit Randbedingungen.

27.6 6. Schwache Ableitungen

Begriff der schwachen Ableitung, Funktionenräume $L^2(B)$ und $H^1(B)$, Beispiele dazu.

27.7 7. Schwache Formulierung

Idee/Ziel der schwachen Formulierung einer PDE, Umgang mit Randbedingungen, abstrakte Form schwacher Formulierungen.

27.8 8. FEM

Idee, Diskretisierung der schwachen Formulieren, weitere Arbeitsschritte, Vorteile, Nachteile.

27.9 9. FEM am Beispiel

Formulieren eines stationären Wärmeleitproblems einschließlich Dirichlet- und Neumann-Randbedingungen, geeignete Gebietszerlegung, Schritte zum Aufbau von Systemmatrix und rechter Seite des zu lösenden Gleichungssystems.

Literaturverzeichnis

[BF14] Steven Bürger and Jens Flemming. Deautoconvolution: A new decomposition approach versus TIGRA and local regularization. *Journal of Inverse and Ill-posed Problems*, 23(3):231–243, 4 2014.

[SL74] L. A. Shepp and B. F. Logan. The Fourier reconstruction of a head section. *IEEE Transactions on Nuclear Science*, 21(3):21–43, 6 1974.